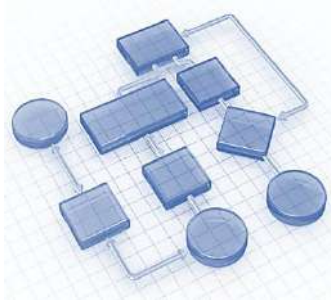


მ. ფხოველიშვილი, ლ. ტოკაძე,
ნ. ბერაია, ი. აფციაური



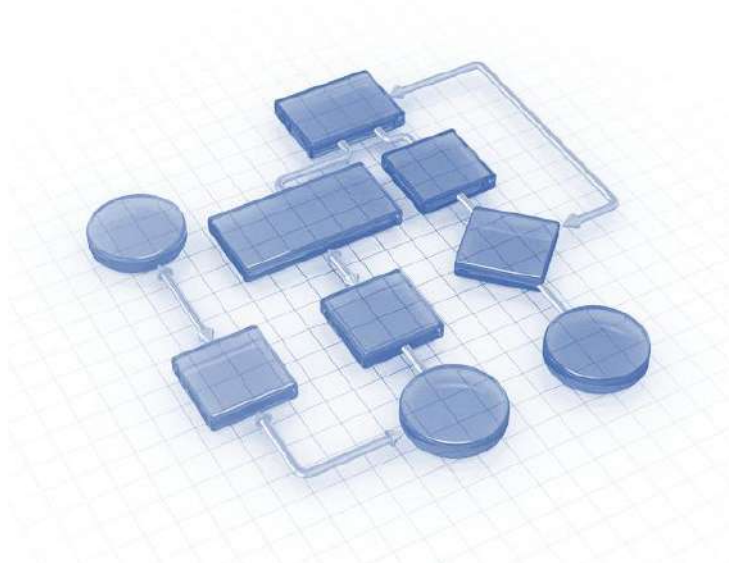
მონაცემთა სტრუქტურები და ალგორითმები



საგამომცემლო სახლი
„ტექნიკური უნივერსიტეტი“

მერაბ ფხოველიშვილი, ლალი ტოკაძე,
ნინო ბერაია, ია აფციაური

მონაცემთა სტრუქტურები და ალგორითმები



SCAN ME

საგამომცემლო სახლი
„ტექნიკური უნივერსიტეტი“

საქართველოს ტექნიკური უნივერსიტეტი

მერაბ ფხოველიშვილი, ლალი ტოკაძე,
ნინო ბერაია, ია აფციაური

მონაცემთა სტრუქტურები და ალგორითმები



დამტკიცებულია სახელმძღვანელოდ

საქართველოს ტექნიკური
უნივერსიტეტის

სარედაქციო-საგამომცემლო საბჭოს

მიერ. 30.06.2025, ოქმი №2

თბილისი
2025

ნაშრომი კომპლექსური სახელმძღვანელოა სტუდენტებისთვის და გადმოგვცემს დისციპლინის „მონაცემთა სტრუქტურები და ალგორითმები“ - საწყის, ძირითად ცნებებსა და პრინციპებს, რაც სტუდენტებს საშუალებას მისცემს, რომ მომავალში აირჩიონ უფრო ეფექტური და ოპტიმალური მიდგომები, რთული ამოცანების გადაჭრისთვის.

სახელმძღვანელოში განხილულია ალგორითმის ცნება და მისი როლი მარტივი ამოცანების გადასაწყვეტად. მოიცავს სხვადასხვა ალგორითმის აღწერას, მათ შორის იტერაციული და რეკურსიული ალგორითმების, ასევე მარტივი და რთული მონაცემთა სტრუქტურების, როგორცაა მასივები, სტეკები, რიგები, დეკები, დაკავშირებული სიები, გრაფები, პრეფიქსების ხეები და ჰეშ-ცხრილები.

ასევე გაშუქებულია კომპიუტერული ალგორითმების შეფასებასა და აღწერასთან დაკავშირებული საკითხები, ალგორითმების მუშაობის დრო, ციკლების ვარიანტები და რეკურსიები. დიდი ყურადღება ეთმობა დახარისხებისა და ძიების ალგორითმებს, მათ შორის ბინარულ ძიებას, შერჩევით, ჩასმით, შერწყმით და სწრაფ დახარისხებას; ასევე ტოპოლოგიური დახარისხების და ორიენტირებულ გრაფებში უმოკლესი გზის პოვნის საკითხები. განსაზღვრულია ალგორითმები სტრიქონებთან მიმართებაში (ან ალგორითმები სტრიქონებზე), P და NP კლასების ამოცანები, მატრიცები და მათთან დაკავშირებული მოქმედებები, კრიპტოგრაფიის საფუძვლები, მონაცემთა შეკუმშვა და სხვა.

განკუთვნილია ინფორმატიკისა და მართვის სისტემების ფაკულტეტის სტუდენტებისთვის, ასევე მკითხველთათვის, რომლებიც დაინტერესებული არიან ინფორმატიკითა და დაპროგრამებით.

რეცენზენტები: საქართველოს ტექნიკური უნივერსიტეტის ინფორმატიკისა და მართვის სისტემების ფაკულტეტის პროფესორი გია ურგულაძე,

საქართველოს ტექნიკური უნივერსიტეტის ინფორმატიკისა და მართვის სისტემების ფაკულტეტის ასოცირებული პროფესორი გულნარა ჯანელიძე

© საგამომცემლო სახლი „ტექნიკური უნივერსიტეტი“, 2025

ISBN 978-9941-520-85-3

<http://www.gtu.ge>



ყველა უფლება დაცულია. ამ წიგნის არც ერთი ნაწილის (იქნება ეს ტექსტი, ფოტო, ილუსტრაცია თუ სხვა) გამოყენება არანაირი ფორმით და საშუალებით (იქნება ეს ელექტრონული თუ მექანიკური) არ შეიძლება გამომცემლის წერილობითი ნებართვის გარეშე.

საავტორო უფლებების დარღვევა ისჯება კანონით.

წიგნში მოყვანილი ფაქტების სიზუსტეზე პასუხისმგებელია ავტორი/ავტორები.

ავტორის/ავტორთა პოზიციას შეიძლება არ ემთხვეოდეს საგამომცემლო სახლის პოზიციას.

სარჩევი

თემა N1 შესავალი. მონაცემთა სტრუქტურები. მონაცემთა მარტივი სტრუქტურები: მასივი, სტეკი, რიგი, დეკი. მონაცემთა რთული სტრუქტურები: ბმული სიები, გრაფები, პრეფიქსული ხეები, ჰეშცხრილები.....	6
თემა N2 ალგორითმის ცნება. ამოცანის ჩამოყალიბება, ალგორითმის ცნება, მარტივ მაგალითებზე ალგორითმის აღწერა, რეკურსიული და იტერაციული ალგორითმების მაგალითები. ფორმალური მტკიცების მეთოდები. მათემატიკური ინდუქციის, დიაგნალიზაციისა და დირიხლეს პრინციპი, მათი გამოყენება თეორემათა მტკიცების დროს.	27
თემა N3 ალგორითმების შეფასება. ალგორითმის ბიჯების რაოდენობის დათვლა, სისწორის მტკიცების მეთოდები. გამოთვლების სისწრაფე. ფუნქციათა ზრდის სისწრაფე და მისი გამოყენება შეფასებისას. ალგორითმის მუშაობის მინიმალური, მაქსიმალური და საშუალო დრო. ალგორითმის ბიჯების რაოდენობის ზედა და ქვედა ზღვრის შეფასების მეთოდები.	46
თემა N4 სორტირების და ძებნის ალგორითმები. ბინარული ძებნა, სორტირება არჩევით, სორტირება ჩასმით, სორტირება შერწყმით, ჩქარი სორტირება, სორტირების დროის ქვედა საზღვარი და მისი დამღევა.....	62
თემა N5 რეკურსიული და იტერაციული მეთოდების შედარება ალგორითმების მაგალითზე, მათი ანალიზი და დროის სირთულის შეფასება. ფიბონაჩის მიმდევრობა და მისი მნიშვნელობა მეცნიერებასა და პრაქტიკაში, მისი გამოთვლის რეკურსიული ალგორითმი. ალგორითმის გაუმჯობესების მეთოდები, ფიბონაჩის მიმდევრობის გამოთვლის არარეკურსიული ფორმულა. ფიბონაჩის კოდის პრაქტიკაში გამოყენების მაგალითები.....	78
თემა N6 მონაცემთა ფორმალიზაცია. ამოცანების მონაცემთა ფორმალური ჩაწერა, მონაცემთა კოდირება და მათი დამუშავება, რიცხვების ჩაწერის სხვადასხვა სისტემა, ანბანი, ენა და გრამატიკა. მწელად ამოსახსნელი ამოცანები. პოლინომიალური და ექსპონენციალური ალგორითმები, მაგალითები.....	102
თემა N7 ორიენტირებული აციკლური გრაფები. ტოპოლოგიური სორტირება, ორიენტირებული გრაფების წარმოდგენა ტოპოლოგიური სორტირების მუშაობის დროს. PERT დიაგრამაში კრიტიკული გზა. უმოკლესი გზა ორიენტირებულ აციკლურ გრაფში.	129
თემა N8 გრაფები. ხე, ტყე და მათი გამოყენება. ძირითადი ამოცანები ხეებზე, მათი გადაჭრის ალგორითმები. რამდენიმე რეალური ამოცანის სიმულაცია გრაფებით. ეილერის და ჰამილტონის ციკლების პოვნა გრაფებში, ალგორითმები. უმოკლესი გზები. დეიქსტრას ალგორითმები, ბელმან-ფორდის ალგორითმი, ფლოიდ უორშელის ალგორითმი. კომივოიაჟერის ამოცანა, წრფივი გადაადგილებების ამოცანა.	158
თემა N9 ალგორითმები სტრიქონებზე. უგრძელესი საერთო ქვემიმდევრობა, ერთი სტრიქონის სხვა სტრიქონად გარდაქმნა. ქვესტრიქონების ძებნა. მატრიცები და მათზე მოქმედებები. მატრიცების მახასიათებლები, მატრიცების გადამრავლების შტრასენის ალგორითმი, მატრიცების შებრუნება.	195

თემა N10 კომბინატორიკის ელემენტები. შესაძლო კომბინაციების სრული რაოდენობის დათვლის მეთოდები, ხელსაყრელი კომბინაციების დათვლა, ალბათობის გამოთვლის ელემენტები. კომბინატორული ამოცანები. პოლინომიალური ამოცანების ამოხსნის ალგორითმები, ეფექტური ამოხსნები. ექსპონენციალური ამოცანების ამოხსნის ალგორითმები, პრობლემები.....	219
თემა N11 კრიპტოგრაფიის საფუძვლები. მარტივი ჩასმადი შიფრი, კრიპტოგრაფია ღია გასაღებით. RSA კრიპტოსისტემების შინაარსი, ჰიბრიდული კრიპტოსისტემები, შემთხვევითი რიცხვების გამოთვლა.....	243
თემა N12 მონაცემების შეკუმშვა. ხოფმანის (ჰაფმანის) კოდები. ფაქსიმილური აპატები.....	263
თემა N13 რთული P და NP და NP-სრული კლასები. NP-სრული ამოცანების კრებული. საერთო სტრატეგიები, გადაუჭრელი ამოცანები.....	278
თემა N14 დიდი მონაცემები. სტრუქტურები, შენახვის, წარმოდგენისა და დამუშავების პრობლემები. დიდ მონაცემთა სისტემები.....	296
თემა N15 პარალელური მონაცემები და პარალელური გამოთვლების ალგორითმები. ძირითადი განმარტებები, გადასვლები მაჩვენებლებით, დანახარჯების ეფექტურობა, პრეფიქსების ეფექტური პარალელური დამუშავება, სიმეტრიული დარღვევა, პროგნოზირების ამოცანები, პროგნოზირების მოდელების შეჯერების მეთოდი.	329

შესავალი. მონაცემთა სტრუქტურები. მონაცემთა მარტივი სტრუქტურები: მასივი, სტეკი, რიგი, დეკი. მონაცემთა რთული სტრუქტურები: ბმული სიები, გრაფები, პრეფიქსული ხეები, ჰეშცხრილები

შესავალი

მონაცემი არის გარკვეული ხასიათის მიზნებისთვის შეგროვებული და გარდაქმნილი ინფორმაცია. მონაცემი შეიძლება იყოს ნებისმიერი სიმბოლო, ტექსტი, ციფრი, რიცხვი, სურათი, ხმა ან ვიდეო, რომელზეც შეიძლება შესრულდეს შეგროვების, გარდაქმნის, გადამცემის, შენახვის, დამუშავების, ჩვენების, შეკუმშვის, დაცვის, ვიზუალიზაციის და ა. შ. ოპერაციები.

მეორენაირად მონაცემები შეგვიძლია ასეც განვსაზღვროთ: მონაცემი არის ერთი მნიშვნელობა ან მნიშვნელობების ნაკრები, რომელიც აღწერს ნებისმიერ ინფორმაციას. ეს ინფორმაცია კი შესაძლებელია დავამუშაოთ კომპიუტერის დახმარებით და მივიღოთ მოთხოვნის შესაბამისი შედეგი.

მონაცემად შეიძლება ჩაითვალოს:

ა) ნებისმიერი მარტივი რიცხვი ან სიმბოლო;
ბ) რთული ელემენტი, მაგალითად, ადამიანის პერსონალური მონაცემი; ინფორმაცია მოსახლეობის როდენობაზე;

გ) ელემენტებს შორის ურთიერთდამოკიდებულება და კავშირი;

მონაცემების კარგად ჩამოყალიბებულ, დალაგებულ, გამართულ და მოწესრიგებულ (წესრიგიან) მოდელს შეგვიძლია ვუწოდოთ მონაცემთა სტრუქტურა.

კომპიუტერულ მეცნიერებებში მონაცემთა სტრუქტურა განიხილება როგორც, ინფორმაციის (მონაცემების) შენახვისა და წარმოდგენის ფორმა. მონაცემთა სტრუქტურის ასეთი განმარტება საკმაოდ ზოგად ხასიათს ატარებს.

თუ უფრო დავაკონკრეტებთ, მაშინ შეგვიძლია ვთქვათ, რომ მონაცემთა სტრუქტურა არის: ა) მათემატიკური ობიექტების კლასი; ბ) მონაცემთა მოწესრიგებისა და მათზე წვდომის მკაფიოდ განსაზღვრული და სისტემატიზებული მეთოდი.

მონაცემთა სტრუქტურა შეიძლება იყოს მარტივიც და რთულიც.

მარტივი მონაცემთა ტიპებია: მთელი, მცოცავ წერტილიანი (ნამდვილი, წილადი), ლოგიკური (ბულის), სტრიქონული, სიმბოლური და ა.შ. ტიპები.

უფრო ზუსტად კი მონაცემთა სტრუქტურა კლასიფიცირდება შემდეგნაირად: მარტივი, სტატიკური, ნახევრადსტატკიური, დინამიკური და ფაილური (ცხრილი 1.1.).

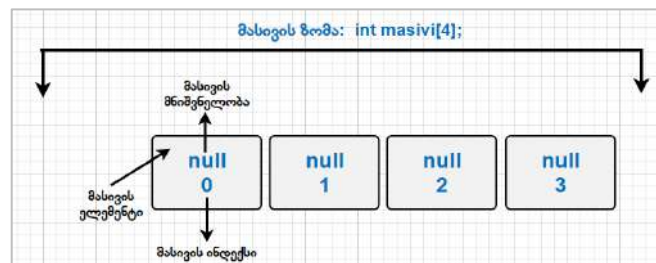
ცხრილი 1.1. მონაცემთა სტრუქტურის კლასიფიცირება

მარტივი	სტატიკური	ნახევრადსტატიკური	დინამიკური	ფაილური
რიცხვი	ვექტორი	სტეკი	ბმული სია	მიმდევრობითი
სიმბოლო	მასივი	დეკი	გრაფი	პირდაპირი წვდომა
ლოგიკური	სიმრავლე	რიგი	გრაფი-ხე	კომბინირებული
ჩამოთვლა	ჩანაწერი	სტრიქონი		წვდომა
მიმთითებელი	ცხრილი			

მონაცემთა სტრუქტურა მასივი

სხვადასხვა ამოცანების გადასაწყვეტად ხშირად გვჭირდება შევინახოთ არა ერთი ცალი მნიშვნელობა, არამედ ერთსა და იმავე ტიპის მნიშვნელობების გარკვეული რაოდენობა ანუ მონაცემების ნაკრები/ერთობლიობა. ერთსა და იმავე ტიპის მონაცემთა ერთობლიობის შესანახად გამოიყენება მონაცემთა სტრუქტურა - მასივი.

მასივი არის თანმიმდევრობით განლაგებული მეხსიერების უჯრედების გარკვეული რაოდენობა, რომლებშიაც ჩაწერილია ერთი და იმავე ტიპის მონაცემები. მონაცემთა მასივის ერთ უჯრედს მასივის ელემენტი ეწოდება. მასივში შეგვიძლია შევინახოთ მაგალითად, თანამშრომელთა ხელფასები, სტუდენტთა შეფასების ქულები და ა. შ. ამდენად თავისუფლად შეგვიძლია ვთქვათ, რომ მასივს აქვს ზომა ანუ ელემენტების რაოდენობა. მასივის თითოეულ ელემენტს კი გააჩნია საკუთარი ნომერი, იგივე ინდექსი. მასივის ელემენტთან მიმართება ხორციელდება ინდექსის საშუალებით (ნახ. 1.1).



ნახ. 1.1. მასივის აღწერა

ნახატ 1.1-ზე ნაჩვენებია მასივი, რომელიც შედგება მთელი ტიპის ოთხი ელემენტი-საგან. როგორც ვხედავთ, მასივის უჯრედებში არ არის ჩაწერილი რაიმე მონაცემი/მნიშვნელობა თუ ინფორმაცია. მიუხედავად ამისა, მასივი მაინც დაიკავებს მეხსიერების 16 ბაიტს (იხ. ნახ. 1.1.). მასივები საკმაოდ მოხერხებული მონაცემთა სტრუქტურაა: მასივის უჯრედების ერთმანეთის თანამიმდევრობით, მიყოლებით განლაგება იძლევა იმის საშუალებას, რომ გავზარდოთ კოდის მუშაობის სისწრაფე (ნახ. 1.2.).

5	-12	-12	9	10	0	-9	-11	-12	21	7	13	21	17	16	3	-8
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]

ნახ. 1.2. მთელი ტიპის ერთგანზომილებიანი მასივი

მნიშვნელოვანია, ვიცოდეთ:

1) მასივი არის ფიქსირებული ზომის მქონე მონაცემთა სტრუქტურა, რადგან წინასწარ არის ცნობილი მასივის ზომა (ინდექსების რაოდენობა). ჩვენ შეგვიძლია მასივის რომელიმე ცალკეული ელემენტის (მონაცემის) შეცვლა, ჩასწორება, მაგრამ არა თვითონ მთლიანი მასივისა.

2) არ შეიძლება მივმართოთ მასივის იმ უჯრედს, რომელიც მასივის ზომის გარეთაა ანუ გასცილდა მასივის ზომას. ვთქვათ, მასივის ზომა არის 5, ჩვენ კი ვცდილობთ მივმართოთ მასივის მე-5, მე-6 და შემდგომ ელემენტებს, ასეთ შემთხვევაში პასუხი იქნება გაურკვეველი. პასუხს კი მივიღებთ, მაგრამ პასუხი იქნება არასწორი, რაც მიგვიყვანს არაკორექტულ და მცდარ შედეგთან.

მონაცემთა სტრუქტურა. სტეკი

სტეკი (stack) ინახავს მონაცემებს თანამიმდევრული ფორმით. სტეკში მონაცემები ერთმანეთთან დაკავშირებულია შემდეგი სახით: თითოეული ელემენტი მიუთითებს მომდევნო ელემენტზე. სტეკი ხასიათდება წრფივი კავშირით - ელემენტები იმყოფება ერთმანეთის გვერდი გვერდ და ელემენტის აღება ხდება რიგის მიხედვით. ელემენტის აღება სტეკის შუა ადგილიდან არ შეიძლება.

სტეკი შეიძლება წარმოვიდგინოთ როგორც ერთმანეთზე დალაგებული თეფშების ან წიგნების ან რვეულების თანამიმდევრობა (ნახ. 1.3.).



ნახ. 1.3. სტეკის მაგალითი

რა შეგვიძლია გავაკეთოთ ერთმანეთზე დალაგებულ წიგნებთან მიმართებით:

ა) ერთიმეორეზე დალაგებული წიგნების თავზე დავამატოთ კიდევ ერთი წიგნი;

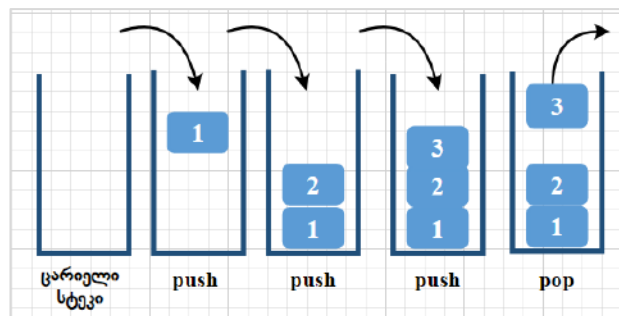
ბ) ავიღოთ ერთიმეორეზე დალაგებული წიგნების თავში მყოფი წიგნი (ზედა წიგნი);

თუ ჩვენ გვინდა ავიღოთ ერთიმეორეზე დალაგებულ წიგნებიდან სულ ბოლო წიგნი, მაშინ მოგვიწევს ყველა იმ წიგნის აღება, რომელებიც იმყოფებიან მის ზევით. რომელი წიგნიც დაემატა ბოლოს, სწორედ მისი აღება მოხდება პირველად (შუა ადგილიდან წიგნის გამოტანა ან წიგნების გადაყრა-გადმოყრა[©] არ განიხილება, რადგან საქმე გვაქვს მოწესრიგებულ, თანამიმდევრულ სტრუქტურასთან).

სტეკებთან მიმართებით არსებობს შემდეგი ძირითადი ოპერაციები: ელემენტის დამატება, ამოღება, შემოწმება (ცარიელია თუ არა). ეს ოპერაციები აღწერილია ცხრილში 1.2.

მეთოდი	სტეკის მეთოდების აღწერა
Top()	სტეკის ზედა ელემენტის დაბრუნება. სტეკიდან ამოღების გარეშე აბრუნებს ზედა ელემენტს.
Push()	ელემენტის მოთავსება სტეკის ზედაპირზე ანუ ელემენტის ჩატვირთვა. ელემენტის ზემოდან დამატება.
Pop()	ელემენტის მოხსნა სტეკის ზედაპირიდან ანუ ამოტვირთვა. სტეკიდან ელემენტის ამოღება (წაშლა).
Size()	სტეკში ელემენტების რაოდენობის დაბრუნება. სტეკის ზომა.
IsEmpty()	ამოწმებს ცარიელია თუ არა სტეკი. თუ სტეკი ცარიელია აბრუნებს True-ს, საწინააღმდეგო შემთხვევაში აბრუნებს False-ს.
IsFull()	ამოწმებს შევსებულია თუ არა სტეკი.
Peek()	ბოლო ელემენტის დაბრუნება ისე, რომ არ ხდება მისი წაშლა.

სტეკის მოქმედების პრინციპია **LIFO** (last-in, first-out) - ბოლოს მოვიდა, პირველი წავიდა. ნახატ 1.4-ზე ნაჩვენებია ორი ძირითადი ოპერაციის push და pop განხორციელების ნიმუში.



ნახ. 1.4. სტეკის ძირითადი ოპერაციები

ნახ. 1.4-დან ვხედავთ, რომ სტეკში მე-3 ელემენტი დაემატა ბოლოს, მაგრამ მისი წაშლა მოხდება სტეკის დასაწყისიდან ანუ თავიდან.

დაწვრილებით განვიხილოთ სტეკის მუშაობის ალგორითმი (ნახ. 1.5.):

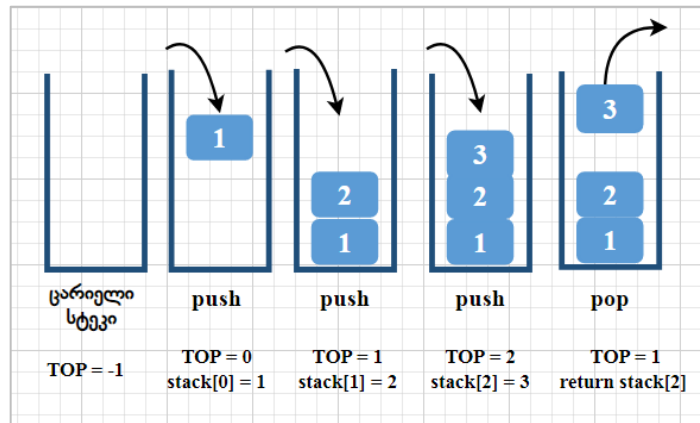
ა) სტეკში **TOP()** მეთოდი გამოიყენება ზედა ელემენტის „თვალთვალისთვის“;

ბ) სტეკის ინიციალიზაციის შემდეგ **TOP()** მეთოდის მნიშვნელობა ხდება -1. ამდენად შეგვიძლია ძალიან მარტივად გავარკვეოთ შევსებულია თუ არა სტეკი: თუ $TOP() = -1$, მაშინ სტეკი არის ცარიელი;

გ) როცა ჩვენ შევეცდებით ელემენტი „ავამუშაოთ, დავძრათ“, მაშინ **TOP()**-ის მნიშვნელობა იზრდება ერთით. მომდევნო ელემენტი გვექნება $TOP() + 1$ პოზიციაზე;

დ) თუ ჩვენ გამოვიყენებთ მეთოდს **POP()**, მაშინ მოხდება იმ ელემენტის წაშლა, რომელზეც მიუთითებს **TOP()** მეთოდი. ამის შემდეგ მისი მნიშვნელობა მცირდება ერთით;

- ე) სტეკში ელემენტის დამატებისას, უნდა შევამოწმოთ შევსებულია თუ არა სტეკი;
- ვ) სტეკიდან ელემენტის ამოშლისას, კი ვამოწმებთ ცარიელია თუ არა იგი.



ნახ. 1.5. სტეკის მუშაობის პრინციპი

პრინციპი **LIFO** განვიხილოთ მასივის მაგალითზე (ცხრილი 1.3., ნახ. 1.6. ა,ბ,გ,დ).

ცხრილი 1.3. სტეკის რეალიზება მასივის მაგალითზე

სტეკის თავდაპირველი მდგომარეობა	სტეკში მოვათავსოთ ორი რიცხვი: პირველი რიცხვია: -3. top ტოლია 1-ის.	სტეკში მოვათავსოთ მეორე რიცხვი: 15. top ტოლია 2-ის.	სტეკში მოვათავსოთ კიდევ ორი რიცხვი: თავიდან 7, შემდეგ 4; სტეკში ოთხი რიცხვია; top ტოლია 4-ის.

top = 0 <table><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr></table>	0	0	0	0	0	0	0	0	top = 1 <table><tr><td>-3</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr></table>	-3	0	0	0	0	0	0	0	top = 2 <table><tr><td>15</td></tr><tr><td>-3</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr></table>	15	-3	0	0	0	0	0	0	top = 4 <table><tr><td>4</td></tr><tr><td>7</td></tr><tr><td>15</td></tr><tr><td>-3</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr></table>	4	7	15	-3	0	0	0	0
0																																			
0																																			
0																																			
0																																			
0																																			
0																																			
0																																			
0																																			
-3																																			
0																																			
0																																			
0																																			
0																																			
0																																			
0																																			
0																																			
15																																			
-3																																			
0																																			
0																																			
0																																			
0																																			
0																																			
0																																			
4																																			
7																																			
15																																			
-3																																			
0																																			
0																																			
0																																			
0																																			

ა)

მონაცემთა სტრუქტურა. რიგი

სტეკის მსგავსად, რიგიც ელემენტებს ინახავს ერთმანეთის მიმდევრობით, გვერდი-გვერდ. რიგი სტეკისაგან განსხვავდება მოქმედების პრინციპით FIFO (first-in, first-out). რიგის მოქმედების პრინციპი FIFO ნიშნავს - პირველი მოვიდა, პირველი წავიდა.

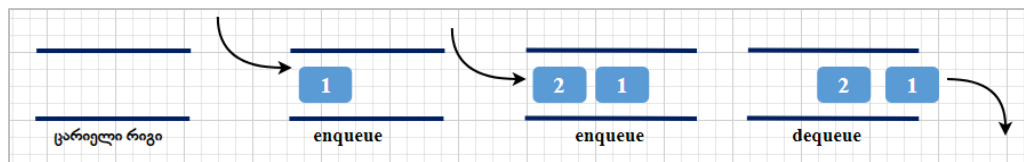
რიგის ერთ-ერთი მაგალითია რომელიმე სავაჭრო ობიექტში წარმოქმნილი ადამიანთა რიგი. ის, ვინც ბოლოს მოვიდა, რიგში დაიკავებს ბოლო ადგილს, ხოლო რიგში მდგარი პირველი ადამიანი კი წავა პირველი. რიგისთვის დამატებული ელემენტი აღმოჩნდება

რიგის ბოლოში, ხოლო წასაშლელი ელემენტი იმყოფება რიგის თავში. რიგის მეთოდები მოცემულია ცხრილში 1.4.

ცხრილი 1.4. რიგის მეთოდები

მეთოდი	რიგის მეთოდების აღწერა
Enqueue()	ელემენტის ჩასმა რიგის ბოლოს.
Dequeue()	ელემენტის წაშლა რიგის თავიდან.
IsEmpty()	ამოწმებს ცარიელია თუ არა რიგი. თუ რიგი ცარიელია აბრუნებს True-ს, საწინააღმდეგო შემთხვევაში აბრუნებს False-ს.
IsFull()	ამოწმებს შევსებულია თუ არა რიგი.
Peek()	რიგის პირველი ელემენტის დაბრუნება ისე, რომ არ ხდება მისი წაშლა.

განვიხილოთ რიგის ორი ძირითადი ოპერაცია enqueue და dequeue (ნახ. 1.7.).



ნახ. 1.7. რიგის ძირითადი ოპერაციები

ახლა კი განვიხილოთ რიგის მუშაობის პრინციპი (იხ. ცხრილი 1.5. და ნახ. 1.8.):

- ა) რიგის მუშაობა ეყრდნობა ორ მეთოდს: FRONT() და REAR();
 - ბ) FRONT() არის მიმთითებელი რიგის პირველ ელემენტზე;
 - გ) REAR() არის მიმთითებელი რიგის მეორე ელემენტზე;
 - დ) FRONT()-ის და REAR()-ის მნიშვნელობები თავიდან იღებენ მნიშვნელობას: -1;
- ახლა კი რიგის რეალიზება განვიხილოთ $Q[1..n]$ მასივის მაგალითზე (იხ. ნახ. 1.9.).

რიგს ყოველთვის აქვს დასაწყისი ანუ თავი (head[Q]). რიგის დასაწყისის ანუ მთავარი ელემენტის ინდექსი არის head[Q]. რიგს ასევე აქვს მეორე ატრიბუტი tail[Q]. tail[Q] კი არის იმ ადგილმდებარეობის (უჯრედის) ინდექსი, სადაც მოხდება ახალი ელემენტის დამატება. რიგში დამატებული ელემენტი აღმოჩნდება მის ბოლოში, ხოლო წასაშლელი ელემენტი იმყოფება რიგის თავში.

რიგის ელემენტები განლაგებული არიან უჯრედებში შემდეგი თანამიმდევრობით: head[Q], head[Q]+1, . . . , tail[Q]-1.

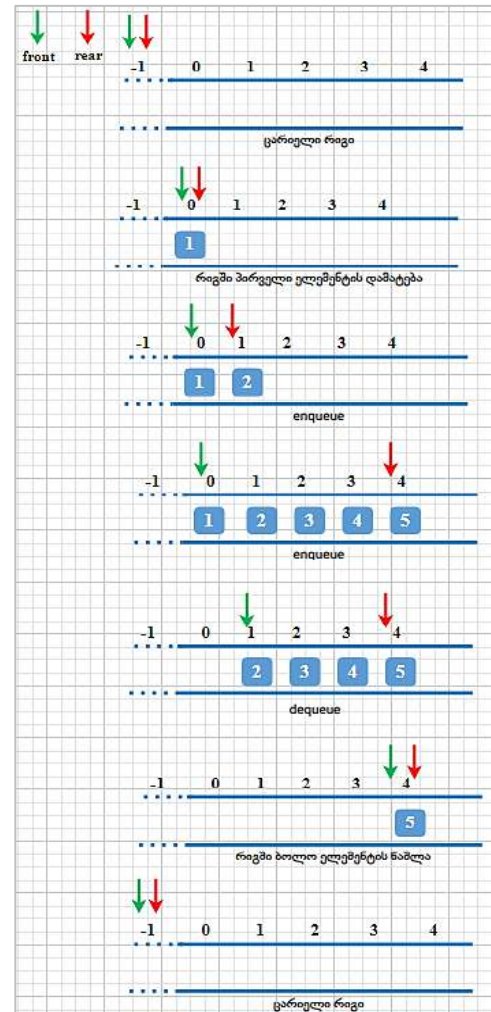
თუ სრულდება პირობის head[Q] = tail[Q], მაშინ რიგი ცარიელია. თავდაპირველად ყოველთვის სრულდება head[Q] = tail[Q] = 1 პირობა.

ცარიელი რიგიდან ელემენტის წაშლის მცდელობა იწვევს რიგის დაცლის (underflow) შეცდომას.

თუ head[Q] = tail[Q]+1 პირობა ჭეშმარიტია, მაშინ რიგი მთლიანადაა შევსებული და მისთვის ელემენტის დამატების მცდელობა იწვევს გადავსების (overflow) შეცდომას.

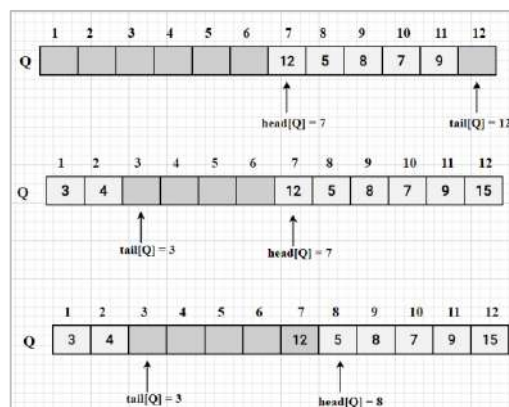
ცხრილი 1.5. რიგის მუშაობის პრინციპი

ოპერაცია enqueue-ს მუშაობის პრინციპი: ა) შევამოწმოთ არის თუ არა რიგი სავსე; ბ) რიგში პირველივე ელემენტის დამატებისას, მიმთითებელ FRONT-ს უნდა მივანიჭოთ 0-ის ტოლი მნიშვნელობა; გ) მიმთითებელი REAR-ის მნიშვნელობა უნდა გავზარდოთ ერთით; დ) რიგში ახალი ელემენტი შეგვიძლია დავამატოთ იმ პოზიციაზე, რომელზეც მიუთითებს მიმთითებელი REAR;	
ოპერაცია dequeue-ს მუშაობის პრინციპი: ა) შევამოწმოთ არის თუ არა რიგი ცარიელი; ბ) დავიმახსოვროთ ის მნიშვნელობა, რომელზეც მიუთითებს მიმთითებელი FRONT; გ) მიმთითებელი FRONT-ის მნიშვნელობა გავზარდოთ ერთით; დ) რიგიდან ბოლო ელემენტის წაშლისას მიმთითებლებს FRONT-ს და REAR-ს უნდა მივანიჭოთ მნიშვნელობა: -1;	



1.8. რიგის მუშაობის სქემა

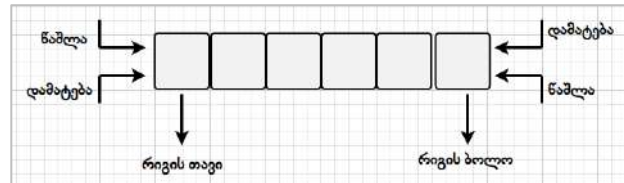
განვიხილოთ რიგის მუშაობა მასივის მაგალითზე. ნახ. 1.9-ზე აღწერილია რიგი, რომელიც შედგება ხუთი ელემენტისაგან. ეს ელემენტები განთავსებულია $Q[7..11]$ უჯრებში. შემდეგ კი ვხედავთ enqueue(Q,15), enqueue(Q,3) და enqueue(Q,4) პროცედურების შესრულების შედეგს, ხოლო dequeue(Q) პროცედურა კი აბრუნებს მე-12 გასაღების მნიშვნელობას, რომელიც enqueue პროცედურების შესრულებამდე იმყოფებოდა რიგის თავში. ახლა რიგს აქვს ახალი თავი და მისი მნიშვნელობა უკვე არის 5.



ნახ. 1.9. რიგის რეალიზება მასივის მაგალითზე. enqueue და dequeue პროცედურების მუშაობა.

მონაცემთა სტრუქტურა. დეკი

დეკი (deque, double-ended queue) არის ისეთი უნივერსალური მონაცემთა სტრუქტურა, სადაც ელემენტთა თანმიმდევრობას აქვს ორი კომპონენტი: თავი და ბოლო. დეკში ელემენტთა დამატება და წაშლა შესაძლებელია თანამიმდევრობის თავსა და ბოლოშიც (ნახ. 1.10.).



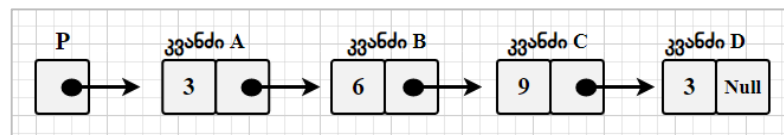
ნახ. 1.10. დეკი

დეკის სტრუქტურას აქვს შემდეგი მახასიათებლები: 1. ელემენტთა დამატება შესაძლებელია თანამიმდევრობის როგორც თავში, ისე ბოლოში; 2. ელემენტთა წაშლა შესაძლებელია თანამიმდევრობის როგორც თავში, ისე ბოლოში.

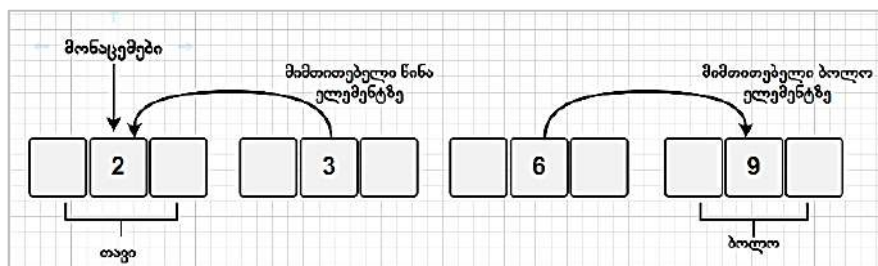
დეკი წარმოადგენს ე. წ. ორმხრივ რიგს.

მონაცემთა სტრუქტურა. ბმული სია

ბმული სია (linked list) არის ისეთი მონაცემთა სტრუქტურა, სადაც თითოეული ელემენტი წარმოადგენილია ცალკე ობიექტის/კვანძის სახით და ეს კვანძები განლაგებულია წრფივი თანამიმდევრობით. თუ მასივში ელემენტების თანამიმდევრობა განსაზღვრულია ინდექსით, ბმულ სიაში ეს თანამიმდევრობა განისაზღვრება თითოეულ ობიექტზე მიმთითებლით. ბმულ სიაში კვანძი აღიწერება როგორც: 1) ცვლადი, რომელიც ინახავს მიმდინარე კვანძის მონაცემს და 2) მომდევნო კვანძის მეხსიერების მისამართს (მიმთითებული მომდევნო ან წინა მონაცემზე) ე. ი. კვანძი ინახავს ინფორმაციას როგორც მასში არსებული მონაცემის, ისე მეზობელი კვანძის მონაცემის შესახებ. სწორედ ამ თვისების გამო ბმულ სიაში ყველა ელემენტი დაკავშირებულია ერთმანეთთან (ნახ. 1.11.ა,ბ)



ა)



ბ)

ნახ. 1.11. ბმული სიის: ა) ერთ-ერთი მაგალითი; ბ) მეორე მაგალითი

ბმული სიის სულ პირველ (თავდაპირველ, head) ელემენტს უწოდებენ თავს, ხოლო საბოლოო ელემენტს კი უწოდებენ ბოლო (კუდს, tail) ელემენტს. ბმული სიის ბოლო ელემენტი (მიმთითებელი) არის null მნიშვნელობის. მასივისაგან განსხვავებით ბმულ სიაში მონაცემები არ არის განლაგებული მოწესრიგებულად და ეს განლაგება არ ემორჩილება რაიმე კანონზომიერებას. სწორედ ამიტომ ბმულ სიაში სიის კვანძების დაკავშირება ხდება მიმთითებლის (ბმულის) საშუალებით. ბმული სიის ამ თვისების გამო, ბმულ სიაში შეგვიძლია დავამატოთ რამდენიც გვინდა იმდენი ელემენტი, მხოლოდ ელემენტებთან წვდომა შესაძლებელი იქნება თანამიმდევრობით.

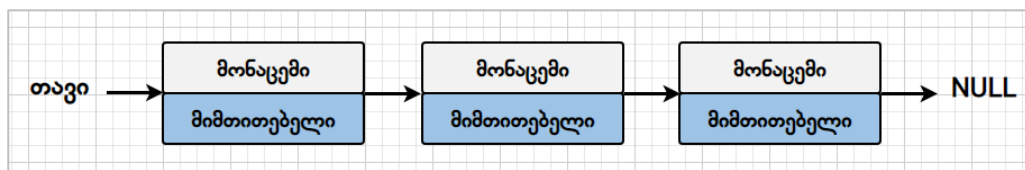
ბმულ სიებთან დაკავშირებით შესაძლებელია ჩავატაროთ შემდეგი ოპერაციები: 1. ელემენტის დამატება (add); 2. ელემენტის ამოშლა (remove); 3. ელემენტის მოძებნა (find); 4. მოცემული პირობის მიხედვით ელემენტების დათვლა (count).

ბმულ სიებთან მიმართებით შესაძლებელია განვიხილოთ სამი ძირითადი ტიპის სია:

1. ცალმხრივად დაკავშირებული სია;
2. ორმხრივად დაკავშირებული სია;
3. წრიულად დაკავშირებული სია.

ცალმხრივად დაკავშირებული სია

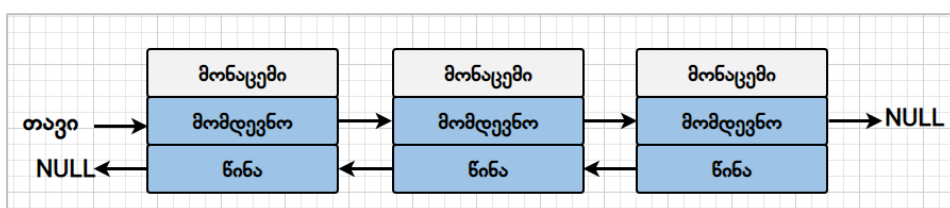
ცალმხრივად დაკავშირებულ სიაში, ყოველი კვანძი ინახავს მისამართს ანუ სიის მომდევნო კვანძის მიმთითებელს. ბოლო კვანძს აქვს მომდევნო კვანძის მისამართი ანუ მიმთითებელი NULL-ზე. ბმული სიებში, ცალმხრივად დაკავშირებული სია ყველზე უფრო მეტად გავრცელებული ტიპია. ყოველ კვანძში არის მიმთითებელი მომდევნო კვანძზე და ელემენტები მოძრაობენ მხოლოდ პირდაპირი მიმართულებით (ნახ. 1.12.).



ნახ. 1.12. ცალმხრივად დაკავშირებული სია

ორმხრივად დაკავშირებული სია

ორმხრივად დაკავშირებული სიაში, გვაქვს ორი მიმთითებელი, რომლებიც დაკავშირებულია თითოეულ კვანძთან, რომელთაგან ერთი მათგანი მიუთითებს მომდევნო კვანძზე და მეორე კი წინა კვანძზე (ნახ. 1.13.).

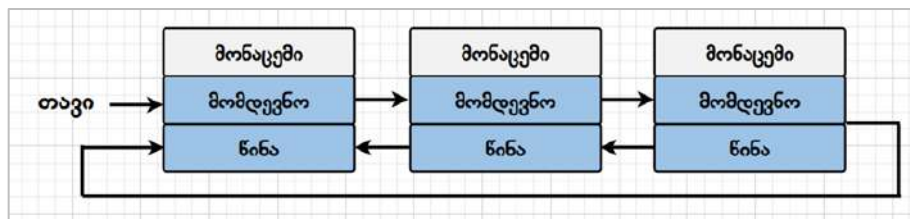


ნახ. 1.13. ორმხრივად დაკავშირებული სია

ორმხრივად დაკავშირებული სიაში ელემენტები მოძრაობენ როგორც პირდაპირი, ისე საპირისპირო მიმართულებით. კვანძებს აქვთ დამატებითი მაჩვენებელი „წინა“ და იგი მიუთითებს წინა კვანძზე.

წრიულად დაკავშირებული სია

წრიულად დაკავშირებულ სიაში არსებული ყველა კვანძი ისე არის ერთმანეთთან დაკავშირებული, რომ ქმნის წრეს. წრიულად დაკავშირებულ სიას ბოლოში არ გააჩნია მნიშვნელობა NULL და ბოლო ელემენტი დაკავშირებულია პირველ ელემენტთან. წრიულად დაკავშირებული სიაში, წინა მაჩვენებლის „თავი“ მიუთითებს „კუდზე“, ხოლო მომდევნო მაჩვენებლის „კუდი“ კი „თავზე“ (ნახ. 1.14.).



ნახ. 1.14. წრიულად დაკავშირებული სია

წრიულად დაკავშირებული სია შეიძლება იყოს როგორც ცალმხრივად, ისე ორმხრივად დაკავშირებული. ბმულ სიებს შორის ყველაზე უფრო გავრცელებული ფორმა არის წრფივი ცალმხრივად დაკავშირებული სია. მაგალითად, ფაილური სისტემა წარმოადგენს წრფივ ცალმხრივად დაკავშირებულ სიას.

მნიშვნელოვანია, ვიცოდეთ: ძირითადი განსხვავება Null და 0 (ნული) მნიშვნელობებს შორის.

Null არის ისეთი სპეციალური მნიშვნელობა, რომელიც მიუთითებს სიცარიელეზე, უფრო სწორად გაურკვევლობაზე, 0 კი კონკრეტული რიცხვითი მნიშვნელობაა და ნიშნავს ელემენტის არარსებობას. ტიპიზებულ ენებში Null არის ისეთი მნიშვნელობა, რომელიც არ მიეკუთვნება არცერთ მონაცემთა ტიპს, ხოლო 0 არის მონაცემთა ტიპის ერთ-ერთი რიცხვითი მნიშვნელობა. Null და 0 არ არის ერთმანეთის ტოლი მნიშვნელობები, რადგან ისინი სხვადასხვა მონაცემს წარმოადგენენ. Null-ის შედარება ნებისმიერი სხვა მნიშვნელობისთან, გარდა undefined-ისა, ყოველთვის იძლევა false-ს. Null გამოიყენება გაურკვეველი მნიშვნელობის აღსანიშნავად, ხოლო 0 გამოიყენება ნულოვანი მნიშვნელობის ან რაიმე ნივთის არარსებობის აღსანიშნავად.

მაგალითად: სტუდენტის საწერ მაგიდაზე არის შემდეგი საკანცელარიო ნივთები: (რა თქმა უნდა, მიყრილ-მოყრილი ☺) საწერი კალამი 7 ცალი, ფანქარი 3 ცალი და რვეული 4 ცალი. სტუდენტი ფიქრობს, რომ საწერი მაგიდის ერთ-ერთ უჯრაში უნდა ჰქონდეს ფლომასტერები, მაგრამ რამდენი ცალია და საერთოდ აქვს თუ არა ისინი მისთვის გაურკვეველია (უჯრის გაღება, ეზარება ☹) ანუ თუ სტუდენტმა ზუსტად იცის რამდენი ცალი კალამი, ფანქარი და რვეული აქვს, ფლომასტერთან მიმართებით მას ზუსტი ინფორმაცია არ აქვს.

მოცემული აღწერა წარმოვადგინოთ ცხრილის ფორმით:

ინდექსი	ელემენტი	რაოდენობა
1	კალამი	7
2	ფანქარი	3
3	რვეული	4
4	ფლომასტერი	Null

სტუდენტმა იცის რომ აქვს ფლომასტერი, მაგრამ არ იცის, თუ რამდენი ცალი აქვს და ამიტომ ველში „რაოდენობა“ მიუთითოთ მნიშვნელობა Null. ფლომასტერების რაოდენობის 0-ით აღნიშვნა არ იქნებოდა სწორი, რადგან 0-ის ჩაწერა ნიშნავს, რომ დანამდვილებით (100%-ით) ვიცით, რომ არ გვაქვს ფლომასტერი ე. ი. სტუდენტს ფლომასტერთან მიმართებით გარკვეული აქვს მისი ზუსტი რაოდენობა და იცის, რომ უჯრაში ფლომასტერი ნამდვილად არ დევს.

მონაცემთა სტრუქტურა. პრეფიქსული ხე

პრეფიქსული ხე ფართოდ გავრცელებული მონაცემთა სტრუქტურაა, რომელიც გამოიყენება ასოციაციური მასივის რეალიზაციისთვის, სადაც გასაღების როლს ასრულებს სტრიქონი. განვიხილოთ თუ რა განსხვავებაა ჩვეულებრივ მასივსა და ასოციაციურ მასივს შორის. ჩვეულებრივი მასივში მონაცემთან წვდომა შესაძლებელია მხოლოდ მთელი ტიპის (რიცხვის) ინდექსის მიხედვით, რაც ყოველთვის არ არის მოხერხებული.

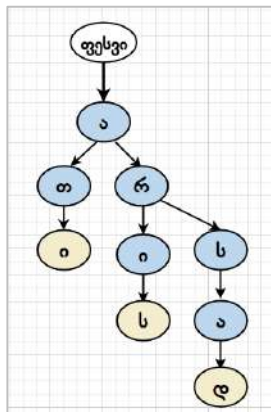
ამიტომ პროგრამირებაში და მონაცემთა სტრუქტურაში შემოტანილ იქნა მასივის უფრო განზოგადოებული სახე, ასოციაციური მასივი, სადაც ელემენტი წარმოდგენილია შემდეგი ფორმით: „გასაღები-მნიშვნელობა“. ასოციაციური მასივის შემთხვევაში მონაცემთან წვდომა ხორციელდება გასაღების მიხედვით, გასაღების როლში კი შეიძლება გამოყენებულ იქნას ნებისმიერი რამ, მაგალითად, სტრიქონი ან სიმბოლოების თანამიმდევრობა.

ასოციაციური მასივი: { გასაღები1 : მნიშვნელობა_1; გასაღები2 : მნიშვნელობა_2; }

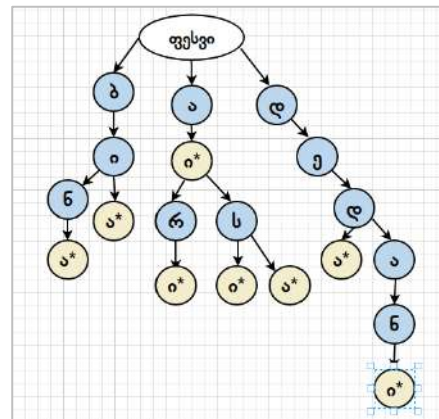
ჩვეულებრივი მასივი: { მნიშვნელობა_1; მნიშვნელობა_2; }

ვთქვათ მოცემული გვაქვს სამი სიტყვა: ათი, არის, არსად. ვნახოთ, თუ როგორ ხდება მოცემული სიტყვების შენახვა პრეფიქსულ ხეში (იხ. ნახ. 1.15.).

ნახ. 1.15-დან ჩანს, რომ სიტყვები განლაგებულია ზევიდან ქვევით. ყვითელი ფერით შეღებილი კვანძი გვიჩვენებს სიტყვის დასარულს. პრეფიქსული ხის თითოეული კვანძი შეიცავს სიტყვის ერთ ასო-ნიშანს. სიტყვის დასაწერად უნდა გავყვეთ პრეფიქსული ხის ფოთლებს/შვილებს, რაც საშუალებას გვაძლევს სიტყვა ჩავწეროთ დაშლილი ასო-ნიშნების სახით. პრეფიქსული ხის ნებისმიერი კვანძი შეიცავს ასო-ნიშანს და იმ ლოგიკურ მნიშვნელობას, რომელიც მიგვითითებს სიტყვის დასასრულზე (ნახ. 1.16.).



ნახ. 1.15. პრეფიქსული ხე



ნახ. 1.16. პრეფიქსული ხე მიმთითებლით

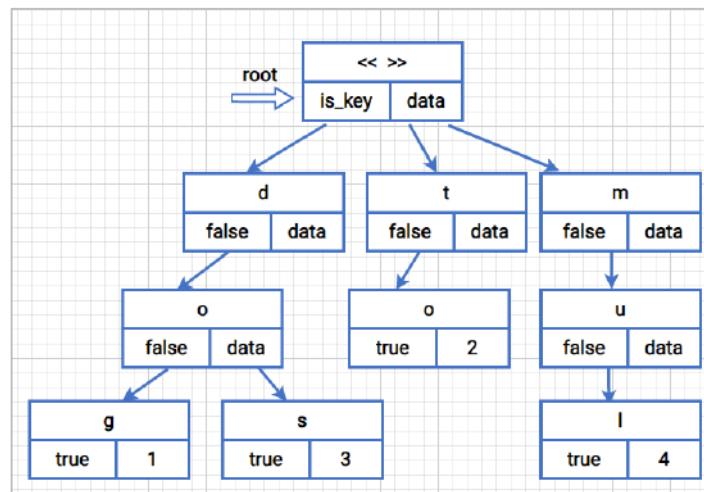
პრეფიქსული ხის მუშაობა დაფუძნებულია ასოციაციურ მასივებზე. ასოციაციური მასივის ყველაზე უფრო გავრცელებული მაგალითი არის დიდი მოცულობის ტექსტში წინასწარ მოცემული სიტყვის დათვლა. ინდექსს ანუ გასაღებს წარმოადგენს სწორედ ეს დასათვლელი სიტყვა.

განვიხილოთ პრეფიქსული ხის აგება/ჩამოყალიბების ალგორითმი. ვთქვათ, მოცემული გვაქვს შემდეგი სახის ოთხი წყვილი „გასაღები-მნიშვნელობა“ ნაკრები:

arrayis["dog"] = 1; arrayis["to"] = 2; arrayis["dos"] = 3; arrayis["mul"] = 4;

გასაღებებსა (dog, to, dos, mul) და გასაღებების შესაბამის მნიშვნელობებს შორის კავშირი შეგვიძლია წარმოვადგინოთ პრეფიქსული ხის საშუალებით (იხ. ნახ. 1.17.).

თავდაპირველად პრეფიქსულ ხეს აქვს მხოლოდ ფესვი (სათავე კვანძი) ყოველგვარი სიმბოლოების გარეშე. პირველი გასაღები, რომელიც მას ემატება, არის სტრიქონი dog. სტრიქონი dog-ისთვის ერთმანეთის თანამიმდევრობით/მიყოლებით უნდა ჩამოვყალიბოთ სამი წვერო/კვანძი d, o და g წვეროებსთვის. წვეროებს d და o აქვთ is_key = false მდგომარეობა, ხოლო ბოლო წვეროს g-ს აქვს მნიშვნელობა is_key = true, რადგან ის პირდაპირ ინახავს მონაცემებს გასაღები dog-ისთვის. ანალოგიურად ხდება გასაღები to-სთვის წვეროების დამატება. ასევე ხდება ორი ცალკეული კვანძის დამატება o წვეროსთვის. შემდეგი გასაღები არის dos. d და o წვეროები უკვე წარმოდგენილია პრეფიქსულ ხეში, ამიტომ საკმარისია დავამატოთ სხვა წვერო, კერძოდ, s წვერო, როგორც o წვეროს შვილი. წვერო s-ის მდგომარეობაა is_key=true და ინახავს მონაცემს data=3. პრეფიქსული ხის ბოლო გასაღები mul იქმნება, სამი ახალი კვანძით. კვანძში l ვწერთ is_key=true და data=4.



ნახ. 1.17. პრეფიქსული ხის მაგალითი

პრეფიქსულ ხეში გასაღების ძებნის ალგორითმი

დავუშვათ, მოცემული გვაქვს უკვე აგებული/ჩამოყალიბებული პრეფიქსული ხე (იხ. ნახ. 1.17.). გვინდა ამოვიღოთ/მოვძებნოთ მონაცემი რომელიმე გასაღებური სიტყვა თვისი შესაბამისი მნიშვნელობით, მაგალითად, მნიშვნელობით: value = arrayis["dos"].

გავარკვით იმყოფება თუ არა გასაღები dos პრეფიქსულ ხეში. თავიდან ავირჩიოთ გასაღების პირველი სიმბოლო d და მოვძებნოთ იგი პრეფიქსულ ხეში ფესვური კვანძიდან დაწყებული შვილობილი კვანძების დასრულებით. აღმოჩნდა, რომ პრეფიქსულ ხეში გვაქვს სიმბოლო d, ამიტომ მაჩვენებელი/მიმთითებელი მიუთითებს მის კვანძზე/წვეროზე.

გასაღების შემდეგი სიმბოლოა o. მაშასადამე, d წვეროს შვილობილ კვანძებს შორის იძებნება o სიმბოლოიანი წვერო. სიმბოლოიანი წვერო o არის პრეფიქსულ ხეში და მაჩვენებელი/მიმთითებელი უკვე გადადის მასზე. დაბოლოს, გასაღების ბოლო სიმბოლოა s.

კვანძი o-ს შვილობილ წვეროებს შორის იძებნება წვერო სიმბოლოთი s და ახლა მაჩვენებელი/მიმთითებელი გადადის მასზე. შემდეგ მოწმდება სიმბოლო s. სიმბოლო s-სთვის დავადგინოთ თუ რომელ კვანძს მიეკუთვნება: შუალედურს თუ ბოლოს. სიმბოლო s-სთვის ალამი არის ჭეშმარიტი (is_key=true ე.ი. სიმბოლო s ბოლოა), რაც ნიშნავს, რომ გასაღები dos იმყოფება პრეფიქსულ ხეში და გაცემს იმ მონაცემს, რომლიც ინახება ველში data (მნიშვნელობა 3).

ახლა დავუშვათ, რომ გვინდა მოვძებნოთ გასაღებური სიტყვა do (value = arrayis["do"]). მოძრაობა უნდა დავიწყოთ პრეფიქსული ხის ფესვიდან. თავიდან ჯერ გადავიდეთ კვანძ d-სთან, შემდეგ კი კვანძ o-სთან. კვანძი o მართლაც არსებობს პრეფიქსულ ხეში, მაგრამ მისი მნიშვნელობა არის is_key=false, რაც ნიშნავს იმას, რომ კვანძი o არის შუალედური და არა ბოლო კვანძი ნებისმიერი გასაღებისთვის. აქედან გამომდინარე, შეგვიძლია ვთქვათ, რომ გასაღები do არ არსებობს მოცემულ პრეფიქსულ ხეში (იხ. ნახ. 1.17.).

ახლა კი გადავიდეთ ცოტა განსხვავებული შემთხვევის განხილვაზე.

მაგალითად, გვჭირდება მოვძებნოთ გასაღები till (value = arrayis["till"]). ძებნას ვიწყებთ ფესვური კვანძიდან და გადავდივართ t კვანძზე. შემდეგ ვეძებთ ამ კვანძის შვილობილ კვანძს სიმბოლოთი i. პრეფიქსულ ხეში სიმბოლო i არ არის.

აქედან დასკვნა: მოცემულ პრეფიქსულ ხეში გასაღები till არ არსებობს. ზემოთ განხილული მაგალითიდან საკმაოდ კარგად ჩანს პრეფიქსულ ხეში გასაღების პოვნის ალგორითმის მუშაობის პრინციპი.

თუ გადავირბენთ გასაღების თითოეულ სიმბოლოზე (ასო/ნიშანზე) და ბოლოს აღმოვაჩენთ კვანძს მნიშვნელობით is_key=true, მაშინ გასაღები იმყოფება ხეში, ყველა სხვა შემთხვევაში კი გასაღები არ იმყოფება პრეფიქსულ ხეში.

პრეფიქსულ ხეში გასაღების წაშლის ალგორითმი

მოცემული გვაქვს წინასწარ აგებული პრეფიქსული ხე და საჭიროა რომელიმე გასაღებური სიტყვის ამოშლა პრეფიქსული ხიდან (იხ. ნახ. 1.18.). ნახატ 1.18-ზე მოცემულ პრეფიქსულ ხეში გვაქვს შემდეგი გასაღებური მნიშვნელობები: do, dog, dos, to, mul.

პრეფიქსული ხიდან ამოვშალოთ გასაღები do.

პრეფიქსული ხის სტრუქტურიდან ვხედავთ, რომ წვერო o არის შუალედური წვერო g და s წვეროებისთვის. სწორედ ამიტომ წვერო o უნდა მოვნიშნოთ შუალედურად ანუ მისი ალამი is_key გადავიყვანოთ მდგომარეობაში false.

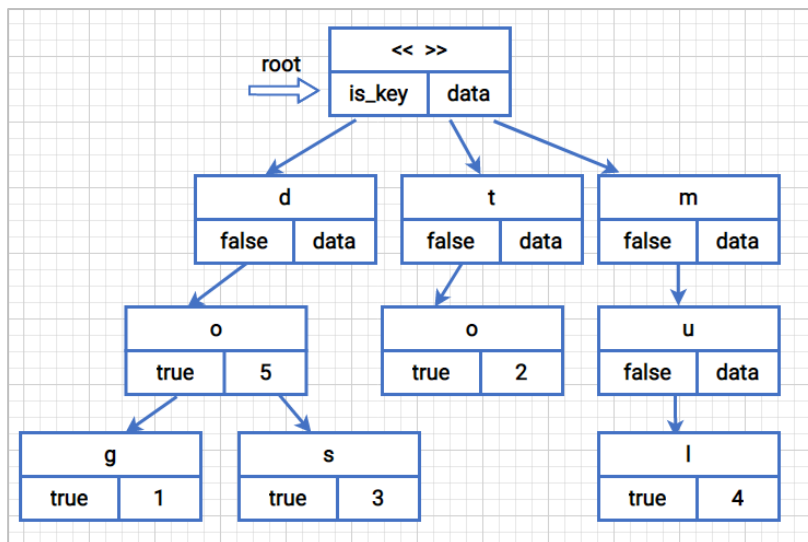
მოხდა პრეფიქსული ხიდან წვერო do-ს ამოიშალა.

ახლა კი განვიხილოთ გასაღები mul-ის ამოშლა. წვეროები m, u და l განლაგებული არიან თანამიმდევრობით.

წვერო l-ს არ ჰყავს მემკვიდრეები. ასეთ შემთხვევაში უნდა განვიხილოთ ამოშლის ორი ვარიანტი:

ვარიანტი 1. წვერო l მოვნიშნოთ როგორც არა საბოლოო წვერო ანუ წვერო l-სთვის is_key = false.

ვარიანტი 2. სათითაოდ ამოვშალოთ ყველა ის წვერო, რომლებიც დაკავშირებულია mul გასაღებურ სიტყვასთან. წაშლა უნდა დავიწყოთ ბოლო l წვეროდან, შემდეგ წავშალოთ წვერო u, ხოლო შემდეგ კი წვერო m. შესაბამისად მოხდება იმ სამივე წვეროს ამოშლა, რომელიც დაკავშირებულია გასაღებურ სიტყვა mul-თან.



ნახ. 1.18. პრეფიქსული ხის მაგალითი

პრეფიქსული ხის სტრუქტურა მეხსიერებას საკმაოდ ეფექტურად იყენებს. ეს ხდება იმიტომ, რომ პრეფიქსულ ხეებში საკმაოდ ხშირად გასაღებური სიტყვების პრეფიქსები ემთხვევა ერთმანეთს.

მაგალითად, გასაღებურ სიტყვებს dog, dos აქვს საერთო პრეფიქსი do ანუ ამ ორი სტრიქონისთვის გამოიყენება წვეროების ერთი და იგივე წყვილი d და o.

თუ მოგვიანებით მოხდება ახალი გასაღებური სიტყვების დამატება იმავე პრეფიქსით, მაგალითად, dora ან doga და ა.შ., მაშინ გამოყენებული იქნება არსებული წვეროები და დაემატება ახალი. ამდენად კარგად ჩანს, თუ რატომ მიიღეს ასეთი სტრუქტურის ხეებმა

პრეფიქსული ხეების სახელი. სიმბოლოების ნებისმიერი საწყისი (პრეფიქსული) თანამიმდევრობისთვის, შეგვიძლია მივიღოთ გასაღებური სიტყვების ნაკრები, რომლებიც შეესაბამება პრეფიქსს. განხილულ მაგალითში (იხ. ნახ. 1.17.) პრეფიქსით *do* შეგვიძლია გამოვყოთ ორი გასაღებური სიტყვა: *dog* და *dos*.

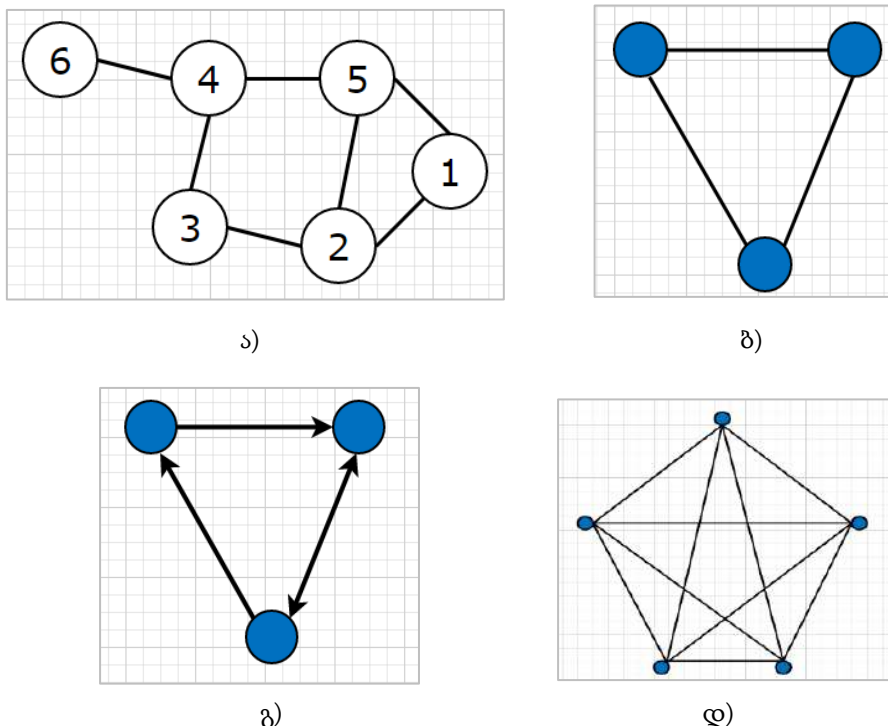
პრეფიქსით *d* კი შესაძლებელია სამი გასაღებური სიტყვის გამოყოფა: *do*, *dog* და *dos*. სწორედ ამიტომ პრეფიქსული ხის სტრუქტურა გამოიყენება მესენჯერში, რადგან იქ პრეფიქსული ხის ალგორითმზე დაყრდნობით ხდება სიტყვების მყისიერი შემოთავაზება.

ამრიგად, პრეფიქსული ხის სტრუქტურის გამოყენება არ შემოიფარგლება მხოლოდ ასოციაციური მასივებით.

მონაცემთა სტრუქტურა. გრაფი

წერტილებისა და მათი შემაერთებული ხაზების სტრუქტურას გრაფი ეწოდება.

გრაფის წერტილებს მისი კვანძები (წვეროები) ეწოდება, ხოლო ხაზებს კი - წიბოები. გრაფი შედგება წვეროებისა და წიბოებისაგან. გრაფი ვიზუალურად წარმოდგენილია წერტილებით ან წრეებით, რომელთაგანაც ზოგიერთი, ყველა ან არც ერთი შეერთებულია ხაზებით ან რკალებით (წიბოებით). თუ გრაფი მიმართულია, წიბოს (ხაზს) გააჩნია ისარი, რომელიც მიმართულია რომელიმე წვეროსაკენ (ნახ. 1.19. ა, ბ, გ, დ).



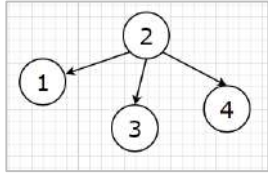
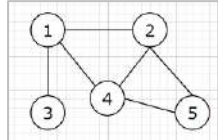
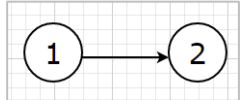
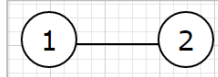
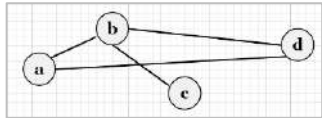
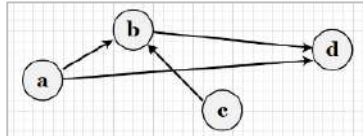
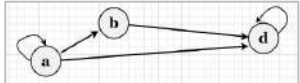
ნახ. 1.19. გრაფი: ა) ექვსი წვეროთი და შვიდი წიბოთი; ბ) სამი წვეროთი და სამი წიბოთი;

გ) მიმართული სამი წვეროთი და ოთხი წიბოთი;

დ) სრული ხუთი წვეროთი და ათი წიბოთი

განვიხილოთ გრაფებთან დაკავშირებული ძირითადი ცნებები და განსაზღვრებები (ცხრილი 1.6. და ნახ. 1.20. ა, ბ, გ, დ, ე, ვ, ზ).

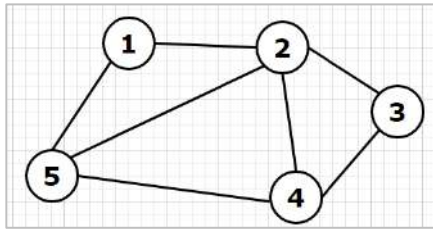
ცხრილი 1.6. გრაფის ტიპები

გრაფის ტიპი	სქემა
ორიენტირებული გრაფი. ორიენტირებულ გრაფში ყველა წიბოს აქვს საკუთარი მიმართულება. ორიენტირებულ გრაფში (1,2) და (2,1) სხვადასხვა წიბოს აღნიშნავს. წიბოს, რომელიც წვეროს საკუთარ თავთან აერთებს, უწოდებენ ციკლურ წიბოს. ორიენტირებულ გრაფს შემოკლებით დიგრაფსაც (digraph) უწოდებენ.	 ა)
არაორიენტირებული გრაფი. არაორიენტირებულ გრაფში წიბოთა სიმრავლე შედგება წვეროთა დაულაგებელი წყვილებისაგან. არაორიენტირებულ გრაფში (1,2) და (2,1) ერთსა და იმავე წიბოს აღნიშნავს, ხოლო ციკლურ წიბოს არ განიხილავენ.	 ბ)
წვეროს და წიბოს ურთიერთმიმართება ორიენტირებულ გრაფში. ორიენტირებულ გრაფში (1,2) წიბოს შესახებ იტყვიან, რომ იგი გამოდის 1-ლი წვეროდან და შედის მე-2 წვეროში.	 გ)
წვეროს და წიბოს ურთიერთმიმართება არაორიენტირებულ გრაფში. არაორიენტირებულ გრაფში (1,2) წიბოს შესახებ იტყვიან, რომ იგი 1-ლი და მე-2 წვეროების ინციდენტურია.	 დ)
წვეროს ხარისხი არაორიენტირებულ გრაფში. არაორიენტირებულ გრაფში წვეროს ხარისხს (degree) უწოდებენ ამ წვეროსადმი ინციდენტური წიბოების რაოდენობას. ნახატზე a და d წვეროების ხარისხია 2, b წვეროს ხარისხია 3, ხოლო c წვეროს ხარისხია 1.	 ე)
წვეროს ხარისხი ორიენტირებულ გრაფში. ორიენტირებულ გრაფში განასხვავებენ შემავალ (in-degree) და გამომავალ (out-degree) ხარისხებს (წვეროში შემავალი და გამომავალი წიბოების რაოდენობის მიხედვით) და მათ ჯამს უწოდებენ წვეროს ხარისხს. ნახატზე b წვეროს ხარისხია 3 (შემავალი ხარისხია 2, გამომავალი ხარისხია 1).	 ვ)
წიბოს, რომელიც წვეროს საკუთარ თავთან აერთებს მარყუჟს უწოდებენ.	 ზ) ნახ. 1.20. გრაფი: ა) ორიენტირებული; ბ) არაორიენტირებული; გ) წვეროს და წიბოს ურთიერთმიმართება; დ) წვეროს და წიბოს ინციდენტურობა; ე) a-ს და d-ს ხარისხია 2, b-ს ხარისხია 3, c-ს ხარისხია 1; ვ) b წვეროს ხარისხია 3. (შემავალი ხარისხი 2, გამომავალი ხარისხი 1); ზ) მარყუჟი;

არსებობს გრაფის წარმოდგენის ორი სტანდარტული მეთოდი:

- ა) მოსაზღვრე წვეროების სია (adjacency-list representation);
- ბ) მოსაზღვრეობის მატრიცა (adjacency matrix).

არაორიენტირებული გრაფის გამოსახულება და მისი წარმოდგენა მოსაზღვრე წვეროების სიის მეთოდით და მოსაზღვრეობის მატრიცის მეთოდით (ნახ. 1.21. ა,ბ,გ).



ა)

1	2	5		
2	1	5	3	4
3	2	4		
4	2	5	3	
5	4	1	2	

ბ)

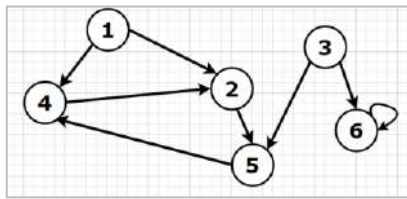
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

გ)

ნახ. 1.21. არაორიენტირებული გრაფის: ა) გამოსახულება;

- ბ) წარმოდგენა მოსაზღვრე წვეროების სიით; გ) წარმოდგენა მოსაზღვრეობის მატრიცით

ორიენტირებული გრაფის გამოსახულება და მისი წარმოდგენა მოსაზღვრე წვეროების სიის მეთოდით და მოსაზღვრეობის მატრიცის მეთოდით (ნახ. 1.22. ა,ბ,გ).



ა)

1	2	4			
2	5				
3	6	5			
4	2				
5	4				
6	6				

ბ)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

გ)

ნახ. 1.22. ორიენტირებული გრაფის: ა) გამოსახულება;

- ბ) წარმოდგენა მოსაზღვრე წვეროების სიით; გ) წარმოდგენა მოსაზღვრეობის მატრიცით

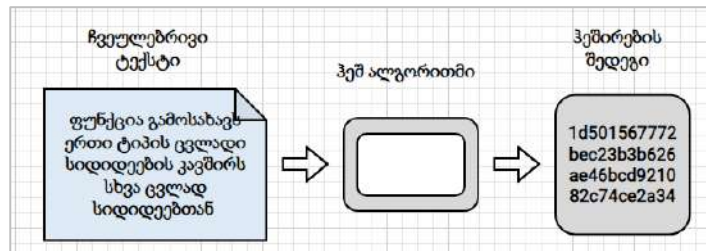
მონაცემთა სტრუქტურა ჰეშცხრილი (ჰეში, ჰეშფუნქცია, ჰეშირება)

ჰეში (ჰეშფუნქცია) არის მათემატიკური ალგორითმი, რომელიც ნებისმიერი ზომის მონაცემთა ნაკრებს გარდაქმნის ფიქსირებული სიგრძის მნიშვნელობად. ჰეშირება არის კონკრეტული ტიპის ნებისმიერი რაოდენობის ინფორმაციის (მასივის) გარდაქმნა სიმბოლოების უნიკალურ ნაკრებად (ფიქსირებული სიგრძის ბიტობრივ სტრიქონად).

გარდაქმნას უწოდებენ ჰეშფუნქციას, ხოლო გარდაქმნის შედეგად მიღებულ სიმბოლოების უნიკალურ ნაკრებს კი უწოდებენ ჰეშს ან ჰეშკოდს. მონაცემის გარდაქმნის შედეგი ჰეში, იდეალურ შემთხვევაში, უნიკალურია ნებისმიერი სახის/ფორმის ინფორმაციის ნაკრებისთვის (იხ. ნახ. 1.23.).

საბუნებისმეტყველო მეცნიერებებში არსებობს ჰეშ-ფუნქციების უამრავი სახეობა. ისინი ერთმანეთისაგან განსხვავდება გამოთვლის მეთოდებით, საიმედოობით, დანიშნუ-

ლების მიხედვით და სხვა პარამეტრებით. ჰეშფუნქცია იღებს შესასვლელ მნიშვნელობას, მაგალითად მონაცემთა სტრიქონს და იბრუნებს ფიქსირებული ზომის რიცხვით მნიშვნელობას.



ნახ. 1.23. ჰეშირების ზოგადი სქემა

იდეალურ ჰეშფუნქციას უნდა გააჩნდეს შემდეგი თვისებები: 1. ჰეშფუნქცია უნდა იყოს მარტივი (გამოთვლის კუთხით) და მუშაობდეს საკმაოდ სწრაფად; 2. ჰეშფუნქციას უნდა შეეძლოს დიდი დიაპაზონის მქონე ჰეშმნიშვნელობის დაბრუნება; 3. ჰეშფუნქციას უნდა შეეძლოს უნიკალური ჰეშის გენერირება ნებისმიერი შესატანი მნიშვნელობისთვის კოლიზიის გარეშე; ჰეშფუნქციამ უნდა დაიყვანოს მინიმუმამდე კოლიზიის რაოდენობა ანუ ისეთი სიტუაცია, როდესაც სხვადასხვა გასაღების წყვილს შეესაბამება ჰეშფუნქციის ერთი და იგივე მნიშვნელობა; 4. ჰეშფუნქციის მიერ გენერირებულ და მიღებულ ჰეშმნიშვნელობებს არ უნდა ჰქონდეთ გამოხატული განაწილების კანონზომიერება; ფუნქციამ ჰეშცხრილში გასაღებები უნდა გადაანაწილოს რაც შეიძლება თანაბრად.

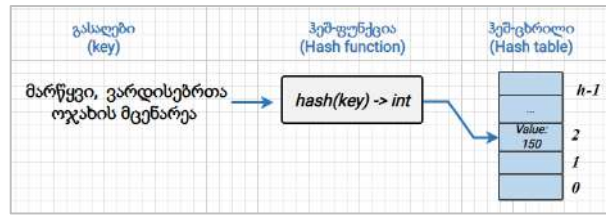
განმარტება: კოლიზია, შეჯახება - მონაცემების დამახინჯება ან არმიღება მათი ორი ან მეტი მოწყობილობიდან ერთდროულად გადაცემისას ერთსა და იმავე მოწყობილობაზე; ჰეშ-ფუნქციას უნდა შეეძლოს ერთმანეთისაგან განსხვავებული ჰეშმნიშვნელობების გენერირება ერთი და იგივე შესატანი მნიშვნელობისთვის;

რეალურად იდეალური ჰეშფუნქცია არ არსებობს, მაგრამ ნებისმიერი ჰეშფუნქციის ალგორითმი ცდილობს მაქსიმალურად მიუახლოვდეს იდეალურ შედეგს.

ჰეშცხრილი (Hash Table) არის დინამიური მონაცემთა სტრუქტურა, სადაც მონაცემების შენახვა ხდება წყვილის: „გასაღები-მნიშვნელობა“ სახით. გასაღები არის ინდექსი ანუ უნიკალური ნატურალური რიცხვი, ხოლო მნიშვნელობა პირდაპირ დაკავშირებულია გასაღებთან. მონაცემზე ეფექტურ წვდომას უზრუნველყოფს ჰეშცხრილის გასაღები. გასაღები შეუიძლება იყოს სტრიქონი, რიცხვი, მიმთითებელი და ა. შ.

ჰეშცხრილი მიიღება შემდეგნაირად: ჰეშ-ფუნქცია გასაღებს გარდაქმნის მასივის ელემენტის ინდექსად. ინდექსი არის მთელი რიცხვი და იმყოფება $0 \div (h-1)$ დიაპაზონში. ჰეშფუნქციის გამოთვლის დრო დამოკიდებულია გასაღების ზომაზე (სიგრძეზე) და საერთოდ არ არის დამოკიდებული ცხრილის ელემენტების რაოდენობაზე (იხ. ნახ. 1.24.). ჰეშცხრილის აგება/ფორმირების შემდეგ შესაძლებელია განვახორციელოთ მონაცემის დამატების, ძებნის და წაშლის ოპერაციები. ჰეშცხრილში დამატების, წაშლისა და ძებნის ოპერაციები

სრულდება უფრო სწრაფად ვიდრე სხვა მონაცემთა სტრუქტურებში, მაგალითად, ისეთებში, როგორიცაა დინამიკური მასივები და ბმული სიები.



ნახ. 1.24. გასაღების ინდექსად გარდაქმნის სქემა

დავუშვათ, მოცემული გვაქვს თანამშრომლების მონაცემთა ბაზა. მოთხოვნა არის შენდები სახის: საჭიროა მოვძებნოთ თანამშრომელი მისი ელექტრონული ფოსტის მიხედვით. მოთხოვნის გადასჭრელად შეგვიძლია შევქმნათ ჰეშცხრილი, რომლის გასაღების როლში იქნება ელექტრონული ფოსტა, ხოლო მნიშვნელობა იქნება თანამშრომლის მონაცემები. თანამშრომლის მოსაძებნად გამოვიყენოთ ჰეშფუნქცია. ჰეშფუნქცია ელექტრონულ ფოსტას გარდაქმნის ჰეშცხრილის ინდექსად და გარდაქმნის შედეგად მცისიერად ვიღებთ თანამშრომლის მონაცემებს, რითაც თავიდან ვიცილებთ მთლიან მონაცემთა ბაზის გადარბენა/დათვალიერებას და საკმაოდ ვამცირებთ დროის დანახარჯებს (ვიგებთ დროში) ე.ი. ჰეშცხრილი უზრუნველყოფს როგორც მონაცემის სწრაფ ძებნას, ისე მცხსიერებაში მონაცემების ოპტიმალურად გადანაწილებას.

ჰეშირების დროს შესაძლებელია ორი განსხვავებული მონაცემის ჰეშმნიშვნელობა ერთი მეორეს დაემთხვეს. ასეთ შემთხვევაში იტყვიან, რომ მოხდა **კოლიზია ანუ შეჯახება** (collision ლათინური სიტყვაა და ქართულად ნიშნავს შეჯახებას, შეხლა-შემოხლას). სხვა სიტყვებით რომ ვთქვათ, კოლიზია არის ისეთი მდგომარეობა, როდესაც სხვადასხვა გასაღებები (შესატანი მნიშვნელობები) ქმნიან ერთსა და იმავე რიცხობრივ ინდექსებს (ჰეშმნიშვნელობებს). **კოლიზია განიხილება როგორც მონაცემების დაუცველობა.** ზოგადად ჰეშირების პროცესი ჰეშფუნქციის გამოძახების გარდა, აუცილებლად მოიცავს კოლიზიის აღმოჩენას და დამუშავებას. არსებობს ჰეშირების შემდეგი ალგორითმები: ა)ჰეშირება ნაშთიანი გაყოფით; ბ)ჰეშირება გამრავლებით; გ)უნივერსალური ჰეშირება; (იხ. ნახ. 1.25.).

ინდექსი	1	5	12	1	12	13	5
გასაღები	111001	111205	111312	114101	111512	111313	111305
				კოლიზია	კოლიზია		კოლიზია

ნახ. 1.25. კოლიზია

ავაგოთ ჰეშცხრილი. ვთქვათ ჰეშცხრილის ზომა არის 11. ჰეშცხრილის ინდექსი დავაგენერიროთ ნაშთიანი გაყოფის მეთოდით: $H(x) = x \bmod 11$; ვიგულისხმობთ, რომ ჰეშცხრილი თავიდან არის ცარიელი (იხ. ნახ. 1.26. ა.ბ.გ.დ.ე.ვ.ზ.).

ჰეშცხრილში რიცხვის (გასაღების) დამატება:

დავამატოთ 52; $52 \bmod 11 = 8$;

დავამატოთ 37; $37 \bmod 11 = 4$;

ჰეშცხრილში რიცხვის (გასაღების) მოძებნა:

მოძებნოთ რიცხვი 16; $16 \bmod 11 = 5$; ელემენტი 16 არ მოიძებნა;

მოძებნოთ რიცხვი 19; $19 \bmod 11 = 8$; ელემენტი 19 არ მოიძებნა;

მოძებნოთ რიცხვი 37; $37 \bmod 11 = 4$; ელემენტი 37 მოიძებნა;

დავამატოთ ახალი რიცხვი (გასაღები) 96. 96-ის დამატება გამოიწვევს კოლიზიას (ჩვენ ვცდილობთ, მონაცემი დავამატოთ უკვე დაკავებულ უჯრაში).

$96 \bmod 11 = 8$; **კოლიზია:**

0	1	2	3	4	5	6	7	8	9	10	11
X	X	X	X	X	X	X	X	52	X	X	X

ა)

0	1	2	3	4	5	6	7	8	9	10	11
X	X	X	X	37	X	X	X	52	X	X	X

ბ)

0	1	2	3	4	5	6	7	8	9	10	11
X	X	X	X	37	X	X	X	52	X	X	X

გ)

0	1	2	3	4	5	6	7	8	9	10	11
X	X	X	X	37	X	X	X	52	X	X	X

დ)

0	1	2	3	4	5	6	7	8	9	10	11
X	X	X	X	37	X	X	X	52	X	X	X

ე)

0	1	2	3	4	5	6	7	8	9	10	11
X	X	X	X	37	X	X	X	52	X	X	X

ვ)

ნახ. 1.26. ჰეშცხრილში რიცხვის ა) დამატება; ბ) დამატება; გ) მოძებნა;

დ) მოძებნა; ე) მოძებნა; ვ) კოლიზია;

კოლიზიის გადაჭრის პირველი გზა

კოლიზიის გადაჭრის ერთ-ერთი გზაა ელემენტთა გადაბმა (chaining). ამ მეთოდს ჰეშირება გადაბმით (ჯაჭვებით, სიებით) ეწოდება. მისი არსი იმაში მდგომარეობს, რომ

ერთნაირი ჰეშმნიშვნელობის მქონე ელემენტებისაგან ყალიბდება ბმული სია. მასივის უჯრაში შესაძლებელია შევინახოთ მნიშვნელობების სიები. ახალი მნიშვნელობა ემატება სიის ბოლოში (იხ. ნახ. 1.27. ა,ბ).

ჰეშცხრილში დავამატოთ შემდეგი ელემენტები: 45, 93, 51, 12.

დავამატოთ 45; $45 \bmod 11 = 1$; **კოლიზია**;

დავამატოთ 93; $93 \bmod 11 = 5$;

დავამატოთ 51; $51 \bmod 11 = 7$;

დავამატოთ 12; $12 \bmod 11 = 1$; **კოლიზია**;

კოლიზიის გადაჭრის მეორე გზა

კოლიზიის გადაჭრის მეორე გზაა ელემენტთა გადაადგილება, გადატანა. ამ მეთოდს ჰეშირება გადაადგილებით ეწოდება (იხ. ნახ. 1.27. ა,ბ).

ზოგჯერ საჭიროა, რომ გამოვიყენოთ ბმული სიის მეთოდზე გაცილებით მარტივი მეთოდი და არ შევინახოთ სიების მასივები. ასეთი მეთოდის არსი კი შემდეგია: თუ ჩვენ ვერ ვახერხებთ ელემენტის ჩამატებას საჭირო უჯრაში, მაშინ ვცდილობთ, ჩავამატოთ შემდეგ უჯრაში და ა. შ. სანამ არ მოვძებნით ცარიელ ადგილს. ცარიელი უჯრის მოსაძებნად გადავურბენთ გასაღებებს მანამდე, სანამ არ ვნახავთ ცარიელ უჯრას. თუ მივალთ მასივის ბოლოს, გადავალთ მასივის პირველ ელემენტზე.

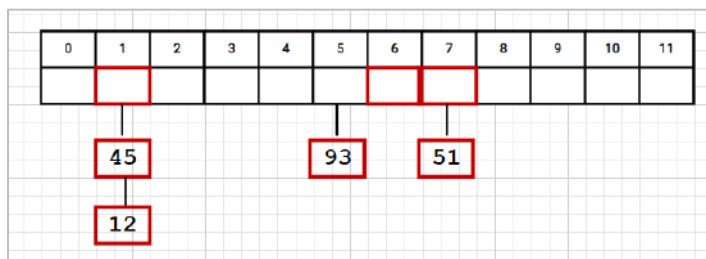
ჰეშცხრილში დავამატოთ შემდეგი ელემენტები: 45, 12, 95, 24.

დავამატოთ 45; $45 \bmod 11 = 1$; **კოლიზია**;

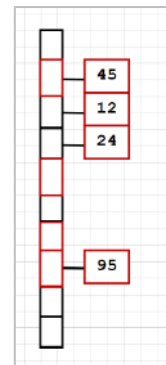
დავამატოთ 12; $12 \bmod 11 = 1$; **კოლიზია**;

დავამატოთ 95; $95 \bmod 11 = 7$;

დავამატოთ 24; $24 \bmod 11 = 2$;



ა)



ბ)

ნახ. 1.27. კოლიზიის გადაჭრა: ა) ჰეშირება გადაბმით; ბ) ჰეშირება გადაადგილებით

ამდენად, პირველ თემაში განვიხილეთ როგორც მონაცემთა მარტივი სტრუქტურები: მასივი, სტეკი, რიგი და დეკი, ასევე მონაცემთა რთული სტრუქტურები: ბმული სიები, გრაფები, პრეფიქსული ხეები, ჰეშცხრილები.

ალგორითმის ცნება. ამოცანის ჩამოყალიბება, ალგორითმის ცნება, მარტივ მაგალითებზე ალგორითმის აღწერა, რეკურსიული და იტერაციული ალგორითმების მაგალითები. ფორმალური მტკიცების მეთოდები. მათემატიკური ინდუქციის, დიაგონალიზაციისა და დირიხლეს პრინციპი, მათი გამოყენება თეორემათა მტკიცების დროს.

ამოცანის ჩამოყალიბება. ალგორითმის ცნება

მთავარი მოთხოვნა დასმული ამოცანის ამოსახსნელად არის მათემატიკური მოდელის, ალგორითმის და თავად ამოცანის ჩამოყალიბება.

ამოცანის ჩამოყალიბების და ალგორითმის აღწერის ერთ-ერთი ყველაზე მნიშვნელოვანი ეტაპია დასმული ამოცანის სწორად და ეფექტურად ამოხსნა. ამოცანის ამოხსნის დროს დაშვებული შეცდომები და უზუსტობები, იწვევს სხვა უკვე შესრულებული ეტაპების დროებით შეფერხებას ან უკან დაბრუნებას ან არასწორი შედეგის მიღებას. ითვლება, რომ რთული ამოცანების ამოსახსნელად ალგორითმის შექმნისა და განხორციელებისთვის საჭიროა დროის 40-50% დანახარჯი.

რა არის ალგორითმი (Algorithm)? ზოგადი განმარტებით ალგორითმი არის ნებისმიერი სწორად განსაზღვრული გამოთვლითი პროცედურა, მას შესავლელზე (input) გადაეცემა მონაცემის ან მონაცემთა გარკვეული რაოდენობა და ალგორითმის შესრულების შედეგად პროცედურის გამოსავალზე (output) მიიღება მონაცემის ან მონაცემთა გარკვეული რაოდენობა.

თავდაპირველად ალგორითმის ცნება გულისხმობდა მრავალნიშნა რიცხვებზე მხოლოდ ოთხი არითმეტიკული მოქმედების შესწავლის წესებს.

დღესდღეობით ალგორითმს მრავალი განმარტება აქვს და მეცნიერები მას სხვადასხვანაირად ახასიათებენ:

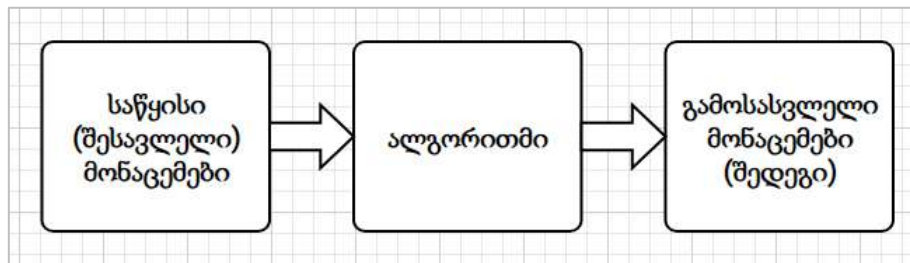
- ალგორითმი არის იმ მოქმედებათა ერთობლიობის ზუსტი და სრული აღწერა, რომელთა მკაცრად განსაზღვრული თანმიმდევრობით შესრულება განაპირობებს დასმული ამოცანის ამოხსნას.
- ალგორითმი მოქმედებათა სასრული მიმდევრობაა, რომელთა ზუსტად შესრულების შემთხვევაში აღწევნ დასახულ მიზანს.
- ალგორითმი წარმოადგენს ამოცანის ამოხსნისათვის საჭირო რაოდენობის ბიჯების თანმიმდევრობას, რომელიც საწყის/შესასვლელ მონაცემებს გარდაქმნის გამოსასვლელ/შედეგ მონაცემებად.

- ალგორითმი არის მკაცრად განსაზღვრული წესების მიხედვით აღწერილი ნებისმიერი გამოთვლითი სისტემა, რომელიც გარკვეული რაოდენობის ბიჯის შესრულების შედეგად მიგვიყვანს დასმული ამოცანის გადაწყვეტამდე (ა. კოლმოგოროვი).

- ალგორითმი არის გამოთვლითი პროცესის ისეთი ზუსტი აღწერა და ჩამოყალიბება, რომელიც საწყისი მონაცემების დამუშავებით მიგვიყვანს სასურველ შედეგამდე.

მე-20 საუკუნეში კომპიუტერული ტექნოლოგიების ფართოდ დანერგვამ ადამიანებს შესაძლებლობა მისცა გადაეწყვიტათ საკმაოდ რთული და შრომატევადი ამოცანები.

კომპიუტერულ მეცნიერებებთან მიმართებაში ალგორითმის აღწერა საჭიროებს დაზუსტებას. კომპიუტერული მეცნიერებების სამყაროში ამოცანის გადაჭრა გულისხმობს ინფორმაციის გარდაქმნას, რაც ნიშნავს იმას, რომ საწყისი მონაცემები და ალგორითმის შესრულების შედეგი წარმოდგენილი უნდა იყოს ინფორმაციის სახით (ნახ. 2.1).



ნახ. 2.1. ალგორითმის მუშაობის ზოგადი სქემა

კომპიუტერულ მეცნიერებებში ალგორითმები უნდა განხორციელდეს პროგრამული კოდების საშუალებით ანუ ალგორითმები განკუთვნილია კომპიუტერული ტექნოლოგიების შესაქმნელად. ალგორითმის შესასრულებლად საჭიროა საკმაოდ დიდი ოპერატიული მეხსიერება და დროითი დანახარჯები. ალგორითმის შედგენისა და აგების პროცესს ალგორითმიზაციას უწოდებენ.

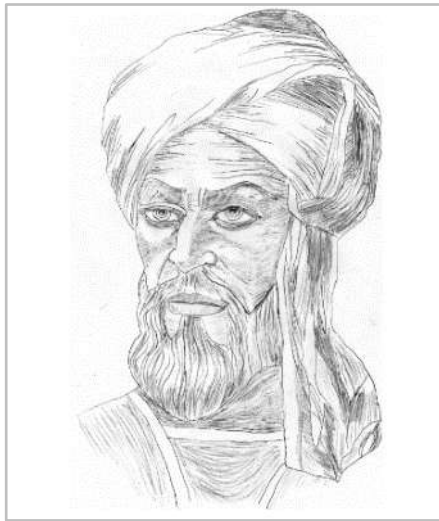
ტერმინი ალგორითმი დაკავშირებულია მსოფლიოში ცნობილი სპარსი მოაზროვნის, მათემატიკოსის და ასტრონომის **მუხამედ ბენ მუსა ალ-ხორეზმის** სახელთან (დაახლოებით 783 - 850 წლები).

მუხამედ ბენ მუსა ალ-ხორეზმი ანალიტიკური გეომეტრიის ფუძემდებელია. მან ჩამოაყალიბა არითმეტიკული მოქმედებების წესები.

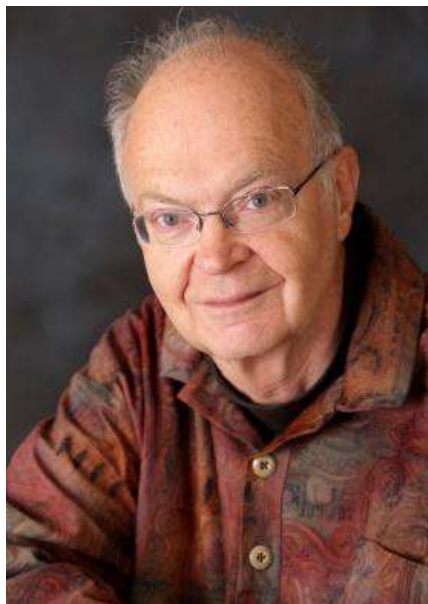
მე-12 საუკუნეში, დასავლეთ ევროპაში, ალ-ხორეზმის მრავალი წიგნი ითარგმნა ლათინურად (იხ. ნახ. 2.2.). ალ-ხორეზმის წიგნების ლათინურ თარგმანში წესებისა და კანონების ჩამოყალიბება უმეტეს შემთხვევაში იწყებოდა სიტყვებით: ალ-ხორეზმმა თქვა (Dixit Algorizmi), ზოგიერთ თარგმანში კი ალ-ხორეზმი მოიხსენიება როგორც Algorithmus. დროთა განმავლობაში ადამიანებმა დაივიწყეს, რომ ალ-ხორეზმი ლათინურად თარგმნილი წიგნების ავტორის გვარია და წესებისა და კანონების ჩამოყალიბებასა და თანამიმდევრობას შემოკლებით უწოდეს ალგორითმი.

ალგორითმის განმარტება დონალდ ერვინ კნუთის მიერ

კომპიუტერული მეცნიერებების წარმოშობასა და განვითარებასთან ერთად, მე-20 საუკუნის თვალსაჩინო ამერიკელი მეცნიერის, სტენფორდის უნივერსიტეტის პროფესორის, პროგრამირების ერთ-ერთი მთავარი იდეოლოგის დონალდ კნუთის (იხ. ნახ. 2.3.) მიერ 1968 წ. შემოტანილ იქნა ალგორითმის შემდეგი განმარტება: **ალგორითმი არის ამოცანის ამოხსნისთვის საჭირო ოპერაციების თანამიმდევრობის განმსაზღვრელი წესები და გააჩნია ხუთი მნიშვნელოვანი მახასიათებელი: სასრულობა, სიზუსტე, შეტანა, გამოტანა, ეფექტურობა.**



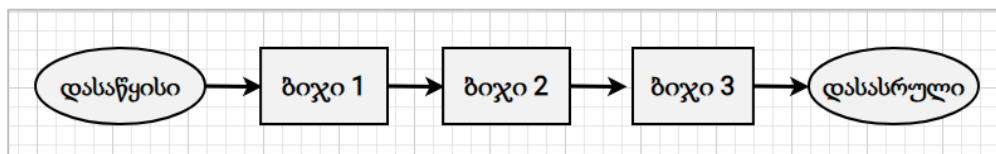
ნახ. 2.2. მუჰამად ბენ მუსა ალ-ხორეზმი (დ. დაახლ. 780, ხვარაზმი - გ. დაახლ. 850) - სპარსი მათემატიკოსი, ასტრონომი და გეოგრაფი. მისი სახელის ლათინურ ვერსიიდან (Algorithmi) მომდინარეობს სიტყვები ალგორითმი და ალგორიზმი.



ნახ. 2.3. დონალდ ერვინ კნუთი (Donald Ervin Knuth; დ. 10 იანვარი, 1938, მილუოკი, უისკონსინის შტატი) - ამერიკელი მეცნიერი, აღიარებული სპეციალისტი კომპიუტერული მეცნიერებების დარგში, სტენფორდის უნივერსიტეტის პროფესორი. ტიურინგის პრემიის ლაურეატი (1974 წელი).

ალგორითმის თვისებები

1. დისკრეტულობა - ამოცანის ამოხსნის ისეთი პროცესი, როდესაც ბრძანებების შესრულება ხდება თანამიმდევრული, მოწესრიგებული მარტივი (ან ადრე განსაზღვრული) ბიჯების (ეტაპების) სახით ანუ მომდევნო ბიჯის შესრულება დაიწყება წინა ბიჯის შესრულების შემდეგ. საქმე გვაქვს ერთმანეთისაგან გამოყოფილი ბიჯების მიმდევრობასთან ანუ ბრძანებათა დისკრეტული სტრუქტურასთან. მაგალითად, სწრაფად მოსამზადებელი კერძის ალგორითმი შეგვიძლია ასე ჩამოვაყალიბოთ: **ბიჯი 1.** სწრაფი მომზადების კერძი გადავიტანოთ ღრმა თევზზე; **ბიჯი 2.** დავასხათ ცხელი წყალი; **ბიჯი 3.** კარგად ავურიოთ; (ნახ. 2.4.)



ნახ. 2.4. ალგორითმის დისკრეტულობა

2. გასაგებობა - ალგორითმის ყოველი ბიჯის თანამიმდევრობა უნდა იყოს ზუსტად განსაზღვრული და ყველა ბრძანება გასაგები უნდა იყოს შემსრულებლისთვის. თითოეულ შემსრულებელს აქვს ბრძანებების ისეთი ნაკრები, რომელთა შესრულებაც მხოლოდ მას შეუძლია. ამასთანავე ნებისმიერ ალგორითმს ჰყავს ერთი ან რამდენიმე შემსრულებელი: შემსრულებლისთვის მკაფიოდ გასაგები უნდა იყოს ბრძანებების შინაარსი და მას უნდა შეეძლოს ალგორითმის მიხედვით გათვალისწინებული ყველა ბრძანების შესრულება. მაგალითად, ინჟინერს შეუძლია მხოლოდ მისთვის დაკისრებული საქმიანობა. მკერავს არ შეუძლია გააკეთოს ინჟინერის საქმე, იმ შემთხვევაშიც კი, თუ მკერავს დაავალებენ ინჟინერის საქმეს.

3. განსაზღვრულობა - ალგორითმის ბრძანებების ყოველი ბიჯი და შესრულების რიგითობა უნდა იყოს თანმიმდევრული, ზუსტი და მკაცრად განსაზღვრული. ყოველმა ბრძანებამ ცალსახად უნდა აღწეროს მოქმედებები.

4. შედეგიანობა - ალგორითმის შესრულება უნდა დასრულდეს გარკვეული შედეგით ანუ უნდა მოხდეს ობიექტის გარდაქმნა საწყისი მდგომარეობიდან საბოლოო მდგომარეობამდე ანუ შედეგამდე. იმ შემთხვევაში თუ შეცდომები იქნება გამორიცხული, ალგორითმმა უნდა მიგვიყვანოს სწორ შედეგთან ნებისმიერი დასაშვები საწყისი მონაცემებისთვის. სასრული რაოდენობის ბრძანებების შესრულების შემდეგ ალგორითმმა უნდა მოგვიყვანოს წინასწარ დაგეგმილ (ცნობილ) შედეგთან.

5. მასობრიობა - შესაძლებელი უნდა იყოს ალგორითმის გამოყენება საწყისი მნიშვნელობების სხვა და სხვა ნაკრებისათვის. ერთი ალგორითმის მიხედვით შესაძლებელია ერთნაირი ტიპის ამოცანების გადაჭრა. მასობრიობის მაგალითია კვადრატული განტო-

ლება. მაგალითად: შევადგინოთ კვადრატული განტოლება სწორი კოეფიციენტებით, რომლის მიხედვითაც კვადრატული განტოლების ამოხსნა მოგვცემს ნამდვილ ფესვებს.

ნამდვილ რიცხვთა სიმრავლე არის ალგორითმის გამოყენების არე, მხოლოდ გარკვეული პირობით, უნდა შეირჩეს კოეფიციენტების ისეთი სამეული, რომლებიც დააკმაყოფილებს შემდეგ პირობას: $a \neq 0, b^2 - 4 \cdot a \cdot c \geq 0$.

დასკვნა: ამდენად კვადრატული განტოლების ამოსახსნელად აღწერილი სამეულის სიმრავლე უნდა ჩავთვალოთ ალგორითმის საწყის მონაცემებად და არა ნებისმიერი სამეული ნამდვილ რიცხვთა სიმრავლიდან.

6. სასრულობა - ალგორითმი სასრული რაოდენობის ბიჯების შესრულების შემდეგ აუცილებლად უნდა დამთავრდეს (დასრულდეს). ალგორითმი, რომელის შესრულებაც არ მთავრდება (ჩაიკვლა) ანუ უსასრულოდ შესრულებადია, იგი, რა თქმა უნდა, შედეგთან არ მიგვიყვანს.

ალგორითმის ჩაწერის ფორმები

ცნობილია ალგორითმის ჩაწერის შემდეგი ფორმა:

სიტყვიერი - ალგორითმის ჩაწერა ბუნებრივ სალაპარაკო ენაზე;

ფსევდოკოდი - ალგორითმის ნახევრად ფორმალიზებული აღწერა, რომელიც შეიცავს დაპროგრამების ენის ელემენტებს, ასევე ბუნებრივი ენის ფრაზებსა და მათემატიკურ ნიშნებს;

გრაფიკული - ალგორითმის ჩაწერა ბლოკსქემის საშუალებით;

მანქანური ენა - ისეთი ენა, რომელზედაც სრულდება ბრძანებები თავად კომპიუტერში;

პროგრამული - ნებისმიერი დაპროგრამების ენა.

ახლა გადავიდეთ მარტივი ამოცანების ალგორითმის ენაზე ამოხსნის განხილვაზე.

მაგალითი 2.1. მოცემულია მართკუთხა სამკუთხედი და მისი კათეტები. ვიპოვოთ ჰიპოტენუსის და პერიმეტრის მნიშვნელობები. შევადგინოთ სიტყვიერი ალგორითმი:

1. დასაწყისი;
2. შემოვიტანოთ a და b რიცხვები;
3. გამოვთვალოთ ჰიპოტენუსა c ($c = \sqrt{a^2 + b^2}$);
4. გამოვთვალოთ ნახევარპერიმეტრი p ;
5. გამოვიტანოთ c -ს და p -ს მნიშვნელობები;
6. დასასრული.

მაგალითი 2.2. მოცემული გვაქვს კვადრატული განტოლება: $ax^2 + bx + c = 0$. დავწეროთ კვადრატული განტოლების ამოხსნის ალგორითმი:

1. დასაწყისი;
2. შემოვიტანოთ საწყისი მნიშვნელობები (კოეფიციენტები): a, b, c ($a, b, c \neq 0$);
3. თუ $a = 0$, მაშინ ფორმულა არ იმუშავებს (ნულზე გაყოფა არ შეიძლება). საქმე გვაქვს წრფივ განტოლებასთან ორი შესაძლო შემთხვევით:
4. თუ $b \neq 0$, მაშინ ფესვი ტოლია $-c/b$;
5. თუ $b = 0$, მაშინ ფესვი არ გვაქვს.
6. თუ $a \neq 0$, მაშინ გამოვითვალოთ დისკრიმინანტი: $D = b^2 - 4ac$;
7. თუ $D < 0$, გადავიდეთ მე-8 პუნქტზე და მივუთითოთ რომ კვადრატულ განტოლებას ამონახსენი არ აქვს;
8. საწინააღმდეგო შემთხვევაში, თუ $D \geq 0$, მაშინ $x_1 = \frac{-b+\sqrt{D}}{2*a}$; $x_2 = \frac{-b-\sqrt{D}}{2*a}$;
9. გამოვიტანოთ x_1 -ის და x_2 -ის მნიშვნელობები;
10. დასასრული.

მაგალითი 2.3. ძვ. წ. აღ. 400-300 წლებში საბერძნეთში ცხოვრობდა დიდი ბერძენი მათემატიკოსი **ევკლიდე**. ევკლიდეს მიერ აღწერილი ალგორითმის მიხედვით შესაძლებელია ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფის პოვნა.

ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფი (უსგ) ეწოდება იმ უდიდეს რიცხვს, რომელზედაც მოცემული რიცხვები უნაშთოდ იყოფა. თავდაპირველად უდიდესი რიცხვი გავყოთ უმცირესზე, შემდეგ უმცირესი რიცხვი გავყოთ მიღებულ ნაშთზე, შემდეგ პირველი ნაშთი გავყოთ მეორე ნაშთზე და ა.შ. ოპერაციები გავაგრძელოთ მანამ, სანამ ნაშთი ნულის ტოლი არ გახდება. რიგით ბოლო გამყოფი უდიდეს საერთო გამყოფს წარმოადგენს. ვინაიდან გაყოფის ოპერაცია მრავალჯერ-განმეორებად გამოკლების ოპერაციაზე დაიყვანება, ამიტომ ალგორითმი შემდეგნაირად შეიძლება ჩამოვყალიბოთ:

1. დასაწყისი;
2. საწყისი მონაცემების შეტანა (a, b);
3. თუ $a > b$ გადავიდეთ მე-5 პუნქტზე, საწინააღმდეგო შემთხვევაში გადავიდეთ მე-4 პუნქტზე;
4. თუ $b > a$ გადავიდეთ მე-6 პუნქტზე, საწინააღმდეგო შემთხვევაში გადავიდეთ მე-7 პუნქტზე;
5. $a = a - b$; გადავიდეთ მე-3 პუნქტზე;
6. $b = b - a$; გადავიდეთ მე-3 პუნქტზე;
7. უსგ = a ;
8. შედეგის (უსგ) ამობეჭდვა;
9. დასასრული.

განვიხილოთ ევკლიდეს ალგორითმის აგების ორივე ვარიანტი კონკრეტული ნატურალური რიცხვისთვის: 30 და 18.

ვარიანტი 1. ევკლიდეს ალგორითმის წარმოდგენა გაყოფის მეთოდით:

$$30 / 18 = 1 \text{ (ნაშთი 12)}$$

$$18 / 12 = 1 \text{ (ნაშთი 6)}$$

$$12 / 6 = 2 \text{ (ნაშთი 0)}$$

$$\text{საბოლოოდ: უსგ} = 6; \quad \text{პასუხი: უსგ} (30, 18) = 6$$

ვარიანტი 2. ევკლიდეს ალგორითმის წარმოდგენა გამოკლების მეთოდით:

$$30 - 18 = 12$$

$$18 - 12 = 6$$

$$12 - 6 = 6$$

$$6 - 6 = 0$$

$$\text{საბოლოოდ: უსგ} = 6; \quad \text{პასუხი: უსგ} (30, 18) = 6$$

მაგალითი 2.4. ერთ-ერთი ძველი ალგორითმი მსოფლიოს აჩუქა ბერძენმა მეცნიერმა **ერატოსთენემ**, რომელიც ძვ. წ. აღ. 276-194 წლებში მოღვაწეობდა. ერატოსთენემ შეიმუშავა მარტივი რიცხვების განსაზღვრის წესი. ერთისაგან განსხვავებულ მთელ, დადებით რიცხვს, რომლებიც უნაშთოდ იყოფა მხოლოდ ერთსა და თავის თავზე, მარტივი რიცხვი ეწოდება. მარტივი რიცხვებია: 2, 3, 5, 7, 11 და სხვა. მარტივ რიცხვთა ცხრილის შედგენის უძველეს ხერხს „ერატოსთენეს საცერი“ უწოდეს. ერატოსთენეს საცერის ალგორითმი საშუალებას გვაძლევს, ვიპოვოთ ყველა მარტივი რიცხვი $[1; n]$ მონაკვეთზე (ნახ. 2.5).

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

ნახ. 2.5. მარტივი რიცხვების ცხრილი 1-დან 100-მდე

მაგალითი 2.5. დავწეროთ ორი მთელი a, b რიცხვების ადგილების გაცვლის ალგორითმი. ამოცანა ამოვხსნათ მესამე c რიცხვის გამოყენებით: $c = a$; $a = b$; $b = c$;

იმავე ამოცანის ამოხსნა მესამე რიცხვის გარეშე შეიძლება:

$$a = a - b; \quad b = a + b; \quad a = b - a;$$

მაგალითად, მოცემულია ორი რიცხვი: $A = 7$; $B = 3$; შევუცვალოთ მათ ადგილები:

$$\text{პირველი ვარიანტი: } C = A; \quad A = B \text{ (} A = 3 \text{)}; \quad B = C \text{ (} B = 7 \text{)};$$

$$\text{მეორე ვარიანტი: } A = A + B; \quad B = A - B \text{ (} B = 7 \text{)}; \quad A = A - B \text{ (} A = 3 \text{)};$$

რეკურსიული და იტერაციული ალგორითმების მაგალითები

რეკურსია. რეკურსია არის აბსტრაქტული მოვლენა, რომელსაც შეგვიძლია დავაკვირდეთ ბუნების სამყაროში, მათემატიკაში და ა. შ. რეკურსია ისეთივე აბსტრაქტულია, როგორც მუსიკალური ჰარმონია. კომპიუტერის საშუალებით რაიმე ამოცანის გადასაწყვეტად, აუცილებელია გარკვეული ინსტრუქციების ჩამოყალიბება. კომპიუტერის მიერ ინსტრუქციების თანამიმდევრობით შესრულებას **პროცესი** ეწოდება, მაგალითად, მომხმარებლის მიერ გამოძახებული ინტერნეტ გვერდი არის პროცესი; ციკლი, რომელიც მონიტორის ეკრანზე 10-ჯერ გამოიტანს რიცხვ „77“-ს არის პროცესი.

რეკურსიის პროცესის აღწერა: რეკურსიულ პროცესს ამოცანა დაყავს რთული მდგომარეობიდან მარტივ მდგომარეობამდე, ასევე რეკურსიის დროს აუცილებელია მარტივი მდგომარეობის დასრულება/შეწყვეტა (ანუ ტერმინალური პირობის შესრულება).

რეკურსიული პროცესი რეკურსიის თითოეულ საფეხურზე გვეუბნება: „მე მახსოვს ეს“. რეკურსიულ პროცესს შეუძლია დასმული ამოცანა დაყოს რამდენიმე მსგავს და ანალოგიურ, მაგრამ უფრო მარტივ ამოცანად ანუ ამოცანა შეიძლება დაიშალოს მარტივ ამოცანებად. რეკურსიული პროცესი შეგვიძლია შევადაროთ იმ სტუდენტს, რომელიც ყველა საქმეს გადადებს პარასკევი დღის საღამოსათვის. მთელი კვირის განმავლობაში სტუდენტი თითქმის არაფერს აკეთებს, ხოლო პარასკევ საღამოს უამრავი საქმე უგროვდება, მაგრამ მას მოსწონს შესასრულებელი სამუშაოების ასეთი გადანაწილება. სიტყვა Recursio არის ლათინური სიტყვა და ნიშნავს „დაბრუნებას“. დაპროგრამებაში ფუნქციას, რომელიც თავის თავს იძახებს - რეკურსიული ფუნქცია ეწოდება. რეკურსიული ალგორითმი მუშაობის პროცესში საკუთარ თავს მიმართავს.

არსებობს ორი სახის რეკურსია:

- 1) პირდაპირი, როდესაც ფუნქცია იძახებს თავის თავს;
- 2) ირიბი, როცა A ფუნქცია იძახებს B-ს, ხოლო B ფუნქცია A-ს (იხ. ნახ. 2.6. ა,ბ).

რეკურსიის მაგალითებად შეიძლება განვიხილოთ შემდეგი ქართულ ხალხური ნაწარმოებები: რწყილი და ჭიანჭველა; თხა და ვენახი.



ა)



ბ)

ნახ. 2.6. ა) პირდაპირი რეკურსია; ბ) ირიბი რეკურსია;

დავწეროთ 1-დან 5-ის ჩათვლით ნატურალური რიცხვების ჯამის ალგორითმი რეკურსიის გამოყენებით:

$$\begin{array}{l}
 SUM(5) = 5 + 4 + 3 + 2 + 1 = 5 + SUM(4) \\
 SUM(4) = 4 + 3 + 2 + 1 = 4 + SUM(3) \\
 SUM(3) = 3 + 2 + 1 = 3 + SUM(2) \\
 SUM(2) = 2 + 1 = 2 + SUM(1) \\
 SUM(1) = 1
 \end{array}
 \left. \vphantom{\begin{array}{l} SUM(5) \\ SUM(4) \\ SUM(3) \\ SUM(2) \\ SUM(1) \end{array}} \right\} SUM(X) = X + SUM(X - 1)$$

← ტერმინალური პირობა

თუ $X = 1$ მაშინ $SUM = 1$, სხვა დანარჩენ შემთხვევაში $SUM = X + SUM(X - 1)$

იტერაცია

იტერაცია (ლათინურად Iteratio) - არის რაიმე მოქმედების გამეორება.

იტერაციული პროცესი არის მონაცემთა დამუშავების ისეთი პროცესი, რომლის დროსაც გარკვეული მოქმედება მრავალჯერ მეორდება და ამასთანავე არ ხდება საკუთარი თავის გამოძახება (ანუ რეკურსია).

იტერაციული პროცესი მუდმივად ამბობს: „ახლა გამოვითვლი ყველაფერს, რაც შესაძლებელია და გავაგრძელებ“.

იტერაცია ყოველთვის ყველაფერს ითვლის პირველივე შესაძლო მომენტში.

იტერაციული პროცესი შეგვიძლია შევადაროთ იმ სტუდენტს, რომელიც ყველაფერს აკეთებს პირველივე შესაძლო შემთხვევაში.

სტუდენტის სამუშაო კვირა საქმიანობის მხრივ თანაბრად გადანაწილებულია, პარასკევს ჩვეულებრივი ბოლო სამუშაო დღეა.

იტერაციული პროცესის აღწერა: იტერაციულ პროცესს ამოცანა დაჰყავს მარტივი მდგომარეობიდან რთულ მდგომარეობამდე.

დაპროგრამებაში იტერაციის განსახორციელებლად გამოიყენება ციკლი.

დავწეროთ 1-დან 5-ის ჩათვლით ნატურალური რიცხვების ჯამის გამოთვლის ალგორითმი იტერაციის გამოყენებით:

$SUM = 0$ $SUM \quad \underbrace{\hspace{1cm}}$ $SUM = 0 + 1$ $SUM \quad \underbrace{\hspace{1cm}}$	$SUM = SUM + 1$
$SUM = 0 + 1 + 2$ $SUM \quad \underbrace{\hspace{1.5cm}}$	$SUM = SUM + 2$
$SUM = 0 + 1 + 2 + 3$ $SUM \quad \underbrace{\hspace{2cm}}$	$SUM = SUM + 3$
$SUM = 0 + 1 + 2 + 3 + 4$ $SUM \quad \underbrace{\hspace{2.5cm}}$	$SUM = SUM + 4$
$SUM = 0 + 1 + 2 + 3 + 4 + 5$	$SUM = SUM + 5$

უნდა შევნიშნოთ, რომ თუ ამოცანა იხსნება იტერაციული ალგორითმით, მაშინ მისი ამოხსნა შესაძლებელია რეკურსიული ალგორითმითაც და პირიქით. ორივე ალგორითმი გულისხმობს, რომ თავიდან მოცემულია საწყისი ამოცანის საბაზო შემთხვევა, რომლის ამონახსნიც ვიცით და გამოსაკვლევი შემთხვევა. პრაქტიკული თვალსაზრისით, ამოცანის რეკურსიული მეთოდით გადაწყვეტა უფრო ნელია, ვიდრე იტერაციულით, მაგრამ რეკურსია უფრო კითხვადია, ვიდრე იტერაცია.

რეკურსიით და იტერაციით გადაწყვეტის კლასიკური მაგალითები.

მაგალითი 2.5. ფაქტორიალი. მთელი დადებითი ნატურალური n რიცხვის ფაქტორიალი მიიღება ერთიდან n -მდე მთელი რიცხვების გამამრავლებით და გამოისახება $n!$ სახით.

რიცხვის ფაქტორიალი შეგვიძლია გამოვითვალოთ რეკურსიული ფორმულით (1):

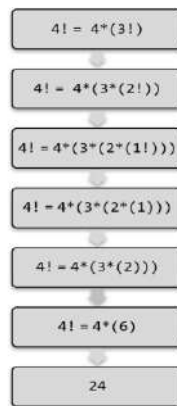
$$n! = \begin{cases} 1 & \text{თუ } n = 1 \\ n * (n - 1) * \dots * 1 & \text{თუ } n > 1 \end{cases} \quad (1)$$

ნატურალური რიცხვის ფაქტორიალი უდრის 1-ს, როცა $n = 0$ და უდრის $1 * 2 * 3 * \dots * (n - 1) * n$, როცა n მეტია ერთზე. აქვე უნდა შევნიშნოთ, რომ $0! = 1$;

მაგალითად, $f(4)$ შეგვიძლია გამოვითვალოთ შემდეგი სახით:

$$f(4) = 4 * f(3) = 4 * 3 * f(2) = 4 * 3 * 2 * f(1) = 4 * 3 * 2 * 1 * f(0) = 4 * 3 * 2 * 1 * 1 = 24$$

$4!$ ფაქტორიალის გამოსათვლელად დაგჭირდა n ცალი რეკურსიული ფუნქციის გამოძახება. ფაქტორიალის გამოთლის გრაფიკული ფორმა (ნახ. 2.7):



ნახ. 2.7. ფაქტორიალის ვიზუალიზაცია

მაგალითი 2.6. ფიბონაჩის მიმდევრობა. სამყაროში, მიუხედავად მისი ქაოსური და მრავალფეროვანი ბუნებისა, ყველაფერი გარკვეული კანონზომიერებითაა აგებული. თუ შევეცდებით ყველაფერი ციფრებისა და რიცხვების სისტემაში გადავიყვანოთ, სამყაროს გენიალურობა დადის ერთ მარტივ 1.618 რიცხვამდე. არსებობს ციფრებისა და რიცხვების ისეთი მიმდევრობა, რომელშიც მიმდევრობის ყოველი მომდევნო წევრი, წინა ორი წევრის ჯამს წარმოადგენს. ასეთ მიმდევრობას **ფიბონაჩის** მიმდევრობა ეწოდება: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 და ა.შ. უსასრულობამდე.

ფიბონაჩის მიმდევრობაში ყოველი მომდევნო მნიშვნელობა მყარადაა დაკავშირებული მის წინა მნიშვნელობასთან, ვიდრე ჩვეულებრივ მიმდევრობებში. ფიბონაჩის მიმდევრობის ყოველი რიცხვის ფარდობა მის მომდევნო რიცხვთან (მაგ., 55/89) დიდი მიახლოებით 0.618-ის ტოლია, ხოლო ყოველი რიცხვის ფარდობა მის წინა რიცხვთან დიდი მიახლოებით ტოლია 1.618-ის (0.618-ის ინვერსია). რაც უფრო იზრდება რიცხვები, მით უფრო ზუსტად უახლოვდება ფარდობა 1.61803399...-ს რიცხვს.

ფიბონაჩის რიცხვებს ხშირად ბუნების ნუმერაციის სისტემასაც უწოდებენ. ფიბონაჩი იყო შუა საუკუნეების უდიდესი მათემატიკოსი. იგი დაიბადა 1170 წელს და გარდაიცვალა 1240 წელს. მან დაგვიტოვა საკმაოდ ღრმა და საფუძვლიანი ცნობები „ოქროს კვეთასა“ და ღვთიურ მიმდევრობაზე. ფიბონაჩის რიცხვი განისაზღვრება შემდეგი ფორმულით (2):

$$F_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F_{n-1} + F_{n-2}, & n > 1 \end{cases} \quad (2)$$

$$F_0 = 0$$

$$F_1 = 1$$

$$F_2 = F_1 + F_0 = 1 + 0 = 1$$

$$F_3 = F_2 + F_1 = 1 + 1 = 2$$

$$F_4 = F_3 + F_2 = 2 + 1 = 3$$

$$F_5 = F_4 + F_3 = 3 + 2 = 5$$

$$F_6 = F_5 + F_4 = 5 + 3 = 8$$

შესაბამისად საკმაოდ მარტივია დაწეროთ რეკურსიული ფუნქცია ფიბონაჩის n -ური რიცხვის გამოსათვლელად. ფიბონაჩის n -ური რიცხვის პოვნა რეკურსიულად:

1. თუ $n = 1$ ან $n = 2$ დავაბრუნოთ მნიშვნელობა 1, რადგან $F_1 = 1$ და $F_2 = 1$;
2. დანარჩენ შემთხვევაში ფუნქცია უნდა გამოვიძახოთ $n - 1$ და $n - 2$ არგუმენტებისთვის.
3. ორი გამოძახების მნიშვნელობა შევკრიბოთ და დავაბრუნოთ მიღებული შედეგი.

ფიბონაჩის მიმდევრობის განსაზღვრიდან გამომდინარე, შეიძლება დაწეროთ მარტივი რეკურსიული ფუნქცია, რომელიც იპოვს ფიბონაჩის n -ურ რიცხვს. ფიბონაჩის რიცხვის მიღების ფუნქცია:

```
int ფიბონაჩის_რიცხვი(int n) {
```

```
    if (n == 0) return 0;
```

```
    if (n == 1) return 1;
```

```
    დავაბრუნოთ ფიბონაჩის_რიცხვი(n - 2) + ფიბონაჩის_რიცხვი(n - 1); }
```

თუ $n = 4$, მაშინ მოხდება ფიბონაჩის_რიცხვი(3) და ფიბონაჩის_რიცხვი(2) რეკურსიული გამოძახება.

ეს საინტერესოა: მცენარეების, ცხოველების და ადამიანების აგებულებაში იმდენად ხშირად ვხვდებით 1.618-ს, რომ ანტიკური დროის მეცნიერებმა მას „ღვთაებრივი პროპორცია“ უწოდეს. ფუტკრების სკაში მდედრების და მამრების რაოდენობა ყოველთვის დაბალანსებულია, მდედრი ფუტკრები რაოდენობრივად სჭარბობენ მამრებს, მაგრამ თუ მდედრების რაოდენობას გავყოფთ მამრი ფუტკრების რაოდენობაზე, ყოველთვის, მსოფლიოს ნებისმიერი სკისათვის, მივიღებთ 1.618-ს. ნაუტილუსის ნიჟარაში ყველა სპირალის დიამეტრის შეფარდება მომდევნოსთან უდრის 1.618-ს. მზესუმზირის მარცვლები სპირალურადაა განლაგებული საათის ისრის საწინააღმდეგოდ. ყოველი წრის დიამეტრის შეფარდება მომდევნოსთან უდრის 1.618-ს და ეს ასეა ფოთლების განლაგებაზე მცენარის ღეროზე, ფიჭვის გირჩის ქერცლების განლაგება, მწერების სხეულის ნაწილების განლაგება და ა.შ. <https://mathtasks.wordpress.com/tag/%E1%83%A4%E1%83%98%E1%83%91%E1%83%9D%E1%83%9C%E1%83%90%E1%83%A9%E1%83%98/>

მაგალითი 2.7. მოცემულია გვაქვს n ცალი ნივთი. გვინტერესებს რამდენი განსხვავებული გზით შეგვიძლია დავალაგოთ n ცალი ნივთი? პირველი ნივთის ასარჩევად გვაქვს n ცალი ვარიანტი. ამ n არჩევანიდან მეორე ნივთის ასარჩევად გვრჩება $n - 1$ ვარიანტი. პირველი ორი ნივთის ასარჩევად გვაქვს $n * (n - 1)$ ვარიანტი. ახლა, ამ ორი არჩევანიდან თითოეული ნივთისათვის გვაქვს $n - 2$ ვარიანტი, ამდენად პირველი სამი ნივთის ასარჩევად გვაქვს $n * (n - 1) * (n - 2)$ ვარიანტი. ასე ვაგრძელებთ მანამ, სანამ არ ჩამოვალთ ბოლო ორ ნივთზე და შემდეგ მხოლოდ ერთ ნივთზე. გამოდის, რომ n ცალი ნივთის დასალაგებლად გვაქვს $n * (n - 1) * (n - 2) \dots 2 * 1$ გზა. ეს ნამრავლი კი არის $n!$ (n -ის ფაქტორიალი), უბრალოდ 1-იდან დაწყებისა და n -მდე ასვლის ნაცვლად ნამრავლი იწყება n -იდან და ჩამოდის 1-მდე.

მაგალითი 2.8. ეხლა კი განვიხილოთ რეკურსია თოჯინა „მატრიოშკის“ მაგალითზე (იხ. ნახ. 2.8.). ვიცით, რომ რეკურსია არის ფუნქცია, რომელიც იძახებს თავის თავს. აქედან გამომდინარე, თოჯინა „მატრიოშკა“ არის:

თოჯინა, რომელშიც განთავსებულია:

თოჯინა, რომელშიც განთავსებულია:

თოჯინა, რომელშიც განთავსებულია:

თოჯინა, რომელშიც განთავსებულია...



ნახ. 2.8. რეკურსია სათამაშოს მაგალითზე

რა საერთო აქვს თოჯინა „მატრიოშკებს“ და ალგორითმებს? როგორც ვხედავთ, ერთ თოჯინაში არის განთავსებული მეორე, უფრო პატარა თოჯინა, მეორე თოჯინა შეიცავს კიდევ უფრო პატარა თოჯინას და ასე გრძელდება იმ პაწაწინა თოჯინამდე, რომელშიც უკვე ვეღარ მოვათავსებთ უფრო პატარა თოჯინას. შესაბამისად, შეგვიძლია დავწეროთ ალგორითმი, რომელიც რომელიმე ამოცანას ხსნის იმავე ამოცანის უფრო პატარა შემთხვევის ამოხსნით, გარდა იმ შემთხვევისა, როცა ამოცანა იმდენად პატარაა, რომ მისი ამოხსნა პირდაპირ შეგვიძლია.

ფორმალური მტკიცების მეთოდები. მათემატიკური ინდუქცია

მათემატიკაში დამტკიცება არის დედუქციური მსჯელობა, რომელიც გამოიყენება გამონათქვამის დასაბუთებისათვის. ამგვარი მსჯელობის დროს შესაძლებელია უკვე არსებული, ადრე დასაბუთებული დებულებების გამოყენება. სურვილის შემთხვევაში, შესაძლებელია რომ დამტკიცებისას გამოყენებულ მსჯელობას მიეყვებოდეთ უკუსვლით იქამდე სანამ არ მივაღებთ ისეთ საწყის დებულებებს, რომელთა ჭეშმარიტება თავისთავად ცხადია ან რომლებიც მიჩნეულია როგორც ჭეშმარიტი დებულებები. ისეთ საწყის დებულებებს კი, რომლებიც დამტკიცების გარეშე მიღებული როგორც ჭეშმარიტი დებულებები, აქსიომები ეწოდება. მათემატიკაში დამტკიცებები მკაცრად ემორჩილება ფორმალური ლოგიკის კანონებს, მაგრამ, როგორც წესი ჩამოყალიბებულია ჩვეულებრივი სასაუბრო ენის წინადადებების საშუალებით. წმინდა ფორმალური დამტკიცებები, რომლებიც იყენებს სიმბოლურ ენას სასაუბრო ენის ნაცვლად, განიხილება მათემატიკის, ფილოსოფიური ლოგიკისა და კომპიუტერული მეცნიერების ერთ-ერთ მიმართულებაში, რომელსაც დამტკიცების თეორია ეწოდება. ამ თეორიის ერთ-ერთი მიზანია ისეთი ინტერაქტიული კომპიუტერული სისტემების შექმნა, რომელთა დახმარებითაც შესაძლებელია დამტკიცების ანალიზი, მაგალითად, ამ დამტკიცების სისწორის შემოწმების მიზნით ან რომელიმე დებულების დამტკიცების მოძებნა. როგორც წესი, არსებული დამტკიცების სისწორის შემოწმება უფრო მარტივია, ვიდრე ახალი დამტკიცების მოძებნა. ეს უკანასკნელი, ფორმალური ლოგიკის გარდა ხშირად მოითხოვს ინტუიციას და შემოქმედებითობას. ამიტომ, შეიძლება ითქვას, რომ მათემატიკური დამტკიცება თავის თავში აერთიანებს როგორც ანალიზს, ისე სინთეზს.

ინდუქცია არის მსჯელობის მეთოდი, როდესაც კერძო მაგალითებიდან ხდება ზოგადი დასკვნის გამოტანა. მათემატიკური ინდუქცია არის დამტკიცების ისეთი მეთოდი, რომლის საშუალებითაც შესაძლებელია განსაზღვრული დებულების სამართლიანობის დასაბუთება ნებისმიერი ნატურალური რიცხვისთვის. იმის ნაცვლად რომ გამონათქვამი დამტკიცდეს ყოველი ნატურალური რიცხვისათვის ცალკე (რაც პრინციპში შეუძლებელია, რადგან ნატურალური რიცხვების სიმრავლე უსასრულოა), იგი მტკიცდება ერთი საბაზისო შემთხვევისათვის, რომელიც დაკავშირებულია ნატურალური პარამეტრის საწყის მნიშვნელობასთან და შემდეგ მტკიცდება ინდუქციის წესი.

ინდუქციის წესის დამტკიცება ნიშნავს იმის დამტკიცებას, რომ თუ გამონათქვამი ჭეშმარიტია ნატურალური პარამეტრის რომელიმე მნიშვნელობისათვის, მაშინ იგი ჭეშმარიტია ამ პარამეტრის მომდევნო მნიშვნელობისთვისაც. ჩამოყალიბოთ მათემატიკური ინდუქციის პრინციპი, რომელზედაც დამყარებულია მტკიცებათა მეთოდი, რომელსაც მათემატიკური ინდუქციის მეთოდი ეწოდება.

თუ მოცემულია $A(n)$ გამონათქვამთა მიმდევრობა (n ნატურალური რიცხვია), რომელიც ჭეშმარიტია $n = 1$ -სთვის და იქიდან, რომ იგი ჭეშმარიტია $n = k$ -სთვის გამომდინარეობს მისი ჭეშმარიტება მომდევნო $n = k + 1$ -სთვის, მაშინ $A(n)$ დებულება ჭეშმარიტი იქნება ნებისმიერი ნატურალური n -სთვის. მათემატიკური ინდუქციის მეთოდით დამტკიცება ორი საფეხურისაგან შედგება:

პირველი საფეხური: ამტკიცებენ (ამოწმებენ) $A(1)$ -ის ჭეშმარიტებას;

მეორე საფეხური: ვარაუდობენ, რომ $A(n)$ ჭეშმარიტია $n = k$ -სთვის და აქედან გამომდინარე ამტკიცებენ მის ჭეშმარიტებას $n = k + 1$ -სთვის ე. ი. ამტკიცებენ, რომ $A(k) = > A(k + 1)$ ჭეშმარიტია.

თუ დამტკიცების ორივე ნაწილი ჩატარებულია, მაშინ მათემატიკური ინდუქციის პრინციპის საფუძველზე $A(n)$ წინადადება ჭეშმარიტი იქნება ნებისმიერი ნატურალური n რიცხვისათვის. პირველ საფეხურს ინდუქციის ბაზისი ეწოდება, მეორე საფეხურს კი ინდუქციური ბიჯი. მტკიცების პირველი საფეხური ქმნის ბაზას ინდუქციისათვის. მეორე საფეხური კი იძლევა საშუალებას შეუზღუდავად, ავტომატურად გავაფართოვოთ ბაზა და გადავიდეთ მოცემული კერძო შემთხვევიდან მის შემდეგ შემთხვევაზე: n -დან $n + 1$ -ზე.

მაგალითი 2.9. დავამტკიცოთ ფორმულის ტოლობა: $1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$

დასმული ამოცანა ამოვხსნათ მათემატიკური ინდუქციის საშუალებით:

შევამოწმოთ $n = 1$ შემთხვევა;

$n = 1$ -სთვის $\frac{n(n+1)(2n+1)}{6} = \frac{1 \cdot 2 \cdot 3}{6} = 1$ ე. ი. $n = 1$ -სთვის ფორმულა ჭეშმარიტია;

დავუშვათ, რომ ფორმულა ჭეშმარიტია $n = k$ შემთხვევისათვის;

თუ $n = k$ -სთვის ფორმულა $1^2 + 2^2 + \dots + k^2 = \frac{k(k+1)(2k+1)}{6}$ ჭეშმარიტია, მაშინ შეგვიძლია დავამტკიცოთ, რომ იგი ჭეშმარიტია $n = k + 1$ შემთხვევისთვისაც:

$$\begin{aligned} 1^2 + 2^2 + \dots + k^2 &= \frac{k(k+1)(2k+1)}{6} + (k+1)^2 = \frac{k(k+1)(2k+1) + 6(k+1)^2}{6} \\ &= \frac{(k+1)(2k^2 + 7k + 6)}{6} = \frac{(k+1)(k+2)(2k+3)}{6} \\ &= \frac{(k+1)((k+1)+1)(2(k+1)+1)}{6} \end{aligned}$$

დასკვნა: რადგან მოცემული ფორმულა ჭეშმარიტია $n = k$ -სთვის, დავამტკიცეთ, რომ იგი ჭეშმარიტია $n = k + 1$ -სთვის, აქედან გამომდინარე, შეგვიძლია ვთქვათ, რომ ფორმულა ჭეშმარიტია ნებისმიერი n -სთვის.

მაგალითი 2.10. დავამტკიცოთ, რომ გამოსახულება $5^n - 4n + 15$, იყოფა 16-ზე ნებისმიერი $n \in N_0$ მნიშვნელობებისთვის. დასმული ამოცანა ამოვხსნათ მათემატიკური ინდუქციის საშუალებით. შევამოწმოთ:

$n = 0$ შემთხვევა: $5^0 - 4 \cdot 0 + 15 = 16$ იყოფა 16-ზე;

$n = 1$ შემთხვევა: $5^1 - 4 \cdot 1 + 15 = 16$ იყოფა 16-ზე;

$n = 2$ შემთხვევა: $5^2 - 4 \cdot 2 + 15 = 32$ იყოფა 16-ზე;

დავუშვათ, რომ მტკიცება ჭეშმარიტია (სწორია) $n = k$ -სთვის ე.ი. გამოსახულება $5^k - 4k + 15$ იყოფა 16-ზე.

ახლა შევამოწმოთ $n = k + 1$ შემთხვევა. $n = k + 1$ შემთხვევის დროს გამოსახულება მიიღებს შემდეგ სახეს:

$$\begin{aligned} 5^{k+1} - 4(k+1) + 15 &= 5 \cdot 5^k - 4k + 11 = 5 \cdot 5^k - 4k \cdot 5 + 15 \cdot 5 + 4k \cdot 5 - 15 \cdot 5 - 4k + 11 \\ &= 5(5^k - 4k + 15) + 20k - 75 - 4k + 11 = 5(5^k - 4k + 15) + 16k - 64 = \\ &= 5(5^k - 4k + 15) + 16(k - 4) \end{aligned}$$

რადგან $5^k - 4k + 15$ იყოფა 16-ზე, მაშინ $5(5^k - 4k + 15)$ -იც იყოფა 16-ზე. რადგან $5(5^k - 4k + 15)$ იყოფა 16-ზე და $16(k - 4)$ იყოფა 16-ზე, მაშინ $5(5^k - 4k + 15) + 16(k - 4)$ გამოსახულებაც იყოფა 16-ზე ე. ი. თუ $5^k - 4k + 15$ იყოფა 16-ზე, მაშინ $5^{k+1} - 4(k + 1) + 15$ გამოსახულებაც იყოფა 16-ზე.

დასკვნა: გამოსახულება ჭეშმარიტია (სწორია) $n = 0, 1, 2$ მნიშვნელობისთვის. თუ გამოსახულება სწორია $n = k$ -სთვის, მაშინ იგი ასევე სწორია $n = k + 1$ -სთვის, აქედან გამომდინარე, შეგვიძლია ვთქვათ, რომ გამოსახულება ჭეშმარიტია ნებისმიერი მთელი დადებითი n -სთვის, რისი დამტკიცებაც გვინდოდა.

დიაგონალიზაციისა და დირიხლეს პრინციპი

მათემატიკა ერთ-ერთი ყველაზე საინტერესო საბუნებისმეტყველო მეცნიერებაა, მაგრამ ადამიანების უდიდეს ნაწილს არ შეუძლიათ ამ სფეროში მარტივი საკითხების გააზრებაც კი, რომ აღარაფერი ვთქვათ მეცნიერული აღმოჩენებზე, თუმცა არიან ისეთი ადამიანები, რომელთა ღვაწლი და აღმოჩენები საბუნებისმეტყველო სფეროში ფასდაუდებელია. ასეთ ადამიანთა შორის არის გამოჩენილი გერმანელი მათემატიკოსი და მეცნიერი იოჰან პეტერ გუსტავ ლეჟენ დირიხლე, რომელმაც მნიშვნელოვანი წვლილი შეიტანა მათემატიკის განვითარებაში.

გერმანია არის მათემატიკოსების სამშობლო, გერმანელმა მეცნიერებმა მრავალი სამეცნიერო აღმოჩენები და მიღწევები აჩუქეს მსოფლიოს. 1805 წლის 13 თებერვალს, გერმანიის პატარა ქალაქ დურენში დაიბადა იოჰან პეტერ გუსტავ ლეჟონ დირიხლე, რომელსაც მოგვიანებით მათემატიკის მეფესაც კი უწოდებდნენ (ნახ. 2.9.).

დირიხლეს სახელს უკავშირდება რამდენიმე მნიშვნელოვანი აღმოჩენა მათემატიკაში, მექანიკასა და ფიზიკაში. რიცხვთა თეორიაში დირიხლემ დაამტკიცა თეორემა არითმეტიკული პროგრესიის შესახებ.



ნახ. 2.9. იოჰან პეტერ გუსტავ ლეჟენ დირიხლე (1805-1859).

გერმანელი მათემატიკოსი, რომელმაც მნიშვნელოვანი წვლილი შეიტანა მათემატიკური ანალიზის, ფუნქციების თეორიისა და რიცხვთა თეორიის განვითარებაში.

დირიხლეს თეორემა: ყოველი არითმეტიკული პროგრესია, რომელშიც პირველი წევრი და სხვაობა ურთიერთმარტივი ნატურალური რიცხვებია, შეიცავს მარტივი რიცხვების უსასრულო რაოდენობას. ტრადიციულად დირიხლეს პრინციპის ახსნას ახდენენ - „კურდღელი და გალია“ - მაგალითის მიხედვით. დირიხლეს პრინციპის გამოყენებით კონკრეტულ ამოცანის გადაწყვეტისას, საკმაოდ რთულია იმის გარკვევა, თუ რომელია „კურდღელი“ და რომელი „გალია“. დირიხლეს პრინციპით, ამოცანის გადაწყვეტისას ეს არის სწორედ ყველაზე რთული პროცესი.

განვიხილოთ დირიხლეს პრინციპის ფორმულირებები:

დირიხლეს პრინციპის პირველი ფორმულირება (მტკიცებულება): თუ n ცალ გალიაში ზის $m = n + 1$ კურდღელი ან უფრო მეტი, მაშინ მოიძებნება ისეთი გალია, რომელშიც სხედან სულ მცირე ორი კურდღელი მაინც.

მაგალითად, გვაქვს n გალია და გვყავს m კურდღელი, თან კურდღელების რაოდენობა უფრო მეტია, ვიდრე გალიების ($m > n$). თუ 4 (ანუ n) გალიაში ზის 5 (ანუ $m = n + 1$) კურდღელი, მაშინ რომელიმე ერთ გალიაში მაინც ზის ერთზე მეტი კურდღელი, ჩვენს შემთხვევაში ორი კურდღელი (ნახ. 2.10)

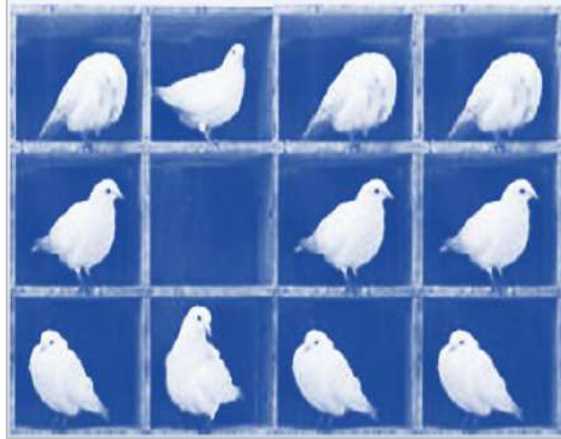


ნახ. 2.10. დირიხლეს პრინციპი „კურდღელი-გალია“. პირველი ფორმულირების (მტკიცებულების) ნიმუში.

თუ n ცალ გალიაში ზის m ცალი მტრედი და ამასთანავე $m < n$ -ზე, მაშინ ერთი გალია მაინც აღმოჩნდება ცარიელი.

მაგალითად, თუ 12 (ანუ n) გალიაში ზის 11 (ანუ $n - 1$) მტრედი, მაშინ რომელიმე ერთ გალია მაინც იქნება ცარიელი/თავისუფალი (იხ. ნახ. 2.11).

დირიხლეს პრინციპის მეორე ფორმულირება (მტკიცებულება): დავუშვათ, რომ m ცალი კურდღელი ზის n რაოდენობის გალიაში. თუ $m > n$, მაშინ რომელიმე ერთ გალიაში იმყოფება არანაკლებ m/n კურდღელი, ამასთანავე რომელიმე სხვა ერთ ცალ გალიაში იმყოფება არაუმეტეს m/n კურდღელი (იხ. ნახ. 2.12.).



ნახ. 2.11. დირიხლეს პრინციპი „მტრედი-გალია“.

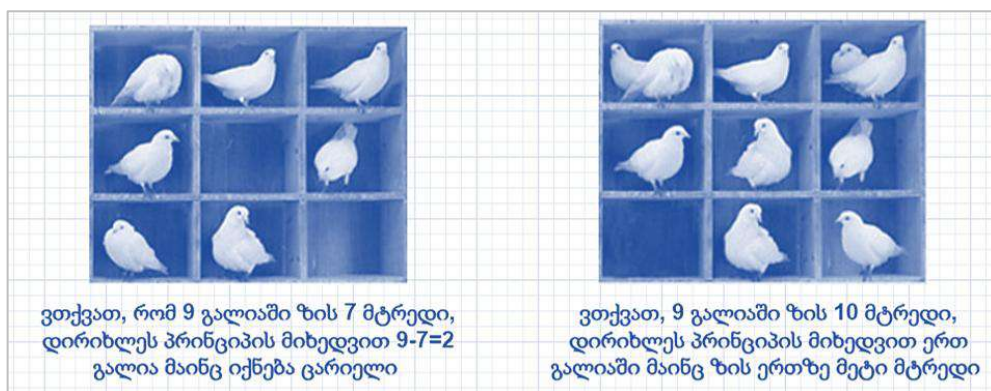
პირველი ფორმულირების (მტკიცებულების) ნიმუში.



ნახ. 2.12. დირიხლეს პრინციპი.

მეორე ფორმულირების (მტკიცებულების) ნიმუში.

დირიხლეს პრინციპის გამოყენების მაგალითი (ნახ. 2.13.)



ნახ. 2.13. დირიხლეს პრინციპის გამოყენების ნიმუში

დირიხლეს პრინციპის გამოყენების ალგორითმი

დასმულ ამოცანაში უნდა განისაზღვროს თუ, რომელი ობიექტია კურდღელი და რომელი, გალია; გამოვიყენოთ დირიხლეს პრინციპთან შესაბამისობაში მყოფი ფორმულირება (მტკიცებულება):

- თუ n გალიაში ზის არა უმეტეს $n - 1$ კურდღელი, მაშინ არსებობს ცარიელი გალია;
- თუ n გალიაში ზის არა უმეტეს $n + 1$ კურდღელი, მაშინ არსებობს ისეთი გალია, რომელშიც ზის არა უმეტეს $2 - x$ კურდღელი;
- თუ n გალიაში ზის არა უმეტეს $nk - 1$ კურდღელი, მაშინ რომელიმე გალიაში ზის არა უმეტეს $k - 1$ კურდღელი;
- თუ n გალიაში ზის არა უმეტეს $nk + 1$ კურდღელი, მაშინ რომელიმე გალიაში ზის არა ნაკლებ $k + 1$ კურდღელი;

ადვილი მისახვედრია, რომ კურდღელი-გალიასთან მიმართებით შეიძლება გამოვიყენოთ არა მარტო მტრედი-გალია კომბინაცია, არამედ ნებისმიერი ობიექტი, რომელიც მათემატიკაში ცნობილია სიმრავლის სახელით: თუ სიმრავლე A შეიცავს $N + 1$ ელემენტს, მაშინ არსებობს ისეთი N რაოდენობის ელემენტები (რომლებიც აკმაყოფილებენ ნებისმიერ განსხვავებულ თვისებებს), რომ ამ ელემენტებიდან ორ ელემენტს მაინც აქვს ერთი და იგივე თვისება. დირიხლეს პრინციპის დამტკიცება:

ვთქვათ, გვაქვს N ცალი გალია და გვყავს $N + 1$ რაოდენობის კურდღელი. დავუშვათ, რომ თითოეულ გალიაში მოთავსებულია არა უმეტეს ერთი კურდღელისა:

პირველ გალიაში - ერთი მაინც კურდღელი,

მეორე გალიაში - ერთი მაინც კურდღელი,

.....

ასევე N -ურ გალიაში - ერთი მაინც კურდღელი;

ჩვენ მივიღეთ N ცალი წინააღმდეგობა, რომელიც უნდა შევკრიბოთ:

$$n_1 + n_2 + n_3 + \dots + n_4 \leq 1 + 1 + 1 + \dots + 1$$

$$n_1 + n_2 + n_3 + \dots + n_4 \leq N$$

$N + 1 \leq N$ - წინააღმდეგობა; უტოლობის მარცხნივ გვაქვს კურდღლების რაოდენობა, მარჯვნივ კი გალიების რაოდენობა.

საბოლოოდ, მივედით წინააღმდეგობასთან: კურდღლების რაოდენობა უფრო ნაკლებია ვიდრე გალიების რაოდენობა, თუმცა პირობაში მოცემულია განსხვავებული მტკიცება.

მაგალითი 2.11. დავუშვათ, რომ საკონტროლო სამუშაო შეასრულა 30 სტუდენტმა. სტუდენტმა *აბრაკადაბრამ* დაუშვა ყველაზე მეტი, 13 შეცდომა. დაამტკიცეთ, რომ მინიმუმ სამმა სტუდენტმა მაინც დაუშვა იმავე რაოდენობის შეცდომა.

დასმულ ამოცანაში მნიშვნელოვანია გავარკვიოთ, თუ რომელია „გალია“ და რომელია „კურდღელი“. მოცემულ ამოცანაში „კურდღლის“ როლში დავადგინოთ სტუდენტი, ხოლო „გალიის“ როლში კი დაშვებული შეცდომები.

თუ პირველ გალიაში მოვათავსებთ იმ სტუდენტებს, რომლებმაც არც ერთი შეცდომა არ დაუშვეს, მეორე გალიაში კი იმ სტუდენტებს, რომლებმაც ორი და ა.შ., ხოლო მე-13

გალიაში მოვათავსებთ სტუდენტს აბრაკადაბრას, მაშინ სტუდენტების ასეთი განთავსების შემდეგ შეგვიძლია გამოვიყენოთ წინააღმდეგობის მეთოდი:

ვთქვათ, სტუდენტებს შორის არ აღმოჩნდნენ ისეთები, რომლებმაც დაუშვეს შეცდომების ერთი და იგივე რაოდენობა, მაშინ თითოეულ გალიაში უნდა განთავსდეს მაქსიმუმ ორი სტუდენტი. რადგან გვაქვს 14 გალია (ბოლო გალიაში არის მარტო სტუდენტი აბრაკადაბრა), სტუდენტების ჯამური რაოდენობა ვერ გადააჭარბებს $13 * 2 + 1 = 27$ სტუდენტს.

ჩვენ მივედით წინააღმდეგობასთან, რადგან საკონტროლო დაწერა 30 სტუდენტმა. ამოცანა ამოხსნილია.

ალგორითმების შეფასება. ალგორითმის ბიჯების რაოდენობის დათვლა, სისწორის მტკიცების მეთოდები. გამოთვლების სისწრაფე. ფუნქციათა ზრდის სისწრაფე და მისი გამოყენება შეფასებისას. ალგორითმის მუშაობის მინიმალური, მაქსიმალური და საშუალო დრო. ალგორითმის ბიჯების რაოდენობის ზედა და ქვედა ზღვრის შეფასების მეთოდები

ალგორითმის ბიჯების რაოდენობის დათვლა და სისწორის მტკიცების მეთოდები

ალგორითმის სწრაფი და ოპტიმალური მუშაობის მიხედვით შეგვიძლია დავადგინოთ ალგორითმის მუშაობის ეფექტურობა. ალგორითმის შესაფასებლად და მისი მუშაობის ოპტიმალური მოდელის ასრჩევად არსებობს შემდეგი მარტივი ორი მეთოდი:

1. ალგორითმის შესრულებისთვის საჭირო დროის შეფასება. უნდა დავადგინოთ ალგორითმის შესრულების დრო ანუ რა დროს ანდომებს ალგორითმი მუშაობის დაწყებიდან დასრულებამდე. სასურველია დავწეროთ ან ავირჩიოთ ისეთი ალგორითმები, რომლებიც მუშაობენ სწრაფად, განსაკუთრებით ეს საჭიროა მაშინ, როდესაც საქმე გვაქვს მონაცემების დიდ რაოდენობასთან.

2. ალგორითმის მიერ რესურსების გამოყენების შეფასება. შესრულების დროის გარდა, ასევე უნდა შევაფასოთ თუ რამდენ მეხსიერებას ან სხვა რესურსს მოითხოვს ალგორითმი. კარგმა ალგორითმმა რესურსები ეფექტურად უნდა გამოიყენოს ისე, რომ არ დაიკავოს მეხსიერების ზედმეტი ადგილი და არ დახარჯოს დამატებითი კომპიუტერული რესურსები.

ალგორითმის სირთულის შეფასება. ალგორითმის სირთულე გვიჩვენებს, თუ რამდენად სწრაფად იცვლება მისი შესრულების დრო ან რესურსების დანახარჯი, შეყვანილი მონაცემების ზომის გაზრდასთან ერთად. ზოგიერთი ალგორითმი შეიძლება იყოს ეფექტური მცირე ზომის მონაცემებისთვის, მაგრამ იგივე ალგორითმი გახდება ნელი ან დაიწყებს უამრავი რესურსის გამოყენებას, როგორც კი მოხდება მონაცემების რაოდენობის გაზრდა. ალგორითმი დასმული ამოცანის გადაჭრის გზაა, მაგრამ ამოცანის გადასაჭრელად უნდა გავითვალისწინოთ შემდეგი სამი პუნქტი:

1. ალგორითმი უნდა შედგებოდეს ერთი ან რამდენიმე ბიჯისაგან;
2. ალგორითმი, ერთი ბიჯის დასრულების შემდეგ, მომდევნო ბიჯის შესრულებაზე უნდა გადავიდეს;
3. ბიჯები შეიძლება პერიოდულად გამეორდეს, მაგრამ საერთო ჯამში ბიჯების საერთო რაოდენობა სასრული უნდა იყოს - ალგორითმი როდესაც უნდა გაჩერდეს.

ალგორითმის აგებისას მნიშვნელოვანია ორი ასპექტი:

1. სისწორე - ალგორითმი მართლა აკეთებს თუ არა იმას, რაც მოეთხოვება;
2. სისწრაფე - რა რაოდენობის ბიჯის გავლას ანდომებს ალგორითმი დაწყებიდან დამთავრებამდე;

ჩვენს ირგვლივ ძალიან ბევრი ამოცანა არსებობს: ხინკლის მოხარშვიდან დაწყებული კოსმოსში რაკეტების გაგზავნით დამთავრებული. შეიძლება თუ არა ყველა ამოცანა ალგორითმულად გადაიჭრას ანუ შეიძლება თუ არა ყველა ამოცანისთვის დავწეროთ ალგორითმი, რომელიც მას ამოხსნის? როგორც აღმოჩნდა, არსებობს ისეთი ამოცანათა სიმრავლე, რომლებსაც ალგორითმულად ვერ ამოვხსნით. უფრო მეტიც, გაცილებით მეტია ისეთი ამოცანები, რომლებსაც ალგორითმულად ვერ ამოვხსნით, ვიდრე ისეთები, რომლებსაც შეიძლება მოვუგონოთ ალგორითმი. ეს კი იმას ნიშნავს, რომ ადამიანის ცხოვრებაში გაცილებით მეტი რამ არის ისეთი, რომელსაც კომპიუტერი ვერ ამოხსნის, ვიდრე ისეთი, რომელსაც ხელოვნური ინტელექტი დაძლევს. აღმოჩნდა რომ, ალგორითმულად ამოხსნად ამოცანებს შორისაც არსებობს ისეთი ამოცანები, რომელთა ამოხსნაც ძალიან დიდ დროს მოითხოვს, უმეტეს შემთხვევებში ჩვენს ხელთ არსებული უძლიერესი გამომთვლელი მანქანებით ასობით ათას წელს ანუ ამ შემთხვევაში ხდება ბიჯების რაოდენობის ძალიან სწრაფი ზრდა. მაგრამ მთავარი აქ ისაა, რომ არ არის ცნობილი, შეიძლება თუ არა ასეთი ამოცანებისათვის დაიწეროს ისეთი ალგორითმი, რომელიც უფრო სწრაფი იქნებოდა. როდესაც წამოიჭრება ახალი ამოცანა, პირველ რიგში, უნდა დავადგინოთ, შეიძლება თუ არა მისი ალგორითმულად ამოხსნა. თუ არ შეიძლება, მაშინ უნდა დავადგინოთ, როგორ შევცვალოთ ამ ამოცანის პირობები ისე, რომ იგი ამოხსნადი გახდეს და ამავედროულად რაც შეიძლება ახლოს იყოს დასმულ ამოცანასთან. თუ ამოცანა ამოხსნადია, უნდა დავადგინოთ, შეიძლება თუ არა მისი სწრაფად ამოხსნა? თუ არ შეიძლება, მაშინ უნდა დავადგინოთ, როგორ შევცვალოთ ამ ამოცანის პირობები ისე, რომ იგი ამოხსნადი გახდეს და ამავედროულად რაც შეიძლება ახლოს იყოს ამ დასმულ ამოცანასთან ან ისეთი სწრაფი ალგორითმი შევქმნათ, რომელიც ზუსტად იმავე მონაცემებზე და პირობებში ზუსტ პასუხთან მიახლოებულ პასუხს მოგვცემს (მიახლოებითი ალგორითმები).

ალგორითმის შესრულების დრო, როგორც წესი, დამოკიდებულია მონაცემების რაოდენობაზე (შეტანის ზომა ანუ შესატანი მონაცემების რაოდენობა). მიღებულია, რომ შესატანი მონაცემების რაოდენობა აღინიშნება n -ით. მნიშვნელოვანია გავარკვიოთ, არა მარტო ის თუ კონკრეტულად რამდენი ოპერაცია დასჭირდება ალგორითმს გარკვეული რაოდენობის მონაცემების შეტანისას, არამედ როგორ მოიქცევა ალგორითმი მონაცემების რაოდენობის გაზრდისას.

მაგალითი 3.1. ორ რიცხვს შორის უდიდესი საერთო გამყოფის (უსგ) პოვნა. ორ დადებით მთელ a და b რიცხვების უდიდესი საერთო გამყოფი ეწოდება ისეთ უდიდეს მთელ რიცხვს, რომელზეც უნაშთოდ იყოფა როგორც a , ისე b რიცხვი.

მაგალითად, რიცხვებს 12-ს და 18-ს აქვთ საერთო გამყოფები: 2, 3, 6. ყველაზე დიდი საერთო გამყოფი არის 6. ე. ი: $\text{უსგ}(12, 18) = 6$.

ვიპოვოთ 34-სა და 157-ის უდიდესი საერთო გამყოფი გაყოფის მეთოდით:

$$\begin{aligned}\text{უსგ}(34, 157) &= \text{უსგ}(34, 157 \bmod 34) = \text{უსგ}(34, 21) = \text{უსგ}(34 \bmod 21, 21) = \\ &= \text{უსგ}(13, 21) = \text{უსგ}(13, 21 \bmod 13) = \text{უსგ}(13, 8) = \text{უსგ}(13 \bmod 8, 8) = \\ &= \text{უსგ}(5, 8) = \text{უსგ}(5, 8 \bmod 5) = \text{უსგ}(5, 3) = \text{უსგ}(5 \bmod 3, 3) = \\ &= \text{უსგ}(2, 3) = \text{უსგ}(2, 3 \bmod 2) = \text{უსგ}(2, 1) = 1\end{aligned}$$

პასუხი: $\text{უსგ}(34, 157) = 1$

უდიდესი საერთო გამყოფის მოძებნის ასეთ მეთოდს ჰქვია ევკლიდეს ალგორითმი. ევკლიდის ალგორითმი სიტყვიერად ჩაიწერება ქვემოთ მოცემული სახით:

1. დიდი რიცხვი გავყოთ მცირე რიცხვზე;
2. მცირე რიცხვი გავყოთ ნაშთზე (გაყოფის შედეგად მიღებული ნაშთი);
3. პირველი ნაშთი გავყოთ მეორე ნაშთზე;
4. მეორე ნაშთი გავყოთ მესამე ნაშთზე და ა. შ.
5. გაყოფა უნდა გავაგრძელოთ მანამდე, სანამ ნაშთი არ გახდება ნულის ტოლი;
6. ბოლო გამყოფი ითვლება უდიდეს საერთო გამყოფად.

მაგალითი 3.2. ჰანოის კოშკი. არსებობს ასეთი ლეგენდა: შორეული აღმოსავლეთის ერთ-ერთ მონასტერში ბერები ცდილობდნენ გადაეტანათ დისკები ერთი ღერძიდან მეორეზე. დისკები ღერძზე ჩამოცმული იყო ისე, რომ მათი ზომა მცირდებოდა ქვემოდან ზემოთ.

დისკების გადატანის დროს აუცილებელი მოთხოვნა იყო შემდეგი წესების დაცვა:

- ერთ გადატანაზე შეიძლება გადაიტანო მხოლოდ ერთი დისკი;
- დიდი ზომის დისკი არასდროს არ უნდა აღმოჩნდეს პატარა დისკის ზემოდან;
- შეგვიძლია გამოვიყენოთ მხოლოდ ერთი დამხმარე ღერძი;

ბერები ცდილობდნენ ეს პროცესი შეესრულებინათ 64 დისკისთვის.

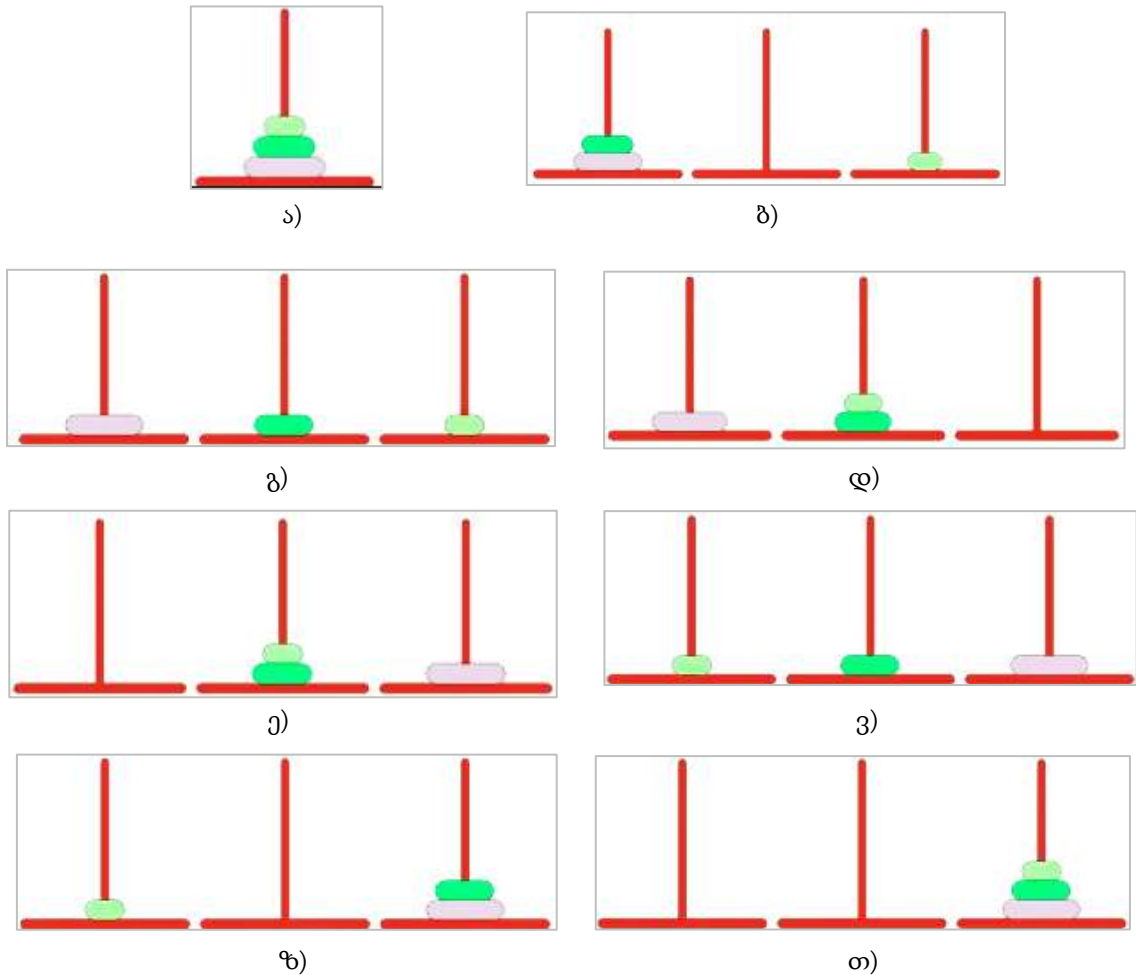
ლეგენდა ასევე ამბობს, რომ როდესაც ისინი ამ პროცესს დაასრულებდნენ (ისინი ამ ყველაფერს კომპიუტერის გარეშე აკეთებდნენ) ქვეყნიერების დასასრულიც დადგებოდა.

ჰანოის კოშკის ამოხსნა. განვიხილოთ ჰანოის კოშკი, რომელზეც მოთავსებულია სამი დისკი. პირველი ღერძიდან მესამე ღერძზე გადასატანია 3 დისკი, მეორე ღერძს ვიყენებთ დისკების დროებით განსათავსებლად ანუ დამხმარე ღერძად. ამოცანას თუ დავუკვირ-

დებით შევამჩნევთ, რომ სამი დისკის გადატანა შეიძლება გამარტივდეს ორი დისკის გადატანის საშუალებით.

n დისკის გადატანა მოხდება $n - 1$ დისკის გადანაცვლებით.

ჰანოის კოშკის დისკები თავდაპირველად ღერძზე დალაგებულია კლებადობის მიხედვით (ნახ. 3.1. ა,ბ,გ,დ,ე,ვ,ზ,თ);



ნახ. 3.1. ჰანოის კოშკი: ა) თავდაპირველი მდგომარეობა; ბ) პირველი ბიჯი;

გ) მეორე ბიჯი; დ) მესამე ბიჯი; ე) მეოთხე ბიჯი; ვ) მეხუთე ბიჯი

ზ) მეექვსე ბიჯი; თ) დასასრული;

პირველი ბიჯი: 1-ლი დისკი გადაადგილდა 1-ლი ღერძიდან მე-3 ღერძზე;

მეორე ბიჯი: მე-2 დისკი გადაადგილდა 1-ლი ღერძიდან მე-2 ღერძზე;

მესამე ბიჯი: 1-ლი დისკი გადაადგილდა მე-3 ღერძიდან მე-2 ღერძზე;

მეოთხე ბიჯი: მე-3 დისკი გადაადგილდა 1-ლი ღერძიდან მე-3 ღერძზე;

მეხუთე ბიჯი: 1-ლი დისკი გადაადგილდა მე-2 ღერძიდან 1-ელ ღერძზე;

მეექვსე ბიჯი: მე-2 დისკი გადაადგილდა მე-2 ღერძიდან მე-3 ღერძზე;

მეშვიდე ბიჯი: 1-ლი დისკი გადაადგილდა 1-ლი ღერძიდან მე-3 ღერძზე;

4 დისკის შემთხვევაში გვექნება 15 ბიჯი.

5 დისკის შემთხვევაში - 31 ბიჯი.

ამდენად შეგვიძლია დავასკვნათ, რომ n რაოდენობის დისკისათვის უნდა შევასრულოთ $2^n - 1$ ბიჯი:

2 დისკი = $(2^2) - 1 = 3$ ბიჯი;

3 დისკი = $(2^3) - 1 = 7$ ბიჯი;

4 დისკი = $(2^4) - 1 = 15$ ბიჯი;

5 დისკი = $(2^5) - 1 = 31$ ბიჯი და ა.შ.

ფუნქციათა ზრდის სისწრაფე და მისი გამოყენება ალგორითმების შეფასებისას

დასმული ამოცანის გადაჭრის პროცესში შესაბამისი ალგორითმის შერჩევა საკმაოდ რთულია. ამდენად, სანამ დასმული ამოცანის გადაწყვეტად ავირჩევთ რომელიმე ალგორითმს, აუცილებელია შეაფასოთ ალგორითმის სირთულე და ეფექტურობა.

ალგორითმის სირთულე არის სიდიდე (მნიშვნელობა), რომელიც გვიჩვენებს საჭირო რესურსების (დრო ან დამატებითი მეხსიერება) წესრიგს დასმულ ამოცანასთან მიმართებით. ამიტომ ალგორითმებს ახასიათებენ ორი სახის სირთულით: დროითი სირთულე $T(n)$ და სივრცითი (მეხსიერებითი) სირთულე $V(n)$.

ფუნქციათა ზრდის სისწრაფე აღწერს, როგორ იზრდება ალგორითმის სირთულე, შესატანი მონაცემების რაოდენობის გაზრდასთან მიმართებაში. ყველაზე ხშირად ფუნქციათა ზრდის სისწრაფე წარმოდგენილია O -ნოტაციის სახით (გერმანულიდან *Ordnung* და ნიშნავს წესრიგს): $O(f(x))$, სადაც $f(x)$ ასახავს ალგორითმის სირთულეს.

პრაქტიკაში ალგორითმის სირთულის შეფასებისას უფრო ხშირად გამოიყენება დროითი სირთულე. ალგორითმის დროითი სირთულე დამოკიდებულია შესატანი მონაცემების რაოდენობაზე (ზომაზე). შესატან მონაცემებს აღწერენ n პარამეტრით. პარამეტრი n დასამუშავებელი მონაცემების პირდაპირპროპორციულია და შეიძლება აღნიშნავდეს:

1) მასივის ან ფაილის ზომას დახარისხებისას ან ძიებისას;

2) სიმბოლოების რაოდენობას სტრიქონში;

3) განსახილველი ამოცანის ოდენობას სხვა აბსტრაქტული ზომის დადგენისას. ამბობენ, რომ ალგორითმის დროით სირთულეს აქვს $T(n)$ რიგი n ზომის შესატან მონაცემებთან მიმართებით.

პრაქტიკაში $T(n)$ -ის ზუსტი განსაზღვრა საკმაოდ რთულია. სწორედ ამიტომ მეცნიერებმა გადაწყვიტეს გამოიყენონ დიდი O -ს ასიმპტოტური სიმბოლო.

დიდი O -ს (O სიმბოლიკა, O ნოტაცია) ცნება მათემატიკიდან შემოიტანეს. O ნოტაცია აღწერს ალგორითმის მუშაობის ასიმპტოტურ დროს და არა ზუსტ დროს. ბუნებრივია განსხვავებულ ოპერაციებს მივაწერთ განსხვავებული ბიჯების რაოდენობას ე. ი. წონებს

ანუ რამდენ ბიჯს „იწონის“ თითოეული ოპერაცია. ალგორითმების ანალიზისთვის მთავარია მხოლოდ, რომ ეს ბიჯები ყოველ სიტუაციაში ერთნაირად ითვლებოდეს.

ვთქვათ, მოცემული გვაქვს რაიმე $f(n)$ ფუნქცია. n წარმოადგენს შესატანი მონაცემების ზომას. ამბობენ, რომ ალგორითმი მუშაობს $O(f(n))$ დროში, თუ არსებობს ისეთი C რიცხვი, რომ ალგორითმი ასრულებს არა უმეტეს, ვიდრე $C * f(n)$ -ზე ოპერაციას.

ფორმალურად $O(f(n))$ ფუნქციაში $f(n)$ ასახავს ალგორითმის სირთულეს.

O-ნოტაცია არის ალგორითმის შესრულების დროითი შეფასება და გვიჩვენებს ალგორითმის შესრულების დროის დამოკიდებულებას შესატანი მონაცემების რაოდენობის ზრდასთან. **O-ნოტაცია** ასევე ცნობილია, როგორც ალგორითმის სირთულის შეფასება.

საინტერესოა შესატანი მონაცემების რაოდენობის გაზრდისას როგორ მოიქცევა ალგორითმი: შენელებს თუ პირიქით მისი მუშაობის სისწრაფე გაიზრდება? სწორედ ასეთ შეკითხვებს პასუხობს **O-ნოტაცია**. მოვიყვანოთ ალგორითმის ზრდის სიჩქარის რაოდენობრივი (რიცხვითი) მნიშვნელობები სხვადასხვა ფუნქციებისთვის, რომლებიც უფრო ხშირად გამოიყენება ალგორითმების დროითი სირთულის შეფასებისას (ცხრილი 3.1).

ცხრილი 3.1. ალგორითმის ზრდის სიჩქარის რაოდენობრივი (რიცხვითი) მნიშვნელობები სხვადასხვა ფუნქციისთვის

n მონაცემების რაოდენობა	$\log(n)$	$n \cdot \log(n)$	n^2
1	0	0	1
16	4	64	256
256	8	2 048	65 536
4 096	12	49 152	16 777 216
65 536	16	1 048 476	4 294 967 296
1 048 476	20	20 969 520	1 099 301 922 576
16 775 616	24	402 614 784	281 421 292 179 456

ჩავთვალოთ, რომ რიცხვები შეესაბამება მიკროწამებს. ალგორითმი 1 048 476 ელემენტის დამუშავებას $O(\log(n))$ დროითი სირთულით მოანდომებს 20 მიკროწამს, ხოლო $O(n^2)$ დროითი სირთულით ალგორითმი მოანდომებდა - 12 დღეზე მეტს.

მაგალითად, ალგორითმის მუშაობისთვის საჭირო მოქმედებები (ბიჯები) გამოითვლება მრავალწევრით: $1.3n^2 + 7n \cdot \log n + 3n + 2$. ასეთი ალგორითმის დროით სირთულეს აქვს $O(n^2)$ სისწრაფე (წონა, რიგი).

ფაქტიურად თუ ალგორითმს სჭირდება $O(n^2)$ დროითი წონა, მაშინ ამოცანის შესრულების დრო არ გაიზრდება უფრო სწრაფად, ვიდრე ელემენტების რაოდენობის კვადრატი.

ალგორითმის ზრდის სისწრაფე

ალგორითმების სირთულე ვასდება ან შესრულების დროით ან გამოყენებული მეხსიერებით. მიუხედავად იმისა, რომ ნამდვილი რიცხვების გაყოფა პროცესორისგან უფრო მეტ დამუშავებას მოითხოვს, ვიდრე მთელი რიცხვების შეკრება, ალგორითმების თეორიაში, ალგორითმის სირთულის მიხედვით, ორივე ოპერაცია თანაბრად ითვლება.

ალგორითმის ზრდის სისწრაფე გვიჩვენებს, შესატანი მონაცემების გაზრდასთან ერთად თუ როგორ იზრდება ალგორითმის სირთულე.

ალგორითმების სირთულის გაზომვისას, ვგულისხმობთ იმ ანალიზის დროს, რომელიც საჭიროა ძალიან დიდი მონაცემების ერთობლიობის დამუშავებისთვის. ასეთი სახის ანალიზს ასიმპტოტური ანალიზი ეწოდება. რამდენი დრო დასჭირდება ათი ელემენტის მქონე ან ათასი ელემენტის მქონე ან ათი მილიონი ელემენტის მქონე მასივის დამუშავებას? თუ ალგორითმი ათას ელემენტს აწესრიგებს ხუთი მილიწამში, რა მოხდება, თუ ჩვენ მას მილიონ ელემენტს გადავცემთ? მონაცემების დამუშავებისთვის ალგორითმს დასჭირდება 5 წუთი თუ 5 წელი?

ვთქვათ, გვაქვს 15 კაციანი სია და გვინდა ამ სიაში ვიპოვოთ ყველა ის ადამიანი, ვისი სახელიც იწყება T ასოთი. დახარისხებისთვის შეგვიძლია გამოვიყენოთ სხვადასხვა ალგორითმები და ამასთანავე ყველა მათგანი განსხვავებული იქნება სირთულის ხარისხით. ზოგიერთი ალგორითმი სხვებზე უფრო სწრაფად იმუშავებს.

ახლა კი წარმოვიდგინოთ, რომ ჩვენ გვაქვს არა 15 სახელი, არამედ მასივი მილიონი სახელით. როგორ ფიქრობთ, რამდენად იმოქმედებს მონაცემთა ასეთი ზრდა ალგორითმის შესრულებაზე? ამ კითხვაზე პასუხს იძლევა უკვე ნაცნობი დიდი O -ნოტაციის გამოყენება.

განვიხილოთ შედარებით უფრო ხშირად გამოყენებადი ალგორითმის ზრდის სისწრაფე, მეორენაირად O -ნოტაციები დროით სირთულესთან მიმართებით:

ალგორითმის ზრდის სისწრაფე $O(1)$ - მუდმივი დროითი სირთულე.

ზრდის სისწრაფე $O(1)$ არის ისეთი გამოთვლითი სირთულე (გამოთვლების რაოდენობა), რომელიც არ არის დამოკიდებული შესატანი მონაცემების ზომაზე. უნდა გვახსოვდეს, რომ ციფრი 1-ანი არ ნიშნავს, რომ ალგორითმი ერთ ოპერაციაში შესრულდება ან მას შესასრულებლად ძალიან ცოტა დრო სჭირდება.

ალგორითმის შესრულება შეიძლება მიკროწამიდან ერთ წლამდე გაგრძელდეს. მნიშვნელოვანია, რომ ეს დრო არ არის დამოკიდებული შეტანილ მონაცემებზე.

მაგალითად, მასივში რაიმე ელემენტთან წედომას ან ჰემცხრილებს აქვს $O(1)$ დროითი სირთულე.

ალგორითმის ზრდის სისწრაფე $O(n)$ - წრფივი დროითი სირთულე

ზრდის სისწრაფე $O(n)$ ნიშნავს, რომ ალგორითმის მუშაობის დრო (გამოთვლების რაოდენობა) პირდაპირ არის დამოკიდებული შემომავალი მონაცემების ზომაზე ანუ ალგორითმის სირთულე წრფივად იზრდება შესატანი მონაცემების ზრდასთან ერთად. თუ მონაცემთა რაოდენობა იზრდება, მაშინ პროპორციულად იზრდება მუშაობის დროც. ერთი ელემენტის დასამუშავებლად წრფივი სირთულის ალგორითმს სჭირდება ხუთი მილიწამი, მაშინ იგი ათას ელემენტს დაამუშავებს ხუთ წამში. მარტივი ძებნის ალგორითმს აქვს წრფივი დროითი სირთულე. რაც უფრო იზრდება ელემენტთა რაოდენობა, მით უფრო დიდხანს მუშაობს ალგორითმი. მასივში ერთი ელემენტის დამატებისას ფუნქციის მუშაობის დრო პროპორციულად გაიზრდება. წრფივი დროითი სირთულის მქონე ალგორითმი გამოიყენება ციკლურ სტრუქტურებში.

ალგორითმის ზრდის სისწრაფე $O(\log(n))$ - ლოგარითმული დროითი სირთულე

ზრდის სისწრაფე $O(\log(n))$ გვეუბნება, რომ ალგორითმის შესრულების დრო (გამოთვლების რაოდენობა) დამოკიდებულია შემომავალი მონაცემების ზომის ლოგარითმზე.

მაგალითად ორობითი ძებნის (binary search tree) ალგორითმს აქვს ლოგარითმული დროითი სირთულე. ისეთი ალგორითმების უმრავლესობა, რომელიც მუშაობს „შუაზე გაყოფის“ პრინციპით, აქვს ლოგარითმული სირთულე.

ვთქვათ მოცემული გვაქვს დახარისხებული რიცხვითი მასივი, საჭიროა მოვძებნოთ ციფრი 9. გამოვიყენოთ შუაზე გაყოფის მეთოდი (იხ. ნახ. 3.2.) პირველ ეტაპზე ხდება შუა ელემენტის 4-ის ამორჩევა. $9 > 4$ -ზე, ამიტომ, ძებნის გასაგრძელებლად უნდა ავიღოთ მასივის მეორე ნახევარი. გავაგრძელებთ შედარებებს, მანამდე სანამ არ მოვძებნით საძებნ ელემენტს.

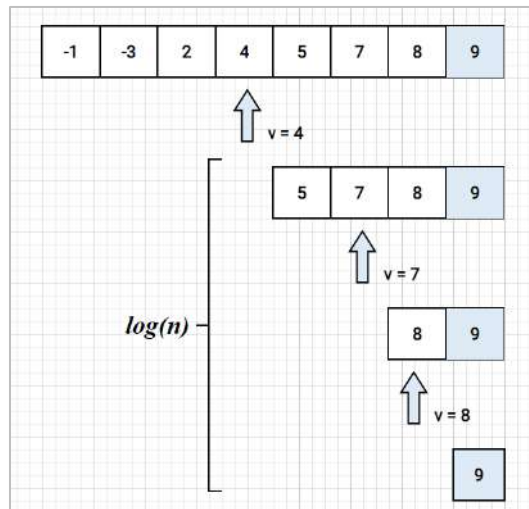
ალგორითმის ზრდის სისწრაფე $O(n^2)$ - კვადრატული დროითი სირთულე

ზრდის სისწრაფე $O(n^2)$ ნიშნავს, რომ ალგორითმის შესრულების დრო (გამოთვლების რაოდენობა) პირდაპირ პროპორციულია შესატანი მონაცემების ზომის კვადრატისა. მარტივად რომ ვთქვათ, ეს არის წრფივი დროის სირთულის კვადრატი.

მაგალითად, მონაცემთა მასივში არის 2 ელემენტი, ალგორითმის მუშაობის განმავლობაში შესრულდება 4 ოპერაცია. თუ მასივში არის 4 ელემენტი, მაშინ გვექნება 16 ოპერაცია, თუ არის 6 ელემენტი გვექნება 36 ოპერაცია და ა.შ.

კვადრატული დროითი სირთულის შემთხვევაში 100 ელემენტიან მასივს დასჭირდება 1000000 ოპერაცია, ხოლო მილიონი ელემენტის მქონე მასივს დასჭირდება 1000000000000 ოპერაცია. კვადრატული სირთულის ალგორითმს ერთი ოპერაციის შესასრულებლად თუ სჭირდება მილიწამი, მაშინ მილიონი ოპერაციის შესასრულებლად მას დასჭირდება 32 წელი, თუ ეს ალგორითმი იქნება ასჯერ ჩქარი, ალგორითმის მუშაობას

დასჭირდება 82 დღე. ბუშტისებრი დახარისხების ალგორითმს გააჩნია კვადრატული დროითი სირთულე კვადრატული დროითი სირთულის მქონე ალგორითმი განეკუთვნება ნელა მომუშავე ალგორითმების რიცხვს.



ნახ. 3.2. ორობითი ძებნის მაგალითი

ალგორითმის ზრდის სისწრაფე $O(n!)$ - ფაქტორიალური დროითი სირთულე

ზრდის სისწრაფე $O(n!)$ გვიჩვენებს, რომ ალგორითმის შესრულების დრო (გამოთვლების რაოდენობა) ფაქტორიალურად იზრდება მონაცემთა ნაკრების ზომის ზრდასთან ერთად. ფაქტორიალური დროითი სირთულე არის დროითი სირთულის ერთ-ერთი ცუდი ვარიანტი, რადგან ოპერაციების რაოდენობა იზრდება ასტრონომიულ ზღვრებამდე მონაცემთა ნაკრების მცირე ზრდითაც კი.

$$2! = 2 * 1 = 2;$$

$$3! = 3 * 2 * 1 = 6;$$

$$4! = 4 * 3 * 2 * 1 = 24;$$

.....

$$8! = 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 40320.$$

ალგორითმის ფაქტორიალური დროითი სირთულის მაგალითია მარტივი რეკურსიული ფუნქცია. ფაქტორიალური დროითი სირთულის მქონე ალგორითმი განეკუთვნება ძალიან ნელა მომუშავე ალგორითმების რიცხვს.

მაგალითად, დავწეროთ მოზაიკური თავსატეხის აწყობის ალგორითმი (იხ. ნახ. 3.3.). თუ მოზაიკური თავსატეხის ფრაგმენტების რიცხვს აღვნიშნავთ n -ით, მაშინ ფრაგმენტების გადატანა/გადმოტანის რიცხვი ტოლი იქნება $n! = 1 * 2 * 3 * ... * n$.

თუ გვაქვს მოზაიკური თავსატეხის ორი ფრაგმენტი $n = 2$, მაშინ გვექნება $2! = 1 * 2 = 2$ ფრაგმენტის გადატანა/გადაადგილების ვარიანტი.

თუ გვაქვს მოზაიკური თავსატეხის სამი ფრაგმენტი $n = 3$, მაშინ გვექნება $3! = 1 * 2 * 3 = 6$ ფრაგმენტის გადატანა/გადაადგილების ვარიანტი.



ნახ. 3.3. მოზაიკური თავსატეხი

მოზაიკურ თავსატეხს გააჩნია ფაქტორიალური ალგორითმული სირთულე.

დასკვნა: მოზაიკური თავსატეხის აწყობის ალგორითმს აქვს $O(n!)$ - ფაქტორიალური დროითი სირთულე.

ალგორითმის ზრდის სისწრაფე $O(n * \log(n))$ - წრფივი-ლოგარითმული სირთულე წრფივი-ლოგარითმული სირთულის მქონე ალგორითმს გააჩნია $O(n * \log(n))$ ზრდა.

ზოგიერთ „დაყავი და იბატონე“ ტიპის ალგორითმს გააჩნია $O(n * \log(n))$ სირთულე. მაგალითად წრფივი-ლოგარითმული სირთულე აქვს სწრაფი დახარისხების ალგორითმებს. განვიხილოთ ალგორითმის ზრდის სისწრაფე ციფრებში (იხ. ნახ. 3.4):

$O(1)$ - 1 (ერთი) ოპერაცია;

$O(\log(n))$ - დაახლოებით 13 (შეიძლება ცოტა მეტიც) ოპერაცია;

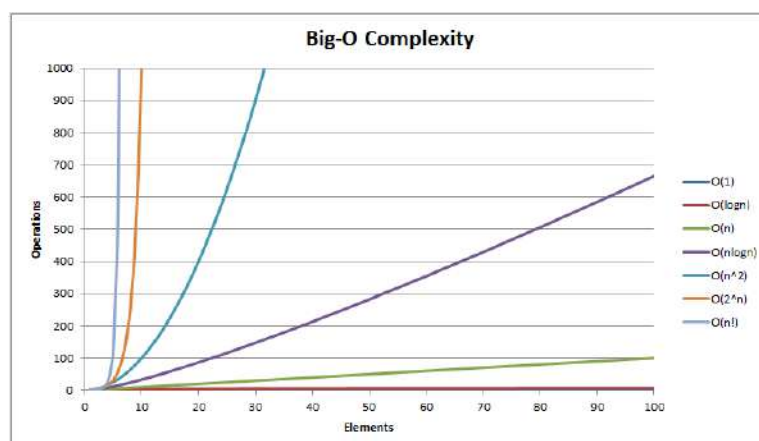
$O(n)$ - 10 000 (ათი ათასი) ოპერაცია;

$O(n^2)$ - 100 000 000 (ასი მილიონი) ოპერაცია;

$O(n^3)$ - 1 000 000 000 000 (ერთი ტრილიონი) ოპერაცია;

$O(n!)$ - 10 000 000 000 000 000 000 000 000... და კიდევ ბევრი ნულით ოპერაცია.

შესრულების დროის შედარება: ტრილიონი წლების შემდეგ ჯერ სამყარო გაქრება (ღმერთმა არასოდეს დაუშვას) და მხოლოდ ამის შემდეგ დაასრულებს ალგორითმი მუშაობას.



ნახ. 3.4. ზრდის სისწრაფის დინამიკა ოპერაციების რაოდენობის დამოკიდებულებასა და ელემენტების რაოდენობას შორის

ალგორითმის მუშაობის მინიმალური, მაქსიმალური და საშუალო დრო. ალგორითმის ბიჯების რაოდენობის ზედა და ქვედა ზღვრის შეფასების მეთოდები

ალგორითმების თეორიაში დასმული ამოცანის ამოხსნის დრო განისაზღვრება იმ ოპერაციების (ბიჯების) რაოდენობით, რომელიც სჭირდება ალგორითმის შედეგის მისაღებად. თუ მცირე ზომის მასივის დასამუშავებლად დაგვჭირდება 10 წამი, მაშინ უფრო დიდი მასივისთვის დრო გაიზრდება 100 წამით. ინტუიციურად, ალგორითმის შესრულების დრო დამოკიდებულია მონაცემების ზომაზე.

ალგორითმების თეორიაში, დროის საზომი ერთეულები: წამები, წუთები და საათები, არ ითვლება ალგორითმის მუშაობის დროის ერთეულად. ალგორითმის მუშაობის დრო დამოკიდებულია იმ სისტემურ ბლოკზე, რომელზეც ხდება ალგორითმის შესრულება და სხვა გარე ფაქტორებზე. ამიტომ დრო დაითვლება არა წამებში და საათებში, არამედ იმ ოპერაციების რაოდენობით, რომელსაც ალგორითმი ასრულებს. ასეთი გადაწყვეტა არის საიმედო და რაც მთავარია, არ არის დამოკიდებული სისტემურ ბლოკზე.

რა იგულისხმება, როდესაც ვამბობთ, რომ ალგორითმის სირთულის ზრდის სისწრაფე არის $O(n)$? ეს საშუალო შემთხვევა? ან ყველაზე ცუდი? ან იქნებ საუკეთესო? შემთხვევაა. ალგორითმების ბიჯების რაოდენობა დამოკიდებულია ალგორითმის მონაცემთა რაოდენობასა და თვითონ მონაცემთა მნიშვნელობებზე.

განასხვავებენ ბიჯების შეფასების სამ შემთხვევას ანუ ალგორითმი ამომწურავად ხასიათდება სამი მახასიათებლით:

უარესი შემთხვევის ანალიზი (worst-case) - მაქსიმუმ რამდენი ბიჯი დაგვჭირდება დასმული ამოცანის გადასაჭრელად, მაშინაც კი, როდესაც ყველაზე „ცუდი“ მონაცემები შემოგვცა?

საუკეთესო შემთხვევის ანალიზი (best-case) - მინიმუმ რამდენი ბიჯი დაგვჭირდება დასმული ამოცანის გადასაჭრელად, როდესაც ყველაზე „კარგი“ მონაცემები შემოგვცა?

საშუალო შემთხვევის ანალიზი (average-case) - საშუალოდ რამდენი ბიჯი დაგვჭირდება დასმული ამოცანის გადასაჭრელად?

საუკეთესო შემთხვევა ($\Omega(n)$) - ნიშნავს შემავალი მონაცემების მიხედვით დროითი სირთულის მინიმალურ მნიშვნელობას ანუ ელემენტარული ბიჯების ყველაზე ნაკლებ რაოდენობას. **უარესი შემთხვევა ($O(n)$)** იქნება ელემენტარული ბიჯების მაქსიმალური რაოდენობა შემავალი მონაცემების მიხედვით. ბევრი მეცნიერი და პროგრამისტი მხოლოდ უარეს მაჩვენებელს აღიარებს ალგორითმის სირთულედ. **საშუალო შემთხვევის ($\theta(n)$)** დროს გვანტერესებს ალგორითმის გარკვეული ოპერაციის შესრულებას რამდენი ელემენტარული ბიჯი სჭირდება საშუალოდ, შემავალი მონაცემების მიხედვით.

მარტივ შემთხვევაში ამის გამოთვლა შემდეგნაირად ხდება: საჭიროა განვიხილოთ შემავალი მონაცემების ყველა შესაძლო ვარიანტი (ყველა დასაშვები ვარიანტის სიმრავლე).

განვიხილოთ ალგორითმის ზრდის სისწრაფის ძირითადი ასიმპტოტური შეფასებები:

O დიდი (O – ნოტაცია) - აფასებს ალგორითმის დროითი ან სივრცითი სირთულის ზედა ზღვარს; დროითი ფუნქციის ზრდის ზედა ასიმპტოტური შეფასება; **O დიდი** განსაზღვრავს ალგორითმის მუშაობის ზედა საზღვარს. ფუნქცია აღწერს დამოკიდებულებას შემავალ პარამეტრებსა და ოპერაციების რაოდენობას შორის, რომლებიც უნდა შესრულდეს.

Ω დიდი (ომეგა) - აფასებს ალგორითმის სირთულის ქვედა ზღვარს; დროითი ფუნქციის ზრდის ქვედა ასიმპტოტური შეფასება; გამოიყენება ალგორითმის მუშაობის ქვედა საზღვრის აღსაწერად.

Θ დიდი (თეტა) - გამოიყენება ალგორითმის ზუსტი სირთულის დასადგენად; დროითი ფუნქციის ზრდის ქვედა და ზედა ასიმპტოტური შეფასებები; გამოიყენება ალგორითმის მოქმედების ზედა და ქვედა საზღვრების დასადგენად (მკაცრი ზედა და ქვედა საზღვრები);

დეველოპერები, როგორც წესი, ირჩევენ ყველაზე უარეს შემთხვევას, დიდ O -ს, რადგან არ იციან, თუ როგორ პირობებში მუშაობს ალგორითმი: საუკეთესო ან თუნდაც საშუალო პირობებში. ეს საშუალებას აძლევთ გააკეთონ შემდეგი განცხადება, როგორიცაა „უარეს შემთხვევაში, ჩემი ალგორითმი საკმაოდ სწრაფად მუშაობს“.

საზოგადოდ ყველაზე საინტერესოდ ითვლება O -ნოტაცია. ალგორითმების სირთულე ძირითადად მხოლოდ ზედა ასიმპტოტური ფუნქციით ფასდება. მონაცემთა რაოდენობის ზომა აღვნიშნოთ n -ით.

ალგორითმის ზრდის ფუნქცია $f(n)$ შეგვიძლია შემოვსაზღვროთ (შევზღუდოთ) ასიმპტოტურად ფუნქციით $g(n)$: (იხ. ნახ. 3.5.)

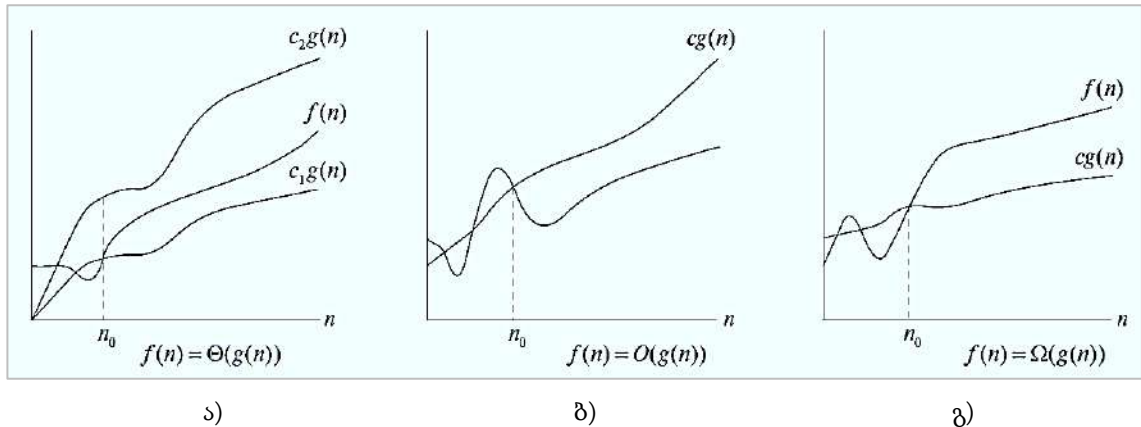
$f(n) \in O(g(n))$ - ფუნქცია $f(n)$ შეზღუდულია ზემოდან $g(n)$ ფუნქციით;

$f(n) \in \Omega(g(n))$ - ფუნქცია $f(n)$ შეზღუდულია ქვემოდან $g(n)$ ფუნქციით;

$f(n) \in \Theta(g(n))$ - ფუნქცია $f(n)$ შეზღუდულია ქვემოდან და ზემოდან $g(n)$ ფუნქციით;

მაგალითად, ოთახის დალაგების დრო წრფივად არის დამოკიდებული ოთახის ფართობზე ($\Theta(S)$). ფართობის n -ჯერ ზრდასთან ერთად, დალაგების დრო ასევე n -ჯერ გაიზრდება. სატელეფონო წიგნში ადამიანის გვარის მოძებნას დასჭირდება წრფივი დრო $O(n)$, თუ გამოვიყენებთ წრფივი ძებნის ალგორითმს, ან დასჭირდება დრო, რომელიც ლოგარით-

მულად იქნება დამოკიდებული მონაცემების (გვარების) რაოდენობაზე ($O(\log(n))$) ორობითი ძეგნის გამოყენების შემთხვევაში.



ნახ. 3.5. ალგორითმის ასიმპტოტური შეფასებები:

ა) საშუალო შემთხვევა; ბ) უარესი შემთხვევა; გ) საუკეთესო შემთხვევა;

პროგრამისტის თვალთ დანახული ალგორითმების სისწრაფე

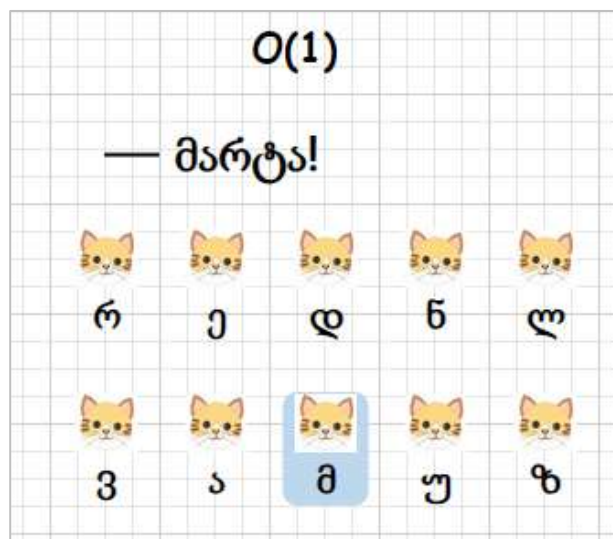
ახლა განვიხილოთ, თუ როგორ აფასებენ პროგრამისტები სწრაფ და ნელ ალგორითმებს.

ალგორითმების სისწრაფის დადგენაში დაგვეხმარება შინაური ცხოველი - კატა.

$O(1)$, წრფივი დრო

ალგორითმის სისწრაფე არ არის დამოკიდებული კატების რაოდენობაზე (ყველაზე საუკეთესო ვარიანტი).

მაგალითად, სახლში გყავთ n ცალი კატა. ყველა კატამ იცის საკუთარი სახელი. თუ დაუძახებთ „მარტას“, მხოლოდ მოირბენს „მარტა“, დანარჩენ $n - 1$ კატას არც კი აინტერესებს, მათთვის სულერთია თქვენი მოქმედება ☺ (ნახ. 3.6).

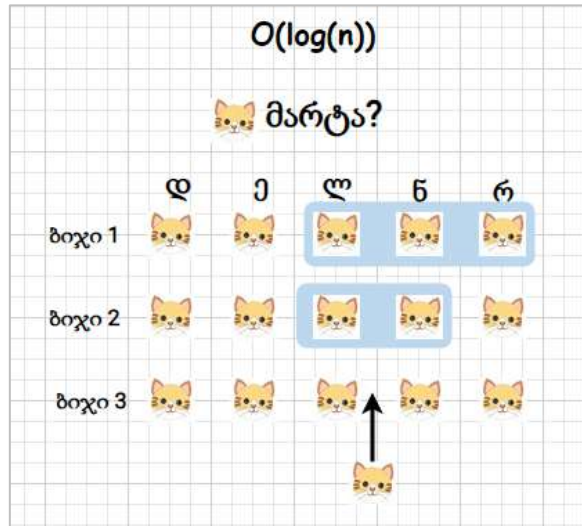


ნახ. 3.6. ალგორითმის მუშაობის წრფივი დრო ($O(1)$)

$O(\log(n))$, ლოგარითმული დრო

ალგორითმი n ცალი რაოდენობის კატისთვის გამოიმუშავებს $\log(n)$ ბიჯს. ეს ალგორითმი საკმაოდ სწრაფად მუშაობს.

მაგალითად, კატების საჭმლის თევშები, განლაგებულია კატის სახელის ანბანის მიხედვით. თუ მოიყვანთ ახალ კატას (მაგ., „მარტას“), მაშინ მისი ლამბაქის ადგილი იქნება $\log(n)$ ბიჯზე (ნახ. 3.7).

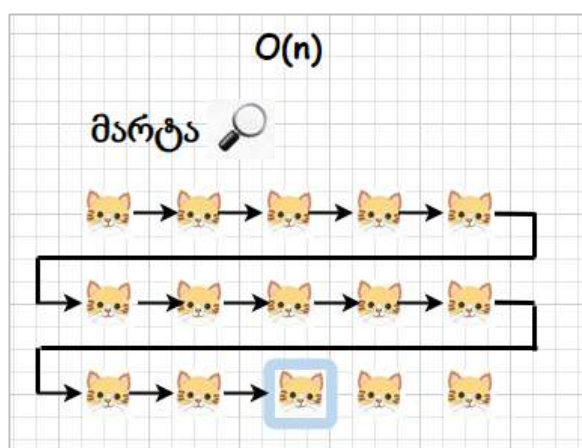


ნახ. 3.7. ალგორითმის მუშაობის ლოგარითმული დრო ($O(\log(n))$)

$O(n)$, წრფივი დრო

n რაოდენობის კატაზე ალგორითმი გამოიმუშავებს n რაოდენობის ბიჯს. ეს ნიშნავს, რომ ყოველ ჯერზე უნდა გადავურბინოთ (გავიაროთ) ყველა კატას.

მაგალითად, კატები აპროტესტებენ რაღაცას და არ პასუხობენ სახელის დამახეზაზე ☹. ამიტომ მოგვიწევს n რაოდენობის კატების გარშემო „გადავირბინოთ“, რათა ვიპოვოთ ის, რომელიც გვპირდება - ჩვენს შემთხვევაში „მარტა“ (ნახ. 3.8).



ნახ. 3.8. ალგორითმის მუშაობის წრფივი დრო ($O(n)$)

$O(n * \log(n))$, წრფივი-ლოგარითმული დრო

n რაოდენობის „კატაზე“ ალგორითმი გამოიმუშავებს $n * \log(n)$ ბიჯს. ეს უფრო მეტი დროა, ვიდრე წრფივი დრო. მატემატიკიდან ვიცით, რომ n -ს ლოგარითმი ბევრად ნაკლებია

ვიდრე n . მაგალითად, ველოდებით სტუმრებს, ამიტომ გადაწყვეტეთ, რომ კატები ზომის მიხედვით დავალაგოთ. სწრაფი დახარისხების ალგორითმი ამას $n * \log(n)$ რაოდენობის ბიჯით შეასრულებს (ნახ. 3.9).

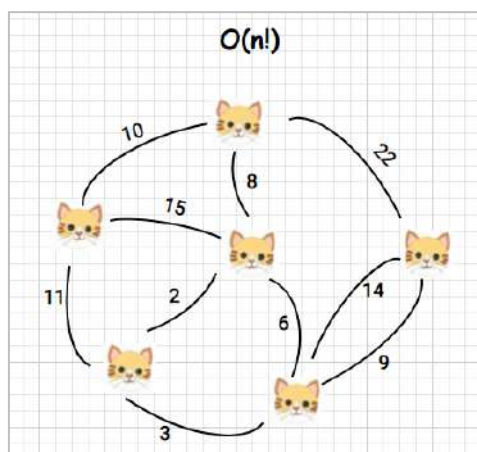


ნახ. 3.9. ალგორითმის მუშაობის წრფივი-ლოგარითმული დრო $O(n * \log(n))$

$O(n!)$, ფაქტორიალური დრო

n რაოდენობის კატაზე ალგორითმი გამოიმუშავებს $n * (n - 1) * (n - 2) * \dots * 1$ ბიჯს. ეს ძალიან დიდი რიცხვია. სულ 20 ცალ კატასთან მიმართებით ალგორითმი მოგვცემს რამდენიმე კვინტილიონ ოპერაციას.

მაგალითად, კატები ბინაში სხვადასხვა ადგილას არიან გაბნეული. პატრონს უნდა, რომ მოეფეროს თითოეულ მათგანს, მაგრამ ეზარება ადგომა და თითოეულ მათგანთან მისვლა. რომელია ის უმოკლესი გზა, რომლითაც პატრონი მოახერხებს თითოეულთან მისვლას. ამ საკითხის გადასაჭრელად, დაგვჭირდება $n!$ შედარება (ნახ. 3.10).

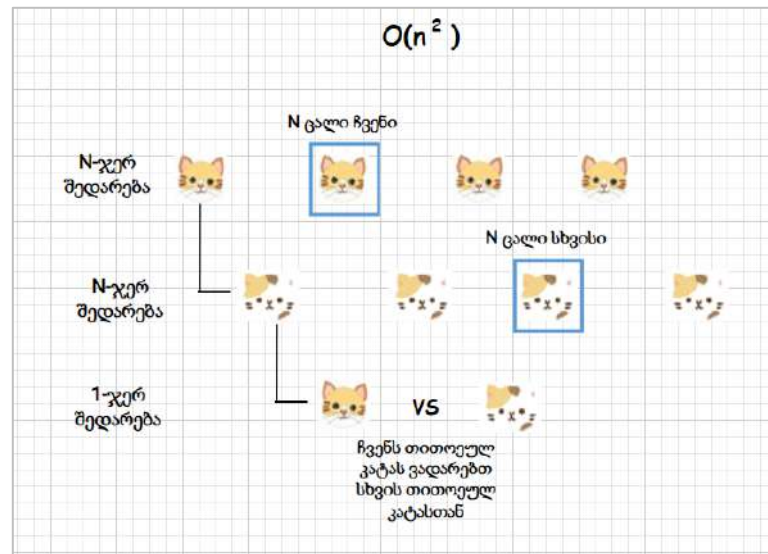


ნახ. 3.10. ალგორითმის მუშაობის ფაქტორიალური დრო $O(n!)$

$O(N^2)$, კვადრატული დრო

n რაოდენობის კატაზე ალგორითმი გამოიმუშავებს n^2 ბიჯს. ალგორითმი მუშაობს ძალიან და ძალიან ნელა. მაგ., კონკურენტი ირწმუნება, რომ მათი n ცალი კატა უფრო მოვლილი და ბედნიერები არიან ვიდრე ჩვენები. ამიტომ სპეციალური კომისია წყვილ-

წყვილად შეადარებს ცხოველებს და სამართლიან განაჩენს გამოიტანს ☺. ამდენად საჭირო გახდება, შევასრულოთ n^2 შედარება ანუ n^2 რაოდენობის ბიჯი (ნახ. 3.11).



ნახ. 3.11. ალგორითმის მუშაობის კვადრატული დრო $O(n^2)$

ამდენად მესამე თემაში ჩვენ განვიხილეთ ალგორითმების შეფასება, ფუნქციითა ზრდის სისწრაფე და მისი გამოყენება შეფასებისას, ალგორითმის მუშაობის მინიმალური, მაქსიმალური და საშუალო დრო, ალგორითმის ბიჯების რაოდენობის ზედა და ქვედა ზღვრის შეფასების მეთოდები, განვიხილეთ მაგალითები ვიზუალიზაციის გამოყენებით, რათა უფრო ადვილად აღვიქვათ მოცემულ თემაში განხილული საკითხები.

სორტირების და ძებნის ალგორითმები. ბინარული ძებნა, სორტირება არჩევით, სორტირება ჩასმით. სორტირება შერწყმით, ჩქარი სორტირება, სორტირების დროის ქვედა საზღვარი და მისი დაძლევა.

დახარისხების, დალაგების ალგორითმები

დახარისხება/დალაგების ალგორითმი, მონაცემების ზრდის მიხედვით დალაგების შემთხვევაში, ელემენტებს ისე განლაგებს, რომ გასაღების (სამეზნი ელემენტის) ყოველი მომდევნო მნიშვნელობა უნდა იყოს უფრო დიდი, ვიდრე წინა. თუ დახარისხება ხდება კლებადობით, გასაღების ყოველი მომდევნო მნიშვნელობა, შესაბამისად, წინაზე ნაკლები უნდა იყოს. მაგალითად, საწყისი მასივია: 4, 5, 3, 1, 9, 0, 7, 2, 6, 8

ზრდადობით დალაგება: $0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9$

კლებადობით დალაგება: $9 > 8 > 7 > 6 > 5 > 4 > 3 > 2 > 1 > 0$

დღესდღეობით არსებობს დახარისხება/დალაგების უამრავი ალგორითმი და გამო-სავალი პრობლემის გადასაჭრელად. თუმცა, არ არსებობს საუკეთესო გამოსავალი, რომელიც შეიძლება გამოყენებულ იქნას ნებისმიერი ამოცანის ამოსახსნელად.

დახარისხების ამა თუ იმ ტიპის არჩევისას მთავარი კრიტერიუმია თავად იმ მონაცემ-თა ერთობლიობა, რომელიც უნდა დალაგდეს. იმის მიხედვით, თუ რა ვიციტ მონაცემების შესახებ, შესაძლებელი ხდება ავირჩიოთ შესაბამისი დახარისხების მეთოდი. როდესაც ვმუშაობთ მონაცემების მასივთან, უნდა გავითვალისწინოთ გამოყენებული ალგორითმის თავისებურებები. დახარისხების ალგორითმების მახასიათებლებია:

1. დრო. დრო არის მთავარი პარამეტრი, რომელიც გვიჩვენებს ელემენტების დახარისხების სიჩქარეს საწყისი მდგომარეობიდან მიღებული შედეგის ჩათვლით. მას ასევე უწოდებენ გამოთვლით სირთულეს. ძირითადად, დრო ფასდება ალგორითმის მიერ შესრულებული ბიჯების რაოდენობით. ბიჯის ქვეშ იგულისხმება ელემენტების შედარე-ბა და ჩანაცვლების რაოდენობა. განასხვავებენ ალგორითმის მუშაობის ყველაზე ცუდ, საშუალო და საუკეთესო შედეგებს. ყველაზე ცუდ შედეგს მივიღებთ მაშინ, თუ ალგორითმი შეასრულებს ბიჯების მაქსიმალურ რაოდენობას. საშუალო შედეგი მიიღება, ალგორითმის მიერ შესრულებული ბიჯების საშუალო რაოდენობის შემთხვევის დროს. საუკეთესო შედეგის მიღება ხშირად შესაძლებელია, თუ მოცემული თანამიმდევრობა უკვე დახარისხებულია ან ახლოსაა დალაგებულ მდგომარეობასთან.

2. მეხსიერება. ალგორითმს შესასრულებლად შეიძლება დასჭირდეს მეხსიერების დამატებითი ადგილის დაკავება. ზოგიერთი ალგორითმის მუშაობისთვის საჭიროა თავდა-

პირველი მასივის ან მის ნაწილების დუბლირება. იმის გათვალისწინებით, რომ თავად მასივი შეიძლება იყოს ძალიან დიდი, მაშინ მეხსიერების ხარჯი შეიძლება იყოს ძალიან მნიშვნელოვანი და მთავარი კრიტერიუმი დახარისხების ალგორითმის არჩევისას.

3. სტაბილურობა. სტაბილური დახარისხება არ ცვლის ერთნაირი მონაცემების პოზიციებს.

4. ქცევის ბუნებრიობა. მეთოდი ეფექტურია უკვე დალაგებული ან ნაწილობრივ დალაგებული მონაცემების დამუშავებისას.

5. გამოყენების სფერო. გამოყენების სფერო განსაზღვრავს ალგორითმის დანიშნულებას და ხასიათდება დახარისხების ორი ტიპით - შიგა და გარე.

შიგა დახარისხების დროს ხდება მუშაობა ისეთ მონაცემთა მასივთან, რომელიც მთლიანად განთავსებულია მეხსიერებაში და შესაძლებელია მასივის ნებისმიერ ელემენტზე წვდომა. გარე დახარისხება მუშაობს დიდი მოცულობის მონაცემებთან და შეუძლია სიის ცალკეულ ელემენტზე თანამიმდევრული წვდომა, მაგალითად, ფაილების ორგანიზება.

ორობითი ძებნის ალგორითმი (Binary search)

ორობითი (ბინარული) ძებნის ალგორითმი პოულობს გარკვეული მნიშვნელობისთვის პოზიციას დახარისხებულ (სორტირებულ) მასივში. ის უარეს შემთხვევაში ლოგარითმულ დროში მუშაობს და ახორციელებს $O(\log(n))$ შედარებას, სადაც n არის ელემენტების რაოდენობა მასივში. დიდ მასივებში ორობითი ძებნა გაცილებით სწრაფია ვიდრე წრფივი ძებნა, თუმცა აუცილებელია მასივი იყოს დახარისხებული, რათა შევძლოთ მოცემული ალგორითმის გამოყენება. ორობითი (ბინარული) ძებნა აუცილებლად უნდა განხორციელდეს დახარისხებულ მასივში.

ორობითი (ბინარული) ძებნის ალგორითმი. დავუშვათ, A მასივის ელემენტები დალაგებულია ზრდადობის მიხედვით. ორობითი (ბინარული) ძებნის ალგორითმის მიხედვით უნდა მოვძებნოთ მასივის შუა ელემენტი, რომლის მნიშვნელობაც უნდა შევადაროთ წინასწარ მოცემულ, საძებნ მნიშვნელობას. თუ მათი მნიშვნელობები ერთმანეთს დაემთხვა, მაშინ მნიშვნელობა მოძებნილია მასივში და ძიების პროცესი წყდება. საწინააღმდეგო შემთხვევაში, თუ საძებნი მნიშვნელობა აღმოჩნდება ნაკლები მასივის შუა ელემენტის მნიშვნელობაზე, ძიება ხორციელდება (იწარმოება) მასივის პირველ ნახევარში, ხოლო თუ ცვლადის მნიშვნელობა გადააჭარბებს მასივის შუა ელემენტის მნიშვნელობას, ძიება ხორციელდება საწყისი მასივის მეორე ნახევარში.

ამდენად, ძიების პირველივე ეტაპზე ადგილი აქვს მასივის განახევრებას, შემდეგ მიღებული ნახევრებიდან ერთ-ერთის იგნორირების გზით და მასივის მეორე ნახევრის

შუაზე გაყოფით ძებნა წარმოებს საწყისი მასივის მეოთხედში და ა. შ. პროცესი გრძელდება მანამდე, სანამ მასივის ერთ-ერთი ნახევრის შუა ელემენტის მნიშვნელობა არ დაემთხვევა მოცემული ცვლადის მნიშვნელობას ანუ სანამ საძებნი ელემენტი არ მოიძებნება.

ორობითი (ბინარული) ძებნის ალგორითმის შედეგი: ალგორითმი გვიჩვენებს მასივში მოიძებნა თუ არა საძებნი ელემენტი.

ორობითი (ბინარული) ფსევდოკოდი:

ორობითი (ბინარული) ძებნის ალგორითმს დავარქვათ Find (A, c):

Find (A, c)

{

სანამ A მიმდევრობა ცარიელი არაა, შევასრულოთ შემდეგი ოპერაციები:

a = A მიმდევრობის შუა ელემენტი;

if (a == c) return “კი”;

else

if (c > a) { A = A მიმდევრობის მარჯვენა ნახევარი; Find (A,c); }

if (c < a) { A = A მიმდევრობის მარცხენა ნახევარი; Find (A,c); }

}

განვიხილოთ მაგალითი: დავუშვათ გვაქვს მასივი 12 ელემენტით. მასივი დალაგებულია ზრდადობის მიხედვით (იხ. ნახ. 4.1.ა,ბ,გ,დ).

ვთქვათ მომხმარებელი მასივში ეძებს ელემენტს მნიშვნელობით 4 (ასევე შეგვიძლია ვუწოდოთ ძებნის გასაღები). პირველ ბიჯზე მასივი იყოფა ორ ნაწილად: ვეძებთ საშუალო ელემენტს: საშუალო = $(0+11)/2 = 5$ (ხდება 0.5-ის იგნორირება).

თავდაპირველად მოწმდება მასივის შუა ელემენტი. თუ ეს ელემენტი დაემთხვევა საძებნი ელემენტის მნიშვნელობას, მაშინ ალგორითმი წყვეტს მუშაობას და პროგრამა გასცემს შეტყობინებას: „მნიშვნელობა მოიძებნა“. ჩვენს შემთხვევაში, საძებნი მნიშვნელობა 4 არ ემთხვევა საშუალო მნიშვნელობას.

თუ საძებნი მნიშვნელობა ნაკლებია საშუალო მნიშვნელობაზე, მაშინ ალგორითმი არ გააგრძელებს ძებნას მასივის იმ ნაწილში, რომელშიც იმყოფებიან საძებნი მნიშვნელობაზე მეტი მნიშვნელობის მქონე ელემენტები (ანუ ჩვენს შემთხვევაში საშუალო მნიშვნელობიდან მასივის ბოლომდე). მარჯვენა საძიებო საზღვარი შეიცვლება და დაიყვანება საშუალოს მინუს ერთი (საშუალო - 1) მნიშვნელობამდე. შემდეგ მასივი ისევ უნდა დაიყოს ორ ნაწილად.

საძებნი მნიშვნელობა ისევ არ არის ტოლი საშუალო მნიშვნელობისა ანუ 3-ის. საძებნი მნიშვნელობა მასზე მეტია ($4 > 3$). ამ შემთხვევაში მარცხენა საზღვარი შეიცვლება საშუალოს

პლუს ერთი (საშუალო + 1) მნიშვნელობამდე. მესამე ბიჯზე საშუალო მნიშვნელობა იქნება მასივის უჯრედი ინდექსით 3 ანუ $(3+4)/2=3$. იგი ტოლია საძებნი მნიშვნელობისა. ალგორითმმა დაამთავრა მუშაობა.

ორობითი ძებნა ეფექტური ალგორითმია - მისი სირთულის შეფასებაა $O(\log(n))$, მაშინ როცა წრფივ თანამიმდევრულ ძიებას აქვს $O(n)$ სირთულე. ორობითი ძებნის ალგორითმი დახარისხებულ მასივში ეძებს განსაზღვრული მნიშვნელობის პოზიციას $O(\log(n))$ დროში.

მაგალითად, 1024 ელემენტისგან შემდგარი მასივისთვის, ცუდ შემთხვევაში წრფივი ძიება (როდესაც საძებნი ელემენტი არ არის მასივში) დაამუშავებს ყველა 1024-ივე ელემენტს, მაგრამ ბინარული ძებნის დროს საკმარისია $\log(1024) = 10$ ელემენტის დამუშავება. ეს შედეგი მიიღწევა იმის გამო, რომ ციკლის პირველი ბიჯის შემდეგ საძიებო არე ვიწროვდება 512 ელემენტამდე, მეორე ბიჯის შემდეგ - 256-მდე და ა.შ.

	ინდექსი 0	ინდექსი 1	ინდექსი 2	ინდექსი 3	ინდექსი 4	ინდექსი 5	ინდექსი 6	ინდექსი 7	ინდექსი 8	ინდექსი 9	ინდექსი 10	ინდექსი 11
საწყისი მასივი	1	2	3	4	5	6	7	8	9	10	11	12

ა)

	ინდექსი 0	ინდექსი 1	ინდექსი 2	ინდექსი 3	ინდექსი 4	ინდექსი 5	ინდექსი 6	ინდექსი 7	ინდექსი 8	ინდექსი 9	ინდექსი 10	ინდექსი 11
პირვალი ბიჯი	1	2	3	4	5	6	7	8	9	10	11	12
მარცხენა						საშუალო						მარჯვენა

ბ)

	ინდექსი 0	ინდექსი 1	ინდექსი 2	ინდექსი 3	ინდექსი 4	ინდექსი 5	ინდექსი 6	ინდექსი 7	ინდექსი 8	ინდექსი 9	ინდექსი 10	ინდექსი 11
მეორე ბიჯი	1	2	3	4	5	///	///	///	///	///	///	///
მარცხენა			საშუალო		მარჯვენა							

გ)

	ინდექსი 0	ინდექსი 1	ინდექსი 2	ინდექსი 3	ინდექსი 4	ინდექსი 5	ინდექსი 6	ინდექსი 7	ინდექსი 8	ინდექსი 9	ინდექსი 10	ინდექსი 11
მესამე ბიჯი	///	///	///	4	5	///	///	///	///	///	///	///
				მარცხენა/საშუალო	მარჯვენა							

დ)

ნახ. 4.1. ორობითი ძებნა: ა) საწყისი მასივი; ბ) საშუალოს პოვნა;

გ) მარჯვენა საზღვარი; დ) ელემენტი მოიძებნა

ორობითი ძებნის ალგორითმის უარყოფითი მხარედ შეიძლება ჩაითვალოს წინასწარ დახარისხებული მონაცემების მოთხოვნა, ასევე მონაცემთა ნებისმიერ ელემენტზე წვდომის შესაძლებლობა მუდმივ დროში (არ არის დამოკიდებული მონაცემთა რაოდენობაზე). ამგვარად, ორობითი ძებნის ალგორითმს არ შეუძლია იმუშაოს დაულაგებელ მასივებთან და ზმული სიების მქონე მონაცემთა სტრუქტურებთან.

შერჩევითი (ამორჩევითი) დახარისხება (SELECTION SORT)

შერჩევითი დახარისხების ალგორითმი შეიძლება ორნაირად განხორციელდეს, ან უდიდესის მოძებნით და შესაბამისად ბოლოში დამატებით, ან უმცირესის მოძებნით და შესაბამისად თავში დამატებით. განვიხილოთ პირველი ვარიანტი, მისი მუშაობის პრინციპი შემდეგნაირია:

შერჩევითი დახარისხების ალგორითმი ირჩევს დაუხარისხებელ ნაწილში ყველაზე დიდ ელემენტს დაუხარისხებელი ნაწილის სხვა ელემენტებთან თითო-თითოდ შედარებით, შემდეგ კი ამ ნაწილის ბოლო ელემენტსა და უდიდეს ელემენტს უცვლის ადგილს. ცხადია, ამ დროს დაუხარისხებელი ნაწილის ზომა ერთით მცირდება. ალგორითმი მეორდება მანამ, სანამ დაუხარისხებელი ნაწილის ზომა ნულზე მეტია. ამორჩევითი დახარისხება მუშაობს $O(n^2)$ დროში, რაც მეტყველებს მის არაეფექტურობაზე, განსაკუთრებით მაშინ, როდესაც საქმე დიდ მასივს ეხება.

მეტი სიცხადისთვის განვიხილოთ მაგალითი. ვთქვათ მოცემული გვაქვს მასივი:

```
int arr[] = { 4, 9, 0, 3, 1, 6, 5, 7, 8, 2 };
```

საწყის ეტაპზე ვირჩევთ მინიმალურ ელემენტს. ამორჩევითი დალაგების ალგორითმის მიხედვით მასივის პირველივე ელემენტი ითვლება მინიმალურ ელემენტად (ჩვენს შემთხვევაში 4-ანია):

```
int arr[] = { 4, 9, 0, 3, 1, 6, 5, 7, 8, 2 };
```

პირველივე ელემენტს ვადარებთ მეორე ელემენტს $4 < 9$ (გაიცა მნიშვნელობა: „ჭეშმარიტი“) და მინიმალურ ელემენტად ისევ 4 დარჩა, შემდეგ მას ვადარებთ 0-ს, $4 < 0$ (გაიცა მნიშვნელობა: „მცდარი“) და მინიმალური ელემენტი ხდება 0, შემდეგ ნული შედარდება სხვა დარჩენილ ელემენტებთან და საბოლოოდ დავადგენთ მინიმალურ ელემენტს.

დავადგინეთ, რომ მინიმალური ელემენტი არის 0 ე.ი. პირველ იტერაციაზე 0 უნდა ჩაჯდეს პირველ ადგილზე ანუ 4-ის ნაცვლად და ასეც მოხდება: 4 და 0 გაცვლიან ადგილებს. ამ ყველაფრის გათვალისწინებით 0 არის დალაგებული ნაწილის პირველი ელემენტი, ასე რომ, ამ ნაწილის შემოწმება შემდეგ იტერაციებზე საჭირო აღარაა.

პირველი შემოწმების შემდეგ მასივს ექნება შემდეგი სახე:

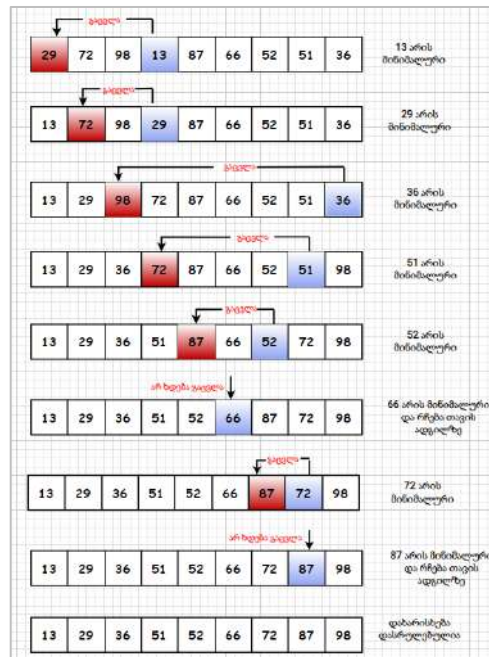
```
int arr[] = { 0, 9, 4, 3, 1, 6, 5, 7, 8, 2 };
```

ახლა ბიჯი ერთით იზრდება და მინიმალური ელემენტის დადგენა მეორე ელემენტიდან დაიწყება ანუ 9 შედარდება დანარჩენ ელემენტებს.

თუ 9 ყველაზე მცირე აღმოჩნდება, მაშინ თავის ადგილზე დარჩება, თუ არადა იგი გაუცვლის ადგილს მინიმალურ ელემენტს, ჩვენ შემთხვევაში 1-იანს. შემდეგ იტერაციაზე 4-იანი 2-იანს გაუცვლის ადგილს:

```
int arr[] = { 0, 1, 2, 3, 9, 6, 5, 7, 8, 4 };
```

კიდევ უფრო მეტი სიცხადისთვის მოდი ვნახოთ სურათი, რომელიც აღწერს შერჩევითი (ამორჩევითი) დახარისხების ალგორითმის მუშაობას (ნახ. 4.2.).



ნახ. 4.2. შერჩევითი (ამორჩევითი) დახარისხება (SELECTION SORT)

ჩასმით დახარისხება (INSERTION SORT)

ჩასმით დახარისხება არის მონაცემთა მარტივი (ანუ არ იყენებს სხვა მასივს მონაცემთა შენახვისთვის) დახარისხების ალგორითმი. ჩასმით დახარისხების ალგორითმი მუშაობს $O(n^2)$ დროში, მაგრამ თუ მასივი თითქმის დალაგებულია, ის დაახლოებით წრფივ დროში მუშაობს. უარეს შემთხვევაში კვადრატულობა მაინც არაეფექტიანობას უსვამს ხაზს, თუმცა მას რამდენიმე უპირატესობა აქვს:

- მარტივი რეალიზება;
- ბევრად ეფექტურია კვადრატული დახარისხების სხვა ალგორითმებთან შედარებით;
- სტაბილურია ანუ არ ცვლის სხვადასხვა ტოლი მნიშვნელობის მქონე ელემენტების თანამიმდევრობას.

ჩასმით დახარისხების ალგორითმის მუშაობის პრინციპი შემდეგია:

იგი მასივს ყოფს ორ ნაწილად: 1) დახარისხებულად, რომლის ზომა თავიდან ერთის ტოლია, რადგან ერთ ელემენტიანი მასივი ყოველთვის დახარისხებულია და 2) დაუხარისხებულად, რომლის პირველ ელემენტს სვამს მასზე ნაკლები ან ტოლი და მასზე მეტი მნიშვნელობის მქონე ელემენტებს შორის. ალგორითმი მეორდება მანამ, სანამ დაუხარისხებელი ნაწილის ზომა იქნება ნულზე მეტი.

განვიხილოთ ჩასმით დახარისხების ალგორითმი დაწვრილებით: დავუშვათ, რომ მარცხენა ნაწილი უკვე დახარისხებულია და მარჯვენა დაუხარისხებელი. საწყის ეტაპზე

(ვიცით, რომ მასივი დაუხარისხებელია) პირველივე ელემენტი ითვლება დახარისხებულად, ამის შემდეგ ყოველი ელემენტი პირველიდან (ინდექსით 0) დაწყებული შედარდება მის მარცხნივ მდგომ ელემენტს (უნდა აღინიშნოს, რომ 0-ოვანი ინდექსის მარცხნივ არცერთი ელემენტი, ასე რომ ათვლა მეორე ელემენტიდან ანუ ინდექსით 1-ანი დაიწყება), თუ ნაკლებია იგი გაუცვლის ადგილს (იხ. ნახ. 4.3.).

ახლა ჩასმით დახარისხების მუშაობის პრინციპი განვიხილოთ მაგალითის სახით:

მოცემული გვაქვს მასივი: {23, 1, 10, 5, 2}

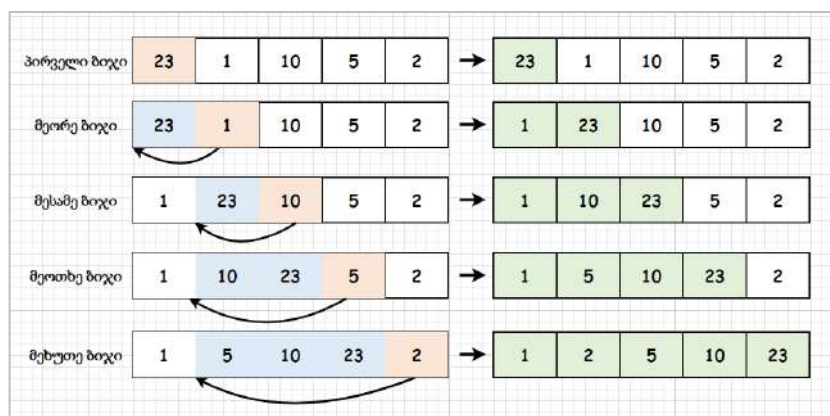
პირველი ბიჯი: მასივის პირველი ელემენტი 23; ჩასმით დახარისხებისას მასივის პირველი ელემენტი ითვლება დახარისხებულად. დახარისხებული ნაწილი ინდექსით 0-იანი არის [23]

მეორე ბიჯი: შევადაროთ 1-იანი 23-თან. ვინაიდან 1-ანი უფრო მცირეა, გავუცვალოთ ადგილები 1-ანსა და 23-ს. დალაგებული ნაწილია: [1, 23]

მესამე ბიჯი: შევადაროთ 10-იანი 1-ანსა და 23-ს. ვინაიდან 10-ანი არის 1-ზე მეტი და 23-ზე ნაკლები, ამიტომ 10-იანი განთავსდება 1-ანსა და 23-ს შორის. დალაგებული ნაწილი არის: [1, 10, 23]

მეოთხე ბიჯი: შევადაროთ 5-ანი 1-ანს, 10-ანს და 23-ს. ვინაიდან 5 არის 1-ზე მეტი და 10-ზე ნაკლები, 5-ანი განთავსდება 1-სა და 10-ს შორის. დალაგებული ნაწილია: [1, 5, 10, 23]

მეხუთე ბიჯი: შევადაროთ 2-ანი 1-ანს, 5-ანს, 10-ანსა და 23-ს. ვინაიდან 2-ანი უფრო მეტია ვიდრე 1-იანი და ნაკლებია ვიდრე 5-ანი, ამიტომ 2-ანი განთავსდება 1-ანსა და 5-ანს შორის. დალაგებული მასივია: [1, 2, 5, 10, 23]



ნახ. 4.3. ჩასმით დახარისხება (INSERTION SORT)

შერწყმით დახარისხება (Merge Sort)

შერწყმით დალაგების ალგორითმი ზრდადობის ან კლებადობის მიხედვით ალაგებს მიმდევრობას. შერწყმით დახარისხების ალგორითმი ორი (თითქმის ტოლ ელემენტებიან), დალაგებული მიმდევრობის შერწყმით იღებს ერთ მიმდევრობას, რომელიც საბოლოოდ თავდაპირველი (ყველა ელემენტიანი) მიმდევრობის დალაგებული ვარიანტი იქნება.

ვთქვათ გვაქვს ორი A და B დალაგებული მიმდევრობები, ჩვენ უნდა მივიღოთ მესამე C მიმდევრობა, რომელიც დალაგებული იქნება და მისი ელემენტები იქნება ამ ორი ელემენტების ერთობლიობა. შერწყმით დახარისხების ალგორითმის ფსევდოკოდი:

Merge (A, B)

C არის ცარიელი მიმდევრობა.

if (A მიმდევრობა ცარიელია) დააბრუნე (C, B);

if (B მიმდევრობა ცარიელია) დააბრუნე (C, A);

if (A მიმდევრობის პირველი ელემენტი < B მიმდევრობის პირველ ელემენტზე)

{

C = (C, A მიმდევრობის პირველი ელემენტი);

A მიმდევრობის პირველი ელემენტი ამოშალე მიმდევრობიდან;

}

else

{

C = (C, B მიმდევრობის პირველი ელემენტი);

B მიმდევრობის პირველი ელემენტი მიმდევრობიდან ამოშალე;

} return (C);

შერწყმის დახარისხების ალგორითმის მუშაობის ძირითადი პრინციპი ასეთია: მასივი უნდა გაიყოს ორ ნაწილად, მასივის თითოეულ ნაწილს ვახარისხებთ შერწყმის ალგორითმის მიხედვით და შემდეგ ორივე მასივს ვაერთიანებთ ერთმანეთთან. თითოეული დაყოფილი მასივიც ასევე ორ ქვემასივად იყოფა, სანამ თითოეულ მასივში ერთი ელემენტი არ აღმოჩნდება. შერწყმით დახარისხებაში გამოიყენება რეკურსია, ასევე შერწყმის ალგორითმს გამოიყენებენ დალაგებული ელემენტების ერთმანეთთან დასაკავშირებლად. მას შემდეგ, რაც ორივე მასივი უკვე დახარისხებულია, უბრალოდ უნდა შევადაროთ ელემენტები ერთმანეთს რიგრიგობით და შევიტანოთ მონაცემები თანამიმდევრობით საბოლოო მასივში.

ახლა კი განვიხილოთ შერწყმის დახარისხების ალგორითმის მუშაობის მაგალითი.

მოცემულია საწყისი მასივი შემდეგი მონაცემებით: [4 2 5 1]

მასივი გავყოთ შუაზე: [4 2] $\leftarrow \rightarrow$ [5 1] (მივიღეთ ორი ახალი მასივი, რაც იმას ნიშნავს, რომ მათთან მიმართებაშიც უნდა გამოვიყენოთ შერწყმით დალაგება)

გავყოთ შუაზე პირველი მასივი: [4] $\leftarrow \rightarrow$ [2]

რადგან მივიღეთ ორი მასივი თითო-თითო ელემენტით, ვალაგებთ მათ და ვაერთიანებთ, როგორც ერთ მასივს: [2 4]

გავყოთ შუაზე მეორე მასივი: $[5] \leftarrow \rightarrow [1]$

რადგან გვაქვს თითო-თითო ელემენტი, ვალაგებთ მათ და ვაერთიანებთ, როგორც ერთ მასივს: $[1\ 5]$

მივიღეთ ორი დახარისხებული ქვემასივი, ახლა უკვე საჭიროა ქვემასივების დახარისხება და გაერთიანება ერთდროულად.

შევადართ ქვემასივების პირველი ელემენტები: 1 და 2. რადგან 1-ანი მცირეა ვიდრე 2-ანი, ძირითად მასივში გადავა 1-ანი და ჩვენ დაგვრჩება შემდეგი ორი მასივი: $[2\ 4]$ და $[5]$

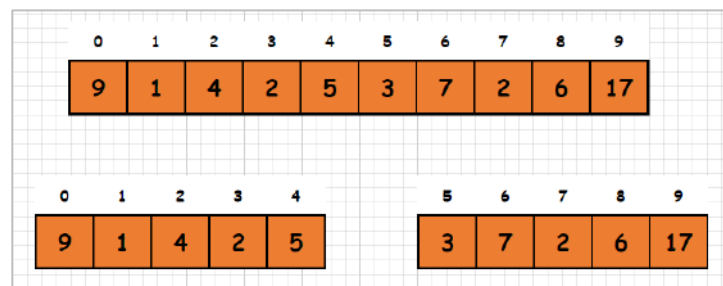
ახლა ისევ შევადართ დარჩენილი ქვემასივების პირველი ელემენტები: 2 და 5. 2-ანი ნაკლებია 5-ზე, ამიტომ ძირითად მასივში გადავა 2-ანი და დაგვრჩება შემდეგი ორი ქვემასივი: $[4]$ და $[5]$

ისევ ვადარებთ ქვემასივების პირველ ელემენტებს ზემოთ განხილული გზით, სანამ არცერთი ელემენტი არ დარჩება (გვექნება) ორივე მასივიდან.

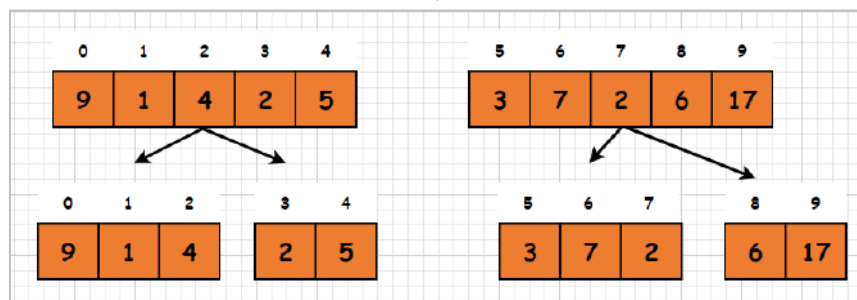
როგორც კი ეს მოხდება, მასივი დალაგებულია.

განვიხილოთ შერწყმით დახარისხების მეორე მაგალითი სქემა-ნახატის სახით, რათა კიდევ უფრო კარგად გავიგოთ შერწყმით დახარისხების მუშაობა (იხ. ნახ. 4.4).

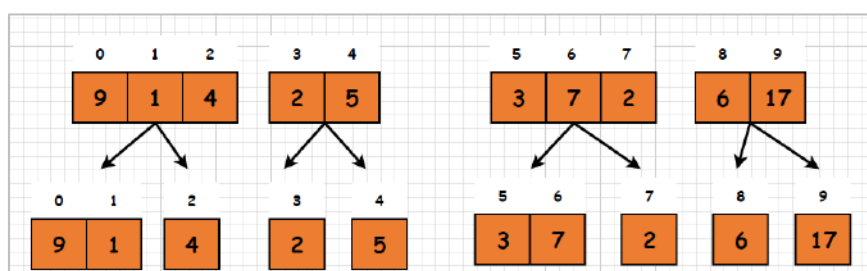
მოცემული გვაქვს მასივი შემდეგი ელემენტებით: $A = [9, 1, 4, 2, 5, 3, 7, 2, 6, 17]$



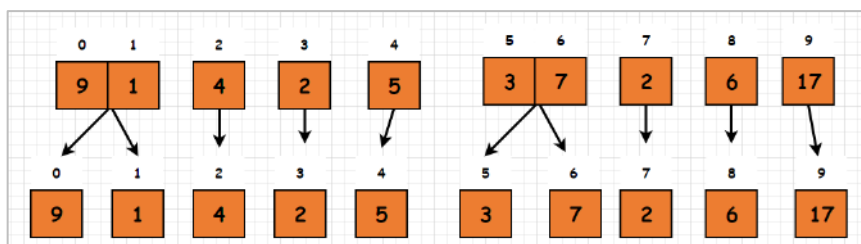
ა)



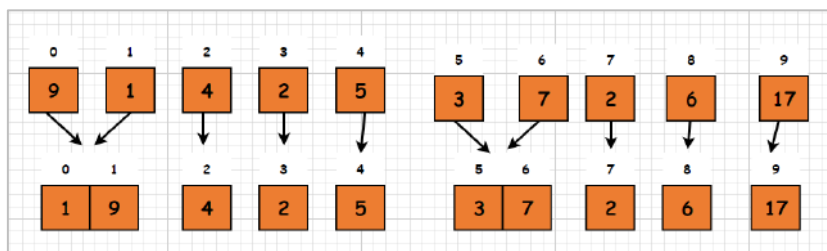
ბ)



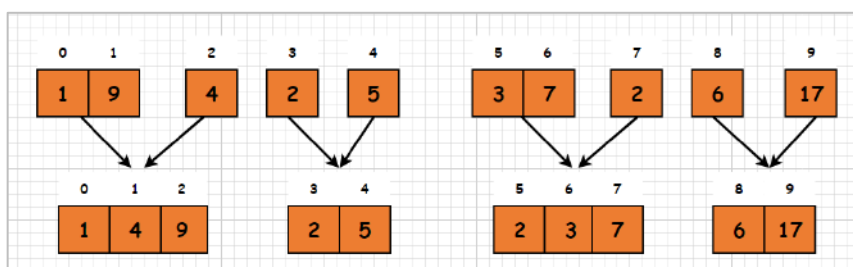
გ)



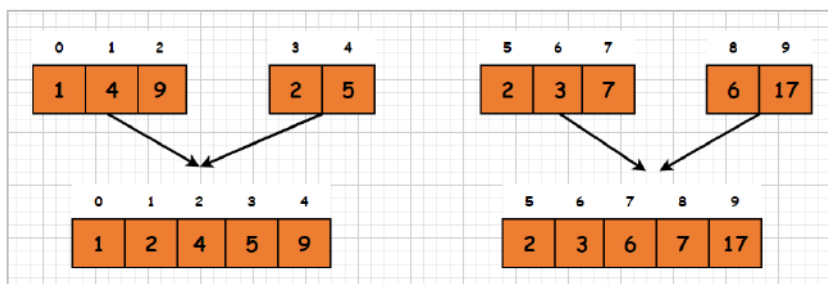
დ)



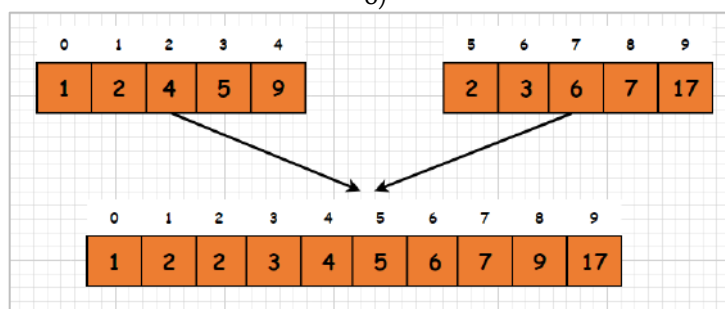
ე)



ვ)



ზ)



თ)

ნახ. 4.4. შერწყმით დახარისხების: ა) ბიჯი 1; ბ) ბიჯი 2; გ) ბიჯი 3; დ) ბიჯი 4; ე) ბიჯი 5;
ვ) ბიჯი 6; ზ) ბიჯი 7; თ) ბიჯი 8;

მე-8 ბიჯზე მივიღეთ დახარისხებულ/დალაგებული მასივი შერწყმით დახარისხების ალგორითმის გამოყენებით.

შერწყმით დახარისხების ალგორითმის მუშაობის შესაფასებლად შევადგინოთ რეკურსიული თანაფარდობა.

დავუშვათ, რომ $T(n)$ არის n ზომის მასივის დახარისხების დრო, მაშინ შერწყმით დახარისხებისთვის ჭეშმარიტი იქნება შემდეგი გამოსახულება:

$$T(n) = 2T(n/2) + O(n)$$

$O(n)$ - არის დრო, რომელიც საჭიროა ორი მასივის გაერთიანებისათვის.

$$T(n) = 2T(n/2) + O(n) = 4T(n/4) + 2O(n) = \dots = T(1) + \log(n) O(n) = O(n \log(n))$$

შერწყმით დახარისხება ითვლება ერთ-ერთ ყველაზე ეფექტურ ალგორითმად ცალმხრივად დაკავშირებულ სიებიდან და ფაილებთან მიმართებით, რადგან შეგვიძლია ელემენტებზე თანამიმდევრული წვდომა. შერწყმით დახარისხების მიზნის არის, რომ იყენებს დამატებით მეხსიერებას, მაგრამ ფაილებთან ან სიებთან მიმართებით, რომლებთანაც წვდომა მხოლოდ თანამიმდევრულად არის შესაძლებელი, ძალიან მოსახერხებელია ამ კონკრეტული ალგორითმის გამოყენება. შერწყმით დახარისხების ალგორითმის უპირატესობაა მისი სტაბილურობა და კარგი შესრულების სისწრაფე $O(n \log(n))$.

სწრაფი დახარისხება (QuickSort)

სწრაფი დახარისხების ალგორითმი 1960 წელს შეიმუშავა ბრიტანელმა მეცნიერმა ტონი ჰოარმა. სწრაფი დახარისხების ალგორითმი დღესდღეობით ყველაზე მეტად გამოყენებადი ალგორითმია.

მიჩნეულია, რომ სწორად განხორციელებული სწრაფი დახარისხების ალგორითმი 2-3-ჯერ უფრო სწრაფია, ვიდრე სხვა დახარისხების ალგორითმები.

სწრაფი დახარისხების ალგორითმის მათემატიკური ანალიზი აჩვენებს, რომ საშუალოდ, ეს ალგორითმი $O(n \log(n))$ შედარებას აკეთებს თუმცა, უარეს შემთხვევას თუ განვიხილავთ, მაშინ სწრაფი დახარისხება $O(n^2)$ შედარებას აკეთებს. უნდა აღვნიშნოთ, რომ ეს ბოლო სიტუაცია ძალიან იშვიათად გვხვდება, სწორედ ამიტომაც სწრაფი დახარისხების ალგორითმი ერთ-ერთი ყველაზე ეფექტური დახარისხების ალგორითმი.

სწრაფი დახარისხების ალგორითმი „დაყავი და იბატონეს“ პრინციპით მუშაობს. სწრაფი დახარისხების ალგორითმი მოცემულ მასივში (სიმრავლეში) ჯერ აირჩევს კონკრეტულ ელემენტს (ზოგჯერ ამბობენ „საყრდენ“ ელემენტს), შემდეგ მასივს (სიმრავლეს) სამ ნაწილად დაყოფს: არჩეულ კონკრეტულ ელემენტებზე ნაკლებ, ტოლ და მეტ ელემენტებად.

სწრაფი დახარისხების ალგორითმი მუშაობს რეკურსიული პრინციპით.

არსებობს კონკრეტული ანუ „საყრდენი“ ელემენტის არჩევის რამდენიმე გზა:

- პირველი ელემენტის არჩევა;
- ბოლო ელემენტის არჩევა;
- შემთხვევითი ელემენტის არჩევა კონკრეტულ ელემენტად;
- ელემენტების მედიანური მნიშვნელობის არჩევა კონკრეტულ ელემენტად.

სწრაფი დახარისხების ალგორითმი შედგება სამი ბიჯისგან:

1. ელემენტის არჩევა. მასივიდან ირჩევენ ელემენტს, რომელსაც ეწოდება კონკრეტული ანუ საყრდენი ელემენტი. ხდება მასივიდან კონკრეტული X ელემენტის არჩევა/დადგენა;

2. დაყოფის ეტაპი. ელემენტების გადანაწილება მასივში უნდა მოხდეს ისე, რომ არჩეულ კონკრეტულ ელემენტზე (საყრდენზე) ნაკლები ელემენტები უნდა გათავსდნენ მის წინ, ხოლო მასზე მეტი ან ტოლი ელემენტები უნდა გათავსდნენ მის შემდეგ ანუ უნდა შესრულდეს პირობა: $A[i] < X$; $A[i] \geq X$; ე. ი. მასივის ერთ ნაწილში იქნება ყველა ის ელემენტი, რომელიც ნაკლებია ან თანაბარია საყრდენ ელემენტზე, ხოლო მეორე ნაწილში განთავსდება ყველა ის ელემენტი, რომელიც მეტია საყრდენ ელემენტზე. აქედან გამომდინარე, მასივის ელემენტები ისე განლაგდებიან, რომ დახარისხებისას პირველი ჯგუფიდან არც ერთი ელემენტი არ მოხვდება მეორე ჯგუფში და პირიქით. ამდენად, საკმარისია მასივის თითოეული ნაწილის ცალკე დახარისხება. სწორედ ეს არის - „დაყავი და იბატონე“ - მიდგომა.

3. პირველი ორი ბიჯის შესასრულებლად გამოიყენება რეკურსიული მეთოდი არჩეული კონკრეტული ელემენტის მარცხნივ და მარჯვნივ მდებარე ორი მასივის (ქვესიმრავლის) მიმართ. რეკურსია არ ვრცელდება ისეთ ქვემასივებზე, რომელიც შეიცავს მხოლოდ ერთ ან არცერთ ელემენტს. თითოეულ ქვემასივში, თუ მასში ორ ელემენტზე მეტია, დაიწყებს მუშაობას რეკურსიული პროცედურა, ისინი შეედარება ერთმანეთს და საჭიროებისამებრ, ადგილს გაცვლს.

სწრაფი დახარისხების ალგორითმის პროცედურას უნდა ჰქონდეს სამი პარამეტრი:

- მასივი, რომელიც უნდა დავახარისხოთ;
- დასახარისხებელი ნაწილის პირველი ელემენტის ინდექსი;
- დასახარისხებელი ნაწილის მეორე ელემენტის ინდექსი;

განვიხილოთ სწრაფი დახარისხების ალგორითმის ფსევდოკოდი:

QUICKSORT($A[]$, START, END)

// თუ პირობა დაკმაყოფილებულია და მასივის სამუშაო ნაწილში

// ორ ელემენტზე ნაკლები ელემენტია,

// მაშინ არაფრის დალაგება აღარ არის საჭირო

თუ $START \geq END$, მაშინ გამოვიდეთ პროცედურიდან

// კონკრეტულ ელემენტად შესაძლებელია ავირჩიოთ

// რაიმე შემთხვევითი რიცხვი

// ან დახარისხებული ნაწილიდან შეიძლება ავირჩიოთ საშუალო ელემენტი

X = მასივიდან შემთხვევითი ელემენტის არჩევა მონაკვეთზე $[START; END]$

$L = START$, $R = END$

სანამ $L \leq R$

// ვეძებთ ელემენტებს ადგილების გასაცვლელად

სანამ $A[L] < X$ $L += 1$

სანამ $A[R] > X$ $R -= 1$

// თუ საჭიროა, მაშინ გადავაადგილოთ მონაცემები

თუ $L \leq R$, მაშინ შევუცვალოთ ადგილები $A[L]$ -ს და $A[R]$ -ს

$L += 1$

$R -= 1$

// დაგვრჩა პირველი და მეორე ნაწილების ცალ-ცალკე დახარისხება

// რეკურსიის გამოძახება

QUICKSORT(A, START, R)

QUICKSORT(A, L, END)

ამ პროცედურას შეუძლია დააღაგოს მასივის ნებისმიერი ნაწილი, იმ შემთხვევაში თუ სწორად მივუთითოებთ START და END პარამეტრებს.

სხვადასხვა დახარისხების ალგორითმების მუშაობის დროის (წამებში) შედარება განსხვავებული ზომის (N) მასივებისთვის (ცხრილი 4.1.):

ცხრილი 4.1. დახარისხების ალგორითმების მუშაობის დროითი შედარების მონაცემები (წამებში)

მასივის ზომა N	ბუშტისებური	ჩასმით	სწრაფი
5000	0.039	0.016	0.00031
15000	0.359	0.141	0.00078
50000	3.985	1.469	0.00297

გაითვალისწინეთ, რომ სწრაფი სორტირების ალგორითმი ნელა იმუშავებს, თუ კონკრეტული (საყრდენი) ელემენტი უდრის მასივის ყველაზე მცირე ან ყველაზე დიდ ელემენტს. ასეთ პირობებში სწრაფი დახარისხება შესრულდება უარეს $O(n^2)$ შემთხვევაში. განვიხილოთ სწრაფი დახარისხების მაგალითი.

მოცემული გვაქვს მნიშვნელობათა თანამიმდევრობა: 2, 6, 7, 9, 3, 2, 5, 1, 4

მიმდევრობის მარცხენა საზღვარი აღვნიშნოთ L -ით, ხოლო მარჯვენა კი R -ით, X -ით კი აღვნიშნოთ a_k -ური ელემენტი, რომლის ინდექსი ტოლია: $k = (L + R)/2$.

პირველი ბიჯი (დაკვირვება-გადარბენა):

$L = 1, \quad R = 9, \quad x = a_5 = 3$

2, 6, 7, 9, 3, 2, 5, 1, 4 - **$a_2 > X, \quad a_8 < X$** შევუცვალოთ ადგილები

2, 1, 7, 9, 3, 2, 5, 6, 4

2, 1, 7, 9, 3, 2, 5, 6, 4 - **$a_3 > X, \quad a_6 < X$** შევუცვალოთ ადგილები

2, 1, 2, 9, 3, 7, 5, 6, 4

2, 1, 2, 9, 3, 7, 5, 6, 4 - დაკვირვებები ერთმანეთს შეხვდა

2, 1, 2, 3, 9, 7, 5, 6, 4

მასივი გაიყო ორ ნაწილად:

2, 1, 2, 3 და 9, 7, 5, 6, 4

თითოეული გაყოფილი ნაწილისთვის უნდა გავიმეოროთ ანალოგიური პროცედურა.

მარცხენა ნაწილისათვის: $L = 1$, $R = 4$, $X = a_2 = 1$

2, 1, 2, 3

1, 2, 2, 3 - მარცხენა ნაწილი დალაგებულია.

მარჯვენა ნაწილისათვის: $L = 1$, $R = 5$, $X = a_3 = 5$

9, 7, 5, 6, 4 - $a_1 > X$, $a_5 < X$ შევუცვალოთ ადგილები

4, 7, 5, 6, 9

4, 7, 5, 6, 9 - დაკვირვებები ერთმანეთს შეხვდა

4, 5, 7, 6, 9

მასივი დაიყო ორ ნაწილად:

4, 5 და 7, 6, 9

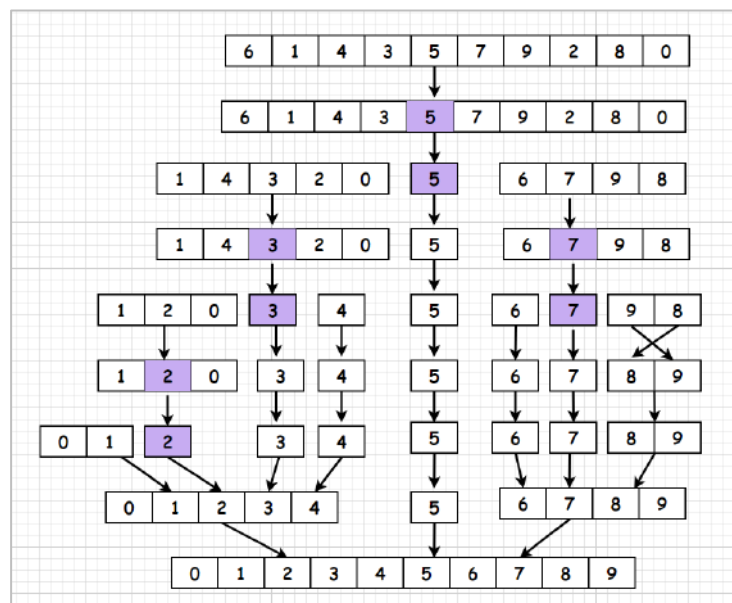
მარცხენა ნაწილი დალაგდა, ხოლო მარჯვენა ნაწილისათვის კი გვჭირდება ერთი გადარბენა:

7, 6, 9

6, 7, 9

გავაერთიანოთ დახარისხებული მასივები, მივიღეთ: 1, 2, 2, 3, 4, 5, 6, 7, 9

სწრაფი დახარისხების ვიზუალური მაგალითი მოცემულია ნახატზე 4.5.



ნახ. 4.5. სწრაფი დახარისხების (QuickSort) ვიზუალური მაგალითი

დახარისხების ალგორითმების სირთულე

მოდით ჯერ გავიხსენოთ, თუ რა არის ზოგადად ალგორითმის სირთულე. ეს საკითხი განხილული გვაქვს მე-3 თემაში და უკვე ვიცით, რომ ალგორითმის სირთულის შეფასების ერთ-ერთი ყველაზე გავრცელებული მეთოდია ასიმპტოტური სირთულე. იგი ფაქტობრივად მიუთითებს იმაზე, თუ რამდენად სწრაფად მუშაობს ალგორითმი, განსაკუთრებით მაშინ, როდესაც შესატანი მონაცემების რაოდენობა იზრდება.

ქვემოთ მოცემულია სამი ძირითადი ასიმპტოტური „O“ სირთულე:

$O(n)$ - წრფივი სირთულე. ალგორითმი მუშაობს შეტანილი მონაცემების ზომის პროპორციულად. რაც უფრო მეტია მონაცემები, მით მეტი დრო იქნება საჭირო. შეტანილი მონაცემების ზომის გაორმაგება ასევე გააორმაგებს ალგორითმის შესასრულებლად საჭირო დროს.

$O(n \log(n))$ - ლოგარითმული სირთულე. ალგორითმის შესრულების დრო პროპორციულია შეტანილი მონაცემების რაოდენობის (ზომის) ლოგარითმისა.

$O(n^2)$, $O(n^3)$, $O(2^n)$ - კვადრატული, კუბური, ექსპონენციალური (მაჩვენებლიანი) სირთულე. სირთულე იზრდება და შესრულების დრო დამოკიდებული ხდება შეტანილი მონაცემების ზომის კვადრატზე, კუბზე, ექსპონენტაზე. ეს საკმაოდ ნელი მუშაობაა, განსაკუთრებით დიდი მონაცემების შემთხვევაში.

მონაცემთა დახარისხების ალგორითმის არჩევა დამოკიდებულია მონაცემთა რაოდენობაზე. დიდი მოცულობის მონაცემებისთვის უმჯობესია გამოიყენოთ ალგორითმები, რომელთა ასიმპტოტური სირთულე დაბალია, ისეთი როგორიცაა სწრაფი და შერწყმით დახარისხება. შევადაროთ სხვადასხვა დახარისხების ალგორითმების ასიმპტოტური სირთულები:

- ბუშტისებური დახარისხება - $O(n^2)$;
- ჩასმით დახარისხება - $O(n^2)$;
- ამორჩევით დახარისხება - $O(n^2)$;
- სწრაფი დახარისხება - $O(n \log(n))$;
- შერწყმით დახარისხება - $O(n \log(n))$.

შედარებიდან ჩანს, რომ სწრაფი და შერწყმით დახარისხებას უფრო დაბალი ასიმპტომური სირთულე აქვთ, რაც მათ ანიჭებს უპირატესობას დიდი რაოდენობის მონაცემების ეფექტურად და სწორად დალაგება/დახარისხებაში.

დასკვნის სახით:

მონაცემების დახარისხება/დალაგებისათვის სწორი ალგორითმის შესარჩევად საჭიროა გავითვალისწინოთ შემდეგი ფაქტორები:

1. შესატანი მონაცემების ზომა. თუ შესატანი მონაცემების რაოდენობა საკმაოდ დიდია, მაგალითად, ათეული ათასობით ან მეტი მონაცემი, მაშინ უნად გამოვიყენოთ, ეფექტური ალგორითმები, როგორიცაა სწრაფი ან შერწყმით დახარისხება.

2. მოთხოვნების სტაბილურობა. ზოგიერთი სიტუაცია მოითხოვს ერთნაირი მნიშვნელობის მქონე მონაცემების თანამიმდევრულობის შენარჩუნებას. ასეთ შემთხვევაში უკეთესია თუ გამოვიყენებთ სტაბილურ ალგორითმებს, როგორიცაა ჩასმითი ან შერჩევითი დახარისხება.

3. შესატანი მონაცემების თავისებურებები. შესატანი მონაცემების მახასიათებლებზე დამოკიდებულების გამო დახარისხების ალგორითმების ქცევა შეიძლება აღმოჩნდეს განსხვავებული. მაგალითად, ნაწილობრივად დალაგებული მონაცემებისთვის ბუშტისებური დახარისხების გამოყენება უფრო ეფექტური იქნებოდა.

როგორც ვხედავთ, შესაბამისი დახარისხების ალგორითმის არჩევა არის მოქნილი და შემოქმედებითი პროცესი, რომელიც მოითხოვს სხვადასხვა ფაქტორის გათვალისწინებას.

დახარისხების ალგორითმი ბუნებრივად იქცევა, თუ იგი ითვალისწინებს ზემოთ განხილულ პარამეტრებს და შესატანი მონაცემების მახასიათებლებს. ყველაფრის ამის გათვალისწინებით შესაძლებელია კონკრეტული პრობლემის გადასაჭრელად შევარჩიოთ დახარისხების სწორი და ოპტიმალური ალგორითმი.

დახარისხება განსაკუთრებით სასარგებლოა შემდეგი ორი მიზეზის გამო:

1. ადამიანის აღქმის კუთხით, მონაცემთა ნაკრები ხდება ბევრად უფრო ადვილად კითხვადი;

2. ამარტივებს მონაცემთა ნაკრებში ძებნისა და მნიშვნელობის ამოღების პროცესის ალგორითმების განხორციელებას.

დახარისხების ეფექტურობის გაზრდა შესაძლებელია პარალელური ალგორითმების გამოყენების გზით. მონაცემების ერთმანეთის მიმდევრობით დამუშავების ნაცვლად, პარალელური ალგორითმები საშუალებას იძლევა გამოთვლები ჩატარდეს პარალელურად და შესრულდეს ერთდროულად რამდენიმე ბირთვზე ან პროცესორზე. მაგალითად, შერწყმის დახარისხება შეიძლება პარალელურად დავყოთ მასივის ნაწილებად და შერწყმის ოპერაციები შევასრულოთ პარალელურად. ეს საშუალებას მოგვცემს დავაჩქაროთ დახარისხების პროცესი, განსაკუთრებით დიდი რაოდენობით მონაცემებისათვის.

რეკურსიული და იტერაციული მეთოდების შედარება ალგორითმების მაგალითზე, მათი ანალიზი და დროის სირთულის შეფასება. ფიბონაჩის მიმდევრობა და მისი მნიშვნელობა მეცნიერებასა და პრაქტიკაში, მისი გამოთვლის რეკურსიული ალგორითმი. ალგორითმის გაუმჯობესების მეთოდები, ფიბონაჩის მიმდევრობის გამოთვლის არარეკურსიული ფორმულა. ფიბონაჩის კოდის პრაქტიკაში გამოყენების მაგალითები

რეკურსიული და იტერაციული მეთოდების შედარება ალგორითმების მაგალითზე, მათი ანალიზი და დროის სირთულის შეფასება.

პროგრამირების საფუძვლების შესწავლისას პირველი პროგრამა, რომელსაც ვწერთ არის: „გამარჯობა, სამყარო!“, შემდეგ ვეცნობით ცვლადებს, მეთოდებს/ფუნქციებს, პირობით და ციკლის ოპერატორებს. ამავე დროს გვიჩნდება სურვილი, დავწეროთ რაიმე მარტივი ფუნქცია: რიცხვის ფაქტორიალის ან ბინომიალური კოეფიციენტის გამოთვლის ფუნქცია. უმეტეს შემთხვევაში დამწყები პროგრამისტი ამ ფუნქციების აღწერისას გამოიყენებს იტერაციას, რადგან, დამწყები პროგრამისტების უმეტესობამ არ იცის, რომ ნებისმიერი იტერაციულად აღწერილი ფუნქციის განხორციელება შესაძლებელია რეკურსიულადაც.

რეკურსია ეწოდება მონაცემთა დამუშავების ისეთ მეთოდს, რომლის დროსაც პროგრამა ან ფუნქცია უშუალოდ იძახებს საკუთარ თავს. ფუნქციას, რომელიც იძახებს საკუთარ თავს, ეწოდება რეკურსიული ფუნქცია ან უბრალოდ რეკურსია.

ყველაზე პოპულარული კითხვა რეკურსიულ ფუნქციებთან დაკავშირებით არის შემდეგი სახის: „რატომ გამოვიყენოთ რეკურსიული ფუნქცია, როდესაც დავალება შეიძლება შესრულდეს იტერაციის გამოყენებით?“.

გამოდის, რომ თურმე ყოველთვის შეგვიძლია გადავჭრათ რეკურსიული პრობლემა იტერაციულით.

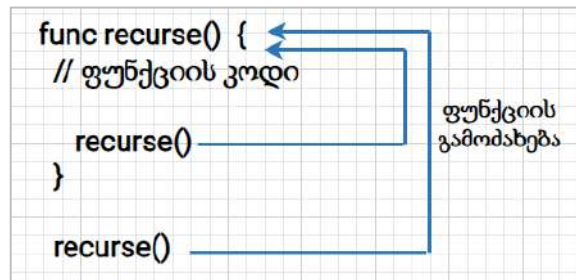
ვთქვათ, მოცემული გვაქვს შემდეგი სახის ფუნქცია:

```
func recurse()
{
    ...
    recurse()
    ...
}
recurse()
```

აქ ფუნქცია `recurse()` დაუსრულებლივ იძახებს თავის თავს. ნახატ 5.1.-ზე ნაჩვენებია `recurse()` ფუნქციის მუშაობის პრინციპი.

თუ რეკურსიულ ფუნქციას არ გააჩნია პირობა, ის უსასრულოდ იმუშავებს. რეკურსიული ფუნქციის უსასრულოდ მუშაობა გამოიწვევს სტეკის გადავსებას, რაც მიგვიყვანს შეცდომამდე. ამიტომ გამოსავალი არის რაიმე პირობის დაწერა, რომელიც უნდა შემოწმდეს ფუნქციის შესრულებისას, დასწყისშივე.

მაგალითად, რეკურსიის დასასრულებლად გამოვიყენოთ ოპერატორი `if-else`:



ნახ. 5.1. უსასრულო რეკურსია

რეკურსიულ ფუნქციას აქვს ორი ნაწილი:

1. რეკურსიის გაგრძელების (გამეორების) პირობა (რეკურსიის ბიჯი);
2. რეკურსიის დასრულების პირობა ანუ საბაზო პირობა.

თუ დაგვავიწყდება რომელიმე ნაწილის აღწერა, მაშინ კოდი შესრულდება უსასრულოდ, რაც გამოიწვევს სტეკის მეხსიერების გადავსებას და მიგვიყვანს შეცდომასთან.

მაგალითად:

```
func recurse() {  
    if (პირობა) { // რეკურსიული გამოძახების შეწყვეტა (რეკურსიიდან გამოსვლა)  
        recurse()  
    }  
    else { // რეკურსიის გამოძახება  
        recurse()  
    }  
}  
// ფუნქციის გამოძახება  
recurse()
```

განვიხილოთ კონკრეტული მაგალითი, კერძოდ, დავბეჭდოთ ციფრები 3-დან 0-მდე უკუ მიმდევრობით:

```
func countDown(number: Int) {  
    print(number) // ციფრის დაბეჭდვა
```

```
// რეკურსიის შეწყვეტის პირობა
if number == 0 {
    print("Countdown Stops")
}

else { // რეკურსიის გამოძახების პირობა
    countDown(number: number - 1) // ციფრის შემცირება
}
}

print("Countdown:")
countDown(number:3)
```

შესრულების შედეგი:

Countdown:

3

2

1

0

Countdown Stops

კოდში შევექმნით რეკურსიული ფუნქცია `countDown()`, რომელიც იძახებს საკუთარ თავს მანამდე, სანამ არ მიიღებს მნიშვნელობას 0-ს, როდესაც ცვლადი `number` გაუტოლდება 0-ს, შემოწმების პირობა `if` რეკურსიის გამოძახებას დაასრულებს ანუ შეწყვეტს.

ახლა კი განვიხილოთ პროგრამის შესრულების ბიჯი:

იტერაცია	ფუნქციის გამოძახება	შედეგი	პირობა: <code>number == 0?</code>
1	<code>countDown(3)</code>	3	false
2	<code>countDown(2)</code>	2	false
3	<code>countDown(1)</code>	1	false
4	<code>countDown(0)</code>	0	true (ფუნქციის გამოძახების შეჩერება)

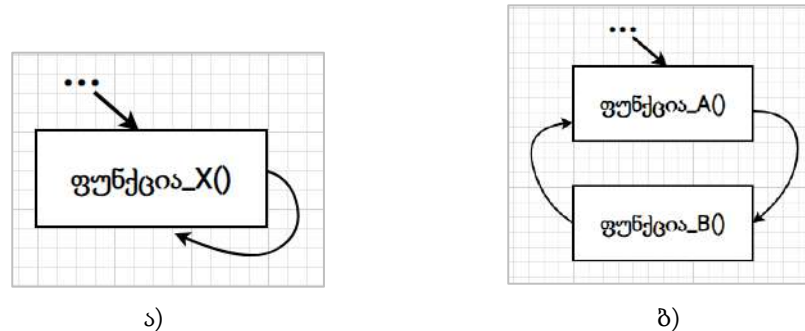
რეკურსიული ფუნქციის დამუშავების დროს ხდება მისი განმეორებითი გამოძახება პირდაპირ ან ირიბად სხვა ფუნქციების გამოძახების ხარჯზე.

პირდაპირია რეკურსია, როდესაც მეთოდი იძახებს საკუთარ თავს (ნახ. 5.2.ა).

ირიბია რეკურსია, როდესაც ერთი მეთოდი იძახებს მეორე მეთოდს, მეორე მეთოდი იძახებს მესამე მეთოდს და მესამე მეთოდი კი იძახებს, ისევ პირველ მეთოდს (ნახ. 5.2.ბ).

რეკურსიული ფუნქცია ამოცანას შლის მარტივ ნაწილებად ანუ ხდება დეკომპოზიცია. რეკურსია მოქმედებს პრინციპით „დაყავი და იბატონე“. რეკურსიული ფუნქციის დასრულება ხდება ამოცანის საბაზო ანუ ყველაზე მარტივ დონემდე დაყვანით.

იტერაცია არის მონაცემთა დამუშავების ისეთი მეთოდი, რომლის დროსაც ზოგიერთი მოქმედება მრავალჯერ მეორდება, ამასთანავე არ ხდება ფუნქციების რეკურსიული გამოძახებები.



ნახ. 5.2. რეკურსიის ტიპები: ა) პირდაპირი რეკურსია ბ) ირიბი რეკურსია

ამოცანის რეკურსიულად გადაწყვეტა მოითხოვს გაცილებით დიდ რესურსს კომპიუტერისგან, ვიდრე ამას საჭიროებს ტრადიციული (იტერაციული) მიდგომა, თუმცა უნდა აღინიშნოს რომ ზოგიერთი ამოცანის გადაწყვეტისას რეკურსიული მიდგომა გაცილებით უკეთესია, ვიდრე იტერაციული. ასეთი ამოცანათა რიცხვს მიეკუთვნება ამოცანა ჰანოის კოშკების შესახებ, რომლის იტერაციული გადაწყვეტა ძალიან დიდ სირთულეს წარმოადგენს მაშინ, როდესაც რეკურსიით ყველაფერი მარტივად და გასაგებად წყდება.

იტერაციული ფუნქციები (ისინი, რომლებიც იყენებენ for ან while) თითქმის ყოველთვის უფრო ეფექტურია, ვიდრე მათი რეკურსიული ანალოგები. ეს გამოწვეულია იმით, რომ ფუნქციის ყოველი გამოძახებისას, გარკვეული რესურსი იხარჯება სტეკიდან ელემენტების დამატებასა და ამოღებაზე. იტერაციული ფუნქციები კი მოიხმარს ასეთი სახის გაცილებით ნაკლებ რესურსებს.

უფრო კონკრეტულად, იტერაცია ხელს უწყობს პროგრამის შესრულების კონტროლს. მაგალითად, თუ ციკლს მივუთითებთ „მოქმედება შეასრულე 10-ჯერ“, მაშინ ის ზუსტად 10-ჯერ შეასრულებს რაიმე მოქმედებას და პირიქით: თუ არ ვიცით რამდენი იტერაცია გვჭირდება, მაშინ ციკლი შეამოწმებს შეწყვეტის პირობას თითოეულ ჯერზე. როდესაც პროგრამისტის მიერ მითითებული პირობის შესრულება მოხდება, ის დასრულდება. ასე რომ, არ იქნება საჭირო იტერაციის ხელით დათვლა.

თუ რეკურსიული ალგორითმის განხორციელება უფრო ადვილია, მაშინ მნიშვნელობა აქვს გამოვიყენოთ რეკურსია და ამის შემდეგ კი მოვახდინოთ კოდის ოპტიმიზაცია იტერაციული ალგორითმით.

წესი: რეკურსიის ნაცვლად რეკომენდირებულია იტერაციის გამოყენება, მაგრამ მხოლოდ იმ შემთხვევებში, როდესაც ეს მართლაც უფრო პრაქტიკულია.

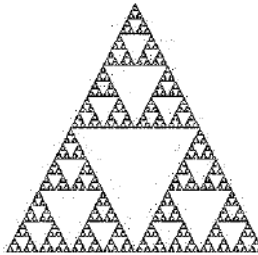
თეორემა: ნებისმიერი ალგორითმი, რომელიც რეკურსიული ფორმით არის განხორციელებული, შეიძლება იტერაციული ფორმით იყოს გადაწერილი და პირიქით.

რეკურსიის ერთ-ერთი მარტივი მაგალითია გრძელ დერეფანში ერთმანეთის პირისპირ მდგარი სარკეები. თუ ერთ სარკეში ჩაიხედავთ, იქ გამოჩნდება მეორე სარკის ანარეკლი, მეორე სარკეში კი პირველი სარკის ანარეკლია და ა. შ. (ნახ. 5.3.).



ნახ. 5.3. უსასრულო რეკურსია

რეკურსიის მეორე, უფრო აკადემიური მაგალითი არის ფრაქტალი. ფრაქტალი ასევე ცნობილია სერპინსკის სამკუთხედის სახელით. სერპინსკის სამკუთხედში ფიგურის ერთი მცირე ნაწილი ამავე დროს არის მთელი ფიგურა, სერპინსკის სამკუთხედი შეგვიძლია მივაკუთვნოთ მრავალჯერად რეკურსიას (ნახ. 5.4.).



ნახ. 5.4. სერპინსკის სამკუთხედი. მრავალჯერადი რეკურსია

სერპინსკის სამკუთხედი შედგება ზუსტად იგივე სამი ცალი სამკუთხედისგან.

ზოგიერთი ამოცანის გადაჭრა შეიძლება განხორციელდეს ამოცანის რამოდენიმე მარტივ ბიჯებად და ვარიანტებად დაყოფის სახით. სწორედ ასეთი ხასიათის ამოცანებს უწოდებენ რეკურსიულს - როცა პირობის ნაწილი მიმართავს მთლიან ამოცანას (ან მის მსგავსს).

განვიხილოთ რიცხვის ახარისხების x^n ფუნქცია რეკურსიული სახით, სადაც x - მთელი რიცხვია, ხოლო n - მთელი არაუარყოფითი რიცხვი.

რიცხვის ახარისხების ფორმულას აქვს შემდეგი სახე (1):

$$x^n = \begin{cases} 1, & \text{თუ } n = 0 \\ x^{n-1} * x, & \text{თუ } n \geq 1 \end{cases} \quad (1)$$

ალგორითმულ ენაზე რიცხვის ახარისხება ჩაიწერება შემდეგი სახით:

function Deg(x, n : integer) : longinteger;

დასაწყისი

if n = 0 მაშინ Deg = 1

წინააღმდეგ შემთხვევაში Deg = Deg(x, n-1)*x;

დასასრული;

მაგალითად, ვიპოვოთ 2^5 -ის მნიშვნელობა:

$$2^5 = 2 * 2 * 2 * 2 * 2$$

1-ლი ბიჯი: 2

მე-2 ბიჯი: $2 * 2$

მე-3 ბიჯი: $2 * 2 * 2$

მე-4 ბიჯი: $2 * 2 * 2 * 2$

მე-5 ბიჯი: $2 * 2 * 2 * 2 * 2$ // რომელი ბიჯიც გავიარეთ, იმდენივე გამრავლება გვაქვს

მეორენაირად რიცხვის ახარისხი შეგვიძლია განვიხილოთ, როგორც 2-ზე გამრავლების თანმიმდევრობა (იხ. ნახ. 5.5.):

$$2^5 = (((2 * 2) * 2) * 2) * 2$$

1-ლი ბიჯი: 2

მე-2 ბიჯი: $2 * 2$ (პირველი ბიჯის შედეგი $* 2$)

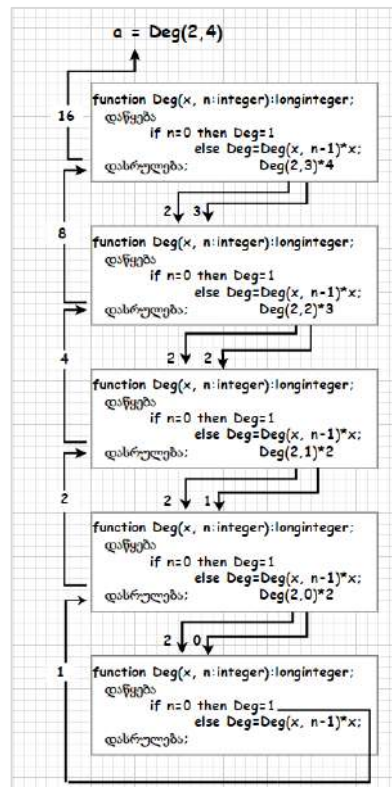
მე-3 ბიჯი: $4 * 2$ (მეორე ბიჯის შედეგი $* 2$)

მე-4 ბიჯი: $8 * 2$ (მესამე ბიჯის შედეგი $* 2$)

მე-5 ბიჯი: $16 * 2$ (მეოთხე ბიჯის შედეგი $* 2$) // ახალი შედეგი = წინა შედეგი $* 2$ -ზე

წარმოდგენილი რიცხვის ახარისხება არის მხოლოდ წინა შედეგის გამრავლება 2-ზე ანუ $2^n = 2^{n-1} * 2$

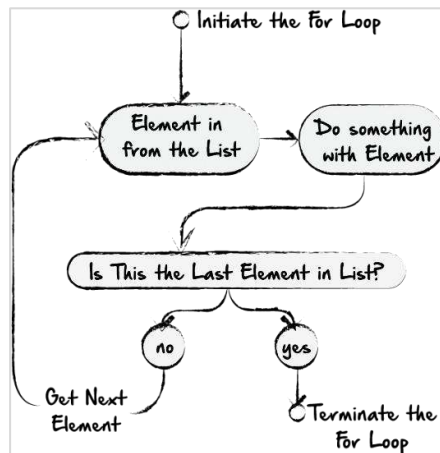
ნახ. 5.5.-დან ჩანს, რომ ნებისმიერ რეკურსიულ ქვეპროგრამას უნდა ჰქონდეს ერთი არარეკურსიული შტო (საბაზო პირობა), ჩვენს შემთხვევაში ეს არის: *if* $n = 0$, მაშინ $Deg = 1$



ნახ. 5.5. რიცხვი ახარისხების ვიზუალიზაცია. პირდაპირი და ირიბი რეკურსია

როგორც კი ახარისხების ფუნქციის მუშაობა საბაზისო პირობასთან მივა, ის გაჩერდება. საბაზო პირობის გარეშე, ნებისმიერი რეკურსიული ფუნქცია განუსაზღვრელი დროის განმავლობაში გაგრძელდება, რადგან ის თავის თავს დაუსრულებლივ გამოიძახებს.

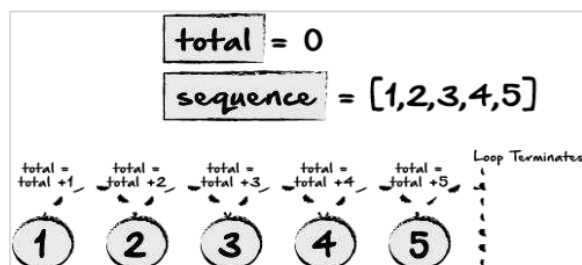
ახლა კი გადავიდეთ რეკურსიული და ციკლური სტრუქტურის შედარებით ანალიზზე. პროგრამირების მეორე ფუნდამენტური ინსტრუმენტი არის ციკლი. არსებობს სხვადასხვა ტიპის ციკლები, მაგრამ თითქმის ყველა მათგანი ასრულებს ერთ ძირითად ფუნქციას: გარკვეულ მოქმედებებს ახორციელებს განმეორებით მონაცემების ანალიზის ან მართვის მიზნით (იხ. ნახ. 5.6.). რეკურსია, მონაცემების ანალიზისა და ქცევის ისეთივე გავრცელებული მეთოდია, როგორც ციკლი, მაგრამ, როგორც წესი, ნაკლებად გასაგები და ხშირად უფრო რთული. თითქმის ყველა რეკურსიული ფუნქცია შეიძლება გადაწერილი იყოს ციკლური ფორმით და პირიქით. მიუხედავად ამისა, თითოეულ ფუნქციას აქვს თავისი უპირატესობა და ნაკლოვანება.



ნახ. 5.6. სტრუქტურა ციკლი (იტერაცია)

ციკლი for გამოიყენება მონაცემთა თანამიმდევრობის გადასარჩევად და მივალწევთ რა თანამიმდევრობის ბოლოს, ციკლიც დასრულდება.

მაგალითად, გვინდა შევკრიბოთ 1-დან 5-მდე ციფრების თანამიმდევრობა და მივიღოთ ჯამის მნიშვნელობა ციკლის საშუალებით. ვიზუალურად ნაჩვენებია ნახ. 5.7.-ზე.



ნახ. 5.7. სტრუქტურა ციკლი (იტერაცია), თანამიმდევრობის შეკრება/შეჯამება

შეგვიძლია $1 + 2 + 3 + 4 + 5$ თანამიმდევრობა უბრალოდ შევკრიბოთ, მაგრამ ფუნქციის გამოყენება გაცილებით უფრო მოსახერხებელია, რადგან ფუნქცია შეგვიძლია გამოვიძახოთ რამდენჯერაც გვინდა იმდენჯერ (არა ერთხელ, არამედ მრავალჯერ) და შევცვალოთ

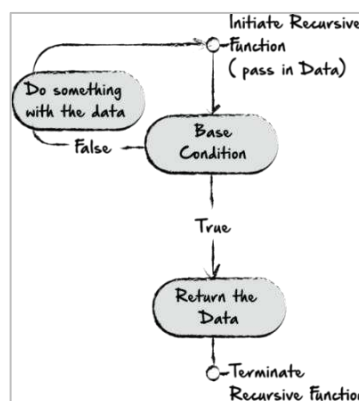
თანამიმდევრობის რაოდენობის მნიშვნელობა, მაშინაც კი, თუ ისინი წინასწარ არ არის ცნობილი. იმისათვის, რომ ციკლმა იმუშაოს, უნდა დავიმახსოვროთ ყველა ციფრის/რიცხვის თანამიმდევრობა, რათა შევძლოთ მათი გადარჩევა და საბოლოო დაჯამებული მნიშვნელობის მიღება. შეკრების ფუნქციას ექნება შემდეგი სახე (ციკლი, იტერაცია):

```
def getTotal(n):
    total = 0
    for number in list(range(n+1)):
        print number
        total = total + number
    return total

print getTotal(5)
```

როდესაც ფუნქცია იმახებს საკუთარ თავს, მაშინ საქმე გვაქვს რეკურსიასთან. რეკურსიის ერთ-ერთი მთავარი განსხვავება ციკლისგან არის რეკურსიული ფუნქციის დასასრული.

ზემოთ მოცემულ მაგალითში ციკლის მუშაობა მთავრდება თანამიმდევრობის ბოლოს, მაგრამ რეკურსიული ფუნქციის მუშაობა შეიძლება გაგრძელდეს უსასრულოდ. ამიტომ, რეკურსიულ ფუნქციას აქვს ე. წ. საბაზისო (ძირითადი) პირობა, რომელიც განსაზღვრავს, როდის უნდა დასრულდეს რეკურსია (ნახ. 5.8.).



ნახ. 5.8. სტრუქტურა რეკურსია

შეკრების მაგალითი განვიხილოთ რეკურსიული სახით. ნახ. 5.9.-ზე ვიზუალურად ნაჩვენებია ნახატზე თანამიმდევრობის შეკრება რეკურსიულად.

	total = 0 number = 5	total = 5 number = 4	total = 9 number = 3	total = 12 number = 2	total = 14 number = 1	total = 15 number = 0
What parameters are passed in	total = 0 number = 5	total = 5 number = 4	total = 9 number = 3	total = 12 number = 2	total = 14 number = 1	total = 15 number = 0
What the function does with the data passed in	is the number 0? no Okay, then add the number to the total. Subtract 1 from the number and run it again	is the number 0? no Okay, then add the number to the total. Subtract 1 from the number and run it again	is the number 0? no Okay, then add the number to the total. Subtract 1 from the number and run it again	is the number 0? no Okay, then add the number to the total. Subtract 1 from the number and run it again	is the number 0? no Okay, then add the number to the total. Subtract 1 from the number and run it again	is the number 0? yes. Okay, return the total
What the function returns	calls the function with these parameters total : 5 number : 4	calls the function with these parameters total : 9 number : 3	calls the function with these parameters total : 12 number : 2	calls the function with these parameters total : 14 number : 1	calls the function with these parameters total : 15 number : 0	15

ნახ. 5.9. სტრუქტურა რეკურსია, თანამიმდევრობის შეკრება/შეჯამება

ნახ. 5.9.-დან ჩანს, რომ ყოველ ჯერზე ფუნქცია ან იძახებს საკუთარ თავს ახალი შესატანი მონაცემით ან აბრუნებს მნიშვნელობას. შეკრების რეკურსიულ ფუნქციას ექნება შემდეგი სახე:

```
def getTotal(n, total)

    print n

    if n == 0 # base condition

        return total

    else

        return getTotal(n-1, (total+(n)))

print getTotal(5, 0)
```

ფუნქციას გადავცემთ ორ მნიშვნელობას: საწყისს და საბოლოოს. ფუნქციის პირველი გამოძახებისას, საბოლოო მნიშვნელობა არის 0, ხოლო საწყისი 5. ვამოწმებთ, არის თუ არა საწყისი მნიშვნელობა 0-ის ტოლი. თუ არ არის, მაშინ ფუნქციას ისევ გამოვიძახებთ, მაგრამ ამჯერად შევცვლით შესავალ მნიშვნელობას 5-1 და 0+5 მნიშვნელობებით და ვიმეორებთ ამ პროცესს მანამ, სანამ n არ გახდება 0. ამ პირობის შესრულების შემდეგ, ვაბრუნებთ საბოლოო მნიშვნელობას 15-ს.

ახლა განვიხილოთ რთული პროცენტის გამოთვლის რეკურსიული და იტერაციული შემთხვევები. განსაზღვროთ სესხის ან ინვესტიციის ღირებულება რთული პროცენტით.

რთული პროცენტი არის დეპოზიტზე, სესხზე ან სხვა ფინანსურ აქტივზე პროცენტის (სარგებლის) დარიცხვის მეთოდი. პროცენტის დარიცხვა შეიძლება მოხდეს ნებისმიერი გრაფიკით, ყოველდღიურად, ყოველთვიურად ან ყოველწლიურად. მაგალითად, თუ თქვენს ანგარიშზე გაქვთ \$10.000, რომლის წლიური საპროცენტო სარგებელია 4% და თქვენ იყენებთ რთულ პროცენტს ხუთი წლის განმავლობაში, თქვენი საბოლოო თანხა იქნება \$12 166.53, რაც \$166.53-ით მეტია იმ თანხასთან შედარებით, რასაც მიიღებდით რთული პროცენტის გამოყენების გარეშე. ნახ. 5.10.-ზე მოცემულია რთული პროცენტის გამოთვლის ფორმულა:

$$A = P \left(1 + \frac{r}{n} \right)^{nt}$$

Labels in the image: Total Amount (A), Interest Rate (r), Principal (P), Number of Times Compounded Per Year (n), time(years) (t).

ნახ. 5.10. რთული პროცენტის გამოთვლის ფორმულა

რთული პროცენტის ფორმულაში:

A არის თანხის მთლიანი რაოდენობა პერიოდის დასასრულს;

P არის ინვესტირებული ან ნასესხები ძირითადი თანხა;

r არის წლიური საპროცენტო განაკვეთი;

n არის კონკრეტულ დროის პერიოდში პროცენტის დარიცხვების რაოდენობა;

t არის გასული დროის პერიოდების რაოდენობა.

ამ გზით ჩვენ შეგვიძლია გამოვითვალოთ მთელი თანხა ერთდროულად, მაგრამ ამის ნაცვლად, გამოვიყენოთ ციკლი ან რეკურსია. ასეთ შემთხვევაში დროის ცვლადი ($n * t$) დამუშავდება იტერაციულად. შემოვიტანოთ ცვლადები, რომლებშიც შევინახავთ პირველად მნიშვნელობებს და გამოვიყენებთ ორივე მეთოდს:

```
durationInYears = 10;
```

```
interestRate = 0.06;
```

```
compoundedPerYear = 12;
```

```
principalAmount = 4000;
```

განვიხილოთ რთული საპროცენტო განაკვეთის გამოთვლა იტერაციულად.

მოდით ახლავე დავიანგარიშოთ გადახდების საერთო რაოდენობა, რათა გამარტივდეს რთული პროცენტის იტერაციული მეთოდით გამოთვლა. რადგან გადახდა ხდება ყოველთვე და წლების რაოდენობა 10-ის ტოლია, შედეგი იქნება 120 ანუ $10 * 12$.

გამოვითვალოთ ყოველთვიური პროცენტი და ძირითად თანხას დავამატოთ ყოველი იტერაციის (გამეორების) შედეგი:

```
def compoundInterest(principal, compounded, duration, rate):
```

```
    totalCompounded = duration * compounded
```

```
    for i in range(1, (totalCompounded+1)):
```

```
        principal = principal*(1+(rate/compounded))
```

```
    return principal
```

```
print(compoundInterest(principalAmount,compoundedPerYear,durationInYears,interestRate))
```

წინა მაგალითისგან განსხვავებით, სადაც იტერაციის რაოდენობა იყო 5, ამ შემთხვევაში კი იტერაციების რაოდენობა გაიზარდა 120-მდე.

გამოთვლის შედეგია: 7277.58693613

განვიხილოთ რთული საპროცენტო განაკვეთის გამოთვლა რეკურსიული მეთოდით.

იტერაციული მეთოდის დროს მონაცემთა თანმიმდევრობა 120-ს უდრის, რაც ასახავს ძირითად თანხის ხელახალი გაანგარიშების რაოდენობას. ციკლი შეწყდება, როდესაც სქემა დასრულდება. რეკურსიული მეთოდი კი საშუალებას გვაძლევს, იგივე გავაკეთოთ ანუ შემოვიტანოთ მთვლელის ინიციალიზება და ორი პირობა:

პირველი პირობა: მთვლელი = 0

გამოვითვალოთ რთული პროცენტი.

საერთო ჯამს ემატება გამოთვლის შედეგი. მრიცხველის მნიშვნელობა შევამცირეთ 1-ით. გავიმეორეთ იგივე მოქმედება: მთვლელს და საერთო ჯამურ შედეგს გადავცეთ ახალი მნიშვნელობა.

მეორე პირობა: მთვლელი = 0;

საერთო ჯამური მნიშვნელობის დაბრუნება.

წინა მაგალითში ფუნქციის ციკლი იწყებოდა 5-დან და მთავრდებოდა 0-ის დროს. იგივე ხდება ამ შემთხვევაში რეკურსიულად, მხოლოდ საწყისი მნიშვნელობა არის 120.

```
def compoundRecursion(principal, compounded, duration, rate, numberOfRecursions):
```

```
    if numberOfRecursions == 0:
```

```
        totalDuration = compounded * duration
```

```
    elif numberOfRecursions != 0:
```

```
        totalDuration = duration
```

```
    if duration == 0:
```

```
        return principal
```

```
    else:
```

```
        newDuration = totalDuration - 1
```

```
        amount = principal*(1+(rate/composed))
```

```
        return compoundRecursion(amount, compounded, newDuration, rate, 1)
```

```
print(compoundRecursion(principalAmount,compoundedPerYear,durationInYears, interestRate, 0))
```

აქ ჩვენ ან კვლავ ვიძახებთ ფუნქციას ან ვაბრუნებთ განახლებულ დაჯამების შედეგს. ყოველ ჯერზე, როდესაც ჩვენ ვიძახებთ ფუნქციას, მთვლელის მნიშვნელობა მცირდება ერთით. საბოლოო დაჯამებული მნიშვნელობის დაბრუნება ხდება მაშინ, როდესაც მთვლელი გაუტოლდება ნულს. განხილული მაგალითი გვიჩვენებს, რომ რთული პროცენტის გამოთვლისას რეკურსია ბევრად უკეთ მუშაობს, ვიდრე ციკლი.

ახლა განვიხილოთ კიდეც ერთი მაგალითი ციკლისა და რეკურსიის გამოყენებით, რათა დავინახოთ განსხვავება ამ ორ სტრუქტურას შორის. დავწეროთ ფაქტორიალის გამოთვლის ფუნქცია ორივე სტრუქტურით.

კითხვაზე: „ამა თუ იმ საკითხის გადასაწყვეტად რა გამოვიყენოთ რეკურსია თუ ციკლი?“ - დამოკიდებულია იმაზე თუ როგორი სახის ამოცანა არის გადასაწყვეტი.

ფაქტორიალის გამოთვლა ციკლით:

```
function factorial(n) {
```

```
    let result = 1
```

```
    for (let i = 0; i < n; i++) {
```

```
        result *= i + 1    }
```

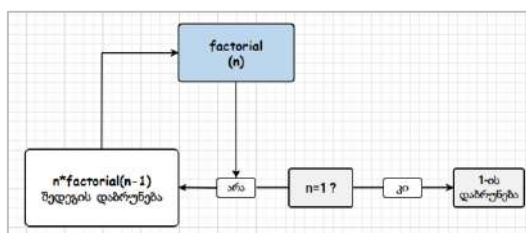
```
return result
```

```
} console.log(factorial(5)) // შედეგი 120
```

ფუნქციაში `factorial()` ვიყენებთ ციკლს. ფუნქციაში თითოეული რიცხვი მრავლდება წინა გამრავლების შედეგზე. იგივე შეგვიძლია რეკურსიულად.

ფაქტორიალის რეკურსიულ ფუნქციაში, პირველ რიგში აღწერთ საბაზისო შემთხვევას და მხოლოდ შემდეგ გადავალთ განმეორებად მოქმედებებზე (იტერაციაზე).

რეკურსიასთან მუშაობისას გვაქვს შემდეგი წესი: პირველ რიგში უნდა აღწეროთ საბაზისო შემთხვევა (early return) და მხოლოდ ამის შემდეგ შეგვიძლია აღვწეროთ ყველაფერი დანარჩენი. ფაქტორიალის გამოთვლის რეკურსიული ალგორითმი ბლოკ-სქემის სახით მოცემულია ნახ. 5.11-ზე.



ნახ. 5.11. ფაქტორიალის რეკურსიულად გამოთვლის ბლოკ-სქემა

ფაქტორიალის გამოთვლის რეკურსიული ფუნქცია:

```
function factorial(n) {
```

```
    // თუ გვინდა ვიპივოთ 1-ის ფაქტორიალი, მაშინ დავაბრუნოთ 1.
```

```
    // საბაზო პირობა.
```

```
    if (n <= 1) { return 1 }
```

```
    // დანარჩენ შემთხვევაში ვაბრუნებთ n-ის ნამრავლს წინა რიცხვის ფაქტორიალზე
```

```
    // ამის შედეგად ჩვენ n-დან მივალთ 1-მდე, თითოეული რიცხვის გადარჩევით
```

```
    return n * factorial(n - 1)
```

```
}
```

```
console.log(factorial(5)) // შედეგი 120
```

როგორც ვხედავთ, გარდა იმისა, რომ ფუნქცია შესამჩნევად მოკლე გახდა, იგი ახლა პირდაპირ გამოხატავს ფაქტორიალის გამოთვლის მათემატიკურ არსს (ნახ. 5.12.).

$$\begin{aligned} \text{factorial}(5) &= \\ &= 5 * \text{factorial}(4) = \\ &= 5 * 4 * \text{factorial}(3) = \\ &= 5 * 4 * 3 * \text{factorial}(2) = \\ &= 5 * 4 * 3 * 2 * \text{factorial}(1) = \\ &= 5 * 4 * 3 * 2 * 1 = 120 \end{aligned}$$

ნახ. 5.12. ფაქტორიალის გამოთვლის მათემატიკური არსი

5-ის ფაქტორიალის გამოთვლის პროცესი შედგება 4 გამოძახებული factorial() ფუნქციისაგან (factorial()). განვიხილოთ თითოეული გამოძახება:

// თავდაპირველად მივდივართ გამოძახებების „სიღრმისკენ“

პირველი გამოძახება:

```
factorial(5) { // ცვლადი n ხდება 5-ის ტოლი;  
  n - 1 => 4; ფუნქცია აბრუნებს 5 * factorial(4);
```

მეორე გამოძახება:

```
  factorial(4) {  
    n => 4  
    n - 1 => 3  
    return 4 * factorial(3)
```

მესამე გამოძახება:

```
    factorial(3) {  
      n => 3  
      n - 1 => 2  
      return 3 * factorial(2)
```

მეოთხე გამოძახება:

```
      factorial(2) {  
        n => 2  
        n - 1 => 1  
        return 2 * factorial(1)
```

ბოლო გამოძახება, საბაზო პირობა:

```
      factorial(1)  
      {  
        n => 1  
        // რადგან ვართ საბაზო პირობასთან, ვაბრუნებთ 1-ს  
        // საბაზო პირობის შემდეგ „მივდივართ ზევით“  
      }  
      // ამ დროს factorial(1)-ის შედეგი ხდება 1-ის ტოლი  
      ვაბრუნებთ return 2 * 1  
    }  
    შედეგი factorial(2) => 2  
    return 3 * 2  
  }
```

```

factorial(3) => 6
return 4 * 6
}
factorial(4) => 24
return 5 * 24
}
// ყოველი return-ის შემდეგ ხდება შესაბამისი ფუნქციის ხილვადობის უბნის გასუფთავება და
// გამოძახების შედეგი ხდება კონკრეტული რიცხვის ტოლი

```

რეკურსიის დადებითი მხარე:

რეკურსიული კოდი არის შედარებით მოკლე და ადვილად გასაგები.

რეკურსია საჭირო მონაცემთა სტრუქტურებთან და რთული ტიპის ალგორითმებთან დაკავშირებული პრობლემების გადასაჭრელად.

რეკურსიის უარყოფითი მხარე:

რეკურსია იკავებს უფრო მეტ მეხსიერებას, ვიდრე იტერაციული პროგრამა.

რეკურსია იყენებს უფრო მეტ პროცესორულ დროს.

განვიხილოთ რეკურსიული სტრუქტურის კიდევ ერთი მაგალითი. როგორც უკვე აღვნიშნეთ, პროგრამირებაში რეკურსიას ყველაზე ხშირად უწოდებენ ფუნქციას, რომელიც იძახებს თავს თავს, მაგრამ რეკურსიული ფუნქციების გარდა, არსებობს რეკურსიული მონაცემთა სტრუქტურებიც.

მონაცემთა სტრუქტურა არის კონტეინერი, რომელიც ინახავს მონაცემებს გარკვეულ ფორმატში. ეს კონტეინერი განსაზღვრავს, თუ როგორ შეიძლება გარე სამყარომ მონაცემები წაიკითხოს ან შეცვალოს. მონაცემთა სტრუქტურები, რომლის ნაწილიც მოიცავს თავისივე მსგავს სტრუქტურებს, ეწოდებათ რეკურსიული. ასეთი ტიპის სტრუქტურებთან ციკლის გამოყენება არ არის მოსახერხებელი. თუ რატომ, განვიხილოთ ერთ-ერთი რეკურსიული მონაცემთა სტრუქტურა - ხე. ხე - არის ისეთი სტრუქტურა, რომელშიც თითოეულ კვანძს შეიძლება ჰქონდეს რამდენიმე შვილობილი კვანძი.

ვნახოთ, თუ როგორ გამოიყურება DOM (Document Object Model) ხე:

```

<!-- ასეთი სახის დოკუმენტისთვის: -->
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello</title>
  </head>
  <body>

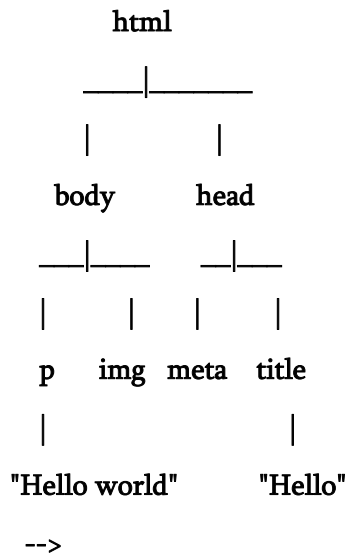
```

```

<p class="text">Hello world</p>

</body>
</html>
<!-- ...მიიღება ქვემოთ ნაჩვენები ხე:

```



ნებისმიერ ხეს აქვს კვანძი - ფესვი, საიდანაც წარმოიქმნება სხვა კვანძები. DOM-ხის ფესვი არის html ელემენტი.

ხესთან მუშაობა ციკლის გამოყენებით (იტერაციულად) საკმაოდ მოუხერხებელია. წარმოვიდგინოთ, რომ გვსურს მივიღოთ ხის ყველა ელემენტის სახელი. შეგვიძლია ციკლის დახმარებით გავიაროთ 1-ლი ან მე-2 დონე, მაგრამ შემდეგ უნდა ვიფიქროთ იმაზე, თუ როგორ განვსაზღვროთ სად გავიარეთ, სად არ გავვივლია ჯერ და სად წავიდეთ შემდეგ. რეკურსიით კი, ხის შემოვლა ხდება შედარებით უფრო მარტივად.

ხის რეკურსიული შემოვლა - შესაძლებელია მოვიფიქროთ ხის შემოვლის რეკურსიული ალგორითმი:

საბაზო შემთხვევა: ხეს საერთოდ არ აქვს შვილობილი კვანძები ან ჩვენ შემოვლარეთ ყველა მათგანს - ვაბრუნებთ ელემენტის სახელს;

განმეორებითი ოპერაციები: რეკურსიულად შემოვივლით ხის თითოეულ შვილობილ ელემენტს, მათი სახელების შეგროვებით.

ამრიგად, დაწყებული <html> ელემენტით, გავვივლით ხის თითოეულ ელემენტს და ჩავწერთ მათ სახელებს.

ახლა კი დავწეროთ აღწერილი ალგორითმის ფსევდოკოდი:

```

function nameOf(element) {
  /* ელემენტის დასახელების დაბრუნება */
}

function eachChild(element, func)

```

```

{
  /* element ელემენტის თითოეული შვილობილ კვანძზე func ფუნქციის გამოძახება */ }
function childrenOf(node)
{
  /* მიმდინარე კვანძის შვილობილი კვანძის დაბრუნება. თუ ასეთი არ არის, მაშინ აბრუნებს null */ }
function traverse(treeNode)
{
  // თუ არ არსებობს შვილობილი კვანძები, მაშინ საქმე გვაქვს საბაზო შემთხვევასთან;
  // ვაბრუნებთ მასივს მიმდინარე ელემენტის სახელდებით;
  // მასივი საჭიროა მიმდევრობებისათვის, რადგან შემდეგ დავაბრუნოთ მასივი სხვა
  // კვანძების სახელდებით. ყოველ ჯერზე, რომ არ მოგვიწიოს შედეგის ტიპის შემოწმება,
  // აუცილებლად ყოველთვის უნდა დავაბრუნოთ მასივი.
  if (!childrenOf(treeNode))
  {
    return [nameOf(treeNode)]
  }
  // თუ არ არის საბაზო შემთხვევა, მაშინ გადავურბინოთ თითოეულ შვილობილ კვანძს
  // და თითოეულისათვის გამოვიძახოთ traverse ფუნქცია.
  // nameArrays-ში მივიღებთ დასახელებების სიას
  const nameArrays = eachChild(treeNode, traverse)
  const names = nameArrays.flat()
  return names
}
// დავიწყეთ რა შემოვლა ხის ფესვი კვანძიდან, მივიღეთ ყველა ელემენტის სახელდების სია
const names = traverse(rootNode)

```

ამა თუ იმ ამოცანის გადასაჭრელად, რომელიმე მიდგომის ამორჩევისას, შეგვიძლია გავითვალისწინოთ შემდეგი რეკომენდაციები:

რეკურსიული მიდგომა: მისაღებია n -ის მცირე მნიშვნელობებისთვის, მაგრამ დიდი მნიშვნელობებისთვის წამოიჭრება გამოთვლის სირთულის პრობლემა. მნიშვნელოვანია, რომ დაკვირვებით შეარჩიოთ ეს მეთოდი და შეაფასოთ მისი ეფექტურობა კონკრეტულ ამოცანასთან მიმართებით.

იტერაციული მიდგომა: იგი ხშირად ყველაზე ეფექტური მეთოდია და შესაბამისობაში მოდის n -ის მნიშვნელობების ფართო დიაპაზონისთვის და უზრუნველყოფს კოდის სტაბილურ მუშაობას.

ფიზონაჩის მიმდევრობა და მისი მნიშვნელობა მეცნიერებასა და პრაქტიკაში, მისი გამოთვლის რეკურსიული ალგორითმი. ალგორითმის გაუმჯობესების მეთოდები, ფიზონაჩის

მიმდევრობის გამოთვლის არარეკურსიული ფორმულა. ფიბონაჩის კოდის პრაქტიკაში გამოყენების მაგალითები.

სამყაროში, მისი ქაოსური და მრავალფეროვანი ბუნების მიუხედავად, ყველაფერი გარკვეული კანონზომიერებითაა აგებული. თუ შევეცდებით ყველაფერი ციფრებისა და რიცხვების სისტემაში გადავიყვანოთ, სამყაროს გენიალურობა დადის ერთ მარტივ 1.618 რიცხვამდე.

არსებობს ციფრებისა და რიცხვების ისეთი თანამიმდევრობა, რომელშიც მიმდევრობის პირველი ორი წევრი არის ან 1-ანი და 1-ანი ან 0-ანი და 1-ანი, ხოლო მიმდევრობის ყოველი მომდევნო წევრი წინა ორი წევრის ჯამს წარმოადგენს. ასეთ მიმდევრობას ფიბონაჩის მიმდევრობა ეწოდება: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711 და ა. შ. უსასრულობამდე.

ფიბონაჩის მიმდევრობა უსასრულოა, მაგრამ ზოგიერთ შემთხვევაში პირველი ათი რიცხვის (წევრის) გამოყენებაც საკმარისია (ცხრილი 5.1).

ცხრილი 5.1. ფიბონაჩის პირველი ათი რიცხვის მიმდევრობა

პირველი	მეორე	მესამე	მეოთხე	მეხუთე	მეექვსე	მეშვიდე	მერვე	მეცხრე	მეათე
0	1	1	2	3	5	8	13	21	34

ფიბონაჩის მიმდევრობაში ყოველი მომდევნო მნიშვნელობა მყარადაა დაკავშირებული მის წინა მნიშვნელობასთან, ვიდრე ჩვეულებრივ მიმდევრობებში.

ფიბონაჩის მიმდევრობას აქვს შემდეგი კანონზომიერება:

ფიბონაჩის მიმდევრობის ყოველი რიცხვი მის წინ მდგომი ორი რიცხვის ჯამის ტოლია. მაგალითად, $8 = 5 + 3$;

ფიბონაჩის მიმდევრობის ყოველი რიცხვის ფარდობა მის წინ მდგომ რიცხვთან დიდი მიახლოებით მიისწრაფის 1.618-ისკენ, მაგალითად: $55/34 = 1.6176.....$;

რიცხვს 1.618-ს უწოდებენ „ოქროს კვეთას“;

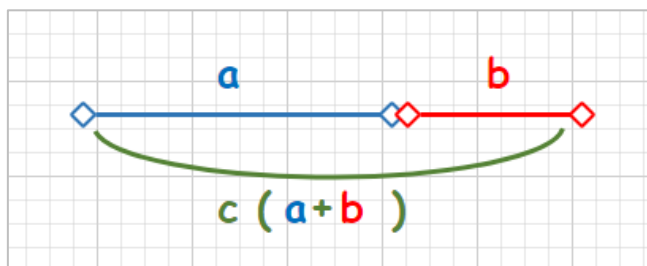
ფიბონაჩის მიმდევრობის ყოველი რიცხვის ფარდობა მის მომდევნო რიცხვთან დიდი მიახლოებით 0.617-ის ტოლია. მაგალითად, $55/89 = 0.6179.....$;

ოქროს კვეთა არის მთელი მონაკვეთის გაყოფა ისეთ ორ არატოლ ნაწილად, როდესაც მთელი ნაწილი ისე შეეფარდება დიდს, როგორც დიდი - მცირეს.

ალგებრულად ოქროს კვეთა დაიყვანება შემდეგი განტოლების ამოხსნამდე:

$$\frac{a+b}{a} = \frac{a}{b} = \varphi, \text{ სადა } \varphi \text{ ოქროს კვეთის რიცხვია.}$$

თუ ოქროს კვეთის თანაფარდობას მიახლოებით აღწერთ, მაშინ ის შეიძლება წარმოვიდგინოთ როგორც პროპორციული დაყოფა ორ არათანაბარ ნაწილად: დაახლოებით 62% და 38%. ოქროს კვეთის თანაფარდობა ტოლია 1.6180339887-ის (ნახ. 5.13.).



ნახ. 5.13. მონაკვეთი ოქროს კვეთით. მთელი სიგრძე $a + b$ ისე შეფარდება დიდ a -ს, როგორც დიდი a მცირე b -ს.

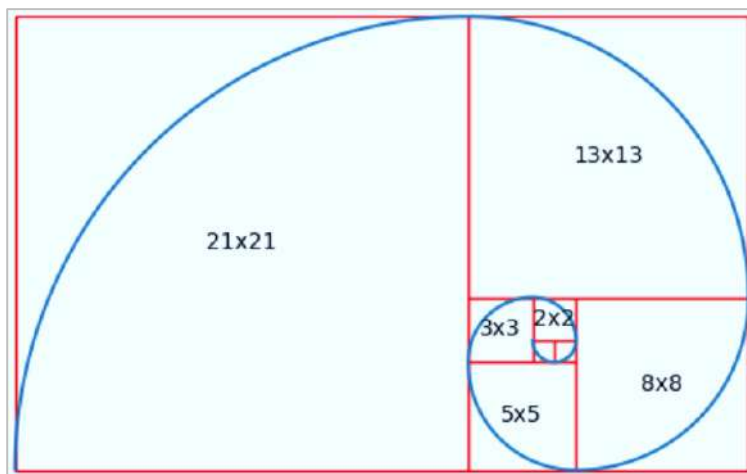
მაგალითად:

მონაკვეთის სიგრძე $c = 1$, $a = 0.618$, $b = 0.382$;

c -ს თანაფარდობა a -სთან $= 1.618$;

c -ს თანაფარდობა b -სთან $= 2.618$;

ფიბონაჩის ოქროს თანაფარდობის/სპირალის გამოსახულება მოცემულია ნახ. 5.14.-ზე



ნახ. 5.14. ფიბონაჩის ოქროს თანაფარდობა

ფიბონაჩის რიცხვებს ხშირად ბუნების ნუმერაციის სისტემასაც უწოდებენ. ფიბონაჩი (დაბ. 1170 - გარ. 1240 წ.წ.) იყო შუა საუკუნეების უდიდესი მათემატიკოსი. მან დაგვიტოვა საკმაოდ ღრმა და საფუძვლიანი ცნობები „ოქროს კვეთასა“ და ღვთიურ მიმდევრობაზე.

ფიბონაჩის n -ური რიცხვი განისაზღვრება შემდეგი ფორმულით (2):

$$F_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \text{ ან } n = 2 \\ F_{n-1} + F_{n-2}, & n > 2 \end{cases} \quad (2)$$

არსებობს ფიბონაჩის n -ური რიცხვის გამოსათვლელი მეთოდები:

- რეკურსიული მეთოდი;
- იტერაციული მეთოდი;
- მატრიცული მეთოდი.

ფიბონაჩის n -ური რიცხვის პოვნის რეკურსიული მეთოდის ალგორითმი:

თუ $n = 1$ ან $n = 2$ დავაბრუნოთ მნიშვნელობა 1, რადგან $F_1 = 1$ და $F_2 = 1$;

დანარჩენ შემთხვევაში ფუნქცია უნდა გამოვიძახოთ $n - 1$ და $n - 2$ არგუმენტებისათვის. ორი გამოძახების მნიშვნელობა შევკრიბოთ და დავაბრუნოთ მიღებული შედეგი.

ფიბონაჩის მიმდევრობის განსაზღვრიდან გამომდინარე, შეიძლება დავწეროთ მარტივი რეკურსიული ფუნქცია, რომელიც იპოვს ფიბონაჩის n -ურ რიცხვს (წევრს).

ფიბონაჩის n -ური რიცხვის მიღების რეკურსიული ფუნქცია:

```
int fibonacci(int n)
{
    if (n <= 0)
        { return 0; }
    else
        if (n == 1 ან n == 2)
            { return 1; }
        else
            { return fibonacci(n - 2) + fibonacci(n - 1); }
}
```

თუ $n = 4$, მაშინ მოხდება fibonacci(3) და fibonacci(2) რეკურსიული გამოძახება.

ზემოთ აღწერილი მარტივი ალგორითმის მიხედვით დაწერილი კოდი შეიძლება ჩავთვალოთ როგორც „კოდი-კატასტროფა“.

ამაში ადვილად დავრწმუნდებით, თუ შევეცდებით გამოვითვალოთ მიმდევრობის F_{40} (მეორმოცე) წევრი, ამ შემთხვევაში კომპიუტერს დასჭირდება დაახლოებით 108 წამი, ხოლო ფიბონაჩის F_{50} (ორმოცდამეათე) წევრის გამოსათვლელად კი გაცილებით მეტი დრო.

გავარკვიოთ, თუ რატომ სჭირდება კოდს ამდენად დიდი დრო.

ნახ. 5.14.-ზე მოცემულია ფიბონაჩის F_5 (მე-5 წევრი) რიცხვის გამოთვლის რეკურსიული გამოძახება: ნახ. 5.14-დან გამომდინარე დავადგინოთ, თუ რამდენჯერ მოხდა რეკურსიული ფუნქციის გამოძახება ფიბონაჩის მიმდევრობის მე-5 წევრის გამოსათვლელად:

F_5 წევრის გამოძახება მოხდება 1-ჯერ;

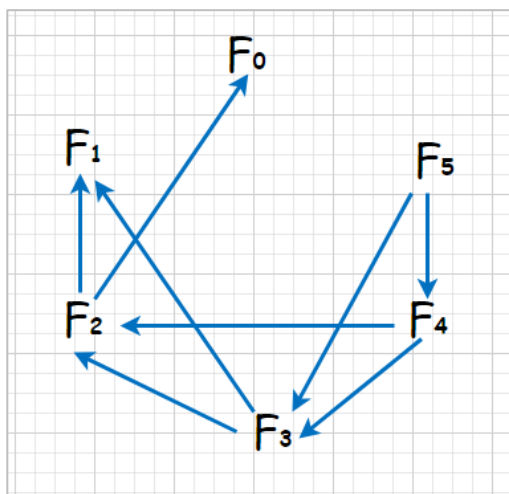
F_4 წევრის გამოძახება მოხდება 1-ჯერ (F_5 -ის გამოსათვლელად);

F_3 წევრის გამოძახება მოხდება 2-ჯერ (F_5 -ის გამოსათვლელად 1-ჯერ და F_4 -ის გამოსათვლელად 1-ჯერ);

F_2 წევრის გამოძახება მოხდება 3-ჯერ (F_4 -ის გამოსათვლელად 1-ჯერ და F_3 -ის გამოსათვლელად 2-ჯერ);

F_1 წევრის გამოძახება მოხდება 5-ჯერ (F_3 -ის გამოსათვლელად 2-ჯერ და F_2 -ის გამოსათვლელად 3-ჯერ);

F_0 წევრის გამოძახება მოხდება 2-ჯერ (F_2 -ის გამოსათვლელად);



ნახ. 5.14. ფიბონაჩის მიმდევრობის მე-5 წევრის რეკურსიული გამოძახება

ნახ. 5.14.-დან ჩანს, რომ ზოგიერთი რიცხვისთვის ფიბონაჩის ფუნქციის გამოძახება მოხდა რამდენიმეჯერ, რაც ანელებს პროგრამის მუშაობას. ასეთი სიტუაციიდან თავის დასარწევად შეგვიძლია გამოთვლილი მნიშვნელობები შევინახოთ რაიმე მასივში (მოვახდინოთ ეგრეთ წოდებული ქეშირება (Caching)) და ყოველი რეკურსიული გამოძახების წინ შევამოწმოთ. თუ უკვე ეს მნიშვნელობა გამოთვლილი გვაქვს, ავიღოთ იგი მასივიდან:

```
int cache[N];

// N - ს შევარჩევთ იმის მიხედვით, თუ მაქსიმუმ რა ნომრის მქონე ფიბონაჩის რიცხვის პოვნა გვჭირდება
int fibonacci(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    if (cache[n - 2] == 0) {
        cache[n - 2] = fibonacci(n - 2);
    }
    if (cache[n - 1] == 0)
    {
        cache[n - 1] = fibonacci(n - 1);
    }
    return cache[n - 2] + cache[n - 1];
}
```

ფიბონაჩის რიცხვის პოვნის რეკურსიული მეთოდი არის ყველაზე მარტივი, მაგრამ უკიდურესად არაეფექტურია მაღალი დროითი დანახარჯის გამო.

შესაძლებელია ფიბონაჩის n -ური რიცხვის პოვნა რეკურსიის ალგორითმის გარეშე, დინამიური პროგრამირების გამოყენებით, თუ მიმდევრობას ავაგებთ ციკლის გამოყენებით.

ფიბონაჩის n -ური რიცხვის პოვნის იტერაციული მეთოდის ალგორითმი: ფიბონაჩის n -ური რიცხვის მოძებნის იტერაციული მეთოდის ალგორითმი არის ჩვეულებრივი იტერაციული ალგორითმი.

თავდაპირველად ხდება თანამიმდევრობის ორი წევრის: F_n და F_{n+1} დამახსოვრება, მათზე დაყრდნობით მოხდება მომდევნო F_{n+2} წევრის გამოთვლა. მნიშვნელობა F_{n+1} გადაეცემა F_n ცვლადს, ხოლო მნიშვნელობა F_{n+2} გადაეცემა F_{n+1} ცვლადს. ციკლის ტანი შესრულდება $n - 2$ -ჯერ, შედეგად გამოითვლება ფიბონაჩის რიცხვი.

ზემოთ აღწერილი მეთოდი გამოდგება მხოლოდ დადებითი n -ისთვის, უარყოფითი n -ებისთვის კი ყველაფერი ზუსტად საპირისპიროდ უნდა გაკეთდეს.

ფიბონაჩის რიცხვის მიღების იტერაციული ფუნქცია:

```
int fibonacci(int n) {  
    int F[n + 1];  
    F[0] = 0; F[1] = 1;  
    for (int i = 2; i <= n; i++) { F[i] = F[i - 2] + F[i - 1]; }  
    return F[n]; }
```

აღნიშნული ფუნქციის მუშაობის დრო არის $O(n)$ და საჭიროა $O(n)$ მეხსიერება.

თუ გვინტერესებს მხოლოდ ფიბონაჩის n -ური წევრი და წინა წევრები არ გვჭირდება, მაშინ მისი პოვნა შესაძლებელია მასივის გარეშეც, ანუ $O(n)$ დროსა და $O(1)$ მეხსიერებაში შემდეგნაირად:

```
int fibonacci(int n) { int a = 0, b = 1;  
    while (n > 0) {  
        int temp = b; b = a + b; a = temp;  
        n--; }  
    return a;  
}
```

მოცემულ მეთოდს აქვს ერთ ნაკლი: თუ გვინდა ვიპოვოთ მიმდევრობის მეშვილიწევრი, მაშინ დროის მიხედვით ალგორითმს დაჭირდება 8 წამი. იტერაციული მეთოდი საშუალებას გვაძლევს სწრაფად გამოვითვალოთ ფიბონაჩის თანამიმდევრობის მხოლოდ პირველი ასიათასი წევრი.

ციკლის რაოდენობის გაზრდამ შეიძლება რადიკალურად გაზარდოს კოდის მუშაობის ხანგრძლივობა. როცა $n > 100.000$ -ზე, ფიბონაჩის რიცხვის მოძებნის დროის შესამცირებლად, უნდა ვიპოვოთ პრობლემის გადაჭრის უფრო ეფექტური მეთოდი. ეს კი არის მატრიცული მეთოდი.

ფიბონაჩის n -ური რიცხვის პოვნის მატრიცული მეთოდის ალგორითმი: ერთ-ერთი ყველაზე სწორი გამოსავალი, როდესაც ვიყენებთ როგორც მცირე დროით, ასევე მეხსიერებით დანახარჯებს, არის ფიბონაჩის n -ური რიცხვის პოვნის მატრიცული მეთოდი.

მათემატიკიდან ცნობილია შემდეგი თანაფარდობა: $\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$

მოცემული ფორმულის განზოგადოებული სახე ნებისმიერი n -სათვის შეგვიძლია ჩავწეროთ შემდეგი თანაფარდობის სახით:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

მიღებული ფორმულა გვეუბნება, რომ კვადრატული მატრიცა (ორი ორზე) მნიშვნელობებით $1, 1, 1, 0$ n -ურ ხარისხში, გვაძლევს იმავე განზომილების ახალ კვადრატულ მატრიცას $F_{n+1}, F_n, F_n, F_{n-1}$ მნიშვნელობებით.

შეგვიძლია ადვილად დავამტკიცოთ შემდეგი მატრიცული თანაფარდობა:

$$\begin{pmatrix} F_{n-2} & F_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} F_{n-1} & F_n \end{pmatrix}$$

და თუ შემოვიტანთ შემდეგ აღნიშვნას $P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, მაშინ მივიღებთ:

$$\begin{pmatrix} F_0 & F_1 \end{pmatrix} \cdot P^n = \begin{pmatrix} F_n & F_{n+1} \end{pmatrix}$$

ამდენად, იმისათვის, რომ ვიპოვოთ ფიბონაჩის მიმდევრობის n -ური რიცხვი საჭიროა მატრიცა P ავიყვანოთ n -ურ ხარისხში.

რატომ არის ეს ფორმულა სასარგებლო? იმიტომ, რომ ხარისხში აყვანა შეიძლება შესრულდეს ლოგარითმულ დროში. ახარისხება უნდა მოხდეს კვადრატში:

$$P^m = \begin{cases} (P^{m/2})^2 & \text{ლუწებისთვის,} \\ P \cdot (P^{\frac{m-1}{2}})^2 & \text{კენტებისთვის.} \end{cases}$$

ყველაფერი ამის შემდეგ დაგვრჩა მატრიცების გადამრავლება.

ახლა კი დავწეროთ ფიბონაჩის მიმდევრობის n -ური რიცხვის პოვნის ფუნქცია:

```
int fibonacci(int n) {
    if (n <= 1) return n;
    int M[2][2] = {{1, 1}, {1, 0}},
        F[2][2] = {{1, 1}, {1, 0}};
    for (int i = 2; i < n; i++)
        matrixMul(F, M);
    return F[0][0];
}

void matrixMul(int F[2][2], int M[2][2]){
    int m00 = F[0][0] * M[0][0] + F[0][1] * M[1][0];
    int m01 = F[0][0] * M[0][1] + F[0][1] * M[1][1];
```

```

int m10 = F[1][0] * M[0][0] + F[1][1] * M[1][0];
int m11 = F[1][0] * M[0][1] + F[1][1] * M[1][1];
F[0][0] = m00, F[0][1] = m01, F[1][0] = m10, F[1][1] = m11;
}

```

მოცემული ფუნქცია $n = 1000000$ წვერის მნიშვნელობას იპოვის ერთი წამის განმავლობაში (0.345 წამი).

ფიბონაჩის n -ური რიცხვის პოვნის მატრიცული მეთოდი აღმოჩნდა ყველაზე სწრაფი და გამოთვლის თვალსაზრისით სწორი. ცხრილ 5.2-ში მოცემულია სამივე მეთოდის კომპიუტერულ/პროგრამული შედარებითი ანალიზი.

ცხრილი 5.2. ფიბონაჩის რეკურსიული, იტერაციული და მატრიცული ალგორითმების

მუშაობის შედეგი

მეთოდი	n -ის მნიშვნელობა				
	1000	10 000	100 000	1000 000	2000 000
რეკურსიული	10 წუთზე მეტი	ბევრი დრო	ბევრი დრო	ბევრი დრო	ბევრი დრო
იტერაციული	0.1 წმ	0.1 წმ	0.2 წმ	8.157 წმ	69.205 წმ
მატრიცული	0.1 წმ	0.1 წმ	0.1 წმ	0.345 წმ	0.689 წმ

დასკვნა: ფიბონაჩის რეკურსიული, იტერაციული და მატრიცული ალგორითმების შედარებიდან ჩანს, რომ ყველაზე ეფექტური აღმოჩნდა მატრიცული მეთოდი, რომელიც შეიძლება გამოყენებულ იქნას ფიბონაჩის თანამიმდევრობის მეშვიშე წვერის მოსაძებნად ერთ წამზე ნაკლებ დროში.

ეს საინტერესოა: მცენარეების, ცხოველებისა და ადამიანების აგებულებაში იმდენად ხშირად ვხვდებით 1.618-ს, რომ ანტიკური დროის მეცნიერებმა მას „ღვთაებრივი პროპორცია“ უწოდეს. ფუტკრების სკაში მდედრების და მამრების რაოდენობა ყოველთვის დაბალანსებულია, მდედრი ფუტკრები რაოდენობრივად სჭარბობენ მამრებს, მაგრამ თუ მდედრების რაოდენობას გავყოფთ მამრი ფუტკრების რაოდენობაზე, ყოველთვის, მსოფლიოს ნებისმიერი სკისთვის, მივიღებთ 1.618-ს. ნაუტილუსის ნიჟარაში ყველა სპირალის დიამეტრის შეფარდება მომდევნოსთან უდრის 1.618-ს. მზესუმზირის მარცვლები სპირალურადაა განლაგებული საათის ისრის საწინააღმდეგოდ. ყოველი წრის დიამეტრის შეფარდება მომდევნოსთან უდრის 1.618-ს და ეს ასეა ფოთლების განლაგებაზე მცენარის ღეროზე, ფიჭვის გირჩის ქერცვლების განლაგება, მწერების სხეულის ნაწილების განლაგება და ა. შ. საინტერესოა, რომ: 0.618 მეორე ხარისხში = $1 - 0.618$, 0.618 მესამე ხარისხში = $0.618 - (0.618 \text{ მეორე ხარისხში})$, 0.618 მეოთხე ხარისხში = $0.6182 - (0.618 \text{ მესამე ხარისხში})$, 0.618 მეხუთე ხარისხში = $0.6183 - (0.618 \text{ მეოთხე ხარისხში})$, და ა. შ. ან პირიქითი ფარდობა:

$$1.618 \text{ მეორე ხარისხში} = 1 + 1.618,$$

$$1.618 \text{ მესამე ხარისხში} = 1.618 + (1.618 \text{ მეორე ხარისხში}),$$

$$1.618 \text{ მეოთხე ხარისხში} = 1.618^2 + (1.618 \text{ მესამე ხარისხში}),$$

$$1.618 \text{ მეხუთე ხარისხში} = 1.618^3 + (1.618 \text{ მეოთხე ხარისხში}), \text{ და ა. შ.}$$

კიდევ რამდენიმე საინტერესო არითმეტიკა ფიბონაჩის რიცხვებით:

$$1) 1.618 - 0.618 = 1,$$

$$2) 1.618 * 0.618 = 1,$$

$$3) 1 - 0.618 = 0.382,$$

$$4) 0.618 * 0.618 = 0.382,$$

$$5) 2.618 - 1.618 = 1,$$

$$6) 2.618 * 0.382 = 1,$$

$$7) 2.618 * 0.618 = 1.618,$$

$$8) 1.618 * 1.618 = 2.618.$$

1-ის და 2-ის გარდა, ფიბონაჩის ნებისმიერი რიცხვი გამრავლებული 4-ზე და დამატებული ნებისმიერი ფიბონაჩის რიცხვი, გვაძლევს ისევ ფიბონაჩის რიცხვს.

$$3 * 4 = 12; + 1 = 13,$$

$$5 * 4 = 20; + 1 = 21,$$

$$8 * 4 = 32; + 2 = 34,$$

$$13 * 4 = 52; + 3 = 55,$$

$$21 * 4 = 84; + 5 = 89, \text{ და ა.შ.}$$

0.618 (ან 1.618) ცნობილია როგორც ოქროს პროპორცია. მისი გეომეტრია სასიამოვნოა თვალისთვის, მიღებულია კაცობრიობის მიერ და ადამიანებისთვის ჰარმონიულობის ნაწილს შეადგენს. ამ პროპორციას ემორჩილება უამრავი რამ ჩვენს ირგვლივ მუსიკა, არქიტექტურა, ბიოლოგია, მზესუმზირა, ლოკოკინას ნიჟარა, ლამბაქი, სპირალური გალაქტიკა და ა. შ. ამავე კანონს მიყვება ძალიან ხშირად ფასის მოძრაობა ბაზარზე.

<https://mathtasks.wordpress.com/tag/%E1%83%A4%E1%83%98%E1%83%91%E1%83%9D%E1%83%9C%E1%83%90%E1%83%A9%E1%83%98/>

მონაცემთა ფორმალიზაცია. ამოცანების მონაცემთა ფორმალური ჩაწერა, მონაცემთა კოდირება და მათი დამუშავება, რიცხვების ჩაწერის სხვადასხვა სისტემა, ანბანი, ენა და გრამატიკა. ძნელად ამოსახსნელი ამოცანები. პოლინომიალური და ექსპონენციალური ალგორითმები, მაგალითები.

გამოჩენილი გერმანელი ფილოსოფოსი გოტფრიდ ვილჰელმ ლაიბნიცი (1646-1716 წ.წ.) ფართო თვალსაწიერის მქონე უნივერსალური მეცნიერი იყო. ლაიბნიცმა დიდი წვლილი შეიტანა მეცნიერების სხვადასხვა დარგის: ლოგიკა, მათემატიკა, სამართალი, ფიზიკა, ისტორია, თეოლოგია, ენის თეორია, ენის ფილოსოფია და სხვათა განვითარებაში. ჯერ კიდევ ახალგაზრდა ლაიბნიცმა გამოაქვეყნა „თხზულება კომბინატორიკის ხელოვნების შესახებ“, რომელშიც ლოგიკური გამოკვლევების ფორმალური საფუძვლებია მოცემული.



ნახ. 6.1. გოტფრიდ ვილჰელმ ლაიბნიცი

ლაიბნიცის სამეცნიერო მიღწევები მოიცავს არაერთ დისციპლინას და დიდი წვლილი აქვს შეტანილი მათ განვითარებაში:

მათემატიკა. ლაიბნიცმა გამოიგონა ინტეგრალური და დიფერენციალური კალკულუსი. მისი ნაშრომები და ნოტაციები, განსაკუთრებით ინტეგრალური ნიშანი $\int$ და დიფერენციალური ნიშანი d , დღემდე გამოიყენება და საფუძვლად უდევს თანამედროვე მათემატიკის ბევრ სფეროს.

ლოგიკა. ლაიბნიცმა მნიშვნელოვანი წვლილი შეიტანა ლოგიკაში (ლაიბნიცი ლოგიკას მეორენაირად უწოდებდა „აზროვნების კანონს“). მან განავითარა იდეა „ლოგიკური კალკულუსის“ შესახებ, რომელიც გახდა წინამორბედი თანამედროვე ლოგიკური და კომპიუტერული თეორიებისათვის.

ისტორია და ლინგვისტიკა. ლაიბნიცმა ასევე მნიშვნელოვანი წვლილი შეიტანა ისტორიისა და ლინგვისტიკის სფეროებში. იგი იყო ერთ-ერთი პირველი, ვინც მიიჩნია, რომ ენების ისტორიული კვლევა და შედარებითი ანალიზი მნიშვნელოვანი იყო მათი განვითარების გასაგებად.

ტექნოლოგია. ლაიბნიცმა გამოიგონა პირველი მექანიკური კალკულატორი, რომელიც ასრულებდა არითმეტიკულ ოპერაციებს. ეს მოწყობილობა, სახელწოდებით „სტეპრეკონერი“ (Step Reckoner), იყო ერთ-ერთი პირველი კომპიუტერული მოწყობილობა, რომელიც შემდგომში საფუძვლად დაედო თანამედროვე კომპიუტერების განვითარებას.

ლაიბნიცი ოცნებობდა შეექმნა „უნივერსალური მეცნიერება“. მას უნდოდა გამოეყო მარტივი ცნებები, რომელთა საშუალებითაც გარკვეული წესების გამოყენებით შესაძლებელი იქნებოდა ნებისმიერი სირთულის ცნებების ფორმულირება. აგრეთვე ოცნებობდა შეექმნა უნივერსალური ენა, რომელზეც შესაძლებელი იქნებოდა ნებისმიერი აზრის ჩაწერა მათემატიკური ფორმულების სახით და კიდევ, ის ფიქრობდა მანქანაზე, რომელსაც შეეძლებოდა თეორემების და აქსიომების გამოყვანა.

ლაიბნიცმა შეიმუშავა „ლოგიკური კალკულუსის“ კონცეფცია, რომელიც მიზნად ისახავდა აზრების ფორმალიზებას და მათ მათემატიკურ სიმბოლოებად გადაქცევას. მან შეიმუშავა ლოგიკური ნოტაციები, რომლებიც დღემდე გამოიყენება და მისმა იდეებმა საფუძველი დაუდო თანამედროვე ლოგიკურ და კომპიუტერულ თეორიებს. მისი ნაშრომები პირდაპირი წინამორბედი იყო ბულეს ალგებრისა და თანამედროვე ლოგიკური სისტემების განვითარებაში.

ლეიბნიცი ასევე არის ფორმალიზებული ლოგიკის დამფუძნებელი. იგი შეისწავლიდა თვლის ორობით სისტემას.

ლაიბნიცისთვის ციფრი 1 ასოცირდებოდა ღმერთთან, ხოლო 0 სიცარიელესთან. ლაიბნიცი თვლიდა, რომ ამ ორი ციფრიდან წარმოიქმნა ყველაფერი.

ფორმალური ენა არის რეალური ენის მათემატიკური მოდელი. ფორმალური ენების დახმარებით ხდება ფორმალური ხასიათის ინფორმაციის მოდელის (მათემატიკური, ლოგიკური და ა. შ.) აგება.

ფორმალური ენების გამოყენებით ინფორმაციის მოდელის აგების პროცესს ფორმალიზაცია ეწოდება.

სალაპარაკო ენები ემსახურება აღწერითი ინფორმაციის მოდელის შექმნას. მეცნიერების ისტორიაში ცნობილია აღწერითი ინფორმაციის მრავალი მოდელი.

მაგალითად, კოპერნიკის მიერ შემოთავაზებული სამყაროს ჰელიოცენტრული მოდელი ჩამოყალიბდა შემდეგნაირად: დედამიწა ბრუნავს თავის ღერძის და მზის გარშემო; ყველა პლანეტა ბრუნავს მზის გარშემო.

მაგალითად, წიგნის სარჩევი (შინაარსი) არის ძირითადი თავების ფორმალიზაცია, ხოლო თავად ტექსტი შეიძლება ჩაითვალოს ავტორის აზრების, იდეების, გამოსახვის ენობრივი კონსტრუქციების ფორმალიზაციად.

ერთ-ერთი ყველაზე ფართოდ გამოყენებული ფორმალური ენა არის მათემატიკური ენა. მათემატიკური ცნებებისა და ფორმულების გამოყენებით ჩამოყალიბებულ მოდელებს მათემატიკური მოდელები ეწოდება.

მათემატიკის ენა არის ფორმალური ენების კრებული. ზოგიერთი მათგანი, მაგალითად ალგებრა, გეომეტრია სკოლაში აითვისეთ, სხვების გაცნობა კი შეგიძლიათ საუნივერსიტეტო განათლების მიღების დროს.

ალგებრის ლოგიკური ენა (წინადადებების ალგებრა) გვამღევს შესაძლებლობას ავაგოთ ფორმალური ლოგიკური მოდელები. ალგებრის დახმარებით რაიმე გამოთქმული მოსაზრება ფორმალიზდება მარტივი და რთული წინადადებების სახით. ლოგიკური მოდელების აგებით შესაძლებელია ლოგიკური ამოცანების გადაჭრა, კომპიუტერული მოწყობილობების ლოგიკური მოდელების შექმნა (შემკრები, ტრიგერი) და ა.შ.

გარემომცველი სამყაროს შემეცნების პროცესში კაცობრიობა მუდმივად მიმართავს პროცესებისა და სისტემების მოდელირებასა და ფორმალიზებას. რაც შეეხება ადამიანის გონებრივ შესაძლებლობებს: შესაძლებელია თუ არა, ადამიანი იმაზე ჭკვიანი გახადო, ვიდრე ის არის? ამ კითხვაზე პასუხის გასაცემად, უნდა განვმარტოთ, რომ ადამიანის მთელი ინტელექტუალური აქტივობა შეიძლება დაიყოს ფორმალიზებულ და არაფორმალიზებულ აქტივობად.

ფორმალიზებადი არის აქტივობა, რომელიც ხორციელდება გარკვეული წესების მიხედვით. მაგალითად გამოთვლების შესრულება, საქაღალდეში ძებნა და გრაფიკული სამუშაოები უდავოდ შეიძლება დაევალოს კომპიუტერს და ყველაფერი, რისი გაკეთებაც კომპიუტერს შეუძლია, ის ამას აკეთებს უკეთესად და უფრო სწრაფად, ვიდრე ადამიანი.

არაფორმალიზებადი არის ისეთი აქტივობა, რომელიც ხდება ჩვენთვის უცნობი წესების გამოყენებით. აზროვნება, მსჯელობა, ინტუიცია, საღი აზრი - ჩვენ ჯერ კიდევ არ ვიცით რა არის ეს და ბუნებრივია, არაფორმალიზებადი აქტივობა არ შეიძლება მივანდოთ კომპიუტერს, თუნდაც იმიტომ, რომ უბრალოდ არ ვიცით, რა დავალება ან მოთხოვნა უნდა გადავცეთ მას.

გადაწყვეტილების მიღება ერთგვარი გონებრივი აქტივობაა. ზოგადად მიღებულია, რომ გადაწყვეტილების მიღება არაფორმალური საქმიანობაა თუმცა, ეს ყოველთვის ასე არ არის. ერთი მხრივ, ზოგჯერ ჩვენ არ ვიცით, როგორ მივიღოთ გადაწყვეტილება, მაგალითად, „ჩვენ ვიღებთ გადაწყვეტილებას საღი აზრის დახმარებით“ არაფერს იძლევა. მეორე მხრივ, მნიშვნელოვანი ამოცანების გადაწყვეტის მიღების ფორმალიზება შესაძლებელია. პრობლემასთან მიმართებით გადაწყვეტილების მიღების ერთ-ერთი სახეა გადაწყვეტილების მიღების ოპტიმალური პრობლემები (შესაძლებელია მისი ფორმალიზება) ან

ოპტიმიზაციის პრობლემები. ოპტიმიზაციის პრობლემა მოგვარებულია მათემატიკური მოდელების დახმარებით და კომპიუტერული ტექნოლოგიების გამოყენებით.

თანამედროვე კომპიუტერები აკმაყოფილებს უმაღლეს მოთხოვნებს. მათ შეუძლიათ შეასრულონ მილიონობით ოპერაცია წამში, მათ შეუძლიათ მეხსიერებაში ჰქონდეთ ყველა საჭირო ინფორმაცია, დისპლეი-კლავიატურის კომბინაცია უზრუნველყოფს დიალოგს ადამიანსა და კომპიუტერს შორის. ამასთან, არ უნდა ავურიოთ კომპიუტერების შექმნის წარმატება და სხვადასხვა სფეროში მათი გამოყენების მიღწევები.

ფაქტობრივად, ყველაფერი, რაც კომპიუტერს შეუძლია არის ადამიანის მიერ აღწერილი პროგრამის თანახმად საწყისი მონაცემების გარდაქმნის უზრუნველყოფა.

კარგად უნდა გვესმოდეს, რომ კომპიუტერი არ და ვერ იღებს გადაწყვეტილებებს. იგი მხოლოდ გადაწყვეტილებების პოვნაში გვეხმარება. გადაწყვეტილება შეიძლება მიიღოს მხოლოდ ადამიანმა და მისთვის კომპიუტერი შესანიშნავი ასისტენტი, რომელსაც შეუძლია განავითაროს და შესთავაზოს სხვადასხვა გადაწყვეტილებების ნაკრები. ამ ნაკრებიდან ადამიანი აირჩევს იმ ვარიანტს, რომელიც, მისი თვალსაზრისით, უფრო შესაფერისი იქნება. რა თქმა უნდა, გადაწყვეტილების მიღების ყველა პრობლემა არ შეიძლება გადაწყდეს კომპიუტერის დახმარებით. მიუხედავად ამისა, მაშინაც კი, თუ კომპიუტერზე პრობლემის გადაწყვეტა სრული წარმატებით არ დასრულებულა, ის მაინც სასარგებლო აღმოჩნდება, რადგან ეს ხელს უწყობს ამ პრობლემის უფრო ღრმა გააზრებას და მის უფრო მკაცრ ფორმულირებას.

ბუნებრივი და ფორმალური ენები. ერთი და იგივე ინფორმაცია შეიძლება გამოითქვას ან აღიწეროს სხვადასხვა გზით. ადამიანს შეუძლია ინფორმაციის წარმოდგენა სიმბოლური ან ფიგურალური ფორმით. ინფორმაციის შესანახად და სხვა პირისთვის გადასაცემად, ადამიანი აფიქსირებს მას ნიშნების დახმარებით. ენა არის ნიშანთა სისტემა, რომელსაც ადამიანი იყენებს თავისი აზრების გამოხატვის, სხვა ადამიანებთან კომუნიკაციისთვის.

არსებობს ბუნებრივი და ფორმალური ენები.

ბუნებრივი ენები (ქართული, გერმანული, რუსული და ა.შ.) ჩამოყალიბდა საზოგადოების განვითარების შედეგად და გამოიყენება ადამიანთა შორის კომუნიკაციისთვის.

ბუნებრივი ენებისთვის დამახასიათებელია:

1. ფართო მასშტაბი. ბუნებრივი ენა ცნობილია მთელი ეროვნული საზოგადოებისთვის;
2. დიდი რაოდენობის წესების არსებობა. ზოგიერთი წესი ფორმულირებულია ცალსახად (გრამატიკის წესები), ზოგიც იმპლიციტურად (მნიშვნელობისა და გამოყენების წესები);

3. მოქნილობა. ბუნებრივი ენა გამოიყენება ნებისმიერი, მათ შორის ახალი, სიტუაციის აღსაწერად;
4. გახსნილობა. ბუნებრივი ენა მოსაუბრეს საშუალებას აძლევს, წარმოქმნას თანამოსაუბრესთვის გასაგები ახალი ნიშნები (სიტყვები), ასევე გამოიყენოს არსებული ნიშნები ახალ მნიშვნელობებში;
5. დინამიზმი. ბუნებრივი ენა სწრაფად ეგუება ადამიანებს შორის ინტერპერსონალური ურთიერთქმედების მრავალფეროვან საჭიროებებს.

მეცნიერებისა და ტექნოლოგიების განვითარებასთან ერთად წარმოიშვა **ფორმალური ენები**, რომლებსაც სპეციალისტები იყენებენ თავიანთ პროფესიულ საქმიანობაში თუმცა, ბევრ ოფიციალურ ენას აქვს საერთაშორისო გამოყენება. ფორმალური ენების თავისებურება ის არის, რომ მათში ყველა წესი ცალსახად არის მითითებული, რაც უზრუნველყოფს ამ ენებზე შეტყობინებების ცალსახა ჩაწერას და აღქმას.

ფორმალური ენა არის ისეთი ენა, რომელშიც სიმბოლოების ერთსა და იმავე კომბინაციებს ყოველთვის აქვთ ერთნაირი მნიშვნელობა. ფორმალურ ენაში თითოეული სიტყვის მნიშვნელობა, ასევე წინადადებების აგების წესები და მათთვის მნიშვნელობის მინიჭება ცალსახად არის განსაზღვრული.

ფორმალური ენებში შესაძლებელია მოვიაზროთ მათემატიკური და ქიმიური სიმბოლოების სისტემები, მუსიკალური ნოტების აღნიშვნები, მორზეს კოდი და ბევრი სხვა. ფორმალური ენას შეიძლება მივაკუთვნოთ ათობითი რიცხვების სისტემა.

ფორმალური ენის მაგალითებია:

- მათემატიკური ფორმულები: $S = v \cdot t$;
- რიცხვების ჩაწერის წესები: 12345;
- სანოტო ჩანაწერი: ;
- ჭადრაკის პარტიის ჩაწერის ენა: g4-f4; B2-C6;

ალგორითმულია ენა:

1. დაწყება;
2. ეკრანზე გამოტანა „საქართველოს ეროვნული ნაკრები, 2024, ყოჩაღ!“;
3. დასრულება;

ნებისმიერი ფორმალური ენა არის ხელოვნური, რადგან შექმნილია ადამიანის მიერ.

ადამიანს შეუძლია ინფორმაციის წარმოდგენა ბუნებრივ ენებზე, ფორმალურ ენებზე, სხვადასხვა ფიგურალური ფორმით.

ინფორმაციის ნებისმიერ ენაზე ან გადატანითი ფორმით წარმოდგენას კოდირება ეწოდება.

მონაცემთა კოდირება, ანბანი, ენა და გრამატიკა. განვიხილოთ ქართული სიტყვები „ანბანი“ და „ენა“. ეს ქართული ენის სიტყვებია, რომელთაც ენაში რაღაცა მნიშვნელობა (სემანტიკა) აქვს. სხვა საქმეა სიტყვა „გჰჰჰჰ“ - ეს ქართული ენის სიტყვა არაა, თუმცა ქართული ანბანით კი არის ჩაწერილი. ამით განსხვავდება ერთმანეთისაგან „ენის სიტყვა“ და „ენის ანბანით ჩაწერილი სიტყვა“.

„ენის ანბანით ჩაწერილი სიტყვა“ ამ ენის ანბანის ასოების მიმდევრობაა, რომელსაც რაღაცა სემანტიკური დატვირთვა (ანუ აზრი) შეიძლება ჰქონდეს, ან არა. რაიმე ანბანით ჩაწერილი სიტყვა შეიძლება იყოს სასრული ან უსასრულო.

როგორც წესი, ჩვენს ყოველდღიურობაში მხოლოდ სასრული სიტყვები გვხვდება. სასრული სიტყვა სასრული ზომისაა, რაც მასში შემავალი ასო-ნიშნების რაოდენობით განისაზღვრება. მაგალითად, $|\text{ანბანი}| = 6$; $|\text{ენა}| = 3$;

სიმრავლეებზე განიხილავენ ორ მოქმედებას: თანაკვეთას და გაერთიანებას.

ორი ან მეტი სიმრავლის თანაკვეთა ეწოდება მათი საერთო ელემენტებისაგან შედგენილ სიმრავლეს და აღინიშნება $\cap$ სიმბოლოთი.

მაგალითად, $A \cap B = \{x | x \in A \text{ და } x \in B\}$

ორი ან მეტი სიმრავლის გაერთიანება ეწოდება იმ ელემენტთა სიმრავლეს, რომლებიც ერთ-ერთში მაინც შედის და აღინიშნება $\cup$ სიმბოლოთი.

მაგალითად, $A \cup B = \{x | x \in A \text{ ან } x \in B\}$

ორი სიმრავლის სხვაობა ეწოდება სიმრავლეს, რომლის ელემენტებიც შედის ერთში და არ შედის მეორეში.

მაგალითად, $A \setminus B = \{x | x \in A; x \notin B\}$

იმ ელემენტებისგან შედგენილ სიმრავლეს, რომლებიც რჩება სხვაობის გარეთ, ეწოდება A -ს დამატება B სიმრავლემდე.

თუ მოცემულია რაღაცა სიტყვა $\omega = \omega_1 \omega_2 \dots \omega_n$ მისი სიგრძე (ანუ სიტყვაში არსებული ასო-ნიშნების რაოდენობა) შემდეგნაირად აღინიშნება: $|\omega| = n$.

$\omega(i)$ მოცემული სიტყვის i -ური ასოა. მაგ.: **ანბანი(4) = ა**; **ელექტროფიკაცია(7) = ი**;

თუ მოცემულია ორი სიტყვა ω_1 და ω_2 , მაშინ $\omega_1 \circ \omega_2 = \omega_1 \omega_2$ (ეს ნიშნავს ორი სიტყვა ერთი მეორეს მიყოლებით).

მაგალითად, **ფეხ $\circ$ ბურთი = ფეხბურთი**

თუ რაღაცა სიტყვა $\omega = u \circ v$ მაშინ ამბობენ, რომ u სიტყვა ω სიტყვის პრეფიქსია, ხოლო v სიტყვა ω სიტყვის სუფიქსია: $u < \omega$ და $v > \omega$.

ω სიტყვის n ასოიანი პრეფიქსი აღინიშნება როგორც $\omega[n]$, ხოლო მისი m ასოიანი სუფიქსი კი აღინიშნება როგორც $\omega\{m\}$ (არ აგერიოთ $\omega(n)$ -ში!!!).

თუ $Q = \{ \alpha, \beta, \gamma, \delta, \dots, \zeta, \zeta \}$ ქართული ანბანია, მაშინ Q^n ყველა იმ სიტყვის სიმრავლეა, რომელიც ქართული ანბანით არის შედგენილი, სადაც n ასოების რაოდენობაა:

$$Q^n = \{ \omega \mid \omega(i) \in Q, (1 \leq i \leq n), |\omega| = n \} \quad (1)$$

ჩანაწერი $\omega(i) \in Q$ ნიშნავს $\omega(i)$ ეკუთვნის Q სიმრავლეს.

Q^* ყველა იმ სასრული სიტყვის სიმრავლეა, რომელიც Q ანბანის ასოებითაა შედგენილი:

$$Q^* = \bigcup_{i=1}^{\infty} Q^i = Q^1 \cup Q^2 \cup \dots \cup Q^n \cup \dots \quad (2)$$

$\cup$ ნიშნავს გაერთიანებას. სასრული და უსასრულო სიგრძის სიტყვების გარდა არსებობს კიდევ ე.წ. „ცარიელი სიტყვა“ ε ანუ ისეთი სიტყვა, რომელიც არცერთი ასოსაგან არ შედგება (ცარიელია).

ცხადია, რომ $|\varepsilon| = 0$, $\varepsilon < \omega$ და $\omega \circ \varepsilon = \varepsilon \circ \omega = \omega$ ნებისმიერი ω სიტყვისათვის.

განმარტება:

ნებისმიერი სასრული სიმრავლე A შეგვიძლია განვიხილოთ, როგორც ანბანი.

რაიმე ენაზე შექმნილი სიტყვა არის ენის ანბანის ელემენტების (ანუ ასო-ნიშნების) მიმდევრობა.

თუ რაიმე $\omega \in A$ ანბანზე შექმნილი სიტყვაა, მაშინ $|\omega|$ ამ სიტყვაში შემავალი ასოების რაოდენობაა.

თუ $|\omega| = 0$, ასეთ სიტყვას ცარიელი ეწოდება და მას აღნიშნავენ ε სიმბოლოთი.

თუ $|\omega| = \infty$, ასეთ სიტყვას ეწოდება უსასრულო.

თუ მოცემულია ორი სიტყვა ω და ν (სადაც ω სასრულია), მაშინ $\omega \circ \nu = \omega\nu$ ამ ორი სიტყვის კონკანეტაციაა ანუ შერწყმა.

ამბობენ, რომ რომ u სიტყვა ω სიტყვის პრეფიქსია ($u < \omega$), თუ $\exists \nu$ სიტყვა ისეთია, რომ $\omega = u \circ \nu$.

სიმბოლო $\exists$ ნიშნავს - არსებობს. გამოთქმა $\exists(x)P(x)$ ნიშნავს: "არსებობს x რომელსაც აქვს თვისება $P(x)$ " ანუ "არსებობს x -ის ისეთი მნიშვნელობა, რომ $P(x)$ არის ჭეშმარიტი" რაც იგივეა, $\exists = x \mid P(x)=\text{true}$.

ანალოგიურად, u სიტყვა ω სიტყვის სუფიქსია ($u > \omega$), თუ $\exists \nu$ სიტყვა ისეთია, რომ $\omega = \nu \circ u$.

თუ ω რაიმე სიტყვაა, მაშინ $\omega(n)$ მისი მე- n ასოა, $\omega[n]$ მისი n ასოსაგან შემდგარი პრეფიქსი, ხოლო $\omega\{n\}$ მისი n ასოსაგან შემდგარი სუფიქსია.

თუ მოცემულია A ანბანი, მაშინ $A^n = \{ \omega \mid |\omega| = n \}$ და $A^* = \{ \omega \mid |\omega| < \infty \}$

განმარტების დასასრული.

ახლა გავარკვიოთ სიმრავლე და ქვესიმრავლე ქართულ ანბანთან მიმართებით:

ამბობენ, რომ $\omega \in A^*$ სიტყვა $v \in A^*$ სიტყვას შეიცავს, თუ $\exists \omega_1, \omega_2 \in A^*$ და $\omega = \omega_1 \circ v \circ \omega_2$ (ω_1 ან ω_2 ცარიელიც შეიძლება იყოს). ამ შემთხვევაში იტყვიან, რომ v სიტყვა ω სიტყვის ქვესიტყვაა.

მაგალითად, თუ გვაქვს სიტყვა „მოდოფიკაცია“, მაშინ მისი ქვესიტყვებია „დიფიკა“, „კაცი“, „მოდო“, „კაცია“. ამას გარდა, „მოდო“ მისი პრეფიქსია, ხოლო „კაცია“ კი - სუფიქსი. მაგრამ „მოდოკაცია“ მისი ქვესიტყვა არაა, თუმცა შედგება ორი ქვესიტყვისაგან.

ანალოგიურად სიტყვები შეიძლება შევადგინოთ ნებისმიერ სხვა ანბანზე, ანუ სასრულ სიმრავლეზე. მაგალითად, თუ მოცემულია ათობითი ანბანი $A = \{0; 1; 2; 3; 4; 5; 6; 7; 8; 9\}$, მასში შეიძლება ყველა ნატურალური რიცხვი ჩაიწეროს. ასეთ ჩანაწერს „რიცხვის ათობითი ჩანაწერი“ ეწოდება, რადგან მის გამოსახატავად (ჩასაწერად) მხოლოდ ეს 10 ასო-ნიშანი ანუ ციფრი გამოიყენება.

როგორც აღმოჩნდა, შეიძლება უსასრულოდ ბევრი ანბანის შექმნა. თუ M_1 და M_2 სხვადასხვა ანბანებია, არსებობს ობიექტიური ასახვა $f: M_1^* \rightarrow M_2^*$, რაც იმას ნიშნავს, რომ ანბანის შერჩევას მნიშვნელობა არ აქვს: რაც ერთი ანბანით ჩაიწერება, იმავე სხვა ნებისმიერი ანბანითაც შეიძლება ჩაიწეროს.

მაგალითად, შეიძლება ქართული ანბანის სიტყვები ათობითი ანბანით ჩაიწეროს.

პირველ რიგში ქართული ანბანის თითო ასო-ნიშანი ათობითი ანბანის სიტყვებად ჩავეწერთ:

ქართული ანბანი - ვარიანტი 1

ა → 00 ბ → 01 გ → 02 დ → 03 ე → 04 ვ → 05 ზ → 06 თ → 07 ი → 08 კ → 09

ლ → 10 მ → 11 ნ → 12 ო → 13 პ → 14 ჟ → 15 რ → 16 ს → 17 ტ → 18 უ → 19

ფ → 20 ქ → 21 ღ → 22 ყ → 23 შ → 24 ც → 25 ჩ → 26 ძ → 27 წ → 28 ჭ → 29

ხ → 30 ჯ → 31 ჰ → 32

მეორე რიგში ქართული ანბანით ჩაწერილი ყოველი სიტყვის ასო-ნიშანი შეესაბამისი ორეულით უნდა შევცვალოთ.

სიტყვა „ფეხბურთი“ შემდეგნაირად ჩაიწერება: 2004300119160708

თუ ცნობილია, რომელ ასოს რომელი ციფრების წყვილი (ორეული) შეესაბამება, ადვილი გამოსაანგარიშებელია ქართული სიტყვა. მაგრამ თუ ათობითი ჩაწერილი ეს სიტყვა ვინმეს ჩაუვარდა ხელში, ვინც არ იცის, თუ რომელ რიცხვს რომელი ასო შეესაბამება, ქართული სიტყვის აღდგენა გაძნელებულია. ძალიან ძნელი იქნება საწყისი სიტყვის აღდგენა, თუ ჩვენ ასოებს ორ ნიშნა რიცხვებს შევუსაბამებთ ისე, როგორც ამას ჩვენ მოვინდომებთ, მაგალითად:

ქართული ანბანი - ვარიანტი 2

ა → 39 ბ → 27 გ → 99 დ → 03 ე → 38 ვ → 21 ზ → 76 თ → 78 ი → 87 კ → 90
 ლ → 10 მ → 11 ნ → 13 ო → 31 პ → 37 ჟ → 65 რ → 16 ს → 17 ტ → 18 უ → 19
 ფ → 47 ქ → 51 ღ → 66 ყ → 08შ → 24 ც → 25 ჩ → 26 ძ → 00წ → 01 ჭ → 09
 ხ → 81 ჯ → 06 ჰ → 32

თუ ცნობილია, რომელ ასოს რა ორეული შეესაბამება, შეგვიძლია რაღაცა $f: Q \rightarrow A \times A$ ფუნქციის შედგენა (აქ Q ქართული ანბანია და $A = \{0,1,2,\dots,9\}$). თუ ეს ფუნქცია შედგენილია ზედა ანბანის (*ქართული ანბანი - ვარიანტი 2*) საშუალებით, მაშინ $f(ა) = 39$, $f(ბ) = 27$, $f(ლ) = 10$, $f(შ) = 24$ და ა.შ.

რაღაცა სიტყვა $\omega \in Q^n$, მაშინ ათობითი ანბანის გამოყენებით შეიძლება ჩავწეროთ რეკურსიული ალგორითმი:

ალგორითმი $P(\omega)$

მონაცემები: $\omega \in Q^{|\omega|}$

თუ $\omega = \epsilon$, ალგორითმი დაასრულე;

პასუხად გამოიტანე სიტყვა: $P(\omega[|\omega| - 1]) \circ f(\omega[|\omega|])$

აქ f ფუნქცია ქართული *ანბანი - ვარიანტი 2*-ით მოყვანილი ანბანითაა განსაზღვრული. ამ ალგორითმში ორი სიახლეა შემოტანილი:

1. ეს ალგორითმი ფორმალურადაა ჩაწერილი. ჩანაწერი „ალგორითმი $P(\omega)$ “ ნიშნავს, რომ ალგორითმს სახელად ეწოდება P , ხოლო მონაცემად (სამეცნიერო ტერმინოლოგია რომ ვიხმაროთ, არგუმენტად) მოცემული აქვს სიტყვა ω .

2. იმის მაგივრად, რომ სიტყვიერად ვწეროთ: „ჩაატარე იგივე ოპერაციები $\omega[|\omega| - 1]$ მონაცემისათვის“, ჩვენ ვწერთ $P(\omega[|\omega| - 1])$ (რადგან ალგორითმს ეწოდება P , ამიტომ $P(\omega[|\omega| - 1])$ ნიშნავს: ჩაატარე ალგორითმი P მონაცემით $\omega[|\omega| - 1]$).

აღსანიშნავია, რომ არსებობს მეთოდები, რომელთა საშუალებითაც შეიძლება *ქართული ანბანი - ვარიანტი 2*-ის მიხედვით კოდირებული ტექსტის გახსნა იმის და მიუხედავად, თუ *ქართული ანბანი - ვარიანტი 2* ცნობილი არ არის: თუ ვიცით, რომ კოდირებულია ქართული ტექსტი, მოვძებნით იმ ორეულს, რომელიც ყველაზე ხშირად გვხვდება ქართულ ანბანში. რადგან ქართულ ენაში ყველაზე ხშირად გვხვდება ასო-ნიშანი „ა“, ამიტომ სავარაუდოა, რომ ის ორეულიც „ა“ ასოს შესაბამისი იქნება. შემდეგ დავითვლით იმ ოთხეულების რაოდენობას, რომელიც „ა“ ასოს შესაბამისი ორეულით იწყება. ასევე დადგენილია ქართულ ენაში, თუ რომელი ასო-ნიშანი გვხვდება ყველაზე ხშირად „ა“ ასოს შემდეგ.

ანალოგიურად და რამდენიმე ექსპერიმენტის ჩატარების შედეგად ტექსტის გაშიფვრა შესაძლებელია.

მონაცემთა ამდაგვარი კოდირებითა და გახსნით დაკავებულია ინფორმატიკისა და მათემატიკის ერთ-ერთი განხრა - კრიპტოგრაფია.

თუ მოცემულია რაიმე ანბანი M , მაშინ $L \subset M^*$ ამ ანბანზე შედგენილი ენა ეწოდება.

ანბანსა და ენას ცენტრალური როლი ენიჭება ინფორმატიკაში, რადგან დამტკიცდა, რომ ნებისმიერი ამოცანა შეიძლება რაღაც ენაში ჩაიწეროს და მისი ამოხსნის ძიება ამ ენაში გარკვეული სიტყვების ძიების ტოლფასია.

ინფორმატიკაში ძალიან მნიშვნელოვანია ე. წ. ორობითი ანბანი $B = \{0,1\}$. ნებისმიერი ინფორმაცია შეიძლება ჩაიწეროს ამ ანბანის სიტყვებით ანუ ორობით კოდში.

მაგალითად, თუ მოცემულია რაიმე ნატურალური რიცხვი $n \in N$, მისი ჩაწერა ორობით კოდში შეიძლება ქვემოთ აღწერილი ალგორითმით.

1. მოცემულია: $n \in N$
2. თუ $n = 0$, ალგორითმი დაასრულე
3. ამობეჭდე $n/2$ გაყოფისას მიღებული ნაშთი:
4. (თუ n კენტია, ამობეჭდე 1);
5. (თუ n ლუწია, ამობეჭდე 0);
6. ეს პროცედურა გაიმეორე $[n/2]$ მონაცემისათვის.

მაგალითად, თუ $n = 5$ ალგორითმი შემდეგნაირად იმუშავებს:

მოცემულია: $n = 5$

- თუ $n = 0$, ალგორითმი დაასრულე - (ამ შემთხვევაში ეს არ სრულდება)
- თუ n კენტია, ამობეჭდე 1 - (ამ შემთხვევაში ეს სრულდება: 5 კენტია);
- ამობეჭდილი რიცხვი: 1
- თუ n ლუწია, ამობეჭდე 0 - (ამ შემთხვევაში ეს არ სრულდება);
- ეს პროცედურა გაიმეორე $[5/2] = 2$ მონაცემისათვის:

მოცემულია: $n = 2$

- თუ $n = 1$, ამობეჭდე 1 და ალგორითმი დაასრულე - (ამ შემთხვევაში ეს არ სრულდება);
- თუ n კენტია, ამობეჭდე 1 - (ამ შემთხვევაში ეს არ სრულდება: 2 ლუწია);
- ამობეჭდილი რიცხვი: 01
- თუ n ლუწია, ამობეჭდე 0 - (ამ შემთხვევაში ეს სრულდება: 2 ლუწია);
- ეს პროცედურა გაიმეორე $[2/2] = 1$ მონაცემისათვის:

მოცემულია: $n = 1$

- თუ $n = 0$, ალგორითმი დაასრულე - (ამ შემთხვევაში ეს არ სრულდება)
- თუ n კენტია, ამობეჭდე 1 - (ამ შემთხვევაში ეს სრულდება: 1 კენტია);

- ამობეჭდილი რიცხვი: 101
- ეს პროცედურა გაიმეორე $[1/2] = 0$ მონაცემისათვის:

მოცემულია: $n = 0$

- თუ $n = 0$, ალგორითმი დაასრულე - (ამ შემთხვევაში ეს სრულდება),
- ალგორითმი დასრულდა.

რიცხვების კოდირება კომპიუტერში. კოდირება ინფორმაციის დამუშავების ერთ-ერთი სახეობაა. კოდირების დროს იცვლება ინფორმაციის წარმოდგენის ფორმა, მაგრამ მისი შინაარსი შენარჩუნებულია. ინფორმაციის შესანახად და გადასაცემად, ის უნდა იყოს რაღაცნაირად დაშიფრული, მაგალითად, დავწეროთ რაიმე ნიშნების (სიმბოლოების) დახმარებით ზოგიერთ ენაზე.

კოდირება არის ინფორმაციის ისეთი ფორმით წარმოდგენა, რომელიც მოსახერხებელია შენახვის, გადაცემისა და ავტომატური დამუშავებისთვის.

კომპიუტერში ინფორმაცია წარმოდგენილია ელექტრონიკულსების სახით: 0-იანებისა და 1-იანების მიმდევრობის სახით, ამიტომაც მონაცემებს ციფრულს უწოდებენ. ინფორმაციის კოდირებისათვის გამოიყენება თვლის ორობითი სისტემა, რომელშიც მხოლოდ ორი ციფრია 0 და 1. ციფრული მონაცემის ერთი თანრიგი, რომელშიც შეიძლება ეწეროს 0 ან 1, წარმოადგენს ინფორმაციის ზომის მინიმალურ ერთეულს და მას **ბიტი** ეწოდება.

კოდირებისათვის გამოიყენება რვაბიტიანი კომბინაცია, რომელსაც ბაიტი ეწოდება. ინფორმაციის ძირითად საზომ ერთეულად ითვლება ბაიტი.

1 ბაიტი = 8 ბიტი.

ამასთანავე არსებობს ინფორმაციის საზომი უფრო დიდი ერთეულებიც: კილობაიტი, მეგაბაიტი, გიგაბაიტი, ტერაბაიტი.

1 კბ (კილობაიტი) ≈ 1024 ბ (ბაიტი);

1 მბ (მეგაბაიტი) ≈ 1024 კბ (კილობაიტი) $\approx 10^{20}$ კილობაიტი;

1 გბ (გიგაბაიტი) ≈ 1024 მბ (მეგაბაიტი) $\approx 10^{30}$ მეგაბაიტი;

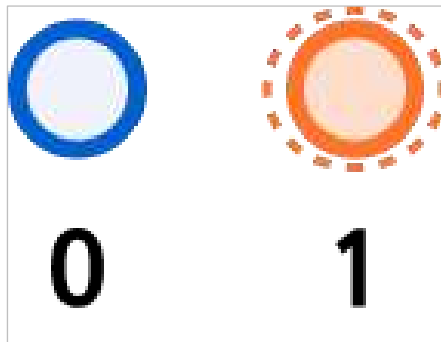
1 ტბ (ტერაბაიტი) ≈ 1024 გბ (გიგაბაიტი) $\approx 10^{40}$ გიგაბაიტი.

ფაილის ზომა მიუთითებს თუ კომპიუტერის მეხსიერების რა ადგილს იკავებს აღნიშნული ფაილი.

ფაილის ზომა დამოკიდებულია მასში არსებულ ინფორმაციის მოცულობაზე. ყველაზე მცირე ადგილს იკავებენ ტექსტური ფაილები, ხოლო ყველაზე დიდი ადგილს კი ვიდეო ფაილები, ამ შემთხვევაში მათ ზომას განსაზღვრავს ვიდეოს ხანგრძლივობა (წუთებში) და ვიდეოს ხარისხი (რეზოლუცია პიქსელებში).

როგორ ინახავს კომპიუტერი ციფრებს. მოდით გავიგოთ, თუ რა არის ორობითი (ბინარული) სისტემა და როგორ უნდა ჩავწეროთ ციფრები კომპიუტერში.

როგორც მოძველებულ, ისე სუპერთანამედროვე კომპიუტერის მეხსიერება ცალკეული ელემენტებისგან შედგება და აქვთ ერთი საერთო თვისება: ეს არის ორი მდგომარეობა, (როგორც ნათურასთან) ან ჩართული, ან გამორთული. წარმოიდგინეთ, რომ პირველი კომპიუტერის შემქმნელები ვართ. ჩვენი ამოცანაა ვიფიქროთ, თუ როგორ უნდა დავშივროთ ციფრები კომპიუტერის მეხსიერებაში არსებული ელემენტების მეშვეობით (ნახ. 6.2.).

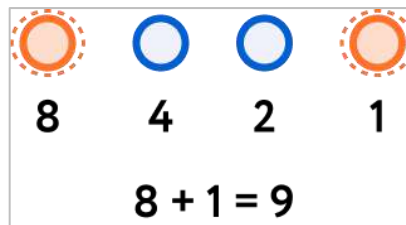


ნახ. 6.2. ნათურის ორი მდგომარეობა

ერთი ნათურით მხოლოდ ორი რიცხვის კოდირება შეგვიძლია: გამორთული, ჩართული. ასე რომ, კოდირება ხდება არა ცალკეული ელემენტების მიხედვით, არამედ ელემენტების ჯგუფების მიხედვით. მაგალითად, შეგვიძლია შევინახოთ რიცხვი, როგორც ჩართული ნათურების თანამიმდევრობა, რომლის სიგრძე რიცხვის ტოლია, ხოლო ციფრები ერთმანეთისგან გამორთული ნათურით გამვყოთ. ასევე შეგვიძლია შევიმუშაოთ ცხრილი, რომელზეც, წესების დაცვით ან წესების გარეშე, სადაც დავწერთ, თუ რომელი კოდი შეესაბამება თითოეულ ციფრს, მაგრამ არ უნდა დაგვავიწყდეს, რომ კომპიუტერმა არითმეტიკული ოპერაციები უნდა შეასრულოს რიცხვებთან.

კომპიუტერი ელექტრონული მოწყობილობაა და ამიტომ საჭიროა, რომ მეხსიერებაში რიცხვები ერთნაირი სიგრძის იყოს, ხოლო მათი კოდები ლოგიკური, ერთიანი წესებით აგებული, სწორედ ასეთ შემთხვევაში შესაძლებელი იქნება ისეთი ელექტრონული წრედის შექმნა, რომელიც მიიღებს შესასვლელზე ორ რიცხვს (ორი რიცხვის შეტანა), ხოლო გამოსასვლელზე კი, მაგალითად, მათ ჯამს.

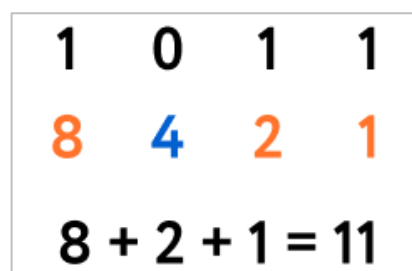
მოდით, ახლა მეხსიერების ელემენტები დავაჯგუფოთ ანუ გავაერთიანოთ ჯგუფებად. დავუშვათ რომ, რომ მარჯვენა ელემენტი ნიშნავს 1-ანს, შემდეგი 2-ანს, შემდეგი 4-ანს, 8-ანს და ასე შემდეგ ანუ ყოველი შემდეგი რიცხვი ორჯერ მეტია წინაზე, ვიდრე წინა რიცხვი. მაშინ, მაგალითად, რიცხვი 9-ანი გამოიყურება ისე, როგორც მოცემულია ნახ. 6.3.-ზე.



ნახ. 6.3. რიცხვი 9-ანის მიღების სქემა

9-ანის ასეთი ჩანაწერი შეიძლება რთულად მოგვეჩვენოს, მაგრამ მოდით შევადაროთ რიცხვთან 111: ვიცით, რომ მარჯვენა ერთეული ნიშნავს 1-ს, შემდეგი 10-ს, შემდეგი 100-ს. თუ რიცხვი უფრო გრძელი იქნებოდა, ერთეული „აიწონიდა“ 1000-ს, 10 000-ს და ასე შემდეგ, ყოველი შემდეგი რიცხვი წინაზე 10-ჯერ მეტი იქნებოდა. კომპიუტერული წარმოდგენით კი რიცხვი 10-ის ნაცვლად ჩვენ გვაქვს რიცხვი 2-ანი: სწორედ ეს არის ორიბითი (ბინარული) თვლის სისტემა. ადამიანი კი ყოველდღიურ ცხოვრებაში ათობითი თვლის სისტემას იყენებს, რადგან ასეთი ათვლის სისტემა კაცობრიობისათვის (დღეისდღეობით) უფრო მოსახერხებელია.

იმისათვის, რომ დავწეროთ რიცხვი ბინარულ სისტემაში, საჭიროა წარმოვიდგინოთ იგი როგორც 1, 2, 4, 8 და ა.შ. რიცხვების ჯამი. ნებისმიერი რიცხვი ამ წარმოდგენაში შეიძლება დაწერილი იყოს მხოლოდ ერთგვარად. მაგალითად, $13 = 8 + 4 + 1 = 1101_2$ სახით. ქვედა ინდექსი ნიშნავს, რომ რიცხვი ჩაწერილია ბინარულ სისტემაში. იმის გასაგებად, თუ რომელი რიცხვია ჩაწერილი ბინარულ სისტემაში, საჭიროა გავარკვიოთ, თუ რას ნიშნავს თითოეული ერთეული. ამისათვის მარჯვნიდან მარცხნივ 1101-ის ქვეშ მივაწეროთ 1, 2, 4, 8 და ა.შ. რიცხვები და შემდეგ დავაჯამოთ მხოლოდ ის რიცხვები, რომლებიც ერთის ქვეშ არის დაწერილი (ნახ. 6.4.).



ნახ. 6.4. ორობით სისტემაში რიცხვის შეკრების სქემა

ამოცანა: რომელი რიცხვია ჩაწერილი ნახატ 6.5.-ზე მოცემულ სქემაზე:



ნახ. 6.5. გამოსაცნობი რიცხვის სქემა

ამოხსნა: $00100001_2 = 32 + 1 = 33$.

პასუხი: გამოსაცნობი რიცხვი არის 33.

რიცხვების კოდირება კომპიუტერში. თვლის ორობითი, რვაობითი და თექვსმეტობითი სისტემები.

ადამიანი აზროვნებს თვლის ათობით სისტემაში ანუ მას უადვილდება ოპერაციების ჩატარება ისეთ რიცხვებზე რომლებიც წარმოდგება მხოლოდ 10 ციფრისაგან 0-დან 9-მდე.

ათობითი რიცხვებია: **500, 10, 1, 59, 627**

თვლის სისტემები შეიძლება იყოს უამრავი. რაც შეეხება ციფრულ ტექნიკას და მათ შორის კომპიუტერებსაც ისინი „აზროვნებენ“ თვლის ორობით სისტემაში ანუ მათში ინფორმაცია შეიძლება წარმოვადგინოთ მხოლოდ ორი ციფრით: ნულიანით და ერთიანით.

ორობითი რიცხვებია: **0110, 101, 110101, 00001, 10**

თუ რას წარმოადგენს თითოეული ეს ჩანაწერი ამას შემდეგში დაწვრილებით განვიხილავთ. ხშირად ამ ციფრებს ასეც უწოდებენ „ლოგიკური 0“ და „ლოგიკური 1“. ისინი შეესაბამებიან ძაბვის ორ განსხვავებულ მნიშვნელობას: დაბალი (ლოგიკური 0) და მაღალი (ლოგიკური 1).

თვლის სისტემების შესწავლა აუცილებელია იმისათვის, რომ ხშირად პროგრამირებაში საჭირო ხდება ე. წ. რვაობითი და თექვსმეტობით რიცხვებთან მუშაობა. ისინი ძირითადად შემოღებულია ორობითი რიცხვების მარტივად წარმოდგენის მიზნით.

უშუალო დიალოგისგან განსხვავებით კომპიუტერული დოკუმენტაციის კითხვისას გამოწვეული, შესაძლო ვირტუალური, მორალურ-იურიდიული ექსცესების თავიდან აცილებისას, წინასწარ შევთანხმდეთ:

პირველი: გამოთვლებისას რიცხვებს სასურველია ყოველთვის მივუწეროთ ინდექსი რომლითაც მივუთითებთ თუ რა სისტემაშია იგი წარმოდგენილი, საწინააღმდეგო შემთხვევაში შეიძლება გამოთვლები ჩავატაროთ შეცდომით.

მაგალითად, გაუგებარია შემდეგი სახით ჩაწერილი რიცხვი **101** რა სისტემაშია წარმოდგენილი: ორობითში, ათობითში, რვაობითში თუ თექვსმეტობითში. ასეთი ციფრები **1** და **0** გვაქვს როგორც ორობით, ასევე რვაობით და თექვსმეტობით სისტემებშიც. ამიტომ, თუ ის **რვაობითშია** უკეთესია მივუწეროთ ინდექსი ე.ი. **101₈** ან მაგალითად, რიცხვი **305** აშკარაა, რომ არ არის **ორობით** სისტემაში, მაგრამ დანარჩენ სისტემებზე ამას ვერ ვიტყვით. მაგალითად, **1101₂** - არის ორობითში, **1001₁₀** - არის ათობითში.

მეორე: ორობითი, რვაობითი და თექვსმეტობითი რიცხვები არ იკითხება ისე, როგორც ათობითი, არამედ უნდა წავიკითხოთ მხოლოდ ციფრებად.

მაგალითად **0110₂** წაიკითხება როგორც “ნული, ერთი, ერთი, ნული” და არა “ასათი”. ასევე **2731₈** წაიკითხება “ორი, შვიდი, სამი, ერთი” და არა “ორიათას შვიდას ოცდათერთმეტი”.

რვაბიტანი რიცხვებით კოდირებისათვის არსებობს საერთაშორისო შეთანხმებით მიღებული ცხრილი, რომელსაც ეწოდება **ASCII** (American Standart Code for Information Inrerchange).

ასევე არსებობს თექვსმეტბიანი კომბინაციებით კოდების სისტემა, რომელსაც ეწოდება **UNICODE**. ყველა შესაძლო 16-ბიტბიანი კომბინაციების რაოდენობა არის 65536, რაც საკმარისია ყველა არსებული ანბანის სიმბოლოთა კოდირებისათვის.

თვლის სისტემა, ნუმერაცია - რიცხვების სახელწოდებისა და აღნიშვნის ხერხების ერთობლიობაა. რიცხვები ჩაიწერება ნიშანთა სისტემის გამოყენებით, რომელთაც თვლის სისტემა ეწოდებათ. თვლის სისტემის ალფავიტი შედგება სიმბოლოებისგან, რომელთაც ციფრები ეწოდებათ.

დროთა განმავლობაში შეიქმნა მრავალი თვლის სისტემა. ისინი შეიძლება დავყოთ ორ ჯგუფად: პოზიციურ და არაპოზიციურ თვლის სისტემებად.

არაპოზიციურ სისტემებში, რიცხვის ჩაწერისას ციფრის (რომელიც გარკვეულ რაოდენობას შეესაბამება) მდებარეობას, პოზიციას არ აქვს მნიშვნელობა.

არაპოზიციური თვლის სისტემის მაგალითია რომაული თვლის სისტემა და გამოყენება შემდეგი სიმბოლოები: ***I – 1, V – 5, X – 10, L – 50, C – 100, D – 500, M – 1000*** და ა.შ.

მაგალითად: ***LXXVII = 50 + 10 + 10 + 5 + 1 + 1 = 77, IXC = 100 – 10 – 1 = 89***

თვლის პოზიციურ სისტემაში ციფრის პოზიცია რიცხვში განმსაზღვრელია. ციფრის მდებარეობას რიცხვში აქვს გარკვეული „წონა“. განვიხილოთ ათობითი თვლის სისტემაში რიცხვის **64532**-ის ჩაწერა:

$$\begin{aligned} 64532 &= 6 * 10000 + 4 * 1000 + 5 * 100 + 3 * 10 + 2 * 1 = \\ &= 6 * 10^4 + 4 * 10^3 + 5 * 10^2 + 3 * 10^1 + 2 * 10^0 \end{aligned}$$

გვაქვს 2 ერთეული, 3 ათეული, 5 ასეული და ა.შ. ამრიგად, ციფრის რაოდენობრივი მნიშვნელობა დამოკიდებულია მის პოზიციაზე რიცხვში. ციფრის პოზიციას რიცხვში თანრიგი ეწოდება. თანრიგი იზრდება მარჯვნიდან მარცხნივ. ციფრების რაოდენობას თვლის სისტემაში თვლის ფუძე (ბაზისი) ეწოდება.

ათობით სისტემაში გვაქვს ათი ციფრი: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**. რიცხვის ჩაწერისას მიეთითება ფუძე შემდეგი სახით **64532₁₀**. დღეს ყოველდღიურ ცხოვრებაში ვიყენებთ ათობით თვლის სისტემას. ზოგადად, ***p***-ობით (***p* > 1**) თვლის სისტემაში, გვაქვს ***p*** ციფრი (**0, 1, 2, ..., p – 1**). თანრიგის „წონა“ ***p***-ჯერ იზრდება (მცირდება) მარჯვნიდან მარცხნივ ერთი თანრიგით გადაადგილებისას. მთელი ***n***-თანრიგბიანი რიცხვი ***p***-ობით თვლის სისტემაში ასე გამოისახება:

$$N_{(p)} = a_{n-1} * p^{n-1} + a_{n-2} * p^{n-2} + \dots + a_1 * p^1 + a_0 * p^0, \text{ სადაც } 0 \leq a_i \leq p - 1$$

ინფორმატიკაში გამოიყენება ორობითი, რვაობითი და თექვსმეტობითი თვლის სისტემები.

ორობითი თვლის სისტემის ფუძე $p = 2$, ციფრები 0,1

$$11001110_{(2)} = 1 * 2^7 + 1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$$

რვაობითი თვლის სისტემის ფუძე $p = 8$, ციფრები 0, 1, 2, 3, 4, 5, 6, 7

$$2457_{(8)} = 2 * 8^3 + 4 * 8^2 + 5 * 8^1 + 7 * 8^0$$

თექვსმეტობითი თვლის სისტემის ფუძე $p = 16$, ციფრები 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

$$4A8D2_{(16)} = 4 * 16^4 + A * 16^3 + 8 * 16^2 + D * 16^1 + 2 * 16^0$$

თექვსმეტობით სისტემაში გვაქვს 16 ციფრი, გარდა ჩვეულებრივი ციფრებისა, არის ლათინური ალფავიტის პირველი 6 ასო ($A = 10, B = 11, C = 12, D = 13, E = 14, F = 15$)

მოდით ახლა უფრო დაწვრილებით და მაგალეთების დახმარებით განვიხილოთ თვლის სისტემები და ერთი სისტემიდან მეორე სისტემაში რიცხვების გადაყვანა.

თვლის ათობითი სისტემა. როგორც უკვე აღვნიშნეთ კომპიუტერი ცნობს ლოგიკურ 1-ანებსა და 0-ანებს. საინტერესოა, როგორ მიიღება მათი საშუალებით რეალური რიცხვები ან ყველა ის პროცესი რასაც ვხედავთ მონიტორზე კომპიუტერული პროგრამის მუშაობის დროს. თვლა ორობით რიცხვებში ემყარება იმავე პრინციპს, რასაც ნებისმიერი სხვა თვლის სისტემა გვთავაზობს. თვლა იწყება ციფრიდან 0 და ყოველი შემდეგი რიცხვი მიიღება ინკრემენტაციის (საფეხურებრივი ზრდის, ერთით გაზრდის) გზით. კერძოდ, თუ ორობითი რიცხვის ყველა პოზიციაზე 1-ანებია, მაშინ ეს უკანასკნელები 0-ებად იქცევიან და მარცხნიდან დაემატება 1-ის ტოლი ახალი პოზიცია, საწინააღმდეგო შემთხვევაში მარჯვენა განაპირა პოზიციიდან მოყოლებული ყველა 0 თანამიმდევრობით იქცევა ერთიანად (ცხრილი 6.1.).

ცხრილი 6.1. შესაბამისობა ორობით და ათობით რიცხვებს შორის

ორობითი	00	01	10	11	100	101	110	111
ათობითი	0	1	2	3	4	5	6	7
ორობითი	1000	1001	1010	1011	1100	1101	1110	1111
ათობითი	8	9	10	11	12	13	14	15

რა კანონზომიერებით მიიღება ასეთი შესაბამისობა? არსებობს გარდაქმნის წესები რომელიც საშუალებას იძლევა გადავიდეთ თვლის ერთი სისტემიდან მეორეში ანუ ათობითი ხუთიდან მივიღოთ ორობითი 101 ან პირიქით. ამისათვის ჯერ უნდა ვნახოთ თუ რას წარმოადგენს ჩვენთვის კარგად ცნობილი თვლის ათობითი სისტემა. რიცხვს რომელიც

ტოლია თვლის მოცემულ სისტემაში ერთმანეთისგან განსხვავებულ ციფრთა რაოდენობისა უწოდებენ სისტემის ფუძეს. ათობით სისტემაში გვაქვს 10 განსხვავებული ციფრი (0, 1, 2...9), ე.ი. ამ სისტემის ფუძეა 10, ორობით სისტემაში გვაქვს ორი ციფრი (0 და 1) ე.ი. ამ სისტემის ფუძეა 2. როგორ შეიძლება წარმოვადგინოთ ნებისმიერი ათობითი რიცხვი სხვა სახით.

რიცხვი 473 შეგვიძლია დავშალოთ შესაკრებებად:

$$473_{10} = 4 * 10^2 + 7 * 10^1 + 3 * 10^0$$

არსებობს ე.წ. **პოზიციის ცნება**, რიცხვში ციფრის პოზიციაზე. 473-ში 4-ის პოზიციაა 2, 7-ისა 1, ხოლო 3-ისა 0. ე.ი. პოზიციის თვლა იწყება 0-დან და არა 1-დან. ფუძის (10-ის) ხარისხები ამიტომაც შეესაბამებიან სამრავლის პოზიციებს.

არსებობს ასევე **წონის ცნება**. მოცემული ციფრის წონა არის ამ სისტემის ფუძე ხარისხად ამ ციფრის პოზიციისა ანუ 4-ის წონაა 10^2 , ხოლო 3-ის წონაა 10^0 .

როგორ წარმოვადგინოთ 10-ობითი წილადი რიცხვები. ასეთ შემთხვევაში მძიმის შემდგომი ციფრის წონა იწყება 10^{-1} - დან, მერე მოსდევს 10^{-2} და ა.შ. რიცხვი 21.457-თვის გვექნება: $21.457_{10} = 2 * 10^1 + 1 * 10^0 + 4 * 10^{-1} + 5 * 10^{-2} + 7 * 10^{-3}$

თვლის ორობითი სისტემა. თვლის ორობით სისტემაში რიცხვები წარმოიდგინებიან მხოლოდ ორი ციფრით: 0 და 1 და ორობითი სისტემის ფუძეა 2.

განვიხილოთ ერთი სისტემიდან მეორეში გადასვლის წესები.

ორობითი სისტემის ათობით სისტემაში გადასვლის წესი ასეთია: ორობითი რიცხვის თითოეული ციფრი მრავლდება თავის წონაზე და ნამრავლები იკრიბება.

მაგალითი 6.1. გადავიყვანოთ 101 და 1101.11 ორობითი რიცხვები ათობით რიცხვებში:

$$101_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 5_{10}$$

$$1101.11_2 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} = 13.75_{10}$$

ათობითი სისტემის ორობით სისტემაში გადაყვანის წესი კი ასეთია: ათობითი რიცხვი იყოფა 2-ზე, ნაშთი იქნება 1 ან 0, ხოლო გაყოფის შედეგი (მხოლოდ მთელი ნაწილი) კიდევ იყოფა 2-ზე, აქაც ნაშთი იქნება 0 ან 1 და ასე გრძელდება მანამ, სანამ გაყოფის შედეგი (მთელი ნაწილი) არ იქნება 1. ბოლოს მიღებული 1-ები და 0-ები (ბოლო გაყოფის შედეგის ჩათვლით) ლაგდება უკუ თანამიმდევრობით.

გადავიყვანოთ ათობითი 12_{10} შესაბამის ორობით რიცხვში:

$$12 : 2 = 6; \text{ ნაშთი } 12 - (6 * 2) = 0$$

$$6 : 2 = 3; \text{ ნაშთი } 6 - (3 * 2) = 0$$

$$3 : 2 = 1; \text{ ნაშთი } 3 - (1 * 2) = 1$$

$$1 : 2 = 0; \text{ ნაშთი } 1 - (0 * 2) = 1$$

მეორენაირად, ალგორითულ ენაზე, ათობითი სისტემის ორობით სისტემაში გადაყვანა შეიძლება წარმოვადგინოთ შემდეგი სახით:

გაყოფის მთელი ნაწილი	გაყოფის ნაშთი
$12 \div 2 = 6$	$12 \bmod 2 = 0$
$6 \div 2 = 3$	$6 \bmod 2 = 0$
$3 \div 2 = 1$	$3 \bmod 2 = 1$
$1 \div 2 = 0$	$1 \bmod 2 = 1$

მიღებული ნაშთი დავალაგოთ უკუ თანმიმდევრობით:

$$1100 \text{ ანუ } 12_{10} = 1100_2$$

$$\text{პასუხი: } 12_{10} = 1100_2$$

მაგალითი 6.2. გადავიყვენოთ ათობითი სისტემის რიცხვი **27. 53** ორობით სისტემაში, ჯერ გარდავეყმნათ რიცხვის მთელი ნაწილი **27**, შემდეგ რიცხვის წილადი **0. 53** ნაწილი.

მთელის გარდაქმნა უკვე ვნახეთ, წილადი გარდაიქმნება ასეთი წესით: წილადი ნაწილი მრავლდება **2**-ზე, მიღებული შედეგის წილადი ნაწილი ისევ მრავლდება **2**-ზე და ა.შ. მანამ, სანამ არ იქნება მიღწეული სასურველი სიზუსტე ან სანამ გამრავლების შედეგი არ იქნება **1**-ის ტოლი.

განვიხილოთ მაგალითი ორ ეტაპად:

პირველი ეტაპი: მთელი ნაწილი **27**

გაყოფის მთელი ნაწილი	გაყოფის ნაშთი
$27 \div 2 = 13$	$27 \bmod 2 = 1$
$13 \div 2 = 6$	$13 \bmod 2 = 1$
$6 \div 2 = 3$	$6 \bmod 2 = 0$
$3 \div 2 = 1$	$3 \bmod 2 = 1$
$1 \div 2 = 0$	$1 \bmod 2 = 1$

$$\text{ე.ი. მივიღეთ } 27_{10} = 11011_2$$

მეორე ეტაპი: წილადი ნაწილი **0.53**

$$0.53 * 2 = 1.06 \text{ (მთელი ნაწილი } 1)$$

$$0.06 * 2 = 0.12 \text{ (მთელი ნაწილი } 0)$$

$$0.12 * 2 = 0.24 \text{ (მთელი ნაწილი } 0)$$

$$0.24 * 2 = 0.48 \text{ (მთელი ნაწილი } 0)$$

$$\text{მივიღეთ } 1000 \text{ ე.ი. } 0.53_{10} = 0.1000_2$$

საბოლოოდ მივიღეთ: **11011.1000₂**

$$\text{პასუხი: } 27.53_{10} = 11011.1000_2$$

თვლის რვაობითი სისტემა. იმისათვის, რომ რაიმე რიცხვი წარმოვადგინოთ ორობით სისტემაში, საჭიროა ერთიანებისა და ნოლიანების საკმაოდ დიდი რაოდენობა. ამიტომ

ორობითი რიცხვების მარტივად ჩაწერის მიზნით შემოღებულია თვლის რვაობითი და თექვსმეტობითი სისტემები. გარდა ამისა მათ აქვთ სხვა დანიშნულებაც, რაც ძირითადად დაკავშირებულია კომპიუტერის მეხსიერების დამისამართებასთან.

რვაობით თვლის სიტემაში გავაქვს მხოლოდ 8 ციფრი: **0, 1, 2, 3, 4, 5, 6, 7**

ნებისმიერი ათობითი რიცხვი შეგვიძლია წარმოვადგინოთ ამ რვა ციფრის საშუალებით. მაგალითად, რვაობითი რიცხვებია: **014, 7037, 0001, 10034, 5612**

ერთიდან მეორე სისტემაზე გადასვლის წესები ანალოგიურია ორობითი სისტემისა, ოღონდ ცხადია აქ გაყოფა ხდება რვაზე და წონა არის 8^n , სადაც n არის ციფრის პოზიცია რვაობით რიცხვში. როგორ ჩავწეროთ ორობითი რიცხვი რვაობითში?

ორობითი რიცხვის დაჯგუფება ხდება სამ-სამად (ტრიადა) და თითოეული სამეულისთვის იწერება რვაობითი რიცხვი შესაბამისობის მიხედვით (ცხრილი 6.2.):

ცხრილი 6.2. შესაბამისობა ორობით და რვაობით რიცხვებს შორის

ორობითი	000	001	010	011	100	101	110	111
რვაობითი	0	1	2	3	4	5	6	7

დაჯგუფება ხდება მარჯვნიდან მარცხნივ. მაგალითად, **011101**-ის დაჯგუფებით მივიღებთ: **011101₂ = 35₈** ე. ი. ჯერ ავიღეთ ჯგუფი **101** და დავწეროთ შესაბამისი რვაობითი რიცხვი **5**, შემდეგ ავიღეთ **011** და დავწეროთ **3**.

ასევე, **100101101₂**-სათვის მივიღებთ: **100101101₂ = 455₈**.

თუ ციფრების რაოდენობა ისეთია, რომ არ ხერხდება დაჯგუფება სამად, მაშინ მარცხნიდან ემატება ციფრი **0**, სანამ არ შეივსება სამ ციფრამდე:

$$1_2 = 001_2 = 1_8 \quad \text{ან} \quad 1010_2 = 001010_2 = 12_8$$

მაგალითი 6.3. გადავიყვენოთ ათობითი რიცხვი **12** რვაობითში ანუ **12₁₀ = ?₈**

გაყოფის მთელი ნაწილი გაყოფის ნაშთი

$$12 \div 8 = 1 \qquad 12 \bmod 8 = 4$$

$$1 \div 8 = 0 \qquad 1 \bmod 8 = 1$$

ე.ი. ათობითი რიცხვი **12**-ის რვაობითში გადაყვანით მივიღეთ **14**

პასუხი: **12₁₀ = 14₈**

მაგალითი 6.4. გადავიყვანოთ რვაობითი რიცხვი **23** ათობითში ანუ **23₈ = ?₁₀**

რვაობითი რიცხვის თითოეული ციფრი მრავლდება თავის წონაზე და ნამრავლები იკრიბება:

$$23_8 = 8^1 * 2 + 8^0 * 3 = 16 + 3 = 19_{10}$$

პასუხი: **23₈ = 19₁₀**

მაგალითი 6.5. გადავიყვანოთ ათობითი რიცხვი **27.53** რვაობითში ანუ **27.53₁₀ = ?₈**

გაყოფის მთელი ნაწილი

გაყოფის ნაშთი

$$27 \text{ div } 8 = 3$$

$$27 \text{ mod } 8 = 3$$

$$3 \text{ div } 8 = 0$$

$$3 \text{ mod } 8 = 3$$

$$0 \text{ div } 8 = 0$$

$$0 \text{ mod } 8 = 0$$

ე. ი. $27_{10} = 033_8$

$0.53 * 8 = 4.24$ (მთელი ნაწილი 4)

$0.24 * 8 = 1.92$ (მთელი ნაწილი 1)

$0.92 * 8 = 7.36$ (მთელი ნაწილი 7)

$0.36 * 8 = 2.88$ (მთელი ნაწილი 2)

მივიღეთ: 4172 ე.ი. $0.53_{10} = 0.4172_8$

საბოლოოდ კი მივიღეთ: 033.4172_8

პასუხი: $27.53_{10} = 033.4172_8$

მაგალითი 6.6. გადავიყვანოთ რვაობითი რიცხვი 27.53 ათობითში ანუ $27.53_8 = ?_{10}$

$$27_8 = 2 * 8^1 + 7 * 8^0 = 16 + 7 = 23_{10}$$

$$53_8 = 5 * 8^{-1} + 3 * 8^{-2} = 0.625 + 0.046875 = 0.671875_{10}$$

პასუხი: $27.53_8 = 23.671875_{10}$

მაგალითი 6.7. გადავიყვანოთ რვაობითი რიცხვი 27.53 ორობითში $27.53_8 = ?_2$

მაგალითი განვიხილოთ ორ ეტაპად.

თავდაპირველად რვაობითი სისტემიდან გადავიყვანოთ ათობითში, შემდეგ კი მიღებული შედეგი გადავიყვანოთ ორობითში.

პირველი ეტაპი:

გადავიყვანოთ 27.53_8 ათობითში:

$$\begin{aligned} 27.53_8 &= 2 * 8^1 + 7 * 8^0 + 5 * 8^{-1} + 3 * 8^{-2} = \\ &= 16 + 7 + 0.625 + 0.046875 = 23.671875_{10} \end{aligned}$$

მეორე ეტაპი:

მიღებული ათობითი რიცხვის მთელი ნაწილი 23_{10} გადავიყვანოთ ორობითში:

გაყოფის მთელი ნაწილი

გაყოფის ნაშთი

$$23 \text{ div } 2 = 11$$

$$23 \text{ mod } 2 = 1$$

$$11 \text{ div } 2 = 5$$

$$11 \text{ mod } 2 = 1$$

$$5 \text{ div } 2 = 2$$

$$5 \text{ mod } 2 = 1$$

$$2 \text{ div } 2 = 1$$

$$2 \text{ mod } 2 = 0$$

$$1 \text{ div } 2 = 0$$

$$1 \text{ mod } 2 = 1$$

მივიღეთ: $23_{10} = 10111_2$

მიღებული ათობითი რიცხვის წილადი ნაწილი 0.671875_{10} გადავიყვანოთ ორობითში:

გამრავლება	შედეგი	მთელი ნაწილი	ნაშთი
$0.671875 * 2$	1.34375	1	0.34375
$0.34375 * 2$	0.6875	0	0.6875
$0.6875 * 2$	1.375	1	0.375
$0.375 * 2$	0.75	0	0.75
$0.75 * 2$	1.5	1	0.5
$0.5 * 2$	1	1	

მივიღეთ: $0.671875_{10} = 0.101011_2$

პასუხი: $27.53_8 = 10111.101011_2$

მაგალითი 6.8. გადავიყვანოთ 375_8 ათობითში და პირიქით 375_{10} რვაობითში.

$$375_8 = 3 * 8^2 + 7 * 8^1 + 5 * 8^0 = 192 + 56 + 5 = 253_{10}$$

პასუხი: $375_8 = 253_{10}$

$375/8 =$ მთელი 46, ნაშთი 7

$46/8 =$ მთელი 5, ნაშთი 6

$5/8 =$ მთელი 0, ნაშთი 5

პასუხი: $375_{10} = 567_8$

თვლის თექვსმეტობითი სისტემა. თვლის თექვსმეტობით სისტემაში გვაქვს 16 ციფრი: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**

თექვსმეტობითი რიცხვებია: **0AF45, 768C, FFFFF, AB50**

გარდაქმნის წესები იგივეა, რაც ორობით სისტემაში, ოღონდ აქ ფუძე არის 16, ხოლო წონა 16^n , სადაც n არის ციფრის პოზიცია თექვსმეტობით რიცხვში, გაყოფაც ხდება 16-ზე.

ახლა როგორ წარმოვადგინოთ ორობითი რიცხვი თექვსმეტობით თვლის სისტემაში;

მოვიქცეთ ისე, როგორც რვაობითში, ოღონდ აქ დაჯგუფება ხდება ოთხ-ოთხად ანუ ტეტრადად (ცხრილი 6.3.).

ათვლის თექვსმეტობით სისტემაშიც დაჯგუფება ხდება მარჯვნიდან მარცხნივ:

$$01111000_2 = 78_{16};$$

$$1101_2 = D_{16};$$

$$100010111111_2 = 8BF_{16};$$

ცხრილი 6.3. შესაბამისობა ორობით და თექვსმეტობით რიცხვებს შორის

ორობითი	0000	0001	0010	0011	0100	0101	0110	0111
თექვსმეტობითი	0	1	2	3	4	5	6	7
ორობითი	1000	1001	1010	1011	1100	1101	1110	1111
თექვსმეტობითი	8	9	A	B	C	D	E	F

როგორც ვხედავთ, რვაობით და თექვსმეტობით სისტემებში ორობითი რიცხვების წარმოდგენა აშკარად მოკლდება.

დავიმახსოვროთ: ერთი სისტემიდან მეორეში გადასვლისას მოსახერხებელია გარდაქმნა მოხდეს ორობითი სისტემის გავლით.

თუ საქმე გვაქვს რიცხვის წილად ნაწილთან და დაჯგუფება არ ხერხდება, მაშინ აქაც უნდა მივუწეროთ ნულები, ოღონდ ამ შემთხვევაში მარჯვნიდან. ეს იმას ნიშნავს, რომ დაჯგუფება წილადური ნაწილის შემთხვევაში ხდება მარცხნიდან მარჯვნივ.

მაგალითი 6.9. გადავიყვანოთ თექვსმეტობითი რიცხვი 27.53_{16} ორობითში.

$$27_{16} = 00100111_2$$

$$53_{16} = 01010011_2$$

$$\text{პასუხი: } 27.53_{16} = 00100111.01010011_2$$

მაგალითი 6.10. გადავიყვანოთ ორობითი რიცხვი 00100111.01010011_2 ათობითში.

$$\begin{aligned} 00100111_2 &= 0 * 2^7 + 0 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = \\ &= 0 + 0 + 32 + 0 + 0 + 4 + 2 + 1 = 39_{10} \end{aligned}$$

$$\begin{aligned} 01010011_2 &= 0 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4} + 0 * 2^{-5} + 0 * 2^{-6} + 1 * 2^{-7} + \\ &+ 1 * 2^{-8} = 0.32421875_{10} \end{aligned}$$

მივიღეთ: 39.32421875

$$\text{პასუხი: } 00100111.01010011_2 = 39.32421875_{10}$$

მაგალითი 6.11. გადავიყვანოთ 27.53_{16} თექვსმეტობიდან ათობითში.

$$27_{16} = 2 * 16^1 + 7 * 16^0 = 32 + 7 = 39_{10}$$

$$53_{16} = 5 * 16^{-1} + 3 * 16^{-2} = 0.32421875_{10}$$

$$\text{პასუხი: } 27.53_{16} = 39.32421875_{10}$$

მაგალითი 6.12. გადავიყვანოთ თექვსმეტობითი რიცხვი $7A.7E_{16}$ ორობით სისტემაში.

თავდაპირველად რიცხვი გადავიყვანოთ ათობით სისტემაში:

$$\begin{aligned} 7A.7E_{16} &= 7 * 16^1 + 10 * 16^0 + 7 * 16^{-1} + 14 * 16^{-2} = \\ &= 112 + 10 + 0.4375 + 0.0546875 = 122.4921875_{10} \end{aligned}$$

ახლა კი მიღებული ათობითი რიცხვი გადავიყვანოთ ორობითში. ჯერ გადავიყვანოთ მთელი ნაწილი:

$$122/2 = \text{მთელი } 61, \text{ნაშთი } 0$$

$$61/2 = \text{მთელი } 30, \text{ნაშთი } 1$$

$$30/2 = \text{მთელი } 15, \text{ნაშთი } 0$$

$$15/2 = \text{მთელი } 7, \text{ნაშთი } 1$$

$$7/2 = \text{მთელი } 3, \text{ნაშთი } 1$$

$$3/2 = \text{მთელი } 1, \text{ნაშთი } 1$$

$$1/2 = \text{მთელი } 0, \text{ნაშთი } 1$$

$$\text{მივიღეთ: } 122_{10} = 1111010_2$$

ეხლა გადავიყვანოთ წილადი ნაწილი:

$$0.4921875 * 2 = 0.984375, \quad \text{მთელი } 0, \quad \text{ნაშთი } 0.984375$$

$$0.984375 * 2 = 1.96875, \quad \text{მთელი } 1, \quad \text{ნაშთი } 0.96875$$

$$0.96875 * 2 = 1.9375, \quad \text{მთელი } 1, \quad \text{ნაშთი } 0.9375$$

$$0.9375 * 2 = 1.875, \quad \text{მთელი } 1, \quad \text{ნაშთი } 0.875$$

$$0.875 * 2 = 1.75, \quad \text{მთელი } 1, \quad \text{ნაშთი } 0.75$$

$$0.75 * 2 = 1.5, \quad \text{მთელი } 1, \quad \text{ნაშთი } 0.5$$

$$0.5 * 2 = 1, \quad \text{მთელი } 1$$

$$\text{მივიღეთ: } 0.4921875_{10} = 0.0111111_2$$

$$\text{პასუხი: } 7A.7E_{16} = 1111010.0111111_2$$

ძნელად ამოსახსნელი ამოცანები. პოლინომიალური და ექსპონენციალური ალგორითმები, მაგალითები

ამოცანათა ამოხსნის ყველა ალგორითმი შეგვიძლია შევაფასოთ ოთხი კრიტერიუმის საშუალებით:

1. **სისრულე.** იძლევა თუ არა ალგორითმი ამოხსნის პოვნის გარანტიას, როდესაც ეს ამოხსნა არსებობს?
2. **ოპტიმალურობა.** უზრუნველყოფს თუ არა ალგორითმი ოპტიმალური ამონახსნის მოძებნას?
3. **დროითი სირთულე.** რა დროში შეუძლია ალგორითმს ამონახსნის მოძებნა?
4. **სივრცითი სირთულე.** რა მოცულობის მეხსიერება სჭირდება ალგორითმს ამონახსნის საპოვნელად?

დროითი და სივრცითი სირთულეები ყოველთვის იანგარიშება ამოცანის სირთულის გათვალისწინებით, მაგრამ არსებობს ისეთი სახის ამოცანები, რომლებთაც არ გააჩნიათ გადაწყვეტის ალგორითმი ანუ ალგორითმულად ამოუხსნადია და მაშინაც კი, თუ ალგორითმი არსებობს, ის შეიძლება ვერ გამოვიყენოთ პრაქტიკაში მისი მაღალი სირთულის გამო.

პოლინომიალური და ექსპონენციალური ალგორითმები

მარტივად ამოხსნადი ამოცანები არის ისეთი ამოცანები, რომელთა ამოხსნა შესაძლებელია პოლინომიალურ დროში.

რთულად ამოხსნადი ამოცანები არის ამოცანები, რომელთა ამოხსნაც შეუძლებელია პოლინომიალურ დროში ან არ არის ნაპოვნი პოლინომიურ დროში მათი ამოხსნის ალგორითმი. ამოცანის სირთულე არ განისაზღვრება და არ გამოიხატება მხოლოდ საუკეთესო ალგორითმის სირთულის შეფასებით.

სირთულის შესაფასებლად შემოღებულია კლასიფიცირებული ფუნქციები, რომლის გამოთვლაც შესაძლებელია მოცემული შეზღუდვების ფარგლებში მოხმარებული რესურსებით.

პოლინომიალური სირთულის ალგორითმი ($O(n^k)$). პოლინომიალური სირთულის ალგორითმის შესრულების დრო დამოკიდებულია შეყვანილი მონაცემის ზომაზე n და იზრდება პოლინომიალური ფუნქციის ($O(n^k)$) მიხედვით, სადაც k არის ფიქსირებული კონსტანტა.

ალგორითმი, რომლის სირთულის ორივე ფუნქცია პოლინომიალურია, ეწოდება **პოლინომიალური სირთულის ალგორითმი**. პოლინომიალური სირთულის ალგორითმის მაგალითია მარტივი დახარისხების ალგორითმი (Bubble Sort).

ექსპოტენციალური სირთულის ალგორითმი - $O(2^n)$, $O(e^n)$, $O(n^n)$ და სხვა. თუ ექსპოტენციალური სირთულის ალგორითმის მუშაობის დრო არ იზრდება ვიდრე n -ის ექსპონანტა, მაშინ საუბარია ექსპონენციალურ ალგორითმზე.

ალგორითმი, რომელსაც სირთულის ორი ფუნქციიდან ერთ-ერთი ფუნქცია ექსპოტენციალურია, ეწოდება **ექსპოტენციალური სირთულის ალგორითმი**. ექსპოტენციალური სირთულის ალგორითმი უფრო უარესი შემთხვევაა, ვიდრე პოლინომიალური სირთულის ალგორითმი.

$O(n^2)$ - ეს პოლინომიალური დროა. მრავალწევრი ტოლია $f(n) = n^2$. მეორეს მხრივ $O(2^n)$ - ეს ექსპოტენციალური დროა, სადაც ექსპოტენციალური ფუნქცია $f(n) = 2^n$.

საბოლოოდ, პოლინომიალური სირთულის ალგორითმი არის სწრაფი. ეს მტკიცდება პრაქტიკიდან გამომდინარეობს.

ამოცანა, რომლის გადაწყვეტაც შეგვიძლია პოლინომიალური ალგორითმით, ითვლება რომ არის მარტივად ამოხსნადი.

ექსპოტენციალური სირთულის ალგორითმი არის საკმაოდ ნელი ალგორითმი. ამოცანა, რომლის ამოხსნაც შეუძლებელია პოლინომიალური სირთულის ალგორითმით, არის რთულად ამოხსნადი. რთულად ამოხსნად ამოცანების ტიპს განეკუთვნება ალგორითმულად ამოუხსნადი ამოცანები.

ამოცანა 6.1. გამოვთვალოთ ფიბონაჩის n -ური რიცხვი (პოლინომიალური სირთულე):

$$f_1 = 1, f_2 = 1, f_k = f_{k-2} + f_{k-1}$$

მიმდევრობის რეკურსიული წაკითხვა „ბოლოდან“: $(f_n, f_{n-1}, \dots)$

$$f_n = f_{n-2} + f_{n-1} \rightarrow f_{n-1} = f_{n-2} + f_{n-3} \rightarrow \dots f_3 = f_1 + f_2$$

ალგორითმის შესრულების სირთულეა: $O(2^n)$

ალგორითმი 1: (იხ. ცხრილი 6.5. და ნახ. 6.6.)

```
double FibFunc(int IndexNumber)           // ფუნქციას გადაეცემა რიცხვი
{
    if (IndexNumber == 1 || IndexNumber == 2) // ვამოწნებთ პირობას
        return 1.0;
    //თუ მივედით, მაშინ ვაბრუნებთ მნიშვნელობას
```



```

else //თუ არა, რეკურსიულად ვიძახებთ ფუნქციას ფორმულის გამოყენებით
return (FibFunc(IndexNumber - 1) + FibFunc(IndexNumber - 2)); }

```

ამოცანა 6.2. გამოვთვალოთ ფიბონაჩის n -ური რიცხვი (წრფივი სირთულე):

ციკლში წაკითხვა „თავიდან“ ($f_1, f_2 \dots$)

$$f_3 = f_1 + f_2 \rightarrow f_4 = f_2 + f_3 \rightarrow \dots f_n = f_{n-2} + f_{n-1}$$

ალგორითმის შესრულების სირთულე არის წრფივი $O(n)$

ალგორითმი 2:

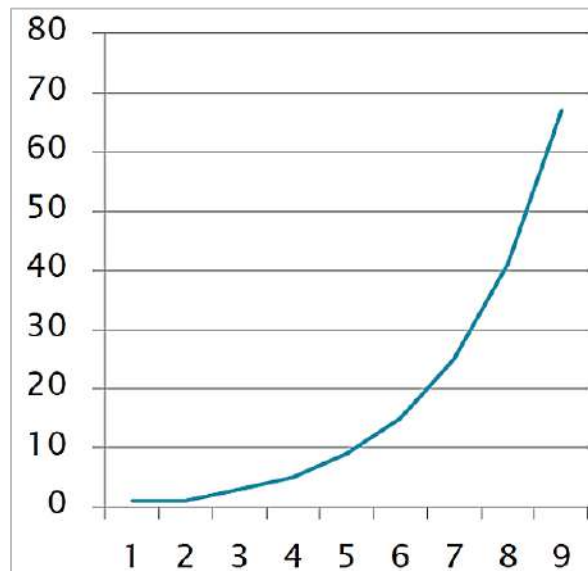
```

double FibFunc(int IndexNumber)
{
double f1, f2, f3; // ცვლადების გამოცხადება
f1 = f2 = 1.0;    // პირველი და მეორე რიცხვის ინიციალიზირება
f3 = 0.0;
int i = 3;        // ბიჯის გამოცხადება და ინიციალიზება
while (i <= IndexNumber) // სანამ არ მივაღწიოთ ფიბონაჩის სასურველ ნომრამდე
{
f3 = f2 + f1; // განვსაზღვროთ i-ური რიცხვი ფორმულით
// ცვლადების შეცვლა მომდევნო ბიჯისათვის
f1 = f2;      // f2 ხდება f1
f2 = f3;      // f3 ხდება f2
i = i + 1;    // ბიჯის გაზრდა ერთით }
return f3; // ვაბრუნეთ ბოლო გამოთვლილი მნიშვნელობა
}

```

ცხრილი 6.5. ფიბონაჩის ფუნქციის რეკურსიულად გამოძახების რაოდენობა

ფუნქციის გამოძახების რაოდენობა		
რიცხვის ნომერი	ფიბონაჩის რიცხვი	ფუნქციის გამოძახების რაოდენობა
1	1	1
2	1	1
3	2	3
4	3	5
5	5	9
6	8	15
7	13	25
8	21	41
9	34	67



ნახ. 6.6. დამოკიდებულება რეკურსიულად გამოძახებული ფიბონაჩის ფუნქციის რაოდენობასა და რიცხვებს შორის

ამოცანა 6.3. გამოვთვალოთ a^n (იხ. ცხრილი 6.6. და ნახ. 6.7.)

ალგორითმი 3: გამოვთვალოთ a^n (წრფივი სირთულე)

წავიკითხოთ ციკლში $a, a^2, a^3, \dots$

ალგორითმის შესრულების სირთულე $O(n)$

ალგორითმი 4: გამოვთვალოთ a^n (ლოგარითმული სირთულე)

გამოვთვალოთ თვისების გამოყენებით: $a^{2k} = (a^2)^k$

ალგორითმის შესრულების სირთულე $O \log_2 n$

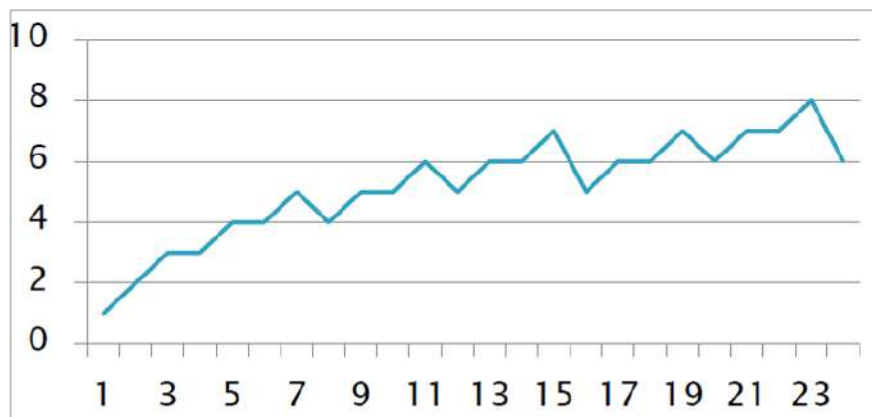
long mypow(long a, long k)

```
{
    long b = 1;
    while(k != 0 )
    {
        if ( k % 2 == 0 )
        {
            k /= 2; a *= a; }

        else
        {
            k--; b *= a; }
    }
    return b;
}
```

ცხრილი 6.6. ახარისხების ფუნქციის იტერაციულად გამოძახების რაოდენობა

იტერაციების რაოდენობა	
ხარისხი	იტერაციების რიცხვი
1	1
2	2
3	3
4	3
5	4
6	4
7	5
8	4
9	5
10	5
11	6
12	5
13	6
14	6



ნახ. 6.7. დამოკიდებულება იტერაციულად გამოძახებული ახარისხების ფუნქციის იტერაციების რაოდენობასა და ახარისხების რიცხვს შორის

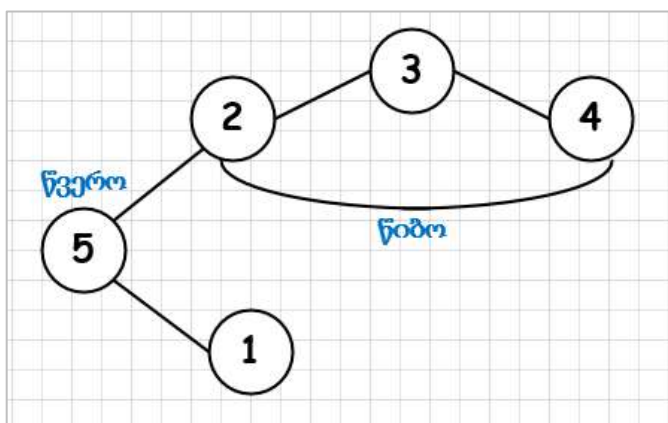
ორიენტირებული აციკლური გრაფები. ტოპოლოგიური სორტირება, ორიენტირებული გრაფების წარმოდგენა ტოპოლოგიური სორტირების მუშაობის დროს. PERT დიაგრამაში კრიტიკული გზა. უმოკლესი გზა ორიენტირებულ აციკლურ გრაფში.

სიტყვა გრაფი ბერძნული წარმოშობისაა და ნიშნავს: ვხატავ, ვწერ, ვხაზავ. გრაფები, პროგრამისტების აზრით, არ არის ის დიაგრამები ან ჰისტოგრამები, რომელთაც სკოლაში ვსწავლობდით.

კომპიუტერული მეცნიერებისა და დისკრეტული მათემატიკის თვალსაზრისით, გრაფი უფრო არის ურთიერთობის წარმოდგენის აბსტრაქტული ხერხი, მაგალითად, როგორიცაა ქალაქების დამაკავშირებელი გზები ან სხვა და სხვა სახის ქსელები. გრაფი შედგება წიბოებისა და წვეროებისგან. წვერო არის წერტილი გრაფზე, ხოლო წიბო არის ის, რაც აკავშირებს გრაფის ორ წერტილს. წვერო აღინიშნება ან ლათინური ალფავიტის სიმბოლოებით ან რიცხვებით, თუმცა შესაძლებელია წვეროს დასახელების აღნიშვნა განვხორციელოთ ქართული (მშობლიური) ანბანით (იხ. ნახ. 7.1.).

გრაფის განსაზღვრა. გრაფი (graph) არის $G = (V, E)$ ორი სიმრავლის წყვილი, სადაც V წვეროების სასრული სიმრავლეა, ხოლო E წიბოების სასრული სიმრავლე. V სიმრავლეს უწოდებენ გრაფის წვეროთა სიმრავლეს (vertex set), ხოლო E -ს წიბოთა სიმრავლეს (edge set). გრაფის ელემენტებს შესაბამისად ეწოდებათ წვერო (vertex) და წიბო (edge):

$$V = \{1, 2, \dots, n\} \quad E = \{(u_1, v_1), (u_2, v_2), \dots, (u_m, v_m)\}$$



ნახ. 7.1. გრაფი. $V = \{1, 2, 3, 4, 5\}$ $E = \{(1, 5), (2, 5), (2, 3), (2, 4), (3, 4)\}$

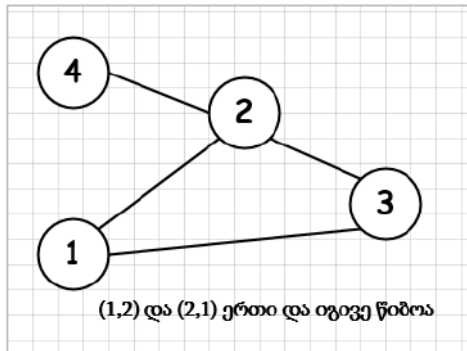
გრაფის სახეობები. გრაფი შეიძლება იყოს ორიენტირებული (მიმართული), არაორიენტირებული, შეწონილი, სრული, მკვრივი, ხალვათი, ბრტყელი და მრავალი სხვა (იხ. ნახ. 7.2.ა,ბ,გ,დ,ე,ვ). სრულ არაორიენტირებულ გრაფში წიბოების რაოდენობა გამოითვლება ფორმულით: $m = n(n - 1)/2$

მკვრივი გრაფი არის ისეთი გრაფი, რომლის წიბოების რაოდენობა წვეროების კვადრატის V^2 პროპორციულია, საწინააღმდეგო შემთხვევაში გრაფი არის ხალვათი.

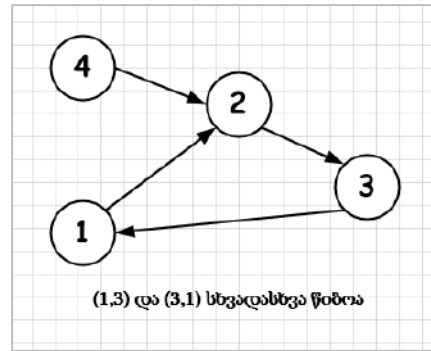
გრაფის სიმკვრივე D გამოითვლება შემდეგი ფორმულით: $D = 2m/(n(n-1))$

სრული გრაფის შემთხვევაში სიმკვრივე $D = 1$.

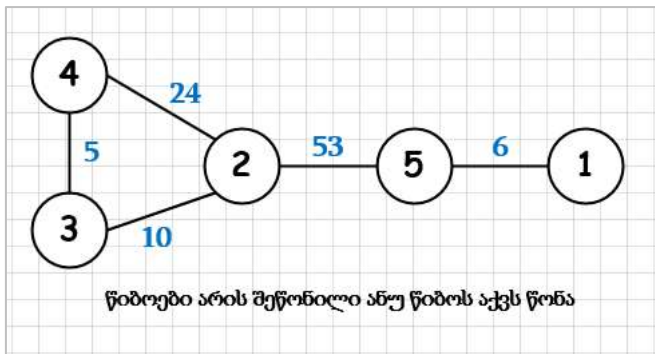
გრაფს ეწოდება ბრტყელი, თუ მისი წიბოები ერთმანეთს არ კვეთენ.



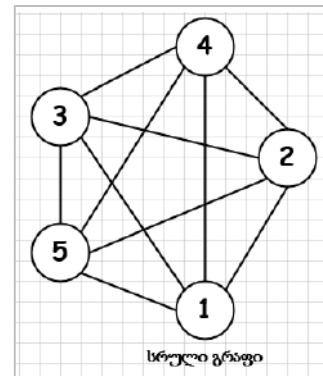
ა)



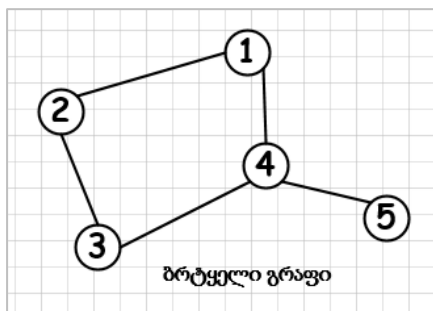
ბ)



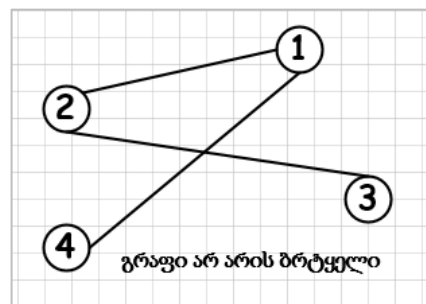
გ)



დ)



ე)



ვ)

ნახ. 7.2. გრაფის სახეები: ა) არაორიენტირებული (არამიმართული) გრაფი;

ბ) ორიენტირებული (მიმართული) გრაფი; გ) შეწონილი გრაფი $\omega_{34} = 5$, $\omega_{42} = 24$;

დ) სრული გრაფი - თითოეული წვერო შეერთებულია დანარჩენ წვეროებთან;

ე) ბრტყელი გრაფი; ვ) გრაფი არ არის ბრტყელი;

ინციდენტობა (incident). ინციდენტობის ცნება გამოიყენება მხოლოდ წიბოსთან ან წვეროსთან მიმართებაში. დავუშვათ, რომ გვაქვს წვეროები V_1 და V_2 , მათი დამაკავშირებელი წიბო არის $E = (V_1, V_2)$, ასეთი მოცემულობის დროს ვიტყვით, რომ წვერო V_1 და წიბო E ინციდენტურია, ასევე წვერო V_2 და წიბო E ინციდენტურია.

გრაფის ორი წვერო ან ორი წიბო არ შეიძლება იყოს ინციდენტური.

უახლესი წვეროების ან წიბოების აღსანიშნავად გამოიყენება მოსაზღვრეობის ცნება. ინციდენტობა შეგვიძლია მეორენაირად ასეც განვსაზღვროთ: გვაქვს წვერო a , a არის t წიბოს ან დასაწყისი. თუ გრაფს დავამატებთ წვერო b -ს, მაშინ შეგვიძლია ვთქვათ, რომ წვეროები a და b ინციდენტურია t წიბოს მიმართ.

მოსაზღვრეობა (adjacent). მოსაზღვრეობის ცნება გამოიყენება ან ორი წიბოს მიმართ ან ორი წვეროს მიმართ. ორ წიბოს, რომელებიც ინციდენტურა ერთი ცალი წვეროს მიმართ, უწოდებენ მოსაზღვრე წიბოებს. ორი წვერო, რომელებიც ინციდენტურა ერთი ცალი წიბოს მიმართ, უწოდებენ მოსაზღვრე წვეროებს.

განვიხილოთ მარყუჭი, არაორიენტირებული და ორიენტირებული გრაფი, რომელებიც ვიზუალურად მოცემულია ნახატზე 7.3.ა,ბ,გ.

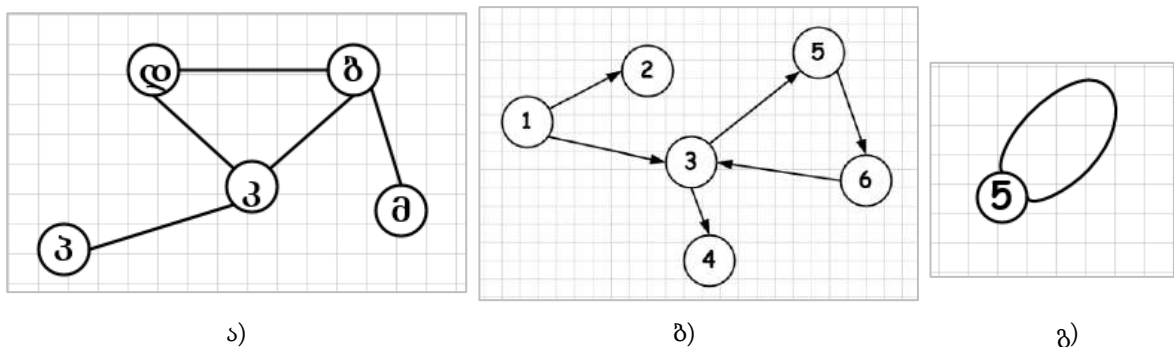
არაორიენტირებულ გრაფში (ნახ.7.3.ა.) მოცემული წვეროები: $პ$ და $კ$ არიან მოსაზღვრე წვეროები, რადგან შეერთებული არიან, წიბოთი $პკ$. ამასთანავე წვერო $კ$ არის $პკ$ წიბოს ბოლო, ხოლო $პ$ არის $პკ$ წიბოს დასაწყისი, ასეთ შემთხვევაში ვიტყვით, რომ წვეროები $პ$ და $კ$ არის ინციდენტური $პკ$ წიბოს მიმართ.

ნახატ 7.3.ა-ზე მოცემულ ორიენტირებულ გრაფში მოძრაობა (გადაადგილება) შეიძლება ორივე მიმართულებით: როგორც $დ$ წვეროდან $კ$ წვეროსკენ, ასევე $კ$ წვეროდან $დ$ წვეროსკენ.

ორიენტირებულ გრაფში კი ყველაფერი ცოტა სხვანაირად გვაქვს (ნახ.7.3.ბ). ორიენტირებულ გრაფში უკვე გვაქვს წიბოს მიმართულება, რომელსაც ვერ შევცვლით. წვერო 1 მოსაზღვრეა წვერო 2-სთვის, მაგრამ წვერო 2 ვერანაირად ვერ იქნება მოსაზღვრე წვერო 1-სთვის. წვეროები 1 და 2 ინციდენტურია წვერო 12-სთვის, მაგრამ ორიენტირებული გრაფის შემთხვევაში პირიქითი ინციდენტობას ადგილი არ აქვს.

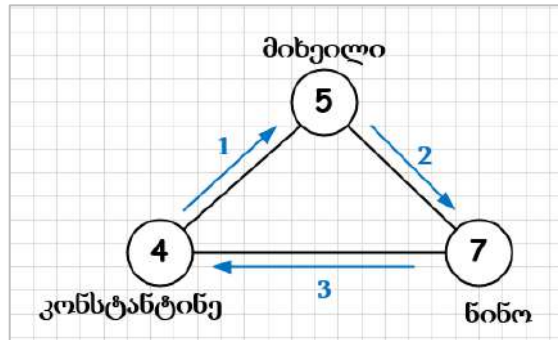
ნახატ 7.3.ბ-ზე მოცემულ გრაფში მოძრაობა (გადაადგილება) შეიძლება 1-ლი წვეროდან მე-2 წვეროსკენ, მაგრამ მე-2 წვეროდან 1-ლისკენ არ შეიძლება მოძრაობა.

წიბოს ეწოდება მარყუჭი, თუ მისი ბოლო და საწყისი ემთხვევა ერთმანეთს (ნახ. 7.3.გ.).



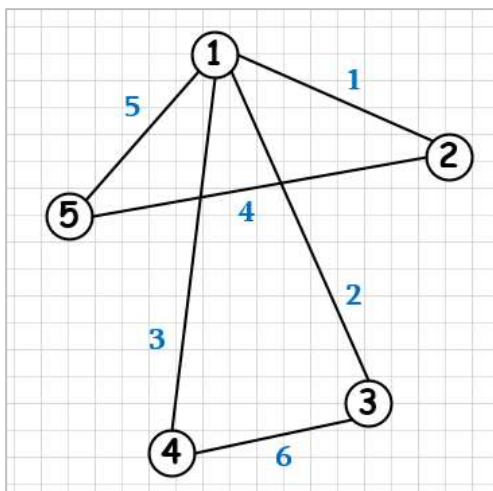
ნახ. 7.3. გრაფი: ა) არაორიენტირებული; ბ) ორიენტირებული; გ) მარყუჭი;

მაგალითი 7.1. მიხეილს აქვს 5 კანფეტი, კონსტანტინეს - 4, ხოლო ნინოს - 7. ნინომ კონსტანტინეს უწილადა 3 კანფეტი, მიხეილმა ნინოს - 2, კონსტანტინემ კი მიხეილს - 1. შესაბამისი გრაფი მოცემულია ნახ. 7.4.

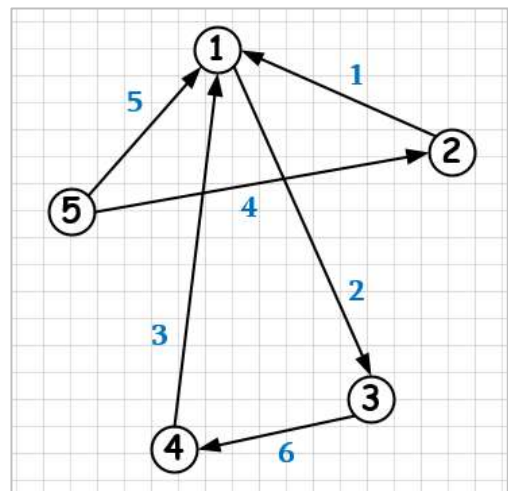


ნახ. 7.4. მაგალითი 1-ის შესაბამისი შეწონილი გრაფი

მაგალითი 7.2. ნახატზე 7.5.ა,ბ-ზე მოცემულია შეწონილი არაორიენტირებული და შეწონილი ორიენტირებული გრაფი. დავწეროთ ორივე შემთხვევისთვის ინციდენტობის მატრიცა.



ა)



ბ)

ნახ. 7.5. შეწონილი ა) არაორიენტირებული გრაფი; ბ) ორიენტირებული გრაფი

შეწონილი არაორიენტირებული გრაფის ინციდენტობის მატრიცა:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

შეწონილი ორიენტირებული გრაფის ინციდენტობის მატრიცა:

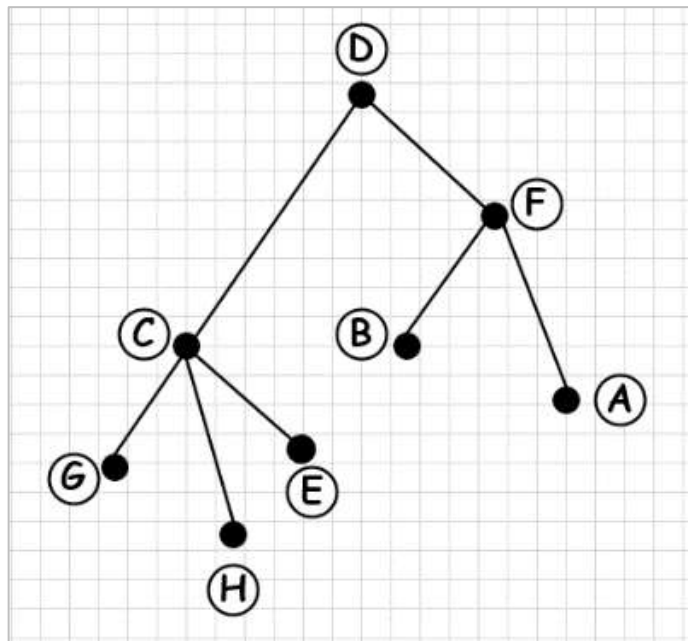
$$\begin{pmatrix} 1 & -1 & 1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 & -1 & 0 \end{pmatrix}$$

გრაფის წვეროს ხარისხი (degree). გრაფის ერთი წვეროდან გამომავალ წიბოების რაოდენობას ეწოდება ამ წვეროს ხარისხი ანუ იმ წიბოების რაოდენობა, რომელიც

ინციდენტურია, მაგალითად A წვეროსათან მიმართებით, ეწოდება A წვეროს ხარისხი და აღინიშნება $\deg(A)$ -ით. წვეროს, რომლის ხარისხი არის 1 ეწოდება დაკიდული წვერო (იხ. ნახ. 7.6.).

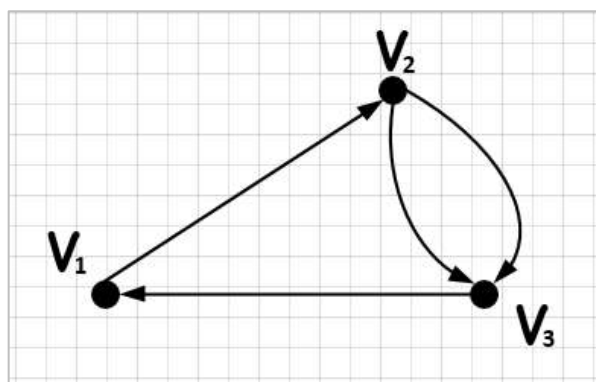
გრაფში ყველა წვეროს ხარისხების ჯამი წიბოთა გაორმაგებული რაოდენობის ტოლია.

გრაფის ყველა წვეროს ხარისხთა ჯამი ლუწი რიცხვია. ნებისმიერ გრაფში კენტი ხარისხის წვეროების რიცხვი ლუწია. ნებისმიერ გრაფში არის მინიმუმ ორი წვერო, რომელსაც აქვს ერთი და იგივე ხარისხი.



ნახ. 7.6. არაორიენტირებულ გრაფში წვეროს ხარისხი: $\deg(A) = 1, \deg(C) = 4, \deg(D) = 2$

ორიენტირებულ გრაფში კი განასხვავებენ შემავალ (in-degree, $\deg_+$) და გამომავალ (out-degree, $\deg_-$) ხარისხებს (წვეროში შემავალი და გამომავალი წიბოების რაოდენობის მიხედვით) და მათ ჯამს უწოდებენ წვეროს ხარისხს (ნახ. 7.7.).



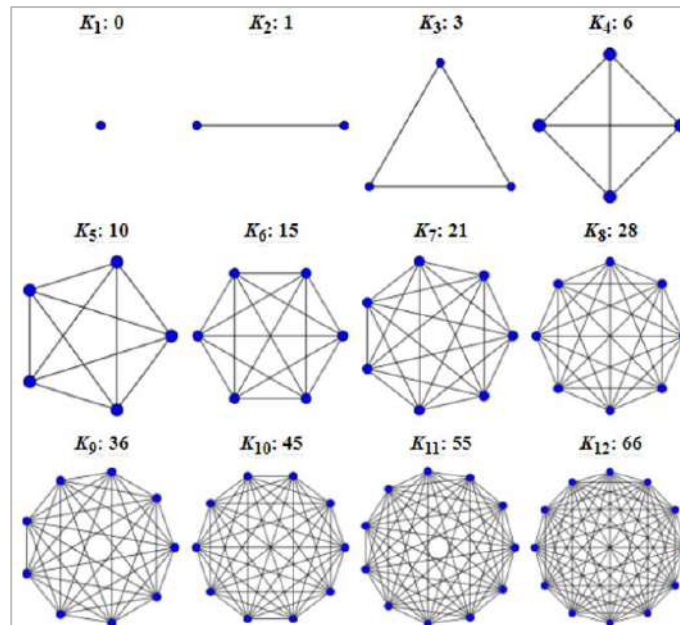
ნახ. 7.7. ორიენტირებულ გრაფში წვეროს ხარისხის განსაზღვრა:

$$\deg_+(V_1) = 1, \deg_+(V_2) = 1, \deg_+(V_3) = 2, \deg_-(V_1) = 1, \deg_-(V_2) = 2, \deg_-(V_3) = 1$$

$$\text{ჯამურად გვექნება: } \deg(V_1) = 2 \quad \deg(V_2) = 3 \quad \deg(V_3) = 3$$

მაგალიტი 7.3. ერთმანეთს შეხვდა 12 მეგობარი. თითოეულმა მეგობარმა დანარჩენ მეგობარს ჩამოართვა ხელი. რამდენი ცალი ხელის ჩამორთმევა მოხდა (იხ. ნახ. 7.8.).

ამოხსნა: მე-3 მაგალითი წარმოვიდგინოთ შემდეგი გრაფის სახით: დავუშვათ, რომ გრაფის წვეროები არის მეგობრები, ხოლო წიბოები ხელის ჩამორთმევა. ასეთი განლაგების შემთხვევაში თითოეული წვეროდან იქნება გამომავალი 11 წიბო (ყველა გარდა მიმდინარე წვეროს წიბო $12 - 1 = 11$) ანუ თითოეული წვეროს ხარისხი იქნება 11, ამდენად ყველა ხელის ჩამორთმევა იქნება $12 * 11 = 132$ ცალი, მაგრამ უნდა გავითვალისწინოთ ერთი ფაქტორი: თუ A მეგობარმა ხელი ჩამოართვა B მეგობარს, მაშინ B მეგობარმაც ხელი ჩამოართვა A მეგობარს და გამოდის, რომ ხელის ჩამორთმევა დავითვალეთ ორჯერ (როგორც A-ს მხრიდან, ისე B-ს მხრიდანაც), 132 ცალი ხელის ჩამორთმევა არის გაორმაგებული და იგი უნდა გავყოთ ორზე. ზემო თქმულიდან გამომდინარე, წიბოების რაოდენობა იქნება: $(12 * 11) / 2 = 132 / 2 = 66$.



ნახ. 7.8. სრული გრაფები. წვერობის რაოდენობა იცვლება 1-დან 12-მდე

პასუხი: მოხდა 66 ცალი ხელის ჩამორთმევა.

გზა გრაფში. გრაფში k სიგრძის გზა u წვეროდან v -მდე განისაზღვრება როგორც წვეროთა $(v_0, v_1, v_2, \dots, v_k)$ მიმდევრობა, სადაც $v_0 = u$, $v_k = v$ და $v_{i-1}, v_i \in E$ ნებისმიერი $i = 1, 2, \dots, k$. ამრიგად, k სიგრძის გზა შედგება k წიბოსაგან. იგი შეიცავს $v_0, v_1, v_2, \dots, v_k$ წვეროებს და $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ წიბოებს.

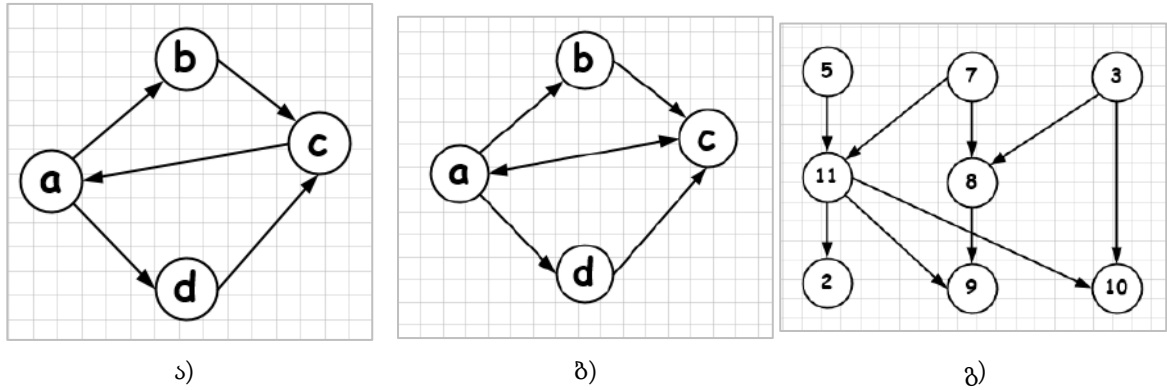
v_0 წვეროს უწოდებენ **გზის დასაწყისს**, ხოლო v_k წვეროს - **გზის ბოლოს** და ამბობენ, რომ გზა მიდის v_0 -დან v_k -საკენ. თუ $v_0 = v_k$, გზას უწოდებენ **ციკლურს** (იხ. ნახ. 7.9.ა,ბ,გ).

თუ მოცემული u და ux წვეროებისთვის არსებობს p გზა u -დან ux -ში, მაშინ ამბობენ, რომ ux მიღწევადია u -დან p გზით (ux is reachable from u via p).

გზას ეწოდება მარტივი (simple), თუკი ყველა წვერო მასში განსხვავებულია.

ორიენტირებულ გრაფში ციკლი (cycle) ეწოდება გზას, რომელშიც საწყისი წვერო ემთხვევა ბოლო წვეროს და რომელიც ერთ წიბოს მაინც შეიცავს.

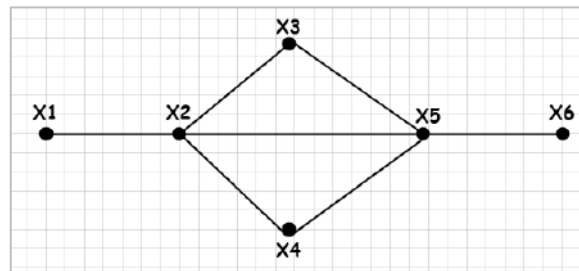
გრაფს, რომელშიც არ არის ციკლები, აციკლური (acyclic) ეწოდება.



ნახ. 7.9. გრაფში: ა) გვაქვს ციკლი a, b, c, a და c, a, d, c ;

ბ) აციკლური გრაფი; გ) აციკლური გრაფი;

მაგალითი 7.4. მოცემულია გრაფი (ნახ. 7.10.ა.). დავადგინოთ მოცემულ გრაფში გზა.



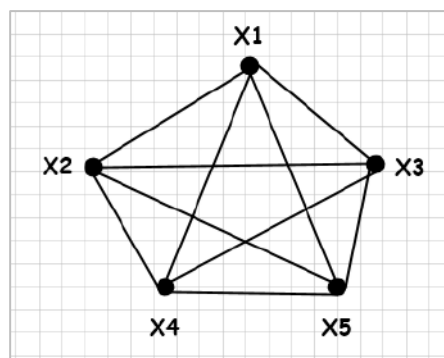
ნახ. 7.10.ა. გრაფში გზის დადგენა

ნახ. 7.10.ა.-ზე მოცემული გრაფისთვის გზის სიგრძის დადგენა:

- 1) x_1, x_2, x_5, x_6 – გზის სიგრძეა 3;
- 2) x_1, x_2, x_3, x_5, x_6 – გზის სიგრძეა 4;
- 3) x_1, x_2, x_4, x_5, x_6 – გზის სიგრძეა 4;
- 4) $x_1, x_2, x_3, x_5, x_4, x_2, x_5, x_6$ – გზის სიგრძეა 7;

პირველი სამი გზა ითვლება მარტივ გზად.

მაგალითი 7.5. მოცემულია გრაფი (ნახ.7.10.ბ.). დავადგინოთ მოცემულ გრაფში ციკლი.



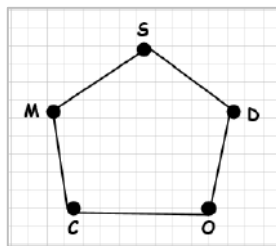
ნახ. 7.10.ბ. გრაფში ციკლის დადგენა.

ნახ. 7.10.ბ.-ზე მოცემული გრაფისთვის ციკლის დადგენა:

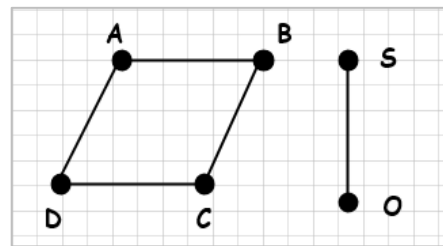
- 1) x_1, x_5, x_4, x_1 – ციკლის სიგრძეა 3;
- 2) x_1, x_2, x_3, x_1 – ციკლის სიგრძეა 3;
- 3) $x_1, x_2, x_4, x_5, x_3, x_1$ – ციკლის სიგრძეა 5;
- 4) $x_1, x_2, x_4, x_5, x_2, x_3$ – ციკლის სიგრძეა 6; და ა.შ.

პირველი სამი ციკლი ითვლება მარტივ ციკლად.

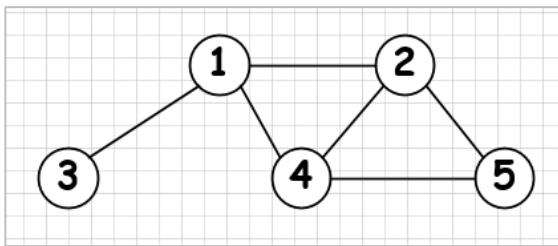
გრაფის ბმულობა. სრული გრაფი. თუ გრაფში ნებისმიერი წყვილი წვერო (ორი წვერო) შეერთებულია გზით, მაშინ მას ეწოდება ბმული. გრაფი სრულია თუ გრაფში არსებობს (გავლებულია) წიბოს ყველა შესაძლო ვარიანტი. გრაფის სრულობის პირობა: თუ გვაქვს n წვერო, მაშინ გვექნება $n(n - 1)/2$ წიბო (ნახ. 7.11.ა,ბ,გ,დ).



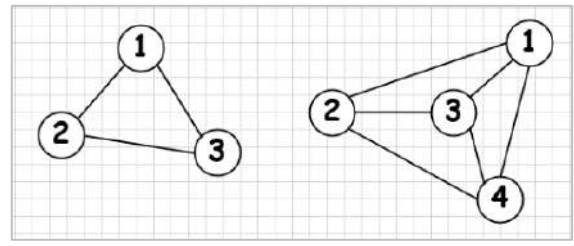
ა)



ბ)



გ)

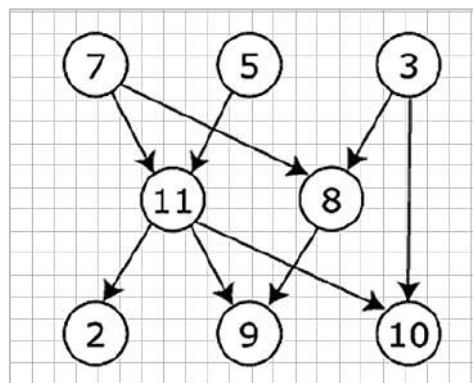


დ)

ნახ. 7.11. გრაფი: ა) ბმულია; ბ) არ არის ბმული; გ) ბმულია; დ) სრულია;

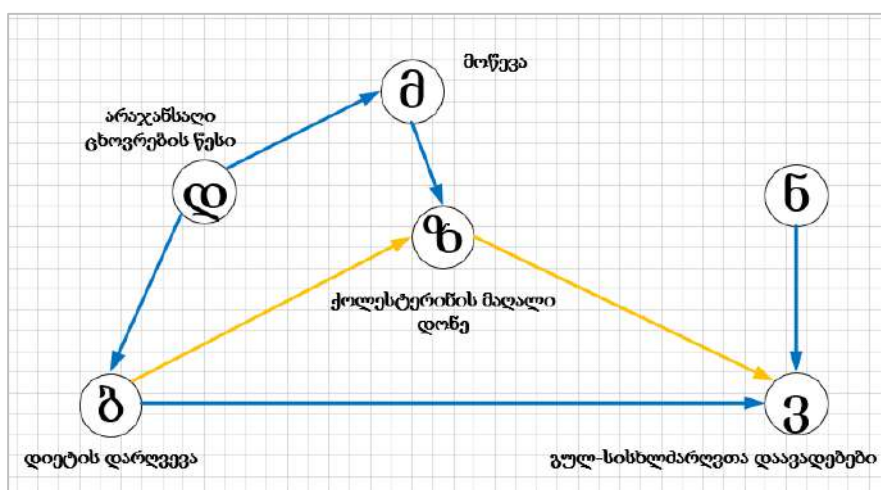
ორიენტირებული აციკლური გრაფები. ორიენტირებული აციკლური გრაფი (მიმართული აციკლური გრაფი - DAG (Directed Acyclic Graph)) არის ორგრაფი ციკლის გარეშე, მაგრამ შეიძლება იყოს „პარალელური გზები“, რომლებიც გამოდის რომელიმე ერთი წვეროდან და საბოლოო წვეროს უერთდება სხვადასხვა გზით (იხ.ნახ. 7.12.).

ორიენტირებული აციკლური გრაფი არის ხე-გრაფის განზოგადოებული სახე.



ნახ. 7.12. ორიენტირებული აციკლური გრაფი (DAG)

მაგალითი 7.6. გულ-სისხლძარღვთა დაავადებების განვითარების მიზეზების დადგენა ორიენტირებული აციკლური გრაფის საშუალებით (ნახ. 7.13.).



ნახ. 7.13. გულ-სისხლძარღვთა დაავადებების განვითარების

ორიენტირებული აციკლური გრაფი

ამოხსნა: დიეტის დარღვევა იწვევს საერთო ქოლესტერინის დონის მომატებას, რაც თავის მხრივ ზრდის გულ-სისხლძარღვთა დაავადებების განვითარების რისკს. ამრიგად, გრაფში გზა $ბ \rightarrow ზ \rightarrow ვ$ არის მიზეზ-შედეგობრივი და წარმოადგენს პირდაპირ გზას, რადგან თითოეული ელემენტი პირდაპირ კავშირშია შემდეგ ელემენტთან. ასევე ვიცით, რომ მოწევა (წვერო $მ$) ზრდის ქოლესტერინის (წვერო $ზ$) დონეს და შესაბამისად იზრდება გულ-სისხლძარღვთა დაავადებების განვითარების რისკი.

გვაქვს სხვა გზაც: გზა $მ \rightarrow ზ \rightarrow ვ$ წარმოადგენს არა პირდაპირ, არამედ ირიბ გზას (კავშირს) ცუდ კვებასა და გულ-სისხლძარღვთა დაავადებებს შორის. შეგვიძლია ვივარაუდოთ, რომ დიეტის დარღვევა ადამიანი უფრო მეტად მიჰყვება არაჯანსაღ ცხოვრების წესს, მაგალითად, ეწევა.

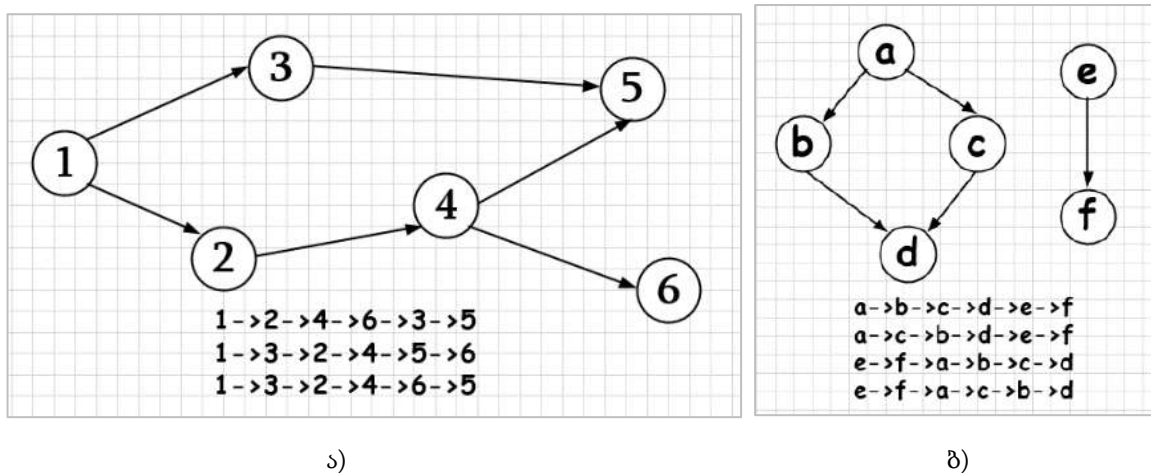
ცხოვრების არაჯანსაღი წესი გამოსახულია როგორც ფარული (გაუზომავი) წვერო $დ$. არაჯანსაღი ცხოვრების წესისადმი მიდრეკილებამ გრაფზე გამოიწვია როგორც დიეტის დარღვევები, ისე მოწევა. ვხედავთ, რომ კავშირი ცუდ კვებასა და მოწევას შორის გამოიხატება განშტოებული გზით: „მოწევა $\leftarrow$ არაჯანსაღი ცხოვრების წესი $\rightarrow$ არასწორი კვება“ და არა პირდაპირი გზით.

დასკვნა: ზემოთ მოცემული ახსნის საფუძველზე ნახატ 7.13-ზე მოცემული გრაფი არის ორიენტირებული აციკლური გრაფი.

ტოპოლოგიური დახარისხება/სორტირება (Topological sort). ტოპოლოგიური დახარისხება გულისხმობს გრაფის წვეროების ისეთი წრფივი თანმიმდევრობით განლაგებას, რომ ნებისმიერი წიბო მიმართული იყოს ამ თანმიმდევრობაში ნაკლები ნომრის მქონე წვეროდან მეტი ნომრის მქონე წვეროსაკენ. თუკი გრაფი შეიცავს ციკლს, ასეთი

თანამიმდევრობა არ იარსებებს და შესაბამისად ტოპოლოგიური დახარისხებაც არ იმუშავებს, რადგან ამ დროს წარმოიქმნება წინააღმდეგობა, რადგან აუცილებლად იარსებებს გზა ერთი წვეროდან მეორეში და პირიქით. ამდენად ტოპოლოგიური დახარისხება გამოიყენება ორიენტირებული აციკლური გრაფისთვის.

ტოპოლოგიური დახარისხება შეიძლება მეორენაირადაც ავხსნათ: განვალაგოთ გრაფის წვეროები ჰორიზონტალურ წრფეზე ისე, რომ ყველა წიბო მიმართული იყოს მარცხნიდან მარჯვნივ. სწორედ ეს არის ტოპოლოგიური დახარისხების ამოცანა. ერთი გრაფისთვის შეიძლება არსებობდეს რამდენიმე ტოპოლოგიური დახარისხების მეთოდი (მაგალითად, როდესაც ის არ არის ბმული) ან შეიძლება საერთოდ არ იყოს (თუ არის ციკლები). განვიხილოთ ტოპოლოგიური დახარისხების ნიმუშები ორიენტირებული აციკლური გრაფებისთვის - DAG (ნახ. 7.14.).



ნახ. 7.14. ტოპოლოგიური დახარისხების შესაძლო ვარიანტები

მაგალითი 7.7. მოცემული გვაქვს დავალებების სია (მიმდევრობა), რომელთა შესრულებაც დამოკიდებულია ერთიმეორეზე.

მაგალითად, მეორე დავალების შესრულება შეუძლებელია, თუ არ შესრულდა პირველი, მესამე და მეოთხე დავალებები, ასევე ვერ შევასრულებთ მესამე დავალებას, თუ არ შესრულდა პირველი და მეოთხე დავალებები. დავალებები გამოვსახოთ ორიენტირებული აციკლური გრაფის სახით (იხ. ნახ. 7.15).

პირველ რიგში, შეგვიძლია შევასრულოთ პირველი და მეოთხე დავალებები (ნებისმიერი თანამიმდევრობით), რადგან მათ შესასრულებლად არ არის საჭირო სხვა დავალებების შესრულება (არ გააჩნიათ შემავალი წიბოები).

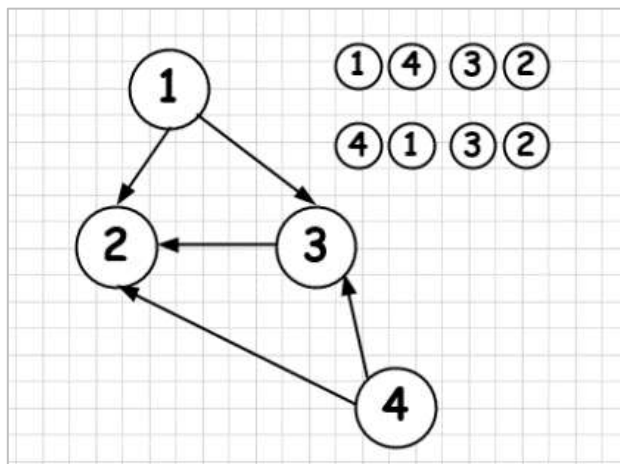
1-ლი და მე-4 დავალებების შესრულების შემდეგ, ჩვენ შეგვიძლია გადავიდეთ მე-3 დავალების შესრულებაზე, რადგან ის დამოკიდებულია 1-ლი და მე-4 დავალებების შესრულებაზე და ბოლოს, როდესაც შევასრულებთ მესამე დავალებას, შეგვიძლია შევასრულოთ მე-2 დავალებაც.

განვიხილოთ ტოპოლოგიური დახარისხების ალგორითმის მუშაობა (იხ. მაგალითი 7.7):

1) თითოეული წვეროსთვის შევინახოთ იმ წვეროების სიმრავლე, რომლებზეც დამოკიდებულია დავალების ამოხსნა. ჩვენს შემთხვევაში ესაა 1-ლი და მე-4 წვერო, რომელთაც არ აქვთ შემავალი წიბოები. მესამე წვეროსთვის არის პირველი და მეოთხე წვეროები, ხოლო მეორე წვეროსთვის არის პირველი, მესამე და მეოთხე წვეროებიდან შემავალი წიბოები. ჰეშცხრილის სახით შეიძლება ასე წარმოვიდგინოთ:

{ 1 -> empty; 4 -> empty; 3 -> {1, 4}; 2 -> {1, 3, 4}; };

2) ვიპოვოთ დამოუკიდებელი წვეროები ანუ ის წვეროები, რომელთაც არ აქვთ შემავალი წიბოები;



ნახ. 7.15. ორიენტირებული აციკლური გრაფში დავალებების სია

3) თითოეული ასეთი დამოუკიდებელი წვეროსთვის განვაახლოთ დამოკიდებულების სიმრავლე 1-ლი პუნქტის მიხედვით, დამოუკიდებელი წვეროდან გამომავალი ყველა წიბოების წაშლით.

4) ვიპოვოთ ახალი დამოუკიდებელ წვეროები და ა.შ.

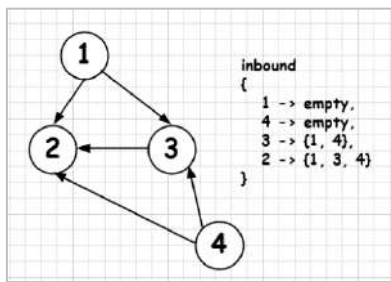
ახლა კი ვნახოთ, თუ როგორ იმუშავებს აღწერილი ტოპოლოგიური დახარისხების ალგორითმი ნაბიჯ-ნაბიჯ მაგალითი 7.7.-ის შემთხვევაში (იხ. ნახ. 7.16. ა,ბ,გ,დ,ე,ვ,ზ).

ბიჯი 1. თავდაპირველად უნდა განვახორციელოთ ჰეშ-ცხრილის (inbound) ინიციალიზება შემომავალი წიბოებისთვის.

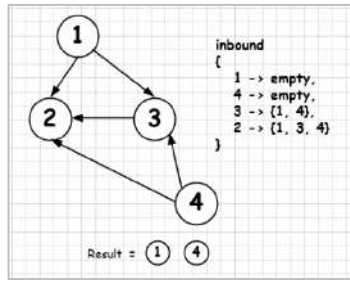
ბიჯი 2. ვიპოვოთ დამოუკიდებელი წვეროები. ესენია 1-ლი და მე-4 წვეროები, რადგან მათ არ აქვთ შემავალი წიბოები. დავამატოთ ისინი შედეგში ანუ result ცვლადში.

ბიჯი 3. წავშალოთ 1-ლი და მე-4 წვეროებიდან გამომავალი წიბოები და განვაახლოთ inbound ჰეშცხრილი.

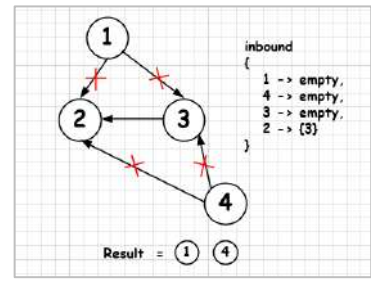
ბიჯი 4. ახლა კი ამოვშალოთ 1-ლი და მე-4 წვეროები inbound ჰეშ-ცხრილიდან, როგორც უკვე განხილული წვეროები.



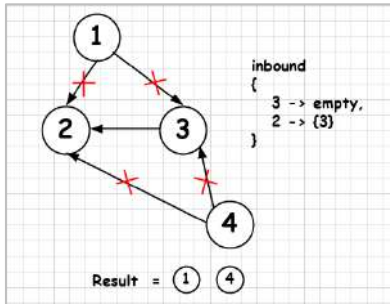
ა)



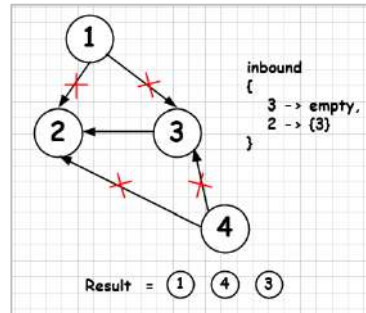
ბ)



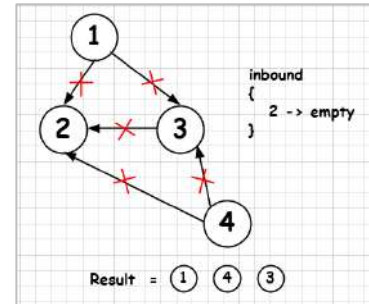
გ)



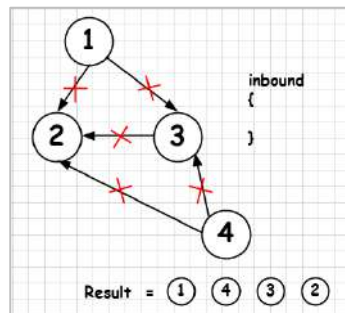
დ)



ე)



ზ)



ს)

ნახ. 7.16. მე-7.7 მაგალითზე დაყრდნობით აგებული ტოპოლოგიური დახარისხების:

ა) პირველი ბიჯი; ბ) მეორე ბიჯი; გ) მესამე ბიჯი; დ) მეოთხე ბიჯი;

ე) მეხუთე ბიჯი; ვ) მეექვსე ბიჯი; ზ) მეშვიდე ბიჯი;

ბიჯი 5. ახალი გადავიდეთ შემდეგი ახალი დამოუკიდებელი წვეროს მოძებნაზე. ეს არის მე-3 წვერო, რადგან ჰემცხრილში ახლა უკვე მას არ გააჩნია შემავალი წიბოები. დავამატოთ ის შედეგს.

ბიჯი 6. მე-3 წვეროდან გამომავალი წიბო წავშალოთ და განვაახლოთ ჰემცხრილი. ამავე დროს წავშალოთ მე-3 წვეროც, როგორც უკვე განხილული.

ბიჯი 7. გავიმეოროთ ყველაფერი თავიდან. ამჟამად დამოუკიდებელი წვერო არის მე-2 წვერო. დავამატოთ იგი შედეგს და ამოვშალოთ ჰემცხრილიდან, როგორც მე-2 წვერო, ასევე მისგან გამომავალი ყველა წიბო. მივიღებთ: inbound { }

მომდევნო იტერაციაზე უკვე აღარ არსებობს ახალი დამოუკიდებელი წვეროები და ალგორითმის შესრულებაც დასრულდა.

გრაფებთან დაკავშირებული ბევრი ალგორითმი საჭიროებს გრაფის წვეროებისა და წიბოების გარკვეულ ძებნა/დამუშავებას მათზე შემოვლის გზით.

გრაფების შემოვლისთვის არსებობს ორი ძირითადი ალგორითმი - **სიგანეში ძებნის (Breadth-First Search, BFS)** და **სიღრმეში ძებნის (Depth-First Search, DFS)** ალგორითმები. ეს ალგორითმები გამოიყენება როგორც არაორიენტირებული, ისე ორიენტირებული გრაფებისთვის. გარდა მათი ძირითადი დანიშნულებისა (წვეროებისა და წიბოების ნახვა/შემოვლა), მათი გამოყენება შესაძლებელია გრაფების რიგი მნიშვნელოვანი თვისების დასადგენად, ისეთის, როგორიცაა, გრაფის ბმულობა, აციკლურობა და ა. შ. შემოვლა/ნახვა ნიშნავს გრაფის ყველა წვეროს ნახვას. სიგანეში ძებნის ალგორითმი (BFS) არის რეკურსიული ალგორითმი გრაფის ან ხე-გრაფის მონაცემთა სტრუქტურის ყველა წვერის მოსაძიებლად.

სიგანეში ძებნის ალგორითმი (BFS). სიგანეში ძებნის შედეგი იკვლევს უმოკლესი სიგრძის გზას აუწონავ გრაფში ე. ი. გზას, რომელიც უმცირესს წიბოთა რიცხვს შეიცავს. ალგორითმი მუშაობს $O(n + m)$, სადაც n - გრაფის წვეროების რიცხვია, m - წიბოთა რიცხვი.

ალგორითმის აღწერა. ალგორითმს მიეწოდება გრაფი და სასტარტო წვეროს ნომერი, გრაფი შეიძლება იყოს ორიენტირებული ან არაორიენტირებული, ალგორითმისთვის ამას მნიშვნელობა არა აქვს.

თვითონ ალგორითმი შეიძლება აღვიქვათ, როგორც გრაფში „ტალღის“ გავრცელება, ნულოვან ბიჯზე ტალღა იწყება წვეროდან. ყოველ მომდევნო ბიჯზე „ტალღის“ კიდეებში არსებული წვეროებიდან მათ მეზობელ წვეროებზე გადადის ანუ ალგორითმის ერთი იტერაციით „ტალღის“ სიგანეში ერთეულზე გაფართოვება ხდება (აქედან გამომდინარეობს ალგორითმის დასახელება).

სიგანეში ძებნის ალგორითმის რეალიზაცია. სიგანეში ძებნის სტანდარტული ალგორითმი გრაფის თითოეულ წვეროსათვის განიხილავს ორ კატეგორიას:

გრაფის დაფარული/ნანახი წვერო (Visited იგივე მონიშნული წვერო);

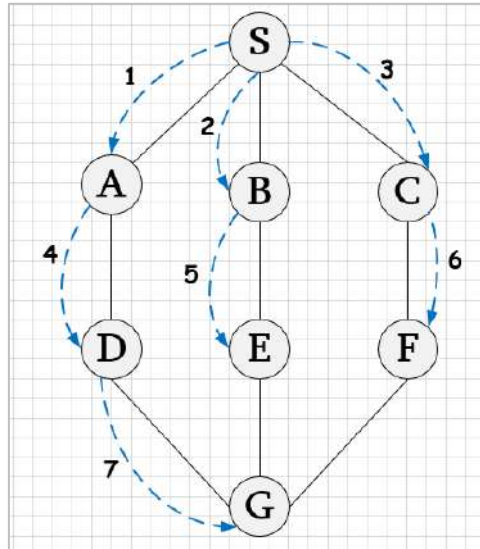
გრაფის დაუფარავი/უნახავი წვერო (Not Visited იგივე წვერო მონიშვნის გარეშე);

სიგანეში ძებნის ალგორითმის მიზანი. მოვნიშნოთ თითოეული წვერო, როგორც ნანახი, ციკლების აცილებით.

BFS ალგორითმი პოულობს უმოკლეს გზებს მოცემული საწყისი წვეროდან ყველა სხვა წვერომდე (გზის სიგრძე ითვლება მასში არსებული წიბოების რაოდენობით).

სიგანეში ძებნის ალგორითმი უზრუნველყოფს, გრაფის სიგანეზე განთავსებული ყველა წვეროს, დაწყებული საწყისი წვეროთი, მონახულებას (იხ.ნახ. 7.17.). როგორ მუშაობს BFS ალგორითმი? საწყისი წვეროდან დაწყებული, ჯერ ხდება ერთი დონის ყველა წვეროს მონახულება და მხოლოდ ამის შემდეგ ხდება გადასვლა მომდევნო დონის წვეროების მოსანახულებლად, სანამ ყველა წვერო არ აღმოჩნდება ნანახი. ალგორითმის მუშაობაში

გვეხმარება მონაცემთა სტრუქტურა რიგი. მიმდინარე დონის ყველა მოსაზღვრე (მეზობელი) მოუნახულებელი წვერო მოთავსდება რიგში, ხოლო მიმდინარე დონის წვეროები მოთავსდება როგორც ნანახი/მოწინააღმდეგე და მოხდება მათი რიგიდან ამოღება (ნახ. 7.17.).



ნახ. 7.17. სიგანეში ძებნა (BFS)

ნახატ 7.17-ზე ნაჩვენებია BFS ალგორითმის მუშაობის ახსნა:

1. წვერო S. წვერო S-ის მეზობელი წვეროებია A, B და C წვეროები. რიგში გვაქვს A, B, C
2. S წვეროს არ ჰყავს მეტი მეზობელი წვეროები, დავუბრუნდეთ A წვეროს. A წვეროს მეზობელია D წვერო. რიგში გვაქვს B, C, D
3. გადავიდეთ B წვეროზე. B წვეროს მეზობელია E წვერო. ახლა რიგში გვაქვს C, D, E
4. გადავიდეთ C წვეროზე. C წვეროს მეზობელია F წვერო. ახლა რიგში გვაქვს D, E, F
5. გადავიდეთ D წვეროზე. D წვეროს მეზობელია G წვერო. რიგში გვაქვს E, F, G
6. მოვინახულოთ E და F წვეროები. ნახატ 7.17-დან ჩანს, რომ მათ არ ჰყავთ მეზობელი წვეროები.
7. ბოლო წვეროა G წვერო.

შედეგი. ნახატ 7.17-ზე ნაჩვენები გრაფის შემოვლამ სიგანეში ძებნის ალგორითმის მიხედვით (BFS) მოგვცა შემდეგი შედეგი: **S A B C D E F G**

ჩამოვყალიბოთ სიგანეში ძებნის ალგორითმი:

1. **ინიციალიზაცია.** საწყისი წვერო განვითავსოთ რიგში და მოწინააღმდეგე ის როგორც დაფარული (ნანახი).

2. **კვლევა.** სანამ რიგი ცარიელი არ არის:

ამოიღეთ რიგიდან წვერო და ჩავთვალოთ დაფარულად (მაგალითად, გამოიტანეთ მისი მნიშვნელობა).

რიგიდან ამოღებული წვეროს თითოეული დაუფარავი მეზობლი წვეროსთვის:

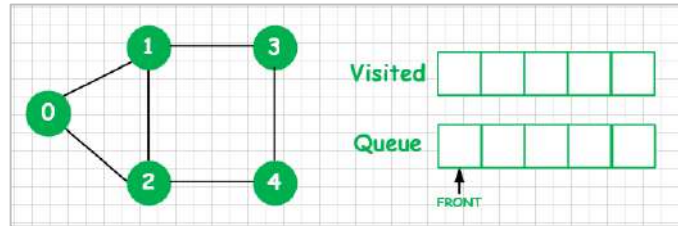
გადაიტანეთ რიგში მეზობელი წვერო;

მონიშნეთ მეზობელი წვერო, როგორც ნანახი;

3. დასასრული. გაიმეორეთ მეორე ბიჯი მანამ, სანამ რიგი არ დაცარიელდება.

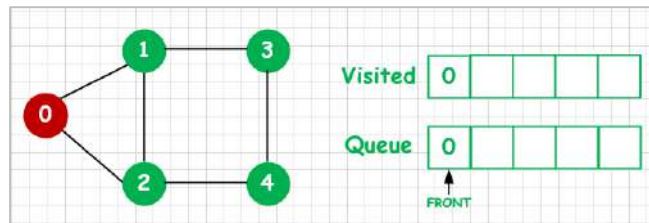
მაგალითი 7.8. სიგანეში ძებნის ალგორითმის მუშაობის განხილვა. გამოვიყენოთ არაორიენტირებული გრაფი 5 წვეროთი (იხ. ნახ. 7.18.ა,ბ,გ,დ,ე,ვ,ზ).

ბიჯი 1. თავდაპირველად რიგი და მასივი ცარიელია (ნახ. 7.18.ა.).



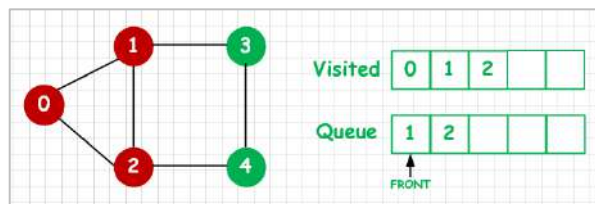
ნახ. 7.18.ა. რიგის და მასივის (მონახულების) თავდაპირველი მდგომარეობა

ბიჯი 2. საწყისი ნულოვანი წვერო გადავიტანოთ რიგის თავში და მონიშნოთ როგორც ნანახი (ნახ. 7.18.ბ.).



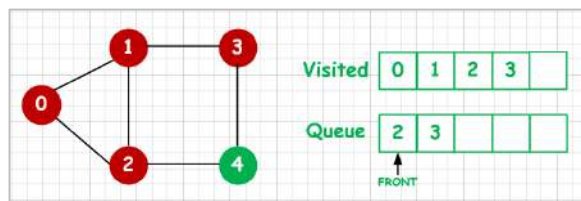
ნახ. 7.18.ბ. ნულოვანი წვეროს გადატანა რიგში და მონიშვნა (ჩავთვალოთ ნანახად)

ბიჯი 3. ამოვიღოთ ნულოვანი წვერო რიგის თავიდან და ვეწვიოთ მოუწახულებელ მეზობელ წვეროებს და გადავიტანოთ ისინი რიგში (ნახ. 7.18.გ.).



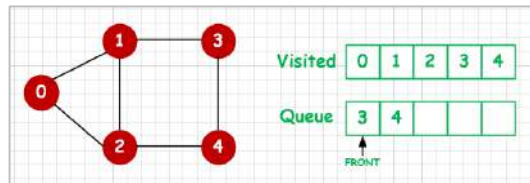
ნახ. 7.18.გ. რიგიდან ნულოვანი წვეროს ამოღება და მეზობელი წვეროების ნახვა/მონიშვნა

ბიჯი 4. ამოვიღოთ პირველი წვერო რიგის თავიდან, ვეწვიოთ დაუთვალეირებელ მეზობლებს რიგში (ნახ. 7.18.დ.).



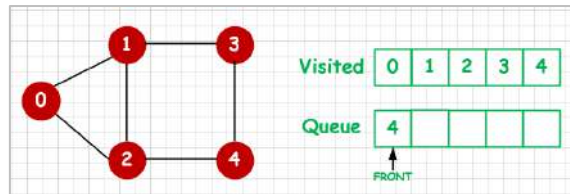
ნახ. 7.18.დ. პირველი წვეროს ამოღება და მოუწახულებელ მეზობელ წვეროსთან გადასვლა

ბიჯი 5. ამოვიღოთ მეორე წვერო რიგის თავიდან და მოუწახულებელ მეზობლი წვეროს გადატანა რიგში (ნახ. 7.18.ე.).



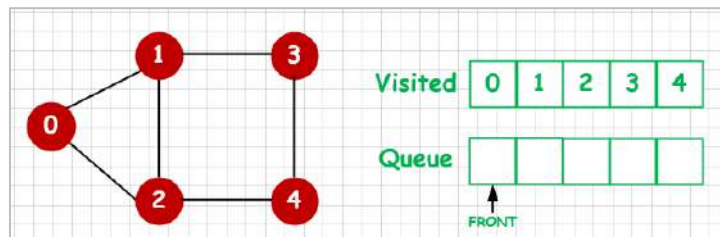
ნახ. 7.18.ე. მეორე წვეროს ამოღება და მოუნახულებელ მეზობელი წვეროს გადატანა რიგში

ბიჯი 6. რიგის თავიდან ამოვიღოთ მესამე წვერო, ვნახოთ მოუნახულებელი მეზობელი წვერო და განვათავსოთ რიგის თავში. როგორც ვხედავთ, მესამე წვეროს ყველა მეზობელი ნანახია, ამიტომ გადავიდეთ იმ წვეროზე, რომელიც არის რიგის თავში (ნახ. 7.18.ვ.).



ნახ. 7.18.ვ. მესამე წვეროს ამოღება და მოუნახულებელ მეზობელი წვეროს გადატანა რიგში

ბიჯი 7. რიგის თავიდან ამოვიღოთ მეოთხე წვეროს, ვეწვიოთ მოუნახულებელ მეზობლებს. როგორც ვხედავთ, მეოთხე წვეროს ყველა მეზობელი მონახულებულია, ამიტომ გადავიდეთ შემდეგ წვეროზე, რომელიც არის რიგის თავში (ნახ. 7.18.ზ.).



ნახ. 7.18.ზ. მეოთხე წვეროს ამოღება. მეოთხე წვეროს ყველა მეზობელი ნანახია

როგორც ვხედავთ, რიგი დაცარიელდა, ასე რომ, შეგვიძლია შევწყვიტოთ გამეორების (იტერაციების) პროცესი. ვინაიდან რიგი დაცარიელდა, ჩვენ დავასრულეთ სიგანეში შემოვლა.

სიღრმეში ძებნის ალგორითმი (DFS). გრაფში სიღრმეში ძებნა შეიძლება ასე ჩამოვყალიბოთ: ვიაროთ გრაფში წინ (სიღრმეში), სანამ არსებობს მოუნახულებელი წიბო. თუ გაუვლელი წიბო აღარ არსებობს, დავბრუნდეთ უკან და ვეძებოთ სხვა გზა. ასე განვაგრძოთ მანამ, სანამ არ ამოიწურება ყველა წვერო, რომელიც მიღწევადია საწყისი წვეროდან. თუკი გაუვლელი წვერო მაინც დარჩა, ავიღოთ ერთ-ერთი და გავიმეოროთ პროცესი, სანამ არ იქნება შემოვლილი გრაფის ყველა წვერო.

სიღრმეში ძებნა ანუ გრაფის „სიღრმეებში“ შემოვლა არის რეკურსიული ალგორითმი, რომელიც ეძებს გრაფის ან ხე-გრაფის ყველა წვეროს. სიღრმეში ძებნა გულისხმობს გრაფის ყველა წვეროს შემოვლა/მონახულებას. DFS ალგორითმი იყენებს მონაცემთა სტრუქტურას სტეკს.

სიღრმეში ძებნის ალგორითმის რეალიზაცია. სიგანეში ძებნის სტანდარტული ალგორითმი გრაფის თითოეულ წვეროსათვის განიხილავს ორ კატეგორიას:

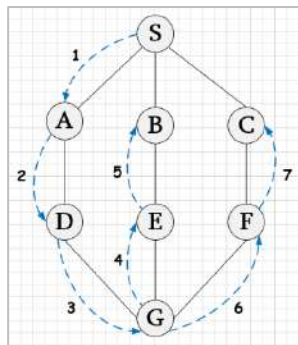
- გრაფის დაფარული/ნანახი წვერო (Visited იგივე მონიშნული წვერო);
- გრაფის დაუფარავი/უნახავი წვერო (Not Visited იგივე წვერო მონიშვნის გარეშე);

სიღრმეში ძებნის ალგორითმის მიზანია მოვნიშნოთ თითოეული წვერო, როგორც ნანახი, ციკლების აცილებით.

სიღრმეში ძებნის ალგორითმის მუშაობა. სიღრმეში ძებნის ალგორითმი მუშაობს შემდეგი სახით (იხ. ნახ. 7.19.):

1. თავიდან გრაფის რომელიმე წვერო გადავიტანოთ სტეკის თავში;
2. ავიღოთ სტეკის ზედა ელემენტი და დავამატოთ იგი მონიშნულთა სიაში;
3. შევქმნათ ამ წვეროს მოსაზღვრე/მეზობელი წვეროების სია. გრაფის ის წვეროები, რომლებიც არ არის მონახულებულთა/შემოვლითა სიაში, დაამატეთ სტეკის თავში;
4. გავიმეოროთ მეორე და მესამე ბიჯები, მანამდე სანამ სტეკი არ დაცარიელდება;

ნახატ 7.19-ზე ნაჩვენებია BFS ალგორითმის მუშაობის ახსნა: სიღრმეში ძებნის ალგორითმი თავიდან გაივლის წვერო S-დან A წვეროსკენ, შემდეგ გაივლის D წვეროსკენ, შემდეგ G წვეროსკენ, შემდეგ E წვეროსკენ, შემდეგ B წვეროსკენ, შემდეგ F წვეროსკენ და ბოლოს C წვეროსკენ.

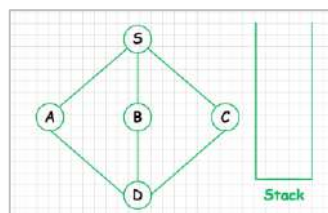


ნახ. 7.19. სიღრმეში ძებნა (DFS)

შედეგი. ნახატ 7.19-ზე ნაჩვენები გრაფის შემოვლამ სიღრმეში ძებნის ალგორითმის მიხედვით (BFS) მოგვცა შემდეგი შედეგი: **S A D G E B F C**

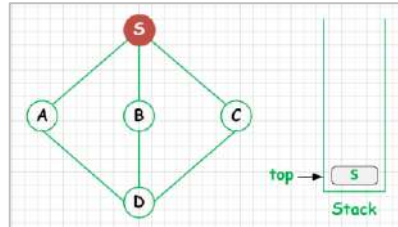
მაგალითი 7.10. სიღრმეში ძებნის ალგორითმის მუშაობის განხილვა. გამოვიყენოთ არაორიენტირებული გრაფი 5 წვეროთი (იხ. ნახ. 7.20. ა, ბ, გ, დ, ე, ვ, ზ, თ).

ბიჯი 1. სტეკის ინიციალიზება (ნახ. 7.20.ა.).



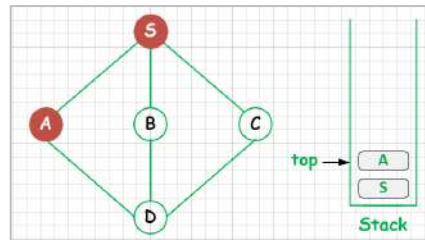
ნახ. 7.20.ა. გრაფი ხუთი წვერით და სტეკის ინიციალიზება

ბიჯი 2. მოვნიშნოთ წვერო S როგორც ნანახი და გადავიტანოთ სტეკში. ახლა გადავიდეთ S წვეროს ნებისმიერ მოუნახულებელ მეზობელ წვეროზე. გვაქვს სამი წვერო და შეგვიძლია ავირჩიოთ ნებისმიერი მათგანი. მოდიოთ ავირჩიოთ ალფავიტის რიგითობის მიხედვით (ნახ. 7.20.ბ.).



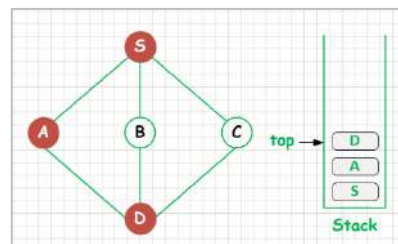
ნახ. 7.20.ბ. წვერო S -ის მონიშვნა/ნახვა

ბიჯი 3. მოვნიშნოთ წვერო A როგორც ნანახი და გადავსვათ სტეკში. დავათვალიეროთ წვერო A -ს მეზობელი წვეროები: ესეთებია D და S წვეროები, მაგრამ ჩვენ ახლა გვაინტერესებს მოუნახულებელი წვერო, D წვერო (ნახ. 7.20.გ.).



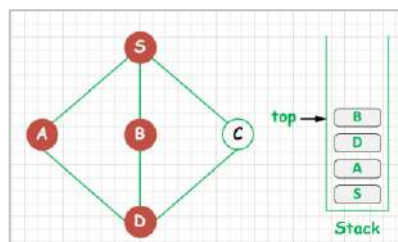
ნახ. 7.20.გ. A წვეროს მონიშვნა/ნახვა

ბიჯი 4. მოვნიშნოთ წვერო D როგორც ნანახი და გადავსვათ სტეკში. D წვეროს მეზობელი წვეროებია B და C წვეროები და ორივე მათგანი მოუნახულებელია, თუმცა განხილვას განვაგრძობთ ანბანური თანამიმდევრობით (ნახ. 7.20.დ.).



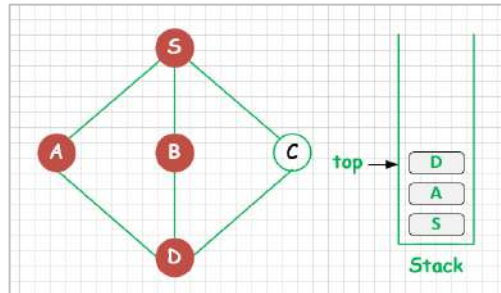
ნახ. 7.20.დ. D წვეროს მონიშვნა/ნახვა

ბიჯი 5. მოვნიშნოთ წვერო B როგორც ნანახი და გადავსვათ სტეკში. როგორც ვხედავთ, B წვეროს მეზობელი წვეროები მონიშნული/ნანახია ანუ მას არ გააჩნია მოუნახულებელი წვეროები და ამიტომ წვერო B ამოვიღოთ/ამოვაგდოთ სტეკიდან (ნახ. 7.20.ე.).



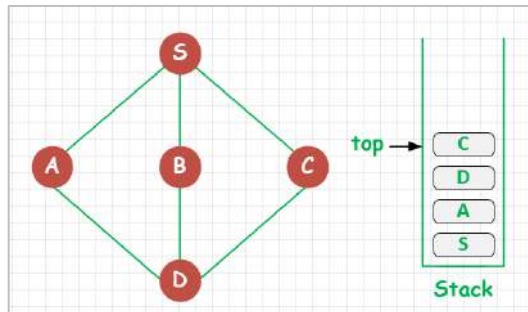
ნახ. 7.20.ე. B წვეროს მონიშვნა/ნახვა. მას არ გააჩნია მოუნახულებელი წვეროები

ბიჯი 6. შევამოწმოთ სტეკის თავი და დავადგინოთ სტეკის თავში არსებულ ელემენტს აქვს თუ არა მოუნახულებელი წვეროები. ვხედავთ, რომ სტეკის თავზე იმყოფება წვერო D (ნახ. 7.20.ვ.).



ნახ. 7.20.ვ. სტეკის დათვალიერება

ბიჯი 7. გრაფში D წვეროს დაუთვალიერებელი მეზობელი წვერო არის C. ასე რომ, მოვნიშნოთ C წვერო, როგორც ნანახი და გადავიტანოთ სტეკში (ნახ. 7.20.ზ.).



ნახ. 7.20.ზ. C წვეროს მონიშვნა/ნახვა

იმის გამო, რომ C წვეროს არ გააჩნია მოუნახულებელი წვეროები, შეგვიძლია იგი ამოვიღოთ სტეკიდან.

ასევე ვაგრძელებთ დათვალიერებას, რათა დავადგინოთ სტეკში მოთავსებული სხვა წვერს ხომ არ ჰყავს მოუნახულებელი წვერო. ასეთი წვეროები აღარ გვაქვს და ამიტომ თანამიმდევრობით ვიღებთ სტეკიდან წვეროებს, მანამ სანამ სტეკი არ დაიცლება (დაცარიელდება).

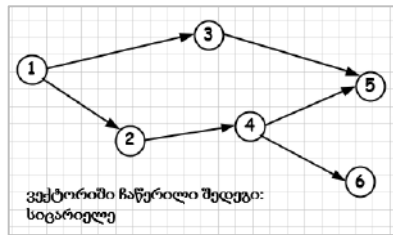
მაგალითი 7.11. ტოპოლოგიური დახარისხების კიდევ ერთი ვიზუალური მაგალითი. მოცემული გვაქვს ორიენტირებული აციკლური გრაფი ექვსი წვეროთი (იხ. ნახ. 7.21.).

თავდაპირველად მოვინახულებთ გრაფის ყველა წვეროს. თუ წვერო თეთრია, მაშინ ამ წვეროდან დავიწყებთ სიგანეში ძებნის ალგორითმის ამუშავებას.

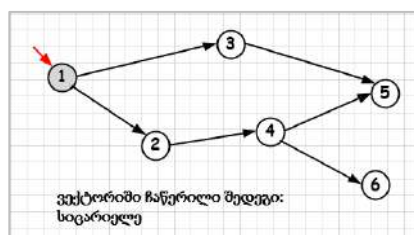
ძებნის განმავლობაში, რომელიმე წვეროდან გამოსვლისას, წვეროს ნომერს დავამატებთ იმ ვექტორში, სადაც ვინახავთ მონაცემებს.

მას შემდეგ, რაც ყველა წვეროს შემოვირბენთ/მოვინახულებთ, ვექტორში გვექნება ტოპოლოგიური დახარისხების შედეგი, მხოლოდ უკუ მიმდევრობით.

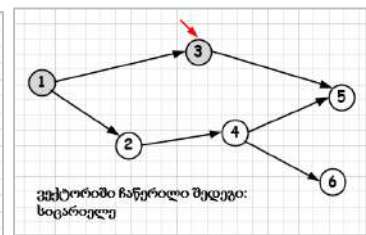
უკუ მიმდევრობით დალაგებულ მონაცემებს შემოვაბრუნებთ და მივიღებთ ტოპოლოგიური დახარისხების შედეგს.



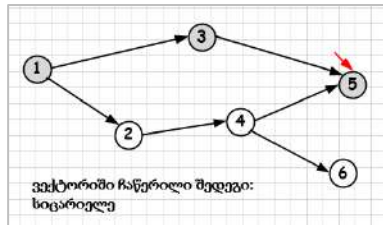
ა)



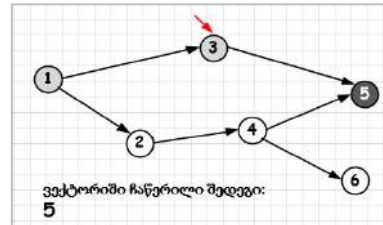
ბ)



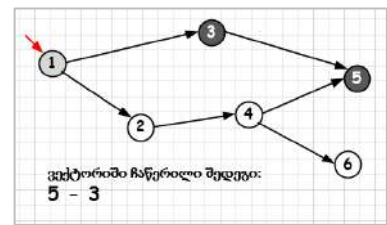
გ)



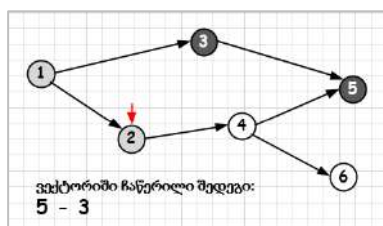
დ)



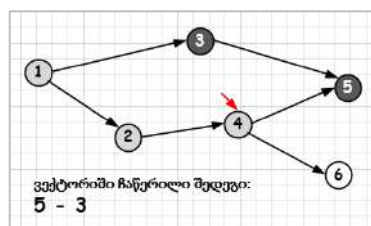
ე)



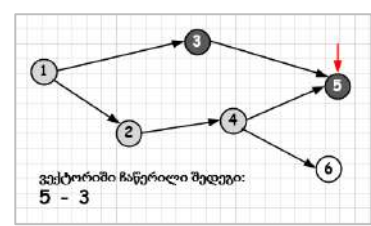
ვ)



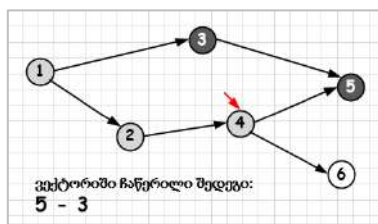
ზ)



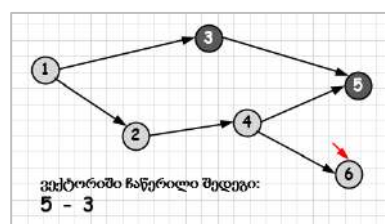
თ)



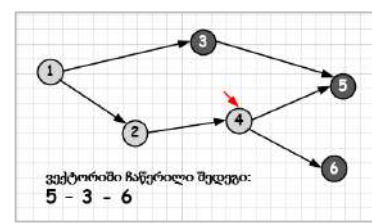
ი)



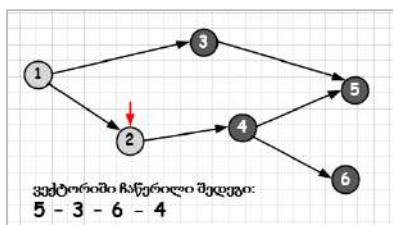
კ)



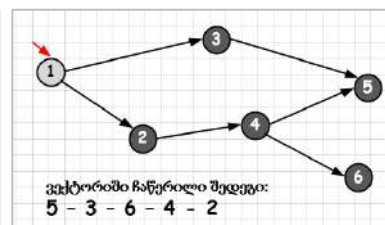
ლ)



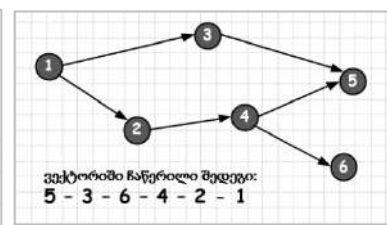
მ)



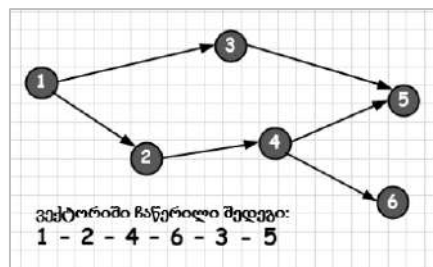
ნ)



ო)



პ)



რ)

ნახ. 7.21. ა,ბ,გ,დ,ე,ვ,ზ,თ,ი,კ,ლ,მ,ნ,ო,პ,რ: ტოპოლოგიური დახარისხება.

იყენებს სივარდი მუხის ალგორითმს: ა) 1-ლი ბიჯი; ბ) მე-2 ბიჯი; გ) მე-3 ბიჯი; დ) მე-4 ბიჯი; ე) მე-5 ბიჯი; ვ) მე-6 ბიჯი; ზ) მე-7 ბიჯით; თ) მე-8 ბიჯი; ი) მე-9 ბიჯი; კ) მე-10 ბიჯი; ლ) მე-11 ბიჯი; მ) მე-12 ბიჯი; ნ) მე-13 ბიჯი; ო) მე-14 ბიჯი; პ) მე-16 ბიჯი; რ) მე-17 ბიჯი. შედეგის შემოზრუნება

PERT დიაგრამა. PERT (Program Evaluation and Review Technique) დიაგრამა არის ინსტრუმენტი, რომელიც გამოიყენება პროექტის დაგეგმვასა და მართვაში. PERT დიაგრამა აჩვენებს პროექტის სხვადასხვა დავალებას, მათ შორის დამოკიდებულებებსა და ხანგრძლივობას.

PERT დიაგრამის ძირითადი ელემენტებია:

მოვლენები (Events) - ძირითადი მოვლენები ან ნიშნები, რომლებიც აღნიშნავენ პროექტის ძირითადი ეტაპების დასრულებას.

აქტივობები (Activities) - მოქმედებები ან დავალებები, რომლებიც საჭიროა მოვლენების შესასრულებლად.

დროის შეფასებები (Time Estimates) - დროის შეფასებები იყენებენ სამ სხვადასხვა ვარიანტს - ოპტიმისტური, პესიმისტური და ყველაზე მოსალოდნელი დრო.

PERT დიაგრამის ძირითადი მახასიათებლებია:

საქმიანობათა განსაზღვრა - პირველ ეტაპზე განისაზღვრება ყველა საჭირო საქმიანობა, რომელიც პროექტის ფარგლებში უნდა განხორციელდეს.

საქმიანობათა თანამიმდევრობა - ხდება ამ საქმიანობათა თანამიმდევრობის დადგენა, ანუ განსაზღვრება, რომელმა საქმიანობამ რომელზე ადრე უნდა დასრულდეს.

დროის შეფასება - თითოეული საქმიანობის შესასრულებლად საჭირო დროის შეფასება.

PERT დიაგრამაში გამოიყენება სამი დროის შეფასება:

ოპტიმისტური დრო (O) - დრო, რომელიც მინიმალურია საქმიანობის შესრულებისთვის.

სავარაუდო დრო (M) - ყველაზე რეალური დრო საქმიანობისთვის.

პესიმისტური დრო (P) - მაქსიმალური დრო, რომელიც შეიძლება საქმიანობის შესრულებას დასჭირდეს.

PERT დიაგრამა საშუალებას იძლევა, უკეთ დაგეგმოთ პროექტის განხორციელება და მართოთ დროითი რესურსები ეფექტურად.

ერთ-ერთი მთავარი ცნება PERT დიაგრამაში არის კრიტიკული გზა. კრიტიკული გზა არის იმ საქმიანობათა თანამიმდევრობა, რომლებიც განსაზღვრავენ პროექტის დასრულების მინიმალურ დროს ანუ კრიტიკულ გზაზე მოთავსებული თითოეული საქმიანობის დაგვიანება გამოიწვევს მთელი პროექტის დასრულების დაგვიანებას.

კრიტიკული გზა წარმოადგენს პროექტის მართვისთვის სასიცოცხლოდ მნიშვნელოვან კონცეფციას, რომელიც აერთიანებს ყველაზე გრძელ და მნიშვნელოვან საქმიანობათა ჯაჭვს, რის გარეშეც შეუძლებელია პროექტის დროული დასრულება. ეს არის

პროექტის გეგმის ის ნაწილი, რომელზეც ნებისმიერი ცვლილება ან შეფერხება პირდაპირ მოქმედებს საერთო დროის გეგმის შესრულებაზე.

კრიტიკული გზა არამარტო განსაზღვრავს პროექტის დასრულების მინიმალურ დროს, არამედ უზრუნველყოფს ხელმძღვანელობას საშუალებით გამოავლინოს და ეფექტურად მართოს დროის რესურსები.

PERT დიაგრამაში კრიტიკული გზის დადგენა:

1. ყველა დავალების და მათი ხანგრძლივობის განსაზღვრა;
2. ერთმანეთთან დამოკიდებული დავალებების განსაზღვრა;
3. დიაგრამის შექმნა, სადაც თითოეული დავალება წარმოდგენილია წვეროსა და წიბოს სახით (დამოკიდებულება);
4. დაწყების და დასრულების დროების განსაზღვრა (თითოეულ დავალებაზე უნდა ვიანგარიშოთ ადრე დაწყების (ES) და ადრე დასრულების (EF) დრო, ასევე დაგვიანებული დაწყების (LS) და დაგვიანებული დასრულების (LF) დრო);
5. ტოტალური ხანგრძლივობის გაანგარიშება. უნდა ვიპოვოთ ისეთი გზა(ები), რომლის საერთო ხანგრძლივობა ყველაზე დიდია. ეს არის კრიტიკული გზა.

დროის ზღვარის განსაზღვრა სხვა დავალებებისთვის ანუ დრო, რომლის მანძილზე შეიძლება ის დავალებები გადაიწიოს პროექტის დასრულების დროის დაუგვიანებლად. კრიტიკულ გზაზე მყოფ დავალებებს დროის ზღვარი არ აქვთ.

კრიტიკული გზა მნიშვნელოვანია, რადგან ის აჩვენებს პროექტის მინიმალურ ხანგრძლივობას და მენეჯერს ეხმარება გაიგოს, რომელ დავალებებს უნდა მიენიჭოს პრიორიტეტი, რომ პროექტი დროულად დასრულდეს.

PERT დიაგრამაში კრიტიკული გზა. პროექტის მართვაში, კრიტიკული გზის დადგენა არც თუ ისე რთულია. თუ წარმოვიდგენთ პროექტს როგორც ურთიერთდაკავშირებული ამოცანების ნაკრებს, მაშინ კრიტიკული გზა იქნება ამოცანების ყველაზე გრძელი ჯაჭვი, რომელიც თანმიმდევრულად უნდა შესრულდეს პროექტის დასაწყისიდან დასასრულამდე. შესაბამისად, თუ რომელიმე სხვა ამოცანის (ამოცანა, რომელიც არ ეკუთვნის ამ ჯაჭვს) შესრულების დრო გაიზრდება, ეს არ იმოქმედებს პროექტის შესრულების საერთო დროზე, მაგრამ თუ კრიტიკულ გზაზე რაიმე დავალების დრო გაიზრდება მინიმუმ ერთი დღით, მაშინ მთელი პროექტის ვადა შესაბამისად გადაიწევა. სწორედ ასეთ ჯაჭვს კრიტიკულ გზას უწოდებენ.

როგორც ხედავთ, ყველაფერი მარტივია - პროექტის მენეჯერმა უნდა იცოდეს მის პროექტში რომელი ამოცანები დგას კრიტიკულ გზაზე და უზრუნველყოს მათი დროულად დასრულება, საწინააღმდეგო შემთხვევაში პროექტის ვადები გადაიწევის.

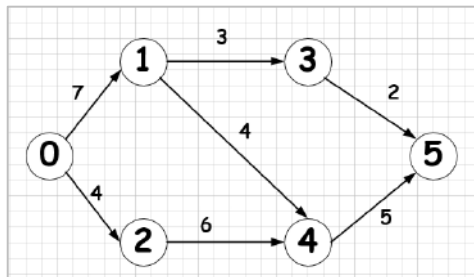
პროექტის კრიტიკული გზის შექმნა:

1. შევადგინოთ პროექტის ყველა დავალების/სამუშაოს სია და მათი ხანგრძლივობა. მაგალითად ოთახის რემონტის სიაში გვექნება შემდეგი დავალებები/სამუშაოები: ძველი კედლების დანგრევა, ახალი კედლების აშენება, კედლების გალესვა, ქაღალდის გაკვრა, ელექტროენერგიის მოწყობა, წყლის მიწოდება, ლამინატის დაგება და ა.შ.

ამ ეტაპზე, ჩვენი მიზანია აღვწეროთ სამუშაოების სრული ჩამონათვალი და გავაცნობიეროთ, თუ რამდენი დრო დასჭირდება თითოეულ მათგანს: გალესვისათვის დაგჭირდება 3 დღე, ელექტროობის მოწესრიგებისთვის - 10 დღე, ლამინატის დაგებისთვის - 4 დღე და ა. შ.

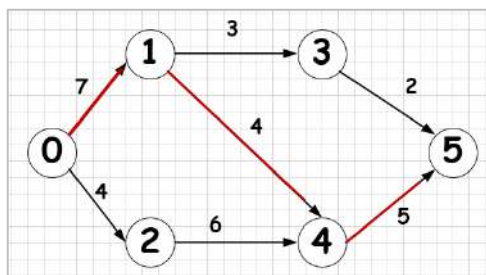
2. დავაზუსტოთ, თუ რა კავშირშია ეს საქმიანობები ერთმანეთთან. მაგალითად, რა არის საჭირო რომ დავიწყოთ ელექტროენერგიის გაყვანა? მინიმუმ ჯერ კედლები უნდა ამოვიყვანოთ. თუ კედლები ამოყვანილია, მაშინ შეგვიძლია ელექტრო გაყვანილობის სამუშაოს დაწყება, ხოლო ქაღალდის გაკვრა შეიძლება დავიწყოთ სულ ბოლოს, როდესაც ლამინატიც დაგებულია, ელექტროობაც დამონტაჟებულია და, რა თქმა უნდა, კედლებიც გალესილია.

3. შემდეგ ეტაპზე შევიმუშაოთ სამუშაოების გრაფიკი ანუ გრაფი - წრეებში ჩავწეროთ სამუშაოს ნომრები (წვეროები), წრეები დააკავშიროთ ისრებით იმის მიხედვით, თუ რომელი სამუშაო რომელი სამუშაოს შემდეგ მიდის და ისრებს დავაწეროთ სამუშაოს ხანგრძლივობა (ნახ. 7.22.).



ნახ. 7.22. სარემონტო სამუშაოების გრაფი

ნახატზე 7.22-დან გამომდინარე, შეგვიძლია დავითვალოთ თუ რომელ სამუშაოთა ჯაჭვი არის ყველაზე გრძელი და გამოვყოთ ეს სამუშაოები განსხვავებული ფერით (ნახ. 7.23.).



ნახ. 7.23. ყველაზე გრძელი სამუშაოთა მიმდევრობების გზა

ნახატ 7.23-ზე დაყრდნობით დავადგინეთ პროექტის კრიტიკული გზა. უკვე ვიცით, რომ პროექტის ხანგრძლივობა დამოკიდებულია 1-ელი, მე-4 და მე-5 ამოცანების შესრულებაზე ანუ თუ ეს სამუშაოები შესრულდება დროულად შესრულდება მთელი პროექტიც.

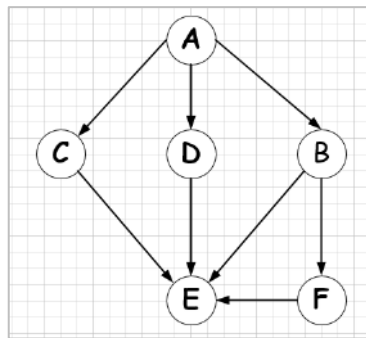
უმოკლესი გზა ორიენტირებულ აციკლურ გრაფში. იმისათვის, რომ გავიგოთ უმოკლესი გზა ორიენტირებულ აციკლურ გრაფში, განვიხილოთ მაგალითი 7.12.

მაგალითი 7.12. მოცემული გვაქვს ორიენტირებულ აციკლურ გრაფი ექვსი წვეროთი. მოვძებნოთ უმოკლესი გზა A წვეროდან F წვერომდე.

დასმული ამოცანის მისაღწევად შეგვიძლია ავირჩიოთ ორი ვარიანტი:

1. გრაფის წვეროებს შორის ვიპოვოთ ყველა გზა და ავირჩიოთ ყველაზე მოკლე გზა;
2. ავირჩიოთ უმოკლესი გზა სასურველ წვერომდე მისაღწევად.

შევჩერდეთ მე-2 ვარიანტზე და გამოვიყენოთ სიგანეში ძებნის ალგორითმი (BFS). როგორც უკვე ვიცით, ალგორითმი BFS იყენებს მონაცემთა სტრუქტურას რიგს (ნახ. 7.24.).

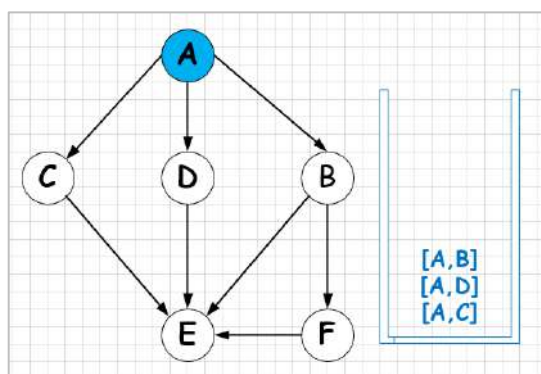


ნახ. 7.24. ორიენტირებულ აციკლური გრაფი

ბიჯი 1. რიგში დავამატოთ წვერო [A].

ბიჯი 2. ამოვიღოთ რიგიდან პირველი ელემენტი, ჩვენს შემთხვევაში წვერო A და გადავიდეთ მისი მეზობელი წვეროების დათვალიერებაზე, კერძოდ, A, B, C წვეროებზე. ამჟამად რიგი არის ცარიელი.

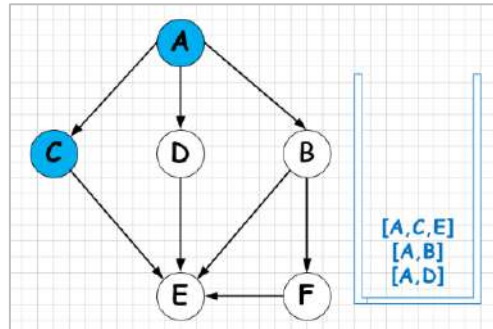
ბიჯი 3. რიგში დავამატოთ A-ს მეზობელი წვეროები და A წვერო მოვნიშნოთ როგორც ნანახი ანუ გავაფერადოთ. ამ ბიჯზე რიგში გვაქვს [A,C] [A,D] [A,B] ელემენტები (ნახ. 7.25.).



ნახ. 7.25. A წვეროს მონიშვნა და რიგში არსებული მონაცემები

ბიჯი 4. რიგიდან ამოვიღოთ პირველი ელემენტი. სამიზნე ელემენტი არის C.

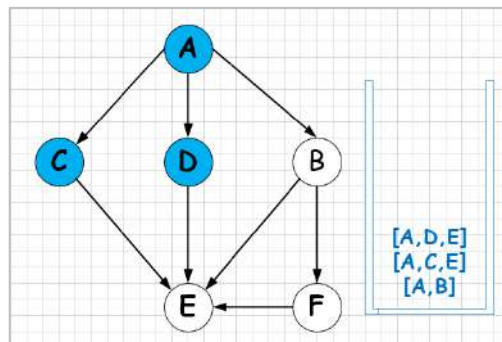
ბიჯი 5. რიგში C წვეროსთვის ვამატებთ მის მიმდებარე ყველა წვეროს. A წვერო ასევე C წვეროს მეზობელია, მაგრამ ვინაიდან A წვერო უკვე მონახულებულია - რიგში არ დავამატებთ. მოვნიშნოთ C წვერო, როგორც მონახულებული, რადგან სხვა წვეროებთან დამაკავშირებელი წიბოები აღარ არსებობს (ნახ. 7.26.).



ნახ. 7.26. C წვეროს მონიშვნა და რიგში არსებული მონაცემები

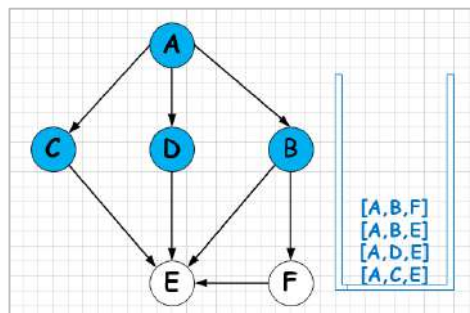
ბიჯი 6. რიგიდან ამოვშალოთ ელემენტი [A,D] და წვეროდ ავირჩიოთ რიგის ბოლო ელემენტი D.

ბიჯი 7. რიგში დავამატოთ D წვეროს მეზობელი წვეროები A და E. რადგან წვერო A მონახულებულთა სიაშია, ამიტომ მას „გავატარებთ“. რიგში დაემატება A, D, E ელემენტები (ნახ. 7.27.).



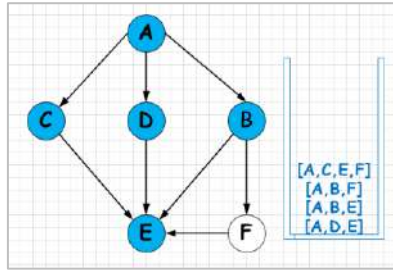
ნახ. 7.27. D წვეროს მონიშვნა და რიგში არსებული მონაცემები

ბიჯი 8. ამოვიღოთ რიგიდან შემდეგი ელემენტი [A,B]. ყველა ის წვერო, რომელიც B წვეროს მეზობელია (A-ს გარდა, რადგან A ნანახია) დავამატოთ რიგში. წვერო B მოვნიშნოთ როგორც მონახულებული (ნახ. 7.28.).



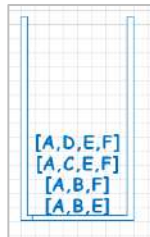
ნახ. 7.28. B წვეროს მონიშვნა და რიგში არსებული მონაცემები

ბიჯი 9. რიგიდან ამოვიღოთ შემდეგი ელემენტი $[A,C,E]$. წვეროს როლში ეხლა გამოდის სიაში მყოფი ბოლო ელემენტი - E. რიგში დავამატოთ E წვეროსთან მიმართებით ყველა მოუწახულებელი მეზობელი წვერო. წვერო E მოვნიშნოთ როგორც ნანახი (ნახ. 7.29.).



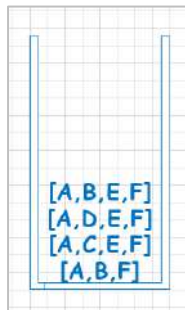
ნახ. 7.29. E წვეროს მონიშვნა და რიგში არსებული მონაცემები

ბიჯი 10. გადავიდეთ $[A,D,E]$ ელემენტზე და მის ბოლო წვეროზე E. წვერო E-სთვის დავამატოთ გზა F წვეროსკენ (ნახ. 7.30.).



ნახ. 7.30. გზა F წვეროსკენ და რიგში არსებული მონაცემები

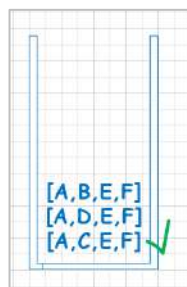
ბიჯი 11. გადავიდეთ $[A, B, E]$ ელემენტზე და მის ბოლო წვეროზე E. წვერო E-სთვის ისევ დავამატოთ გზა F წვეროსკენ (ნახ. 7.31.).



ნახ. 7.31. გზა F წვეროსკენ და რიგში არსებული მონაცემები

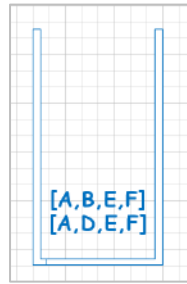
ბიჯი 12. გადავდივართ $[A, B, F]$ ელემენტზე, ბოლო წვეროა F. F არის საძიებო წვერო.

ჩვენ კვლავ ვცდილობთ რიგში დავამატოთ F წვეროს მეზობელი წვეროები, მაგრამ ამ შემთხვევაში ყველა მეზობელი წვერო უკვე შემოვლილია/ნანახია, ამიტომ ვაფიქსირებთ ნაპოვნ გზას და გადავდივართ რიგის მომდევნო ელემენტზე (ნახ. 7.32.).



ნახ. 7.32. ნაპოვნი გზის დაფიქსირება და რიგში არსებული მონაცემები

ბიჯი 13. ელემენტი $[A, C, E, F]$ შეიცავს სამივე F წვეროს, ჩავწერთ ნაპოვნი გზა და შევცადოთ რიგში დავამატოთ F წვეროს ყველა დაუთვალეირებული ელემენტი (ნახ. 7.33.).



ნახ. 7.33. რიგში არსებული მონაცემები

ბიჯი 14. ასევე ელემენტები $[A, D, E, F]$ და $[A, B, E, F]$ დავაფიქსიროთ როგორც გზა F წვეროსკენ და ამოვშალოთ ისინი რიგიდან. ამ ოპერაციების შემდეგ რიგი დაცარიელდება, რაც იმას ნიშნავს, რომ ჩვენ შემოვუარეთ გრაფის ყველა წვეროს (ნახ. 7.34.).



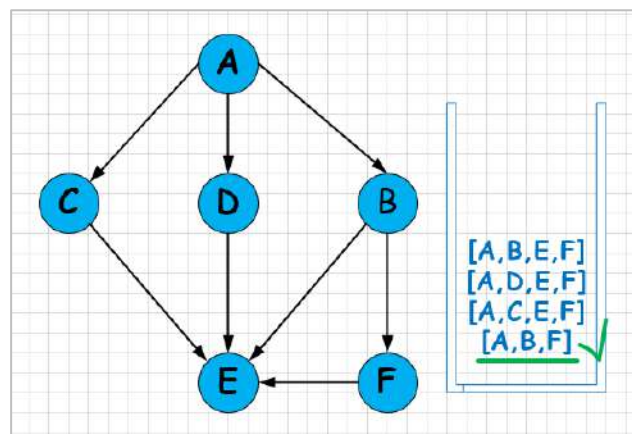
ნახ. 7.34. დაცარიელებული რიგი

შევაჯამოთ უმოკლესი გზის პოვნა A წვეროდან F წვერომდე ორიენტირებულ აციკლურ გრაფში (იხ. ნახ. 7.35.):

გრაფში შემოვლილი/მონიშნული წვეროების საბოლოო სია: A, C, D, B, E, F

გრაფში მოძებნილი გზების საბოლოო სია: $[A, B, F]$, $[A, C, E, F]$, $[A, D, E, F]$, $[A, B, E, F]$.

უმოკლეს გზად შეგვიძლია ჩავთვალოთ $[A, B, F]$ გზა.



ნახ. 7.35. გრაფში ნაპოვნი უმოკლესი გზა BFS ალგორითმის საფუძველზე

სირთულის შეფასება.

დროითი სირთულე არის $O(V + E)$, სადაც V არის მწვერვალების რაოდენობა, E კიდეების რაოდენობა. უარეს შემთხვევაში მოგვიწევს გრაფის ყველა წვეროს და წიბოს შემოვლა.

სივრცითი სირთულე არის $O(V)$. უარეს შემთხვევაში, რიგში V რაოდენობა წიბო იქნება შენახული. ასევე საჭირო იქნება V ცალი ნაპოვნი გზის შენახვა ყველაზე უარესი შემთხვევისთვის.

მაგალითი 7.13. მოცემული გვაქვს დასახლებული A, B, C, D, E, F, G პუნქტები. ამ პუნქტებს შორის დამაკავშირებელი გზების სიგრძე მოცემულია ცხრილში (იხ. ცხრილი 7.1.).

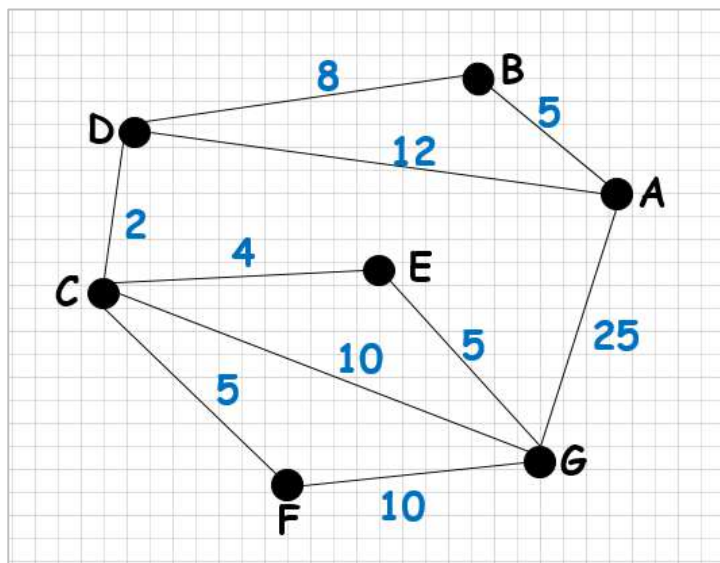
ცხრილში რიცხვის არ არსებობა, არ ნიშნავს, რომ პუნქტებს შორის პირდაპირი გზა არ არსებობს.

ცხრილი 7.1. დასახლებული A, B, C, D, E, F, G პუნქტებს შორის დამაკავშირებელი გზების სიგრძე

	A	B	C	D	E	F	G
A		5		12			25
B	5			8			
C				2	4	5	10
D	12	8	2				
E			4				5
F			5				5
G	25		10		5	5	

ცხრილი 7.1.-ის მონაცემების მიხედვით ავაგოთ გრაფი (იხ. ნახ. 7.36).

განვსაზღვროთ A პუნქტიდან G პუნქტამდე ყველაზე მოკლე გზის სიგრძე, იმ პირობით, რომ გადაადგილება შეიძლება მხოლოდ არსებული დამაკავშირებელი გზებით და არა სხვა რომელიმე გზით მაგ., მინდვრით, ტყით ან რომელიმე სხვა გზით.



ნახ. 7.36. ცხრილი 7.1.-ში მოცემული მონაცემების მიხედვით აგებული გრაფი

ამოხსნა: გრაფზე თვალნათლივ ჩანს რომ არის A პუნქტიდან G პუნქტამდე ერთი წიბოს სიგრძე არის 25.

მოდით შევეცადოთ დავადგინოთ A პუნქტიდან G პუნქტამდე მისასვლელი კიდევ უფრო მოკლე გზა. რადგან დანარჩენი წიბოების სიგრძეები არის დადებითი, ამიტომ უმოკლესი გზა არ შეიცავს ციკლს.

1. უმოკლესი გზა A პუნქტიდან B პუნქტამდე არის 5-ის ტოლი (ეს ერთი წიბოა), რადგან ყველა სხვა დანარჩენი A-დან გამომავალი წიბოს აქვს უფრო მეტი სიგრძე.

2. უმოკლესი გზის სიგრძე A პუნქტიდან D პუნქტამდე არის 12-ის ტოლი (ესეც ერთი წიბოა), რადგან ერთადერთი ალტერნატიული გზის სიგრძე $A \rightarrow B \rightarrow D$ პუნქტებს შორის = 13.

3. უმოკლესი გზის სიგრძე A პუნქტიდან C პუნქტამდე არის 14-ის ტოლი ($A \rightarrow D \rightarrow C$).

4. C პუნქტიდან G პუნქტამდე მიდის სამი გზა (ციკლების გარეშე):

- $C \rightarrow E \rightarrow G$ გზის სიგრძე = 9;
- $C \rightarrow G$ გზის სიგრძე = 10;
- $C \rightarrow F \rightarrow G$ გზის სიგრძე = 15;

5. C პუნქტიდან G პუნქტამდე ყველაზე მოკლე გზის სიგრძე არის 9 ($C \rightarrow E \rightarrow G$).

6. A პუნქტიდან G პუნქტამდე უმოკლესი „შემოვლითი“ გზა არის:

$$14 (A \rightarrow D \rightarrow C) + 9 (C \rightarrow E \rightarrow G) = \mathbf{23} < 25;$$

პასუხი: A პუნქტიდან G პუნქტამდე ყველაზე მოკლე გზის სიგრძე = **23**.

გრაფები. ხე, ტყე და მათი გამოყენება. ძირითადი ამოცანები ხეებზე, მათი გადაჭრის ალგორითმები. რამდენიმე რეალური ამოცანის სიმულაცია გრაფებით. ეილერის და ჰამილტონის ციკლების პოვნა გრაფებში, ალგორითმები. უმოკლესი გზები. დეიქსტრას ალგორითმები, ბელმან-ფორდის ალგორითმი, ფლოიდ უორშელის ალგორითმი. კომპიუტაციური ამოცანა, წრფივი გადაადგილებების ამოცანა.

გრაფები. ხე, ტყე და მათი გამოყენება. ხე-გრაფი არის წვეროების (კვანძების, nodes) ერთობა, რომელიც ვიზუალურად ჰგავს ჩვეულებრივ ხეს. აქედანაა მისი დასახელება. ხე-გრაფი არის იერარქიული სტრუქტურული მონაცემი. ხე-გრაფში ნებისმიერი ინდივიდუალური კვანძი, რომელიც ინახავს ინფორმაციას, წარმოადგენს წვეროს. წვერო ინახავს რეალურ ინფორმაციას და მომდევნო წვეროს მიმითითებულს.

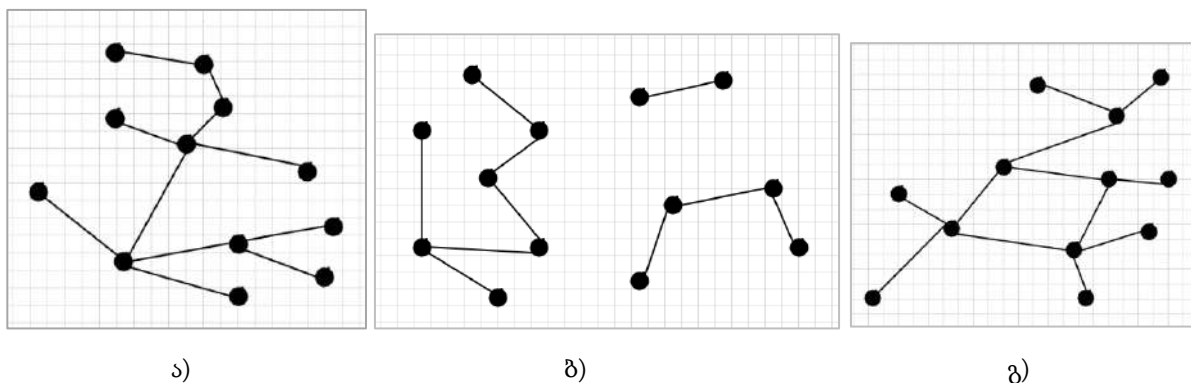
თუ ხე-გრაფში გვაქვს N რაოდენობის ელემენტი, მაშინ ელემენტებს შორის კავშირი უდრის $N - 1$ -ს.

ხე ეწოდება ისეთ ბმულ ხე-გრაფს, სადაც ციკლები არ გვაქვს. ხე-გრაფს ციკლების გარეშე ხშირად აციკლურს ან ტყეს (forest) უწოდებენ. ტყე შედგება ხეებისგან, რომლებიც მის ბმულ კომპონენტებს წარმოადგენენ. ხე-გრაფებისთვის ვარგისი ბევრი ალგორითმი გამოსადეგია ტყეებისთვისაც. ხის მწვერვალებს, რომელთა ხარისხი ერთია, ხის ფოთლები ეწოდება (იხ. ნახ. 8.1.). განვიხილოთ განსაზღვრებები ხე-გრაფებთან მიმართებით:

ხე - არის ბმული გრაფი ციკლების გარეშე (ბმული აციკლური გრაფი).

ხე - არის ბმული გრაფი, რომელის N რაოდენობის მქონე წვეოების დროს ყოველთვის გვაქვს $N - 1$ წიბო.

ხე - არის გრაფი, რომელშიც ნებისმიერ ორ წვეროს შორის არსებობს ზუსტად ერთი გზა.



ნახ. 8.1. მოცემული გრაფებიდან: ა) ხე-გრაფია;

ბ) ტყე (ის არ წარმოადგენს ხეს იმის გამო, რომ ბმული არაა);

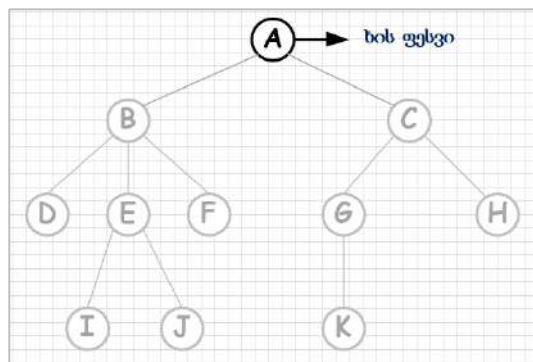
გ) არც ხე-გრაფია და არც ტყე, რადგან იგი ციკლს შეიცავს.

ხე-გრაფის გამოყენება შეიძლება ყველგან, სადაც იერარქიას ვხედავთ. ხე-გრაფის ყველაზე გავრცელებული მაგალითია ფაილური სისტემა, ოპერაციული სისტემა, რომელიც არსებულ ინფორმაციას აკონტროლებს, ალაგებს და ამუშავებს ხის დახმარებით. სამიეზო სისტემები ძირითადად იყენებენ ხის სტრუქტურას ინფორმაციის მოძიებისთვის. თამაშები იყენებენ ხე-გრაფს, როდესაც უწევთ ობიექტების ვიზუალიზაცია. ხე-გრაფი ფართოდ გამოიყენება Wi-Fi კავშირების კონტროლისთვის. მონაცემთა ბაზების ინდექსური სისტემა აგებულია ხე-გრაფის სტრუქტურის მიხედვით.

ხე-გრაფების ყველაზე გავრცელებული ფორმებია: Binary Tree(BT), Binary Search Tree(BST), Red-Black Tree(RBT), AVL Tree, AA Tree, B Tree, Heap. ჩამოთვლილთაგან ყველაზე გავრცელებული ხის ტიპია Binary Tree(BT), Binary Search Tree(BST).

ტერმინოლოგია ხე-გრაფებთან დაკავშირებით:

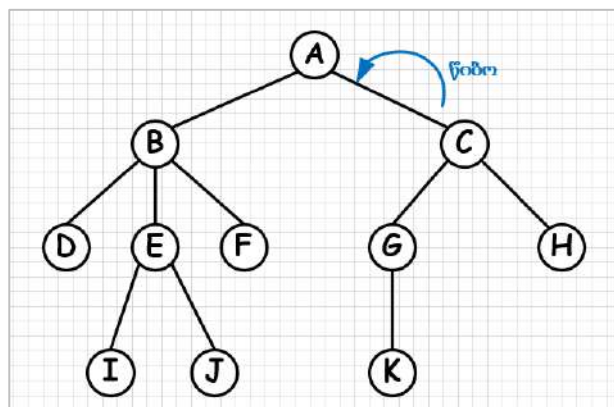
1. **ფესვი (Root).** წვეროს, რომელიც მოქცეულია ხის სათავეში უწოდებენ ხე-გრაფის ფესვს. არ არსებობს ხე-გრაფი რომელსაც არ გააჩნია ფესვი, აგრეთვე ვერ შეხვდებით ხე-გრაფს, რომელსაც ერთზე მეტი ფესვი აქვს. ნახატ 8.2.-ზე ხე-გრაფის ფესვი არის A წვერო (კვანძი).



ნახ. 8.2. გრაფზე ხე-გრაფის ფესვს წარმოადგენს A წვერო

2. **წიბო (Edge).** ხე-გრაფში წვეროებს შორის კავშირს წიბო (Edge) ეწოდება. ხეში წიბოს რაოდენობა უდრის წვეროების რაოდენობას მინუს ერთი.

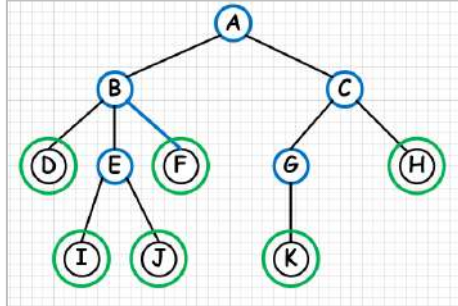
ნახატ 8.3.-ზე გვაქვს $N = 11$ წვერო, (Nodes) და $N - 1 = 10$ წიბო (Edge).



ნახ. 8.3. გრაფზე ხე-გრაფის ფესვს წარმოადგენს A წვერო

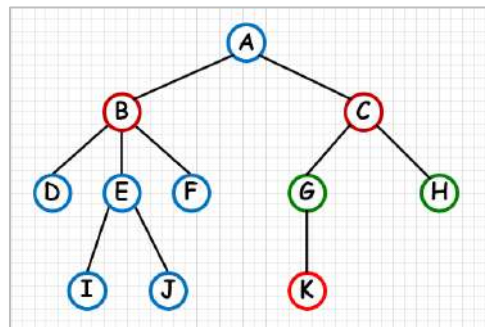
3. მშობელი წვერო (Parent). ხე-გრაფში მშობელი არის ისეთი წვერო, რომელსაც აქვს შვილობილი წვერო და არის ქვემდგომი წვერო ზემდგომი წვეროსი.

ხე-გრაფში მშობელს ზემდგომი წვერო არ გააჩნია, რადგან ის არის მშობელი ყველა დანარჩენი წვეროებისა. ნახატ 8.4.-ზე გვაქვს E, G, B, C, A მშობელი წვეროები, მათ აქვთ ზემდგომი და ქვემდგომი წვეროები, მწვანე ფერის წრეში მოქცეულ წვეროებს, როგორც ვხედავთ, არ გააჩნიათ ქვემდგომი წვეროები.



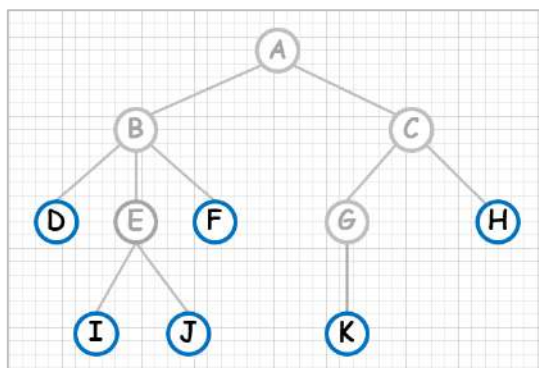
ნახ. 8.4. მშობელი წვერო ლურჯი ფერის მქონე წვეროებია

4. შვილი (მემკვიდრე, შთამომავალი) წვერო (Child). ხე-გრაფში წვეროს, რომელსაც აქვს კავშირი მშობელ წვეროსთან ვუწოდებთ შვილ წვეროს. ფესვი გამონაკლისია, მას არ ჰყავს მშობელი წვერო (კვანძი). ნახატ 8.5.-ზე B და C არიან შვილი წვეროები A წვეროსთვის, G და H არიან შვილი წვეროები C წვეროსთვის, K არის შვილი წვერო G წვეროსთვის.



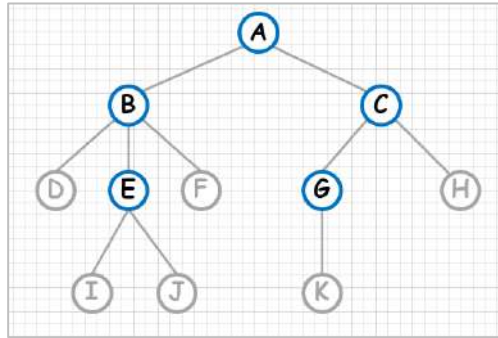
ნახ. 8.5. ხე-გრაფში B, C, G, H, K არის შვილი წვეროები

5. ფოთოლი (Leaf). ფესვის მქონე ხე-გრაფის წვეროს (კვანძს), რომელსაც არა ჰყავს შვილები, უწოდებენ ფოთოლს (Leaf, external node). ნახატ 8.6.-ზე მოცემულ ხე-გრაფში I, J, K, D, F, H არიან ფოთლები.



ნახ. 8.6. ხე-გრაფში ლურჯი ფერის წვეროები არის ხის ფოთლები

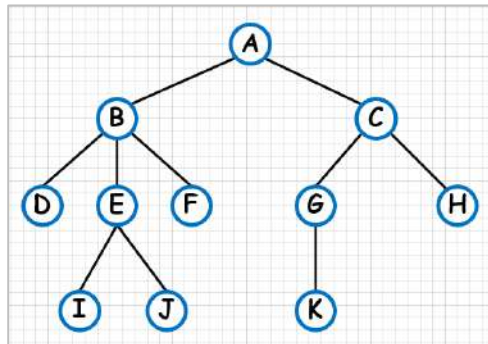
6. შიგა წვეროები (Internal, შიდა კვანძი). კვანძებს, რომელთაც ჰყავთ შვილები, უწოდებენ შიგა წვეროს (კვანძს, Internal node). ნახატ 8.7-ზე მოცემულ ხე-გრაფში A, B, C, E, G არიან შიგა წვეროები.



ნახ. 8.7. ხე-გრაფში ლურჯი ფერის წვეროები არის შიდა წვეროები

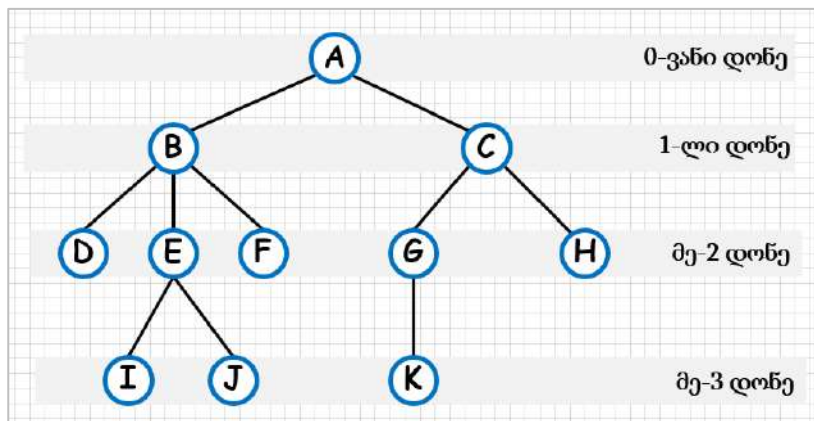
7. ხე-გრაფის წვეროს ხარისხი/წონა (Degree). ფესვის მქონე ხე-გრაფის წვეროებზე მყოფ შვილი წვეროების რაოდენობას უწოდებენ ხე-გრაფის ხარისხს (Degree, Degree of Tree).

ნახატ 8.8-ზე მოცემულ ხე-გრაფში E წვეროს ხარისხი = 2, B წვეროს ხარისხი = 3, A წვეროს ხარისხი = 2, H წვეროს ხარისხი = 0.



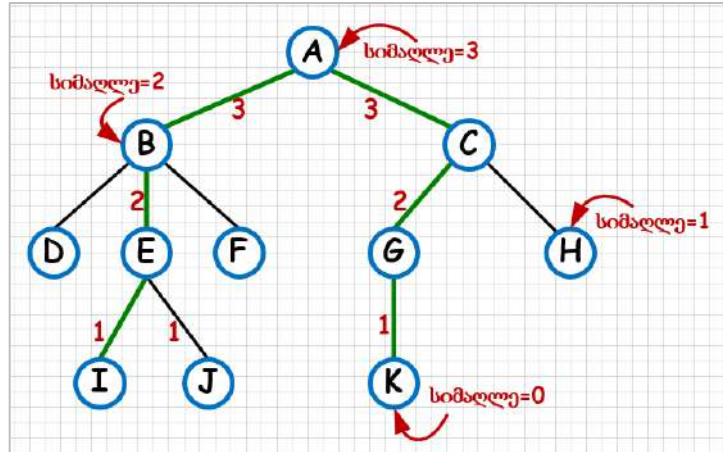
ნახ. 8.8. ხე-გრაფში E -ს ხარისხი = 2, B -ს ხარისხი = 3, A -ს ხარისხი = 2, H -ს ხარისხი = 0

8. საფეხური, დონე (Level). ხე-გრაფს აქვს საფეხურები: ყველა ახალი წვეროს დასაწყისი ახალ საფეხურად ითვლება, ხე-გრაფის ელემენტი ფესვი (Root) იმყოფება ნულოვან საფეხურზე, მისი შვილი ელემენტი იმყოფება პირველ საფეხურზე, შვილის შვილი იმყოფება მეორე საფეხურზე და ა. შ (ნახ. 8.9.).



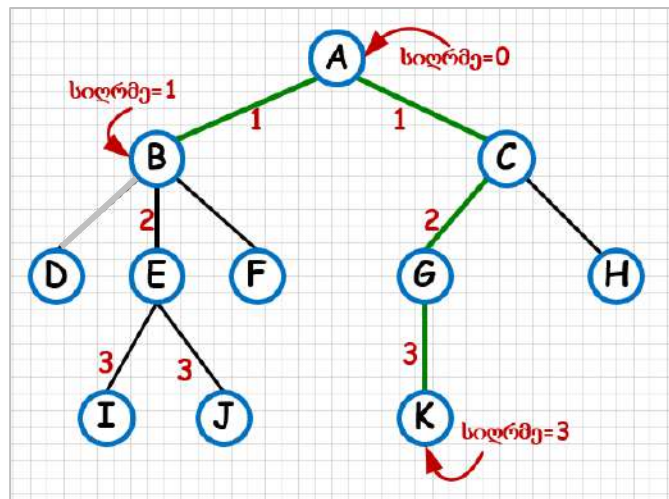
ნახ. 8.9. ხე-გრაფში საფეხურების (დონეების) თანმიმდევრობა

9. ხე-გრაფის წვეროს სიმაღლე (Height). ხე-გრაფში წვეროს სიმაღლე (Height) არის ყველაზე გრძელ გზაზე არსებული წიბოების რაოდენობა ამ წვეროდან მის შთამომავალ ფოთლამდე. ხე-გრაფის სიგრძელ ითვლება მისი ფესვის სიგრძე. დავუშვათ თუ ვზომავთ ფოთლიდან ფესვამდე სიმაღლეს, მაშინ ამ სიმაღლეს ხის სიმაღლე ქვია (Height Of Tree). ხის სიმაღლე = სათავის სიმაღლე = ყველაზე „ღრმა“ ფოთლის სიღრმე (ნახ. 8.10.).



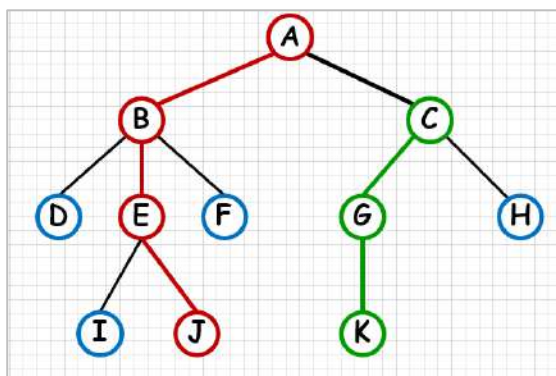
ნახ. 8.10. ხე-გრაფში წვეროთა სიმაღლის თანმიმდევრობა

10. ხე-გრაფის წვეროს სიღრმე (Depth). ხე-გრაფის წვეროს სიღრმე (Depth) არის გზის სიგრძე ხის ფესვიდან (სათავიდან) წვერომდე. გზას ფესვიდან ყველაზე შორს მდგარ ფოთლამდე უწოდებენ წვეროს სიღრმეს. ნახ. 8.11.-ში A წვეროდან D წვერომდე სიღრმე უდრის 2-ს, A-დან K წვერომდე ხე-გრაფის სიღრმე უდრის 3-ს, იმიტომ რომ K წვერო ყველაზე შორს მდგარი წვეროა.



ნახ. 8.11. ხე-გრაფში წვეროს სიღრმის თანმიმდევრობა

11. გზა და მისი სიგრძე (Path, Length). ორ წვეროს შორის არსებული წიბოების მიმდევრობა წარმოადგენს გზას ამ ორ წვეროს შორის, ხოლო წიბოების რაოდენობა არის ამ გზის სიგრძე. ხე-გრაფში იმ წვეროების თანმიმდევრობას, რომლებიც დაკავშირებულია წიბოებით, ჰქვია გზა (Path). ნახატ 8.12.-ზე A, B, E, J არის გზა, ასევე C, G, K აგრეთვე არის გზა. გზას, როგორც წესი აქვს სიდიდე, A, B, E, J-ის სიდიდე უდრის 4-ს.



ნახ. 8.12. ხე-გრაფში წითელი და მწვანე ფერით მოცემულია გზა

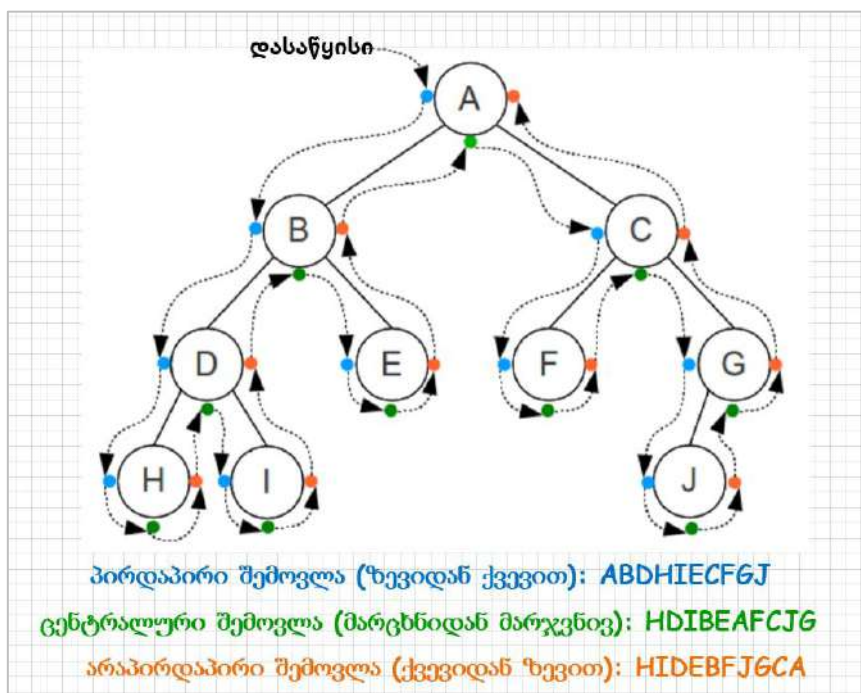
12. Visiting - ეს არის პროცესი, როდესაც ხეში ყველა წვეროს წაკითხვა ხდება. მარტივი ენით იტერაცია.

13. Traversing - არის პროცესი როდესაც გარკვეული ალგორითმით და თანამიმდევრობით ვკითხულობთ ელემენტებს.

14. Key - ინახავს წვეროს ხარისხს. მათზე დაყრდნობით ხდება იტერაცია და სხვა მნიშვნელოვანი ოპერაციები.

ხე-გრაფებთან მიმართებათ ერთ-ერთი მნიშვნელოვანი საკითხია ხე-გრაფის ელემენტების შემოვლა (იხ. ნახ. 8.13.). არსებობს ხე-გრაფის შემოვლის სამი ვარიანტი:

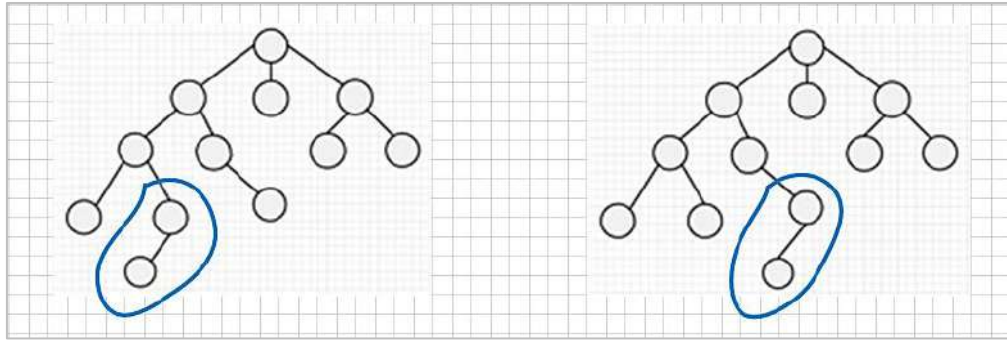
1. პირდაპირი შემოვლა (ზევიდან ქვევით): ფესვი $\rightarrow$ მარცხენა ქვეხე $\rightarrow$ მარჯვენა ქვეხე;
2. ცენტრალური შემოვლა (მარცხნიდან მარჯვნივ): მარცხენა ქვეხე $\rightarrow$ ფესვი $\rightarrow$ მარჯვენა ქვეხე;
3. არაპირდაპირი შემოვლა (ქვევიდან ზევით): მარცხენა ქვეხე $\rightarrow$ მარჯვენა ქვეხე $\rightarrow$ ფესვი;



ნახ. 8.13. ხე-გრაფის ელემენტების შემოვლა

ახლა განვიხილოთ ხე-გრაფის სახეობები:

1. დალაგებული ხე-გრაფი (ordered tree). დალაგებული ხე-გრაფი ეწოდება ფესვის მქონე ხეს, რომელსაც დამატებითი სტრუქტურა აქვს, ყოველი წვეროსთვის მისი შვილების სიმრავლე დალაგებულია (ცნობილია, თუ რომელია წვეროს პირველი შვილი, მეორე შვილი და ა.შ.). ნახატ 8.14.-ზე გამოსახულ ორ ხე-გრაფს ერთი და იგივე ფესვი აქვს, მაგრამ ისინი განსხვავდება შვილების დალაგებით.



ნახ. 8.14. ხე-გრაფში შვილების დალაგების ორი ვარიანტი

2. ორობითი ხე (Binary Tree, შემოკლებით BT). ორობითი ხე არის ისეთი ხე-გრაფი, რომლის ნებისმიერ წვეროს აქვს არაუმეტეს ორი შვილისა.

ორობითი ხე ყველაზე მარტივი სახით განისაზღვრება როგორც წვეროთა სასრული სიმრავლე, რომელიც ან ცარიელია (არ შეიცავს წვეროებს) ან გაყოფილია სამ არაგადამკვეთ ნაწილად:

წვერო, რომელსაც ეწოდება ფესვი (root);

ორობითი ხე, რომელსაც ეწოდება ფესვის **მარცხენა ქვეხე (left subtree)**;

ორობითი ხე, რომელსაც ეწოდება ფესვის **მარჯვენა ქვეხე (right subtree)**.

ორობით ხეს, რომელიც არ შეიცავს კვანძებს, ეწოდება ცარიელი (empty). ზოგჯერ მას აღნიშნავენ NIL-ით.

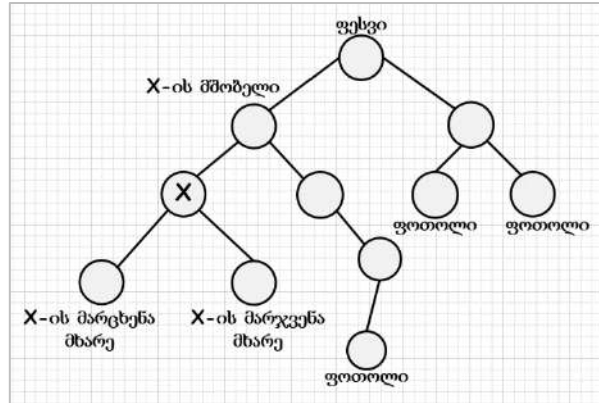
თუ მარცხენა ქვეხე არ არის ცარიელი, მაშინ მის ფესვს უწოდებენ მთლიანი ხის ფესვის **მარცხენა შვილს (left child)**, შესაბამისად განისაზღვრება **მარჯვენა შვილიც (right child)**.

არასწორი იქნებოდა განგვესაზღვრა ორობითი ხე, როგორც დალაგებული ხე, რომელშიც თითოეული წვეროს ხარისხი არ აღემატება 2-ს. ამის მიზეზია ის, რომ ორობით ხეში მნიშვნელობა აქვს თუ შვილი წვეროს რომელი მხარეს იმყოფება: მარცხენა მხარეს თუ მარჯვენა მხარეს, ხოლო დალაგებული ხისათვის ასეთი განსხვავება არ არსებობს.

ნახატ 8.14-ზე ორობით ხე-გრაფს აქვს სათავე, სიმაღლის ყველაზე მეტი დონით; ყოველ წვეროს აქვს 0, 1 ან 2 შვილი; წვეროები შვილების გარეშე წარმოადგენენ ფოთლებს;

X წვეროში განისაზღვრება მისი მარცხენა შვილი ($\text{left}(x)$), მარჯვენა შვილი ($\text{right}(x)$) და მშობელი ($\text{parent}(x)$).

ორობითი ხე-გრაფის სიმაღლედ ითვლება წიბოების რაოდენობათა ჯამური მნიშვნელობა, სათავიდან წვერომდე არსებულ ყველაზე გრძელ გზაში. ნახატ 8.15-ზე ნაჩვენებ ორობითი ხე-გრაფის სიმაღლე ტოლია 4-ის.



ნახ. 8.15. ორობითი ხე-გრაფი

სრული ორობითი ხე-გრაფი ეწოდება ისეთ ხეს, რომელშიც:

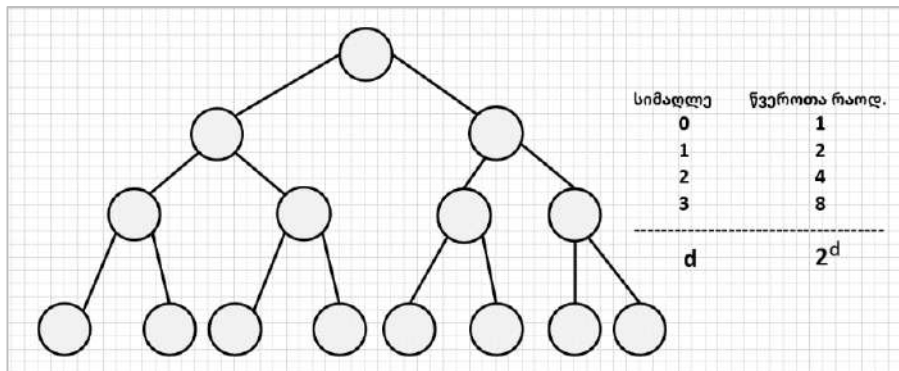
ყველა წვეროს აქვს 0 (ფოთოლი) ან 2 შვილი;

ყველა ფოთოლს ერთნაირი სიღრმე აქვს;

დამოკიდებულება წვეროსა და ხის სიმაღლეს შორის (იხ. ნახ. 8.16.):

N წვეროსგან შედგენილი სრული ორობითი ხის სიმაღლეა $O(\log N)$;

d სიმაღლის მქონე სრული ორობითი ხე შეიცავს $2^{d+1}-1$ წვეროს.



ნახ. 8.16. სრული ორობითი ხე-გრაფი

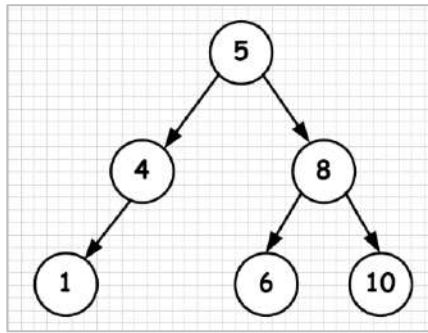
3. ძეხნის ორობითი ხე (BST - Binary Search Tree). ძეხნის ორობითი ხე განეკუთვნება დინამიურ მონაცემთა სტრუქტურას (იხ. ნახ. 8.16.).

თვისებები, რომლებიც განასხვავებს ძეხნის ორობით ხეს ჩვეულებრივი ორობითი ხისგან:

ძეხნის ორობითი ხის თითოეულ წვეროს უნდა ჰქონდეს არა უმეტეს ორი შვილი;

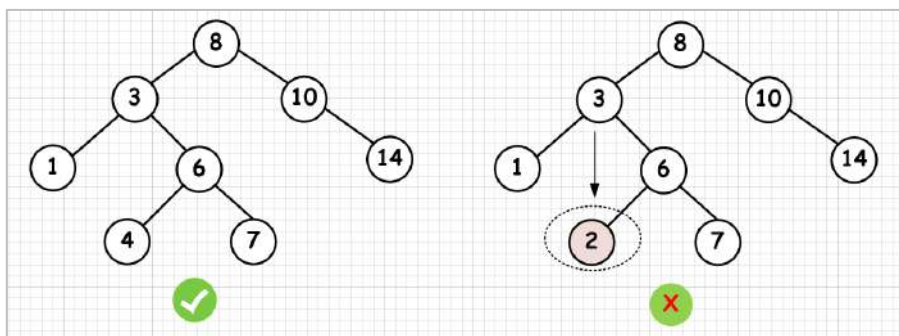
მარცხენა ქვეხის ყველა წვერო უნდა იყოს უფრო მცირე ვიდრე ფესვის მნიშვნელობა;

მარჯვენა ქვეხის ყველა წვერო უნდა იყოს უფრო დიდი, ვიდრე ფესვის მნიშვნელობა;



ნახ. 8.16. ძებნის ორობითი ხე-გრაფი

ნახატ 8.17-ზე მოცემულია ორობითი ხე-გრაფის ორი ვარიანტი, მაგრამ მარჯვნივ მდგომი ხე-გრაფი არ წარმოადგენს ძებნის ორობით ხე-გრაფს, რადგან მე-3 წვეროს მარჯვენა ქვეხე შეიცავს 3-ზე ნაკლებ მნიშვნელობას. იგი წითელი ფერით არის მონიშნული.



ა)

ბ)

ნახ. 8.17. ხე-გრაფი: ა) წარმოადგენს ორობითი ძებნის ხე-გრაფს;

ბ) არ წარმოადგენს ორობითი ძებნის ხე-გრაფს

ძებნის ორობით ხე-გრაფში რაიმე მნიშვნელობის მოძებნა ხდება $O(\log(n))$ დროში. ეს ნიშნავს, იმას რომ მილიონობით ან მილიარდობით ჩანაწერში შეგიძლიათ ძალიან სწრაფად იპოვოთ საძებნი მნიშვნელობა. თუ შევეცდებით შევაფასოთ ძებნის ორობითი ხის ალგორითმის მუშაობის სიჩქარე, აღმოვაჩენთ, რომ მას აქვს $O(\log(n))$ სიჩქარე.

შედარებისთვის, ჩვეულებრივ ძიებას აქვს სიჩქარე $O(n)$ ე.ი მოგვიწევს მასივის ყველა ელემენტის განხილვა. რას ნიშნავს ეს პრაქტიკაში:

მაგალითად, თუ გვაქვს 100 ელემენტი:

- ჩვეულებრივი ძებნით დაგვჭირდება 100 მცდელობა;
- ბინარული ძებნით დაგვჭირდება 7 მცდელობა.

თუ გვაქვს 4 000 000 000 ელემენტი:

- ჩვეულებრივი ძებნით დაგვჭირდება 4 000 000 000 მცდელობა;
- ბინარული ძებნით დაგვჭირდება 32 მცდელობა.

ვთქვათ, ძებნის ორობით ხეში ვეძებთ წვეროს რაიმე x -ის მნიშვნელობით. BST-ში მნიშვნელობის სწრაფად მოსაძებნად გამოიყენება შემდეგი ფსევდოკოდი:

1. ძებნას ვიწყებთ ხის ფესვიდან;

2. თუ წვეროს მნიშვნელობა უდრის x -ის მნიშვნელობას: გაჩერება;
3. თუ $x <$ წვეროს მნიშვნელობაზე: გადადით მარცხენა შვილ წვეროზე;
4. თუ $x >$ წვეროს მნიშვნელობაზე: გადადით მარჯვენა შვილ წვეროზე;
5. გადადით მეორე ბიჯზე;

BST-ში მნიშვნელობის სწრაფად მოძებნის ალგორითმი:

```

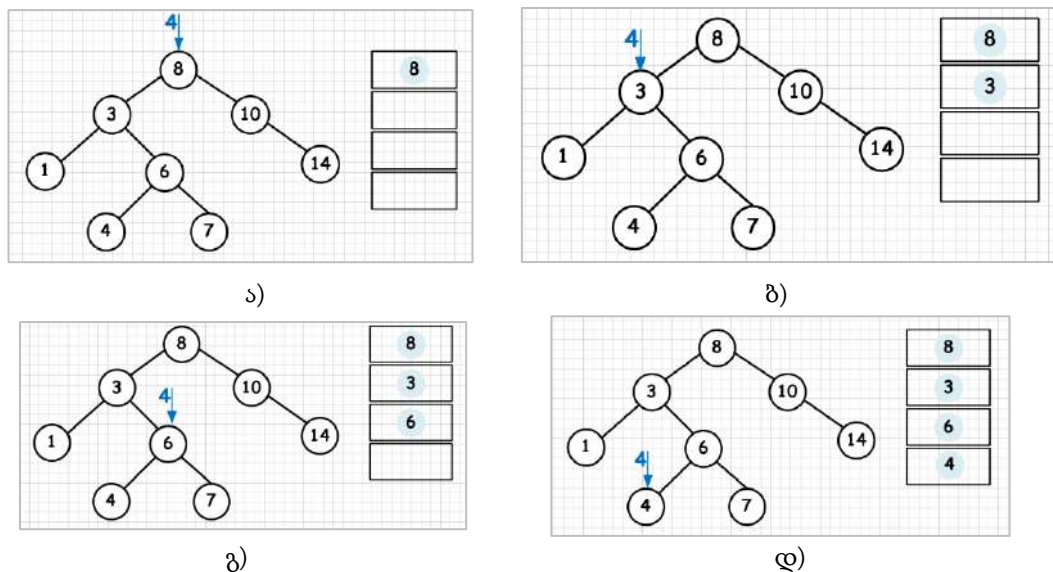
if root == NULL return NULL;
if number == root->data
    return root->data;
if number < root->data
    return search(root->left)
if number > root->data
    return search(root->right)

```

ალგორითმი დამოკიდებულია ხე-გრაფის თვისებებზე: თითოეულ მარცხენა ქვეხეს აქვს მნიშვნელობები, რომლებიც ნაკლებია, ვიდრე ფესვის მნიშვნელობა; თითოეულ მარჯვენა ქვეხეს აქვს მნიშვნელობები, რომლებიც მეტია ვიდრე ფესვის მნიშვნელობა.

თუ მნიშვნელობა ნაკლებია ფესვის მნიშვნელობაზე, შეგვიძლია დარწმუნებით ვთქვათ, რომ ის იმყოფება ისეთ ქვეხეზე, სადაც მნიშვნელობა არ აქვს მის მოძებნას, ამიტომ მხოლოდ მარცხენა ქვეხეში უნდა ვეძებოთ. თუ მნიშვნელობა უფრო მეტია, ვიდრე ფესვი, მაშინ შეგვიძლია დარწმუნებით ვთქვათ, რომ მნიშვნელობა არ აქვს მის მარცხენა ქვეხეში ძებნას, ამიტომ ვეძებთ მხოლოდ მარჯვენა ქვეხეში.

ვცადოთ ალგორითმის ვიზუალიზაცია (ნახ. 8.18.). სამეზნი მნიშვნელობა არის 4.



ნახ. 8.18. ძებნა ორობითი ხეში: ა) არ მოიძებნა; $4 < 8$; გადავდივართ 8-ის მარცხენა ქვეხეზე;

ბ) არ მოიძებნა; $4 > 3$; გადავდივართ 3-ის მარცხენა ქვეხეზე;

გ) არ მოიძებნა; $4 < 6$; გადავდივართ 6-ის მარცხენა ქვეხეზე; დ) მოიძებნა; $4 = 4$;

ეილერის და ჰამილტონის ციკლების პოვნა გრაფებში, ალგორითმები

განვიხილოთ ეილერის და ჰამილტონის ციკლები და გზა.

ეილერის გზა.

გრაფში ეილერის გზა არის ისეთი გზა, რომელიც გრაფის თითოეულ წიბოს გაივლის მხოლოდ (ზუსტად) ერთხელ. ეილერის გზა არის გზა, რომელიც გადის გრაფის ყველა წიბოზე და თან გადის ერთხელ და მხოლოდ ერთხელ.

ეილერის ციკლი.

ეილერის ციკლი არის ეილერის ჩაკეტილი გზა, რომელიც გრაფის თითოეულ კიდეზე ზუსტად ერთხელ გადის. თუ ეილერის გზის საწყისი წვერო ემთხვევა გზის საბოლოო წვეროს, მაშინ მიიღება ეილერის ციკლი (იხ. ნახ. 8.20.ა,ბ).

ჩაკეტილი ფიგურა, რომელშიც ყველა წვერო ლუწი ხარისხისაა, შეიძლება დახაზული იყოს ხელის აულებლად ნებისმიერი წვეროდან დაწყებული.

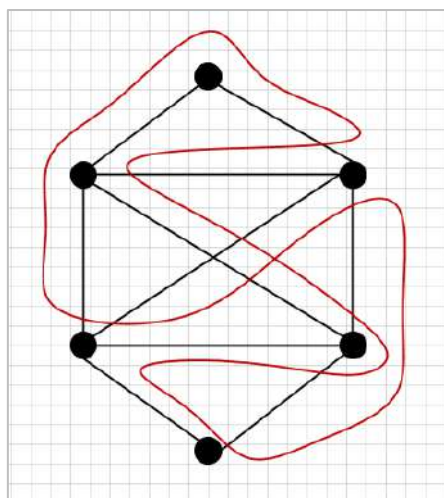
გრაფში ეილერის ციკლის პოვნა უკავშირდება ქაღალდზე ფიგურის ხელის აულებლად დახაზვას.

პრაქტიკაში ეილერის გრაფი შეიძლება იყოს სამხატვრო გალერეაში საგამოფენო დარბაზების დათვალიერების გეგმის აღწერა.

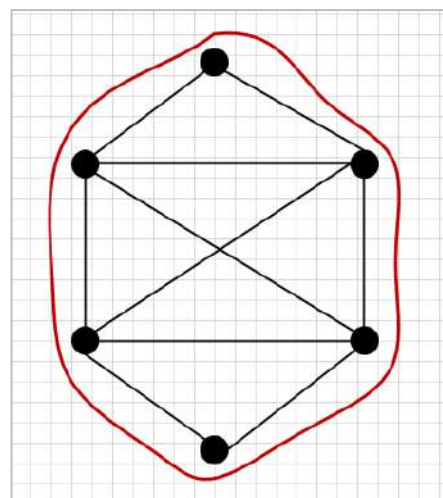
გეგმის მარშრუტი შეიძლება დაიხაზოს ისე, რომ დამთვალიერებელმა შეძლოს თითოეული საგამოფენო დარბაზის გავლა ზუსტად ერთხელ.

ჰამილტონის ციკლი. ჰამილტონის ციკლი ეწოდება ისეთ გზას, რომელიც გაივლის გრაფის ყველა წვეროზე და თითოეულ წვეროზე გაივლის ზუსტად ერთხელ და მხოლოდ ერთხელ. სხვა სიტყვებით რომ ვთქვათ, ეს არის მარტივი ციკლი, რომელიც მოიცავს გრაფის ყველა წვეროს (იხ. ნახ. 8.19.ა,ბ).

გრაფს უწოდებენ ჰამილტონის გრაფს, თუ მასში ჰამილტონის ციკლი არსებობს.



ა)



ბ)

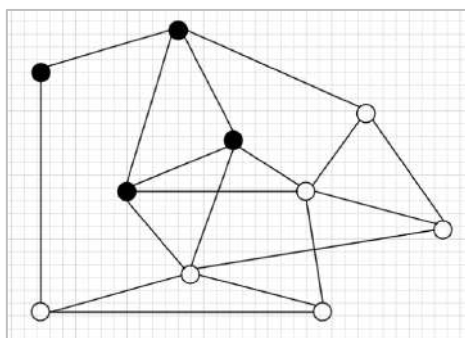
ნახ. 8.19. გრაფი: ა) ეილერის ციკლით; ბ) ჰამილტონის ციკლით;

ეილერის და ჰამილტონის გზები, ერთი შეხვედრით, თითქოს ერთმანეთის მსგავსია. ეილერი მოიცავს ყველა წიბოს და თანაც მხოლოდ ერთხელ, ჰამილტონი კი ყველა წვეროს და ისიც ერთხელ.

მიუხედავად გარეგნული მსგავსებისა, ეილერის და ჰამილტონის გზების პოვნის ამოცანები მკვეთრად განსხვავდება სირთულის მიხედვით. ეილერის ციკლის არსებობის საკითხის გადასაჭრელად საკმარისია იმის დადგენა, არის თუ არა მისი ყველა ყველა წვეროს ლუწი. ჰამილტონის ციკლის არსებობის კრიტერიუმი კი ჯერ კიდევ არ არის ნაპოვნი. თუმცა, არსებობს რამდენიმე საკმარისი პირობა ჰამილტონის ციკლის არსებობისთვის:

1. ნებისმიერი სრული გრაფი ჰამილტონის გრაფია, რადგან ის შეიცავს მარტივ ციკლს, რომელსაც გადის გრაფის ყველა წვეროზე.
2. თუ გრაფი, გარდა მარტივი ციკლისა, რომელიც გადის ყველა წვეროზე, შეიცავს სხვა წიბოებს, მაშინ ისიც ასევე ჰამილტონის ციკლია.
3. თუ გრაფს აქვს ერთი ჰამილტონის ციკლი, მას შეიძლება ჰქონდეს სხვა ჰამილტონის ციკლებიც.

ეილერის თეორემა გრაფში ციკლების შესახებ. მოცემულია გრაფი $G = \{V; E\}$, ვიცით რომ, მისი ნებისმიერი $v \in V$ წვეროს რიგი, იგივე ხარისხი, $(\deg(v))$ ეწოდება მასთან მიერთებულ წიბოთა რაოდენობას. ნებისმიერი გრაფის წვეროს ხარისხი (რიგი) შეიძლება იყოს ან კენტი, ან ლუწი. თუ წვეროს ხარისხი ლუწია, მას ლუწიან წვეროს, საწინააღმდეგო შემთხვევაში კი კენტიან წვეროს ვუწოდებთ. ნახატ 8.20.-ზე მოყვანილია გრაფი, რომლის კენტი ხარისხის მქონე წვეროები თეთრადაა ნაჩვენები, ხოლო ლუწი ხარისხის მქონე - შავად.



ნახ. 8.20. გრაფის წვეროები კენტი და ლუწი ხარისხით

ეილერის თეორემა. ეილერმა დაამტკიცა, რომ ნებისმიერ ბმულ გრაფში ეილერის ციკლი იარსებებს მაშინ და მხოლოდ მაშინ, თუ გრაფის ყველა წვეროს აქვს ლუწი ხარისხი ანუ გრაფს აქვს ლუწხარისხიანი წვეროები.

ლეონარდ ეილერის თეორემის კრიტერიუმები:

ეილერის ციკლი არსებობს მაშინ და მხოლოდ მაშინ, როცა გრაფს აქვს მხოლოდ ლუწი ხარისხის მქონე წვეროები.

ეილერის ციკლი არსებობს მაშინ და მხოლოდ მაშინ, როცა გრაფის კენტხარისხიანი წვეროების რაოდენობა ორს არ აღემატება ანუ გრაფს აქვს ორი ცალი კენტი ხარისხის მქონე წვერო.

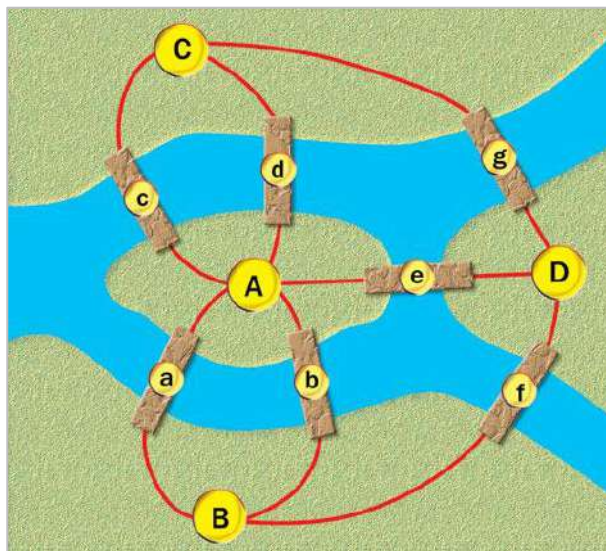
გრაფის თვისებები:

1. ნებისმიერ გრაფს აქვს მინიმუმ ორი წვერო, რომელსაც აქვს ერთი და იგივე ხარისხი.
2. ნებისმიერი გრაფის კენტ ხარისხიანი წვეროების რაოდენობა ყოველთვის არის ლუწი რიცხვი.
3. გრაფის ყველა წვეროს ხარისხების ჯამი უდრის წიბოების გაორმაგებულ რიცხვს.

გრაფთა თეორია. მე-18 საუკუნეში ქალაქი კენიგსბერგი (დღევანდელი კალინინ-გრადი, რუსეთი) ეკუთვნოდა აღმოსავლეთ პრუსიას და გაშენებული იყო მდინარე პრეგელის ორივე მხარეს. თვითონ მდინარეშიც იყო ორი კუნძული.

ხმელეთის ერთი ნაწილიდან მეორე ნაწილზე დასაკავშირებლად მდინარე პრეგელზე გადებული იყო შვიდი ხიდი.

მოქალაქეები დაინტერესებული იყვნენ შვიდივე ხიდზე ისე გადასულიყვნენ, რომ ერთი და იგივე ხიდი არ გაევილით ხელმეორედ? ეს გასართობი მალე თავსატეხად გადაიქცა და იგი 1736 წელს განიხილა მსოფლიოში ერთ-ერთმა უდიდესმა მათემატიკოსმა ლეონარდ ეილერმა. მან დახატა კენიგსბერგის შვიდი ხიდის შესაბამისი გრაფი. კვლევების შემდეგ კი ლეონარდ ეილერმა დაამტკიცა, რომ ამ საკითხის გადაჭრა შეუძლებელია (ნახ. 8.21.).



ნახ. 8.21. ქალაქ კენიგსბერგის, მდინარე პრეგელის და მასზე გადებული შვიდი ხიდის სქემა

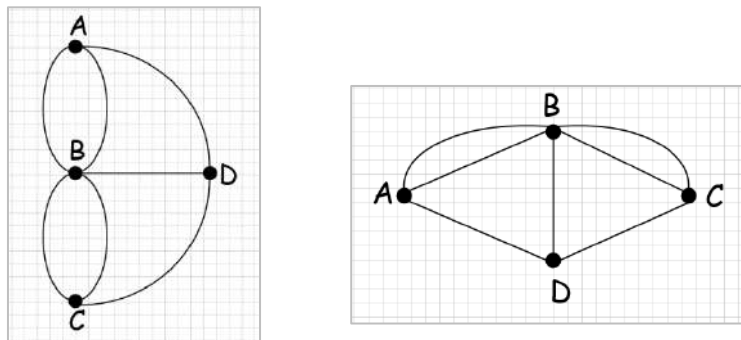
ეილერის ამოცანა. წიბოთა შემოვლის ამოცანა. კენიგსბერგის შვიდი ხიდის ამოცანა. ეილერის ამოცანას მეორენაირად შეიძლება ვუწოდოთ **წიბოთა შემოვლის ამოცანა**.

ეილერის მიერ დასმული ამოცანა (იგივე კენიგსბერგის შვიდი ხიდის ამოცანა) შეიძლება აღვწეროთ ქვემოთ მოცემული ორი ვარიანტის სახით:

1. შეიძლება თუ არა მოცემული გრაფის ყველა წიბოს ისე შემოვლა, რომ თითოეულ წიბოზე გავიაროთ მხოლოდ ერთხელ? არსებობს თუ არა გრაფში ისეთი გზა, რომელიც გრაფის ყოველ წიბოს თითოჯერ შეიცავს? ასეთ გზას ეილერის გზა ეწოდება.

2. შეიძლება თუ არა გრაფის შემოვლა ასე: დავიწყოთ რომელიმე წვეროდან, შემოვიაროთ გრაფი ისე, რომ ყველა წიბო მხოლოდ ერთხელ გავიაროთ და დავბრუნდეთ საწყის წვეროში. არსებობს თუ არა გრაფში ისეთი ციკლი, რომელიც გრაფის ყოველ წიბოს თითოჯერ შეიცავს? ასეთ ციკლს ეილერის ციკლი ეწოდება.

ლეონარდ ეილერმა ხმელეთის ცალკეული ნაწილი აღიქვა წვეროებად, ხოლო ამ ნაწილების შემაერთებელი ხიდები - წიბოებად. შვიდი ხიდის ამოცანის გრაფს აქვს ოთხი წვერო (ოთხივე წვერო არის კენტი ხარისხის) და შვიდი წიბო. საბოლოოდ ეილერმა მიიღო წიბოებისა და წვეროებისაგან შემდგარი სურათი, რასაც შემდგომში ეილერის გრაფი ეწოდა (ნახ. 8.22.).



ნახ. 8.22. კენიგსბერგის შვიდი ხიდის ამოცანის გრაფი, იგივე ეილერის გრაფი, ორივე სურათი ერთსა და იმავე გრაფს აღწერს

ამ გრაფის ყველა წვერო კენტი ხარისხისაა, ამიტომ ეილერის თეორემის მიხედვით, მათი შემოვლა ისე, რომ ყოველ ხიდზე მხოლოდ ერთხელ გავიაროთ და ბოლოს საწყის წერტილში დავბრუნდეთ, შეუძლებელია.

ეილერის აღმოჩენა არის, სწორედ ის, რომ მოხდა თითოეული წვეროდან გამომავალი წიბოების დათვლა. ნახატ 8.22-ზე ჩანს, რომ:

- A წვეროდან გამოდის სამი წიბო;
- B წვეროდან გამოდის ხუთი წიბო;
- C წვეროდან გამოდის სამი წიბო;
- D წვეროდან გამოდის სამი წიბო;

ოთხივე წვერო აღმოჩნდა კენტი ხარისხის მქონე. ამიტომ ეილერის ამოცანის პასუხი უარყოფითია. რატომ? ეილერის თეორემის თანახმად.

სწორედ ეილერის თეორემა გვამღევს უარყოფით პასუხს, რადგან თითოეული წვეროს ხარისხი ამ ამოცანის შესაბამის გრაფში კენტი რიცხვია.

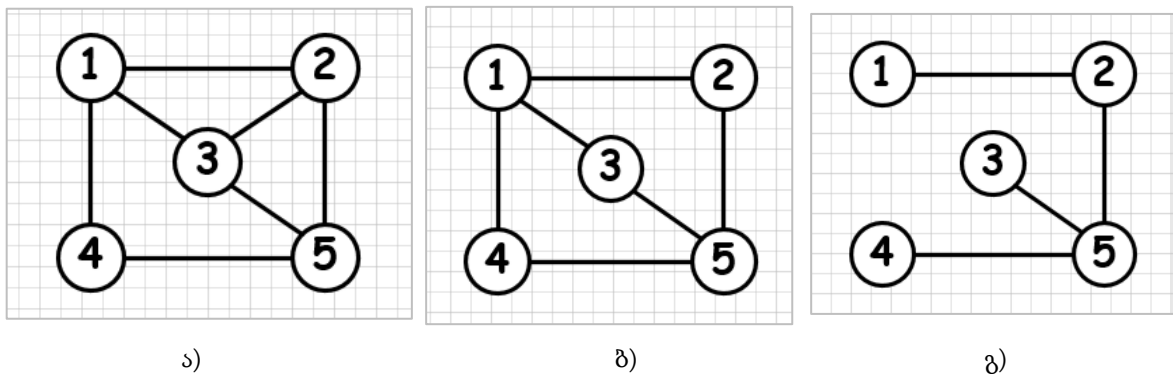
დასკვნა: კენიგსბერგის შვიდი ხიდის ამოცანის შესაბამისი გრაფის ოთხივე წვერო კენტი ხარისხისაა ე. ი. შუძლვბელია გადავიდეთ შვიდივე ხიდზე მხოლოდ ერთხელ და თან ხიდების შემოვლა დავასრულოთ იქ, საიდანაც დავიწყეთ იგი.

ჰამილტონის ამოცანა. წვეროთა შემოვლის ამოცანა. ჰამილტონის ამოცანას მეორენაირად შეგვიძლია ვუწოდოთ **გრაფის წვეროთა შემოვლის ამოცანა**.

ჰამილტონის მიერ დასმული ამოცანა (იგივე ამოცანა მოგზაურობა მსოფლიოს გარშემო) შეიძლება ქვემოთ მოცემული ვარიანტების სახით აღვწეროთ:

1. შეიძლება თუ არა მოცემული გრაფის ყველა წვეროს ისე შემოვლა, რომ თითოეულ წვეროზე გავიაროთ მხოლოდ ერთხელ? არსებობს თუ არა გრაფში ისეთი გზა, რომელიც გრაფის ყოველ წვეროს თითოჯერ შეიცავს? ასეთ გზას ჰამილტონის გზა ეწოდება.

2. შეიძლება თუ არა გრაფის შემოვლა ასე: დავიწყეთ რომელიმე წვეროდან, შემოვიაროთ გრაფი ისე, რომ ყველა წვერო მხოლოდ ერთხელ გავიაროთ და დავბრუნდეთ საწყის წვეროში. არსებობს თუ არა გრაფში ისეთი ციკლი, რომელიც გრაფის ყოველ წვეროს თითოჯერ შეიცავს? ასეთ ციკლს ჰამილტონის ციკლი ეწოდება (ნახ. 8.23. ა,ბ,გ).



ნახ. 8.23. გრაფში: ა) ჰამილტონის ციკლს წარმოადგენს მიმდევრობა: 1,2,3,5,4,1;

ბ) არ არის ჰამილტონის ციკლი, მაგრამ არის ჰამილტონის გზა: 2,1,3,5,4;

გ) არ არის არც ჰამილტონის ციკლი და არც გზა;

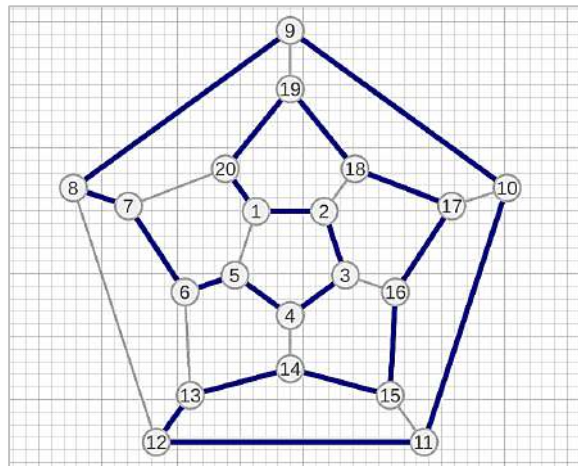
ჰამილტონის ციკლის შემოტანა დაკავშირებულია ამოცანასთან **მოგზაურობა მსოფლიოს გარშემო**. ჰამილტონმა ამ ამოცანის ამოსახსნრლად გამოიყენა გეომეტრიული ფიგურა წესიერი დოდეკაედრი/თორმეტწახნაგა (წესიერ დოდეკაედრს აქვს 12 წესიერი ხუთკუთხა წახნაგი, 30 წიბო და 20 წვერო. თითოეულ წვეროში თავს იყრის 3 წიბო).

თავდაპირველად ამოცანაში **მოგზაურობა მსოფლიოს გარშემო** დოდეკაედრის მწვერვალები განასახიერებდა ცნობილ ქალაქებს: ბრიუსელი, ამსტერდამი, ედინბურგი, პეკინი, პრაღა, დელი, ფრანკფურტი და ა. შ., ხოლო კიდეები ასახავდა ქალაქებს შორის დამაკავშირებელ გზებს.

მოგზაურმა უნდა გაიაროს „მსოფლიოს გარშემო“, მოძებნოს გზა, რომელიც გადის ყველა ქალაქზე ზუსტად ერთხელ და დაუბრუნდეს მოგზაურობის საწყის წერტილს.

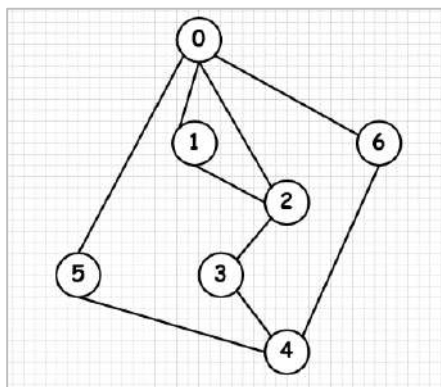
დავალება უფრო საინტერესო რომ ყოფილიყო, ქალაქების გავლის პროცედურა წინასწარ იყო დადგენილი.

იმისთვის, რომ გაადვილებულიყო იმის დამახსოვრება, თუ რომელი ქალაქები იყო უკვე გავლილი, ხდებოდა დოდეკაედონის თითოეულ წვეროზე ლურსმნის ჩასმა, გავლილი გზა კი აღინიშნებოდა პატარა თოკით, რომელსაც შემოახვევდნენ ლურსმანს. თუმცა, ასეთი დიზაინი ძალიან მოცულობითი და დამაბნეველი აღმოჩნდა. გამილტონმა შემოიტანა თამაშის „მსოფლიოს გარშემო“ ახალი ვერსია, შეცვალა დოდეკაედრი ბრტყელი გრაფით, რომელიც ააგო დოდეკაედრის წიბოებზე/კიდეებზე (ნახ. 8.24.).

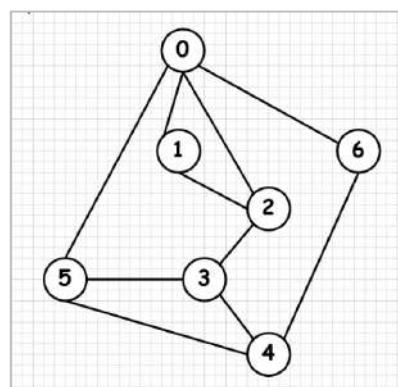


ნახ. 8.24. ბრტყელ გრაფში ჰამილტონის გზა მოცემულია ლურჯი ფერის წიბოებით ეილერის და ჰამილტონის ციკლების პოვნა გრაფებში.

მაგალითი 8.1. მოცემული გვაქვს ორი გრაფი (ნახ. 8.25.ა,ბ). დავადგინოთ მოცემულ გრაფებში არსებობს თუა არა ეილერის ციკლი. ასევე წარმოვადგინოთ ეილერის ალგორითმის შესრულების დინამიკა.



ა)



ბ)

ნახ. 8.25. გრაფში: ა) არის ეილერის ციკლი; ბ) არ არის ეილერის ციკლი;

ნახატ 8.25.ა.-ზე მოცემული გრაფის შემოწმება დავიწყეთ ნულოვანი წვეროდან. ნახატ 8.25.ა.-ზე მოცემული გრაფის ყველა წიბოს შემოვლის ერთ-ერთი შესაძლო გზაა:

0 → 1 → 2 → 0 → 6 → 4 → 2 → 3 → 4 → 5 → 0

წვეროების მიმდევრობაში, ნულოვანი წვეროდან დაწყებული, თითოეული წვერო ერთსა და იმავე დროს, წარმოდგენილია როგორც წინა წიბოს დასასრული და მომდევნო წიბოს დასაწყისი. ეილერის თეორემის მიხედვით:

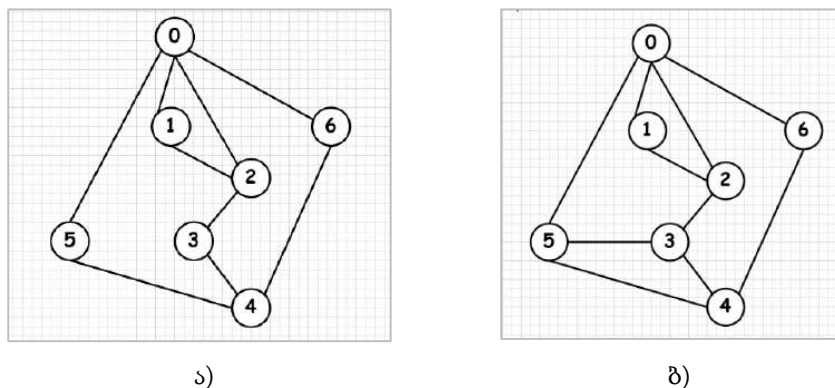
ნულოვანი წვეროს ხარისხია 4; 1-ლი წვეროს ხარისხია 2; მე-2 წვეროს ხარისხია 4; მე-3 წვეროს ხარისხია 2; მე-4 წვეროს ხარისხია 4; მე-5 წვეროს ხარისხია 2; მე-6 წვეროს ხარისხია 2;

როგორც ვხედავთ, გრაფის ყველა წვეროს ხარისხი ლუწია, ამიტომ ნახატ 8.25.ა.-ზე მოცემულ გრაფში არსებობს ეილერის ციკლი.

ნახატ 8.25.ბ-ზე მოცემულ გრაფში მოხდა ერთი წიბოს დამატება: ეს არის მესამე და მეხუთე წვეროებს შორის გავლებული წიბო. მე-3 და მე-5 წვეროების ხარისხი არის კენტი მნიშვნელობის. ეილერის ციკლის არსებობის შემოწმების ალგორითმის თანახმად, რომელიც დაფუძნებულია თითოეული წვეროს ხარისხის ლუწობაზე, ამ გრაფში ციკლი არ შეიძლება არსებობდეს.

პასუხი: ნახ. 8.25.ა-ზე მოცემულ გრაფში არსებობს ეილერის ციკლი, ხოლო ნახატ 8.25.ბ-ზე მოცემულ გრაფში არ არსებობს ეილერის ციკლი.

მაგალითი 8.2. მოცემული გვაქვს ორი გრაფი (ნახ. 8.26.ა,ბ). შევამოწმოთ არსებობს თუ არა მოცემულ გრაფებში ჰამილტონის ციკლის და ვნახოთ რეკურსიული ალგორითმის შესრულების დინამიკა.



ნახ. 8.26. გრაფში: ა) არ არის ჰამილტონის ციკლი; ბ) არის ჰამილტონის ციკლი;

ნახატ 8.26.ა.-ზე მოცემულ გრაფში შემოწმება დავიწყეთ ნულოვანი წვეროდან:

0-1; 1-2; 2-3; 3-4; 4-5; 4-6; 2-4; 4-3; 4-5; 4-6; 0-2; 2-1; 2-3; 3-4; 4-5; 4-6; 2-4; 4-3; 4-5; 4-6; 0-5; 5-4; 4-2; 2-1; 2-3; 4-3; 3-2; 2-1; 4-6; 0-6; 6-4; 4-2; 2-1; 4-3; 3-2; 2-1; 4-5;

როგორც ვხედავთ ნახატ 8.26.ა.-ზე ჰამილტონის ციკლი არ არსებობს.

ახლა კი გადავიდეთ ნახატ 8.26.ბ.-ზე მოცემულ გრაფზე და გავარკვიოთ გრაფს აქვს თუ არა ჰამილტონის ციკლი.

ერთ-ერთი ვარიანტი ციკლისა შეიძლება იყოს ასეთი:

0 → 6 → 4 → 2 → 1 → 3 → 5 → 0

ნახატ 8.26.ბ.-ზე მოცემულ გრაფზე ჰამილტონის ციკლის პოვნა დავწყოთ პირველი წვეროდან:

1-0; 0-5; 5-3; 3-2; 2-4; 4-6; 3-4; 4-2; 4-6; 5-4; 4-2; 2-3; 4-6;

0-6; **6-4;** 4-2; 2-3; 3-5; 4-3; 3-2; 3-5;

4-5; **5-3;** **3-2;** **2-1;**

ნახატ 8.26.ბ.-ზე მოცემულ გრაფზე ჰამილტონის მოძებნილი ციკლი არის შემდეგი სახის:

1 → 0 → 6 → 4 → 5 → 3 → 2 → 1

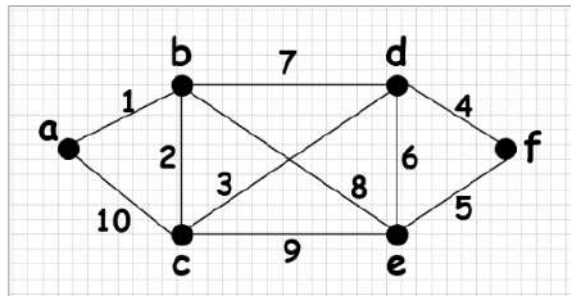
რეალურ პრაქტიკაში ეილერის თეორემა ცოტა სხვაგვარი ალგორითმის სახით გამოიყენება და მას **ფლერის ალგორითმს** (Fleury's) უწოდებენ.

განვიხილოთ ფლერის ალგორითმი. დაწყებული გრაფის ნებისმიერი წვეროდან, გავიაროთ გრაფის წიბოები შემდეგი წესების დაცვით:

1. გრაფიდან ამოვშალოთ უკვე გავლილი წიბოები და ის იზოლირებული წვეროები, რომლებიც წარმოიქმნება გავლილი წიბოს ამოშლისას.

2. ხიდზე მხოლოდ მაშინ გავდივართ, როცა სხვა გზა არ გვაქვს.

მაგალითი 8.3. მოცემულია გრაფი (ნახ. 8.27.). დაადგინეთ გრაფი წარმოადგენს თუ არა ეილერის გრაფს.

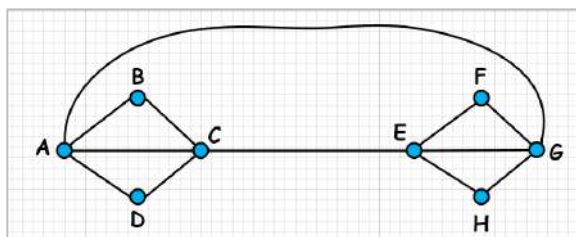


ნახ. 8.27. ეილერის გრაფი

ნახატ 8.27-ზე მოცემული გრაფი ეილერის თეორემის თანახმად, წარმოადგენს ეილერის გრაფს. დიაგრამაზე წიბოები დანომრილია მათი შემოვლის თანამიმდევრობით.

ეილერის ციკლია: **$a \rightarrow b \rightarrow c \rightarrow d \rightarrow f \rightarrow e \rightarrow d \rightarrow b \rightarrow e \rightarrow c \rightarrow a$**

მაგალითი 8.4. ფლერის ალგორითმის გამოყენებით ავაგოთ ეილერის ციკლი გრაფში (ნახ. 8.28.). გრაფი ბმულია და მისი ყველა წვერო ლუწია.

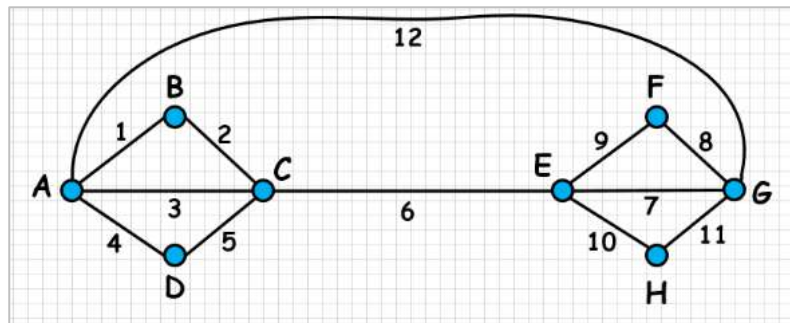


ნახ. 8.28. ბმულ გრაფში ეილერის ციკლის პოვნა ფლერის ალგორითმით

ნახატ 8.28.-ზე მოცემულ გრაფში მოძრაობა დავიწყეთ A წვეროდან:

1. მიმდინარე გზა A , მიმდინარე წიბო $\{A, B\}$
2. მიმდინარე გზა A, B მიმდინარე წიბო $\{B, C\}$
3. მიმდინარე გზა A, B, C მიმდინარე წიბო $\{C, A\}$
4. მიმდინარე გზა A, B, C, A მიმდინარე წიბო $\{A, D\}$
5. მიმდინარე გზა A, B, C, A, D მიმდინარე წიბო $\{D, C\}$
6. მიმდინარე გზა A, B, C, A, D, C მიმდინარე წიბო $\{C, E\}$
7. მიმდინარე გზა A, B, C, A, D, C, E მიმდინარე წიბო $\{E, G\}$
8. მიმდინარე გზა A, B, C, A, D, C, E, G მიმდინარე წიბო $\{G, F\}$
9. მიმდინარე გზა $A, B, C, A, D, C, E, G, F$ მიმდინარე წიბო $\{F, E\}$
10. მიმდინარე გზა $A, B, C, A, D, C, E, G, F, E$ მიმდინარე წიბო $\{E, H\}$
11. მიმდინარე გზა $A, B, C, A, D, C, E, G, F, E, H$ მიმდინარე წიბო $\{H, G\}$
12. მიმდინარე გზა $A, B, C, A, D, C, E, G, F, E, H, G$ მიმდინარე წიბო $\{G, A\}$
13. მიმდინარე გზა $A, B, C, A, D, C, E, G, F, E, H, G, A$

ახლა დავნომროთ მოცემული გზის შესაბამისი წიბოები. გრაფს ექნება ნახატ 8.29.-ზე მოცემული სახე:



ნახ. 8.29. ფლერის ალგორითმით ნაჩვენები ეილერის ციკლი

ახლა კი გადავიდეთ კომპიუტერის ამოცანის განხილვაზე.

დღესდღეობით ერთ-ერთი ცნობილი ამოცანა ჰამილტონის ციკლის შესახებ არის კომპიუტერის ამოცანა. კომპიუტერის ამოცანას 200 წელზე მეტი ხნის ისტორია აქვს.

კომპიუტერის ამოცანის ამოხსნის ზუსტი მეთოდები პირობითად შეიძლება შემდეგ ჯგუფებად დავყოთ: დინამიკური დაპროგრამება, შტოებისა და საზღვრების მეთოდი, ლაგრანჟის მამრავლთა მეთოდი, მკვეთი სიბრტყეების მეთოდი და მრავალი სხვ.

ზოგადად კომპიუტერის ამოცანა შეიძლება ჩამოყალიბდეს როგორც გრაფში უმოკლესი სიგრძის ჰამილტონის ციკლის მოძებნის ამოცანა.

კომპიუტერის ამოცანა ამოხსნადია, მაშინ და მხოლოდ მაშინ, როცა ამ ამოცანის შესაბამისი გრაფი ჰამილტონის გრაფს წარმოადგენს.

კომპიუტერმა, მოხეტიალე ვაჭარმა, უნდა შემოიაროს n რაოდენობის ქალაქი. n რაოდენობის ქალაქი დანომრილია $1, 2, \dots, n$ სახით. თითოეულ წყვილ ქალაქს შორის მანძილი

მოცემულია $c[i, j]$ სახით. კომივოიაჟერმა მოძრაობა უნდა დაიწყოს 1-ლი ქალაქიდან, უნდა შევიდეს ყველა დანარჩენ ქალაქში მხოლოდ ერთხელ და დაბრუნდეს ისევ 1-ელ ქალაქში. კომივოიაჟერმა უნდა დაადგინოს თუ როგორი თანამიმდევრობით უნდა შემოიაროს ქალაქები, რომ გავლილი მანძილის ჯამური შედეგი იყოს ყველაზე მინიმალური. (კომივოიაჟერმა ეს ისე უნდა გააკეთოს, რომ მთელი მოგზაურობის ჯამური დანახარჯები იყოს მინიმალური. დანახარჯებში შეიძლება ვიგულისხმოთ ან თანხა, ან დრო, ან ჯამური გავლილი მანძილი).

მოკლედ რომ ვთქვათ, კომივოიაჟერის მარშრუტი უნდა იყოს ოპტიმალური.

დავუშვათ, შემოვლილი ქალაქების თანამიმდევრობა აღვწეროთ $1, p_1, p_2, \dots, p_{n-1}, 1$ ანუ p_i არის i -ურ ბიჯზე მონახულებული ქალაქის ნომერი, ამდენად $i = 0, \dots, n, p_0 = p_n = 1$ გავლილი მანძილი ტოლია:

$$\sum_{j=0}^{n-1} c[p_j, p_{j+1}] \rightarrow \min$$

$p_1, p_2, \dots, p_{n-1}$ რიცხვებს შორის ზუსტად ერთხელ გვხვდება თითო რიცხვი $2..n$ ინტერვალიდან. ამდენად უნდა მოიძებნოს მთელი რიცხვების გადანაცვლება 2-დან n -მდე.

ვთქვათ, მოცემული გვაქვს $n = 5$ შემოვლილი ქალაქების მატრიცა და ვიცით ასევე მათ შორის მანძილებით:

$$C = \begin{pmatrix} 0 & 5 & 3 & 1 & 2 \\ 5 & 0 & 1 & 3 & 6 \\ 3 & 1 & 0 & 4 & 2 \\ 1 & 3 & 4 & 0 & 7 \\ 2 & 6 & 2 & 7 & 0 \end{pmatrix}$$

უნდა დავადგინოთ ქალაქების ისეთი სახით გადარჩევა/გადანაცვლება, რომელთა შორის გავლილი მანძილი აღმოჩნდება მინიმალური. პირველი სამეზბნო ელემენტი ყოველთვის 1-ის ტოლია. ამიტომ სრული გადარჩევის დროს საჭირო იქნება დარჩენილი $n - 1$ ქალაქის გადარჩევა. სულ ასეთი გადანაცვლება იქნება $(n - 1)!$ ცალი.

$n = 5$ შემთხვევაში მივიღებთ $4! = 24$ სხვადასხვა გადაადგილებას, ხოლო თუ მატრიცა სიმეტრიულია, მაშინ გვექნება 12 გადანაცვლება.

ყველა შესაძლებელი მარშრუტი მოცემულია შემდეგი სახით:

მარშრუტი: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$ მანძილი: 19

მარშრუტი: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$ მანძილი: 16

მარშრუტი: $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 1$ მანძილი: 16

მარშრუტი: $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 1$ მანძილი: 20

მარშრუტი: $1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 1$ მანძილი: 18

მარშრუტი: $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 1$ მანძილი: 25

მარშრუტი: $1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 1$ მანძილი: 16

მარშრუტი: $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1$ მანძილი: 18

მარშრუტი: $1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 1$ მანძილი: 18

მარშრუტი: $1 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 4 \rightarrow 1$ მანძილი: 15

მარშრუტი: $1 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 1$ მანძილი: 9

მარშრუტი: $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 1$ მანძილი: 14

კომივოიაჟერის ამოცანა ადვილად ამოხსნადია n -ის მცირე მნიშვნელობებისთვის, მაგრამ n -ის დიდი მნიშვნელობებისთვის ეს კომივოიაჟერის ამოცანის ალგორითმი უძლურია, რადგან ის ზედმეტად დიდხანს მუშაობს. მაგალითად, $n = 100$ -ის შემთხვევაში ამოცანის ამოსახსნელად დაგჭირდება მინიმუმ 10^{21} ოპერაცია. თანამედროვე კომპიუტერების გამოყენებით ასეთი ამოცანის გადაწყვეტას დასჭირდება დაახლოებით 30 წელი.

დღესდღეობით კომივოიაჟერის ამოცანის პოლინომიალური გადაწყვეტა არ არსებობს.

კომპიუტერულ მეცნიერებაში ერთ-ერთ მნიშვნელოვან გადაუჭრელ პრობლემას წარმოადგენს ჰამილტონის ციკლების და წრედების არსებობის პირობების დადგენა. ეს შეიძლება იმით იყოს გამოწვეული, რომ ამ ამოცანისთვის ჯერჯერობით ვერავინ მოიფიქრა სწრაფი ალგორითმი, ან იმით, რომ ასეთი ალგორითმი არ არსებობს, მაგრამ რადგან ჰამილტონის ციკლის ამოცანა უადრესად მნიშვნელოვანია, ეს ცენტრალური ღია საკითხია კომპიუტერულ მეცნიერებაში.

დეიქსტრას ალგორითმი. გრაფებთან მიმართებით ძალიან ხშირად საჭიროა მოიძებნოს უმოკლესი გზის სიგრძე გრაფის წვეროებს შორის. გრაფში მოცემული წვეროდან ნებისმიერ სხვა წვერომდე უმოკლესი გზის მოსაძებნად გამოიყენება ერთ-ერთი ყველაზე ცნობილი და ფუნქციური ალგორითმი - დეიქსტრას ალგორითმი.

დეიქსტრას ალგორითმი მუშაობს მხოლოდ დადებითი მნიშვნელობის მქონე წონის წიბოებზე. დეიქსტრას ალგორითმი ძალიან ჰგავს BFS-ს.

დეიქსტრას ალგორითმის მუშაობის სირთულეა: $O(n^2)$

დეიქსტრას ალგორითმის მიხედვით უმოკლესი გზა განისაზღვრება შემდეგი სახით:

if $d[u] > d[v] + c[v, u]$ then

$d[u] = d[v] + c[v, u]$

$p[u] = (p[v], u)$

სადაც u - არის გრაფის მონახულებული წვერო;

v - გრაფის მიმდინარე წვერო;

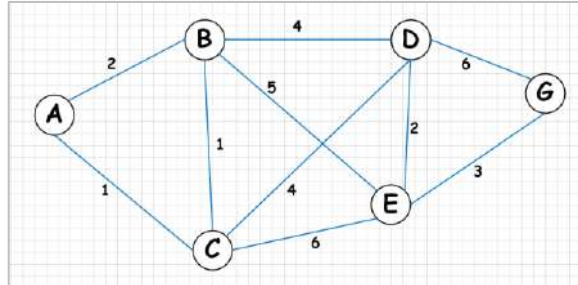
$d[u]$ - უმოკლესი გზის სიგრძე v წვერომდე, მინიმალური მანძილი;

$p[u]$ - მონახულებული წვეროების სია;

$c[i, j]$ - გრაფში ყველა წიბო;

დეიქსტრას ალგორითმი გამოითვლის უმოკლეს მანძილს საწყისი წვეროდან ყველა დანარჩენ წვეროებამდე. ალგორითმი იდეალურია ისეთ შემთხვევაშიც, როცა ვიცით მანძილი ყველა წვეროებამდე და არა მხოლოდ ბოლო წვერომდე.

მაგალითი 8.5. მოცემული გვაქვს გრაფი (იხ. ნახ. 8.30). უნდა ვიპოვოთ უმოკლესი გზა A წვეროდან G წვერომდე დეიქსტრას ალგორითმის გამოყენებით.

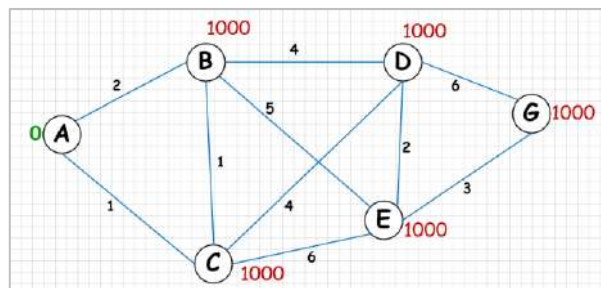


ნახ. 8.30. შეწონილი არაორიენტირებული გრაფი

ამოხსნა:

თავდაპირველად შევასრულოთ გრაფის ინიციალიზაცია.

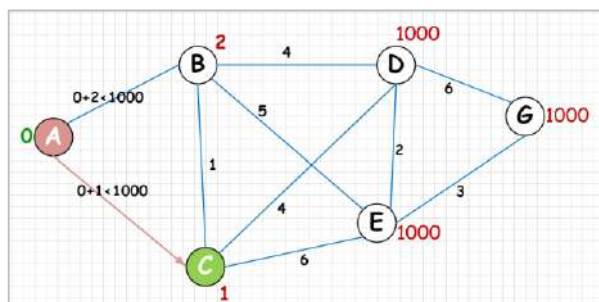
გრაფის საწყის წვეროს A-ს ვანიჭებთ ნულის ტოლ მნიშვნელობას $A = 0$, დანარჩენ წვეროებს ვანიჭებთ საკმაოდ დიდ რიცხვით მნიშვნელობას: ან 1000-ს ან უსასრულობას ∞ . ჩვენს შემთხვევაში დავტოვოთ 1000 (ნახ. 8.31.).



ნახ. 8.31. გრაფის ინიციალიზაცია

პირველი ბიჯი:

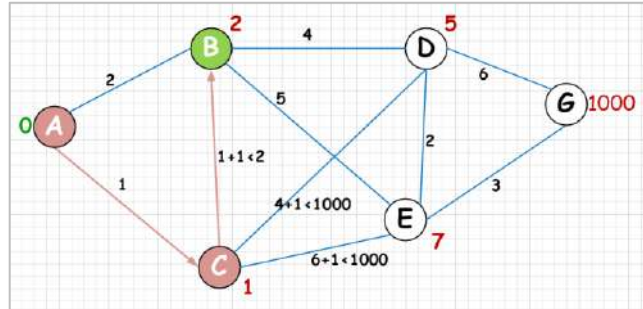
A წვეროს მეზობელი წვეროებია C და B წვეროები. თავდაპირველად მოვინახულოთ მინიმალური წონის წვერო C და მოვნიშნოთ, როგორც მიმდინარე წვერო, შემდეგ კლებადობის მიხედვით დარჩენილი B წვერო. წვერო A მოვნიშნოთ, როგორც მონახულებულ წვერო ანუ მას აღარ განვიხილავთ (ნახ. 8.32.).



ნახ. 8.32. დეიქსტრას ალგორითმის მუშაობის პირველი ბიჯი

მეორე ბიჯი:

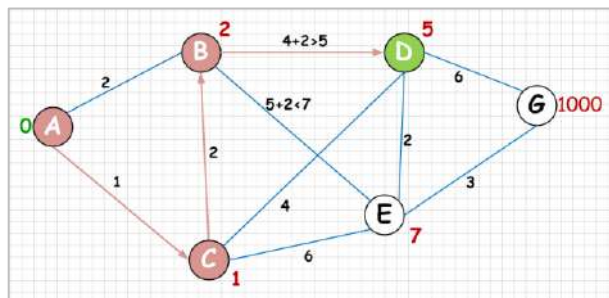
C წვეროს მიმდინარე წვეროებია B, D და E წვეროები. პირველად გადავიდეთ მინიმალური წონის მქონე წვეროზე - B წვეროზე და მოვნიშნოთ იგი როგორც მიმდინარე წვერო, შემდეგ გადავიდეთ D და E წვეროების მონახულებაზე. წვერო C მოვნიშნოთ, როგორც მონახულებულ წვერო (ნახ. 8.33.).



ნახ. 8.33. დეიქსტრას ალგორითმის მუშაობის მეორე ბიჯი

მესამე ბიჯი:

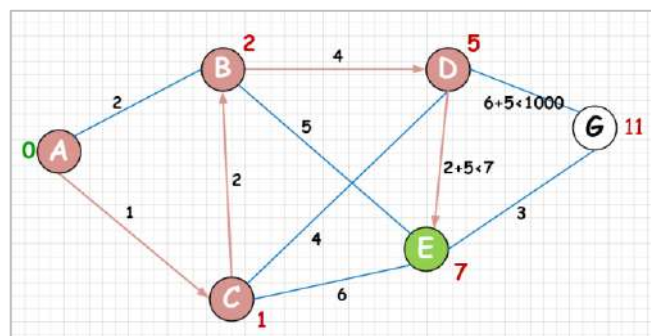
B წვეროს მეზობელი წვეროებია D და E წვეროები. მოვინახულოთ წვერო უმოკლესი წონით - წვერო D და აღვნიშნავთ D წვერო, როგორც მიმდინარე წვერო, შემდეგ მოვინახულოთ E წვერო. წვერო B მოვნიშნოთ, როგორც მონახულებულ წვერო ანუ მას აღარ განვიხილავთ (ნახ. 8.34.).



ნახ. 8.34. დეიქსტრას ალგორითმის მუშაობის მესამე ბიჯი

მეოთხე ბიჯი:

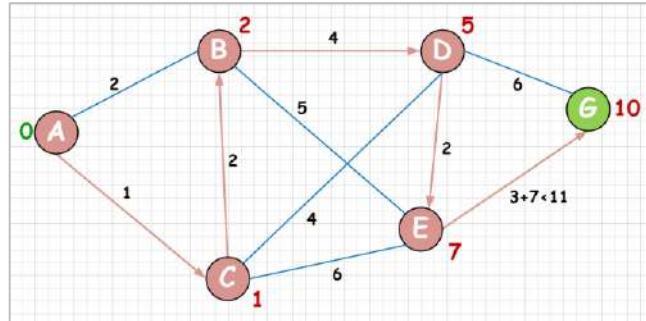
D წვეროს მეზობელი წვეროებია G და E წვეროები. მოვინახულოთ უმოკლესი გზის წვერო E წვერო და მოვნიშნავთ E როგორც მიმდინარე, გადავდივართ G წვეროზე. წვერო D მოვნიშნოთ, როგორც მონახულებულ წვერო ანუ მას აღარ განვიხილავთ (ნახ. 8.35.).



ნახ. 8.35. დეიქსტრას ალგორითმის მუშაობის მეოთხე ბიჯი

მეხუთე ბიჯი:

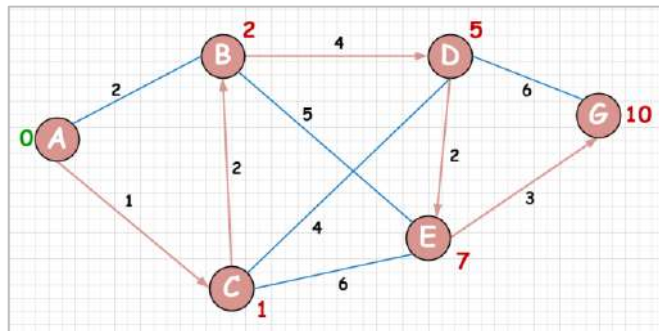
E წვეროს მეზობელი წვეროა ბოლო დარჩენილი G წვერო. გადავიდეთ G წვეროზე და მოვნიშნავთ როგორც მონახულებული წვერო ანუ მას აღარ განვიხილავთ. ამით დასრულდა ალგორითმის მუშაობა (ნახ. 8.36.).



ნახ. 8.36. დეიქსტრას ალგორითმის მუშაობის მეხუთე ბიჯი

პასუხი:

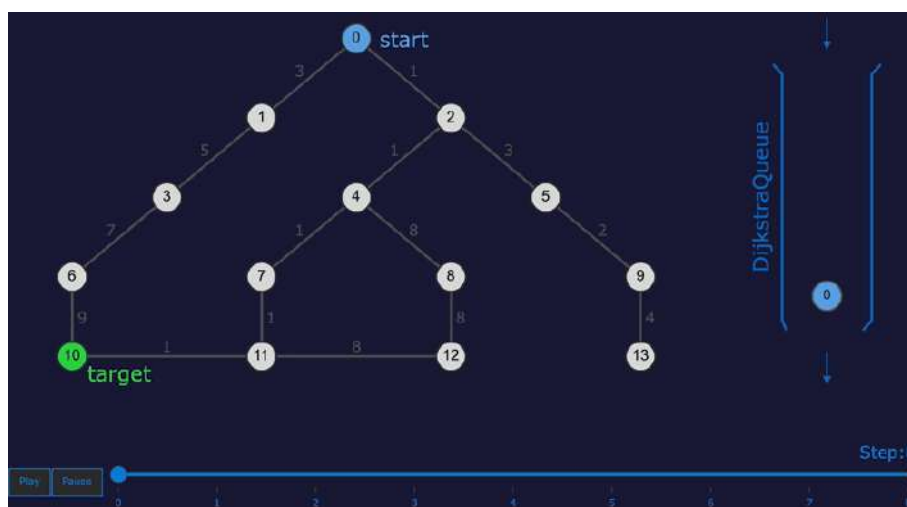
დეიქსტრას ალგორითმის მიხედვით ყველაზე მოკლე გზაა: $A C B D E G$ მარშრუტი და იგი 10-ის ტოლია (ნახ. 8.37.)



ნახ. 8.37. დეიქსტრას ალგორითმის მუშაობის დასასრული.

ყველაზე მოკლე გზაა: $A C B D E G = 10$

დეიქსტრას მუშაობის ვიზუალიზაცია ვიდეოს სახით (ვიდეო 1):



ვიდეო 1. უმოკლესი გზის მოძებნა დეიქსტრას ალგორითმით

ბელმან-ფორდის ალგორითმი. ბელმან-ფორდის ალგორითმი გამოიყენება უმოკლესი გზის მოსაძებნად ორიენტირებულ გრაფში საწყისი წვეროდან ყველა სხვა დანარჩენ წვერომდე.

დეიქსტრას ალგორითმისგან განსხვავებით, ბელმან-ფორდის ალგორითმი მუშაობს უარყოფითი წონის მქონე წიბოებთან.

ბელმან-ფორდის ალგორითმი მუშაობს დეიქსტრას ალგორითმის ანალოგიურად, მაგრამ მუშაობს უფრო ნელა, ვიდრე დეიქსტრას ალგორითმი.

დეიქსტრას ალგორითმისაგან განსხვავებით, ბელმან-ფორდის ალგორითმი აღიქვამს უარყოფითი წონის მქონე გრაფებს. ბელმან-ფორდის ალგორითმს შეუძლია აღმოაჩინოს უარყოფითი წონის ციკლები (ციკლის წიბოების ჯამი უარყოფითია), რაც ძალიან მნიშვნელოვანია. **ბელმან-ფორდის ალგორითმი უფრო მარტივია, ვიდრე დეიქსტრას ალგორითმი და გამოიყენება განაწილებული პარამეტრების მქონე სისტემებში.**

ბელმან-ფორდის ალგორითმი $O(n * m)$ დროში პოულობს ყველაზე მოკლე გზას გრაფის ერთი წვეროდან ყველა დანარჩენ წვერომდე უარყოფითი წონის მქონე წიბოების ჩათვლით.

ბელმან-ფორდის ალგორითმის ასიმპტოტური სირთულეა $O(|V| * |E|)$, სადაც $|V|$ და $|E|$ შესაბამისად არის გრაფში წვეროებისა და გვერდების რაოდენობა.

ალგორითმი საშუალებას იძლევა, ძალიან მარტივად განსაზღვროს, არსებობს თუ არა G გრაფში უარყოფითი ციკლი, რომელიც მიღწევადია s წვეროდან.

უარყოფითი ციკლის შესამოწმებლად საჭიროა ციკლის შიგა იტერაცია გავიმეოროთ $|V|$ -ჯერ. თუ ბოლო იტერაციის შესრულებისას გრაფის რომელიმე წვერომდე უმოკლესი გზის სიგრძე საკმაოდ შემცირდა, მაშინ გრაფში არსებობს უარყოფითი ციკლი.

ბელმან-ფორდის ალგორითმის ფსევდოკოდი:

```
procedure BellmanFord(list vertices, list edges, vertex source)
```

```
// This implementation takes in a graph, represented as lists of vertices and edges, and modifies the
```

```
// vertices so that their distance and predecessor attributes store the shortest paths.
```

```
// Step 1: Initialize graph
```

```
for each vertex  $v$  in vertices:
```

```
    if  $v$  is source then  $v.distance := 0$ 
```

```
    else  $v.distance := \text{infinity}$ 
```

```
     $v.predecessor := \text{null}$ 
```

```
// Step 2: relax edges repeatedly
```

```
for  $i$  from 1 to  $\text{size}(\text{vertices})-1$ :
```

```
    for each edge  $uv$  in edges:
```

```
         $u := uv.source$ 
```

```

v := uv.destination
// uv is the edge from u to v
if v.distance > u.distance + uv.weight:
    v.distance := u.distance + uv.weight
    v.predecessor := u
// Step 3: check for negative-weight cycles
for each edge uv in edges:
    u := uv.source
    v := uv.destination
    if v.distance > u.distance + uv.weight:
        error "Graph contains a negative-weight cycle"

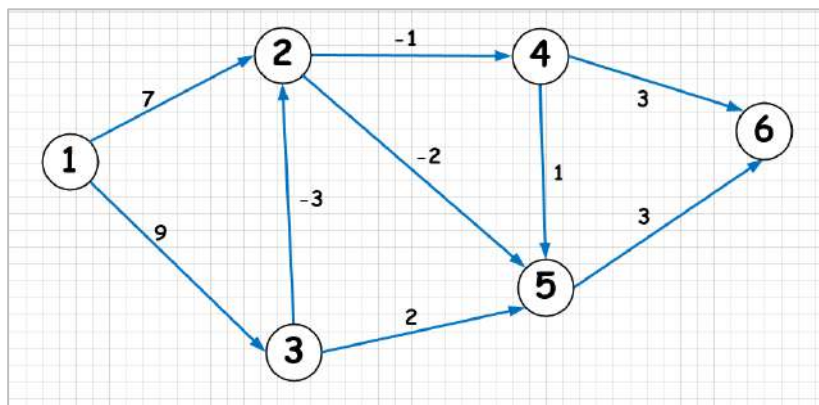
```

ბელმან-ფორდის ალგორითმის ფსევდოკოდის შემოკლებული ვარიანტი:

Bellman_Ford(G, w, s)

1. **Initialize_Single_Source(G, s)**
2. **for** $i = 1$ **to** $|G.V| - 1$
3. **for** თითოეული წიბოსთვის $(u, v) \in G.E$:
4. **RELAX**(u, v, w)
5. **for** თითოეული წიბოსთვის $(u, v) \in G.E$:
6. **if** $v.d > u.d + w(u, v)$
7. **return FALSE**
8. **return TRUE**

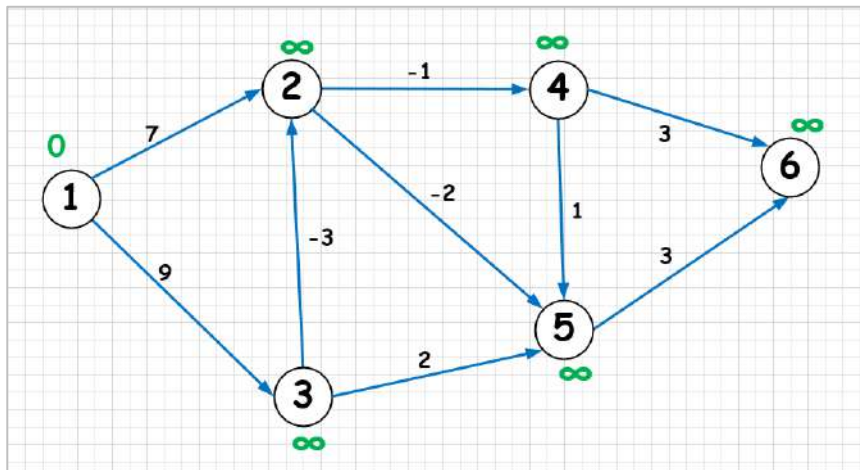
მაგალითი 8.6. განვიხილოთ ორიენტირებული გრაფი უარყოფითი წონის მქონე წიბოებით (ნახ. 8.38.).



ნახ. 8.38. ორიენტირებული გრაფი უარყოფითი წონის წიბოებით მოვძებნოთ უმოკლესი გზა 1-ლი წვეროდან მე-6 წვერომდე.

ამოხსნა:

თავიდან უნდა შევასრულოთ გრაფის ინიციალიზება, ამისთვის გრაფის საწყის წვეროს მივანიჭოთ ნულის ტოლი მნიშვნელობა, დანარჩენ წვეროებს კი - უსასრულოება ∞ (ნახ. 8.39.).



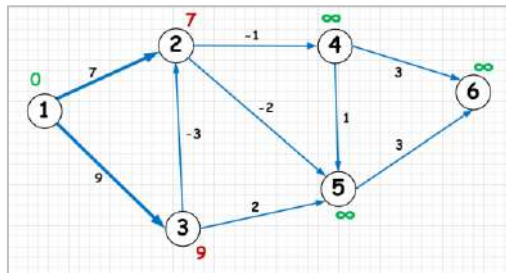
ნახ. 8.39. ორიენტებული გრაფის ინიციალიზება

ყველა გამომავალი წიბო ჩაწეროთ რიგითობის თანმიმდევრობით:

(1,2) (1,3) (2,4) (2,5) (3,2) (3,5) (4,5) (4,6) (5,6)

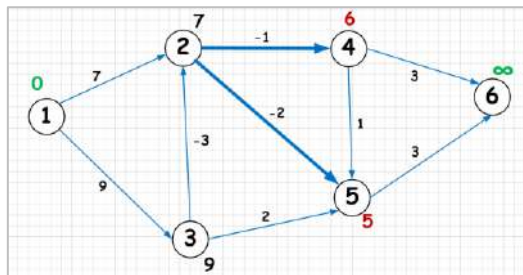
პირველი იტერაცია (ნახ. 8.43, 8.44, 8.45, 8.46, 8.47).

პირველი ბიჯი:



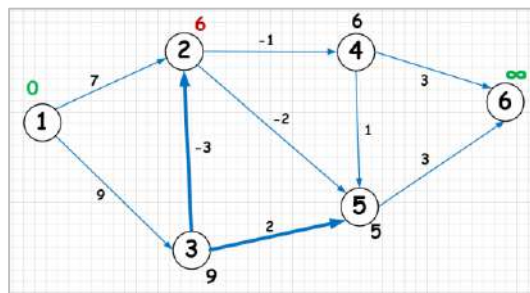
ნახ. 8.43. პირველი იტერაცია. პირველი ბიჯი

მეორე ბიჯი:



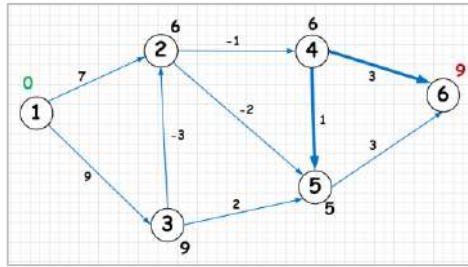
ნახ. 8.44. პირველი იტერაცია. მეორე ბიჯი

მესამე ბიჯი:



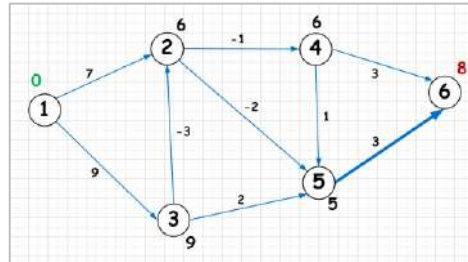
ნახ. 8.45. პირველი იტერაცია. მესამე ბიჯი

მეოთხე ბიჯი:



ნახ. 8.46. პირველი იტერაცია. მეოთხე ბიჯი

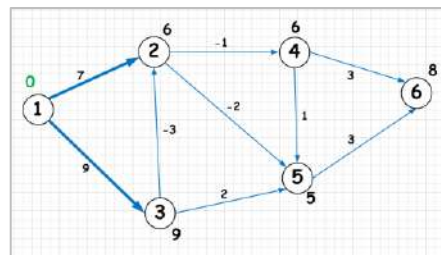
მეხუთე ბიჯი:



ნახ. 8.47. პირველი იტერაცია. მეხუთე ბიჯი

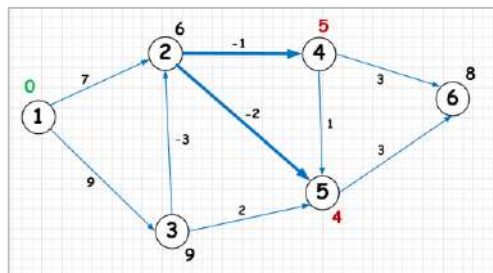
მეორე იტერაცია (ნახ. 8.48, 8.49, 8.50, 8.51, 8.52).

პირველი ბიჯი:



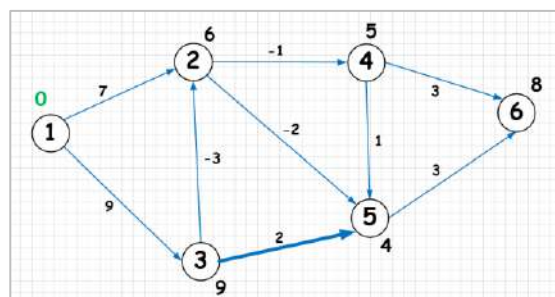
ნახ. 8.48. მეორე იტერაცია. პირველი ბიჯი

მეორე ბიჯი:



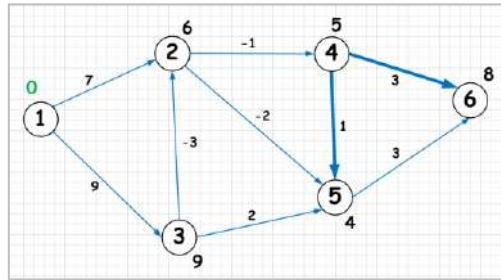
ნახ. 8.49. მეორე იტერაცია. მეორე ბიჯი

მესამე ბიჯი:



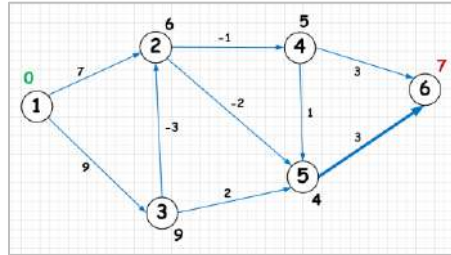
ნახ. 8.50. მეორე იტერაცია. მესამე ბიჯი

მეოთხე ბიჯი:



ნახ. 8.51. მეორე იტერაცია. მეოთხე ბიჯი

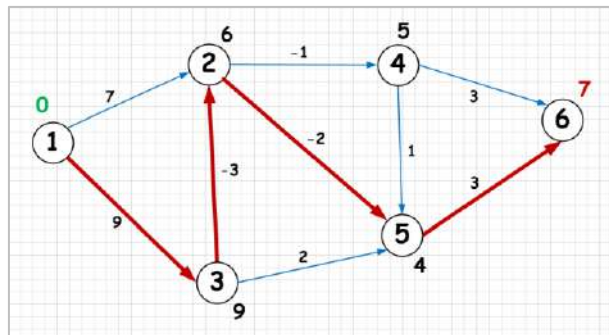
მეხუთე ბიჯი:



ნახ. 8.52. მეორე იტერაცია. მეხუთე ბიჯი

იგივე გავიმეოროთ მესამე იტერაციისთვის. შედეგად, მანძილის გაანგარიშებები დარჩება იგივე, ამიტომ ბელმან-ფორდის ალგორითმის მეშვეობით უმოკლესი გზა არის შემდეგი თანამიმდევრობა: $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 6$ და იგი ტოლია 7-ის.

პასუხი: ბელმან-ფორდის ალგორითმის მიხედვით დასმული ამოცანა გადაწყდა ნახატ 8.53-ზე ნაჩვენები გზის (წითელი ფერის) თანამიმდევრობით: $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 6 = 7$



ნახ. 8.53. ბელმან-ფორდის მიხედვით მიღებული უმოკლესი გზა: $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 6$

ფლოიდ უორცელის ალგორითმი. ფლოიდ უორცელის ალგორითმი გამოიყენება შეწონილი გრაფის ყველა წვეთს წვეთის შორის უმოკლესი გზების მოსაძებნად. ეს ალგორითმი ძალზე ეფექტურია და მას შეუძლია იმუშაოს როგორც დადებითი, ისე უარყოფითი წონის მქონე წიბოების მქონე გრაფებში, მაგრამ ფლოიდ უორცელის ალგორითმი არ მუშაობს უარყოფითი ციკლის გრაფებისთვის ანუ როცა ციკლის წვეთების ჯამი უარყოფითია. ფლოიდ უორცელის ალგორითმის გამოთვლითი სირთულე:

- დროითი სირთულე $O(V^3)$;
- სივრცითი სირთულე $O(V^2)$;

სადაც V არის გრაფის წვეთების სიმრავლე.

ფლოიდ უორშელის ალგორითმის ფსევდოკოდი:

ფუნქცია FloydWarshell(graph)

i -> vertexes

dists[i][i] <- 0

j -> vertexes

თუ graph[i][j] არ არის ∞

dists[i][j] <- graph[i][j]

k -> vertexes

i -> vertexes

j -> vertexes

dist[i][j] <- min(dist[i][j], dist[i][k] + dist[k][j])

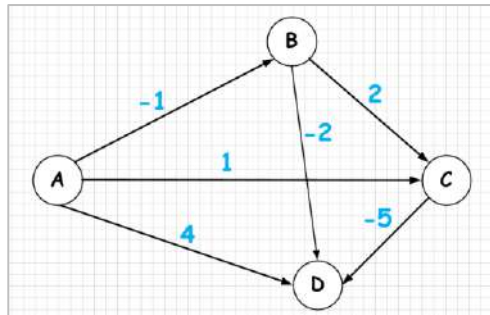
return dist

graph - მოსაზღვრეობის მატრიცა, სადავ ინახება გრაფი;

vertexes - გრაფის წვეროების სია;

dist - მანძილი მატრიცაში, რომელიც შევსებულია უსასრულობით.

მაგალითი 8.7. განვიხილოთ ფლოიდ უორშელის ალგორითმის მუშაობა. მოცემულია გრაფი (ნახ. 8.54.).



ნახ. 8.54. საწყისი გრაფი

თავდაპირველად საჭიროა ყველა წვეთს წვეროს შორის მანძილების ინიციალიზაცია: წვეროებს შორის მანძილი აღვნიშნოთ უსასრულობით ∞ , ხოლო მანძილი i -ური წვეროდან თავისთავან აღვნიშნოთ 0-ით (ცხრილი 8.1.). ასევე დავადგინოთ წვეროებს შორის მანძილის წონები მატრიცის სახით (ცხრილი 8.2.)

ცხრილი 8.1. გრაფის საწყისი მონაცემების ინიციალიზება

	A	B	C	D
A	0	∞	∞	∞
B	∞	0	∞	∞
C	∞	∞	0	∞
D	∞	∞	∞	0

ცხრილი 8.2. წვეროებს შორის დამაკავშირებელი მანძილთა წონა

	A	B	C	D
A	0	-1	1	4
B	∞	0	2	-2
C	∞	∞	0	-5
D	∞	∞	∞	0

ფლოიდ უორშელის ალგორითმი საშუალებას გვაძლევს, ვიპოვოთ უმოკლესი გზა შეწონილ გრაფში, მხოლოდ გრაფს არ უნდა ჰქონდეს უარყოფითი ციკლი.

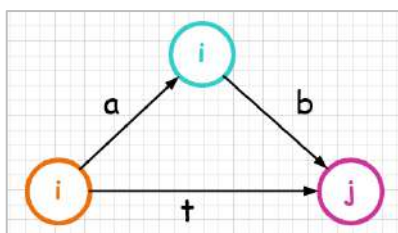
ახლა (i, j) წვეროების წყვილისათვის აუცილებელია ჩამოვთვალოთ უმოკლესი მანძილები მესამე k წვეროს დახმარებით:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$$

სადაც $\text{dist}[X][Y]$ არის X წვეროდან Y წვერომდე მანძილი; i, j, k არის გრაფის ნებისმიერი წვეროები; i, j და k შეიძლება ერთმანეთს დაემთხვეს.

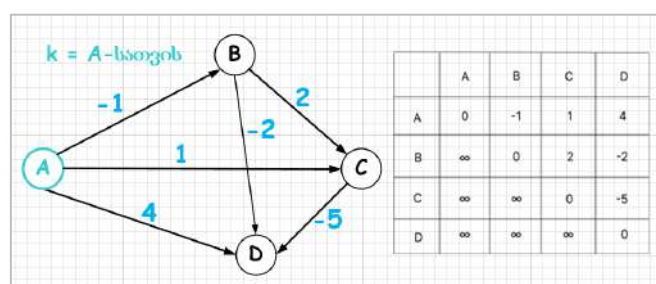
უმოკლესი გზა i წვეროდან j წვერომდე ტოლია: $\min(t, a+b)$.

უმოკლესი გზა ვიზუალურად ნაჩვენებია ნახატ 8.55.-ზე.

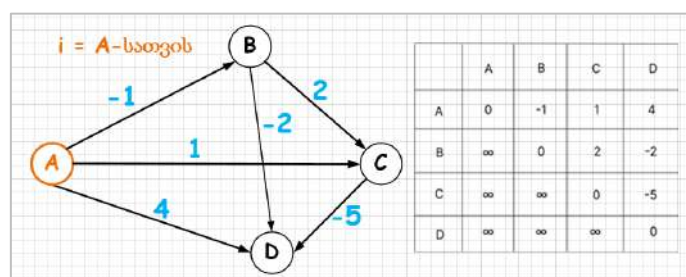


ნახ. 8.55. უმოკლესი გზა

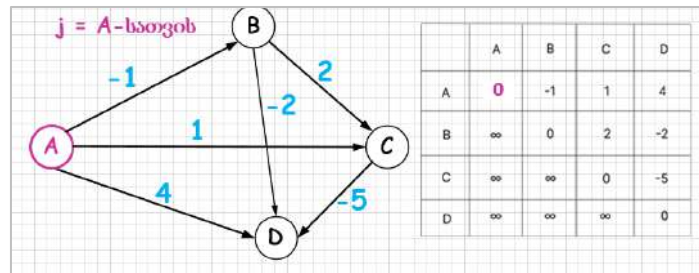
1. $k=A, i=A$ და $j=A$ შემთხვევის შესაბამისი გრაფები და მატრიცა (ნახ. 5.56., 5.57., 5.58.).



ნახ. 8.56. $k=A$ შემთხვევა და შესაბამისი მატრიცა



ნახ. 8.57. $i=A$ შემთხვევა და შესაბამისი მატრიცა

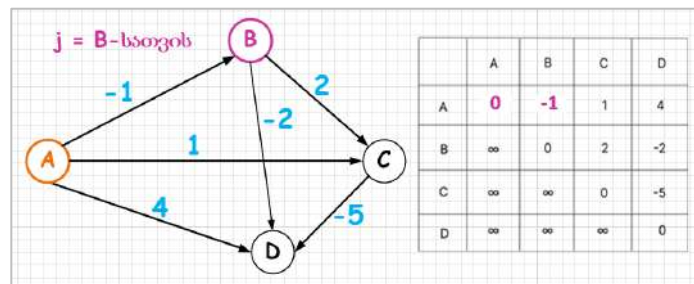


ნახ. 8.58. $j=A$ შემთხვევა და შესაბამისი მატრიცა

A წვეროდან A წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[A][A], \text{dist}[A][A] + \text{dist}[A][A]) = \min(0, 0+0) = 0$$

2. $j=B$ შემთხვევის შესაბამისი გრაფები და მატრიცა (ნახ. 8.59.).

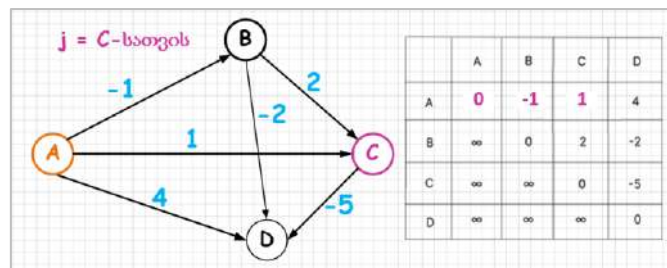


ნახ. 8.59. $j=B$ შემთხვევა და შესაბამისი მატრიცა

A წვეროდან B წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[A][B], \text{dist}[A][A] + \text{dist}[A][B]) = \min(-1, 0+(-1)) = -1$$

3. A წვეროდან C წვერომდე მანძილი (ნახ. 8.60.):

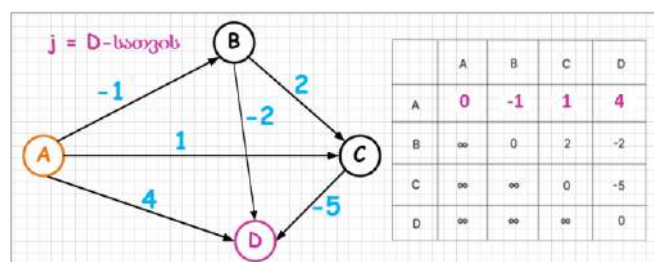


ნახ. 8.60. $j=C$ შემთხვევა და შესაბამისი მატრიცა

A წვეროდან C წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[A][C], \text{dist}[A][A] + \text{dist}[A][C]) = \min(1, 0+1) = 1$$

4. A წვეროდან D წვერომდე მანძილი (ნახ. 8.61.):

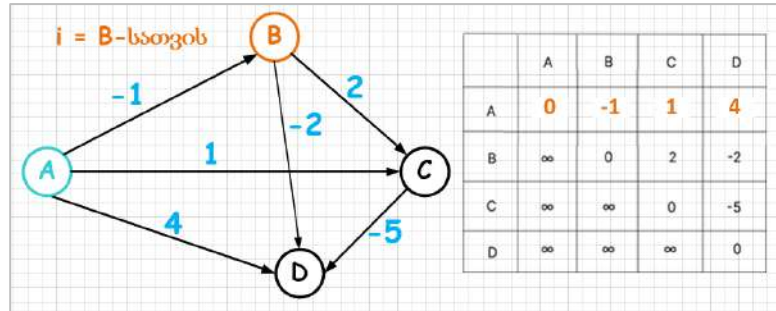


ნახ. 8.61. $j=D$ შემთხვევა და შესაბამისი მატრიცა

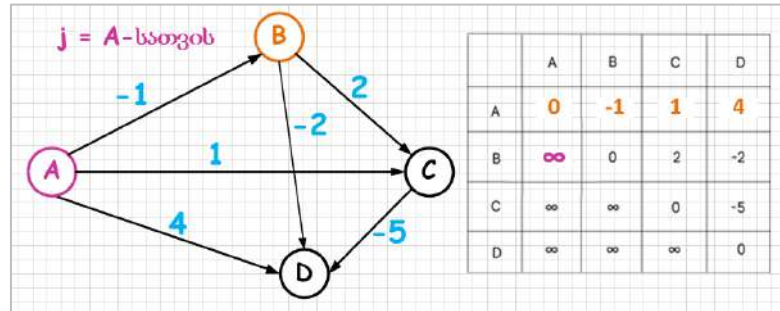
A წვეროდან D წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[A][D], \text{dist}[A][A] + \text{dist}[A][D]) = \min(4, 0+4) = 4$$

5. $i=B$ და $j=A$ შემთხვევის შესაბამისი გრაფები და მატრიცა (ნახ. 8.62., 8.63.).



ნახ. 8.62. $i=B$ შემთხვევა და შესაბამისი მატრიცა

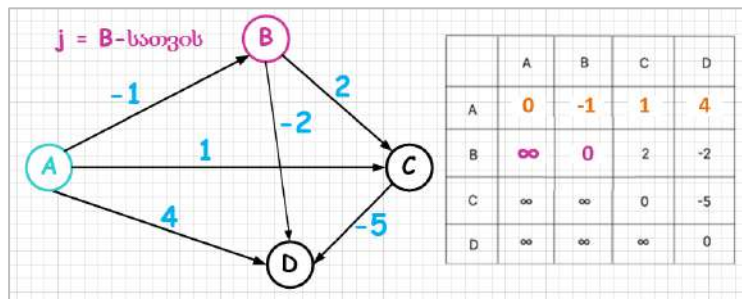


ნახ. 8.63. $j=A$ შემთხვევა და შესაბამისი მატრიცა

B წვეროდან A წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[B][A], \text{dist}[B][A] + \text{dist}[A][A]) = \min(\infty, \infty + 0) = \infty$$

6. $j=B$ შემთხვევის შესაბამისი გრაფები და მატრიცა (ნახ. 8.64.).

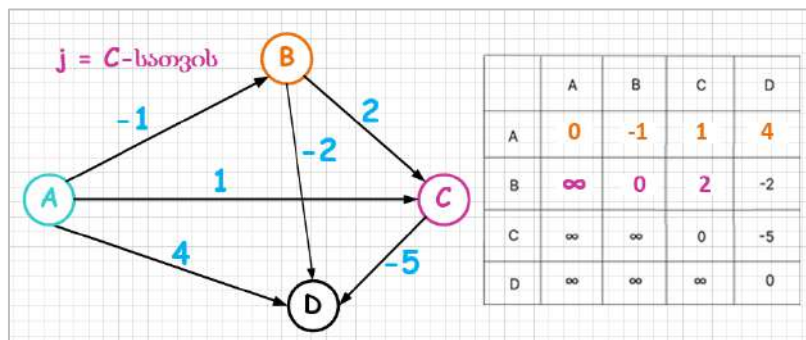


ნახ. 8.64. $j=B$ შემთხვევა და შესაბამისი მატრიცა

B წვეროდან B წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[B][B], \text{dist}[B][A] + \text{dist}[A][B]) = \min(0, \infty + -1) = 0$$

7. B წვეროდან C წვერომდე მანძილი (ნახ. 8.65.):

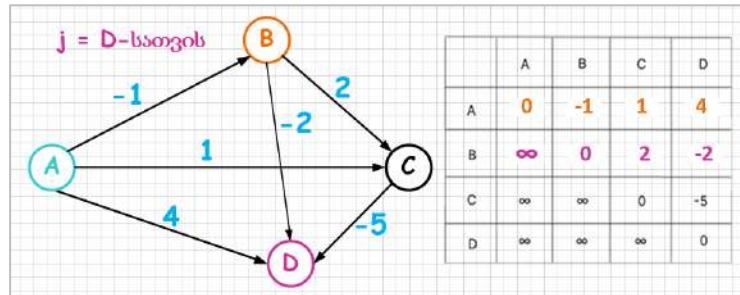


ნახ. 8.65. $j=C$ შემთხვევა და შესაბამისი მატრიცა

B წვეროდან C წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[B][C], \text{dist}[C][A] + \text{dist}[A][B]) = \min(2, \infty + -1) = 2$$

8. B წვეროდან D წვერომდე მანძილი (ნახ. 8.66.):

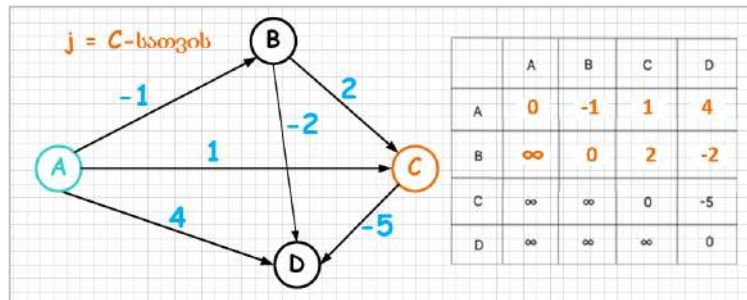


ნახ. 8.66. j=D შემთხვევა და შესაბამისი მატრიცა

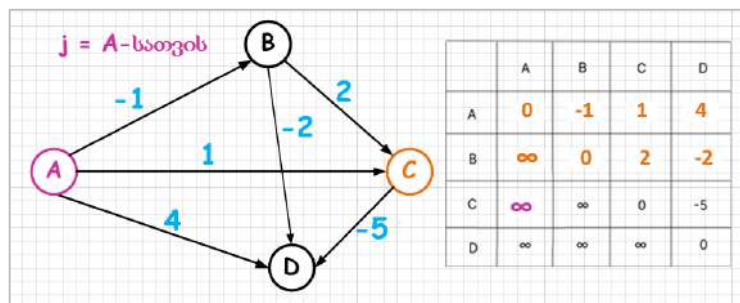
B წვეროდან D წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[B][D], \text{dist}[D][A] + \text{dist}[A][B]) = \min(-2, \infty + -1) = -2$$

9. j=C და j=A შემთხვევის შესაბამისი გრაფები და მატრიცა (ნახ. 8.67., 8.68.):



ნახ. 8.67. j=C შემთხვევა და შესაბამისი მატრიცა

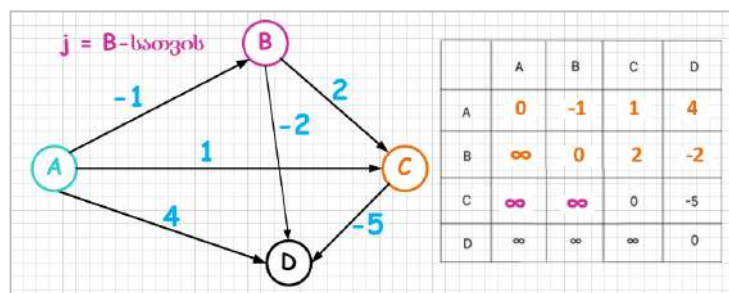


ნახ. 8.68. j=A შემთხვევა და შესაბამისი მატრიცა

C წვეროდან A წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[C][A], \text{dist}[C][A] + \text{dist}[A][A]) = \min(\infty, \infty + 0) = \infty$$

10. j=B შემთხვევის შესაბამისი გრაფები და მატრიცა (ნახ. 8.69.):

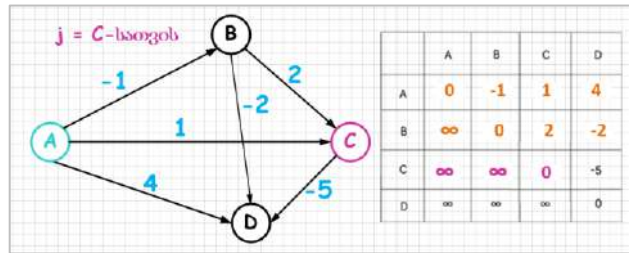


ნახ. 8.69. j=B შემთხვევა და შესაბამისი მატრიცა

C წვეროდან B წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[C][B], \text{dist}[C][A] + \text{dist}[A][B]) = \min(\infty, \infty + -1) = \infty$$

11. $j=C$ შემთხვევის შესაბამისი გრაფი და მატრიცა (ნახ. 8.70.):

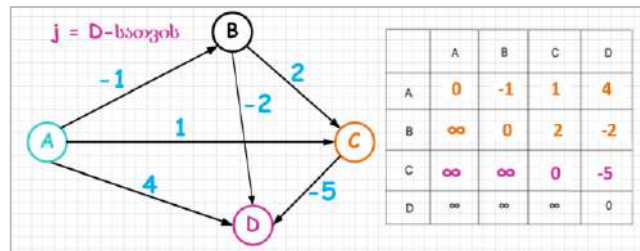


ნახ. 8.70. $j=C$ შემთხვევა და შესაბამისი მატრიცა

C წვეროდან C წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[C][C], \text{dist}[C][A] + \text{dist}[A][C]) = \min(0, \infty + 1) = 0$$

11. $j=D$ შემთხვევის შესაბამისი გრაფი და მატრიცა (ნახ. 8.71.):

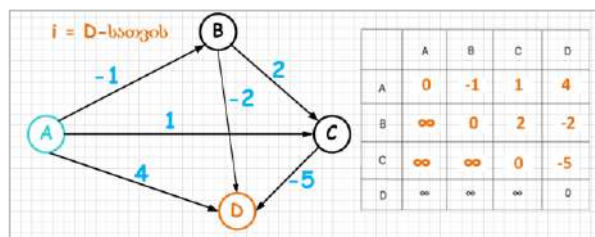


ნახ. 8.71. $j=D$ შემთხვევა და შესაბამისი მატრიცა

C წვეროდან D წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[C][D], \text{dist}[C][A] + \text{dist}[A][D]) = \min(-5, \infty + 4) = -5$$

12. $i=D$ და $j=A$ შემთხვევის შესაბამისი გრაფები და მატრიცა (ნახ. 8.72.):

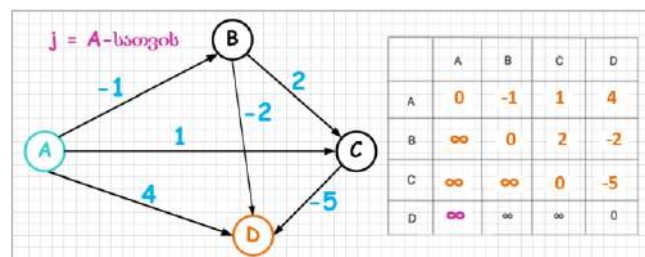


ნახ. 8.72. $i=D$ შემთხვევა და შესაბამისი მატრიცა

C წვეროდან D წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[C][D], \text{dist}[C][A] + \text{dist}[A][D]) = \min(-5, \infty + 4) = -5$$

13. $j=A$ შემთხვევის შესაბამისი გრაფი და მატრიცა (ნახ. 8.73.):



ნახ. 8.73. $j=A$ შემთხვევა და შესაბამისი მატრიცა

D წვეროდან A წვერომდე მანძილი:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]) = \min(\text{dist}[D][A], \text{dist}[D][A] + \text{dist}[A][A]) = \min(\infty, \infty + 0) = \infty$$

იტერაცია $k=A$ წვეროსთვის დასრულდა. შესაბამისი უმოკლესი გზის მატრიცა მოცემულია ცხრილი 8.3.-ზე.

ცხრილი 8.3. $k=A$ წვეროს იტერაციებით მიღებული მატრიცა

	A	B	C	D
A	0	-1	1	4
B	∞	0	2	-2
C	∞	∞	0	-5
D	∞	∞	∞	0

შემდეგ უნდა შევასრულოთ იგივე ქმედებები $k = B, C, D$ წვეროსთვის.

$k=B$ იტერაციის შესრულების შემდეგ შეიცვალა უმოკლესი მანძილები A წვეროდან D წვერომდე (ცხრილი 8.4.):

ცხრილი 8.4. $k=B$ წვეროს იტერაციებით მიღებული შეცვლილი მატრიცა

	A	B	C	D
A	0	-1	1	-3
B	∞	0	2	-2
C	∞	∞	0	-5
D	∞	∞	∞	0

$k=C$ იტერაციის შემთხვევაში შეიცვალა უმოკლესი მანძილი A წვეროდან D წვერომდე და B წვეროდან D წვერომდე (ცხრილი 8.5.):

ცხრილი 8.5. $k=C$ წვეროს იტერაციებით მიღებული შეცვლილი მატრიცა

	A	B	C	D
A	0	-1	1	-4
B	∞	0	2	-3
C	∞	∞	0	-5
D	∞	∞	∞	0

$k = D$ იტერაციის შემთხვევაში უმოკლესი გზის მატრიცა არ შეიცვალა (ცხრილი 8.6.):

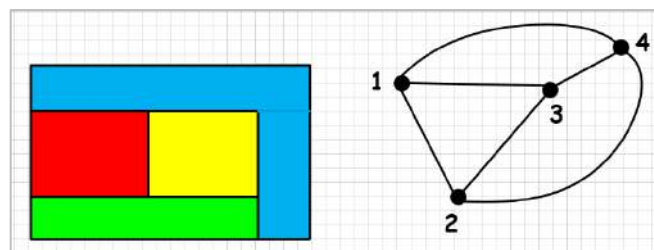
ცხრილი 8.6. $k = D$ იტერაციის შემთხვევაში უმოკლესი გზის მატრიცა დარჩა შეუცვლელი

	A	B	C	D
A	0	-1	1	-4
B	∞	0	2	-3
C	∞	∞	0	-5
D	∞	∞	∞	0

თუ აღმოჩნდება მოკლე გზის მატრიცაში მის მთავარ დიაგონალზე ნულზე ნაკლები ელემენტები, მაშინ ასეთ გრაფაში არსებობს უარყოფითი ციკლი.

ოთხი ფერის ამოცანა. ერთ-ერთი ყველაზე ცნობილი ამოცანა გრაფთა თეორიაში არის **ოთხი ფერის პრობლემა** - შესაძლებელია თუ არა სიბრტყეზე დახატული ნებისმიერი რუკის გაფერადება მაქსიმუმ ოთხი ფერით, ისე, რომ არც ერთი ორი მეზობელი გაფერადებული ქვეყნის ფერი არ უნდა დაემთხვეს ერთმანეთს. პრობლემა პირველად 1852 წელს წარმოჭრა მათემატიკოსმა ფრენსის გოსრიმ, რაზეც პასუხის გაცემას საუკუნეზე მეტი დასჭირდა. 1976 წელს აპელმა და ჰაკენმა ამოცანას დადებითი პასუხი გასცეს.

მინიმუმ რამდენი ფერია საჭირო ნებისმიერი რუკის ისე შესაღებად, რომ მეზობელ ქვეყნებს სხვადასხვა ფერი ჰქონდეთ? ამოცანის გადასაჭრელად სამი ფერი არ კმარა, მაგრამ 5 ფერი საკმარისია ნებისმიერი რუკისათვის. პრობლემად რჩებოდა სწორედ 4 სხვადასხვა ფერით გაფერადება. ეს პრობლემა გადაიჭრა მხოლოდ კომპიუტერების დახმარებით ჩატარებული ვრცელი გამოთვლებით. პრობლემა შეიძლება გრაფების ენაზე ითარგმნოს. ამისათვის ნებისმიერ რუკას შევუთანადოთ ასეთი გრაფი: წვეროები იყოს ქვეყნების დედაქალაქები, ორი წვერო (დედაქალაქი) შევაერთოთ წიბოთი (რკინიგზით), თუ ამ ქვეყნებს საერთო საზღვრის მონაკვეთი აქვთ, მაშინ რუკის შეღებვის პრობლემა დადის შემდეგ ამოცანაზე: მოცემული გრაფის წვეროებს მივანიჭოთ ნომრები ისე, რომ მეზობელ წვეროებს სხვადასხვა ნომერი ჰქონდეთ. მინიმუმ რამდენი ნომერი არის ამისათვის საკმარისი? ნახატ 8.74.-ზე მოცემულია რუკის გრაფი, მის გადანომრვას დაჭირდა 4 ფერი.



ნახ. 8.74. ოთხი ფერის შესაბამისი გრაფი

ამდენად მე-8 თავში აღვწერთ ხე-გრაფები და ბევრი ალგორითმი, რომელიც გრაფებში ეძებს უმოკლეს გზას.

ალგორითმები სტრიქონებზე. უგრძლესი საერთო ქვემიმდევრობა, ერთი სტრიქონის სხვა სტრიქონად გარდაქმნა. ქვესტრიქონების ძებნა. მატრიცები და მათზე მოქმედებები. მატრიცების მახასიათებლები, მატრიცების გადამრავლების შტრასენის ალგორითმი, მატრიცების შებრუნება.

უგრძლესი საერთო ქვემიმდევრობა. სიმბოლოების თანამიმდევრობა ქმნის სტრიქონს. სიმბოლო შესაძლებელია წარმოვადგინოთ, როგორც ცალკე ტიპი, მთელი რიცხვი ან ერთ სიმბოლოიანი სტრიქონი. სტრიქონი ან სტრიქონები გამოიყენება ტექსტის აღსაწერად.

სტრიქონის სიგრძე $|S|$ ტოლია მიმდევრობით არსებული სიმბოლოების რაოდენობისა.

$S[i..j]$ არის სტრიქონის უწყვეტი ქვესტრიქონი, რომელიც იწყება სტრიქონის i -ური პოზიციიდან და მთავრდება სტრიქონის j -ურ პოზიციაზე.

$S[0..i]$ - არის S სტრიქონის პრეფიქსი.

$S[i..|S| - 1]$ - არის S სტრიქონის სუფიქსი.

მაგალითად:

მოცემულია სტრიქონი **აბრაკადაბრა** - ამ სტრიქონის სიგრძეა 11;

ბრაკ - არის **აბრაკადაბრა** სტრიქონის ქვესტრიქონი;

აბრაკა - არის **აბრაკადაბრა** სტრიქონის პრეფიქსი;

კადაბრა - არის **აბრაკადაბრა** სტრიქონის სუფიქსი.

ძირითადი ოპერაციები სტრიქონებზე:

1. კონკატენაცია

ეს ოპერაცია საშუალებას გვძლევს, გააერთიანოთ ორი სტრიქონი ერთ სტრიქონად.

მაგალითად, გვაქვს ორი სტრიქონი: $A = ABC$ და $B = DEF$

მაშინ A -ს კონკატენაცია B -სთან იქნება სტრიქონი: $ABCDEF$

ხოლო B -ს კონკატენაცია A -სთან იქნება: $DEFABC$

2. სტრიქონის სიგრძე

ეს ოპერაცია გვამძლევს, საშუალებას დავადგინოთ სტრიქონში სიმბოლოების რაოდენობა.

3. ქვესტრიქონის მიღება

ქვესტრიქონი არის მოცემული სტრიქონის/ტექსტის ნაწილი, რომელიც შეიძლება მიღებულ იქნეს გარკვეული რაოდენობის სიმბოლოების (შესაძლოა ნულის) ამოშლით საწყისი სტრიქონის დასაწყისიდან ან ბოლოდან.

მოცემული გვაქვს სტრიქონი $S = ABCDE$.

სტრიქონი S -ის ქვესტრიქონი, რომელიც დაიწყება პირველი სიმბოლოდან და მისი სიგრძე იქნება 3, არის ქვესტრიქონი: ABC

სტრიქონი S -ის ქვესტრიქონი, რომელიც დაიწყება მეორე სიმბოლოდან და მისი სიგრძე იქნება 4, არის ქვესტრიქონი: $BCDE$

სტრიქონი S -ის ქვესტრიქონი, რომელიც დაიწყება მესამე სიმბოლოდან და მისი სიგრძე იქნება 1, არის ქვესტრიქონი: C

3. ქვესტრიქონის ძებნა

ქვესტრიქონის ძებნის ოპერაცია საშუალებას გაძლევთ მიიღოთ პირველივე X ქვესტრიქონის ინდექსი საწყის S სტრიქონში, თუ ის არსებობს. სტრიქონში სიმბოლოების ინდექსირება იწყება ნულიდან.

მაგალითად, მოცემული გვაქვს სტრიქონი $S = ABCDE$.

საწყის S სტრიქონში $X = BC$ ქვესტრიქონის ინდექსი არის 1

საწყის S სტრიქონში $X = D$ ქვესტრიქონის ინდექსი არის 3

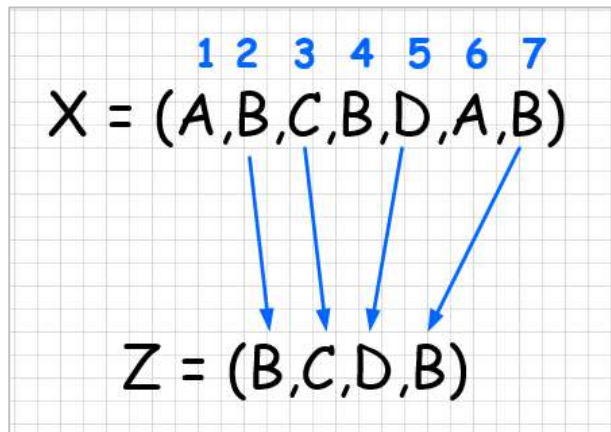
საწყის S სტრიქონში ქვესტრიქონი $X = BCE$ არ არსებობს

მიმდევრობა არის ელემენტების მოწესრიგებული ნაკრები. სტრიქონი კი არის მიმდევრობის კერძო შემთხვევა.

ქვემიმდევრობა. მოცემულია A და B მიმდევრობები: $a_1, a_2, \dots, a_n$ და $b_1, b_2, \dots, b_m$. რაიმე Z მიმდევრობა ჩაითვლება A მიმდევრობის ქვემიმდევრობად მაშინ და მხოლოდ მაშინ, თუ იარსებებს A მიმდევრობის ელემენტების მკაცრად მზარდი ინდექსების ნაკრები, რომელთაგანაც მივიღებთ Z ქვემიმდევრობას.

მოცემულია $X = (A, B, C, B, D, A, B)$ მიმდევრობა, ხოლო $Z = (B, C, D, B)$ არის X მიმდევრობის ქვემიმდევრობა.

დავადგინოთ Z ქვემიმდევრობის ინდექსთა თანამიმდევრობა (ნახ. 9.1.).



ნახ. 9.1. ქვემიმდევრობაში ინდექსების დადგენა

პასუხი: Z ქვემიმდევრობის ინდექსთა თანამიმდევრობაა $[2, 3, 5, 7]$

საერთო ქვემიმდევრობა. მოცემული გვაქვს ორი მიმდევრობა: $A = (a_1, a_2, \dots, a_n)$ და $B = (b_1, b_2, \dots, b_m)$. მათი საერთო p სიგრძის ქვემიმდევრობა არის p -ური ინდექსების მკაცრად ზრდადი ნაკრები:

$$1 \leq i_1 < i_2 < \dots < i_p \leq n, \quad 1 \leq j_1 < j_2 < \dots < j_p \leq m,$$

$$\text{სადაც } a_{i_1} = b_{j_1}, a_{i_2} = b_{j_2}, \dots, a_{i_p} = b_{j_p}.$$

ითვლება, რომ X -ისა და Y -ის საერთო ქვემიმდევრობა არის ისეთი Z მიმდევრობა, რომელიც არის ქვემიმდევრობა, როგორც X -ისთვის, ასევე Y -ისთვის.

უგრძესი საერთო ქვემიმდევრობა. უგრძესი საერთო ქვემიმდევრობა არის მაქსიმალური სიგრძის მქონე საერთო ქვემიმდევრობა საწყისი მიმდებრობებისთვის. უგრძესი საერთო ქვემიმდევრობის აღსაწერად გამოიყენება აბრევიატურა: **უსქ** ან **LCS** (Longest Common Subsequence). უდიდესი საერთო ქვემიმდევრობა შეიძლება რამოდენიმეც იყოს.

მაგალითად, მოცემული გვაქვს ორი მიმდევრობა:

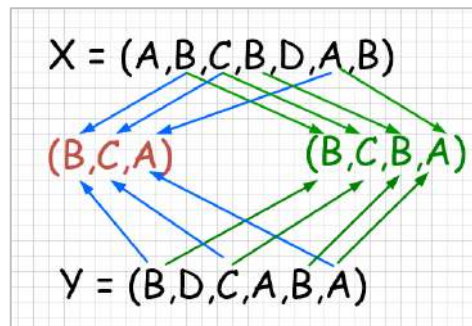
$$X = \text{HABRAHABR} \text{ და } Y = \text{HARBOUR},$$

$$\text{მაშინ } \text{უსქ}(X, Y) = \text{HARBR} \text{ ან შეიძლება ასეც ჩავწეროთ } \text{LCS}(X, Y) = \text{HARBR}$$

ნახატ 9.2-ზე მოცემულია ორი მიმდევრობა:

$$X = (A, B, C, B, D, A, B), Y = (B, D, C, A, B, A).$$

დავადგინოთ მათი უგრძესი საერთო ქვემიმდევრობა.



ნახ. 9.2. უგრძესი საერთო ქვემიმდევრობა მოცემულია მწვანე ფერით

პასუხი: $X = (A, B, C, B, D, A, B)$ და $Y = (B, D, C, A, B, A)$ მიმდევრობათა უგრძესი საერთო ქვემიმდევრობაა $(B, C, B, A) = 4$

თეორემა უგრძესი საერთო ქვემიმდევრობის (უსქ, LCS) შესახებ.

მოცემული გვაქვს ორი თანამიმდევრობა: $x_1, x_2, \dots, x_i$ და $y_1, y_2, \dots, y_j$.

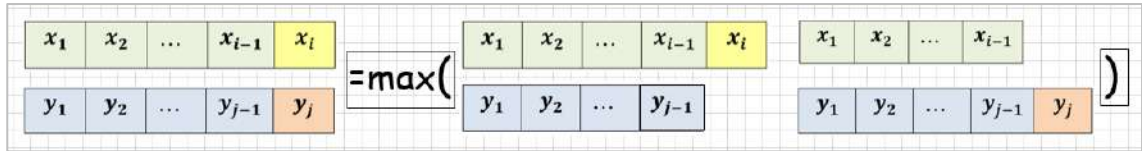
უგრძესი საერთო ქვემიმდევრობის სიგრძის დასადგენად რეკურენტულ თანაფარდობას აქვს შემდეგი სახე:

$$f(i, j) = \begin{cases} \max(f(i, j-1), f(i-1, j)) & \text{თუ } i, j > 0, x_i \neq y_j \\ f(i-1, j-1) + 1 & \text{თუ } i, j > 0, x_i = y_j \\ 0 & \text{თუ } i = 0 \text{ ან } j = 0 \end{cases} \quad (1)$$

ვთქვათ $f(i, j)$ არის მოცემული ორი თანამიმდევრობის უგრძესი საერთო ქვემიმდევრობა, მაშინ

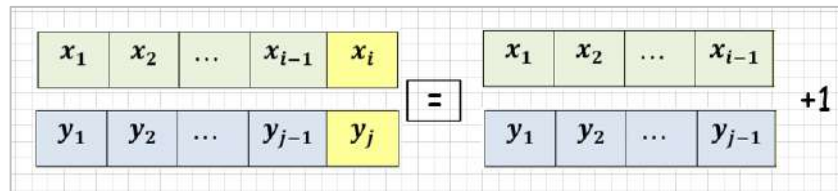
1) თუ $x_i \neq y_j$, მაშინ უსქეს ვეძებთ, როგორც $x_1, x_2, \dots, x_i$ და $y_1, y_2, \dots, y_{j-1}$ მიმდევრობებში, ასევე $x_1, x_2, \dots, x_{i-1}$ და $y_1, y_2, \dots, y_j$ მიმდევრობებში (იხ. ნახ. 9.3.).

შედეგად ვაბრუნებთ მათ შორის უდიდეს: $f(i, j) = \max(f(i, j-1), f(i-1, j))$



ნახ. 9.3. შემთხვევა როცა $x_i \neq y_j$

2) თუ $x_i = y_j$, მაშინ უსქეს ვეძებთ $x_1, x_2, \dots, x_{i-1}$ და $y_1, y_2, \dots, y_{j-1}$ მიმდევრობებს შორის (იხ. ნახ. 9.4.). ვაბრუნებთ $f(i, j) = 1 + f(i-1, j-1)$ შედეგს.

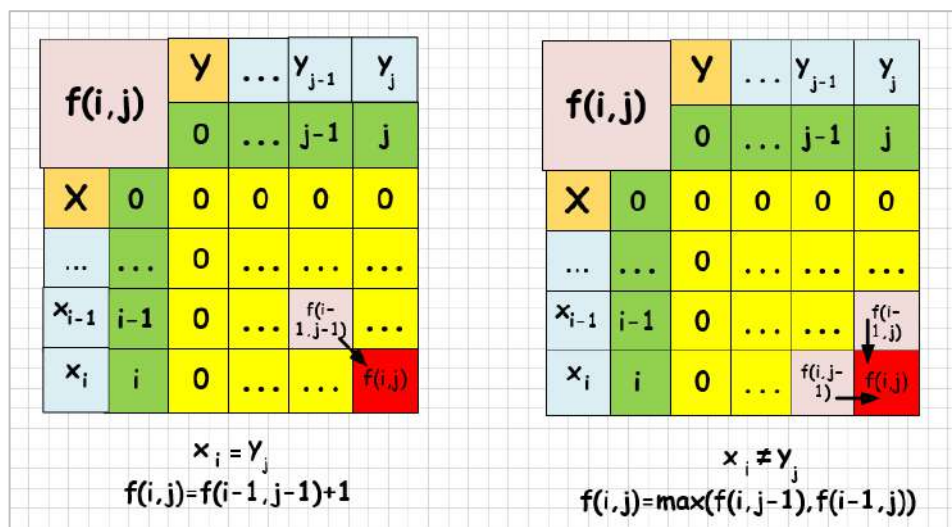


ნახ. 9.4. შემთხვევა როცა $x_i = y_j$

3) თუ ერთ-ერთი მიმდევრობა ცარიელია, მაშინ ასეთი მიმდევრობების უსქეს ცარიელია: $f(0, j) = f(i, 0) = 0$

შევაჯამოთ აღწერილი თეორემა. მიღებული $f(i, j)$ -ის მნიშვნელობები შევინახოთ $m[0..1000, 0..1000]$ ორგანზომილებიან მასივში (მატრიცაში), სადაც $m[0][i] = m[i][0] = 0$.

$m[i][j]$ ორგანზომილებიანი მასივის ყოველი შემდეგი სტრიქონი გამოითვლება წინა სტრიქონზე დაყრდნობით. ამიტომ, პასუხის მისაღებად საკმარისია მეხსიერებაში შევინახოთ მხოლოდ ორი სტრიქონი, რომელთა სიგრძეა 1000 (ნახ. 9.5.).



ნახ. 9.5. $f(i, j)$ მნიშვნელობების მატრიცაში შენახვის სქემა

მაგალითი 9.1. მოცემულია ორი მიმდევრობა: $X = abcdgh$, $Y = aedfhr$.

ვიპოვოთ X და Y მიმდევრობის უგრძესი საერთო ქვემიმდევრობა (ნახ. 9.6.).

$f(i, j)$		X	a	b	c	d	g	h
		0	1	2	3	4	5	6
Y	0	0	0	0	0	0	0	0
a	1	0	1(a)	1	1	1	1	1
e	2	0	1	1	1	1	1	1
d	3	0	1	1	1	2(d)	2	2
f	4	0	1	1	1	2	2	2
h	5	0	1	1	1	2	2	3(h)
r	6	0	1	1	1	2	2	3

ნახ. 9.6. უსქ = adh , $f(6, 6) = 3$

პასუხი:

უგრძესი საერთო ქვემიმდევრობაა adh , ხოლო სიგრძეა $f(6, 6) = 3$

$f(6, 6) = \max(f(6, 5), f(5, 6)) = \max(2, 3) = 3$, რადგან $Y[6] = r \neq h = X[6]$.

$f(5, 6) = 1 + f(4, 5) = 1 + 2 = 3$, რადგან $Y[5] = h = X[6]$.

ყველაზე უგრძესი საერთო ქვემიმდევრობის გამოთვლა რეკურსიის გამოყენებით.

იდეა მდგომარეობს იმაში, რომ ერთმანეთს უნდა შევადაროთ მიმდევრობების ბოლო ორი სიმბოლო.

შედარებისას უნდა განვიხილოთ ორი ვარიანტი:

1) ემთხვევა: შედარება გავიმეოროთ დანარჩენი $m - 1$ და $n - 1$ სიგრძის სტრიქონებისთვის და მიღებულ შედეგს დავუმატოთ 1.

2) არ ემთხვევა: შევასრულოთ ორი რეკურსიული გამოძახება:

პირველია $m - 1$ და n სიგრძის სტრიქონებისთვის, ხოლო მეორე m და $n - 1$ სიგრძის სტრიქონებისთვის. მიღებული ორი შედეგიდან ავირჩიოთ მაქსიმალური მნიშვნელობა.

თუ რომელიმე სტრიქონი ცარიელია, ჩვენ ვაბრუნებთ 0-ს.

მაგალითად, დავუშვათ, რომ გვაქვს ორი სტრიქონი: ABX და ACX .

ვიპოვოთ: LCS .

$LCS(ABX, ACX) = 1 + LCS(AB, AC)$ (ბოლო სიმბოლოების თანხვედრა)

$LCS(AB, AC) = \max(LCS(A, AC), LCS(AB, A))$ (ბოლო სიმბოლოები არ დაემთხვა)

$LCS(A, AC) = \max(LCS(„,“, AC), LCS(A, A)) = \max(0, 1 + LCS(„,“, „“)) = 1$

$LCS(AB, A) = \max(LCS(A, A), LCS(AB, „“)) = \max(1 + LCS(„, „“, 0)) = 1$

პასუხი: შედეგი არის $1 + 1 = 2$.

უსქ (LCS) ალგორითმის დროითი და სივრცითი სირთულე.

დროითი სირთულე. უსქ (*LCS*) ალგორითმის დროითი სირთულეა $O(m * n)$, სადაც m და n არის X და Y მიმდევრობების სიგრძე. ეს განპირობებულია ორი ჩადგმული ციკლის გამო, რომელიც გადაირბენს/დაათვალიერებს X და Y მიმდევრობების ყველა სიმბოლოს.

სივრცითი სირთულე. უსქ (*LCS*) ალგორითმის სივრცითი სირთულეც არის იგივე $O(m * n)$, რადგან ხდება $(m + 1) * (n + 1)$ ზომის ცხრილის (მატრიცის) გამოყენება.

ერთი სტრიქონის სხვა სტრიქონად გარდაქმნა. ერთი სტრიქონის მეორე სტრიქონად გარდასაქმნელად გამოიყენება ლევენშტეინის მანძილი, იმავე ცვლილებათა მანძილი (Edit Distance ან Levenshtein Distance).

ერთი სტრიქონის სხვა სტრიქონად გარდაქმნა ნიშნავს, დავადგინოთ თუ რამდენი მინიმალური ცვლილებაა საჭირო ერთი სტრიქონის მეორე სტრიქონად გარდასაქმნელად.

ლევენშტეინის ამოცანა. ლევენშტეინის ამოცანა არის ამოცანა ერთი სიტყვის მეორე სიტყვად გარდაქმნის შესახებ, სადაც ხდება სიმბოლოს (ასო-ნიშნის) დამატების, წაშლის და შეცვლის, ოპერაციების დათვლა. დათვლილი ოპერაცია უნდა იყოს მინიმალური რაოდენობის. ამოცანა შეიძლება ასეც ჩამოყალიბდეს: ვიპოვოთ მინიმალური „მანძილი“ ორ სიტყვას შორის ანუ დავითვალოთ ცვლილებების მინიმალური რაოდენობა. მანძილი იქნება იმ ოპერაციების მინიმალური რაოდენობა, რომელიც უნდა შესრულდეს პირველი სიტყვიდან მეორე სიტყვის მისაღებად (ან პირიქით).

ლევენშტეინის ამოცანის ამოსახსნელად გვაქვს შემდეგი ოპერაციები:

1) სიმბოლოს დამატება (**insert**): სტრიქონის ნებისმიერ ადგილში ახალი სიმბოლოს ჩასმა;

2) სიმბოლოს წაშლა (**delete**): სტრიქონის ნებისმიერი ადგილიდან სიმბოლოს ამოშლა;

3) სიმბოლოს ჩანაცვლება (**replace**): ერთი სიმბოლოს შეცვლა მეორე სიმბოლოთი.

ვთქვათ მოცემული გვაქვს ორი სტრიქონი: $str1 = \text{kitten}$ და $str2 = \text{sitting}$

$kitten$ -ის $sitting$ -ად გადასაქცევად საჭიროა შემდეგი ცვლილებების შესრულება:

1. $kitten \rightarrow sitten$ (პირველი სიმბოლოს შეცვლა: k -ს შეცვლა s -ით)
2. $sitten \rightarrow sittin$ (მეხუთე სიმბოლოს შეცვლა: e -ს შეცვლა i -ით)
3. $sittin \rightarrow sitting$ (ბოლოში g -ის დამატება)

ლევენშტეინის მანძილი არის 3, რადგან საჭირო გახდა $kitten$ -ის $sitting$ -ად გადასაქცევად სამი ცვლილების ჩატარება.

ლევენშტეინის ამოცანის ალგორითმის აღწერა: ლევენშტეინის მანძილის მოსაძებნად, ყველაზე ხშირად გამოიყენება ისეთი სახის ალგორითმი, სადაც ხდება $n + 1$ და $m + 1$ ზომის რაიმი D მატრიცის შევსება (n და m არის იმ A და B სტრიქონების სიგრძე, რომლებიც უნდა შევადაროთ ერთმანეთს) შემდეგი წესების მიხედვით:

$$D_{0,0} = 0$$

$D_{i,j}$ = ავირჩიოთ მინიმალური შემდეგიდან:

$D_{i-1,j-1} + 0$ (სიმბოლოები ერთნაირია)

$D_{i,j-1} + w_{delete}$ (სიმბოლოს წაშლა)

$D_{i-1,j} + w_{insert}$ (სიმბოლოს ჩასმა)

$D_{i-1,j-1} + w_{replace}$ (სიმბოლოს შეცვლა)

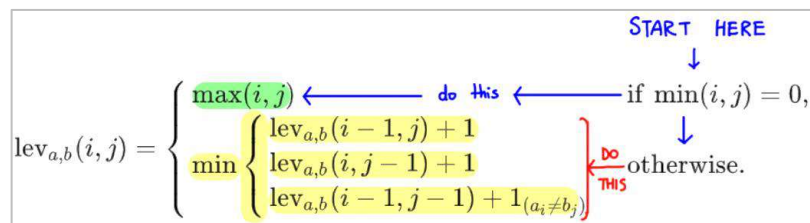
სადაც w_{delete} , w_{insert} , $w_{replace}$ არის სიმბოლოს წაშლის, ჩასმის და შეცვლის ღირებულება.

ამასთან $D_{n-1,m-1}$ შეიცავს ლევენშტეინის მანძილის მნიშვნელობას.

აღწერილი ალგორითმიდან გამომდინარე ლევენშტეინის ფორმულას ექნება ქვემოთ მოცემული სახე:

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{სხვა შემთხვევაში} \end{cases} \quad (1)$$

ლევენშტეინის ფორმულა უფრო დაწვრილებითი აღწერით მოცემული გვაქვს ნახატ 9.7.-ზე.



ნახ. 9.7. ლევენშტეინის ფორმულა

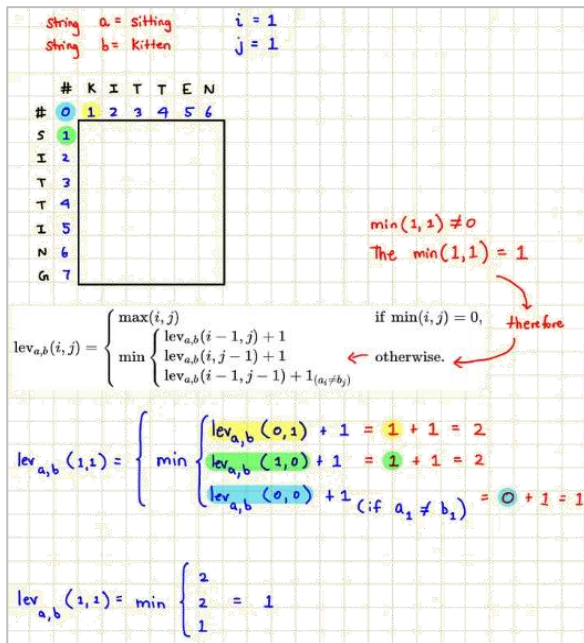
მაგალითი 9.2. მოცემული გვაქვს ორი სტრიქონი: $a = \text{SITTING}$ და $b = \text{KITTE}$. დავადგინოთ ლევენშტეინის მანძილი ანუ SITTING -სგან უნდა მივიღოთ KITTE .

ყურადღება მივაქციოთ იმას, რომ მიმდევრობის პირველი სიმბოლოს ინდექსაცია იწყება 1-ით, როგორც მათემატიკაშია მიღებული და არა 0-ით (ნახ. 9.8.).

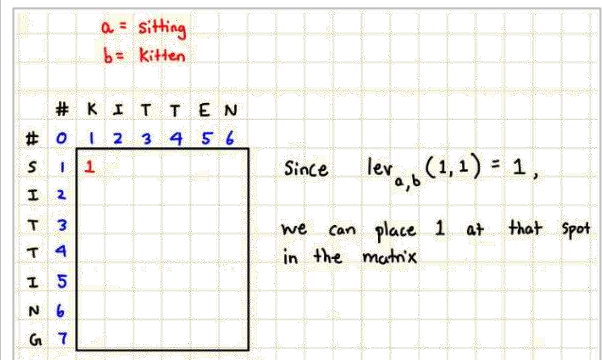
String a = sitting							
String b = kitten							
	#	K	I	T	T	E	N
#	0	1	2	3	4	5	6
S	1						
I	2						
T	3						
T	4						
I	5						
N	6						
G	7						

ნახ. 9.8. ლევენშტეინის მანძილის დასადგენი საწყისი მატრიცა

გადავიდეთ მატრიცის/ცხრილის შევსებაზე (იხ. ნახ. 9.9.ა,ბ). ჩვენი მიზანია შევავსოთ მთელი მატრიცა, დაწყებული ზედა მარცხენა კუთხიდან ქვედა მარჯვენა კუთხემდე. ქვედა მარჯვენა კუთხეში არსებული მნიშვნელობა მოგვცემს ლევენშტეინის მანძილს.



ა)

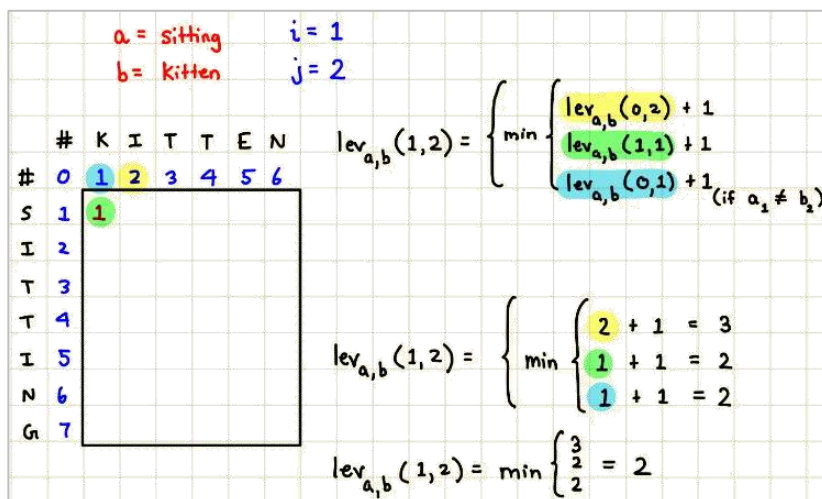


ბ)

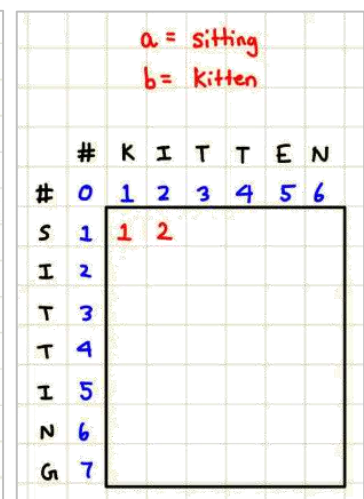
ნახ. 9.9. ლევენშტეინის: ა) მანძილის გამოთვლა $i = 1$ და $j = 1$ ელემენტებისთვის;

ბ) მატრიცის პირველი უჯრის შევსება;

მატრიცის დანარჩენ ნაწილი შევავსოთ იგივე ფუნქციის გამოყენებით მატრიცის ყველა დანარჩენი უჯრებისათვის. კიდევ ერთი ამოხსნა $i = 1$ და $j = 2$ უჯრისათვის (ნახ. 9.10.ა,ბ.)



ა)



ბ)

ნახ. 9.10. ლევენშტეინის: ა) მანძილის გამოთვლა $i = 1$ და $j = 2$ ელემენტებისთვის;

ბ) მატრიცის მომდევნო უჯრის შევსება;

ასეთი სახით გავნაგრძოთ მატრიცის დანარჩენი უჯრების შევსება (ნახ. 9.11.).

X ქვესტრიქონს ეწოდება Y სტრიქონის სუფიქსი, თუ არსებობს ისეთი Z ქვესტრიქონი, რომ შესრულდეს $Y = ZX$ ტოლობა. ანალოგიურად, სტრიქონი შეიძლება იყოს საკუთარი თავის სუფიქსი. მაგალითად, ქვესტრიქონი $bf g$ არის $vsenf b f g$ სტრიქონის სუფიქსი.

განვიხილოთ სტრიქონში ქვესტრიქონის ძებნის ამოცანა. ვთქვათ, რომ მოცემულია სტრიქონი, რომელიც შედგება გარკვეული რაოდენობის სიმბოლოებისგან. შევამოწმოთ შეიცავს თუ არა სტრიქონი მოცემულ ქვესტრიქონს. თუ სტრიქონი შეიცავს ქვესტრიქონს, ვიპოვოთ სტრიქონში ქვესტრიქონების შემოსვლის რიცხვი.

არსებობს ქვესტრიქონის მოძებნის მრავალი ალგორითმი. ჩვენ განვიხილავთ ორ მათგანს: პირდაპირი ძებნის ალგორითმს და კნუტ-მორის-პრატის ალგორითმს.

პირდაპირი ძებნა. პირდაპირ ძებნას ხშირად უწოდებენ თანამიმდევრობით ძებნასაც. იგი განეკუთვნება ძებნის უმარტივეს ალგორითმთა რიცხვს. პირდაპირი ძებნის ალგორითმის იდეა მარტივია და ახდენს სტრიქონის სიმბოლოების შედარებას ქვესტრიქონის სიმბოლოებთან.

თავდაპირველად ხდება სტრიქონის პირველი სიმბოლოს შედარება ქვესტრიქონის პირველ სიმბოლოსთან, სტრიქონის მეორე სიმბოლოს შედარება ქვესტრიქონის მეორე სიმბოლოსთან და ა.შ. თუ სტრიქონისა და ქვესტრიქონის სიმბოლოები ზუსტად დაემთხვევა ერთმანეთს, მაშინ მოხდება ქვესტრიქონის მოძებნის დაფიქსირება.

საწინააღმდეგო შემთხვევაში, ქვესტრიქონი გადაინაცვლებს/წაინაცვლებს ერთი პოზიციით მარჯვნივ და გამეორდება სიმბოლო-ნიშანთა შედარება ანუ სტრიქონის მეორე სიმბოლო შეედარება ქვესტრიქონის პირველ სიმბოლოს, სტრიქონის მესამე სიმბოლო ქვესტრიქონის მეორე სიმბოლოს და ა.შ. საბოლოოდ, უნდა დავადგინოთ ქვესტრიქონის დასაშვები წანაცვლების რაოდენობა (იხ. ნახ. 9.12.). პირდაპირი ძებნის ალგორითმის აღწერა:

პირველი ბიჯი. მოცემული გვაქვს ორი სტრიქონი: ძირითადი სტრიქონი/ტექსტი T და ქვესტრიქონი P . უნდა დავადგინოთ თუ რამდენი წანაცვლების შემდეგ მოხდება ქვესტრიქონის დამთხვევა სტრიქონის სეგმენტთან (ან არ დამთხვევა).

მეორე ბიჯი. შედარებას ვიწყებთ ტექსტის პირველივე პოზიციიდან და ვმოძრაობთ ტექსტის ბოლოსკენ. დავიწყეთ ქვესტრიქონის შედარება ძირითად T სტრიქონთან.

მესამე ბიჯი. სტრიქონის/ტექსტის თითოეულ i -ურ პოზიციაზე ვამოწმებთ, მოხდა თუ არა P ქვესტრიქონის T სტრიქონთან თანხვედრა. თუ P ქვესტრიქონი სრულად დაემთხვა T სტრიქონის/ტექსტის რომელიმე სეგმენტს, მაშინ გამოდის, რომ მოვძებნეთ ქვესტრიქონი.

მეოთხე ბიჯი. თუ არა, მაშინ გადავადგილებთ/წავანაცვლებთ P ქვესტრიქონს ერთი პოზიციით მარჯვნივ და კვლავ ვიმეორებთ შემოწმების პროცედურას.

მეხუთე ბიჯი. შემოწმების პროცესს გავაგრძელებთ მანამ, სანამ არ მივიღებთ თანხვედრას სტრიქონსა და ქვესტრიქონს შორის. თუ ალგორითმი ვერ იპოვის ქვესტრიქონს, მაშინ დავასკვნით, რომ ქვესტრიქონი ტექსტში არ მოიძებნა.

პირდაპირი ძებნის ალგორითმის დროითი სირთულეა $O(m*n)$, სადაც n არის სტრიქონის სიგრძე, ხოლო m არის ქვესტრიქონის სიგრძე.

მაგალითად, მოცემულია სტრიქონი $T = abcdefgh$ და ქვესტრიქონი $P = def$.

შედარება დავიწყოთ სტრიქონის პირველივე სიმბოლოდან. შევადაროთ $abcdefgh$ სტრიქონს ქვესტრიქონი def . ისინი არ ემთხვევა ერთმანეთს.

გადავწიოთ/წავანაცვლოთ P ქვესტრიქონი ერთი ნაბიჯით მარჯვნივ. ახლა შევადაროთ $bcdefgh$ სტრიქონს def ქვესტრიქონი. კვლავ არ ემთხვევა ქვესტრიქონი სტრიქონს.

ქვესტრიქონი კვლავ გადავწიოთ/წავანაცვლოთ მარჯვნივ. ახლა შევადაროთ $cdefgh$ სტრიქონს def ქვესტრიქონი. ისევ არ ემთხვევა.

ქვესტრიქონი კიდევ ერთხელ გადავწიოთ/წავანაცვლოთ მარჯვნივ. შევადაროთ $defgh$ სტრიქონს def ქვესტრიქონი. მოხდა სტრიქონის სეგმენტისა და ქვესტრიქონის დამთხვევა. ქვესტრიქონი მოიძებნა.

პასუხი: სტრიქონში ქვესტრიქონის შემოსვლა მოხდა ერთხელ, წანაცვლება მოხდა სამჯერ $S = 3$, ინდექსი $i = 4$.

პირდაპირ ძებნის ალგორითმი ეფექტურია მცირე ტექსტებთან მიმართებით. მისი გამოყენება დიდი მოცულობის ტექსტებში შეიძლება აღმოჩნდეს არაეფექტური, რადგან პირდაპირ ძებნის ალგორითმი ახდენს ყველა შესაძლო პოზიციის შემოწმებას.

სტრიქონი	A	B	C	A	B	C	A	A	B	C	A	B	D
ქვესტრიქონი	A	B	C	A	B	D							
		A	B	C	A	B	D						
			A	B	C	A	B	D					
				A	B	C	A	B	D				
					A	B	C	A	B	D			
						A	B	C	A	B	D		
							A	B	C	A	B	D	
								A	B	C	A	B	D

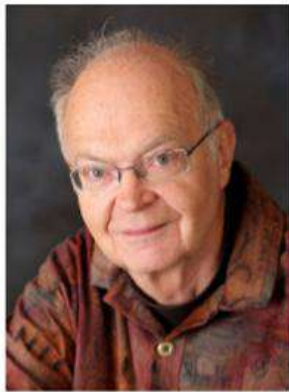
ნახ. 9.12. პირდაპირი ძებნის ალგორითმის დემონსტრირება

კნუტ-მორის-პრატის ალგორითმი. კნუტ-მორის-პრატის ალგორითმი არის პირდაპირი ძებნის ალგორითმის საკმაოდ გაუმჯობესებული ვარიანტი, რომელიც მუშაობს წრფივ დროში.

კნუტ-მორის-პრატის ალგორითმი აღმოაჩინეს დ. კნუტმა და დ. მორისმა და მათგან დამოუკიდებლად აღმოაჩინა ვ. პრატმა. სამივე მეცნიერის მუშაობის შედეგი გამოქვეყნდა 1977 წელს (იხ. ნახ. 9.13.ა,ბ,გ). კნუტ-მორის-პრატის ალგორითმი (Knuth-Morris-Pratt, KMP) ფაქტობრივად მხოლოდ $O(n)$ შედარებას მოითხოვს ყველაზე ცუდ შემთხვევაშიც კი.

თავდაპირველად სტრიქონის საწყის სიმბოლოებს ვადარებთ ქვესტრიქონის/შაბლონის საწყის სიმბოლოებს მარცხნიდან მარჯვნივ მიმართულებით. თუ მოხდება ქვესტრიქონის საწყისი სიმბოლოების სტრიქონის სიმბოლოებთან ნაწილობრივი თანხვედრა, მაშინ ცნობილი ხდება სტრიქონის გავლილი ნაწილი და შესაძლებელია წანაცვლების სიდიდის გამოთვლა. წანაცვლების სიდიდის დახმარებით შესაძლებელია სწრაფად გადავადგილდეთ სტრიქონის გასწვრივ. კნუტ-მორის-პრატის ალგორითმში წანაცვლების სიდიდე გამოითვლება პრეფიქს-ფუნქციის გამოყენებით.

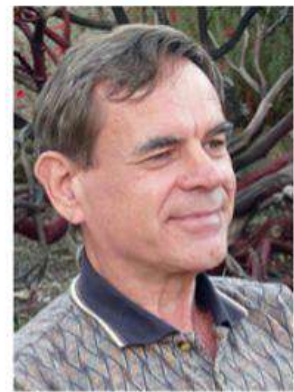
კნუტ-მორის-პრატის ალგორითმი არის ერთ-ერთი ყველაზე ეფექტური სტრიქონული ძებნის ალგორითმი, რომელიც ეძებს ქვესტრიქონს სტრიქონში/ტექსტში. მისი ერთ-ერთი მნიშვნელოვანი კომპონენტი არის პრეფიქსფუნქცია.



ა)



ბ)



გ)

ნახ. 9.13. მეცნიერები: ა) დონალდ კნუტი; ბ) ჯეიმს მორისი გ) ვონ პრატი

პრეფიქსფუნქცია. პრეფიქსფუნქცია (π) ადგენს, თუ რა სიგრძის ყველაზე გრძელი პრეფიქსი(სტრიქონის დასაწყისი) ემთხვევა ქვესტრიქონის/შაბლონის სუფიქსს (სტრიქონის დასასრულს). სხვა სიტყვებით, რომ ვთქვათ, პრეფიქს-ფუნქცია (π) სტრიქონის თითოეული პოზიციისთვის ადგენს, თუ რა სიგრძის ყველაზე გრძელი პრეფიქსი არსებობს, რომელიც ასევე არის სუფიქსი ამ სტრიქონის იმავე ნაწილში. ქვესტრიქონის ძიების პროცესის დროს KMP ალგორითმი იყენებს პრეფიქს-ფუნქციას და ახდენს წანაცვლებების ოპტიმიზაციას. როდესაც ქვესტრიქონის/შაბლონის სიმბოლო არ ემთხვევა ტექსტის/სტრიქონის შესაბამის სიმბოლოს, პრეფიქს-ფუნქცია საშუალებას გვაძლევს, წავინაცვლოთ შესაბამის პოზიციაზე ისე, რომ არ მოგვიწიოს ძებნის თავიდან დაწყება.

პრეფიქსფუნქციის (π) მნიშვნელობების გამოთვლა. ვთქვათ, მოცემული გვაქვს $s[0 \dots n - 1]$ სტრიქონი. გამოვითვალოთ მოცემული სტრიქონის პრეფიქს-ფუნქცია ანუ შევე-

მნათ $\pi[0 \dots n - 1]$ სახის მასივი, სადაც $\pi[i]$ არის ქვესტრიქონის ყველაზე გრძელი სუფიქსის მაქსიმალური სიგრძე, რომელიც ემთხვევა მის პრეფიქსს.

პრეფიქსი არ უნდა ემთხვეოდეს მთელ სტრიქონს. ითვლება, რომ $\pi[0]$ -ის მნიშვნელობა ტოლია ნულის, რადგან პირველი სიმბოლოსთვის სუფიქსი არ არსებობს.

პრეფიქსფუნქციის მათემატიკური აღწერა:

$$\pi[i] = \max\{k: s[0 \dots k - 1] = s[i - k + 1 \dots i]\}, \text{ სადაც } k = 0 \dots i \quad (2)$$

მაგალითად, სტრიქონის **abcabcd** პრეფიქს-ფუნქცია არის შემდეგი სახის მასივი:
[0,0,0,1,2,3,0]

განვიხილოთ დაწვრილებით:

სტრიქონს **a** არ აქვს პრეფიქსი, რომელიც ემთხვევა სუფიქსს;

სტრიქონს **ab** არ აქვს პრეფიქსი, რომელიც ემთხვევა სუფიქსს;

სტრიქონს **abc** არ აქვს პრეფიქსი, რომელიც ემთხვევა სუფიქსს;

სტრიქონს **abca** აქვს პრეფიქსი, რომლის სიგრძე ემთხვევა სუფიქსს;

სტრიქონს **abcab** აქვს პრეფიქსი, რომლის სიგრძე ემთხვევა სუფიქსს;

სტრიქონს **abcabc** აქვს პრეფიქსი, რომლის სიგრძე ემთხვევა სუფიქსს;

სტრიქონს **abcabcd** არ აქვს სუფიქსის შესაბამისი პრეფიქსი.

პრეფიქსფუნქცია შეგვიძლია ასე აღვწეროთ:

პრეფიქს-ფუნქცია $\pi(i)$ წარმოადგენს ისეთი სახის მაჩვენებელს, რომელიც ითვლის მაქსიმალურ სიგრძეს, სადაც $P[1:i]$ ქვესტრიქონი/შაბლონი ემთხვევა მის წინ არსებულ პრეფიქსს.

მაგალითად, მოცემულია $P = ABABACA$ სტრიქონი.

გამოვითვალოთ მოცემული სტრიქონის პრეფიქსფუნქციის მნიშვნელობები:

$i = 1: P[1:1] = A \rightarrow$ აქამდე პრეფიქსი არ არის, ამიტომ $\pi(1) = 0$;

$i = 2: P[1:2] = AB \rightarrow$ არ არსებობს პრეფიქსი, რომელიც ემთხვევა სუფიქსს, ამიტომ $\pi(2) = 0$;

$i = 3: P[1:3] = ABA \rightarrow$ პრეფიქსი **A** ემთხვევა სუფიქსს **A**-ს, ამიტომ $\pi(3) = 1$;

$i = 4: P[1:4] = ABAB \rightarrow$ პრეფიქსი **AB** ემთხვევა სუფიქსს **AB**-ს, ამიტომ $\pi(4) = 2$;

$i = 5: P[1:5] = ABABA \rightarrow$ პრეფიქსი **ABA** ემთხვევა სუფიქსს **ABA**-ს, ამიტომ $\pi(5) = 3$;

$i = 6: P[1:6] = ABABAC \rightarrow$ არ არსებობს პრეფიქსი, რომელიც ემთხვევა სუფიქსს, ამიტომ $\pi(6) = 0$;

$i = 7: P[1:7] = ABABACA \rightarrow$ არ არსებობს პრეფიქსი, რომელიც ემთხვევა სუფიქსს, ამიტომ $\pi(7) = 1$.

პასუხი: პრეფიქსფუნქციის მნიშვნელობებია: $\pi = [0,0,1,2,3,0,1]$

KMP ალგორითმის მთავარი მახასიათებელი არის ის, რომ იგი არ განიხილავს თითოეულ სიმბოლოს ცალკე, არამედ იყენებს პრეფიქსფუნქციას, რათა უფრო ეფექტურად მართოს ტექსტში ქვესტრიქონის/შაბლონის ძიების პროცესი.

KMP ალგორითმი ემყარება იმ ფაქტს, რომ ქვესტრიქონის საწყისი ნაწილის ნაწილობრივი დამთხვევის შემდეგ სტრიქონის შესაბამის სიმბოლოებთან, სტრიქონის გავლილი ნაწილი რეალურად ხდება ცნობილი და პრეფიქსფუნქციის გამოთვლის დახმარებით შეიძლება სწრაფად წავინაცვლოთ სტრიქონის გასწვრივ.

კნუტ-მორის-პრატის ალგორითმსა და პირდაპირი ძებნის ალგორითმს შორის მთავარი განსხვავება ისაა, რომ ალგორითმის თითოეულ ბიჯზე ქვესტრიქონის წანაცვლება ხდება არა ერთი სიმბოლოთი, არამედ სიმბოლოების გარკვეული ცვლადი რაოდენობით.

ამიტომ, მომდევნო წანაცვლების მისაღებად, აუცილებელია წანაცვლების სიდიდის დადგენა. ალგორითმის ეფექტურობის გასაზრდელად აუცილებელია, რომ წანაცვლება თითოეულ ბიჯზე იყოს რაც შეიძლება დიდი (ნახ. 9.14.).

ნახატ 9.14-ზე შედარებული სიმბოლოები მონიშნულია ლურჯი ფერით.

სტრიქონი	A	B	C	A	B	C	A	A	B	C	A	B	D
ქვესტრიქონი	A	B	C	A	B	C	A	B	D				
				A	B	C	A	B	C	A	B	D	
							A	B	C	A	B	D	
								A	B	C	A	B	D

ნახ. 9.14. კნუტ-მორის-პრატის ძებნის ალგორითმის დემონსტრირება

KMP ალგორითმის მიახლოებითი აღწერა:

1. ქვესტრიქონებისთვის უნდა ავაგოთ წანაცვლების ცხრილი $d[m]$;
2. $d[0] = -1$, $d[1] = 0$;
3. $d[j]$ - ქვესტრიქონის სიმბოლოების ყველაზე გრძელი თანამიმდევრობის სიგრძე, წინ უსწრებს j -ს და მთლიანად ემთხვევა ქვესტრიქონის/შაბლონის დასაწყისს;
4. ქვესტრიქონის ტექსტთან j -ური დამთხვევისას, შეგვიძლია შევასრულოთ წანაცვლება $j - d[j]$ მანძილზე.

მოცემულია სტრიქონი/ტექსტი: **abcabcaabcababcbabcbabd**;

დავადგინოთ არსებობს თუ არა მოცემულ სტრიქონში ქვესტრიქონი **abcabd** კნუტ-მორის-პრატის ალგორითმის მიხედვით (ნახ. 9.15.).

a	b	c	a	b	d	d -1 0 0 0 1 2						წანაცვლება j-d[j]												
a	b	c	a	b	c	a	a	b	c	a	b	a	b	c	b	a	b	c	a	b	d			
a	b	c	a	b	d	გვაქვს 5 დამთხვევა, წანაცვლება 5-2=3																		
			a	b	c	a	b	d	გვაქვს 4 დამთხვევა, წანაცვლება 4-1=3															
						a	b	c	a	b	d	გვაქვს 1 დამთხვევა, წანაცვლება 1-0=1												
							a	b	c	a	b	d	გვაქვს 5 დამთხვევა, წანაცვლება 3											
										a	b	c	a	b	d	გვაქვს 2 დამთხვევა, წანაცვლება 2								
											a	b	c	a	b	d	წანაცვლება 3							
														a	b	c	a	b	d	წანაცვლება 1				
																a	b	c	a	b	d	!!!		

ნახ. 9.15. კნუტ-მორის-პრატის ალგორითმის ვიზუალიზაცია

მატრიცები და მათზე მოქმედებები. მატრიცა არის რიცხვების ან სიმბოლოების მასივი (ცხრილი). მატრიცა შედგება სტრიქონების და სვეტებისგან, ამდენად შეგვიძლია ვთქვათ რომ იგი განეკუთვნება ორ განზომილებიან სისტემას. მატრიცის სვეტებსა და სტრიქონებში განლაგებულ მონაცემებს მატრიცის ელემენტები ეწოდებათ. სხვანაირად შეიძლება ასეც განვსაზღვროთ: მონაცემებს, რომლებიც ქმნიან მატრიცას, ეწოდება მატრიცის ელემენტები.

მატრიცის თითოეულ ელემენტს/წევრს გააჩნია ორი ინდექსი, იმის მიხედვით, თუ რომელი სტრიქონისა და რომელი სვეტის გადაკვეთაზე იმყოფება მატრიცის ელემენტი.

მატრიცის ერთ-ერთი მთავარი მახასიათებელია რიგითობა. მატრიცის რიგითობა გვიჩვენებს, თუ რამდენი სტრიქონისა და რამდენი სვეტისგან შედგება მატრიცა.

მიღებულია, რომ მატრიცის სტრიქონების რაოდენობა აღვნიშნოთ m -ით, ხოლო სვეტების რაოდენობა კი n -ით, მაშინ გამოდის, რომ მატრიცა შეიცავს $m * n$ რაოდენობის ელემენტს.

თუ $m \neq n$, მაშინ მატრიცას ეწოდება მართკუთხა მატრიცა;

თუ $m = n$, მაშინ მატრიცას ეწოდება კვადრატული მატრიცა, რიგითობა $= n$;

მატრიცა შეგვიძლია აღვნიშნოთ ბეჭდური ლათინური ასო-ნიშნით, მაგალითად A ასო-ნიშნით.

მატრიცა შეიძლება ასეთი სახითაც იყოს აღწერილი: $A = [a_{ij}]$, სადაც a_{ij} -ს უწოდებენ მატრიცის ელემენტებს, ხოლო i -ირსა და j -ურს კი უწოდებენ მატრიცის ინდექსებს.

i -ურით აღინიშნება სტრიქონების რაოდენობა: $i = 1, 2, \dots, m$, ხოლო j -ურით აღინიშნება სვეტების რაოდენობა: $j = 1, 2, \dots, n$. მაგალითად, ჩანაწერი $A_{24} = 9$ ნიშნავს, რომ მატრიცის ელემენტი 9-ანი იმყოფება მე-2 სტრიქონისა და მე-4 სვეტის გადაკვეთაზე.

მატრიცის ჩაწერა ხდება კვადრატულ ფრჩხილებში $[]$, თუმცა შეიძლება იგი მრგვალი ფრჩხილების დახმარებითაც აღვწეროთ:

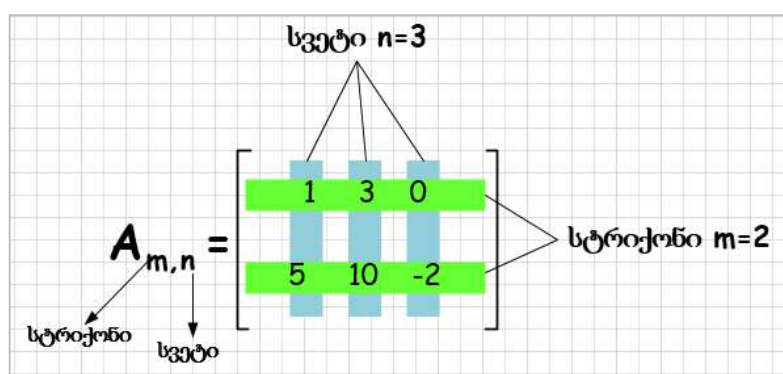
$$A = [a_{ij}] = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \quad A = (a_{ij}) = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

მაგალითად, მატრიცა სამი სტრიქონით და ორი სვეტით ($m = 3, n = 2$) ჩაიწერება

შემდეგი სახით: $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$

მნიშვნელოვანია ვიცოდეთ: მატრიცაში $m * n$ რიგითობის პირველი რიცხვი m ყოველთვის არის სტრიქონების რაოდენობის აღმნიშვნელი, ხოლო რიგითობის მეორე რიცხვი n ყოველთვის არის სვეტების რაოდენობის აღმნიშვნელი.

დავიმახსოვროთ: მატრიცის რიგითობაში პირველი არის სტრიქონი, მეორე კი სვეტი და არა პირიქით (ნახ. 9.16.).



ნახ. 9.16. მატრიცა ორი სტრიქონით ($m = 2$) და სამი სვეტით ($n = 3$)

მატრიცას, რომლის ყველა ელემენტი არის ნულის ტოლი ნულოვანი მატრიცა ეწოდება (ნახ. 9.16.)

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}; \quad \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}; \quad \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

ნახ. 9.16. ნულოვანი მატრიცა

ერთეულოვანი მატრიცა არის კვადრატული ($n * n$) სახის მატრიცა, სადაც მთავარ დიაგონალზე მყოფი ელემენტი უდრის ერთს, ხოლო ყველა დანარჩენი ელემენტი (მთავარი დიაგონალის გარეთ) არის ნულის ტოლი (ნახ. 9.17.).

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}; \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

ნახ. 9.17. ერთეულოვანი მატრიცა

თუ მატრიცის სტრიქონებსა და სვეტებს შევუცვლით ადგილებს, მივიღებთ ტრანსპონირებულ მატრიცას. ტრანსპონირებული მატრიცა აღინიშნება ლათინური ბეჭდური T ასო-ნიშნით (ნახ. 9.18.).

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

ნახ. 9.18. საწყისი და ტრანსპონირებული მატრიცა

ოპერაციები მატრიცებზე. ოპერაციები მატრიცებთან მიმართებით ძალიან ჰგავს რიცხვებზე მათემატიკურ ოპერაციებს, თუმცა არსებობს გარკვეული განსხვავებაც.

მატრიცის შეკრება. ერთსა და იმავე რიგითობის მქონე ორი მატრიცის ჯამი, არის ისეთი მესამე მატრიცა, რომლის თითოეული ელემენტი უდრის შესაბამისი მატრიცების შესაბამისი ელემენტების ჯამს:

თუ მოცემულია $A = [a_{ij}]_{mn}$ და $B = [b_{ij}]_{mn}$ ორი მატრიცა,

მაშინ $A + B = [c_{ij}]_{mn}$, სადაც $c_{ij} = a_{ij} + b_{ij}$ ($i = 1, \dots, m; j = 1, \dots, n$)

მაგალითად,

$$1. \begin{bmatrix} -3 & 4 & 5 \\ 2 & -1 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} -3+1 & 4+2 & 5+3 \\ 2+0 & -1+1 & 0+2 \end{bmatrix} = \begin{bmatrix} -2 & 6 & 8 \\ 2 & 0 & 2 \end{bmatrix}$$

$$2. \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$$

მატრიცების გამოკლება. ერთსა და იმავე რიგითობის მქონე ორი მატრიცისთვის შეგვიძლია ჩავატაროთ გამოკლების ოპერაცია: $A - B = A + [-B]$

თუ მოცემულია $A = [a_{ij}]$, $B = [b_{ij}] \in M_{m \times n}$:

მაშინ $A - B = A + [-B] = [a_{ij} - b_{ij}]_{mn}$, სადაც $i = 1, \dots, m; j = 1, \dots, n$

მაგალითად,

$$\begin{bmatrix} 10 & 9 & 8 \\ 7 & 6 & 5 \end{bmatrix} - \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 8 & 7 & 6 \\ 5 & 4 & 3 \end{bmatrix}$$

მატრიცის გამრავლება რიცხვზე. იმისათვის, რომ რიცხვი გადავამრავლოთ მატრიცაზე (ან მატრიცა გადავამრავლოთ რიცხვზე) საჭიროა მატრიცის თითოეული ელემენტი გადავამრავლოთ ამ რიცხვზე:

თუ $r \in R$, $A = [a_{ij}]_{mn}$,

მაშინ $rA = [c_{ij}]_{mn}$, სადაც $c_{ij} = r * a_{ij}$ ($i = 1, \dots, m; j = 1, \dots, n$)

მაგალითად,

$$3 \begin{bmatrix} -1 & 2 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} -3 & 2 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3 * (-1) - (-3) & 3 * 2 - 2 \\ 3 * 0 - 1 & 3 * 1 - 0 \end{bmatrix} = \begin{bmatrix} 0 & 4 \\ -1 & 3 \end{bmatrix}$$

მატრიცის ტრანსპონირება. მატრიცა A -ს უწოდებენ ტრანსპონირებულ მატრიცას (A^T), თუ იგი მიიღება A მატრიცის სტრიქონებისა და სვეტების მდებარეობის ურთიერთ-გაცვლით.

თუ $A = [a_{ij}]_{mn}$, მაშინ $A^T = [c_{ij}]_{nm}$, სადაც

$$c_{ij} = a_{ji} \quad (i = 1, \dots, m; j = 1, \dots, n)$$

მაგალითად,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

მატრიცების გამრავლება. ორი A და B მატრიცის გადამრავლება შესაძლებელია თუ ხდება შემდეგი პირობის შესრულება: A მატრიცის სვეტების რაოდენობა ტოლი უნდა იყოს B მატრიცის სტრიქონების რაოდენობისა.

A და B მატრიცების გადამრავლების შედეგად მიიღება ისეთი მესამე $m * k$ რიგითობის $C = c_{ij}$ მატრიცა, რომლის სტრიქონების რაოდენობა ტოლია A მატრიცის სტრიქონების რაოდენობისა, ხოლო სვეტების რაოდენობა კი ტოლია B მატრიცის სვეტების რაოდენობისა (იხ. ნახ. 9.19. ა,ბ).

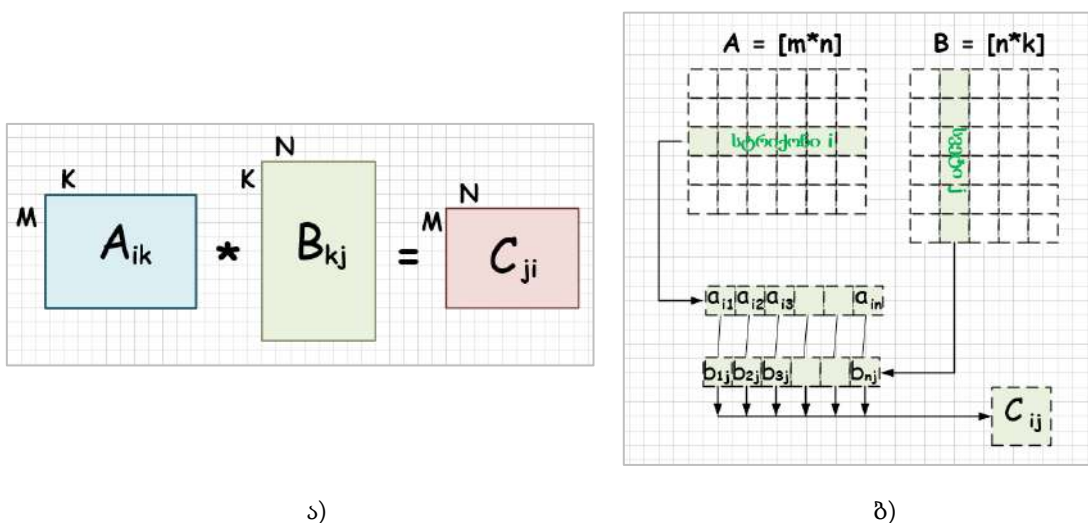
გადამრავლების შედეგად მიღებული C მატრიცის თითოეული c_{ij} ელემენტი გამოითვლება გარკვეული წესის დაცვით:

თუ მოცემულია $A = [a_{ij}]_{mn}$ და $B = [b_{ij}]_{np}$ ორი მატრიცა, მაშინ $A * B = [c_{ij}]_{mp}$,

სადაც

$$c_{ij} = \sum_{k=1}^n a_{ik} * b_{kj} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + a_{i3} * b_{3j} + \dots + a_{in} * b_{nj}$$

$$(i = 1, \dots, m; j = 1, \dots, p)$$



ა)

ბ)

ნახ. 9.19. ორი მატრიცის: ა) გადამრავლების ვიზუალიზაცია; ბ) თანაბარი ელემენტების კოორდინატების ინდექსების აღწერა

მაგალითი 9.1. კვადრატული მატრიცების გადამრავლება.

თანრიგი $A(2 \times 2) = B(2 \times 2)$.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

$$C = A * B = \begin{bmatrix} A_{11} * B_{11} + A_{12} * B_{21} & A_{11} * B_{12} + A_{12} * B_{22} \\ A_{21} * B_{11} + A_{22} * B_{21} & A_{21} * B_{12} + A_{22} * B_{22} \end{bmatrix} =$$

$$= \begin{bmatrix} 1 * 5 + 2 * 7 & 1 * 6 + 2 * 8 \\ 3 * 5 + 4 * 7 & 3 * 6 + 4 * 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

მაგალითი 9.2. მართკუთხა მატრიცების გადამრავლება. თანრიგი $A(3 \times 2)$ და $B(2 \times 3)$.

$$A = \begin{bmatrix} 2 & 1 \\ -3 & 0 \\ 4 & -1 \end{bmatrix} \quad B = \begin{bmatrix} 5 & -1 & 6 \\ -3 & 0 & 7 \end{bmatrix}$$

$$C = A * B = \begin{bmatrix} 2 & 1 \\ -3 & 0 \\ 4 & -1 \end{bmatrix} * \begin{bmatrix} 5 & -1 & 6 \\ -3 & 0 & 7 \end{bmatrix} = \begin{bmatrix} 7 & -2 & 19 \\ -15 & 3 & -18 \\ 23 & -4 & 17 \end{bmatrix}$$

ახლა მატრიცების გადამრავლება განვიხილოთ დაწვრილებით:

$$c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} = 2 \cdot 5 + 1 \cdot (-3) = 10 - 3 = 7$$

$$c_{12} = a_{11} \cdot b_{12} + a_{12} \cdot b_{22} = 2 \cdot (-1) + 1 \cdot 0 = -2 + 0 = -2$$

$$c_{13} = a_{11} \cdot b_{13} + a_{12} \cdot b_{23} = 2 \cdot 6 + 1 \cdot 7 = 12 + 7 = 19$$

$$c_{21} = a_{21} \cdot b_{11} + a_{22} \cdot b_{21} = (-3) \cdot 5 + 0 \cdot (-3) = -15 + 0 = -15$$

$$c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22} = (-3) \cdot (-1) + 0 \cdot 0 = 3 + 0 = 3$$

$$c_{23} = a_{21} \cdot b_{13} + a_{22} \cdot b_{23} = (-3) \cdot 6 + 0 \cdot 7 = -18 + 0 = -18$$

$$c_{31} = a_{31} \cdot b_{11} + a_{32} \cdot b_{21} = 4 \cdot 5 + (-1) \cdot (-3) = 20 + 3 = 23$$

$$c_{32} = a_{31} \cdot b_{12} + a_{32} \cdot b_{22} = 4 \cdot (-1) + (-1) \cdot 0 = -4 + 0 = -4$$

$$c_{33} = a_{31} \cdot b_{13} + a_{32} \cdot b_{23} = 4 \cdot 6 + (-1) \cdot 7 = 24 - 7 = 17$$

მატრიცის შეკრების და გამოკლების ოპერაციებს გააჩნიათ შემდეგი თვისებები:

1. მატრიცების შეკრება კომუტატურია ანუ მატრიცების შეკრება შესაძლებელია ნებისმიერი თანამიმდევრობით: $A + B = B + A$;

2. მატრიცების გადამრავლებისას კომუტაციას ადგილი არ აქვს;

3. მატრიცების შეკრების და გამრავლების შემთხვევაში ადგილი აქვს ასოციურობას ანუ ფრჩხილების თანამიმდევრობას მნიშვნელობა არ აქვს:

$$(AB)C = A(BC), (A + B) + C = A + (B + C);$$

მატრიცის რიცხვზე გამრავლების ოპერაციას აქვს შემდეგი თვისებები:

$$1A = A;$$

$$(rs)A = r(sA);$$

$$(r + s)A = rA + sA;$$

$$r(A + B) = rA + rB.$$

მატრიცის მატრიცაზე გამრავლების ოპერაციას აქვს შემდეგი თვისებები:

$$AE = EA = A, A0 = 0A = 0;$$

$$(AB)C = A(BC);$$

$$(A + B)C = AC + BC;$$

$$C(A + B) = CA + CB.$$

მატრიცის ტრანსპონირების ოპერაციას აქვს შემდეგი თვისებები:

$$(A^T)^T = A;$$

$$(A + B)^T = A^T + B^T;$$

$$(\alpha A)^T = \alpha A^T;$$

$$(A \cdot B)^T = B^T \cdot A^T.$$

მატრიცების გადამრავლების შტრასენის ალგორითმი. ორი მატრიცის (თანრიგით ორი ორზე) გადამრავლების კლასიკური მეთოდი იყენებს 8 გადამრავლებისა და 4 შეკრების ოპერაციას. ამდენად, $n * n$ თანრიგის მქონე ორი მატრიცის გადამრავლების კლასიკურ მეთოდს n^3 რაოდენობის გადამრავლება დასჭირდება (ნახ. 9.20.).

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

$$\begin{aligned} 1*6 + 2*8 &= 22 \\ 1*5 + 2*7 &= 19 \\ 3*5 + 4*7 &= 43 \\ 3*6 + 4*8 &= 50 \end{aligned}$$

8 გამრავლება
4 შეკრება

ნახ. 9.20. ორი მატრიცის (თანრიგი ორი ორზე) გადამრავლების კლასიკური ნიმუში

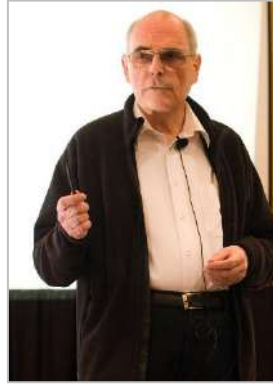
მატრიცის თანრიგის გაზრდასთან ერთად, გადამრავლებების რაოდენობა ბევრად უფრო სწრაფად იზრდება, ვიდრე შეკრებათა რაოდენობა.

ამდენად ორი ორზე თანრიგის მქონე მატრიცების გადასამრავლებლად საჭიროა სულ რაღაც 8 გადამრავლების ოპერაცია, ხოლო ოთხი ოთხზე თანრიგის მქონე მატრიცების გადასამრავლებლად საჭიროა 64 გამრავლების ოპერაცია, მაგრამ არსებობს ისეთი ალგორითმი, რომელიც იყენებს გადამრავლების არა 8, არამედ გადამრავლების 7 და შეკრების 18 ოპერაციას. ამ ალგორითმს ეწოდება მატრიცების გადამრავლების **შტრასენის ალგორითმი**, რომელიც აღმოაჩინა გერმანელმა მეცნიერმა ვოლკერ შტრასენმა 1969 წელს (იხ. ნახ. 9.21).

შტრასენის ალგორითმი არის პირველი ალგორითმი, რომელიც საშუალებას გვაძლევს, დიდი თანრიგის მქონე მატრიცები გადავრავლოთ $O(n^3)$ ნაკლებ დროში.

მაგალითად, მესამე თანრიგის მატრიცების გადასამრავლებლად დაგვჭირდება 3^3 -ზე ნაკლები გამრავლების ოპერაცია.

შტრასენი თავის ალგორითმში იყენებს „დაყავი და იზატონეს“ მეთოდს.



ნახ. 9.21. ვოლკერ შტრასენი (დაბ. 1936, 29 აპრილი), გერმანელი მათემატიკოსი დაუშვათ, რომ $\mathbf{R}$ არის წრე, ანუ შეკრებისა და გამრავლების ოპერაციები ასოციურია, შეკრება კომუტატურია, ელემენტები ნეიტრალურია, საპირისპირო ელემენტი შეკრებასთან მიმართებით (საპირისპირო ელემენტის გარეშე - ნახევარ წრე).

მატრიცების ჯამი $\mathbf{A} \in \mathbf{R}^{m \times n}$, $\mathbf{B} \in \mathbf{R}^{m \times n}$ - მატრიცა $\mathbf{A} + \mathbf{B} = \mathbf{C} \in \mathbf{R}^{n \times n}$.

$$c_{i,j} = a_{i,j} + b_{i,j}$$

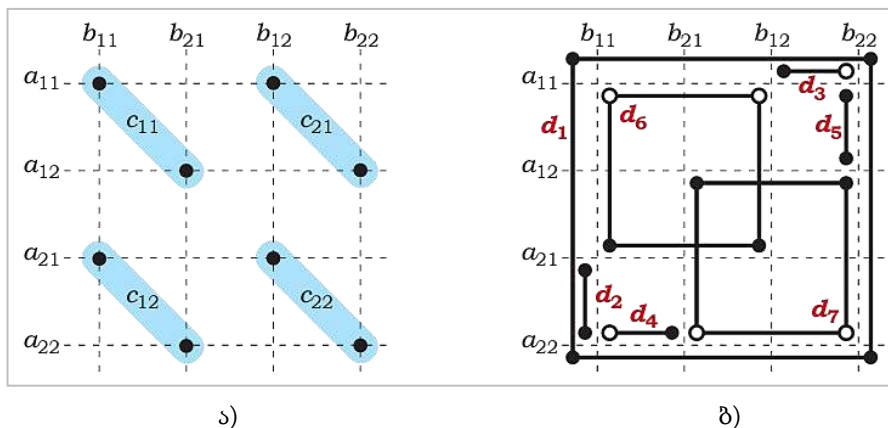
შეკრება არის ასოციური, კომუტატიური, არის ნეიტრალური და საპირისპირო ელემენტები. მატრიცების ნამრავლი $\mathbf{A} \in \mathbf{R}^{m \times l}$, $\mathbf{B} \in \mathbf{R}^{l \times n}$ - მატრიცა $\mathbf{A} * \mathbf{B} = \mathbf{C} \in \mathbf{R}^{m \times n}$.

$$c_{i,j} = \sum_{k=1}^l a_{i,k} * b_{k,j}$$

გამრავლება ასოციაციურია, არსებობს ნეიტრალური ელემენტი.

ლემა (შტრასენი, 1969). მოცემული გვაქვს 2×2 -ზე თანრიგის მქონე ორი $\mathbf{A}$ და $\mathbf{B}$ მატრიცა $\mathbf{R}$ წრის ირგვლივ: $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ და $\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$

მათი ნამრავლი $\mathbf{A} * \mathbf{B} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$ შესაძლებელია გამოვითვალოთ $\mathbf{R}$ წრის გარშემო 7 გამრავლებისა და 18 შეკრების ოპერაციების საშუალებით (ნახ. 9.22.ა,ბ).



ნახ. 9.22. ორი-ორზე თანრიგის მქონე ორი $\mathbf{A}$ და $\mathbf{B}$ მატრიცის გადამრავლების: ა) რვა ოპერაცია (კლასიკური მეთოდი); ბ) შვიდი ოპერაცია შტრასენის მიხედვით.

ნახატ 9.22.ა.-ზე ნაჩვენები სქემის მიხედვით **A** და **B** მატრიცის გადამრავლებისთვის გვჭირდება 8 გადამრავლება:

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21}$$

$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22}$$

$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21}$$

$$c_{22} = a_{21} * b_{12} + a_{22} * b_{22}$$

შტრასენის ალგორითმის მიხედვით გადამრავლების რვა ოპერაცია დაიყვანება გადამრავლების შვიდ ოპერაციამდე (ნახ. 9.22.ბ.).

ამდენად შტრასენის ალგორითმის შესაბამისი ფორმულა შეიძლება ასე გამოვსახოთ:

$$C = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} (d_1 + d_4 - d_5 + d_7) & (d_3 + d_5) \\ (d_2 + d_4) & (d_1 - d_2 + d_3 + d_6) \end{bmatrix},$$

სადაც

$$d_1 = (a_{11} + a_{22}) * (b_{11} + b_{22}),$$

$$d_2 = (a_{21} + a_{22}) * b_{11},$$

$$d_3 = a_{11} * (b_{12} - b_{22}),$$

$$d_4 = a_{22} * (b_{21} - b_{11}),$$

$$d_5 = (a_{11} + a_{12}) * b_{22},$$

$$d_6 = (a_{21} - a_{11}) * (b_{11} + b_{12}),$$

$$d_7 = (a_{12} - a_{22}) * (b_{21} + b_{22}).$$

მიღებული შვიდი მნიშვნელობების დახმარებით კი გამოვითვლით ნამრავლს:

$$c_{11} = d_1 + d_4 + d_7 - d_5$$

$$c_{12} = d_3 + d_5$$

$$c_{21} = d_2 + d_4$$

$$c_{22} = d_1 + d_3 + d_6 - d_2$$

მატრიცის გადამრავლების ტრადიციულ (კლასიკურ) ალგორითმს სჭირდება $O(n^3) = O(n^{\log_2 8})$ ოპერაცია: თითოეულ $O(n^2)$ მნიშვნელობის მისაღებად საჭიროა $O(n)$ ოპერაცია.

მატრიცის გადამრავლების კლასიკურ ალგორითმის $T(n)$ მუშაობის დროა:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2) = O(n^{\log_2 8})$$

მატრიცების გადამრავლების შტრასენის ალგორითმს კი სჭირდება $O(n^{2.81}) = O(n^{\log_2 7})$ ოპერაცია. მატრიცის გადამრავლების შტრასენის ალგორითმის $T(n)$ მუშაობის დროა:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2) = O(n^{\log_2 7})$$

ამ ორი ალგორითმის მუშაობის დროითი განსხვავება საკმაოდ საგრძნობი ხდება დიდი თანრიგის მატრიცებთან მიმართებით.

პრაქტიკული თვალსაზრისით მატრიცების გადამრავლების შტრასენის ალგორითმი იძლევა საკმაოდ დიდ მოგებას მაღალი თანრიგის (64×64 -ზე და უფრო მაღალი) მქონე მატრიცებთან მიმართებით.

მატრიცების შებრუნება. მათემატიკაში არსებობს შებრუნებული რიცხვის ცნება. როგორ მიიღება შებრუნებული რიცხვი: ავიღებთ რაიმე რიცხვს, მას მივუწერთ უარყოფით ხარისხს და მივიღებთ შებრუნებულ რიცხვს.

მაგალითად,

7 და 7^{-1} - ურთიერთშებრუნებული რიცხვებია;

7 და $1/7$ - ეს რიცხვებიც ურთიერთშებრუნებულია, მხოლოდ ჩაწერილია სხვა ფორმით.

ურთიერთშებრუნებული რიცხვები ერთმანეთზე გამრავლებისას ყოველთვის იძლევა ერთს: $7 * 7^{-1} = 7 * 1/7 = 7/7 = 1$

წრფივ ალგებრაში კი გვაქვს შებრუნებული მატრიცის ცნება. შებრუნებულ მატრიცას გააჩნია ურთიერთსაპირისპირო რიცხვების მსგავსი თვისებები. თუ ჩვეულებრივ მატრიცას გაამრავლებთ მის შებრუნებულ მატრიცაზე, მივიღებთ ერთეულოვან მატრიცას.

შებრუნებული მატრიცა აღინიშნება A^{-1} სახით.

ურთიერთშებრუნებული მატრიცები კი შეგვიძლია ასე წარმოვადგინოთ: A და A^{-1} .

თუ გადავამრავებთ ურთიერთშებრუნებულ მატრიცებს, მაშინ მივიღებთ ერთეულოვან (E) მატრიცას: $A * A^{-1} = E$.

ერთეულოვანი მატრიცის და ჩვეულებრივი მატრიცის დამოკიდებულება არის შემდეგი სახის: $A * E = A$ ან $E * A = A$

შებრუნებული მატრიცა არის მხოლოდ კვადრატული მატრიცა ანუ შებრუნებული მატრიცის თანრიგი არის $n * n$ -ზე. შებრუნებული მატრიცის დეტერმინანტი არ უდრის ნულს.

დეტერმინანტი. A მატრიცის დეტერმინანტი აღინიშნება $|A|$ სახით. თუ მატრიცა არის $n * n = 2$ თანრიგის, მაშინ მატრიცის დეტერმინანტი გამოითვლება შემდეგნაირად:

$$|A| = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} = \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix}$$

$$|A| = A_{11} * A_{22} - A_{12} * A_{21} = 1 * 4 - 3 * 2 = -2$$

შებრუნებული მატრიცის თვისებები:

$$(A^{-1})^{-1} = A$$

$$(\gamma * A)^{-1} = \frac{1}{\gamma} * A^{-1}$$

$$(A * B)^{-1} = B^{-1} * A^{-1}$$

$$(A^{-1})^T = (A^T)^{-1}$$

მაგალითი 3. მოცემულია მატრიცა $A = \begin{bmatrix} 7 & 4 \\ 5 & 3 \end{bmatrix}$. ვიპოვოთ მისი შებრუნებული A^{-1} მატრიცა.

ამოხსნა:

1. თავდაპირველად ვიპოვოთ დეტერმინანტი: $|A| = 7 * 3 - 5 * 4 = 1 \neq 0$ ე.ი. მოცემულ მატრიცას აქვს შებრუნებული მატრიცა.

2. მოცემულ A მატრიცას მარჯვნივ მივუწეროთ მეორე თანრიგის ერთეულოვანი მატრიცა:

$$[A | E] = \left[\begin{array}{cc|cc} 7 & 4 & 1 & 0 \\ 5 & 3 & 0 & 1 \end{array} \right]$$

3. ჩვენი მიზანია სტრიქონებზე ელემენტარული ალგებრული გამოთვლების ჩატარებით გავიღეთ შემდეგ შედეგზე: $[E | A^{-1}]$

4. პირველი სტრიქონის ელემენტებს გამოვაკლოთ მეორე სტრიქონის ელემენტები. მივიღებთ:

$$[A | E] = \left[\begin{array}{cc|cc} 2 & 1 & 1 & -1 \\ 5 & 3 & 0 & 1 \end{array} \right]$$

5. მეორე სტრიქონს გამოვაკლოთ 2-ჯერ პირველი სტრიქონი:

$$[A | E] \sim \left[\begin{array}{cc|cc} 2 & 1 & 1 & -1 \\ 1 & 1 & -2 & 3 \end{array} \right]$$

6. პირველ და მეორე სტრიქონებს შევუცვალოთ ადგილები:

$$[A | E] \sim \left[\begin{array}{cc|cc} 1 & 1 & -2 & 3 \\ 2 & 1 & 1 & -1 \end{array} \right]$$

7. მეორე სტრიქონს გამოვაკლოთ 2-ჯერ პირველი სტრიქონი:

$$[A | E] \sim \left[\begin{array}{cc|cc} 1 & 1 & -2 & 3 \\ 0 & -1 & 5 & -7 \end{array} \right]$$

8. მეორე სტრიქონი გავამრავლოთ $[-1]$ -ზე, ხოლო პირველ სტრიქონს მივუმატოთ მეორე სტრიქონი:

$$[A | E] \sim \left[\begin{array}{cc|cc} 1 & 0 & 3 & -4 \\ 0 & 1 & -5 & 7 \end{array} \right]$$

9. მარცხენა მხარეს მივიღეთ ერთეულოვანი მატრიცა, ხოლო მარჯვენა მხარეს მყოფი მატრიცა კი წარმოადგენს საწყისი მატრიცის შებრუნებული მატრიცას: $A^{-1} = \begin{bmatrix} 3 & -4 \\ -5 & 7 \end{bmatrix}$

პასუხი: $A^{-1} = \begin{bmatrix} 3 & -4 \\ -5 & 7 \end{bmatrix}$

კომბინატორიკის ელემენტები. შესაძლო კომბინაციების სრული რაოდენობის დათვლის მეთოდები, ხელსაყრელი კომბინაციების დათვლა, ალბათობის გამოთვლის ელემენტები. კომბინატორული ამოცანები. პოლინომიალური ამოცანების ამოხსნის ალგორითმები, ეფექტური ამოხსნები. ექსპონენციალური ამოცანების ამოხსნის ალგორითმები, პრობლემები.

კომბინატორიკის ელემენტები. კომბინატორიკა მათემატიკის ერთ-ერთი დარგია და სწავლობს სიმრავლის ელემენტების შერჩევას ან განლაგებას მოცემული წესების შესაბამისად. ტერმინი „კომბინატორიკა“ დაამკვიდრა გოტფრიდ ვილჰელმ ლაიბნიცმა, რომელმაც იგი გამოიყენა 1666 წელს გამოქვეყნებულ ნაშრომში „თხზულება კომბინატორიკის ხელოვნების შესახებ“.

ხშირად კომბინატორიკის ნაწილად მოიხსენიებენ გრაფების თეორიასაც. კომბინატორიკა მომდინარეობს ლათინური სიტყვისაგან „combina“, რომელიც ნიშნავს დაკავშირებას.

გოტფრიდ ვილჰელმ ლაიბნიცი (1646-1716) - გამოჩენილი გერმანელი ფილოსოფოსი, ფართო თვალსაწიერის მქონე უნივერსალური სწავლული და მეცნიერი. ლაიბნიცმა დიდი წვლილი შეიტანა ლოგიკის, მათემატიკის, სამართალის, ფიზიკის, ისტორიის, თეოლოგიის, ენის თეორიის, ენის ფილოსოფიის და მეცნიერების სხვადასხვა დარგის განვითარებაში.



ნახ. 10.1. გოტფრიდ ვილჰელმ ლაიბნიცი (1646-1716 წ.წ.)

კომბინატორიკა მოიცავს სხვადასხვა სახის კავშირებს, რომელთა მიღებაც შესაძლებელია გარკვეული სასრული სიმრავლის ელემენტების გამოყენებით.

კომბინატორიკაში გვაქვს შემდეგი კავშირები: ა) გადანაცვლება, გადანაცვლებადობა; ბ) წყობა, წყობადობა; გ) ჯუფთება, ჯუფთებადობა.

კომბინატორიკის ძირითადი პრინციპი შემდეგში მდგომარეობს:

თანამიმდევრობით უნდა შესრულდეს k -ური მოქმედება.

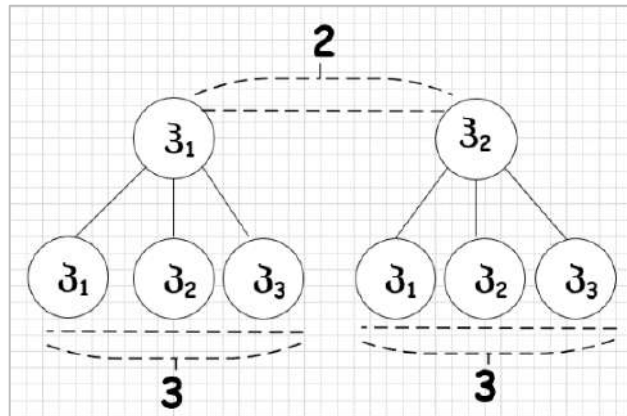
თუ პირველი მოქმედება შეიძლება შევასრულოთ n_1 , მეორე n_2 , მესამე n_3 ..., მაშინ k -ური მოქმედება შეიძლება შევასრულოთ n_k განსხვავებული წესით. ამდენად მთლიანობაში k -ური მოქმედება შეიძლება შესრულდეს $n_1 * n_2 * n_3 * ... * n_k$ განსხვავებული წესით.

კომბინატორიკის მარტივი სქემები. სასწავლო მიზნით სასარგებლოა კომბინატორიკის მარტივი სქემების მიმოხილვა ხისებრი დიაგრამის (დენდოგრამის) გამოყენებით.

მაგალითი 10.1. ლუკას აქვს 3 პერანგი და 2 ჰალსტუხი. დავადგინოთ ლუკას პერანგის და ჰალსტუხის არჩევის რამდენიარის შესაძლებლობა აქვს (ნახ. 10.2.)?

ამოხსნა.

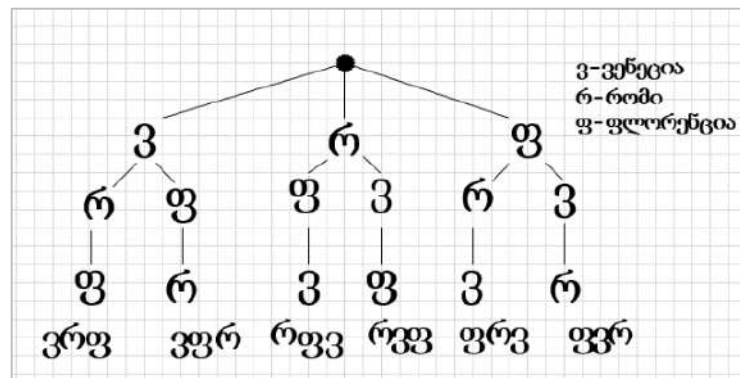
ლუკას აქვს ჰალსტუხის არჩევის 2 ვარიანტი (2 განსხვავებული წესი) და ყოველ ვარიანტთან პერანგის შერჩევის 3 ვარიანტი (3 განსხვავებული წესი) ე. ი. ლუკას სულ $2 * 3 = 6$ არჩევანი აქვს.



ნახ. 10.2. პერანგის და ჰალსტუხის არჩევის ვარიანტები

მაგალითი 10.2. ტურისტული ფირმა მსურველებს სთავაზობს მოგზაურობის ტურს იტალიის სამ ქალაქში შემდეგი თანამიმდევრობით: ვენეცია, რომი და ფლორენცია.

ქალაქების განლაგების მარშრუტის რამდენი ვარიანტი არსებობს (ნახ. 10.3.).

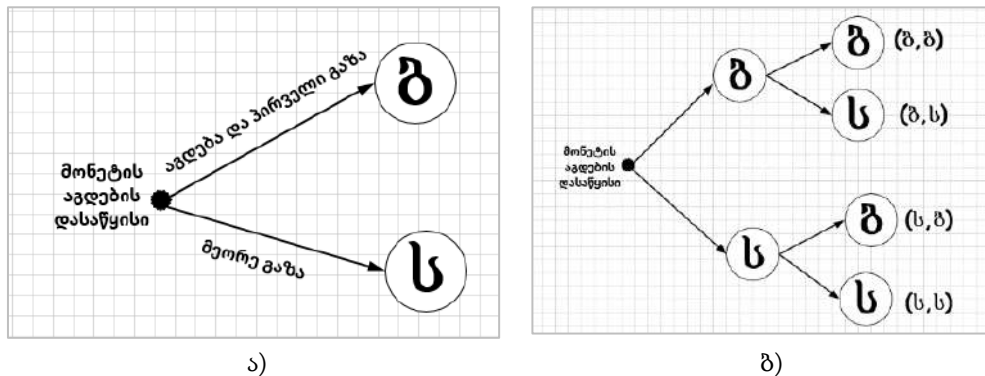


ნახ. 10.3. იტალიის ქალაქების მარშრუტის ვარიანტები

მაგალითი 10.3. გვაქვს ვერცხლის ორი მონეტა. დავადგინოთ ერთი მონეტის და ორი მონეტის სიბრტყეზე დაგდებისას შესაძლო შედეგების სიმრავლე. მონეტის ზედა მხარე ბორჯღალი (ზოგჯერ უწოდებენ გერბს) აღვნიშნოთ **ბ** სიმბოლოთი, ხოლო მონეტის ქვედა მხარე საფასური აღვნიშნოთ **ს** სიმბოლოთი (იხ. ნახ. 10.4.ა,ბ.).

განვიხილოთ:

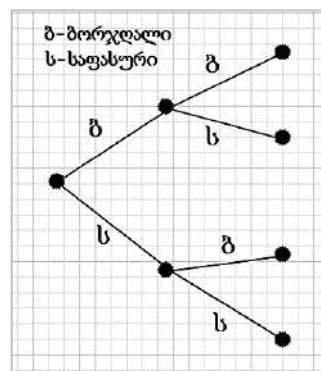
1. ერთი მონეტის აგდების შესაძლო ვარიანტები;
2. ორი მონეტის (ან ერთი მონეტის ორჯერ აგდების) აგდების შესაძლო ვარიანტები.



ნახ. 10.4. მონეტის სიბრტყეზე დაგდების: ა) ერთი მონეტის შემთხვევაში გვაქვს ორი მოსალოდნელი შედეგი: {ბ,ს}; ბ) ორი მონეტის შემთხვევაში გვაქვს

ოთხი მოსალოდნელი შედეგი: {{ბ,ბ}, {ბ,ს}, {ს,ბ}, {ს,ს}}

დენდროგრამის დახმარებით გამოვსახოთ ვერცხლის მონეტის ორჯერ აგდების შესაბამისი შედეგების სიმრავლე (ნახ. 10.5.).



ნახ. 10.5. დენდროგრამით გამოსახული მონეტის ორჯერ აგდების სიმრავლე

ახლა კი ჩამოვყალიბოთ კომბინატორიკული ანალიზის ორი ძირითადი წესი: ნამრავლისა და ჯამის წესები.

კომბინატორიკის ჯამის წესი. მოცემულია A და B სიმრავლეები. A სიმრავლე შედგება n ელემენტისაგან ანუ არსებობს A სიმრავლის ელემენტთა შერჩევის n შესაძლებლობა, რომელსაც პირობითად აღვნიშნავთ $n(A)$ სიმბოლოთი, B სიმრავლე შედგება m ელემენტისაგან ანუ არსებობს B სიმრავლის ელემენტთა შერჩევის m შესაძლებლობა, რომელსაც პირობითად აღვნიშნავთ $n(B)$ სიმბოლოთი. ელემენტების შერჩევის რაოდენობა ან A ან B სიმრავლიდან იქნება: $n(A \cup B) = n(A) + n(B)$, როცა მოცემული A და B სიმრავლეები არათანამკვეთი სიმრავლეებია ($A \cap B = \emptyset$) ანუ მათ საერთო ელემენტი არ გააჩნიათ. ელემენტთა შერჩევის რაოდენობა A ან B სიმრავლიდან $n(A \cup B) = n(A) + n(B) - n(A \cap B)$, როცა მოცემული A

და B სიმრავლეები თანმკვეთი სიმრავლეებია ($A \cap B \neq \emptyset$) ანუ მათ აქვთ საერთო ელემენტები.

კომბინატორიკის ნამრავლის წესი. ვთქვათ მოცემულია ორი, ესა თუ ის ობიექტი ან თუნდაც ორი სახის გარკვეული ქმედება. ამასთან პირველის შერჩევის ინდივიდუალური რაოდენობა იყოს n , ხოლო მეორის კი m , მაშინ პირველი და მეორე ობიექტის (თუ ქმედების) თანამიმდევრობით შერჩევის საერთო რაოდენობა იქნება ($m \cdot n$) ანუ კომბინატორიკის ნამრავლის წესის გამოყენების დროს ვითვლით ცალკეულ კომბინაციათა ინდივიდუალურ რაოდენობებს და ვამრავლებთ ერთიმეორეზე.

ახლა კი განვიხილოთ კომბინატორიკის სამი კავშირი:

გადანაცვლებადობა. თავდაპირველად გადანაცვლების უკეთ შესასწავლად განვიხილოთ მარტივი მაგალითი.

პატარა ლუკა სასკოლო არდადეგებს ატარებს კახეთის ერთ-ერთ ლამაზ სოფელში. დილით ბებიამ ლუკას მიაწოდა ვაშლი, ატამი და ყურძენი.

ვაშლის, ატმისა და ყურძნის დასაგემოვნებლად ხილის არჩევის თანამიმდევრობათა რამდენი განსხვავებული ვარიანტი (რიგითობა) აქვს ლუკას.

ამოხსნა:

ვაშლი $\rightarrow$ ატამი $\rightarrow$ ყურძენი;

ვაშლი $\rightarrow$ ყურძენი $\rightarrow$ ატამი;

ატამი $\rightarrow$ ყურძენი $\rightarrow$ ვაშლი;

ატამი $\rightarrow$ ვაშლი $\rightarrow$ ყურძენი;

ყურძენი $\rightarrow$ ვაშლი $\rightarrow$ ატამი;

ყურძენი $\rightarrow$ ატამი $\rightarrow$ ვაშლი;

პასუხი: ლუკას აქვს ხილის არჩევის თანამიმდევრობის 6 ვარიანტი.

როდესაც მხოლოდ სამი ელემენტი გვაქვს, მაშინ შეგვიძლია უბრალოდ ჩამოვწეროთ მათი გადანაცვლებების ვარიანტები, მაგრამ თუ გვექნება ხუთი და მეტი, მაგალითად, ათი ან ასი ელემენტი, მაშინ ყველა გადანაცვლებების ჩამოწერა უბრალოდ შეუძლებელი ხდება.

ელემენტების სიმრავლის გადანაცვლებადობა წარმოადგენს ამ სიმრავლის ყველა n რაოდენობის ელემენტის ნებისმიერ მოწესრიგებულ ნაკრებს.

სხვანაირად რომ ვთქვათ გადანაცვლება არის $1, 2, 3, \dots, n$ ნატურალური რიცხვების დალაგება რიცხვების გამეორების გარეშე.

გადანაცვლებათა რაოდენობა აღინიშნება P_n სიმბოლოთი.

ახლა დავადგინოთ სხვადასხვა n რაოდენობის მქონე ელემენტებისთვის გადანაცვლების რიცხვის პოვნის საერთო ფორმულა.

ორი ელემენტისთვის გვექნება 2 გადანაცვლების ვარიანტი:

12 21

სამი ელემენტისთვის გვექნება 6 გადანაცვლების ვარიანტი:

123 213 312

132 231 321

ოთხი ელემენტისთვის გვექნება 24 გადანაცვლების ვარიანტი:

1234 2134 3124 4123

1243 2143 3142 4132

1324 2314 3214 4213

1342 2341 3241 4231

1423 2413 3412 4312

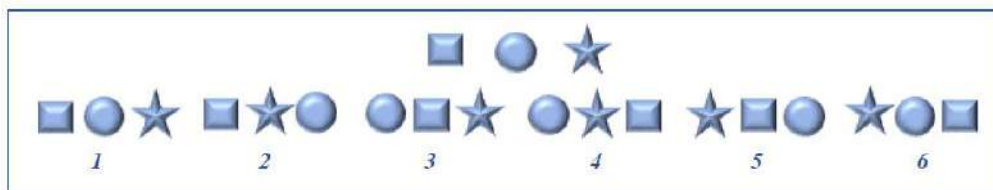
1432 2431 3421 4321

დაისმის შეკითხვა: n რაოდენობის მქონე ელემენტთა გადანაცვლების სულ რამდენი ვარიანტი არსებობს? მოდით დავითვალოთ: პირველი ადგილზე შეგვიძლია დავწეროთ ნებისმიერი ელემენტი n რაოდენობის მქონე ელემენტების სიმრავლიდან, მეორე ადგილზე კი მხოლოდ $n - 1$ რაოდენობის მქონე ელემენტი (ნებისმიერი, გარდა იმისა, რომ უკვე პირველ ადგილზეა), მესამე ადგილზე - $n - 2, \dots$, ბოლოს წინა ადგილზე ერთი ვარიანტი ორი დარჩენილი ვარიანტიდან და ბოლო ადგილზე შეიძლება ჩავწეროთ მხოლოდ ბოლო ელემენტი.

საბოლოოდ მივიღეთ გადანაცვლებადობათა რაოდენობის გამოსათვლელი ფორმულა: $P_n = n * (n - 1) * (n - 2) * \dots * 2 * 1 = n!$

ამდენად, n რაოდენობის ელემენტის მქონე სიმრავლის (ელემენტების) ყველა შესაძლო გადანაცვლებათა რიცხვია (ნახ. 10.6):

$$P_n = n!$$



ნახ. 10.6. გადანაცვლებადობა

მაგალითი 10.4. რამდენი განსხვავებული სია შეიძლება შევადგინოთ 7 სხვადასხვა გვარისგან?

ამოხსნა: $P_7 = 7! = 2 * 3 * 4 * 5 * 6 * 7 = 5040$

მაგალითი 10.5. დავადგინოთ, რამდენ ელემენტს უნდა შეიცავდეს სიმრავლე, რომ ამ სიმრავლის ელემენტებისაგან შედგენილი ყველა გადანაცვლებათა რიცხვი იყოს:

ა) არა უმეტეს 1000-ისა; ბ) არა ნაკლებ 500-ისა.

ამოხსნა:

ა) ამოცანის პირობით $P_n \leq 1000$ ანუ $n! \leq 1000$ საიდანაც $n \leq 6$. მართლაც, $6! = 720 \leq 1000$ და $7! = 5040 \geq 1000$

ბ) ამოცანის პირობით, $P_n \geq 500$ ანუ $n! \geq 500$ საიდანაც $n \geq 6$. მართლაც, $5! = 120 \leq 500$ და $6! = 720 \geq 500$

მაგალითი 10.6. მასწავლებელმა სტუდენტებთან (სტუდენტთა რაოდენობა = 4) ჩაატარა ტესტირება და გადაწყვიტა, რომ სტუდენტებმა თვითონ შეაფასონ ერთმანეთის ნამუშევრები. ცხადია, რომ მასწავლებელს არ უნდა რომელიმე სტუდენტს საკუთარი ნაშრომი შეხვდეს გასასწორებლად. შესაძლებელი ტესტების დარიგების (გადანაცვლებების) რაოდენობა არის $4! = 24$.

კითხვა: ტესტების დარიგების რამდენი ისეთი გზა არსებობს, რომ არც ერთ სტუდენტ არ შეხვდეს (ნახოს/მიიღოს) საკუთარი ნაშრომი? (ნახ. 10.7.).

ABCD	ABDC	ACBD	ACDB	ADBC	ADCB
BACD	BADC	BCAD	BCDA	BDAC	BDCA
CABD	CADB	CBAD	CBDA	CDAB	CDBA
DABC	DACB	DBAC	DBCA	DCAB	DCBA

ნახ. 10.7. ტესტების დარიგების შესაძლებელი 24 გადანაცვლებებიდან

აღმოჩნდა ისეთი 9 გადანაცვლება, როდესაც არც ერთ სტუდენტს საკუთარი ნაშრომი არ შეხვდება

პასუხი: არსებობს მხოლოდ 9 გადანაცვლება (აღნიშნულია ლურჯი ფერით). დანარჩენ სხვა გადანაცვლებებში არსებობს ერთი ისეთი სტუდენტი მაინც, რომელსაც გასასწორებლად შეხვდება საკუთარი ნაშრომი, რაც მოცემულობის მიხედვით აკრძალულია.

წყობა. n რაოდენობის მქონე ელემენტებიანი სირავლის ელემენტებისგან შედგენილ ყოველ m ელემენტიან დალაგებულ ქვესიმრავლეს ეწოდება წყობა. სხვა სიტყვებით, წყობა არის n ელემენტიანი სიმრავლის m ელემენტიანი დალაგებული ქვესიმრავლეთა რაოდენობა, ამასთან $0 \leq m \leq n$. განასხვავებენ წყობას გამეორების გარეშე და გამეორებით (იხ. ნახ. 10.8.ა,ბ). გამეორების გარეშე წყობა გამოითვლება ფორმულით:

$$A_n^m = \frac{n!}{(n-m)!}$$

სადაც m არის წყობის ელემენტთა რაოდენობა, ხოლო n - ელემენტების საერთო რაოდენობაა.

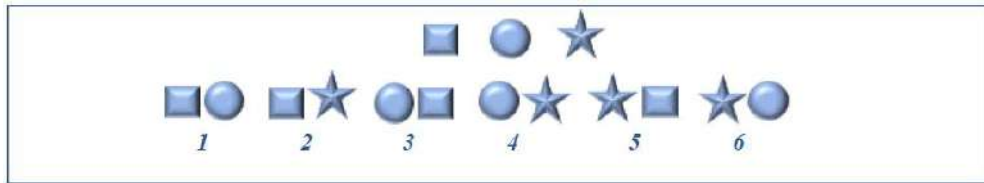
წყობას გამეორებებით აქვს შემდეგი სახე: $A_n^m = n^m$

წყობათა თვისებები:

პირველი თვისება. $A_n^0 = 1$

მეორე თვისება. $A_n^1 = n$

მესამე თვისება. $A_n^n = P_n = n!$



ა)



ბ)

ნახ. 10.8 წყობა: ა) გამეორებს გარეშე; ბ) გამეორებებით

წყობას შეგვიძლია შევხედოთ, როგორც n რაოდენობის მქონე ელემენტებიდან m ელემენტთა შერჩევის ამოცანას. არსებობს პირველი ელემენტის არჩევის n შესაძლებლობა, პირველი ელემენტის არჩევის შემდეგ არსებობს მეორე ელემენტის არჩევის $n - 1$ შესაძლებლობა და ა. შ., მე- m -ური ელემენტის არჩევის $n - m + 1$ შესაძლებლობა, ამიტომ გამრავლების პრინციპის საფუძველზე გვაქვს:

$$A_n^m = n * (n - 1) * (n - 2) \dots (n - m + 1) = \frac{n!}{(n - m)!}$$

მაგალითი 10.7. სპორტულ შეჯიბრებაში მონაწილეობს 10 სპორტსმენი. შეჯიბრებაში მონაწილე 10 სპორტსმენიდან პირველ სამ ადგილზე გასული სამი გამარჯვებული სპორტსმენი რამდენ ცალ სხვადასხვანაირ ვარიანტად შეიძლება განლაგდეს დასაჯილდოებელ კვარცხლბეკზე?

ამოხსნა:

$$A_{10}^3 = \frac{10!}{(10 - 3)!} = \frac{10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2}{7 * 6 * 5 * 4 * 3 * 2} = 10 * 9 * 8 = 720$$

მაგალითი 10.8. დავადგინოთ, რამდენი სხვადასხვა ხერხით შეიძლება 5 კაცის არჩევა ხუთ სხვადასხვა თანამდებობაზე, თუ ამ თანამდებობაზე კანდიდატთა რიცხვი 11-ია.

ამოხსნა: ამორჩევათა ყველა შესაძლო რაოდენობა ტოლი იქნება A_{11}^5 რიცხვის.

ამიტომ,

$$A_{11}^5 = \frac{11!}{(11 - 5)!} = \frac{11!}{6!} = 7 * 8 * 9 * 10 * 11 = 55440$$

მაგალითი 10.9. მოცემული გვაქვს სიტყვა - **ფანდური**. დავადგინოთ სიტყვა **ფანდურის** ასო-ნიშნების გამოყენებით რამდენი ცალი სხვადასხვა სამ-ასო-ნიშნის სიტ-

ყვის მიღება არის შესაძლებელი (არ არის აუცილებელი სამ-ასონიშნიან სიტყვებს რაიმე აზრი ჰქონდეთ).

ამოხსნა: სიტყვაში **ფანდური** გვაქვს 8 ცალი განსხვავებული ასონიშანი/სიმბოლო. მოცემულობიდან გამომდინარე, შეგვიძლია გამოვიყენოთ სამი ცალი ასონიშანი, რომელთა სხვადასხვა რიგითობით დალაგება მოგვცემს სხვადასხვა სიტყვას. ასეთი სიტყვების რაოდენობა იქნება:

$$A_8^3 = \frac{8!}{(8-3)!} = \frac{8!}{5!} = 336.$$

პასუხი: 336 ცალი განსხვავებული სიტყვა.

ჯუფდება. ჯუფდება არის n რაოდენობის მქონე სიმრავლის ელემენტებიდან შედგენილი დაულაგებელი m ელემენტისანი ქვესიმრავლეები ანუ ისეთი ერთობლიობები, რომლებიც ერთმანეთისაგან განსხვავდება მხოლოდ ელემენტების შემადგენლობით (იხ. ნახ. 10.9). ჯუფდებადობის რიცხვი ტოლია:

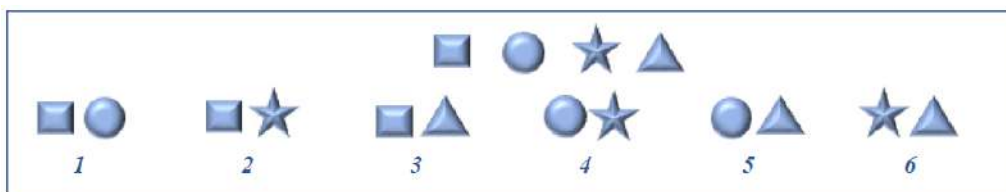
$$C_n^m = \frac{n!}{m!(n-m)!}$$

ვინაიდან, n რაოდენობის მქონე ელემენტის სიმრავლის ყველა m ელემენტისანი დალაგებული ქვესიმრავლის მისაღებად შეგვიძლია ავიღოთ n ელემენტისანი სიმრავლის ყველა დაულაგებელი m ელემენტისანი ქვესიმრავლე (რომელთა რაოდენობაა C_n^m) და თითოეულთან შევასრულებთ ყველანაირ გადანაცვლებებს (ამის შესაძლებლობაა $m!$), მაშინ ადგილი ექნება შემდეგ თანაფარდობას:

$$C_n^m * m! = A_n^m$$

აქედან, ცხადია, რომ

$$A_n^m = \frac{C_n^m}{m!} = \frac{n!}{m!(n-m)!}$$



ნახ. 10.9. ჯუფდება

ჯუფდებადობის თვისებები:

პირველი თვისება. $C_n^0 = 1$

მეორე თვისება. $C_n^1 = n$

მესამე თვისება. $C_n^n = C_n^{n-m}$

ვთქვათ, მოცემული გვაქვს მიმდებრობა 1, 2, 3, 4. ამ თანამიმდევრობებისთვის გვაქვს 6 გადაადგილება 2 ჯუფთებადობით:

- 1) 1, 2;
- 2) 1, 3;
- 3) 1, 4;
- 4) 2, 3;
- 5) 2, 4;
- 6) 3, 4;

მაგალითი 10.10. შესარჩევ შეჯიბრებაში მონაწილეობს 10 ადამიანი, რომელთაგან ფინალში გავიდა სამი სპორტსმენი. ფინალისტების რამდენი განსხვავებული სამეული შეიძლება გამოვლინდეს?

ამოხსნა. წინა მაგალითისაგან განსხვავებით, აქ ფინალისტების რიგითობას (დალაგებას) მნიშვნელობა არა აქვს. ამიტომ ვიყენებთ ჯუფდების ფორმულას. შესაბამისად, საძიებელი რიცხვია:

$$C_{10}^3 = \frac{10!}{3! * 7!} = \frac{8 * 9 * 10}{6} = 120$$

მაგალითი 10.11. დავადგინოთ საჭადრაკო ტურნირში მონაწილეთა რაოდენობა, თუ თითოეულმა მონაწილემ თითოეულ დანარჩენ მონაწილესთან თითო პარტია ითამაშა და სულ 55 პარტია შედგა?

ამოხსნა. თუ ტურნირში მონაწილეთა რაოდენობას n -ით აღვნიშნავთ, მაშინ ტურნირში სულ გათამაშებულ პარტიათა რაოდენობა ტოლი იქნება C_n^2 .

ამოცანის პირობით $C_n^2 = 55$ გვექნება

$$\frac{n!}{2! * (n-2)!} = \frac{n * (n-1)}{2} = 55$$

ანუ $n * (n-1) = 110$ საიდანაც $n = 11$.

კომბინატორიკა გამოიყენება მრავალწევრთა ალგებრაშიც, მაგალითად, ნიუტონის ბინომის ფორმულაში.

ნიუტონის ბინომი. ბინომური კოეფიციენტების გამოსათვლელი ფორმულა და მათი თვისებები. განვიხილოთ ორი რიცხვის ჯამის ნატურალურ მაჩვენებლიანი ხარისხი $(a+b)^n$. ნატურალურ მაჩვენებლიანი ხარისხის განმარტების მიხედვით შეგვიძლია გამოვითვალოთ:

$$\begin{aligned} (a+b)^0 &= 1 \\ (a+b)^1 &= a+b \\ (a+b)^2 &= a^2 + 2ab + b^2 \\ (a+b)^3 &= a^3 + 3a^2b + 3ab^2 + b^3 \\ (a+b)^4 &= a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4 \\ (a+b)^5 &= a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5 \end{aligned}$$

აღმოჩნდა, რომ ნატურალურ მაჩვენებლიანი n ხარისხისთვის მართებულია შემდეგი ფორმულები:

$$(a + b)^n = C_n^0 * a^n + C_n^1 * a^{n-1}b + C_n^2 * a^{n-2}b^2 + \dots + C_n^m * a^{n-m}b^m + \dots + C_n^n * b^n$$

$$(a - b)^n = C_n^0 * a^n - C_n^1 * a^{n-1}b + \dots + (-1)^k C_n^k * a^{n-k}b^k + \dots + (-1)^n C_n^n * b^n$$

მიღებულ ფორმულას ნიუტონის ფორმულა იგივე ნიუტონის ბინომი ეწოდება.

ნიუტონის ბინომის C_n^m კოეფიციენტებს ბინომური კოეფიციენტები ეწოდებათ.

ნიუტონის ბინომის თვისებები:

პირველი თვისება. ბინომური კოეფიციენტების რაოდენობა (შესაბამისად, ნიუტონის ბინომის ფორმულაში შესაკრებთა რიცხვი) ტოლია $(n + 1)$ -ის;

მეორე თვისება. ნიუტონის ბინომის ფორმულაში პირველი შესაკრების ხარისხი კლებულობს n -დან 0 -მდე, ხოლო მეორე შესაკრების ხარისხი იზრდება 0 -დან n -მდე;

მესამე თვისება. ნიუტონის ბინომის ფორმულაში ბოლოებიდან თანაბრად დაშორებული წევრების კოეფიციენტები ტოლია $C_n^m = C_n^{n-m}$;

მეოთხე თვისება. ბინომური კოეფიციენტები ჯერ იზრდება, შემდეგ კი კლებულობს. თუ ბინომის ხარისხის მაჩვენებელი ლუწია, მაშინ დაშლის შუა წევრის ბინომური კოეფიციენტი უდიდესია, ხოლო თუ ბინომური ხარისხის მაჩვენებელი კენტია, მაშინ ორი შუა ბინომური კოეფიციენტი ერთმანეთის ტოლია და ამასთან უდიდესია;

მეხუთე თვისება. ბინომური ხარისხის დაშლის $k + 1$ შესაკრები T_k ტოლია:

$$T_k = C_n^k * a^{n-k}b^k$$

პასკალის სამკუთხედი. პასკალის სამკუთხედი წარმოადგენს ნიუტონის ბინომური კოეფიციენტების მიხედვით შედგენილ სამკუთხა ფორმის რიცხვების ცხრილს. დიდმა ფრანგმა მეცნიერმა, მათემატიკოსმა, ფიზიკოსმა, ფილოსოფოსმა, კიბერნეტიკოსმა, ალბათობის თეორიის ერთ-ერთმა ფუძემდებელმა ბლეზ პასკალმა 1665 წელს გამოაქვეყნა წიგნი „ტრაქტატი არითმეტიკული სამკუთხედის შესახებ“ და აღწერა სამკუთხედი, რომელიც შემდეგ პასკალის სამკუთხედის სახელით გახდა ცნობილი (იხ. ნახ. 10.10.), თუმცა პასკალზე ადრე ასეთი სამკუთხედის შესახებ იცოდნენ სხვა მეცნიერებმაც: გერმანიაში სტიფელისმა და სამკუთხედს სტიფელის სამკუთხედს უწოდებდნენ (1544 წ.), იტალიაში - ტარტალისმა, ამ სამკუთხედს იცნობდა ომარ ხაიამიც.

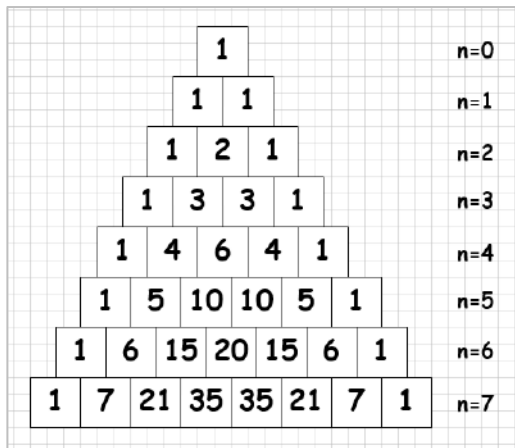


ნახ. 10.10. დიდი ფრანგი მეცნიერი ბლეზ პასკალი (1623-1662)

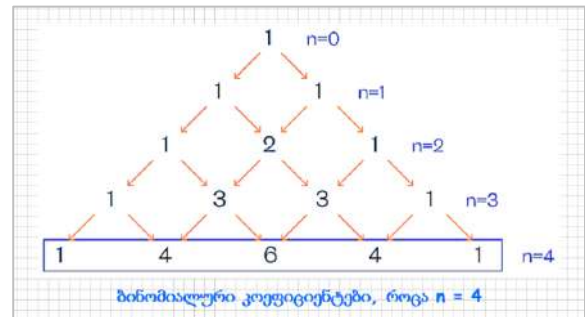
პასკალის სამკუთხედი გამოიყენება სხვადასხვა სფეროში: კომბინატორიკაში, სადაც მისი დახმარებით ხდება სხვადასხვა კომბინაცია რაოდენობის დათვლა. ასევე გამოიყენება ალგებრაში და ალბათობის თეორიაში, ციფრულ მეთოდებში გამოთვლების დასაჩქარებლად.

პასკალის სამკუთხედის სტრიქონები დანომრილია. სტრიქონების ნუმერაცია იწყება 0-ით, $n = 0$. n ცალ სტრიქონში გვაქვს $n + 1$ რაოდენობის ელემენტი.

პასკალის სამკუთხედის აგება ხდება ნიუტონის ბინომურ კოეფიციენტებზე C_n^k დაყრდნობით. პასკალის სამკუთხედის შიგნით ყოველი რიცხვი ტოლია მის თავზე მდგომი ორი რიცხვის ჯამისა (ნახ. 10.11.ა,ბ).



ა)



ბ)

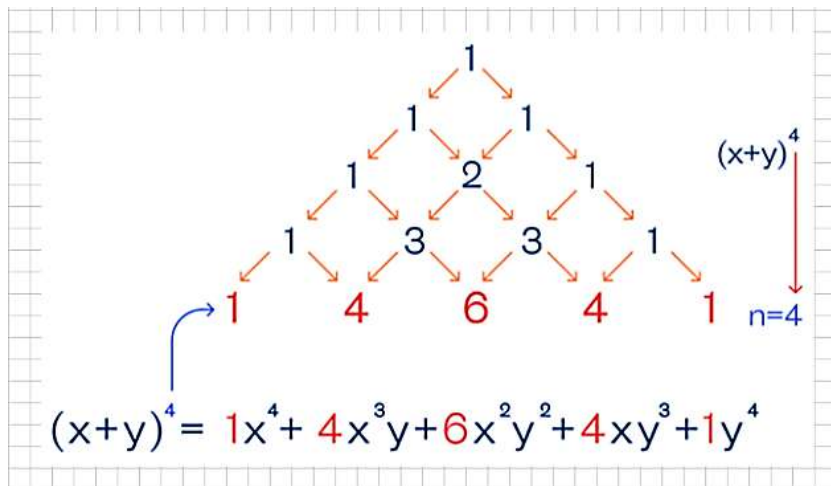
ნახ. 10.11. პასკალის სამკუთხედის: ა) სტრიქონების განლაგების სქემა;

ბ) $n = 4$ -ის დროს ბინომური კოეფიციენტებია 1, 4, 6, 4, 1

პასკალის სამკუთხედის მე-4 სტრიქონის წაკითხვა ხდება შემდეგი სახით:

$$C_4^0 = 1; C_4^1 = 4; C_4^2 = 6; C_4^3 = 4; C_4^4 = 1;$$

ახლა კი ვნახოთ თუ როგორ შეგვიძლია პასკალის სამკუთხედის დახმარებით დავწეროთ მრავალწევრი, როცა $n = 4$ (ნახ. 10.12.).



ნახ. 10.12. პასკალის სამკუთხედის მიხედვით ($n = 4$) დაწერილი მრავალწევრი

პასკალის სამკუთხედის n -ური სტრიქონის და k -ური სვეტის გადაკვეთაზე მყოფი ელემენტი ჩაიწერება როგორც $\binom{n}{k}$.

ბინომური კოეფიციენტების გამოსათვლელი ფორმულაა:

$$\binom{n}{k} \equiv C_n^k = \frac{n!}{k!(n-k)!}$$

0-ვანი სტრიქონისა და 0-ვანი სვეტის გადაკვეთაზე მოთავსებული ელემენტი 1-ის ტოლია ანუ $\binom{0}{0} = 1$.

ამ უკანასკნელის გამოთვლა ხდება ფორმულით:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Xსადაც n ნებისმიერი არაუარყოფითი რიცხვია და $0 \leq k \leq n$.

ეს ფორმულა პასკალის წესის სახელითაა ცნობილი.

ახლა განვაღებთ ბინომური კოეფიციენტებით შედგენილი პასკალის სამკუთხედი:

$$C_0^0 = 1$$

$$C_1^0 = 1 \quad C_1^1 = 1$$

$$C_2^0 = 1 \quad C_2^1 = 2 \quad C_2^2 = 1$$

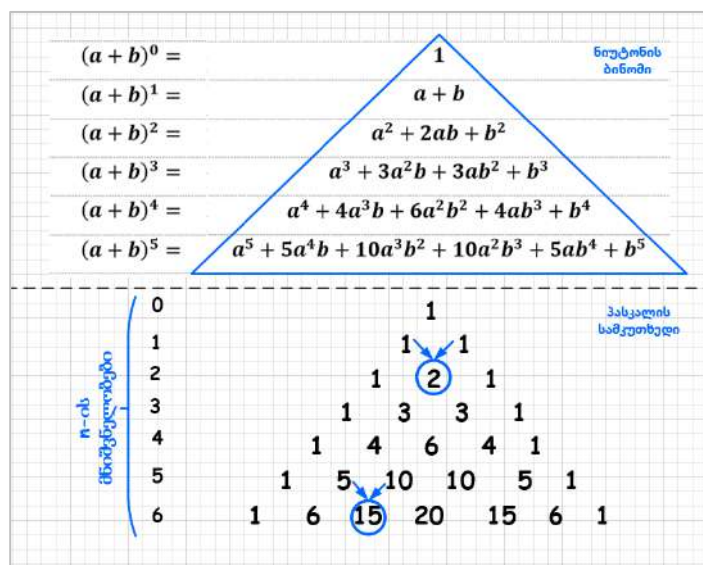
$$C_3^0 = 1 \quad C_3^1 = 3 \quad C_3^2 = 3 \quad C_3^3 = 1$$

$$C_4^0 = 1 \quad C_4^1 = 4 \quad C_4^2 = 6 \quad C_4^3 = 4 \quad C_4^4 = 1$$

ზემოთ მოცემული სახის განლაგებას პასკალის სამკუთხედი ჰქვია. ალბათ, შეამჩნევდით, რომ აქ ყოველი რიცხვი მისი ორი ზედა მეზობლის ჯამია.

$$\text{საზოგადოდ: } C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

ვინაიდან პასკალის სამკუთხედი ბინომური კოეფიციენტების გამოყენებითაა აგებული, თითოეული მწკრივი შეესაბამება $(a+b)$ -ს აყვანილს სტრიქონის რიგის ტოლ ხარისხში. ვნახოთ თუ როგორი კავშირია ნიუტონის ბინომსა და პასკალის სამკუთხედს შორის (ნახ. 10.13.)



ნახ. 10.13. კავშირი ნიუტონის ბინომსა და პასკალის სამკუთხედს შორის

პასკალის სამკუთხედის თვისებები:

პირველი თვისება. n -ურ სტრიქონის წევრთა ჯამი 2^n -ის ტოლია.

მეორე თვისება. პასკალის ყოველი სტრიქონი სიმეტრიულია.

მაგალითი 10.12. პასკალის სამკუთხედის გამოყენებით გამოვთვალოთ $(1+x)^5$ გამოსახულების მნიშვნელობა.

ამოხსნა.

პასკალის სამკუთხედის მე-5 სტრიქონის ელემენტები წარმოადგენენ $(1+x)^5$ გამოსახულების კოეფიციენტებს: **1, 5, 10, 10, 5, 1**

$$\text{ამიტომ } (1+x)^5 = 1 + 5x + 10x^2 + 10x^3 + 5x^4 + x^5$$

მაგალითი 10.13. პასკალის სამკუთხედის გამოყენებით ვიპოვოთ $(a+b)^6$ გამოსახულების მნიშვნელობა. გვაქვს 7 კოეფიციენტი:

$$(a+b)^6 = a^6 + c_1 a^5 b + c_2 a^4 b^2 + c_3 a^3 b^3 + c_4 a^2 b^4 + c_5 a b^5 + b^6$$

მაგრამ როგორ შეგვიძლია დავადგინოთ თითოეული კოეფიციენტის მნიშვნელობა c_i ?. ერთ-ერთი გზა არის მომდევნო სტრიქონების დამატება (იხ. ნახ. 10.14.):

ბოლო სტრიქონის მიღების თანამიმდევრობა:

პირველი და ბოლო რიცხვი არის 1;

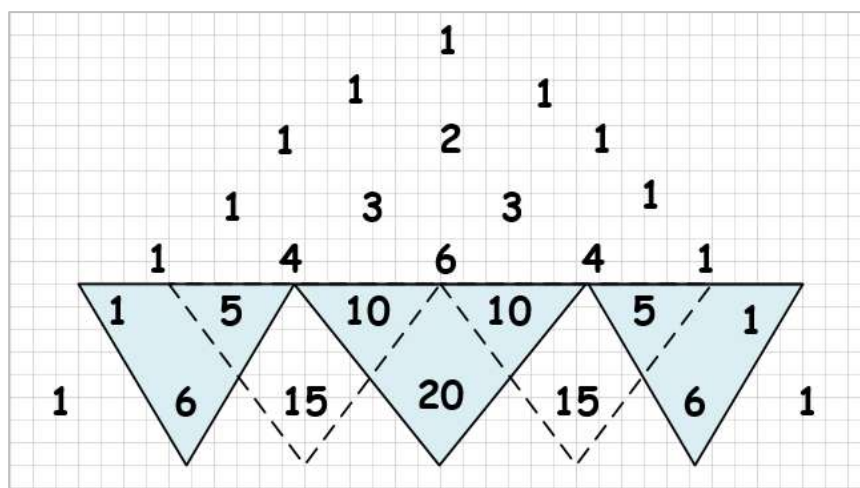
მეორე რიცხვი ტოლია 1+5 ანუ 6;

მესამე რიცხვი ტოლია 5+10 ანუ 15;

მეოთხე რიცხვი ტოლია 10+10 ანუ 20;

მეხუთე რიცხვი ტოლია 10+5 ანუ 15;

მეექვსე რიცხვი ტოლია 5+1 ანუ 6.



ნახ. 10.14. კოეფიციენტების მნიშვნელობების პოვნა

მომდევნო სტრიქონის დამატების გზით

მივიღეთ:

$$(a+b)^6 = 1a^6 + 6a^5b + 15a^4b^2 + 20a^3b^3 + 15a^2b^4 + 6ab^5 + 1b^6$$

ალბათობის გამოთვლის ელემენტები. ალბათობის თეორია არის მეცნიერება, რომელიც შეისწავლის მასობრივ შემთხვევით მოვლენათა რაოდენობრივი ხასიათის კანონზომიერებს. ალბათობის ძირითად ცნებას ხდომილობა წარმოადგენს.

ცდათა მოცემულ სერიაში რაიმე შემთხვევითი ხდომილობის ფარდობითი სიხშირე არის ამ ხდომილობის განხორციელებათა ოდენობის შეფარდება ცდათა საერთო ოდენობასთან.

ცდათა მრავალჯერ გამეორებისას შემთხვევითი ხდომილობის ალბათობა ამ ხდომილობის ფარდობითი სიხშირის ტოლია (მიახლოებით).

აუცილებელი ხდომილობა არის ხდომილობა, რომელიც მოცემული ცდის ჩატარებისას ყოველთვის ხორციელდება - თუ ცდათა რიცხვია n , აუცილებელი ხდომილობის მოხდენათა რიცხვი n -ია, ფარდობითი სიხშირე $= 1$, ალბათობა $= 1$.

ხდომილობას, რომელსაც მოცემული ექსპერიმენტი არც ერთი შედეგი არ შეესაბამება, შეუძლებელი ხდომილობა ეწოდება. მისი ალბათობა ნულია.

თუ A ნებისმიერი ხდომილობაა და $P(A)$ მისი ალბათობა, მაშინ $0 \leq P(A) \leq 1$.

განვიხილოთ კამათლის გაგორება, ვთქვათ, გვანტერესებს A ხდომილობა: „მოვიდა რიცხვი, რომელიც 3-ის ჯერადია“. ასეთი რიცხვი გვაქვს ორი: 3 და 6. ეს შედეგი არის A ხდომილობის ხელშემწყობი შედეგი.

ალბათური ხდომილება ეწოდება ხელშემწყობი ხდომილობის ოდენობის შეფარდებას ხდომილობის საერთო ოდენობასთან, ამიტომ კამათლის გაგორების ალბათობა იქნება: $P(A) = 2/6$.

თუ რაიმე ექსპერიმენტის შედეგები თანაბრად შესაძლებელია, მათი რიცხვი n -ის ტოლია და ამ ექსპერიმენტთან დაკავშირებული A ხდომილობას m ცალი შედეგი შეესაბამება (A -ს ხელშემწყობი შედეგების რიცხვია m), მაშინ A ხდომილობის ალბათობა გამოითვლება:

თუ A შეუძლებელი ხდომილობაა, მაშინ $m = 0$, $P(A) = 0$

თუ A აუცილებელი ხდომილობაა, მაშინ $m = n$, $P(A) = 1$

A ხდომილობასთან ერთად შეგვიძლია განვიხილოთ ხდომილობა - „არა A “ (არ გვაქვს A). მას ასე აღვნიშნავთ $\bar{A}$. იგი A ხდომილობის საწინააღმდეგო ხდომილობაა.

თუ A -ს ხელშემწყობი შედეგების რიცხვია m , მაშინ $\bar{A}$ ხელშემწყობი შედეგების რიცხვია $n - m$.

A და B ხდომილობათა ჯამი (გაერთიანება) ეწოდება იმ ელემენტარული ხდომილობებისაგან შედგენილ C ხდომილობას, რომლებიც ეკუთვნის A და B ხდომილობებიდან ერთ-ერთს მაინც. A და B ხდომილობათა ჯამი აღინიშნება $A + B$ ან $A \cup B$ სიმბოლოთი.

A და **B** ხდომილობათა ნამრავლი (თანაკვეთა) ეწოდება იმ ელემენტარული ხდომილობებისაგან შედგენილ **C** ხდომილობას, რომლებიც ერთდროულად ეკუთვნის **A** და **B** ხდომილობებს. **A** და **B** ხდომილობათა ნამრავლი აღინიშნება $A * B$ ან $A \cap B$ სიმბოლოთი.

ორ **A** და **B** ხდომილობას ეწოდება დამოუკიდებელი, თუ სამართლიანია ტოლობა:

$$P(A * B) = P(A) * P(B)$$

ორ **A** და **B** ხდომილობას არათავსებადი ეწოდება, თუ მათ საერთო ელემენტარული ხდომილობა არ აქვთ და სამართლიანია ტოლობა: $P(A + B) = P(A) + P(B)$

ნებისმიერი ორი **A** და **B** ხდომილობისათვის სამართლიანია ტოლობა:

$$P(A + B) = P(A) + P(B) - P(A * B)$$

პირობითი ალბათობა. **A** ხდომილობის ალბათობა იმ პირობით, რომ განხორციელდა **B** ხდომილობა, გამოითვლება ფორმულით:

$$P(A|B) = \frac{P(A * B)}{P(B)} \quad \text{ან} \quad P(A|B) = \frac{n(A * B)}{n(B)}$$

სადაც $n(A * B)$ და $n(B)$ -ხდომილობათა რაოდენობებს აღნიშნავს.

შესაბამისად გვაქვს ნამრავლის ალბათობის ფორმულები:

$$P(A * B) = P(B) * P(A|B) \quad \text{ან} \quad P(A * B) = P(A) * P(B|A)$$

ეს ფორმულები სამართლიანია ნებისმიერი ოდენობის თანამამრავლის შემთხვევაში.

თუ **A** და **B** დამოუკიდებელი ხდომილობებია, მაშინ $P(A * B) = P(A) * P(B)$ და პირობითი ალბათობა გამოითვლება ფორმულით: $P(A|B) = P(A)$

იაკობ ბერნულის ფორმულა: $P_n(k) = C_n^k p^k q^{n-k}$

ბერნულის თეორემა. თუ n განმეორებითი, დამოუკიდებელი ცდიდან თითოეულში **A** ხდომილობის მოხდენის ალბათობა p -ს ტოლია, ხოლო **A** ხდომილობის განხორციელების სიხშირე S_n , მაშინ ნებისმიერი დადებითი ε რიცხვისათვის:

$$\left(P \left| \frac{S_n}{n} - p \right| > \varepsilon \right) < \frac{P(1-p)}{n\varepsilon^2} < \frac{1}{4n\varepsilon^2}$$

ეს იმას ნიშნავს, რომ „წარმატებათა“ ფარდობითი სიხშირე „ახლოსაა“ „წარმატების“ p ალბათობასთან, როცა $n \rightarrow \infty$. (n -ის ზრდასთან ერთად უტოლობის მარჯვენა მხარე $\frac{1}{4n\varepsilon^2}$ მიისწრაფის ნულისკენ).

ხდომილების ალბათობა **ლაპლასის** მიერ განმარტებული იყო შემდეგნაირად: $P(A) = \frac{m}{n}$, სადაც n - თანაბრად შესაძლებელ ხდომილებათა (შედეგთა) საერთო რაოდენობაა, ხოლო m ხდომილებათა რაოდენობა, რომლის დროსაც ხდება **A** ხდომილება (ხელშემწყობი შედეგების რაოდენობა). მაგალითად, ვთქვათ, ორი კამათლის გაგორებისას მოსულ ქულათა ჯამია 8. გამოსათვლელია **A** ხდომილების ალბათობა. ორი კამათლის გაგორებისას შესაძლებელია მივიღოთ შემდეგი თანაბრად შესაძლებელი შედეგები:

(1,1) (1,2) (1,3) (1,4) (1,5) (1,6)
 (2,1) (2,2) (2,3) (2,4) (2,5) (2,6)
 (3,1) (3,2) (3,3) (3,4) (3,5) (3,6)
 (4,1) (4,2) (4,3) (4,4) (4,5) (4,6)
 (5,1) (5,2) (5,3) (5,4) (5,5) (5,6)
 (6,1) (6,2) (6,3) (6,4) (6,5) (6,6)

როგორც ვხედავთ, სულ შესაძლებელი ვარიანტებია 36. ცალკე გამოყოფილია ის ვარიანტები, როცა მოხდა A ხდომილება. ასეთი შემთხვევა 5 ცალია და ყველა ისინი თანაბრად შესაძლებელია. ამიტომ ლაპლასის განმარტების თანახმად $P(A) = \frac{5}{36}$.

ბუნებრივად, იზადება კითხვა: როდის და რომელი შემთხვევითი ხდომილებები შეიძლება ჩაითვალოს თანაბრად შესაძლებლად?

განვიხილოთ **დალამბერის ცნობილი შეცდომა**. ცნობილი ფრანგი მათემატიკოსი და ფილოსოფოსი ჟ. დალამბერი (1717-1783), მონეტის ორჯერადი აგდების ამოცანის განხილვისას თვლიდა, რომ გერბისა და საფასურის მოსვლის შანსი (ალბათობა) იყო $1/3$. სინამდვილეში ეს არის $1/2$.

დალამბერის შეცდომამ თავის დროზე ბევრი კამათი გამოიწვია და ამიტომ გახდა ცნობილი. ეს შეცდომა ძალიან ჭკუის სასწავლებელია.

დალამბერი მსჯელობდა შემდეგნაირად. არსებობს ამ ექსპერიმენტის სამი შესაძლო შედეგი: 1) მოვიდა ორი გერბი; 2) მოვიდა გერბი და საფასური; 3) მოვიდა ორი საფასური. დალამბერი თვლიდა რა ამ შედეგებს ტოლ შესაძლებლად, ასკვნიდა, რომ გერბისა და საფასურის მოსვლის ალბათობაა $1/3$ -ია, რაც შეცდომაა.

ცნობილი მათემატიკოსის შეცდომა მდგომარეობდა იმის დაშვებაში, რომ აღნიშნული სამი შედეგს თვლიდა ერთნაირად შესაძლებლად.

სინამდვილეში კი ეს ასე არ არის. გერბის მოსვლა აღვნიშნოთ ასოთი **გ**, ხოლო საფასურის - ასოთი **ს**. მაშინ მონეტის ორჯერადი აგდებისას ერთნაირად შესაძლებელი შედეგებია: **გგ**, **გს**, **სგ**, **სს**. ამ შედეგების რაოდენობა ოთხია (ე. ი. $n = 4$). აქედან „მოვიდა გერბი და საფასური“ ხდომილებას ხელს უწყობს ორი შედეგი: **გს** და **სგ** ანუ $m = 2$.

შესაბამისად, გერბისა და საფასურის მოსვლის ალბათობაა: $2/4 = 1/2$.

მაგალითი 10.14. განვიხილოთ ცნობილი მაგალითი „ბედნიერი“ ბილეთი. მოცემული გვაქვს 25 საგამოცდო ბილეთი. ამ 25 ბილეთებიდან 5 ბილეთი „ბედნიერია“, ხოლო დანარჩენი 20 ბილეთი „არ არის ბედნიერი“. რომელ სტუდენტს აქვს „ბედნიერი“ ბილეთის ალების მეტი ალბათობა: ვინც პირველი იღებს ბილეთს, თუ ვინც მეორე?

ამოხსნა. აღვნიშნოთ პირველი სტუდენტის მიერ აღებული ბილეთის ნომერი i -თი, ხოლო მეორე სტუდენტის მიერ აღებული ბილეთის ნომერი j -თი.

დავუშვათ, რომ „ბედნიერი“ ბილეთების ნომრებია: 1, 2, 3, 4, 5.

მაშინ, ცხადია, რომ $\Omega = \{(i, j): i, j = 1, 2, \dots, 25, i \neq j\}$, $|\Omega| = 25 * 24 = 600$ და შესაბამისად, ყველა ელემენტარული ხდომილება ტოლ-ალბათურია: $P(i, j) = 1/600$.

შემოვიღოთ ხდომილებები:

$A = \{ \text{პირველმა სტუდენტმა აიღო „ბედნიერი“ ბილეთი} \}$,

$B = \{ \text{მეორე სტუდენტმა აიღო „ბედნიერი“ ბილეთი} \}$.

მაშინ ამ ხდომილებებს აქვთ შემდეგი სახე:

$A = \{(i * j): i = 1 \dots 5; j = 1 \dots 25; i \neq j\}$

$B = \{(i * j): i = 1 \dots 25; j = 1 \dots 5; i \neq j\}$

$|A| = |B| = 5 * 24 = 120$

ალბათობის კლასიკური განმარტების თანახმად:

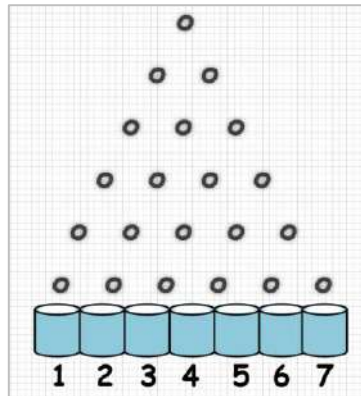
$$P(A) = P(B) = \frac{120}{600} = \frac{1}{5}$$

ე. ი. ორივე სტუდენტს აქვს „ბედნიერი“ ბილეთის აღების ერთი და იგივე ალბათობა.

მაგალითი 10.15. განვიხილოთ კიდევ ერთი საინტერესო ამოცანა, რომელიც გალტონის დაფის სახელითაა ცნობილი.

დავაზე გვაქვს სამკუთხედის ფორმით დალაგებული რგოლები, ისე, რომ წვეროში ერთი რგოლია, მეორე რიგში წინ რგოლისგან თანაბარ მანძილზე დაშორებული ორი, მესამე რიგში ზედა ორი რგოლიდან თანაბარ მანძილებზე დაშორებული სამი რგოლი და ა.შ. ბოლოში არის ექვსი რგოლი (იხ. ნახ. 10.15).

მე-7 რიგში კი არის ბოლო მე-6 რგოლიდან თანაბარ მანძილებზე 7 ღრმული. ზედა რგოლზე აგდებენ ბურთს და მას შეუძლია იმოძრაოს თანაბარი ალბათობით ან მარჯვნივ ან მარცხნივ რგოლიდან რგოლზე, რაც საბოლოოდ სრულდება რომელიმე ღრმულში ჩავარდნით. როგორია ალბათობა იმისა, რომ ბურთი ჩავარდება მეხუთე ღრმულში?



ნახ. 10.15. გალტონის დაფა

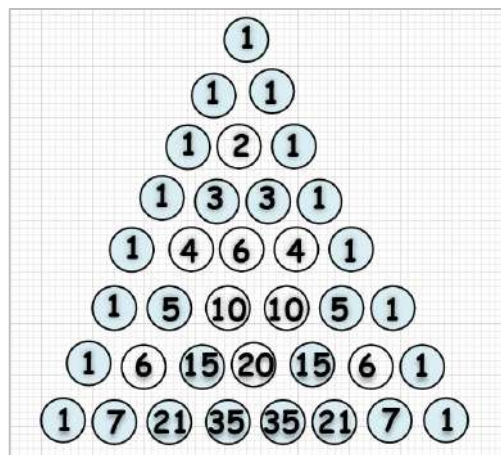
ამოხსნა. როგორც ვხედავთ, არსებობს პირველ და მე-7 ღრმულებში ბურთის ჩავარდნის ერთადერთი გზა (ტრაექტორია), მეორე და მე-6 ღრმულებში ბურთის ჩავარდნის ექვს-ექვსი გზა, მესამე და მეხუთე ღრმულებში თხუთმეტი-თხუთმეტი გზა და ბოლოს, მეოთხე ღრმულში ბურთის ჩავარდნის 20 გზა.

გზების (შედეგების) სრული რაოდენობაა:

$$1 + 6 + 15 + 20 + 15 + 6 + 1 = 64$$

და ყველა ეს შედეგი თანაბრად მოსალოდნელია, ვინაიდან თითოეული ტრაექტორიის გავლისას ბურთი განიცდის ექვს დაჯახებას რგოლებზე და ყოველი დაჯახებისას ის თანაბარი ალბათობებით გადაადგილდება ან მარჯვნივ ან მარცხნივ. თანაბარალბათური 64 შედეგიდან მეხუთე ღრმულში ჩავარდნას ხელს უწყობს 15 შედეგი და შესაბამისად, სამეზბნი ალბათობა იქნება $15/64$.

ქვემოთ მოყვანილია გალტონის დაფაზე რგოლების 7 რიგის შემთხვევაში თითოეულ პოზიციაზე ბურთის მოხვედრის შესაძლო გზების რიცხვი (ნახ. 10.16.).



ნახ. 10.16. გალტონის დაფაზე რგოლების შვიდი რიგის შემთხვევაში თითოეულ პოზიციაზე ბურთის მოხვედრის შესაძლო გზების რიცხვი

კომბინატორული ამოცანები. ამოცანათა ამოხსნის ყველა ალგორითმი შეგვიძლია შევაფასოთ ოთხი კრიტერიუმის საშუალებით: სისრულე, ოპტიმალურობა, დროითი სირთულე, სივრცითი სირთულე.

ალგორითმის სირთულე. ალგორითმის სირთულე არის რესურსების რაოდენობრივი მახასიათებელი, რომელიც საჭიროა ალგორითმის მიერ პრობლემის წარმატებით გადასაჭრელად.

ალგორითმების სირთულის ანალიზი. ალგორითმების სირთულის ანალიზის მიზანია მოიძიოს ოპტიმალური ალგორითმი პრობლემის გადასაჭრელად ე.ი. ალგორითმმა მუშაობისას უნდა გამოიყენოს მინიმალური დრო და მეხსიერება. სირთულის თეორია, როგორც გამოთვლის თეორიის ნაწილი, სწავლობს გამოთვლის რესურსებს ან ხარჯებს, რომლებიც საჭიროა მოცემული პრობლემის დასასრულებლად.

ალგორითმის გამოთვლითი სირთულე არის ფუნქცია, რომელიც განსაზღვრავს ზოგიერთი ალგორითმის მიერ შესრულებული სამუშაოს რაოდენობის დამოკიდებულებას შეყვანილი მონაცემების თვისებებთან.

ალგორითმის მიერ შესრულებული სამუშაოს ოდენობა/მოცულობა იზომება დროისა და სივრცის აბსტრაქტული ცნებებით, რომელსაც მანქანური გამოთვლითი რესურსები ეწოდება

დრო განისაზღვრება პრობლემის გადასაჭრელად საჭირო ელემენტარული ბიჯების რაოდენობით, ხოლო სივრცე მეხსიერების მოცულობით განისაზღვრება.

ალგორითმის დროითი სირთულე, უარეს შემთხვევაში, შეყვანის მონაცემების ზომის ფუნქციაა, რომელიც აჩვენებს ელემენტარული ოპერაციების მაქსიმალურ რაოდენობას, რომელსაც ალგორითმი ხარჯავს მითითებული ზომის დავალების ამოხსნაზე.

საუკეთესო შემთხვევაში, ეს არის შეყვანის მონაცემების ზომის ფუნქცია, რომელიც აჩვენებს ალგორითმის მიერ მოთხოვნილი ელემენტარული ოპერაციების მინიმალურ რაოდენობას განსაზღვრული ზომის პრობლემის გადასაჭრელად.

ალგორითმის სივრცითი სირთულე, უარეს შემთხვევაში, შეყვანის მონაცემების ზომის ფუნქციაა, რაც მიუთითებს მეხსიერების მაქსიმალურ ოდენობაზე, რომელსაც ალგორითმი იყენებს განსაზღვრული ზომის დავალების მაგალითის გადასაჭრელად.

საუკეთესო შემთხვევაში, ეს არის შეყვანის მონაცემების ზომის ფუნქცია, რომელიც აჩვენებს მეხსიერების მინიმალურ რაოდენობას, რომელსაც ალგორითმი იყენებს განსაზღვრული ზომის ამოცანის ამოსახსნელად.

გამოთვლითი სირთულებებისა და დროითი დანახარჯის მიხედვით განასხვავებენ პოლინომიალურ და ექსპონენციალურ ალგორითმებს.

პოლინომიალური ალგორითმი. თუ ალგორითმის სირთულის ორივე ფუნქცია პოლინომიალურია, მაშინ ალგორითმი არის პოლინომიალური სირთულის ალგორითმი.

პოლინომიალური სირთულის ალგორითმის მაგალითია მარტივი დახარისხების ალგორითმი.

პოლინომიალური სირთულის ალგორითმის მუშაობის დრო შესაძლებელია წარმოვადგინოთ $O(n^2)$, $O(n^3)$ და ა.შ. სახით.

ექსპონენციალური ალგორითმები. ალგორითმი, რომელსაც სირთულის ორი ფუნქციიდან ერთ-ერთი ფუნქცია ექსპონენციალურია, ეწოდება ექსპონენციალური სირთულის ალგორითმი.

თუ ექსპონენციალური სირთულის ალგორითმის მუშაობის დრო არ იზრდება ვიდრე n -ის ექსპონენტი, მაშინ საუბარია ექსპონენციალურ ალგორითმზე.

ექსპონენციალური არიან ის ალგორითმები, რომელთა მუშაობის დრო წარმოდგენილი ექსპონენციალური ფუნქციის სახით, მაგალითად, $O(2^n)$, $O(3^n)$ ან $(n!)$.

ეს ალგორითმი სწრაფად იზრდება მონაცემების ზომასთან ერთად.

ექსპონენციალური ალგორითმების მუშაობა ხშირად მიუღებლად დიდ დროს საჭიროებს, როდესაც n დიდი ზომისაა.

ექსპონენციალური სირთულის ალგორითმი უფრო უარესი შემთხვევაა, ვიდრე პოლინომიალური სირთულის ალგორითმი.

$O(n^2)$ - ეს პოლინომიალური დროა. მრავალწევრი ტოლია $f(n) = n^2$.

მეორეს მხრივ $O(2^n)$ - ეს ექსპონენციალური დროა, სადაც ექსპონენციალური ფუნქცია $f(n) = 2^n$.

ძირითადი განსხვავება **პოლინომიალურ და ექსპონენციალურ ალგორითმები**.

პოლინომიალური ალგორითმები უმეტესწილად ითვლებიან პრაქტიკულად განხორციელებადად, რადგან მათი მუშაობის დრო სწრაფად არ იზრდება. ხოლო ექსპონენციალური ალგორითმები ითვლებიან უფრო რთულად და არაეფექტურად დიდ მონაცემებზე.

პოლინომიალური ალგორითმების დრო იზრდება ნელა, ხოლო ექსპონენციალური ალგორითმების დრო იზრდება ძალიან სწრაფად მონაცემების ზომასთან ერთად.

პოლინომიალური და ექსპონენციალური ალგორითმების შედარება. პოლინომიალური ალგორითმები უფრო პრაქტიკული და ეფექტურია დიდი მოცულობის მონაცემების დამუშავებისას.

ექსპონენციალური ალგორითმები რთულია და მოითხოვს დროით დანახარჯებს, ეფექტურია მცირე მოცულობის მონაცემებთან და არაპრაქტიკულია დიდ მონაცემებთან მიმართებით. პოლინომიური ალგორითმით მოგვარებულ პრობლემას **ადვილად გადაჭრის პრობლემა** ეწოდება. პრობლემას, რომლის მოგვარება შეუძლებელია პოლინომიური ალგორითმით, ეწოდება **რთული პრობლემის მოგვარება**.

ამოცანის სირთულე არ განისაზღვრება და არ გამოიხატება მხოლოდ საუკეთესო ალგორითმის სირთულის შეფასებით. სირთულის შესაფასებლად შემოღებულია კლასიფიცირებული ფუნქციები, რომლის გამოთვლაც შესაძლებელია მოცემული შეზღუდვების ფარგლებში მოხმარებული რესურსების გამოყენებით.

დროითი და სივრცითი სირთულები ყოველთვის გამოითვლება ამოცანის სირთულის გათვალისწინებით, მაგრამ არსებობს ისეთი სახის ამოცანები, რომლებსაც არ გააჩნიათ გადაწყვეტის ალგორითმი ანუ ალგორითმულად ამოუხსნადია და მაშინაც კი, თუ ალგორითმი არსებობს, ის შეიძლება ვერ გამოვიყენოთ პრაქტიკაში მისი მაღალი სირთულის გამო.

განსაზღვრა. მარტივად ამოხსნადი ამოცანები არის ისეთი ამოცანები, რომელთა ამოხსნა შესაძლებელია პოლინომიალურ დროში. რთულად ამოხსნადი ამოცანები არის ამოცანები, რომელთა ამოხსნაც შეუძლებელია პოლინომიალურ დროში ან არ არის ნაპოვნი პოლინომიურ დროში მათი ამოხსნის ალგორითმი. გარდა ამისა, როგორც ა. ტიურინგმა დაამტკიცა, არსებობს პრინციპულად გადაუჭრელი ამოცანები.

გადაუჭრელი ამოცანები და პრობლემები ალგორითმების თეორიაში. არსებობს გადაუჭრელი ამოცანების ორი სახე:

1. ამოცანები, რომლებისთვისაც არ არის დადასტურებული, რომ ისინი გადაუჭრელია, მაგრამ არ მოიძებნა მათი ამოხსნის ალგორითმები.

2. ამოცანები, რომლებიც ამოხსნადია, მაგრამ ამოხსნა შეუძლებელია დროის ლიმიტისა და დიდი ზომის მესხიერების გამოყენების გამო.

რატომ უნდა ვიცოდეთ ალგორითმის ეფექტურობა/სირთულე?

ვთქვათ, ერთი და იგივე ამოცანის ამოსახსნელად გვაქვს ორი ალგორითმი:

პირველი ალგორითმი გადაირბენს ყველა ელემენტს მხოლოდ ერთხელ და რაღაცას ასრულებს $O(N)$ დროში;

მეორე ალგორითმი კი გადაირბენს ყველა ელემენტს $O(N^2)$ დროში.

თუ გვაქვს $N = 10$ ელემენტი, ციკლში პირველი ალგორითმის მიხედვით ის შეასრულებს 10 ოპერაციას, ხოლო მეორის მიხედვით კი 100 (ანუ 10-ჯერ მეტს) ოპერაციას, მაგრამ $N = 1000$ -ის შემთხვევაში კი პირველი ალგორითმის მიხედვით 1000 გადარბენა იქნება და მეორე ალგორითმის მიხედვით კი უკვე 1 000 000!

ალგორითმულად ამოუხსნადი ამოცანები.

პრობლემა N1. n რიცხვის მარტივობის შემოწმება.

n რიცხვის მარტივობის შესამოწმებელი ალგორითმი შემდეგია: შემოწმდეს, იყოფა თუ არა n რიცხვი ნებისმიერ მთელ რიცხვზე 2-დან $\sqrt{n}$ -ის ჩათვლით. თუ n იყოფა ამ შუალედში მოთავსებულ რომელიმე მთელ რიცხვზე, მაშინ ის მარტივ რიცხვს არ წარმოადგენს და მას შედგენილი რიცხვი ეწოდება. ხოლო თუ n ამ შუალედში მოთავსებულ არც ერთ მთელ რიცხვზე არ იყოფა, მაშინ ის მარტივია. რიცხვის მარტივობის შემოწმების ალგორითმს $\sqrt{n}$ რაოდენობის გაყოფის ოპერაციის ჩატარება ესაჭიროება, რაც, ცხადია, დიდი რიცხვებისთვის მეტად მოუხერხებელია.

მარტივი რიცხვების თეორმის თანახმად, შემთხვევითად არჩეული n რიცხვის მარტივობის ალბათობა n რიცხვის ციფრების რაოდენობის ან მისი ლოგარითმის უკუპროპორციულია. მიუხედავად მარტივი რიცხვების ფართომასშტაბიანი შესწავლისა, მათემატიკაში დღემდე არსებობს მათთან დაკავშირებული გადაუჭრელი ამოცანები.

მარტივი რიცხვები ფართოდ გამოიყენება ინფორმატიკასა და კრიპტოგრაფიაში (რადგან დიდი რიცხვების მარტივ მამრავლებად დაშლა რთული პროცედურაა). დიდი მარტივი რიცხვების ძიებითაა მოტივირებული განსაკუთრებული მარტივი რიცხვების შესწავლა. მაგალითად, მერსენის მარტივი რიცხვები, რომელთა მარტივობა შედარებით ადვილი შესამოწმებელია. 2010 წლის მონაცემებით, ჩვენთვის ცნობილი უდიდესი მარტივი რიცხვის ათობით ჩანაწერი დაახლოებით 13 მილიონი ციფრისგან შედგება.

პრობლემა N2. ალგორითმის შეჩერების შეუძლებლობა ანუ შეჩერდება თუ ჩაიცვლება ალგორითმი მოცემული მნიშვნელობისათვის.

განვიხილოთ ალგორითმი, რომელიც რიგ-რიგობით გადაამოწმებს ლუწ რიცხვებს და შეჩერდება მაშინ და მხოლოდ მაშინ, თუ განხილული რიცხვი მაქსიმუმ ორი მარტივი რიცხვის ჯამისაგან არ შედგება. ეს ამოცანა უდიდესი პრობლემაა (გოლდბახის ჰიპოთეზა) და ცხადია ჯერ-ჯერობით წინასწარ ვერ ვიტყვით შეჩერდება თუ არა ეს ალგორითმი. ამრიგად ზოგი ალგორითმისათვის შეიძლება შეჩერება და ზოგისათვის კი - არა. დაისმის კითხვა? არსებობს თუ არა რაღაც ერთი ზოგადი მეთოდი, რომელიც რაიმე პასუხს გაგვცემს? როგორც აღმოჩნდა, ასეთი ზოგადი მეთოდი არ არსებობს და ვერც იარსებებს (ამის დამტკიცება ტიურინგის მანქანით ხდება).

პრობლემა N3. π რიცხვის ჩაწერისას ათწილად ნაწილში ცხრიანების განაწილება ან ნებისმიერი ციფრის გადანაწილება (ცხრილი 10.1.).

ცხრილი 10.1. π რიცხვის ჩაწერისას ათწილად ნაწილში ცხრიანების განაწილება

π- ნამდვილი ტრანსცენდენტური (და ამიტომ ირაციონალური) რიცხვი, რომელიც დაახლოებით 3.14159-ს უდრის და წრეწირის სიგრძის მის დიამეტრთან შეფარდების ტოლია.							
მძიმეს შემდეგ პირველი ათასი სიმბოლო π რიცხვში							
3.1415926535	8979323846	2643383279	5028841971	6939937510	5820974944	5923078164	
0628620899	8628034825	3421170679	8214808651	3282306647	0938446095	5058223172	
5359408128	4811174502	8410270193	8521105559	6446229489	5493038196	4428810975	
6659334461	2847564823	3786783165	2712019091	4564856692	3460348610	4543266482	
1339360726	0249141273	7245870066	0631558817	4881520920	9628292540	9171536436	
7892590360	0113305305	4882046652	1384146951	9415116094	3305727036	5759591953	
0921861173	8193261179	3105118548	0744623799	6274956735	1885752724	8912279381	
8301194912	9833673362	4406566430	8602139494	6395224737	1907021798	6094370277	
0539217176	2931767523	8467481846	7669405132	0005681271	4526356082	7785771342	
7577896091	7363717872	1468440901	2249534301	4654958537	1050792279	6892589235	
4201995611	2129021960	8640344181	5981362977	4771309960	5187072113	4999999837	
2978049951	0597317328	1609631859	5024459455	3469083026	4252230825	3344685035	
2619311881	7101000313	7838752886	5875332083	8142061717	7669147303	5982534904	
2875546873	1159562863	8823537875	9375195778	1857780532	1712268066	1300192787	
6611195909	2164201989						

განვსაზღვროთ ფუნქცია $f(n) = i$, სადაც n არის π რიცხვის ათწილად ნაწილში ცხრიანების რაოდენობა, ხოლო i არის ყველაზე მარცხნივ მდებარე ცხრიანი, n რაოდენობის ცხრიანებს შორის.

$$\pi = 3.141592 \dots f(1) = 5,$$

მიზანი: $f(n)$ ფუნქციის გამოთვლა.

პრობლემა N3-ის ამოუხსნადობა. რადგან π რიცხვი არის ტრანსცენდენტური (მას ვერ მივიღებთ მთელ რიცხვებზე ალგებრული მოქმედების საშუალებით) და ირაციონალური (მის მნიშვნელობას წილადით ვერ გამოვხატავთ), ამიტომ ჩვენ არაფერი ვიცით π ათწილად ნაწილში ცხრიანის განაწილების (ასევე არა მარტო ცხრიანის, არამედ ნებისმიერი სხვა რიცხვის) შესახებ.

პრობლემა N4. სრულყოფილი რიცხვის გამოთვლა.

რიცხვს უწოდებენ სრულყოფილს, თუ იგი მისივე გამყოფების ჯამის ტოლია (გამყოფებში თავად ეს რიცხვი არ ითვლება). სრულყოფილი რიცხვებია:

$$6 = 1 + 2 + 3$$

$$28 = 1 + 2 + 4 + 7 + 14$$

$$496 = 1 + 2 + 3 + \dots + 29 + 30 + 31$$

$$8128 = 1 + 2 + 3 + \dots + 125 + 126 + 127$$

განვსაზღვროთ ფუნქცია: $S(n) = n$ -ურ სრულყოფილ რიცხვს.

დავსვათ ამოცანა: გამოვთვალოთ $S(n)$ ფუნქცია შემთხვევითად მიწოდებული n -სათვის.

პრობლემა N4-ის ამოუხსნადობა. არ არსებობს სრულყოფილი რიცხვების გამოსათვლელი ზოგადი მეთოდი. ჩვენ არც კი ვიცით, სრულყოფილი რიცხვების სიმრავლე სასრულია თუ არა, ამიტომ ალგორითმი უნდა განმეორდეს ზედიზედ ყველა რიცხვზე, სრულყოფილების შესამოწმებლად. იმის გამო, რომ არ არსებობს გადაწყვეტის ზოგადი მეთოდი, ჩვენ არ შეგვიძლია ვთქვათ რაიმე **ალგორითმის შეჩერების** შესახებ სასრული რაოდენობის ბიჯების შემდეგ.

დასკვნა: პოლინომიალური სირთულის ალგორითმის ზრდის ტემპი არის ზომიერად ნელი - შეყვანის ოდნავი ზრდა არ იწვევს დროის კატასტროფულ მატებას. შესაძლებელია როგორც მცირე ზომის, ასევე დიდი მონაცემების დამუშავება. ათობით მილიონი ელემენტის დალაგება შესაძლებელია წამებში, რადგან პოლინომიალური სირთულე მასშტაბირებადია.

თანამედროვე ალგორითმების უდიდესი ნაწილი სწორედ პოლინომიალურ დროში მუშაობს. **პოლინომიალური დრო** - პრობლემის პრაქტიკულად გადაჭრის შესაძლებლობა.

ექსპონენციალური სირთულე კი მიუთითებს ისეთ ალგორითმებზე, სადაც დრო ძალიან სწრაფად იზრდება, ხშირად იმდენად სწრაფად, რომ მცირე მასშტაბის ამოცანებზეც კი გამორიცხულია მათი რეალურ სისტემებში გამოყენება. **ექსპონენციალური დრო** - პრობლემა პრაქტიკულად გადაუჭრელია დიდ მონაცემებზე. სწორედ ამიტომ, კომპიუტერულ მეცნიერებაში უდიდესი მნიშვნელობა აქვს ამ ორი ტიპის ერთმანეთისგან გარჩევასა და ექსპონენციალური ამოცანების პოლინომიალურ ალგორითმებად გადაქცევის ძიებას.

კრიპტოგრაფიის საფუძვლები. მარტივი ჩასმადი შიფრი, კრიპტოგრაფია ღია გასაღებით. RSA კრიპტოსისტემების შინაარსი, ჰიბრიდული კრიპტოსისტემები, შემთხვევითი რიცხვების გამოთვლა.

კრიპტოგრაფია (Cryptography). კრიპტოგრაფია წარმოადგენს მონაცემთა გარდაქმნის (დაშიფვრის) სპეციალურ სისტემას და მისი დანიშნულებაა გაუგებარი გახადოს სისტემა იმ პირებისათვის, რომლებიც ამ სისტემას არ იცნობენ და არ ფლობენ ამოშიფვრის გასაღებს. კრიპტოგრაფია გამოიყენება საიდუმლო დოკუმენტების შესანახად ან გადასაგზავნად.

უფლებამოსილ პირს შეუძლია მონაცემების აღდგენა პირველადი ფორმით კრიპტოგრაფიული პროგრამა/გასაღების საშუალებით. კრიპტოგრაფია წარმოადგენს მონაცემთა უსაფრთხოების სისტემის მნიშვნელოვან კომპონენტს. გერმანული კრიპტოგრაფიული დანადგარი - ლორენცის მანქანა, გამოიყენებოდა მეორე მსოფლიო ომის დროს ყველაზე გასაიდუმლოებული შეტყობინების შიფრაციისთვის (ნახ. 11.1.).



ნახ. 11.1. ლორენცის მანქანა

კრიპტოგრაფია წარმოიშვა ბერძნული სიტყვებიდან: κρυπτός - ფარული (კრიპტოს) და ზმნიდან γράφω - წერა (გრაფო) ანუ ფარული წერა. კრიპტოგრაფია არის მეცნიერება ინფორმაციის დაფარვის შესახებ. კრიპტოგრაფია განიხილება, როგორც მათემატიკისა და კომპიუტერული მეცნიერებების განაყოფი და მჭიდროდ დაკავშირებულია მეცნიერების ისეთ დარგებთან, როგორებიცაა ინფორმაციის თეორია, კომპიუტერული უსაფრთხოება და ინჟინერია. დღესდღეობით იგი გამოიყენება ისეთ ტექნოლოგიურად განვითარებულ სფეროებში, როგორიცაა: საკრედიტო ბარათები, კომპიუტერული პაროლები, ელექტრონული კომერცია და მრავალი სხვა.

ტერმინოლოგია. კრიპტოგრაფიული სისტემა ყოველთვის გულისხმობს ორ ან მეტ მონაწილეს: გამგზავნს და მიმღებს, რომელთაც სურთ ერთმანეთს გადასცენ რაიმე სახის ინფორმაცია, ისე რომ ამ სისტემის გარეშე მყოფმა პირმა ვერ შეძლოს ამ ინფორმაციის მოპოვება (კერძო შემთხვევებში მონაწილე შეიძლება იყოს ერთი პირი, რომელსაც სურს

მოახდინოს ინფორმაციის საიმედოდ შენახვა ანუ ინფორმაცია გადასცეს თავის თავს, ოღონდ მომავალში). თვითონ გადასაცემ ინფორმაციას ეწოდება ღია ტექსტი.

ინფორმაციის სახეცვლილებას ისე, რომ დაფარულ იქნას მისი აზრი, **შიფრაციას** უწოდებენ, უკუპროცესს - **დეშიფრაციას**, მიღებულ შედეგს - **შიფროტექსტს**, ხოლო შიფრაცია/დეშიფრაციის ალგორითმს - **შიფრს**. ეს პროცესი უმეტეს შემთხვევაში დამოკიდებულია გასაღებზე. ეს არის საიდუმლო პარამეტრი, რომელიც ცნობილია მხოლოდ ურთიერთ-მოკავშირე მხარეებისათვის.

შიფრები გასაღების გარეშე უსარგებლოა, რადგან არსებობს მათი გატეხვის ტრივიალური ხერხები. კრიპტოსისტემა წარმოადგენს შიფრის, ყველა შესაძლო ღია ტექსტის, შიფროტექსტის და გასაღების ერთობლიობას. კრიპტოგრაფიული პროტოკოლები წარმოადგენენ წესების ერთობლიობას, რომლებიც სრულდება კრიპტოგრაფიული სისტემის მონაწილეების მიერ, თანამიმდევრობით და უპირობოდ.

კრიპტოგრაფიის ისტორია. ადრე კრიპტოგრაფია გამოიყენებოდა შეტყობინებათა საიდუმლოების დასაცავად: ხდებოდა ტექსტის გარდაქმნა სიმბოლოების გაურკვეველ ნაკრებად, ისე რომ უკუგარდაქმნა იოლი ყოფილიყო მხოლოდ შეტყობინების ადრესატისათვის.

უკანასკნელ ათწლეულებში კრიპტოგრაფიამ ისეთი ფუნქციები შეიძინა, როგორიცაა შეტყობინების მთლიანობის შემოწმება, ნამდვილობის (ვინაობის, იგივეობის) დადგენა, ციფრული ხელმოწერა, პირადობის შემოწმება, საიდუმლო გამოთვლები და სხვა.

შიფრის კლასიკური ტიპებია გადანაცვლებადი შიფრი (ტექსტში ხდება სიმბოლოების გარკვეული გადაადგილება) და ჩანაცვლებადი შიფრი (ტექსტში ხდება ერთი სიმბოლოს ან სიმბოლოების ჯგუფის სხვებით შეცვლა). ჩანაცვლებადი შიფრის პირველი მაგალითია იულიუს კეისრის შიფრი, რომელშიც ლათინური ანბანის ყოველი ასო იცვლებოდა მისგან მარჯვნივ სამი პოზიციით დაცილებული სიმბოლოთი.

კომპიუტერების გამოჩენამ შესაძლებელი გახადა შექმნილიყო გაცილებით რთული და დახვეწილი შიფრები. შესაძლებელი გახდა, დაშიფრულიყო ყველანაირი მონაცემი ბინარულ ფორმატში. კომპიუტერული შიფრები უმეტესწილად მანიპულირებენ ბიტებზე ან ბიტების ჯგუფებზე, განსხვავებით კლასიკური პოლიალფაბეტური შიფრებისაგან, რომლებიც ტექსტის ცალკეულ სიმბოლოებს ამუშავებდნენ. ამავე დროს გაადვილდა შიფრების კრიპტოანალიზი.

მიუხედავად ამისა, თანამედროვე შიფრები უზრუნველყოფენ მონაცემების დაცვის მაღალ ხარისხს, მათი კრიპტოანალიზის გზით გატეხვა იმდენად კოლოსალურ გამოთვლით და დროით რესურსებს მოითხოვს, რომ პრაქტიკულად შეუძლებლად მიიჩნევა. თუ

მეოცე საუკუნემდე კრიპტოგრაფია ძირითადად მანიპულირებდა სიმბოლოებზე და იყენებდა ლინგვისტიკის ელემენტებს, მეოცე საუკუნიდან ხდება მათემატიკის ინტენსიური გამოყენება, ინფორმაციის თეორიის, სტატისტიკის, კომბინატორიკის, აბსტრაქტული ალგებრის და რიცხვთა თეორიის ჩათვლით. კრიპტოგრაფიული მეთოდები მოითხოვენ ძალიან დიდი რაოდენობით გამოთვლების ჩატარებას.

კრიპტოანალიზი. კრიპტოანალიზის უმთავრესი ამოცანაა იპოვოს სუსტი წერტილები კრიპტოგრაფიულ სისტემაში, რათა შეძლოს ამ სისტემით დაცული ინფორმაციის მოპოვება.

კრიპტოანალიზი შეიძლება გამოყენებულ იქნას, როგორც ბოროტგანმზრახველის მიერ, ასევე კრიპტოსისტემის შემქმნელის მიერ ამ სისტემაში ნაკლის აღმოსაჩენად (გაცილებით იოლია სისტემის ნაკლის პოვნა, ვიდრე მისი სრულყოფილების დამტკიცება).

ფართოდ გავრცელებულია (და ამავე დროს მცდარი) აზრი იმის შესახებ, რომ ნებისმიერი შიფრის გატეხვა შესაძლებელია სასრულ დროში.

არსებობს კრიპტოანალიზის რამდენიმე ტიპი. ძირითადი განსხვავება მათ შორის არის მოწინააღმდეგისათვის ცნობილი ინფორმაციის ოდენობა და სახე, რის საფუძველზეც ხდება ანალიზი. ცნობილი შიფროტექსტის შემთხვევაში კრიპტოანალიტიკოსს ხელთ აქვს მხოლოდ დაშიფრული ინფორმაცია.

ცნობილი ღია ტექსტის შემთხვევაში კრიპტოანალიტიკოსს აქვს შიფროტექსტი და მისი შესაბამისი ტექსტი, მაგრამ არ აქვს გასაღები. არჩეული ტექსტის შემთხვევაში ანალიტიკოსს შეუძლია თვითონ შეარჩიოს ტექსტი და მიიღოს მისი შესაბამისი შიფროტექსტი და ბოლოს არჩეული შიფროტექსტის შემთხვევაში ანალიტიკოსს შეუძლია შეარჩიოს შიფროტექსტები და შეისწავლოს მათი შესაბამისი ტექსტები.

გასაღები კრიპტოგრაფიაში. ტერმინი გასაღები აღნიშნავს ინფორმაციას, რომელიც ცალსახად განსაზღვრავს კრიპტოგრაფიული ალგორითმის და შიფრის გამომავლ შედეგს. გასაღები შესაძლოა გამოყენებულ იქნას სხვა კრიპტოგრაფიულ ალგორითმებში, კერძოდ ციფრულ ხელმოწერებსა და შეტყობინების აუთენტიფიკაციის კოდებში. გასაღების საიდუმლოდ შენახვა გაცილებით იოლია, რადგან ეს ინფორმაციის უფრო მცირე ნაწილია. ამიტომაც თანამედროვე ალგორითმების უსაფრთხოება მთლიანად ეფუძნება გასაღების საიდუმლოდ შენახვის პრინციპს.

გასაღების საიდუმლოდ შენახვა კრიპტოგრაფიის ერთ-ერთი ყველაზე რთულად გადასაწყვეტი პრობლემაა.

ბოროტგანმზრახველს, რომელსაც აქვს გასაღები (მოპოვებული მოპარვით, გამოძალვით, დაშინებით, სოციალური ინჟინერიით ან სხვა გზებით), შეუძლია ნებისმიერ დროს აღადგინოს ამ გასაღებით დაცული ნებისმიერი დაშიფრული ინფორმაცია.

კრიპტოგრაფიული ალგორითმები, რომლებიც იყენებენ ერთ გასაღებს შიფრაციის და დეშიფრაციის პროცესში, ეწოდებათ სიმეტრიული ალგორითმები, ხოლო ალგორითმებს, რომლებიც შიფრაციისთვის იყენებენ ერთ, ხოლო დეშიფრაციისთვის მეორე გასაღებს, ეწოდებათ ასიმეტრიულ ალგორითმებს. ამ ორი გასაღებიდან ერთი (შიფრაციის) შესაძლოა იყოს ფართოდ ცნობილი, ხოლო მეორე (დეშიფრაციის) იქნას შენახული საიდუმლოდ. ეს საშუალებას იძლევა ნებისმიერმა გააგზავნოს მეორე გასაღების მფლობელთან დაშიფრული ინფორმაცია.

დღეს კრიპტოგრაფიაში გავრცელებულია მონაცემების დაცვის ოთხი მიდგომა:

- დაშიფვრა სიმეტრიული გასაღებით;
- დაშიფვრა ღია გასაღებით;
- ჰიბრიდული კრიპტოსისტემები;
- ჰეშფუნქციები.

დაშიფვრა სიმეტრიული გასაღებით დაფუძნებულია ბიტების საგულდაგულოდ არევა-დარევაზე, ხოლო დაშიფვრას ღია გასაღებით საფუძვლად უდევს დიდი რიცხვების (1024-თანრიგიანი რიცხვების) გამრავლება და ხარისხში აყვანა, რაც დაკავშირებულია დიდ დროით დანახარჯებთან.

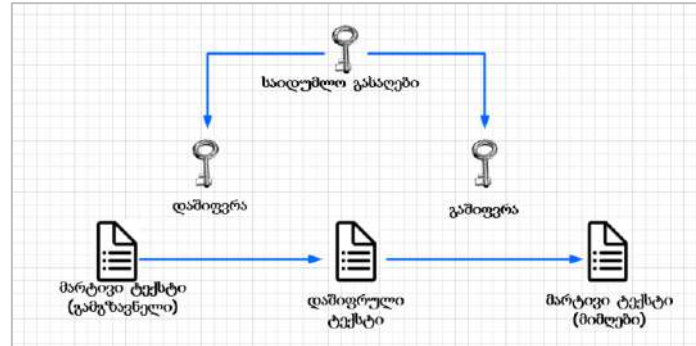
სიმეტრიული კრიპტოგრაფია (კრიპტოგრაფია დახურული გასაღებით). სიმეტრიული კრიპტოგრაფია იყენებს ერთ გასაღებს, როგორც შიფრაციისთვის, ისე დეშიფრაციისთვის. ამ მეთოდის უპირველეს ნაკლს გასაღების მართვის აუცილებლობა წარმოადგენს.

ქსელში ყოველ სხვადასხვა წყვილს უწევს იქონიოს ცალკე გასაღები, რაც წყვილთა რაოდენობის გაზრდისას გასაღებების რაოდენობის კვადრატული პროპორციით გაზრდას იწვევს. ორ მოკავშირე მხარეს შორის გასაღების გაცვლა, მაშინ როცა ჯერ არ არსებობს დაცული საკომუნიკაციო არხი, კვერცხის და ქათმის პრობლემას ემსგავსება (გასაღების გაცვლა უნდა მოხდეს ფარულად, ფარულად გაცვლა ითხოვს დაშიფრვას, დაშიფვრა თავის მხრივ ითხოვს გასაღების გაცვლას და ა. შ.). სიმეტრიული კრიპტოგრაფია იყენებს მეთოდებს, რომლის დროსაც ინფორმაციის გამგზავნი და მიმღები იყენებენ ერთსა და იმავე გასაღებს (იშვიათად სხვადასხვას, მაგრამ ამ შემთხვევაში ერთი გასაღები იოლად გამოითვლება მეორიდან).

1976 წლამდე ეს შიფრაციის ერთადერთი მეთოდი იყო. თანამედროვე სიმეტრიული კრიპტოგრაფია დაკავშირებულია ძირითადად ბლოკურ შიფრებთან, ნაკადურ შიფრებთან და მათ გამოყენებასთან (იხ. ნახ. 11.2.).

სიმეტრიული კრიპტოგრაფია იყენებს ერთსა და იმავე უნიკალურ ტექსტს (გასაღებს) როგორც დაშიფვრის, ისე დაშიფრული მონაცემების გაშიფრის პროცესში. სიმეტრიული

კრიპტოგრაფია არის კრიპტოგრაფიის ყველაზე მარტივი ფორმა. მის ერთ-ერთი მთავარ უპირატესობას წარმოადგენს მუშაობის კარგი სიჩქარე. სიმეტრიული დაშიფვრის გამოყენების მთავარი პრობლემა არის დახურული გასაღების უსაფრთხო გადაცემა ერთი მონაწილედან მეორესთან.



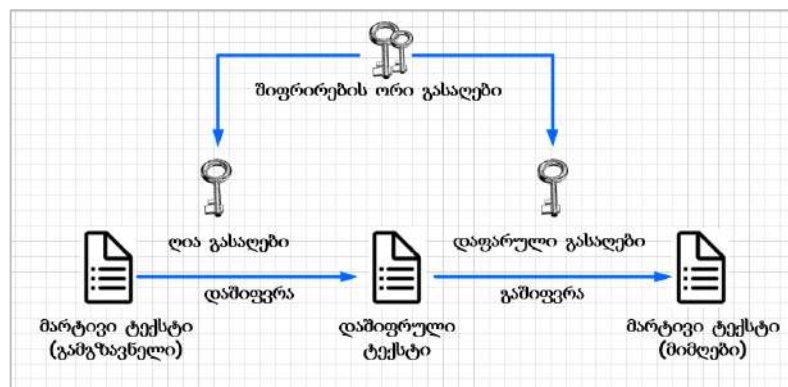
ნახ. 11.2. სიმეტრიული კრიპტოგრაფია

სიმეტრიული კრიპტოგრაფიის გამოყენება ხდება:

ფაილების დაშიფვრაში. ფაილი შეიძლება დაშიფრულ იქნას სიმეტრიული ალგორითმით, რომელიც უზრუნველყოფს მის დაცვას, გადაცემას ან შენახვისას. მხოლოდ ის, ვისაც აქვს გასაღები, შეძლებს ფაილის გაშიფვრას.

კომუნიკაციების დაცვაში. მაგალითად, TLS (Transport Layer Security) პროტოკოლი, რომელიც იცავს ინტერნეტზე გადასაცემ მონაცემებს, იყენებს სიმეტრიულ კრიპტოგრაფიას მონაცემთა უსაფრთხო გადასაცემად.

ასიმეტრიული კრიპტოგრაფია (კრიპტოგრაფია ღია გასაღებით). ასიმეტრიული კრიპტოგრაფია იყენებს ორი ტიპის გასაღებს: ღია და დაფარულ (საიდუმლო) გასაღებებს. ღია გასაღები ღიად ხელმისაწვდომია და გამოიყენება მონაცემების დაშიფვრაში, ხოლო დაფარული გასაღები საიდუმლოა და გამოიყენება დაშიფვრული მონაცემების გაშიფვრის პროცესში. RSA (Rivest Shamir Adleman) არის ერთ-ერთი ყველაზე ცნობილი ასიმეტრიული კრიპტოგრაფიის ალგორითმი. ასიმეტრიული კრიპტოგრაფია ღია გასაღებს იყენებს დაშიფვრაში და საიდუმლო (დაფარულ) გასაღებს გაშიფვრის პროცესში (ნახ. 11.3.).



ნახ. 11.3. ასიმეტრიული კრიპტოგრაფია

ერთი გასაღები საიდუმლოდ ინახება და მას დაფარული (დახურული, საიდუმლო) გასაღები ეწოდება, მეორე გასაღები - ღია გასაღები თავისუფლად არის ხელმისაწვდომი და მისი გამოყენება ნებისმიერ ადამიანს შეუძლია.

დაფარული გასაღები მხოლოდ მფლობელთან უნდა დარჩეს, ხოლო ღია გასაღების გადაცემა შესაძლებელია.

მიღებულია, რომ გასაღებთა წყვილს, რომელიც გამოიყენება ინფორმაციის გასაშიფრად, ეწოდება საიდუმლო გასაღები, ხოლო გასაღებთა წყვილს, რომლითაც იშიფრება ინფორმაცია ღია გასაღები.

პირველად ეს იდეა 1976 წელს წამოაყენეს სტენფორდის უნივერსიტეტის მეცნიერებმა დიფმა (Diffie) და ჰელმანმა (Hellman). გავრცელებულია შემდეგი ასიმეტრიული შიფრაციის სისტემები: RSA; EDH; DSA, ECDSA; დიფი-ჰელმანის სქემა; შნორის (Schnorr) სქემა; სქემა BLS (Boneh-Lynn-Shacham); სქემა GMR (Goldwasser-Micali-Rivest) და სხ.

RSA სისტემა (ასიმეტრიული შიფრაცია). ასიმეტრიული კრიპტოგრაფიის სისტემა RSA შეიქმნა 1977 წელს. მისი დასახელება წარმოდგება სისტემის შემქმნელების - Ron Rivest, Adi Shamir, Leonard Adleman - გვარების პირველი ასო-ნიშნებისგან. განსხვავებით სიმეტრიული კრიპტოგრაფიის სისტემებისგან, სადაც როგორც შიფრაცია, ისე დეშიფრაცია ხდება ერთი საიდუმლო გასაღების (Private Key) გამოყენებით, RSA კრიპტოსისტემაში გამოიყენება ორი გასაღები: ღია გასაღები (Public Key) და დაფარული/საიდუმლო გასაღები (Private Key). ღია ანუ საჯარო გასაღები ადვილად ხელმისაწვდომია და იგი გამოიყენება დეშიფრაციის (Decryption) ოპერაციაში. სისტემის ანონსირება მოხდა 1978 წელს. სისტემის არსი იმდენად მარტივია, რომ მას ხშირად არითმეტიკულსაც კი უწოდებენ. RSA ალგორითმი არის ასიმეტრიული კრიპტოგრაფიის ერთ-ერთი ძირითადი მეთოდი, რომელიც გამოიყენება მონაცემთა შიფრაციისა და დეშიფრაციისთვის. ალგორითმი ეფუძნება მათემატიკურ ოპერაციებს, კერძოდ, დიდი რიცხვების ფაქტორიზაციას.

RSA ალგორითმის უსაფრთხოება პირდაპირ არის დამოკიდებული იმაზე, რამდენად რთულია n -ის ფაქტორიზაცია ორ დიდ მარტივ რიცხვად. ფაქტორიზაციის სირთულე პრაქტიკულად შეუძლებელს ხდის საიდუმლო გასაღების გამოთვლას მხოლოდ ღია გასაღების საფუძველზე. RSA ალგორითმის ბიჯები:

1. გასაღების გენერაცია. RSA ალგორითმი იყენებს ორ გასაღებს: ღია (public key) და საიდუმლო (private key). გასაღების გენერაციის პროცესი ასეთია:

- ავირჩიოთ ორი მარტივი რიცხვი p და q ;
- გამოვითვალოთ მათი ნამრავლი $n = p * q$, სადაც n არის მოდული, რომელიც გამოიყენება როგორც ღია გასაღების ნაწილი;

- გამოვითვალოთ ეილერის ფუნქცია: $\phi(n) = (p - 1) * (q - 1)$;
- ავირჩიოთ რიცხვი e , რომელიც უნდა აკმაყოფილებდეს პირობას $1 < e < \phi(n)$ და $\gcd(e, \phi(n)) = 1$;

- გამოვთვალოთ d - მოდულარული ინვერსია e -სთან $\phi(n)$ -ის მიმართ ანუ

$$d * e \equiv 1 \pmod{\phi(n)};$$

- ღია გასაღები არის წყვილი (e, n) ;
- საიდუმლო გასაღები არის d .

2. შიფრაცია (დაშიფვრა). მონაცემთა შიფრაცია ხდება ღია გასაღების გამოყენებით: თუ m არის შეტყობინება (რიცხვი), მაშინ შიფრირებული ტექსტი c გამოითვლება ფორმულით: $c \equiv m^e \pmod{n}$

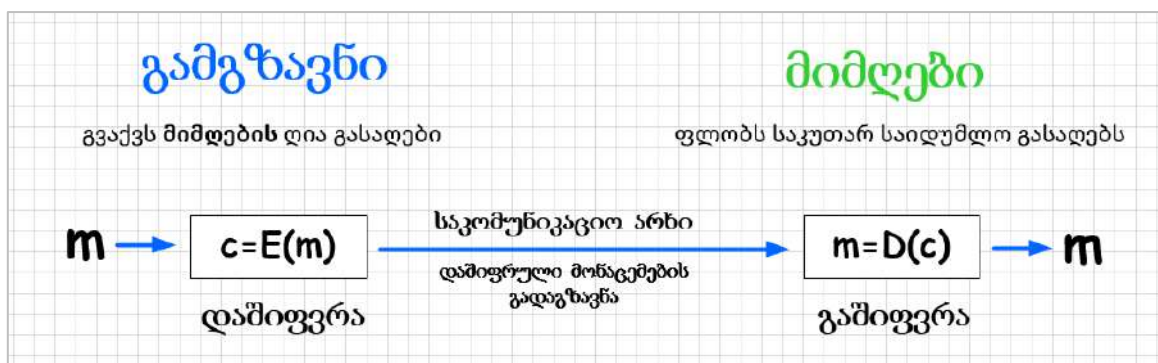
3. დეშიფრაცია (გაშიფვრა). დეშიფრაცია ხდება საიდუმლო გასაღების გამოყენებით: შიფრირებული ტექსტი c განიშიფრება ფორმულით: $m \equiv c^d \pmod{n}$, სადაც m არის დეშიფრირებული შეტყობინება.

RSA-ს უსაფრთხოება დამოკიდებულია იმაზე, თუ რამდენად რთულია n -ის (რომელიც არის ორი დიდი მარტივი რიცხვის ნამრავლი) ფაქტორიზაცია ორ ნაწილად: p და q ნაწილებად. პრაქტიკულად, დიდი ზომის n -ისთვის ამ ოპერაციის შესრულება არაეფექტური და ძალზე რთულია, რაც RSA-ს უსაფრთხო ალგორითმად აქცევს.

RSA ფართოდ გამოიყენება: **მონაცემთა შიფრაციაში** (უსაფრთხო კომუნიკაციისთვის, როცა საჭიროა ინფორმაცია მხოლოდ ავტორიზებულმა პირებმა წაიკითხონ) და **ციფრულ ხელმოწერებში** (მონაცემთა ნამდვილობის შესამოწმებლად და მათი ცვლილების თავიდან ასაცილებლად).

RSA-ს ძლიერი მხარე ისაა, რომ ის იძლევა შიფრაციის მაღალი დონის უსაფრთხოებას, მაგრამ ამავდროულად შედარებით ნელია სხვა შიფრაციის ალგორითმებთან შედარებით.

ასიმეტრიული შიფრაციის RSA ალგორითმის სქემა. შიფრაცია (დაშიფვრა) - დეშიფრაცია (გაშიფვრა). გამგზავნელს სურს გაუგზავნოს მიმღებს m ინფორმაცია. ინფორმაცია წარმოადგენს მთელ რიცხვებს 0 -დან $n - 1$ -მდე დიაპაზონში ე.ი. $m = Z_n$ (ნახ. 11.4.).



ნახ. 11.4. RSA ალგორითმის სქემა - დაშიფვრა და გაშიფვრა

გაგზავნის მხარის ალგორითმი:

1. ავირჩიოთ მიმღების ღია გასაღები (e, n) ;
2. ავირჩიოთ ღია ტექსტი m ;
3. c შეტყობინების დაშიფვრა მიმღების მხარის ღია გასაღების გამოყენებით:

$$c = E(m) = m^e \bmod n; \quad (11.1)$$

მიმღების მხარის ალგორითმი:

1. დაშიფრული c შეტყობინების მიღება;
2. ავირჩიოთ საიდუმლო/დაფარული გასაღები: (d, n) ;
3. შეტყობინების გასაშიფრად გამოვიყენოთ საიდუმლო/დაფარული გასაღები:

$$m = D(c) = c^d \bmod n; \quad (11.2)$$

(11.1) და (11.2) გამოსახულებები, რომლებზეც დაფუძნებულია RSA სქემა, განსაზღვრავს Z_n სიმრავლის ურთიერთშებრუნებულ გარდაქმნებს.

ვთქვათ, გამგზავნმა გაგზავნა დაშიფრული შეტყობინება **111111** და მიმღებმა, თავისი პირადი გასაღების გამოყენებით, გაშიფრა იგი.

განვიხილოთ RSA ალგორითმის ეტაპები (ცხრილი 11.1).

ცხრილი 11.1. RSA ალგორითმის ეტაპების განხილვა

ეტაპები	ოპერაცია	ოპერაციის შედეგი
გასაღების გენერირება	ავირჩიოთ ორი მარტივი რიცხვი	$p = 3557 \quad q = 2579$
	გამოვითვალთ ნამრავლი	$n = p * q = 3557 * 2579 = 9173503$
	ეილერის ფუნქციის გამოთვლა	$\varphi(n) = (p - 1) * (q - 1) = 9167368$
	ავირჩიოთ ღია ექსპონენტი	$e = 3$
	გამოვითვალთ ფარული ექსპონენტი	$d = e^{-1} \bmod \varphi(n)$ $d = 6111579$
	გამოვაქვეყნოთ „ღია გასაღები“	$\{e, n\} = \{3, 9173503\}$
	შევიწახოთ „საიდუმლო გასაღები“	$\{d, n\} = \{6111579, 9173503\}$
დაშიფვრა	ავირჩიოთ დასაშიფრი ტექსტი	$m = 111111$
	გამოვითვალთ შიფრტექსტი	$c = E(m) = m^e \bmod n =$ $= 111111^3 \bmod 9173503 = 4051753$
გაშიფვრა	გამოთვალეთ საწყისი ტექსტი	$m = D(c) = c^d \bmod n =$ $= 4051753^{6111579} \bmod 9173503 =$ $= 111111$

RSA სისტემის ფუნქციონირების ალგორითმი. მოცემული გვაქვს ორი მარტივი რიცხვი p და q რიცხვები. მათ საფუძველზე იანგარიშება $n = p * q$ და ეილერის ფუნქცია: $\varphi(n) = (p - 1) * (q - 1)$

n რიცხვის ათობითი თანრიგების სიდიდეა და განსაზღვრავს ასიმეტრიული შიფრაციის ალგორითმის მთავარ პარამეტრს - გასაღების სიგრძეს.

ვირჩევთ შემთხვევით დიდ რიცხვს $d > 1$, რომლისათვისაც სრულდება პირობა $\text{mod}(d, \varphi(n)) = 1$ და ვანგარიშობთ e რიცხვს, შემდეგი პირობების დაცვით $1 < e < \varphi(n)$ და $(e * d, \text{mod } \varphi(n)) = 1$. ამ რიცხვებიდან e და n ქმნიან ღია გასაღებთა წყვილს, რომლებიც გამოიყენება ინფორმაციის დასაშიფრად, ხოლო რიცხვები p, q, d და $\varphi(n)$ ქმნიან საიდუმლო გასაღების წყვილს.

ეს წყვილი გამოიყენება დაშიფრული ინფორმაციის გასაშიფრად. ცხადია, რომ თუნდაც ერთ-ერთი p ან q რიცხვის ცოდნა იწვევს საიდუმლო გასაღების სხვა შემადგენლების განსაზღვრას და მთლიანობაში სისტემის კომპრომეტირებას.

განვიხილოთ შიფრაციის თანამიმდევრობა:

1. ხდება საწყისი ინფორმაციის თითოეული სიმბოლოს კოდირება ათობითი ათვლის სისტემაში (შესაძლებელია ორობითშიც);

2. ხდება მიღებული ინფორმაციის დაყოფა კრიპტოტექსტის ბლოკებად. ბლოკის სიგრძე i იანგარიშება შემდეგი უტოლობიდან: $10^{i-1} \leq n < 10^i$

3. რეკომენდებულია ბლოკის სიგრძედ აღებული იქნეს i -ის მნიშვნელობა და არა $(i-1)$.

4. შემდეგში ხდება თითოეული კრიპტოტექსტის w ინდივიდუალური ბლოკების დაშიფვრა: $c = (w^e, \text{mod } n)$, სადაც c წარმოადგენს საწყისი ტექსტის w ბლოკის დაშიფრულ ბლოკს.

5. ხდება მიღებული დაშიფრული c ბლოკების განშიფვრა: $w = (c^d, \text{mod } n)$

თუ შევადარებთ სიმეტრიული და ასიმეტრიული შიფრაციის ალგორითმებს, ცალსახად ჩანს ასიმეტრიული შიფრაციის ალგორითმის სიმარტივე. p და q დაბალი მნიშვნელობებისათვისა (< 100 ათობით თანრიგისათვის) განმარტებული.

RSA მეთოდს ახასიათებს ორი მნიშვნელოვანი სირთულე:

1. მძლავრი გამოთვლითი რესურსების მიმართ მოთხოვნის არსებობა და მნიშვნელოვანი დროითი დანახარჯები ($n > 1000$ ათობით თანრიგისთვის) p და q მარტივი რიცხვების გამოთვლისას.

მაგალითად, ორ ბირთვიან პერსონალურ კომპუტერზე ≈ 5 საათია საჭირო 1500-თანრიგიანი (ათობითი) მარტივი რიცხვის მოსაძებნად;

2. დეშიფრაციის პროცესში პრაქტიკულად შეუძლებელია, მისაღებ პრაქტიკულ დროში, $w = (c^d, \text{mod } n)$ გამოთვლის ჩატარება სპეციალური ალგორითმების გამოყენების გარეშე. ეს ალგორითმი ცნობილია ხარისხში მოდულური აყვანის სახელით.

ხარისხში მოდულური აყვანის არსი მდგომარეობს შემდეგში: N რიცხვი ორობით სისტემაში შეიძლება წარმოვიდგინოთ შემდეგნაირად:

$$N = \sum_{j=0}^k x_j * 2^j$$

სადაც $x_j = 0, 1$; $k = (\log_2 N) + 1$; $0 \leq j \leq k$;

ეს ნიშნავს იმას, რომ ნებისმიერი $e * d$ მნიშვნელობებისთვის შესრულება მხოლოდ $(k - 1)$ რაოდენობის ნამრავლი და თითოეული ნამრავლისთვის მოხდება $\text{mod}(n)$ -ის გამოთვლა.

გამოვითვალთ: $7^{23} \text{ mod } 61$

ცხადია, რომ $23_{10} = 10111_2$. შესაბამისად მივიღებთ შემდეგ მნიშვნელობებს:

j	0	1	2	3	4
$(7^2)^j$	7	49	22	54	16

მეორე სტრიქონის ელემენტების და რიცხვი 23-ის ორობით ფორმის გათვალისწინებით მივიღებთ: $7^{23} \text{ mod } 61 = (16 * 22 * 49 * 7) \text{ mod } 61 = 17$

თუ ჩვეულებრივად 7^{23} გამოსათვლელად საჭირო არის 23 ცალი გამრავლების ჩატარება - ხარისხში მოდულური აყვანის ალგორითმის მიხედვით გვაქვს მხოლოდ 4 გამრავლება. მიახლოებით ცნობილია, რომ 100 ბიტის სიდიდის სიმეტრიული შიფრაციის გასაღების სიგრძე, დაცულობის თვალსაზრისით, ტოლფასია ასიმეტრიული შიფრაციის 700 თანრიგიანი გასაღების. შესაბამისად დიდი მოცულობის საწყისი ინფორმაციების დასაშიფრად რეკომენდებულია (დროითი რესურსების შიფრაცია/დეშიფრაციაზე დანახარჯების გათვალისწინებით) გამოყენებული იქნას სიმეტრიული შიფრაციის ალგორითმები. თუმცა ასიმეტრიული შიფრაციის RSA სისტემა შეუცვლელია ელექტრონული ხელმოწერის პროცედურების პროგრამული უზრუნველყოფის შესაქმნელად.

მაგალითი 11.1. ვთქვათ, მოცემულია გასაღებების შემდეგი მნიშვნელობები:

- საიდუმლო გასაღები: $p = 47$, $q = 59$, $\phi(n) = (p - 1) * (q - 1) = 2668$, $d = 157$;
- ღია გასაღები: $n = p * q = 2773$, $e = 17$.

ასევე შექმნილი გვაქვს ცხრილი ქართული ანბანის მხოლოდ საწყისი ასო-ნიშნებით (იხ. ნახ. 11.5.). ბლოკის სიგრძე $i = 4$.

ვთქვათ, საწყისი ტექსტია: აი მზე

ნახ. 11.5.-ის მიხედვით ვიღებთ: $\omega = 0109\ 2512\ 0705$

ახლა გადავიდეთ თითოეული ბლოკის დაშიფვრაზე:

$$c_1 = 109^7 \text{ mod } n = 170$$

$$c_2 = 2512^7 \text{ mod } n = 2362$$

$$c_3 = 0705^7 \text{ mod } n = 1962$$

შესაბამისად დაშიფრული ტექსტი იქნება:

$$c = 0170\ 2362\ 1692$$

გასაშიფრად ვიყენებთ საიდუმლო გასაღებს: $d = 157$

გადავიდეთ თითოეული ბლოკის დაშიფვრაზე:

$$\omega_1 = c_1^{157} \bmod n = 0109$$

$$\omega_2 = c_2^{157} \bmod n = 2512$$

$$\omega_3 = c_3^{157} \bmod n = 0705$$

ა	01	ლ	11
ბ	02	მ	12
გ	03	ნ	13
დ	04	ო	14
ე	05	პ	15
ვ	06	რ	16
ზ	07	ს	17
თ	08	ტ	18
ი	09	და ა.შ.	...
კ	10	Space	25

ნახ. 11.5. ქართული ანბანის საწყისი ასო-ნიშნები

საბოლოოდ მივიღებთ $\omega = 0109\ 2512\ 0705$

შესაბამისი ალფაბეტიური გამოსახულება: აი მზე.

ასიმეტრიული შიფრაცია, მარტივი რიცხვების გენერაცია. ფერმას მცირე თეორემა არის რიცხვთა თეორიის ერთ-ერთი მარტივი თეორემა, რომლის მიხედვითაც ნებისმიერი p მარტივი რიცხვისთვის და მისი a თანამარტივი (ამ შემთხვევაში - არაჯერადი) რიცხვისთვის სრულდება შემდეგი იგივეობა: $a^{p-1} \equiv 1 \pmod{p}$, სადაც $\pmod{p}$ აღნიშნავს იგივეობის შესრულებას p -ს მოდულით (ანუ იგივეობის ორივე ნაწილი ერთმანეთის ტოლი არ არის. მხოლოდ მათი p -ზე გაყოფის ნაშთებია ტოლი). მოდულის გარეშე თეორემა შეგვიძლია შემდეგნაირად ჩამოვაყალიბოთ:

გამოსახულება $a^{p-1} - 1$ იყოფა რიცხვ p -ზე ნებისმიერი p და a -სთვის.

ფერმას დიდი თეორემისგან განსხვავებით ფერმას მარტივი თეორემა საკმაოდ ადვილი დასამტკიცებელია. ფერმას თეორემას რამდენიმე განზოგადება აქვს.

ერთ-ერთი განზოგადოებული მათგანი გვეუბნება, რა მოხდება იმ შემთხვევაში თუ მარტივი რიცხვის მაგივრად მარტივი რიცხვის ხარისხს განვიხილავთ.

ასეთ შემთხვევაში, თუ a არის p -ს თანამარტივი და p^n კი - p მარტივი რიცხვის რაიმე ხარისხია, მაშინ გვაქვს შემდეგი იგივეობა: $a^{p^{n-1}(p-1)} \equiv 1 \pmod{p^n}$

ზოგადად ნებისმიერი n რიცხვისთვის, თუ განვმარტავთ ეილერის მაჩვენებელს $\Phi(n)$ -ით, რომელიც უდრის 1-დან n -მდე n -ის თანამარტივ რიცხვთა რაოდენობას, მაშინ ნებისმიერი n -ის თანამარტივი რიცხვი a -სთვის სრულდება ფერმას თეორემის შემდეგი განზოგადებული სახე, რომელსაც ეილერის თეორემა ეწოდება: $a^{\Phi(n)} \equiv 1 \pmod{n}$

პრაქტიკულად მარტივი რიცხვის მოძებნა ხორციელდება შემდეგნაირად: ვიღებთ მარტივ დიდ რიცხვს S , ვირჩევთ შემთხვევით კენტ რიცხვს R , რიცხვების დიაპაზონიდან $R \leq 4 * S + 2$ და ვანგარიშობთ სავარაუდო მარტივ რიცხვს $N = S * R + 1$. მიღებული რიცხვის შემოწმება სიმარტივეზე ხდება ელემენტარული ტესტით ცხრილებში მოყვანილი პატარა მარტივ რიცხვებზე გაყოფით. ამ ეტაპზე ასეთი სახის შემოწმება დიდი ალბათობით ვერ ადგენს მიღებული N რიცხვის სიმარტივეს და ამიტომაც რეკომენდებულია მარტივი რიცხვების დიაპაზონი განვსაზღვროთ ≈ 30000 .

თუ ეს მინიმალური პირობა არ შესრულდა, ვიღებთ R რიცხვის სხვა მნიშვნელობას და შემოწმების ციკლს ვიმეორებთ, სანამ არ მივიღებთ სასურველ შედეგს. დამატებით ამ შემთხვევაში, მიღებულ შედეგს (კერძოდ, N და R რიცხვებს) ვამოწმებთ ევკლიდეს გაფართოებული ალგორითმით და თუ $უსგ = 1$, რაღაც ალბათობით ვთვლით, რომ შედეგი დამაკმაყოფილებელია. თუ მიღებული N რიცხვის სიდიდე არ დაგვაკმაყოფილებს, ძებნის ციკლს ვიმეორებთ, ოღონდ საწყის სიდიდედ ვიღებთ N და ა.შ სანამ არ მივიღებთ სასურველ შედეგს.

RSA კრიპტოსისტემა რიცხვთა თეორიაში. RSA კრიპტოსისტემა ეყრდნობა რიცხვთა თეორიაში მიღებულ შედეგებს და მისი ძირითადი შინაარსი შეიძლება შემდეგნაირად ჩამოყალიბდეს:

შესაძლებელია ვიპოვოთ მთელი რიცხვები e, d და n ისეთი, რომ ნებისმიერი მთელი m რიცხვისათვის, რომელიც აკმაყოფილებს უტოლობას $0 \leq m < n$, ადგილი აქვს ტოლობას $(m^e)^d \equiv m \pmod{n}$, სადაც $\equiv$ კონგრუენტულობის სიმბოლოა. როგორც ვხედავთ, ამ ფორმულაში ძირითად ოპერაციებს წარმოადგენს ახარისხება და მოდულური გამოთვლები.

1. კრიპტოსისტემის შექმნა იწყება n -ის განსაზღვრით: $n = p * q$, სადაც p და q მარტივი რიცხვებია. რიცხვთა მცირე დიაპაზონისთვის ეს ადვილად კეთდება, რადგან $p * q$ პოვნა იოლი ოპერაციაა, მაგრამ ამოცანა მოულოდნელად გართულდება თუ p და q იქნება დიდი მარტივი რიცხვები. წამოიჭრება შეკითხვა: როგორ ვიპოვოთ დიდი მარტივი რიცხვები? ამისათვის შეიძლება გამოვიყენოთ ერატოსთენეს საცერი, რომელიც საშუალებას იძლევა შევადგინოთ მარტივი რიცხვების სია, რომელიმე დიდ N რიცხვამდე.

რამდენადაც მარტივი რიცხვები საკმაოდ მკვრივად არიან განლაგებული მთელ რიცხვთა სიმრავლეში, ამიტომ მივიღებთ ძალიან დიდ სიას, რომლის დამახსოვრება და

გამოყენება პრობლემებთანაა დაკავშირებული. ეხლა წარმოვიდგინოთ რეალური სიტუაცია, როცა მოითხოვება, ვთქვათ, 512 ბიტის მარტივი რიცხვების გამოყენება.

ამ შემთხვევაში უკვე ცხრილების გამოყენება სრულიად არაეფექტურია და ამ მიზნით დამუშავებულია სპეციალიზებული online კალკულატორები, რომლებიც საშუალებას იძლევა, შემოწმდეს რომელიმე დასახელებული რიცხვი სიმარტივეზე.

ამ შემთხვევაში მოწმდება არის თუ არა აღებული დიდი რიცხვი მარტივი. თუ არა აიღება ახალი რიცხვი და ეს გრძელდება, სანამ არ იქნება ნაპოვნი დიდი მარტივი რიცხვი. ამგვარად, საბოლოოდ განისაზღვრება n -ის მნიშვნელობა.

RSA კრიპტოსისტემაზე შეტყვის ძირითადი მიმართულებაა n -ს მარტივ მამრავლებად დაშლა, რადგან p და q განსაზღვრის შემთხვევაში, როგორც ქვემოთ ვნახავთ, ადვილია კრიპტოსისტემის გატეხვა.

ამიტომ RSA სისტემაში გამოიყენება მართლაც დიდი რიცხვები, რომელთა მარტივ მამრავლებად დაშლას იმდენად დიდი დრო სჭირდება, რომ ამ მიმართულებით მცდელობას აზრი არ აქვს და ეს მდგომარეობა კიდევ დიდ ხანს იქნება შენარჩუნებული.

2. მთელი რიცხვების e და d განსაზღვრისათვის გამოიყენება n -ის მნიშვნელობა. კერძოდ, უნდა განისაზღვროს ე.წ. კარმაიკლის ფუნქცია $\Lambda(n)$ -ის მნიშვნელობა n -თვის.

ჩვენი შემთხვევისთვის როცა $n = p * q$, სადაც p და q მარტივი რიცხვებია, ეს ფუნქცია ადვილად გამოითვლება:

$$\Lambda(n) = LCM(p-1, q-1)$$

სადაც LCM (Least Common Multiple) $p-1$ და $q-1$ რიცხვების უმცირესი საერთო ჯერადია (უსჯ). დიდი რიცხვების შემთხვევაში უსჯ-ის გამოთვლისათვის გამოიყენება სათანადო კალკულატორი, აგებული ევკლიდეს ალგორითმზე.

მას შემდეგ, რაც ნაპოვნია $\Lambda(n)$, შეიძლება განისაზღვროს e და d .

e შეიძლება იყოს ნებისმიერი რიცხვი, რომელიც აკმაყოფილებს პირობებს $e < \Lambda(n)$, სადაც e და $\Lambda(n)$ ურთიერთმარტივია.

რაც შეეხება d -ს იგი არის e -ს მულტიპლიკაციური ინვერსია მოდულით $\Lambda(n)$ ანუ $e * d = 1 \bmod \Lambda(n)$

ამგვარად განსაზღვრული n , e და d პირობებში ღია გასაღებს წარმოადგენს წყვილი (e, n) , ხოლო დახურულ გასაღებს წყვილი (d, n) .

ცხადია, p, q და $\Lambda(n)$ საიდუმლო რიცხვებია, რომლებიც კრიპტოსისტემის შედგენის შემდეგ ნადგურდებიან (ან საიდუმლოდ ინახებიან ყოველი შემთხვევისთვის).

მაგალითი 11.2. ვთქვათ, $p=3$ და $q=11$ ანუ $n=33$.

$$\Lambda(n) = LCM(p-1, q-1) = LCM(2, 10) = 10, \text{ ანუ } \Lambda(33) = 10$$

ავიღოთ $e=3$, რადგან $3 < 10$ და 3 და 10 ურთიერთმარტივია.

გამოვთვალოთ d .

რადგან $3 * d = 1 \bmod 10$, გვექნება $d = 7$.

ამგვარად, ღია გასაღები იქნება: $(3, 33)$

ხოლო საიდუმლო გასაღები - $(7, 33)$

მაგალითი 11.3. განვიხილოთ მაგალითი საშუალო დიაპაზონის მონაცემებით, როცა ხელით გამოთვლები პრაქტიკულად აზრს კარგავს.

RSA კრიპტოსისტემის შექმნა გულისხმობს შემდეგ ნაბიჯებს:

1. ავიღოთ ორი მარტივი რიცხვი $p = 101$, $q = 199$. ამ რიცხვების სიმარტივეზე შესამოწმებლად გამოვიყენოთ primality test online calculator.

2. ვიპოვოთ $n = 101 * 199 = 20099$, $n = 20099$. ამ ნამრავლის გამოსათვლელად გამოვიყენოთ Big Numbers Multiplication Calculator

3. ვიპოვოთ კარმაიკლის ფუნქციის მნიშვნელობა:

$\lambda(n) = \lambda(20099) = LCM(100, 198) \lambda(n) = 9900$ ვისარგებლოთ Online Carmichael Function Calculator.

4. ავიღო $e = 29$ და შევამოწმოთ e და $\lambda(n)$ ურთიერთსიმარტივეზე. ვისარგებლოთ კალკულატორით Coprime Calculator.

5. ვიპოვოთ e -ს მოდულური მულტიპლიკაციური ინვერსია $29 * d \equiv 1 \bmod 9900$ ვისარგებლოთ კალკულატორით Modular Multiplicative Inverse Calculator. **$d = 7169$** .

6. დავშიფროთ ASCII ასო-ნიშანი, რომლის ათობითი ნომერია 49. გამოვთვალოთ შიფრი $m = 49$ -თვის. გვექნება $c = 49^{29} \bmod 20099$.

ვისარგებლოთ კალკულატორით Modular exponentiation. **$C = 16540$** .

7. მოვახდინოთ დეშიფრაცია. $m \equiv c^d \bmod 20099$.

ვისარგებლოთ Modular exponentiation კალკულატორით. **$m = 49$** .

8. ამგვარად, შექმნილი RSA კრიპტოგრაფული სისტემის ღია გასაღებია $(e, n) = (29, 20099)$, ხოლო საიდუმლო $(7169, 20099)$.

განხილული მაგალითი წარმოდგენას გვაძლევს RSA კრიპტოსისტემასთან დაკავშირებულ გამოთვლებზე, მაგრამ მხედველობაში უნდა იქნეს მიღებული, რომ რეალური სისტემები გაცილებით დიდ რიცხვებთან არის დაკავშირებული და სწორედ ეს უზრუნველყოფს RSA სისტემის საიმედოობას.

მაგალითი 11.4. RSA კრიპტოსისტემის გამოყენების მაგალითი. განვიხილოთ ურთიერთკავშირი (კომუნიკაცია) ანასა (A) და ბექას (B) შორის.

ურთიერთკავშირის (კომუნიკაციის) წამომწყებია ანა (A). ბექასათან (B -სთან) დასაკავშირებლად ანას სჭირდება ბექას მხრიდან ღია გასაღები გზავნილის დასაშიფრად.

შევადგინოთ B -ს ღია და დახურული გასაღებები. ამისათვის:

- ავირჩიოთ ორი მარტივი რიცხვი $p = 13$ და $q = 7$.
- გამოვთვალოთ $n = p * q = 13 * 7 = 91$.

შემდგომში რიცხვი $n = 91$ შიფრაცია/დეშიფრაციის ალგორითმებში გამოიყენება როგორც მოდული (Modules), ხოლო 91-ის ორობით წარმოდგენაში ბიტების რაოდენობა, როგორც შიფრის გასაღების (key) სიდიდე. ჩვენს შემთხვევაში, რადგან $6 * y < 91 < 128$, შიფრის სიდიდე იქნება $key = 7$.

- გამოვთვალოთ უმცირესი საერთო ჯერადი:

$Z = LCM(p - 1, q - 1) = LCM(12, 6) = 12$. რიცხვი $Z = 12$ გამოიყენება შიფრაციის ექსპონენტის e და დეშიფრაციის ექსპონენტის d გამოთვლაში.

- ავირჩიოთ e -ს მნიშვნელობა, რომელიც ურთიერთმარტივი იქნება Z -თან, არ ექნება საერთო გამყოფი Z -თან და ნაკლები იქნება Z -ზე. ასეთად გამოდგება $e = 5$, რადგან 5-სა და 12-ს საერთო გამყოფი არ აქვთ. e გამოიყენება როგორც ღია გასაღების კომპონენტი.

- გამოვთვალოთ დეშიფრაციის ექსპონენტი d .

რადგან d არის e -ს მოდულური ინვერსია მოდულით z , ამიტომ $(d * e) \bmod 12 = 1$ ანუ $(d * 5) \bmod 12 = 1$. ამ განტოლებას აკმაყოფილებს $d = 5$. ამგვარად ბეჭად ღია გასაღები იქნება $(e, n) = (5, 91)$, ხოლო საიდუმლო $(d, n) = (5, 91)$.

დავუშვათ, რომ ანა აგზავნის ტექსტს ASCII კოდის მაღალი რეგისტრების სიმბოლოების გამოყენებით, რომელთაც შეესაბამებათ ათობითი რიცხვები (ASCII კოდში რიგითი ნომერი) როგორც ნახატ 11.6-ზე არის ნაჩვენები.

აქ A -ს შეესაბამება რიცხვი 65, B -ს შეესაბამება 66 და ა.შ.

A	B	C	D	E	F	G	H	I	J	K	L	M
65	66	67	68	69	70	71	72	73	74	75	76	77
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
78	79	80	81	82	83	84	85	86	87	88	89	90

ნახ. 11.6. ანას მიერ გაგზავნილი ტექსტის ASCII კოდის მაღალი რეგისტრების სიმბოლოები, რომელთაც ათობითი რიცხვები შეესაბამებათ

დავუშვათ, რომ ანას მიერ გზავნილია $M = HELLO$. ამისათვის საჭიროა ეს გზავნილი წარმოდგეს, როგორც რიცხვი m , მაგრამ მეორე მხრივ მოითხოვება, რომ აკმაყოფილებდეს უტოლობას $0 \leq m < n$ ანუ $m < 91$. ამ მოთხოვნას აკმაყოფილებს გზავნილის თითოეული სიმბოლო, ხოლო რამდენიმე სიმბოლოს ერთად წარმოდგენა n -ის გაცილებით დიდ მნიშვნელობას მოითხოვს. ამიტომ, პრაქტიკულად გზავნილი HELLO უნდა წარმოდგეს გზავნილების მიმდევრობად H, E, L, L, O და წარმოდგება რიცხვით. რომელიც მისი რიგითი ნომრის ტოლია.

ამგვარად, ანა ახდენს გზავნილის თითოეული სიმბოლოს შიფრაციას ბექას ღია გასაღებით $(e, n) = (5, 91)$, როგორც ნახატ 11.7-ზე არის ნაჩვენები.

სიმბოლო	H	E	L	L	O	ანას მხარე
სიმბოლოს ნომერი, m	72	69	76	76	79	
შიფრი, $c = m^e \bmod n$	11	62	20	20	53	

ნახ. 11.7. ანას (A) მხარე

განვიხილოთ, მაგალითად სიმბოლო H . მისთვის გვაქვს $m = 72$, ხოლო ამ სიმბოლოს შიფრი არის $c = m^e \bmod n = 72^5 \bmod 91 = 11$.

სიმბოლო E -სთვის გვექნება $c = m^e \bmod n = 69^5 \bmod 91 = 62$ და ა.შ. (გამოთვლები სრულდება მოდულური ახარისხების online კალკულაციით).

ამგვარად, ანა ბექას უგზავნის ასო-ნიშნების შიფრების კრებულს 11, 62, 20, 20, 53.

ბექა, გზავნილის მიღების შემდეგ ახდენს მათ დეშიფრაციას თავისი დახურული გასაღების $(d, n) = (5, 91)$ გამოყენებით, როგორც ნახატ 11.8-ზეა ნაჩვენები:

მიღებული შიფრები c	11	62	20	20	53	ბექას მხარე
დეშიფრაცია $m = c^d \bmod n$	72	69	76	76	79	
სიმბოლოები	H	E	L	L	O	

ნახ. 11.8. ბექას (B) მხარე

აქ გამოთვლები წარმოებს ფორმულით $m = c^d \bmod n$.

შიფრ 11-სთვის გვექნება $m = 11^5 \bmod 91 = 72$ (ანუ H),

62-სთვის გვექნება $m = 62^5 \bmod 91 = 69$ (ანუ E) და ა.შ.

ცხადია m -ის შესაძლო მნიშვნელობა $m < n$.

მაგალითად, თუ შიფრის გასაღები n იქნება 1024 ბიტისანი, მაშინ რამდენიმე სიმბოლოს ნომრები შეგვეძლო წარმოგვედგინა როგორც ერთი 6 თანრიგიანი ათობითი რიცხვი m , რაც გაართულებს გამოთვლებს და გაამარტივებს პროცედურებს. საერთოდ უნდა ითქვას, რომ ასიმეტრიული კრიპტოსისტემები (RSA), გაცილებით ნელა მუშაობს, ვიდრე კრიპტოსისტემები სიმეტრიული გასაღებით, როგორცაა, მაგალითად, AES.

RSA კრიპტოსისტემის ერთ-ერთ ღირსებად უნდა ჩაითვალოს, რომ იგი მნიშვნელოვნად ამარტივებს გასაღებების მართვას. RSA კრიპტოსისტემის განხილვისას შიფრაცია/დეშიფრაციის ალგორითმი დაფუძნებულია ტოლობაზე: $(m^e)^d \equiv 1 \bmod n$, სადაც m მონაცემია (ღია ტექსტია), e შიფრაციის ექსპონენტია, d - დეშიფრაციის ექსპონენტია, ხოლო n მოდულია. ეს ტოლობა შეიძლება გადავწეროთ შემდეგნაირად: $(m^e)^d = m^{e*d} = (m^d)^e$,

რაც ნიშნავს, რომ შეგვიძლია მოვიქცეთ სხვანაირად, კერძოდ, შეიძლება მოვახდინოთ შიფრაცია საიდუმლო გასაღებით ანუ ვიპოვოთ $m = c^e$.

ეს მიდგომა გამოყენებას პოულობს ინფორმაციის ნამდვილობის დადგენისა და ციფრული ხელმოწერის ალგორითმებში.

დასკვნის სახით. დაშიფრვის ალგორითმები იყოფა ორ კატეგორიად: **სიმეტრიული და ასიმეტრიული შიფრაცია**. ამ ორი შიფრაციის მეთოდის ფუნდამენტური სხვაობა დამოკიდებულია იმ ფაქტზე, რომ სიმეტრიულ კრიპტოგრაფიაში გამოიყენება მხოლოდ ერთი გასაღები, ხოლო ასიმეტრიულ კრიპტოგრაფიაში ორი განსხვავებული, მაგრამ ერთმანეთთან დაკავშირებული გასაღები. ეს სხვაობა, რომელიც საკმაოდ უბრალოდ შეიძლება მოგვეჩვენოს, ადგენს ფუნქციონალურ სხვაობას ამ სისტემების შიფრაციის ტექნიკისა და მათი გამოყენების მეთოდებს შორის. სიმეტრიული კრიპტოგრაფიის მაღალი სიჩქარიდან გამომდინარე, სიმეტრიული შიფრაცია ფართოდ გამოიყენება მრავალი თანამედროვე კომპიუტერული სისტემების ინფორმაციულ უსაფრთხოებაში. ასიმეტრიული შიფრაცია შეიძლება იყოს გამოყენებული ისეთ სისტემაში, სადაც ბევრმა მომხმარებელმა უნდა დაშიფროს და განშიფროს შეტყობინებები ან მონაცემთა ნაკრებები, განსაკუთრებით, იმ შემთხვევებში როცა სიჩქარე და გამოთვლის სიმძლავრე არ არის პირველადი ამოცანა. ასეთი სისტემის ერთ-ერთი მაგალითია დაშიფრული ელექტრონული ფოსტა, რომელშიც ღია გასაღები გამოიყენება შეტყობინების დასაშიფრად, ხოლო ფარული გასაღები - ამ შეტყობინების დეშიფრაციისთვის.

ჰიბრიდული სისტემები. ბევრ პროგრამაში სიმეტრიული შიფრაცია ასიმეტრიულთან ერთად გამოიყენება. ასეთი ჰიბრიდული სისტემის მაგალითებია Security Sockets Layer (SSL) და Transport Layer Security (TLS) კრიპტოგრაფიული პროტოკოლები, რომლებიც შეიქმნა ინტერნეტში უსაფრთხო კავშირის (კომუნიკაციის) დასამყარებლად. SSL პროტოკოლები დღეისთვის ითვლება საფრთხის შემცველ პროტოკოლად და მისი გამოყენება უნდა შეწყდეს. ამასთან ერთად, TLS პროტოკოლები ითვლება უსაფრთხოდ და ინტენსიურად გამოიყენება ყველა ძირითად ვებ ბრაუზერში.

ჰემ-ფუნქციები. ჰემირება არის პროცესი, რომლის მეშვეობითაც ნებისმიერი ზომის მონაცემები გარდაიქმნება ფიქსირებული ზომის სტრიქონად. კრიპტოგრაფიული ჰემ-ფუნქციები (ტექსტის ანაბეჭდის ფუნქციები) წარმოადგენენ კრიპტოგრაფიული ალგორითმების მნიშვნელოვან კლასს. ისინი იღებენ საწყის მნიშვნელობად ტექსტს და უკან აბრუნებენ ფიქსირებული სიგრძის ჰემს, რომლის დაკავშირება საწყის მნიშვნელობასთან პრობლემას წარმოადგენს. ასეთ ფუნქციებს ცალმხრივ ფუნქციებსაც ეძახიან. საუკეთესო ალგორითმებისათვის კოლიზიები (ორი ტექსტი, რომელთა ჰემი ერთი და იგივეა ანუ მათი

ჰეშები ერთმანეთს ემთხვევა) რთული მოსაძებნია და ამის ალბათობა მინიმუმამდე უნდა იქნეს დაყვანილი.

ჰეშ-ფუნქციის გამოყენების შემთხვევებია:

- **პაროლების დაცვა.** მომხმარებელი ქმნის პაროლს, სისტემა ჰეშირების ალგორითმის მიხედვით ქმნის ჰეშირებულ მნიშვნელობას და ინახავს მას. შემდგომში, როდესაც მომხმარებელი შედის სისტემაში, პაროლი კვლავ ჰეშირდება და ხდება მისი შედარება ჰეშირებულ პაროლთან ბაზაში;
- **მონაცემთა ხელმოწერა.** ჰეშირება გამოიყენება მონაცემების ვალიდაციისთვის, სადაც ჰეშის მნიშვნელობა შეიძლება იყოს ხელმოწერის ნაწილი, რათა დადასტურდეს მონაცემთა ხელუხლებლობის ფაქტი;
- **შეტყობინების ნამდვილობის (აუთენტიფიკაციის) კოდები;** შეტყობინების ნამდვილობის (აუთენტიფიკაციის) კოდები ჰეშფუნქციების მსგავსია, იმ განსხვავებით, რომ ჰეშ-მნიშვნელობის შესამოწმებლად გამოიყენება საიდუმლო გასაღები.

მონაცემთა ჰეშირება. თანამედროვე კრიპტოგრაფიული ჰეშფუნქციები ფართო გამოყენებას პოულობს კრიპტოგრაფიაში. კრიპტოგრაფიის განვითარების კვალობაზე თანდათანობით გამოიკვეთა გამკაცრებული მოთხოვნები ჰეშირების პროცედურების საიმედოობისადმი, რამაც განაპირობა კრიპტოგრაფიული ჰეშირების ალგორითმის შექმნა, რომლებიც მჭიდროდ არის დაკავშირებული კრიპტოგრაფიული შიფრაცია/დეშიფრაციის მეთოდებთან.

კრიპტოგრაფიული ჰეშფუნქცია. კრიპტოგრაფიული ჰეშფუნქცია H წარმოადგენს ფუნქციას, რომელიც ნებისმიერი ზომის ბიტურ მონაცემს m გარდასახავს ფიქსირებული სიგრძის ბიტურ მონაცემად, რასაც m -ის ჰეში ეწოდება.

ჰეშფუნქციის განსაზღვრის არეს წარმოადგენს ნებისმიერი ბიტური მონაცემი, რადგან თუ საწყისი მონაცემი წარმოდგენილია სხვა, მაგალითად, ტექსტური ფორმით, ხდება მისი ორობით მონაცემად გარდაქმნა სიმბოლოების ბიტური კოდირების გამოყენებით, რაც შეეხება ამ ფუნქციის მნიშვნელობათა სიმრავლეს, იგი შეიძლება იყოს ფიქსირებული სიგრძის, მაგალითად 128 ბიტიანი, 256 ბიტიანი ან სხვა სიგრძის ორობითი მონაცემი.

ამგვარად ჰეშფუნქციისათვის ვლემულობთ ფუნქციურ ასახვას: $H(m) = h$, სადაც ჰეში m -ის სახეა (emage), ხოლო m არის h -ის პრეველსახე (preemage).

როგორც ითქვა, m შეიძლება იყოს ნებისმიერი ზომის ბიტური რიცხვი და $H(m)$ ფუნქციის განსაზღვრის არე ძალიან ფართოა (შემოუსაზღვრავია), ხოლო ჰეში, თუ მას განვიხილავთ როგორც ორობით რიცხვს, მოქცეულია დიაპაზონში: $0 \div 2^d - 1$, სადაც d არის ჰეშის ბიტების რაოდენობა.

ფუნქცია $H(m)$ -ის მოთხოვნაა ის, რომ იგი ცალმხრივი ფუნქციაა: m -თვის ადვილად (სწრაფად) ხდება მისი სახის (ანასახის) დადგენა, მაგრამ მოცემული h -ათვის მისი წინასახის დადგენა ანუ $h = H^{-1}(m)$ გარდასახვის განხორციელება რთული და არაცალსახაა.

კრიპტოგრაფიული ჰეშ-ფუნქციის თვისებები. m მონაცემის ჰეშის მნიშვნელობა h შეიძლება განვიხილოთ როგორც „თითის ანაბეჭდი“, იგი იმის მსგავსად ახასიათებს მონაცემს, როგორც თითის ანაბეჭდი ადამიანს: თითოეული ადამიანს შეესაბამება უნიკალური თითის ანაბეჭდი და უკიდურესად ძნელია ვიპოვოთ ორი ადამიანი, რომელთაც ერთნაირი თითის ანაბეჭდი გააჩნიათ, რაც ფართოდ გამოიყენება კრიმინოლოგიურ გამოკვლევებში.

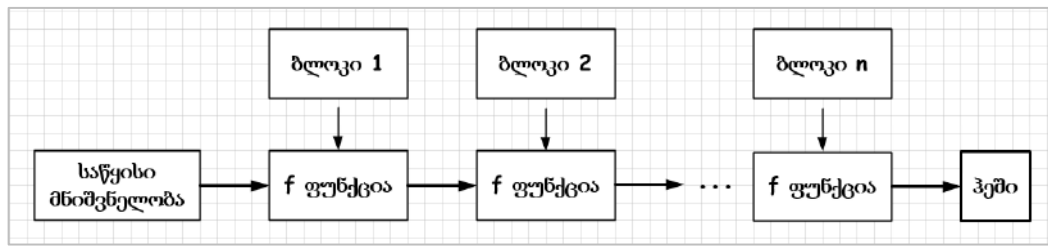
იმისათვის, რომ ჰეშ-ფუნქცია იყოს საიმედო ინსტრუმენტი კრიპტოგრაფიაში, მას უნდა გააჩნდეს შემდეგი თვისებები:

- გამოთვლითი თვალსაზრისით რთული უნდა იყოს h -ის მიხედვით მისი პირველ-სახის m -ის პოვნა. ეს უკიდურესად გაუძნელებს კრიპტოგრაფიულ შეტევას ჰაკერს, რომელიც ცდილობს h -ით იპოვოს m ;
- თუ H ფუნქცია m_1 მონაცემისთვის იძლევა $H(m_1) = h$, ძნელი უნდა იყოს ისეთი m_2 მონაცემის პოვნა, რომლისთვისაც $H(m_2) = h$ ე.ი. ჰეშის ერთ მნიშვნელობას ორი განსხვავებული მნიშვნელობა არ უნდა შეეფარდებოდეს; ეს ნიშნავს, რომ თუ ჰაკერმა იცის მონაცემი m და მისი ჰეში h , მისთვის ძნელი უნდა იყოს m -ის სხვა მონაცემით ჩანაცვლება;
- რთულია ისეთი m_1 და m_2 პოვნა, რომ $H(m_1) = H(m_2)$. ეს თვისება გამორიცხავს კოლიზიებს მნიშვნელობათა შორის და განსაკუთრებით მკაცრ მოთხოვნას წარმოადგენს;
- ჰეშ-ფუნქცია არის დეტერმინირებული ფუნქცია, რაც ნიშნავს, რომ ერთიდაიგივე m -თვის H ფუნქციის გამოყენება უნდა იძლეოდეს ერთსადაიმდე პასუხს;
- ნებისმიერი m -თვის $H(m)$ -ის გამოთვლა სწრაფი უნდა იყოს;
- ჰეშ ფუნქცია უნდა ხასიათდებოდეს ზვავისებური ეფექტით. ეს ნიშნავს, რომ ორი m_1 და m_2 ერთმანეთისგან განსხვავდება მცირედ, თუნდაც ერთი ბიტი. ჰეშ-ფუნქციის შესრულების პროცესში ეს სხვაობა უნდა გაძლიერდეს და მიღებულ იქნეს h_1 და h_2 ჰეშები, რომლებიც, ერთმანეთისგან ძლიერ განსხვავდებიან.

ჰეშირების ალგორითმი. როგორც შიფრაციის, ისე ჰეშირების ალგორითმებში მონაცემი m იყოფა ტოლი სიდიდის ბლოკებად $m_1, m_2, m_3, \dots, m_n$ დაწყებული მარცხნიდან.

საჭიროების შემთხვევაში, თუ m მონაცემის სიგრძე ბლოკის სიგრძის ჯერადი არ არის, ხდება მისი შევსება დამატებითი მონაცემებით მონაცემის ბოლოში (იხ. ნახ. 11.9.).

აქ გამოთვლები დაყოფილია ეტაპებად. ყოველ ეტაპზე f ფუნქცია ასრულებს დაშიფრვის ოპერაციას, სადაც მონაცემთა ბლოკს იყენებს როგორც გასაღებს (key), ხოლო წინა ეტაპიდან გადმოცემულ შიფრს როგორც დასაშიფრავ მონაცემს.



ნახ. 11.9. ჰეშირების ალგორითმის გამარტივებული სქემა

პირველ ეტაპზე გამოიყენება საწყისი მონაცემი (seed) და ბლოკი 1, მიღებული შედეგი გადაეცემა მეორე ბლოკს და ა.შ. ბლოკების ამოწურვამდე. მიღებული შედეგი წარმოადგენს საწყისი m მონაცემის ჰეშს. ამგვარად, თუ ბლოკის სიგრძე ბაიტებში იქნება, მაგალითად, 128 ბიტი, ფუნქცია f ყოველ ნაბიჯზე შესავალზე იღებს ორ 128 ბიტს და გამოსავალზე გამოიმუშავებს ისევ 128 ბიტს, რომელიც გადაეცემა მომდევნო ეტაპს. საბოლოოდ შეიძლება ითქვას, რომ აღწერილი პროცედურა ახდენს ბლოკის მონაცემების თანდათანობით შეკუმშვას, რის შედეგად გამოსავალზე მიიღება m მონაცემის 128 ბიტის ჰეში.

წლების მანძილზე დამუშავებული იქნა ჰეშირების ალგორითმების რამდენიმე ოჯახი, რომელთა გავრცელება ხდება სახელით - უსაფრთხო ჰეშირების ალგორითმი, SHA (Secure Hash Algorithm), ორგანიზაცია NIST (National Institute of Standards and Technology) რეკომენდაციით დამუშავებულ იქნა ოჯახები SHA-1, SHA-2, SHA-3, SHA-4, სადაც მომდევნო ოჯახი ხასიათდება გაზრდილი ეფექტურობით.

ამჟამად ყველაზე მეტად გავრცელებულია SHA-2, რომელშიც შედის SHA-256, ჰეშალგორითმი, რომელიც წარმოქმნის 256 ბიტის ჰეშს და SHA-512 512 ბიტის ჰეშით.

ჰეშირების მეთოდი, შიფრაცია/დეშიფრაციის მეთოდებთან ერთად, გამოიყენება მონაცემთა ნამდვილობის დასადგენად, მონაცემთა უცვლელობის დასაცავად, ელექტრონული ხელისმოწერაში და სხ.

მონაცემების შეკუმშვა. ხოფმანის (ჰაფმანის) კოდები. ფაქსიმილური აპატები.

მონაცემების/ინფორმაციის შეკუმშვა. მონაცემების/ინფორმაციის შეკუმშვა არის პროცესი, რომელიც უზრუნველყოფს მონაცემთა მოცულობის შემცირებას ისე, რომ მოხდეს მესხიერების ნაკლები ადგილის გამოყენება ან ნაკლები სიჩქარე ინფორმაციის გადაცემისას.

მონაცემთა შეკუმშვა შეიძლება დაიყოს ორ ძირითად ტიპად: ინფორმაციის უდანაკარგოდ შეკუმშვა (Lossless Compression) და ინფორმაციის დანაკარგით შეკუმშვა (Lossy Compression).

1. ინფორმაციის უდანაკარგოდ შეკუმშვა (Lossless Compression) - მთლიანად ხდება საწყისი ინფორმაციის აღდგენა. უდანაკარგოდ შეკუმშვის შემდეგ ხდება მონაცემების სრულად აღდგენა, ზუსტად ისეთი სახით, როგორებიც ისინი შეკუმშვამდე იყვნენ. იგი გამოიყენება ისეთ მონაცემებთან, სადაც მნიშვნელოვანია მათი 100%-ით აღდგენა, მაგალითად, ტექსტური დოკუმენტების ან პროგრამების შეკუმშვისას.

მაგალითად, ZIP, RAR (ფაილის ფორმატები); PNG (გამოსახულების ფორმატი); FLAC (აუდიო ფორმატი).

უდანაკარგოდ ინფორმაციის შეკუმშვა გამოყენებითი მათემატიკის, კერძოდ, კი ინფორმაციის თეორიის ერთ-ერთი დარგია. ის შეისწავლის ინფორმაციის შეკუმშვის ისეთ მეთოდებს, რომელთა გამოყენებითაც საწყისი ინფორმაციის ზუსტი ასლის აღდგენა შეიძლება. თითოეული ტიპის მონაცემებისთვის, როგორც წესი, არსებობს ოპტიმალური შეკუმშვის ალგორითმები.

ინფორმაციის უდანაკარგოდ შეკუმშვის მთავარი პრინციპი მდგომარეობს იმაში, რომ ის ცდილობს აღმოფხვრას მონაცემთა სტრუქტურაში არსებული ასლები ან მოჭრას არასაჭირო განმეორებადი ინფორმაცია.

2. ინფორმაციის დანაკარგით შეკუმშვა (Lossy Compression) - საწყისი ინფორმაციის აღდგენის პროცესში ხდება მონაცემების დამახინჯება. დანაკარგით შეკუმშვა უზრუნველყოფს ინფორმაციის მოცულობის მაქსიმალურ შეკუმშვა/შემცირებას, მაგრამ მონაცემების ნაწილობრივი დაკარგვის ხარჯზე. დანაკარგით შეკუმშვა ძირითადად გამოიყენება მედია ფაილებისთვის (მაგალითად, ვიდეო და აუდიო ფაილებისთვის), სადაც მცირე ხარისხის დაკარგვა მომხმარებლისთვის შეუმჩნეველია ან უმნიშვნელოა. მაგალითად, JPEG (გამოსახულების ფორმატი); MP3 (აუდიო ფორმატი); MPEG (ვიდეო ფორმატი). ტექსტური, ციფრული და ცხრილური სახის მონაცემებისთვის არ არის რეკომენდებული, ისეთი პროგრამების გამოყენება, რომლებიც იყენებენ დანაკარგით შეკუმშვის მეთოდებს. ინფორმაციის დანაკარგით შეკუმშვისას მონაცემთა ზოგიერთი ნაწილი იკარგება, რაც ხშირად ხდება

ისეთი სახის ელემენტების ამოშლით, რომლებიც ადამიანის აღქმისთვის ნაკლებად მნიშვნელოვანია.

მონაცემების/ინფორმაციის შეკუმშვის ალგორითმები:

- უდანაკარგოდ შეკუმშვის ალგორითმები: ჰაფმანის (Huffman Coding) კოდირება და Run-Length Encoding (RLE) კოდირება;
- დანაკარგით შეკუმშვის ალგორითმები: Discrete Cosine Transform (DCT) და Discrete Wavelet Transform (DWT).

მონაცემთა შეკუმშვის ალგორითმი (არქივირების ალგორითმი). მონაცემთა შეკუმშვის ალგორითმი არის ისეთი სახის ალგორითმი, რომელიც გამორიცხავს მონაცემთა ჩაწერის სიჭარბეს.

კოდის ალფავიტი/ანბანი. კოდის ალფავიტი/ანბანი არის შემოსული ნაკადის ყველა სიმბოლოს ერთობლიობა. ინგლისურენოვანი ტექსტების შეკუმშვისას, გამოიყენება 128 ASCII კოდების სიმრავლე. გამოსახულების შეკუმშვისას პიქსელების ნაკრები შეიძლება შეიცავდეს 2, 16, 256 ან სხვა რაოდენობის ელემენტებს.

კოდური სიმბოლო. კოდური სიმბოლო არის მონაცემთა ყველაზე მცირე ერთეული, რომელიც ექვემდებარება შეკუმშვას. როგორც ვიცით, ერთი სიმბოლო არის 1 ბაიტი, მაგრამ იგი შეიძლება იყოს ბიტიც, კვანტური ტრიტიც {0,1,2} ან რაიმე სხვა.

კოდური სიტყვა. კოდური სიტყვა არის ანბანიდან კოდის სიმბოლოების თანამიმდევრობა. თუ ყველა სიტყვას აქვს ერთნაირი სიგრძე (სიმბოლოების რაოდენობა), მაშინ ასეთ კოდს უწოდებენ თანაბარს ანუ ფიქსირებული სიგრძის კოდს, ხოლო თუ სიტყვები სხვადასხვა სიგრძისაა, მაშინ კოდს ეწოდება არათანაბარი ანუ ცვლადი სიგრძის კოდი.

კოდი. კოდი არის სიტყვების სრული ნაკრები.

კოდირება. კოდირება არის მონაცემთა შეკუმშვის პროცესი.

დეკოდირება. დეკოდირება არის კოდირების საპირისპირო პროცესი, რომლის დროსაც ხდება მონაცემების აღდგენა.

შეკუმშვის თანაფარდობა. შეკუმშვის თანაფარდობა არის ერთ-ერთი ყველაზე ხშირად გამოყენებული სიდიდე, რომელიც მიუთითებს შეკუმშვის მეთოდის ეფექტურობაზე.

$$\text{შეკუმშვის თანაფარდობა} = \frac{\text{გამომავალი ნაკადის ზომა}}{\text{შემომავალი ნაკადის ზომა}}$$

მაგალითად, მნიშვნელობა 0.6 ნიშნავს, რომ მონაცემების შეკუმშვის შემდეგ ინფორმაცია იკავებს თავდაპირველი ზომის 60%-ს.

თუ თანაფარდობის მნიშვნელობა 1-ზე მეტია, ეს ნიშნავს, რომ გამომავალი ნაკადი უფრო დიდია, ვიდრე შემომავალი ნაკადი (ანუ საქმე გვაქვს უარყოფით შეკუმშვასთან ან გაფართოებასთან).

შეკუმშვის კოეფიციენტი (Compression Ratio). შეკუმშვის კოეფიციენტი არის მაჩვენებელი, რომელიც მიუთითებს, თუ რამდენად ეფექტურად შეიკუმშა მონაცემები. იგი გვიჩვენებს, შეკუმშვის შემდეგ თუ რამდენად შემცირდა მონაცემების ზომა პირვანდელ ზომასთან შედარებით. შეკუმშვის კოეფიციენტი შეკუმშვის თანაფარდობის საპირისპირო სიდიდეა.

$$\text{შეკუმშვის კოეფიციენტი} = \frac{\text{შემომავალი ნაკადის ზომა}}{\text{გამომავალი ნაკადის ზომა}}$$

შეკუმშვის კოეფიციენტის 1-ზე მეტი მნიშვნელობა მიუთითებს შეკუმშვაზე, ხოლო 1-ზე ნაკლები მნიშვნელობა მიუთითებს გაფართოებაზე.

მაგალითად, თუ ფაილის ზომა არის 1000 ბაიტი და შეკუმშვის შემდეგ მისი ზომა გახდა 250 ბაიტი, შეკუმშვის კოეფიციენტი იქნება:

$$\text{შეკუმშვის კოეფიციენტი} = \frac{1000}{250} = 4$$

მიღებული შეკუმშვის კოეფიციენტის სიდიდე ნიშნავს, რომ მონაცემები შეკუმშულია 4-ჯერ და თავდაპირველი ზომის 25%-ია.

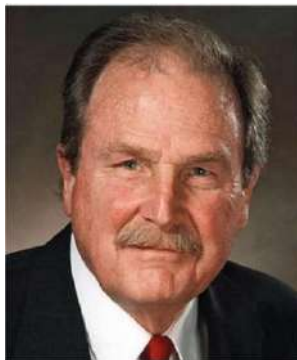
არსებობს ინფორმაციის/მონაცემების შეკუმშვის ორი ძირითადი მეთოდი:

სტატისტიკური მეთოდი. სტატისტიკური მეთოდი არის შეკუმშვის ისეთი მეთოდი, რომლებიც ცვალებადი სიგრძის კოდებს ანიჭებენ შემოსული ნაკადის სიმბოლოებს. საუკეთესო სტატისტიკური მეთოდები იყენებენ **ჰაფმანის** კოდირებას.

ლექსიკონური შეკუმშვა. ლექსიკონური შეკუმშვა არის შეკუმშვის ისეთი მეთოდი, რომელიც მონაცემთა ფრაგმენტებს ინახავს „ლექსიკონში“ (მონაცემთა გარკვეული სტრუქტურა). თუ შემავალი მონაცემების ახალი სტრიქონი იგივეა ანუ ზუსტად მსგავსია ლექსიკონში უკვე არსებული ზოგიერთი ფრაგმენტისა, მაშინ ასეთ ფრაგმენტის მიმთითებელი მოთავსებული იქნება გამომავალ ნაკადში. საუკეთესო ლექსიკონური მეთოდები გამოიყენება ლემპელ-ზივის მეთოდში. მაგალითად, ნებისმიერი კომპიუტერული არქივატორი (ZIP, RAR) იყენებს უდანაკარგო შეკუმშვის მეთოდებს. ზოგიერთი ინფორმაციის შეკუმშვა შესაძლებელია ასევე დანაკარგებითაც. მაგ., JPEG ფორმატში შესაძლებელია ნახატების შეკუმშვა მათი ხარისხის გაუარესების ხარჯზე. ასეთი შეკუმშვის შედეგად ჩვენ ვეღარ აღვადგენთ საწყის ნახატს შეკუმშული ნახატიდან, რადგან ინფორმაციის ნაწილი დაიკარგა. ინფორმაციის უდანაკარგოდ შეკუმშვის ყველაზე მარტივი და ხშირად გამოყენებული ალგორითმი, რომელიც ასევე ყველა სხვა ალგორითმის ძირითადი პრინციპია, არის ჰაფმანის კოდირების ალგორითმი. ყველა კოდს შორის ჰაფმანის კოდირება ითვლება ოპტიმალურ კოდად თუ მონაცემთა გამოჩენის სიხშირეები (ალბათობები) წინასწარ გვაქვს მოცემული. თუმცა, რადგან ეს ძირითადად ასე არ არის, პრაქტიკაში უფრო რთული კოდები გამოიყენება.

მაგალითად, ჰაფმანის ალგორითმზე დაფუძნებული, მაგრამ შედარებით უფრო რთული და ეფექტური ალგორითმებია: 1. **არიტმეტიკული კოდირების ალგორითმი**, რომელიც ჰაფმანის კოდირების მოდიფიკაციაა გრძელი გზავნილებისთვის (ფაილებისთვის); 2. **ლემპელ-ზივის ალგორითმი**, რომლის მცირე მოდიფიკაციები ძირითადად გამოიყენება თანამედროვე შემკუმშვის პროგრამებში.

ჰაფმანის კოდირების ალგორითმი (Huffman Algorithm). ინფორმაციის კოდირების/შეკუმშვის ერთ-ერთი ალგორითმი აღწერა ამერიკელმა მეცნიერმა დ.ა. ჰაფმანმა 1952 წელს (იხ. ნახ. 12.1.). ჰაფმანის კოდირება (შეკუმშვა) არის ინფორმაციის შეკუმშვის ფართოდ გამოყენებული მეთოდი, რომელიც ალფავიტის/ანბანის სიმბოლოებს ანიჭებს ცვლადი სიგრძის კოდებს, ამ სიმბოლოების გამოჩენის სიხშირის მიხედვით. მოკლედ რომ ავხსნათ ჰაფმანის კოდირების (შეკუმშვის) დროს, ტექსტის შეკუმშვა ხდება ისე, რომ ხშირად გამოყენებულ სიმბოლოებს მიენიჭებათ მოკლე კოდები, ხოლო იშვიათად გამოყენებულ სიმბოლოებს უფრო გრძელი კოდები.



ნახ. 12.1. დ.ა. ჰაფმანი, ერთ-ერთი პირველი მეცნიერი ინფორმაციის თეორიის დარგში

ჰაფმანის კოდირების ალგორითმი არის ცნობილი კოდირების მეთოდი, რომელიც გამოიყენება მონაცემთა შეკუმშვისთვის. მისი მთავარი მიზანია, სიმბოლოებს მიანიჭოს მინიმალური სიგრძის ბინარული კოდები მათი სიხშირის მიხედვით, ისე რომ უფრო ხშირი სიმბოლოები მიიღებენ მოკლე კოდებს, ხოლო იშვიათი სიმბოლოები - უფრო გრძელ კოდებს. ეს მეთოდი ძალიან ეფექტურია ინფორმაციის შესაკუმშად, რადგან იძლევა საშუალებას მინიმუმამდე დაიყვანოს კოდირებული მონაცემების სიგრძე. თუ ცნობილია სიმბოლოების შეცვლის სიხშირე, შეგვიძლია აღვწეროთ ცვლადი სიგრძის კოდების აგების პროცედურა, რომელიც შედგება ბიტების მთელი რიცხვისგან. უფრო სავარაუდოდ სიმბოლოებს უფრო მოკლე კოდები ენიჭებათ. ამრიგად, ამ მეთოდით, როდესაც მონაცემები შეკუმშულია, თითოეულ სიმბოლოს ენიჭება ოპტიმალური პრეფიქსული კოდი, ტექსტში მისი გაჩენის სიხშირის მიხედვით.

პრეფიქსული კოდი. პრეფიქსული კოდი არის ისეთი კოდი, რომელშიც არცერთი კოდური სიტყვა არ არის სხვა კოდური სიტყვის პრეფიქსი. ასეთი კოდები ცვლადი სიგრძისაა. **ოპტიმალური პრეფიქსული კოდი** არის ისეთი პრეფიქსული კოდი, რომელსაც აქვს მინიმალური საშუალო სიგრძე.

ჰაფმანის კოდირების ალგორითმი შეიძლება დაიყოს ორ ნაწილად:

1. **სიმბოლოების წარმოდგენის სიხშირის განსაზღვრა.** თავდაპირველად საჭიროა მთლიანად წაიკითხოთ საწყისი ტექსტი და დათვალოთ სიმბოლოების გამოჩენის სიხშირე (ზოგჯერ ითვლიან რამდენჯერ გვხვდება თითოეული სიმბოლო).

2. **ოპტიმალური პრეფიქსის კოდის პოვნა.** მოვძებნოთ ისეთი a და b სიმბოლოები, რომელთა გამოჩენის სიხშირე ყველაზე ნაკლებია. ისინი შევცვალოთ ერთი ფიქტიური x სიმბოლოთი, რომლის გამოჩენის სიხშირე ტოლია a და b სიმბოლოების გამოჩენის სიხშირის ჯამისა. შემდეგ, ხდება ამ პროცედურის რეკურსიული გამოყენება. ვპოულობთ ოპტიმალურ კოდს სიმბოლოების უმცირესი რაოდენობისთვის (სადაც სიმბოლოები a და b შეიცვლება ერთი სიმბოლო x -ით). სიმბოლოების საწყისი ნაკრების კოდი მიიღება შემცვლელი სიმბოლოების კოდების წინ 0-ის ან 1-ის დამატებით და ეს ორი ახალი კოდი მიიღება როგორც შემცვლელი სიმბოლოების კოდები. მაგალითად, სიმბოლო a -ს კოდი შეესაბამება ისეთ სიმბოლო x -ს, რომლის წინ ნულია ჩაწერილი, ხოლო სიმბოლო b -ს კი შეესაბამება x -ის წინ ჩაწერილი ერთიანი.

ჰაფმანის კოდირების იდეა ეფუძნება სტრიქონში სიმბოლოს გამოჩენის სიხშირეს. სიმბოლო, რომელიც ყველაზე უფრო ხშირად გვხვდება თანამიმდევრობაში, იღებს ახალ და ძალიან მოკლე კოდს, ხოლო სიმბოლო, რომელიც გვხვდება შედარებით ნაკლები სიხშირით, იღებს, პირიქით, ძალიან გრძელ კოდს. მეთოდი ეფექტურად ამცირებს მონაცემების ზომას.

ჰაფმანის კოდირების განსაზღვრა: ვთქვათ $A = \{a_1, a_2, \dots, a_n\}$ არის n რაოდენობის განსხვავებული სიმბოლოს მქონე ანბანი, ხოლო $W = \{w_1, w_2, \dots, w_n\}$ არის ანბანის შესაბამისი დადებითი მთელი რიცხვების წონების ნაკრები, მაშინ ბინარული კოდების ნაკრებს $C = \{c_1, c_2, \dots, c_n\}$, სადაც c_i -ური არის a_i -ური სიმბოლოს კოდი, ისე რომ:

- c_i -ური არ წარმოადგენს c_j -ურის პრეფიქსს, როცა $i \neq j$,
- $\sum_{i \in [1, n]} w_i * |c_i|$ გამოსახულების ჯამი მინიმალურია ($|c_i|$ არის c_i კოდის სიგრძე),

უწოდებენ ჰაფმანის კოდს.

ჰაფმანის კოდირების ხის აგების ალგორითმი.

განვიხილოთ ჰაფმანის კოდირების ხის აგების ალგორითმის ბიჯები:

1. თითოეული სიმბოლოს (ან სიმბოლოების ჯგუფის) სიხშირეების დათვლა: უნდა დავთვალოთ თითოეული სიმბოლოს სიხშირე ტექსტში ან მონაცემებში. შედარებით ხშირად გამოყენებული სიმბოლოები მიიღებენ უფრო მოკლე კოდებს. სიმბოლოში იგულისხმება სტრიქონში შემავალი განმეორებადი ელემენტი: ეს შეიძლება იყოს დაბეჭდილი სიმბოლო (character) ან ბიტის ნებისმიერი თანამიმდევრობა. კოდის ქვეშ კი - არა ASCII ან UTF-8 სიმბოლოს კოდი, არამედ ბიტების კოდირების თანამიმდევრობა.

დავუშვათ, გვაქვს ტექსტი: *ABBCCCDDDD*. დავითვალოთ თითოეული სიმბოლოს სიხშირე: *A*-ს სიხშირეა 1; *B*-ს სიხშირეა 2; *C*-ს სიხშირეა 3; *D*-ს სიხშირეა 4.

2. პრიორიტეტული რიგის შექმნა. თითოეული სიმბოლოს სიხშირის მიხედვით უნდა შევქმნათ პრიორიტეტული რიგი. რიგში შედარებით დაბალი სიხშირის სიმბოლოები განთავსდება რიგის დასაწყისში.

3. სიმბოლოების სიხშირის მიხედვით ხდება ორობითი ხის აგება:

- ავირჩიოთ რიგიდან ორი ყველაზე დაბალი სიხშირის სიმბოლოები;
- შექმენით მათი მშობელი კვანძი/სიმბოლო, რომლის სიხშირე იქნება არჩეული ორი სიმბოლოს სიხშირეთა ჯამი;
- რიგში გადაიტანეთ ახალი კვანძი/სიმბოლო შესაბამისი პოზიციაში, სიხშირის მიხედვით;
- გაიმეორეთ ეს პროცესი მანამ, სანამ არ მიიღებთ მხოლოდ ერთ კვანძს/სიმბოლოს, რომელიც იქნება ხის ფესვი.

4. სიმბოლოებისთვის კოდების მინიჭება: მიღებულ ხეში, ყოველი მარცხენა სიმბოლოსთვის გამოიყენეთ 0, ხოლო მარჯვენასთვის - 1. ასე განისაზღვრება თითოეული სიმბოლოს კოდი. შედეგად, თითოეული სიმბოლო მიიღებს უნიკალურ კოდს, რომელიც დამოკიდებულია სიმბოლოების სიხშირეზე. **ხშირად გამოყენებულ სიმბოლოებს აქვთ უფრო მოკლე კოდები, ხოლო იშვიათად გამოყენებულ სიმბოლოებს - უფრო გრძელი.**

ალგორითმის სირთულე: ჰაფმანის კოდირების ხის აგება ხორციელდება $O(n * \log(n))$ დროში, სადაც n არის სიმბოლოების რაოდენობა, რადგან რიგის განახლება ხდება მონაცემთა ყოველი გაერთიანების დროს.

მაგალითი 12.1. სტრიქონის დამიფვრა ჰაფმანის ალგორითმით. ჰაფმანის კოდირება დაყოფილია ორ ეტაპად: პირველ ეტაპზე ხორციელდება ჰაფმანის ალგორითმი, ხოლო მეორე ეტაპზე ხდება ჰაფმანის ხის აგება. მოცემული გვაქვს სტრიქონი: ***abracadabra***

მოდით დავშიფროთ სიტყვა: *abracadabra*

სიმბოლოების სიმრავლე იქნება: $A = \{a, b, r, c, d\}$, ხოლო სიხშირე კი $W = \{5, 2, 2, 1, 1\}$

ჰაფმანის ხეში გვექნება ხუთი კვანძი/სიმბოლო:

კვანძი/სიმბოლო:	<i>a</i>	<i>b</i>	<i>r</i>	<i>c</i>	<i>d</i>
სიხშირე:	5	2	2	1	1

ალგორითმის მიხედვით ავირჩიოთ ორ სიმბოლო ყველაზე დაბალი სიხშირით - ესეთი სიმბოლოებია: *c* და *d*. ჩამოვყალიბოთ მათგან ახალი კვანძი *cd* წონით 2 და დავამატოთ იგი კვანძების/სიმბოლოების სიაში:

კვანძი/სიმბოლო:	<i>a</i>	<i>b</i>	<i>r</i>	<i>cd</i>
სიხშირე:	5	2	2	2

შემდეგ კვლავ ვაერთიანებთ მინიმალური წონის ორ კვანძს ერთ კვანძში: *r*-ს და *cd*-ს:

კვანძი/სიმბოლო:	<i>a</i>	<i>rcd</i>	<i>b</i>
სიხშირე:	5	4	2

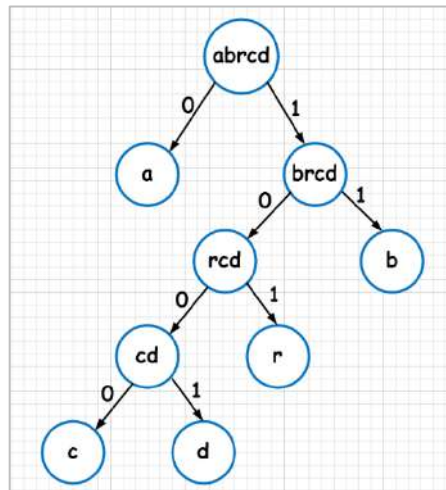
გავიმეორეთ იგივე ოპერაცია, მაგრამ კვანძებისთვის *rcd*-სთვის და *b*-სთვის:

კვანძი/სიმბოლო:	<i>brcd</i>	<i>a</i>
სიხშირე:	6	5

ბოლო ეტაპზე გავაერთიანოთ ორი კვანძი *brcd* და *a*:

კვანძი/სიმბოლო:	<i>abrcd</i>
სიხშირე:	11

დარჩა ერთი კვანძი, რაც ნიშნავს, რომ მივედით ჰაფმანის ხის ფესვთან (იხ. ნახ. 12.2). თითოეული სიმბოლოსთვის დავსვით კოდური სიტყვა (ორობითი თანამიმდევრობა, რომელიც აღნიშნავს გზას ხეზე ფესვიდან ამ სიმბოლომდე). ახლა კი, თითოეული სიმბოლოს კოდის მისაღებად, უნდა გავიაროთ ხე და ყოველ სიმბოლოზე გადასვლისას დავუმატოთ 0, თუ მივდივართ მარცხნივ, ხოლო 1, თუ მივდივართ მარჯვნივ:



ნახ. 12.2. ხეზე ორობითი თანამიმდევრობის გზა ფესვიდან სიმბოლომდე

შესაბამისად მივიღეთ:

კვანძი/სიმბოლო:	<i>a</i>	<i>b</i>	<i>r</i>	<i>c</i>	<i>d</i>
კოდი:	0	11	101	1000	1001

ამგვარად სიტყვა *abracadabra* დაშიფრული სახით არის 01110101000010010111010 დაშიფრული სიტყვის სიგრძე – 23 ბიტი.

აღსანიშნავია, რომ თუ გამოვიყენებდით კოდირების ალგორითმს ყველა კოდის სიტყვის ერთნაირი სიგრძით, მაშინ კოდირებული სიტყვა იქნება 33 ბიტის ტოლი, რომელიც გაცილებით დიდია.

მაგალითი 12.2. მოცემული გვაქვს სიმბოლოების თანამიმდევრობა: ***beep boop beer!*** დავშიფროთ მოცემული სიმბოლოების ნაკრები ჰაფმანის კოდირებით და დასმული ამოცანა გადავწყვიტოთ ვიზუალური სქემების დახმარებით (ნახ. 12.3.ა,ბ,გ,დ,ე, ვ,ზ).

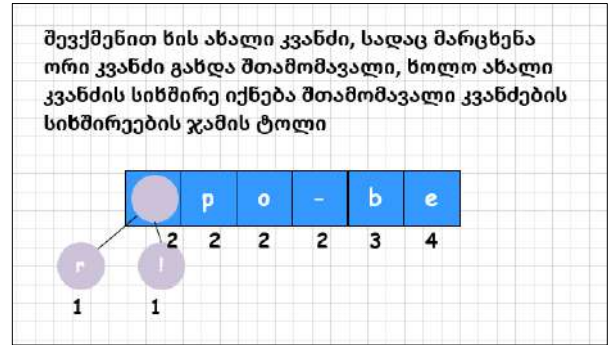
თავდაპირველად დავადგინოთ სიმბოლოების სიხშირე

სიმბოლო	b	e	p	-	o	r	!
სიხშირე	3	4	2	2	2	1	1

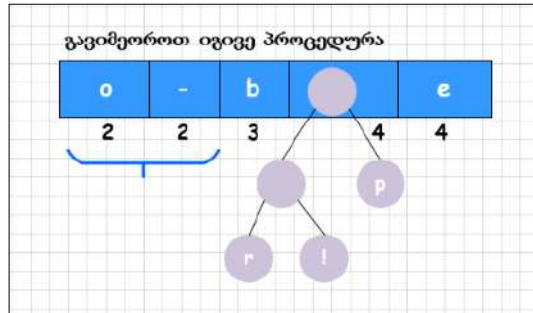
თითოეული სიმბოლოსთვის შევქმნათ ბინარული ხის კვანძები და დავუმატოთ ისინი რიგში, სიხშირის პრიორიტეტების მიხედვით:

სიმბოლო	r	!	p	o	-	b	e
სიხშირე	1	1	2	2	2	3	4

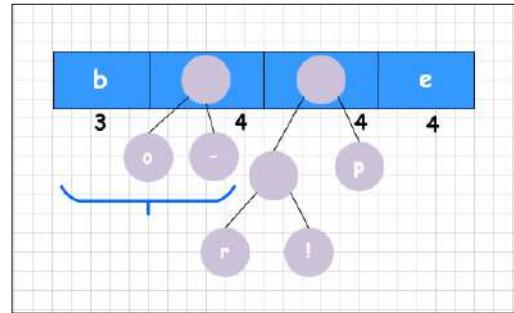
ა)



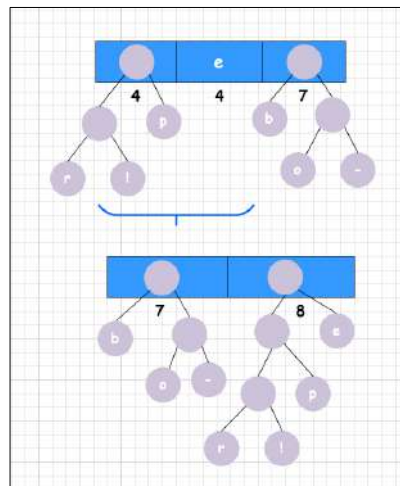
ბ)



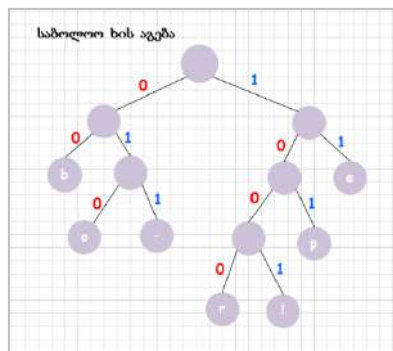
გ)



დ)



ე)



ვ)

კოდის მარტივი ცხრილი

სიმბოლო	b	e	p	-	o	r	!
კოდი	00	11	101	011	010	1000	1001

შემოსული სტრიქონი:
0110 0010 0110 0101 0110 0101 0111 0000 0010 0000 0110 0010
0110 1111 0110 1111 0111 0000 0010 0000 0110 0010 0110 0101
0110 0101 0111 0010 0010 000 = 120 ბიტი

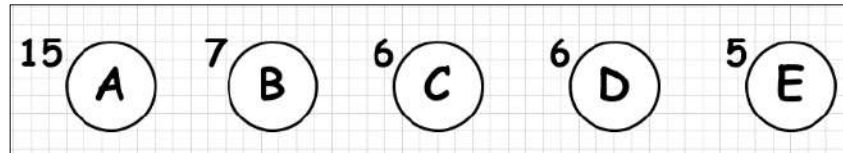
დამიფრული სტრიქონი:
0011 1110 1011 0001 0010 1010 1100 1111 1000 1001 = 40 ბიტი

ზ)

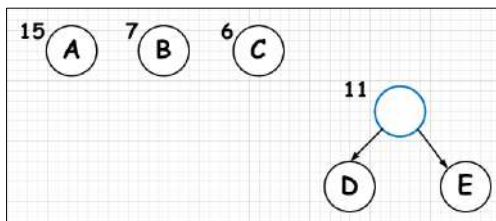
ნახ. 12.3. სიმბოლოთა ნაკრების beep boop beer! დაშიფვრა ჰაფმანის კოდირებით:

ა) პირველი; ბ) მეორე; გ) მესამე; დ) მეოთხე; ე) მეხუთე; ვ) მეექვსე; ზ) მეშვიდე - სქემები

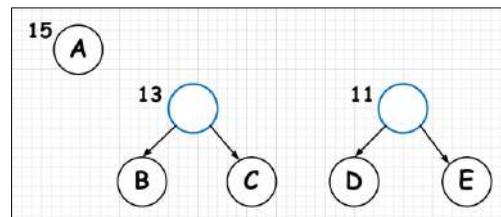
მაგალითი 12.3. ხის აგება ჰაფმანის ალგორითმის გამოყენებით. წარმოვიდგინოთ ტექსტი, რომელიც შედგება მხოლოდ A, B, C, D და E სიმბოლოებისგან. მათი გამოვრების სიხშირეა 15, 7, 6, 6 და 5. ნახატ 12.4. ა,ბ,გ,დ,ე-ზე მოცემულია ჰაფმანის ალგორითმის მიხედვით ხის აგების ეტაპები.



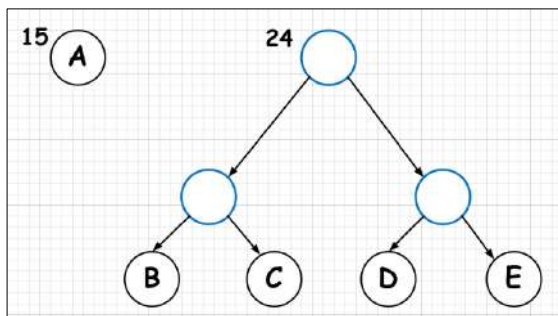
ა)



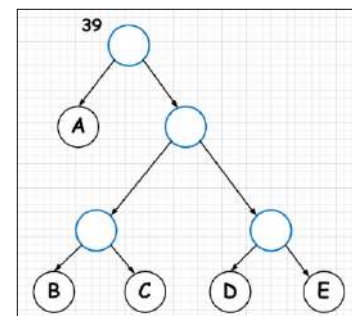
ბ)



გ)



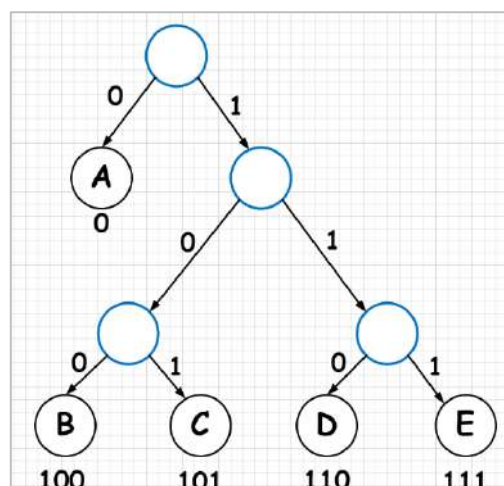
დ)



ე)

ნახ. 12.4. ხის აგება ჰაფმანის ალგორითმის მიხედვით: ა) პირველი ეტაპი; ბ) მეორე ეტაპი; გ) მესამე ეტაპი; დ) მეოთხე ეტაპი; ე) მეხუთე ეტაპი.

გზა ხის ფესვიდან ნებისმიერ ბოლო კვანძამდე შეინახავს ოპტიმალურ პრეფიქსის კოდს (ასევე ცნობილია როგორც ჰაფმანის კოდირება), რომელიც შეესაბამება ამ ბოლო კვანძთან ასოცირებულ სიმბოლოს (ნახ. 12.5.):



ნახ. 12.5. ჰაფმანის ალგორითმის გამოყენებით აგებული ხე

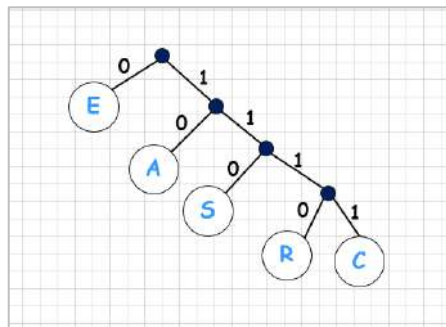
მაგალითი 12.4. ჰაფმანის კოდირებით დაშიფრული ტექსტის გაშიფვრა.

მცირედი განმარტება: სიმარტივისათვის გამოვტოვოთ სიხშირეები და ძალიან მოკლედ ვისაუბროთ ჰაფმანის კოდირებით დაშიფრული ტექსტის გაშიფვრაზე შესაბამისი ხის საშუალებით. მარტივი ASCII კოდი ინგლისური ანბანის ყოველ ასო-ნიშანს წარმოადგენს 7 ბინარული ციფრის საშუალებით.

ჰაფმანის კოდირებაც ბინარულ ციფრებს იყენებს, მაგრამ მათ სიგრძე ცვლადია, სხვადასხვა ასო-ნიშანს სხვადასხვა სიგრძის სტრიქონი შეესაბამება. თითოეული სიმბოლოს შესაბამისი ბინარული სტრიქონი წარმოადგენს გზას ფესვიდან იმ ფოთლამდე, რომელიც შეესაბამება ამ ასო-ნიშანს.

ხის გზაში 0 გვიჩვენებს, რომ უნდა წავიდეთ მარცხნივ, ხოლო 1-ანი მარჯვნივ. გზავნილის გასაშიფრად ჩვენ მივყევით ფესვიდან გზას ციფრების მიხედვით. როდესაც ასო-ნიშანს მივადწევთ, შესაბამის ბინარულ სტრიქონს ჩავანაცვლებთ ამ ასო-ნიშნით, შემდეგ ისევ ფესვიდან დავიწყებთ მოძრაობას დარჩენილი ციფრების მიხედვით.

მაგალითად, ნახატ 12.6-ზე მოცემული ხის გამოყენებით გავშიფროთ შეტყობინება: 0101100



ნახ. 12.6. ხის გამოყენებით 0101100 შეტყობინების გაშიფვრა

ფესვიდან, პირველივე ციფრი მიგვიყვანს *E* სიმბოლომდე. ე.ი. გვაქვს *E*101100

გავყევთ დარჩენილ ციფრებს. 10 შეიცვლება *A*-თი და გვექნება *EA*1100 და ასე შემდეგ, მივიღებთ სიტყვას: **EASE**

ჰაფმანის კოდების არსებითი ნაკლი არის მცირე შეცდომისას უსაზღვროდ მძიმე შედეგები. საკმარისია ერთი მნიშვნელობა გადაიგზავნოს შეცდომით, რომ მთელი გზავნილი აზრს დაკარგავს. ჰაფმანის ალგორითმი უნივერსალურია, მისი გამოყენება შესაძლებელია ნებისმიერი ტიპის მონაცემების შეკუმშვისთვის, მაგრამ ის ნაკლებად ეფექტურია მცირე ზომის ფაილებისთვის (სიტყვათა ლექსიკონის შენახვის აუცილებლობის გამო).

ამჟამად ეს მეთოდი თითქმის არ გამოიყენება სუფთა სახით, როგორც წესი, იგი გამოიყენება როგორც შეკუმშვის ერთ-ერთი ეტაპი უფრო რთულ სქემებში. ეს არის ერთადერთი ალგორითმი, რომელიც არ ზრდის საწყისი მონაცემების ზომას ყველაზე ცუდ შემთხვევაშიც კი (თუ არ ჩავთვლით გადაკოდირების ცხრილის შენახვას ფაილთან ერთად).

ფაქსიმილური აპარატი. ფაქსი (fax) - სატელეკომუნიკაციო ტექნოლოგია, რომლის მეშვეობით გადასცემენ გამოსახულებას ელექტრული სიგნალებით. წარმოადგენს ტელეფონისა და ტელექსის სინთეზს. იგი განიხილება როგორც ასლის გადაღების სისტემა, რომელიც ადრესატს აძლევს დოკუმენტის ასლს.

ფაქსიმილური კავშირგაბმულობა ფაქსიმილური ანუ ფოტოსატელეგრაფო კავშირგაბმულობა ეს არის გრაფიკული ინფორმაციის ელექტრული გადაცემის სახეობა. გადაცემა ხდება ფაქსიმილური აპარატების, ტელეფაქსების, საშუალებით, გადაცემა ძირითადად ხდება კავშირგაბმულობის სატელეფონო არხების საშუალებით. პირველი ტელეფაქსი დაპატენტებული იქნა შოტლანდიელი გამომგონებლის ალექსანდრე ბეინის მიერ 1843 წელს. მისი „ჩამწერი ტელეგრაფი“ მოქმედებდა კავშირგაბმულობის სატელეგრაფო არხებზე და შეეძლო შავთეთრი გამოსახულებების გადაცემა ნახევარტონების გადაცემის გარეშე. 1855 წელს ჯოვანი კასელიმ გამოიგონა აპარატი პანტელეგრაფი (Pantelegraph), რომლის საშუალებითაც შესაძლებელი იყო დოკუმენტების გადაცემა კავშირის არხებში, რომლებიც ერთმანეთს აკავშირებდა პარიზსა და ლიონს. შემდგომში ამ ქალაქებს მიუერთდა კიდევ ბევრი ქალაქი (იხ. ნახ. 12.7.).

XX საუკუნის 30-იან წლებში ალექსანდრე ბეილისა და ჯოვანი კასელის პრინციპებზე აგებული კავშირის არხების საფუძველზე მოქმედებდა კავშირგაბმულობის საშუალებები საგამომცემლო ოფისებს შორის (გაზეთების ახალი ნომრების გაცვლისათვის), სახელმწიფო სამსახურებს შორის (სასწრაფო დოკუმენტების გაცვლისათვის), მართლწესრიგის დაცვის სამსახურებს შორის (სურათებისა და სხვა გრაფიკული მასალების გადაცემისათვის).



ნახ. 12.7. Samsung-ს ფირმის ფაქსის მანქანა

დოკუმენტების გადაცემისათვის გამოიყენებოდა ანალოგური ტექნოლოგიები, რომელთა საშუალებითაც შეუძლებელი იყო სათანადო ხარისხის უზრუნველყოფა და მხოლოდ XX საუკუნის 80-იან წლებში, ციფრული ტექნოლოგიების დანერგვის შედეგად გახდა

შესაძლებელი სატელეფონო კავშირის არხების გამოყენებით მაღალი ხარისხის მიღება არა მარტო ტექსტური დოკუმენტების, არამედ გრაფიკული გამოსახულებების მიღებისთვისაც.

ფაქსიმილური აპარატი განკუთვნილია შავ-თეთრი გრაფიკული გამოსახულებების გადასაცემად ელექტროკავშირის ანალოგური არხებით. ფაქსის აპარატი ვერ გადასცემს ნახევარტონულ სურათებს (მაგალითად, ფოტოებს), ამიტომ ფაქსის კავშირი ძირითადად გამოიყენება მხოლოდ ოფიციალური დაბეჭდილი დოკუმენტების და ხელნაწერი ტექსტის გადასაცემად.

ფაქსიმილე - ხელმოწერის ნიმუშია. სწორედ აქედან წარმოსგება ტერმინი „ფაქსი“. ფაქსიმილური შეტყობინება შავ-თეთრი გამოსახულებაა, რომელიც გადაიცემა ფაქსის აპარატით სატელეფონო კავშირის საშუალებით. ფაქსიმილური დოკუმენტები სულ უფრო დიდ როლს ასრულებენ საქმიან მიმოწერაში. ყოველ გვერდზე მანქანით ნაბეჭდს ვიწრო ზოლად უნდა ჰქონდეს ადგილობრივი საფოსტო ინფორმაცია, ანუ კომპანიის ნომერი, ფაქსის ნომერი და ა.შ. შეტყობინების გადაცემა ხდება ტელეფონის ხაზებით. ქალაქის ფარგლებში ფაქსის გადაცემა ხდება ქვეყნის ან ქალაქის კოდის გარეშე.

ფაქსიმილური კავშირი (facsimile communication) არის უძრავი გამოსახულებისა და ტექსტის დისტანციური გადაცემის პროცესი; მისი ძირითადი ფუნქციაა დოკუმენტების გადაცემა გამგზავნი მხარის ქაღალდის ფურცლებიდან მიმღების ქაღალდის ფურცლებზე; ასეთი დოკუმენტები შეიძლება იყოს ტექსტები, ნახატები, დიაგრამები, ფოტოები და ა.შ. არსებითად, ინფორმაციის გადაცემის ფაქსის მეთოდი მოიცავს დოკუმენტების დისტანციურ კოპირებას.

ფაქსიმილურ კავშირს ადრე უწოდებდნენ ფოტოტელეგრაფიულ კავშირს, მაგრამ საერთაშორისო კომერციული კავშირის რეკომენდაციების თანახმად, ტერმინი „ფოტოტელეგრაფიული კავშირი“ უნდა გამოყენებულ იქნას მხოლოდ ნახევარტონალური სურათების გადაცემის სისტემებისთვის; უფრო ზოგადი ტერმინი „ფაქსიმილური კავშირი“ განეკუთვნება როგორც ნახევარტონალურ, ისე დოკუმენტების გადაცემის სისტემებს.

ფაქსიმილური კავშირის საფუძველს წარმოადგენს ელექტრული სიგნალების დროითი თანმიმდევრობის გადაცემის მეთოდი, რომელიც ახასიათებს გადაცემული დოკუმენტის ცალკეული ელემენტების სიკაშკაშეს. გადაცემული სურათის ელემენტებად დაშლას ეწოდება განლაგება, ხოლო ამ ელემენტების გადახედვას და წაკითხვას სკანირება.

ფაქსიმილური კავშირის მნიშვნელოვანი უპირატესობაა გადაცემის სრული ავტომატიზაცია, მათ შორის ინფორმაციის წაკითხვა ქაღალდის წყაროს დოკუმენტიდან და ინფორმაციის რეგისტრაცია ქაღალდის მიმღების დოკუმენტზე. ფაქსიმილური კავშირის ორგანიზებისთვის გამოიყენება ფაქსიმილური აპარატები (ტელეფაქსები) და საკომუნიკაციო

არხები: ყველაზე ხშირად სატელეფონო არხები, უფრო იშვიათად ციფრული არხები ინტეგრალური სერვისით (ISDN) და საკომუნიკაციო რადიო არხები.

ფაქსის სტანდარტები და რეჟიმები. ფაქსიმილურ კავშირში გამოიყენება მონაცემთა გადაცემის სხვადასხვა სტანდარტები და გარჩევადობის რეჟიმები (სრულად მხარდაჭერილი მხოლოდ ყველაზე სრულყოფილი ტელეფაქსებით).

ტელეფონის საკომუნიკაციო არხების მეშვეობით ფაქსიმილური ინფორმაციის გადაცემის სიჩქარე მერყეობს 4800-28800 ბიტი/წამში; ციფრული არხების გამოყენებით შესაძლებელია ინფორმაციის უფრო მაღალი ხარისხით შეკუმშვა და გადაცემის სიჩქარე 64000 ბიტ/წამამდე აღწევს.

ფაქსის აპარატებს შეუძლიათ ავტომატურად დააყენონ მონაცემთა სიჩქარე, თუ მიმღებ ფაქსს ან საკომუნიკაციო არხს აქვს მაღალი დონის ჩარევა. ამ შემთხვევებში, გადაცემის თავდაპირველად დაყენებული, როგორც წესი, მაქსიმალური შესაძლო სიჩქარე მცირდება მანამ, სანამ არ მიიღწევა შეტყობინებების საიმედო მიღება, რაც დადასტურებულია მიმღები ტელეფაქსით (გადაცემის სესიის დასაწყისში გადამცემი ტელეფაქსი აგზავნის სპეციალურ სიგნალს; მიმღები აპარატი, ამ სიგნალის ამოცნობის შემდეგ, აგზავნის მიღების დამადასტურებელ შეტყობინებას).

მაგალითად, A4 ტექსტური დოკუმენტის გადაცემის დრო, 9600 ბიტ/წამი გადაცემის სიჩქარით, არის დაახლოებით 20 წმ; თუ საკომუნიკაციო არხის ხარისხი დაბალია, ტელეფაქსი შეამცირებს გადაცემის სიჩქარეს 4800 ბიტ/წამამდე, დოკუმენტის გადაცემის დრო გაორმაგდება, ხოლო 2400 ბიტ/წამი სიჩქარით გადაცემისას გაიზრდება 4-ჯერ ანუ დოკუმენტის გადაცემას დასჭირდება უფრო მეტი დრო, დაახლოებით 1 წუთი.

გარჩევადობის (რეზოლუციის) დამუშავების რეჟიმები, რომლებიც გამოიყენება ფაქსის აპარატებში:

- სტანდარტული - ნორმალური, გარჩევადობა 100 * 200 dpi;
- Fine (მაღალი) - მაღალი ხარისხის (მაღალი), გარჩევადობა 200 * 200 dpi;
- Superfine (superhigh) - მაღალი ხარისხის (ულტრა მაღალი), გარჩევადობა 400 * 200 dpi;
- Halftone (Photo) - ნახევარტონალური (ფოტო რეჟიმი), ნაცრისფერი ელფერის 64 გრადაციამდე.

ფაქსის დოკუმენტის სრული სტრიქონი, როცა გვაქვს 210 მმ სიგანის მქონე ფურცელი, შედგება 1600 ელემენტისგან ანუ ფაქსის გარჩევადობა ჰორიზონტალური სტრიქონის გასწვრივ თითქმის ყოველთვის არის 200 ელემენტი/წერტილი 1 ინჩზე ან 200 dpi (dot per inch - წერტილები ინჩზე; 1 ინჩი = 25.4 მმ). 1 ელემენტი იკავებს დაახლოებით 1/8 მმ-ს.

მაღალი ხარისხის რეჟიმს აქვს გარჩევადობა 300 – 400 dpi, რაც დამოკიდებულია გამოყენებული ფაქსის აპარატზე, ნახევარტონიანი რეჟიმი უზრუნველყოფს ნაცრისფერი ფერის ტონების გადაცემას და გამოიყენება სურათის ან ნახევარტონიანი ნახატის გადაცემის აუცილებლობის შემთხვევაში. ნაცრისფერი გრადია - მნიშვნელოვანი პარამეტრია, რომელიც განსაზღვრავს ნახევარტონალური გამოსახვის შესაძლებლობას. ფოტოსურათები, ნახატები, რეპროდუქციები, ფერადი დოკუმენტები შეიძლება გადაეცეს შავ-თეთრ გამოსახულებაში და რაც უფრო მეტია ნაცრისფერი გრადიაციის რაოდენობა (ნახევარტონები, ელფერები), მით უფრო მაღალი იქნება გადაცემული გამოსახულების ხარისხი.

რაც უფრო მაღალია გარჩევადობის (რეზოლუციის) რეჟიმი, მით მეტია წერტილების რაოდენობა, რომლებიც იკითხება დოკუმენტიდან და შესაბამისად უფრო მეტი დროა საჭირო მთელი დოკუმენტის წასაკითხად. მონაცემთა გადაცემა *fine* რეჟიმში დაახლოებით ორჯერ აორმაგებს გადაცემის დროს სტანდარტულ რეჟიმთან შედარებით, ხოლო *superfine* რეჟიმი ამ დროს 4-ჯერ იზრდება; ნახევარტონალურ რეჟიმში გადაცემის დრო მინიმუმ 8-ჯერ მეტია ვიდრე სტანდარტულ რეჟიმში.

ზოგადად, დოკუმენტის ერთი ფურცლის გადაცემაზე დახარჯული დრო დამოკიდებულია ამ ფურცლის ზომაზე, მასზე, გამოსახულების ბუნებაზე, გადაცემის სიჩქარეზე და რეზოლუციის რეჟიმზე. ფაქსით კომუნიკაცია შეიძლება გამოყენებულ იქნას კომპიუტერში გადაცემული ინფორმაციის ავტომატურად შესატანად, თუ ეს უკანასკნელი აღჭურვილია ფაქსის მოდემით.

კომპიუტერული ფაქსის სისტემები. კომპიუტერი სულ უფრო და უფრო მეტად იძენენ მძლავრი კომპიუტერიდან ძლიერ საკომუნიკაციო ინსტრუმენტად გადაქცევის შესაძლებლობებს. მართლაც, მრავალფეროვანი ინფორმაციისა და კომპიუტერული ქსელების მეშვეობით შეგიძლიათ გაგზავნოთ და მიიღოთ შეტყობინებები მსოფლიოს ყველაზე შორეულ ადგილებში, გაცვალოთ მონაცემები და პროგრამები ასობით და ათასობით აბონენტთან და მიიღოთ ნებისმიერი საცნობარო ინფორმაცია ოპერატიული სერვისის სისტემებიდან.

როგორც უკვე აღვნიშნეთ, კომპიუტერი შეიძლება დაუკავშირდეს აბონენტთა სატელეფონო ქსელს და მოიპოვოს წვდომა ამ ქსელის სხვა აბონენტებთან, ელექტრონულ ფოსტაზე და ტელეფაქსებზე, რომლებიც მუშაობენ ამ ქსელთან (მსგავსი სერვისის ქსელები უკვე არსებობს: *Rosnet*, *REX 400* და ა. შ.).

ფაქსის მოდემის მექანე კომპიუტერი მუშაობს ბევრად უფრო საიმედოდ (არ „ღეჭავს“ ქაღალდს) და უფრო სტაბილურია ვიდრე ტელეფაქსი და გთავაზობთ ბევრ დამატებით

სერვისს: მნიშვნელოვნად უფრო მოსახერხებელი და ეფექტური ავტომატიზაცია ფაქსის ტექსტების მომზადების კომპიუტერული ხელსაწყოების მთელი არსენალის გამოყენებით. ელექტრონულ ფოსტასთან, ტელექსთან და კომპიუტერულ მონაცემთა ბაზასთან ინტეგრაცია, დიდი ელექტრონული საცნობარო წიგნის არსებობა, რომელიც შეიცავს მრავალფეროვან სასარგებლო ინფორმაციას, თანამშრომლებისა და გარე აბონენტებისთვის წვდომის უფლებების დელიმიტაცია ფაქსზე, კორესპონდენციის გავლის კონტროლი, დეტალური სტატისტიკა, ფაქსით მუშაობა და ა.შ.

უკვე მოხმარებაში გამოდის კომპიუტერების ისეთი სახის კლავიატურები, რომლებითაც შესაძლებელია პირდაპირ აკრიფოთ აბონენტის ტელეფონის ნომერი (კლავიატურა *Compu Phone 2000*), ასევე უკვე გამოჩნდა კომპიუტერები, რომლებიც აღჭურვილია ვიდეოკამერით და მიკროფონით, რაც საშუალებას იძლევა არა მხოლოდ ფაქსების გაცვლა პარტნიორთან, არამედ მისი დანახვა და მასთან საუბარი.

რომელი P და NP და NP -სრული კლასები. NP -სრული ამოცანების კრებული.

საერთო სტრატეგიები, გადაუჭრელი ამოცანები.

P , NP და NP -სრული კლასები არის კომპიუტერული მეცნიერების ისეთი ცნებები, რომლებიც ასახავენ ალგორითმების გამოთვლით სირთულეს. თავდაპირველად განვმარტოთ თუ რას ნიშნავს P და NP კლასები.

P კლასი (Polynomial Time). P არის ამოცანების კლასი, რომელიც შეიძლება გადაიჭრას/გადაწყდეს პოლინომიალურ დროში (დამოკიდებულია შესატანი მონაცემების ზომაზე).

P კლასში შემავალი პრობლემების ან ამოცანების გადაჭრა/გადაწყვეტა ხდება ისეთი სახის ალგორითმებით, რომლებიც მუშაობენ პოლინომიალურ დროში ანუ თუ შესატანი მონაცემების ზომა არის n , მაშინ P კლასის ამოცანის ამოხსნის დრო იქნება $O(n^k)$, სადაც k არის მუდმივა (ფიქსირებული დადებითი რიცხვი).

პრაქტიკულად, P კლასის პრობლემები ითვლება „მარტივებად“ ან „გადაჭრად“ სტანდარტული კომპიუტერული ალგორითმებით.

P კლასის ამოცანებს განეკუთვნება:

1. **დახარისხების ალგორითმები (Sorting Algorithms).** ალგორითმები, რომლებიც ახარისხებენ მონაცემებს, მაგალითად, შერწყმის (Merge Sort) ან სწრაფი დახარისხების (Quick Sort) ალგორითმები. მათი დროის სირთულე არის $O(n * \log(n))$.

2. **გრაფში უმოკლესი გზის პოვნის ალგორითმი (Shortest Path in a Graph).** დიექსტრას ალგორითმი, რომელიც დადებითი წონის მქონე გრაფებში ეძებს ყველაზე მოკლე გზას. დიექსტრას ალგორითმის დროის სირთულე არის $O(n^2)$ ან $O(E * \log(V))$, სადაც E არის გრაფის წიბოების რაოდენობა, ხოლო V კი არის გრაფის წვეროების რაოდენობა.

3. **ორი მატრიცის გამრავლების ალგორითმი (Matrix Multiplication).** ორი მატრიცის გამრავლების ალგორითმის დროითი სირთულე არის $O(n^3)$.

4. **ევკლიდეს ალგორითმი (Finding the Greatest Common Divisor (GCD)).** ევკლიდეს ალგორითმი - ორ რიცხვს შორის უდიდესი საერთო გამყოფის პოვნა. დროითი სირთულე არის $O(\log(n))$.

5. **რიცხვის შემოწმება მარტივობაზე (Prime Number Checking).** რომელიმე n რიცხვის მარტივობაზე შესამოწმებლად გამოიყენება მილერ-რაბინის ტესტი.

პოლინომიალური დრო (Polynomial Time) აღწერს ამოცანის ამოხსნის დროით სირთულეს, რომელიც დამოკიდებულია შესატანი მონაცემების ზომაზე n და იზრდება პოლინომის სახით. ალგორითმის შესრულების დრო ითვლება პოლინომიალურ დროდ,

თუ მისი სირთულე შეიძლება გამოიხატოს $O(n^k)$ ფორმულით, სადაც n არის შესატანი მონაცემების ზომა, ხოლო k არის დადებითი მუდმივი რიცხვი.

პოლინომიალური დრო იზრდება შესატან მონაცემების ზომასთან შედარებით „მსუბუქად“ და დროის ზრდა არის მართვადი. მაგ., ამოცანის გადასწყვეტად საჭიროა 10 ელემენტის შეტანა ანუ შესატანი მონაცემების ზომა არის $n = 10$, მაშინ:

1. თუ ალგორითმის სირთულე არის $O(n^2)$, იგი შესრულდება დაახლოებით $10^2 = 100$ ერთეულ დროში.

2. თუ შესატანი მონაცემების რაოდენობა გაიზრდება და გახდება $n = 100$, მაშინ ამოცანის შესრულების დრო გაიზრდება და გახდება $100^2 = 10000$ ერთეული.

პოლინომიალური დროის ალგორითმები პრაქტიკულად გამოყენებადია, რადგან ისინი ეფექტურად დიდ მონაცემებთან მუშაობენ.

NP კლასი (Nondeterministic Polynomial Time). NP კლასის ამოცანები არის ისეთი ამოცანები, რომელთა ამოხსნის სისწორის შემოწმება/გადამოწმება შესაძლებელია პოლინომიალურ დროში. NP კლასის ამოცანების პასუხები შეიძლება გადამოწმდეს პოლინომიალურ დროში ანუ თუ ვინმე გადაწყვეტს ამოცანას, შეგიძლიათ სწრაფად (პოლინომიალურ დროში) გადაამოწმოთ, არის თუ არა ამოცანა სწორად ამოხსნილი.

NP კლასის ალტერნატიული განმარტება შეიძლება ასეთი სახითაც დავწეროთ: NP არის ამოცანების კლასი, რომელიც შეიძლება გადაიჭრას პოლინომიურ დროში არადეტერმინირებული ტურინგის მანქანის გამოყენებით. NP კლასის ამოცანებს განეკუთვნება:

1. **მოგზაურის ამოცანა.** მოცემულია ქალაქების სია და მათ შორის მანძილი. საჭიროა მოგებნით ყველაზე მოკლე მარშრუტი, რომელიც ყველა ქალაქს ერთჯერ მოინახულებს და უკან დაბრუნდება საწყის წერტილში;

2. **ზურგჩანთის პრობლემა.** მოცემულია ნივთების სია, რომელთაგან თითოეული აქვს წონა და ფასი. საჭიროა განვსაზღვროთ, თუ რომელი ნივთები გადავიტანოთ ზურგჩანთაში ისე, რომ ფასების ჯამი მაქსიმალური იყოს, ხოლო ზურგჩანთის ტევადობა არ იყოს გადაჭარბებული;

3. **ჰამილტონის გრაფი.** გრაფში არსებობს თუ არა ისეთი გზა, რომელიც ყველა წვეროს ზუსტად ერთხელ გაივლის და სხ.

4. **სუდოკუს ამოხსნა.** თუ ვინმე მოგცემთ ამოხსნილ სუდოკუს, შეგიძლიათ სწრაფად შეამოწმოთ მისი სისწორე.

ეს ამოცანები შედიან NP კლასში, რადგან მათი გადაჭრის შედეგის შემოწმება შეიძლება პოლინომიური დროის განმავლობაში, თუმცა მათი ოპტიმალური გადაჭრის პროცესი შეიძლება არ იყოს პოლინომიური. ეს არ ნიშნავს, რომ თავად პრობლემის ამოხსნა

ხდება პოლინომურ დროში, არამედ თუ გაქვთ ამოცანის ამოხსნის პასუხი, შეგვიძლიათ სწრაფად დავადგინოთ, არის თუ არა ეს პასუხი სწორი.

NP-სრული (NP-Complete). NP-სრული პრობლემები არის NP კლასის ყველაზე რთული პრობლემები.

თუ რომელიმე NP-სრული პრობლემა გადაიჭრება პოლინომიალურ დროში, მაშინ ყველა NP პრობლემა იქნება გადაჭრადი პოლინომიალურ დროში (ანუ $P = NP$).

NP-სრული პრობლემის ძირითადი თვისებებია:

1. **ამოცანა ეკუთვნის NP კლასს.** ამოცანა NP-ს (Non-deterministic Polynomial time) კლასისაა, თუ მისი გადაწყვეტილების ვალიდაცია შესაძლებელია პოლინომიალური დროში ანუ თუ არსებობს ალგორითმი, რომელიც ამოცანის გადაწყვეტის შემოწმებას პოლინომიალურ დროში შეძლებს.

2. **NP-სირთულე.** NP-სრული პრობლემა ასევე NP-რთულია, რაც ნიშნავს, რომ ნებისმიერი სხვა NP-პრობლემა შეიძლება გარდაიქმნას ამ ამოცანად პოლინომიალურ დროში. ეს არის მათი მთავარი თვისება და ამით გამოირჩევა სხვა რთული პრობლემებისგან.

3. **გარდაქმნა (Reduction).** თუ ერთი პრობლემა NP-სრულია, ნებისმიერი სხვა NP-სრული პრობლემა შეიძლება გარდაიქმნას ამ პრობლემად პოლინომიალურ დროში. ამის გამო, თუ რომელიმე NP-სრული პრობლემა პოლინომიალურად გადაიჭრება, მაშინ ყველა NP-სრული ამოცანა გადაიჭრება.

4. **გადაუჭრელობა პოლინომიალურ დროში (ვარაუდი).** ჯერჯერობით არც ერთი NP-სრული პრობლემა არ არის გადაჭრილი პოლინომიალური დროის ალგორითმით. შესაბამისად, მეცნიერები ფიქრობენ, რომ შესაძლოა, $P \neq NP$, რაც ნიშნავს, რომ NP-სრული პრობლემების გადაჭრა არ შეიძლება პოლინომიალურ დროში.

5. **პრაქტიკული მნიშვნელობა.** ბევრი რეალური პრობლემა არის NP-სრული. მათი გადაჭრის სირთულე მნიშვნელოვან გავლენას ახდენს სხვადასხვა სფეროზე, როგორიცაა ოპტიმიზაცია, ქსელების ანალიზი და სხვა. ეს თვისებები NP-სრულ პრობლემებს განსაკუთრებულ მნიშვნელობას ანიჭებს კომპიუტერულ თეორიაში, რადგან ისინი ერთ-ერთ ყველაზე რთულ ამოცანებს წარმოადგენენ.

მნიშვნელოვანი საკითხი NP ამოცანებთან დაკავშირებით: თუ ვიპოვით ისეთ ამოცანას, რომელიც NP კლასს განეკუთვნება და მისი ამოხსნა შესაძლებელია პოლინომიალურ დროში, ეს ნიშნავს, რომ ყველა NP ამოცანა შეიძლება ამოიხსნას პოლინომიალურ დროში.

კომპიუტერულ მეცნიერებაში P და NP კლასების შედარება არის ერთ-ერთი ფუნდამენტური თემა, რომელიც ეხება გამოთვლითი სირთულის ანალიზს. P და NP კლასებს შორის ძირითადი განსხვავებები და მსგავსებები მოცემულია ცხრილში 13.1.

ცხრილი 13.1. P და NP კლასების შედარებითი ანალიზი

თვისება	<i>P</i> კლასი	<i>NP</i> კლასი
გადაჭრა	შესაძლებელია პოლინომიალურ დროში	უცნობია, შესაძლებელია თუ არა პოლინომიალურ დროში გადაჭრა
გადამოწმება	პასუხის გადამოწმება მარტივია და სწრაფი	პასუხის გადამოწმება შესაძლებელია პოლინომიალურ დროში
გამოთვლითი სირთულე	შედარებით მარტივი პრობლემები	შეიძლება შეიცავდეს ისეთ რთულ პრობლემებს, რომელთა გადაჭრა საკმაოდ რთულია
მნიშვნელოვანი კითხვა	ყველა <i>P</i> კლასის პრობლემა <i>NP</i> -კლასთანაც გვაქვს	უცნობია, არის თუ არა ყველა <i>NP</i> ამოცანა ასევე <i>P</i> ამოცანა ანუ $P = NP$?
პრაქტიკულობა	პრაქტიკულად ამოხსნადია	შესაძლებელია, რომ ზოგი პრობლემა პრაქტიკულად არ იყოს ამოხსნადი

$P \subseteq NP$. ყველა პრობლემა, რომელიც ეკუთვნის **P** კლასს, ასევე შედის **NP** კლასში, რადგან თუ პრობლემა შეიძლება გადაჭრილ იქნას პოლინომიალურ დროში, მისი პასუხი ავტომატურად გადამოწმდება პოლინომიალურ დროში. ამიტომ, **P** კლასის ამოცანები ასევე ეკუთვნის **NP** -ს.

$P = NP$? ყველაზე მნიშვნელოვანი კითხვაა კომპიუტერულ მეცნიერებაში: არიან თუ არა P და NP ერთნაირი კლასები ანუ არის თუ არა ყველა პრობლემა, რომლის პასუხის გადამოწმება შესაძლებელია პოლინომიალურ დროში, ასევე გადაჭრადი პოლინომიალურ დროში? ამ კითხვაზე პასუხი ჯერ არ არის ნაპოვნი და ის ერთ-ერთ გადაუჭრელ პრობლემად რჩება კომპიუტერულ მეცნიერებაში.

დასკვნა: P კლასი აერთიანებს პრობლემებს, რომლებიც შეიძლება გადაჭრილ იქნას ეფექტურად. NP კლასი აერთიანებს პრობლემების უფრო ფართო სპექტრს, სადაც პრობლემები შეიძლება რთული იყოს გადასაჭრელად, მაგრამ მარტივი პრაქტიკული თვალსაზრისით. P და NP კლასების გრაფიკული გამოსახვა შეიძლება წარმოვადგინოთ ვენის დიაგრამით, რომელიც აჩვენებს ამ კლასებს შორის დამოკიდებულებას (ნახ. 13.1.).



ნახ. 13.1. ვენის დიაგრამა - P და NP კლასების ურთიერთდამოკიდებულება

ნახ. 13.1.-დან ჩანს, რომ P წრე მთლიანად მოთავსებულია NP წრეში. არსებობს დიდი NP წრე, რომელშიც შეიძლება იყოს ისეთი პრობლემები, რომლებიც არ გვაქვს P -კლასში.

თუ $P \neq NP$, ეს ნიშნავს, რომ არსებობს პრობლემები, რომლებიც ეკუთვნის NP -კლასს, მაგრამ არა P -კლასს.

ნახ. 13.1-დან ჩანს, რომ P კლასი არის მცირე წრეში, დიდი წრის შიგნით და მოიცავს ყველა იმ პრობლემას, რომელიც შეიძლება გადაჭრილი იყოს პოლინომიალურ დროში. NP -კლასი მოიცავს უფრო დიდ წრეს და ყველა იმ პრობლემას, რომელთა გადამოწმება შეიძლება პოლინომიალურ დროში. P კლასის პრობლემები ასევე NP -კლასის პრობლემების ნაწილია ანუ P არის NP -კლასის ქვეკლასი ($P \subseteq NP$).

ტიურინგის მანქანა. ალან მეთისონ ტიურინგი (დ. 23 ივნისი, 1912 - გ. 7 ივნისი, 1954) იყო ინგლისელი მათემატიკოსი, კრიპტოანალიტიკოსი და კომპიუტერული მეცნიერების დარგის მოღვაწე. ხელოვნური ინტელექტის მამად წოდებული, ალან ტიურინგი, ბავშვობიდან ამჟღავნებდა მათემატიკურ ნიჭს. მან უმნიშვნელოვანესი წვლილი შეიტანა კომპიუტერული მეცნიერებების შექმნასა და ჩამოყალიბებაში, ალგორითმების ძირითადი კონცეფციების ფორმალიზაციასა და თანამედროვე კომპიუტერის შექმნაში (ნახ. 13.2.).



ნახ. 13.2. ალან მეთისონ ტიურინგი

1939 წელს, ალან ტიურინგი, ბლეტჩლის პარკში მდებარე ანალიტიკოსთა ცენტრში შიფრატორად მუშაობდა. მან მთავრობისგან მიიღო დავალება გაეშიფრა გერმანული სამხედრო შიფრი ენიგმა, ამისთვის ალანმა შექმნა მანქანა/კომპიუტერი, რომელსაც გააჩნდა ხელოვნური ინტელექტი. მანქანამ შეძლო ენიგმას გატეხვა, რამაც საბოლოოდ გადაწყვიტა ომის შედეგები. ისტორიკოსთა შეფასებით ტიურინგის ამ მიღწევამ მეორე მსოფლიო ომის ხანგრძლივობა ორი წლით შეამცირა და დაალოებით 14 მილიონი ადამიანის სიცოცხლე გადაარჩინა.

ტიურინგის მანქანა წარმოადგენს თეორიულ გამოთვლით მოდელს, რომელიც ალან ტიურინგმა 1936 წელს წარმოადგინა. ეს მოდელი განსაზღვრავს აბსტრაქტულ მექანიზმს,

რომელსაც შეუძლია ნებისმიერი გამოთვლითი ამოცანის შესრულება, რომელიც შესრულებადია ალგორითმულად. ტიურინგის მანქანა ითვლება ყველა თანამედროვე კომპიუტერის პროტოტიპად და ხშირად გამოიყენება კომპიუტერულ მეცნიერებაში, რათა გამოიკვლიონ გამოთვლითი სირთულებები.

ტიურინგის მანქანის კომპონენტებია:

1. უსასრულო ლენტი - ეს არის მეხსიერების მოწყობილობა, რომელიც იყოფა უჯრედებად. თითოეული უჯრედი ინახავს სიმბოლოს და ლენტს შეუძლია იყოს უსასრულოდ გრძელი.

2. კონტროლის მდგომარეობა - ეს არის ტიურინგის მანქანის „გონება“, რომელიც განსაზღვრავს მიმდინარე მდგომარეობას და იღებს გადაწყვეტილებას, რა ქმედება უნდა შესრულდეს.

3. ჩაწერისა და წაკითხვის თავი - ეს კომპონენტი გადაადგილდება ლენტზე მარცხნივ ან მარჯვნივ და ანიჭებს ან კითხულობს სიმბოლოებს.

4. ქმედებების ნაკრები (დასკვნის ფუნქცია) - ყოველი მოქმედება განსაზღვრავს, თუ როგორ უნდა შეიცვალოს სიმბოლო ლენტზე, რა მიმართულებით უნდა გადაადგილდეს ჩაწერა/წაკითხვის თავი და რა მდგომარეობაში უნდა გადავიდეს ტიურინგის მანქანა.

ტიურინგის მანქანის მუშაობის პროცესი:

1. მანქანა კითხულობს ლენტზე არსებულ სიმბოლოს;

2. დამოკიდებულია რა მიმდინარე მდგომარეობაზე და წაკითხულ სიმბოლოზე, მანქანა ჩაანაცვლებს ამ სიმბოლოს სხვა სიმბოლოთი, გადაადგილებს თავს მარცხნივ ან მარჯვნივ და გადადის ახალ მდგომარეობაში;

3. ეს პროცესი გრძელდება მანამ, სანამ მანქანა არ მიაღწევს „დასრულების“ მდგომარეობას ან ჩათვლის, რომ ამოცანა დასრულებულია.

ტიურინგის მანქანა მნიშვნელოვანი მოდელია, რადგან ის აჩვენებს, რომ ნებისმიერი გამოთვლითი პროცესი შეიძლება აღწერილი იყოს მარტივი ინსტრუქციებით, რაც საფუძველს უდებს ისეთ თეორიას, რომ ამგვარი მანქანები ალგორითმულად შესრულებად ნებისმიერი ამოცანის შესრულებას შეძლებენ.

ტიურინგის მანქანის მუშაობის მაგალითი. განვიხილოთ ტიურინგის მანქანის გამოყენებით ლენტაზე ჩაწერილი ბინარული რიცხვის (მაგალითად, 101) ერთით გაზრდის პროცესი. ესაა მარტივი ბინარული ზრდის ოპერაცია, რომელიც ბიტების დონეზე ხდება.

ამოცანის დასმა:

ლენტზე არსებული ბინარული რიცხვის გაზრდა ერთით.

ლენტზე მოცემულია რიცხვი, მაგალითად: 101

მიზანია:

ამ რიცხვის გაზრდა ერთით, რაც შედეგში მოგვცემს 110

ამოცანის შესრულების თანამიმდევრობის ბიჯები:

1. **ლენტის საწყისი მდგომარეობა:** ლენტზე ჩაწერილი გვაქვს 101, ხოლო ტიურინგის მანქანის „ჩაწერისა და წაკითხვის თავი“ (კონტროლის თავი) მარჯვენა ბოლოდან იწყებს მუშაობას.

ლენტი: 1 0 1

თავი: ^

2. **წესების ნაკრები:**

თუ შეხვდება 1 და ჯერ კიდევ არ ვამატებთ ერთს: გადაანაცვლე $1 \rightarrow 0$ და გადაადგილდი მარცხნივ.

თუ შეხვდება 0: გადაანაცვლე $0 \rightarrow 1$ და ამით დასრულდა რიცხვის ზრდა.

თუ ყველა სიმბოლო უკვე 1-ია (მთელი რიცხვი შეიცვალა 0-ებით): დაამატე მარცხნივ ახალი 1.

ტიურინგის მანქანის მუშაობის დეტალური ნაბიჯები:

ბიჯი 1:

ჩაწერისა და წაკითხვის თავი წაკითხავს მარჯვენა, ყველაზე შორს მყოფ სიმბოლოს, რომელიც არის 1. რადგან ეს სიმბოლო 1-ია, მას ვცვლით 0-ით და გადავადგილდებით მარცხნივ.

ლენტი: 1 0 0

თავი: ^

ბიჯი 2:

ახლა თავი დგას 0-ზე. როცა მანქანა ხედავს 0-ს, ის მას ცვლის 1-ით და პროცესი ამით მთავრდება, რადგან ბინარული რიცხვის ერთით გაზრდა დასრულდა.

ლენტი: 1 1 0

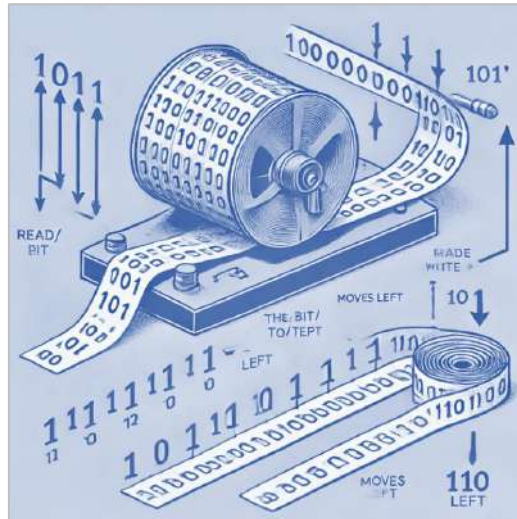
თავი: ^

საბოლოო შედეგი:

ამ პროცესის შემდეგ ლენტზე ჩნდება ახალი ბინარული რიცხვი 110, რაც არის 101-ის ერთით გაზრდილი მნიშვნელობა.

ტიურინგის მანქანამ გაზარდა ლენტაზე არსებული ბინარული რიცხვი ერთით, გადახედა/გადაურბინა ბიტებს მარჯვენიდან მარცხნივ და საჭიროებისამებრ შეცვალა ისინი.

ტიურინგის მანქანის მუშაობის მაგალითის ზოგადი სქემა მოცემულია ნახატ 13.3-ზე.



ნახ. 13.3. ტიურინგის მანქანის მუშაობის სქემა - ლენტზე არსებული ბინარული

მათემატიკაში არსებობს შვიდი ამოუხსნელი ამოცანა, რომლებიც ცნობილია „სასწაულებრივი ამოცანების“ (Millennium Prize Problems) სახელით.

კლემას მათემატიკის ინსტიტუტი (Clay Mathematics Institute) არის კერძო ორგანიზაცია, რომელმაც 2000 წელს ათასწლეულის 7 ამოუხსნადი ამოცანები დაასახელა. მათ ამოხსნაზე დაწესებულია ჯილდო 1 მილიონი დოლარი. ბოლო 17 წლის განმავლობაში მსოფლიოს საუკეთესო მათემატიკოსებმა მხოლოდ ერთი ჰიპოთეზის ამოხსნა შეძლეს.

ეს ამოცანებია:

1. P და NP კლასების ტოლობა;
2. ხოჯის ჰიპოთეზა;
3. პუანკარეს ჰიპოთეზა;
4. რიმანის ჰიპოთეზა;
5. იანგ-მილსის თეორია;
6. ნავიე-სტოქსის განტოლება;
7. ბიორჩ და სვინერთონ-დაიერის ჰიპოთეზა.

ამ დროისთვის მხოლოდ პუანკარეს ჰიპოთეზაა ამოხსნილი, დანარჩენი ამოცანები დღემდე ამოუხსნელია. განვიხილოთ თითოეული მათგანი:

1. P და NP კლასების ტოლობა (1971 წ.).

კვლევის სფერო: ალგორითმების თეორია. თუ რომელიმე კითხვის დადებითი პასუხი შეიძლება სწრაფად (პოლინომიულ დროში) შემოწმდეს, მაშინ მართალია თუ არა, რომ ამ კითხვის პასუხიც შეიძლება სწრაფად მოიძებნოს? გვაქვს ორი კლასი: P და NP კლასები. P არის ამოცანების ერთობლიობა, რომელიც კომპიუტერს შეუძლია გადაჭრას პოლინომიულ დროში, ე.ი. საკმაოდ სწრაფად. NP არის ამოცანების კლასი, რომელთა პასუხების სისწორეც ასევე შეიძლება საკმაოდ სწრაფად შემოწმდეს. თანდათან ვუახლოვდებით

ათასწლეულის ერთ-ერთ პრობლემას: არის თუ არა P და NP კლასები თანაბარი ანუ თანატოლი? მეცნიერთა დიდი ნაწილი თვლის, რომ P და NP კლასები არ არის ტოლი ანუ თანაბარი კლასები, მაგრამ ჯერ ვერავინ შეძლო ამ აზრის დამტკიცება, მაგრამ რა მოხდება, თუ აღმოჩნდება, რომ $P = NP$.

P და NP კლასების თანასწორობის პრობლემა ალგორითმების თეორიის ერთ-ერთი უმნიშვნელოვანესი გადაუჭრელი პრობლემაა.

2. რიმანის ჰიპოთეზა (1859 წ.).

კვლევის სფერო - რიცხვთა თეორია.

1859 წელს ბერნჰარდ რიმანმა ჩამოაყალიბა ჰიპოთეზა, რომელიც ეხება მარტივი რიცხვების განაწილების კანონზომიერების დადგენას. რიმანის ჰიპოთეზა უარყოფს, რომ მარტივი რიცხვების განაწილება შემთხვევითია. ის ამბობს, რომ ასეთი რიცხვების განაწილების აღწერა შესაძლებელია სპეციალური ძეტა ფუნქციით.

ამ დრომდე შემოწმებულია 15 ტრილიონი მარტივი რიცხვი და ყველა მათგანი რიმანის ჰიპოთეზასთან თანხვედრაშია, თუმცა არ არსებობს მტკიცებულება, რომ ყველა მარტივი რიცხვი ამ კანონზომიერებას მიჰყვება.

რიმანის ძეტა-ფუნქცია არის კომპლექსური $z = a + i * b$ ცვლადის ანალიზური ფუნქცია, რომელიც აღინიშნება $\zeta(z)$ და განისაზღვრება მწკრივით:

$$\zeta(z) = \sum_{n=1}^{\infty} \frac{1}{n^z} = \frac{1}{1^z} + \frac{1}{2^z} + \frac{1}{3^z} + \dots$$

მწკრივი შეიძლება ანალიზურად გაგრძელდეს მთელ კომპლექსურ სიბრტყეზე და რეგულარულია z -ის ყველა მნიშვნელობისათვის, გარდა $z = 1$. როცა $a > 1$, მაშინ სამართლიანია ამ ფუნქციის წარმოდგენა ეილერის ნამრავლის სახით:

$$\zeta(z) = \prod_p \left(1 - \frac{1}{p^z}\right)^{-1}$$

სადაც p დებულობს ყველა მარტივი რიცხვის მნიშვნელობას.

$\zeta(z)$ მწკრივისა და ეილერის ნამრავლის იგივეობა წარმოადგენს ძეტა-ფუნქციის ერთ-ერთ ძირითად თვისებას. იგი საშუალებას იძლევა მივიღოთ მრავალრიცხოვანი თანაფარდობები, რომლებიც ძეტა-ფუნქციას აკავშირებენ მნიშვნელოვან თეორიულ-რიცხვით ფუნქციასთან. ამიტომ, ძეტა-ფუნქცია დიდ როლს თამაშობს რიცხვთა თეორიაში.

3. იანგ-მილსის თეორია (1954 წ.).

კვლევის სფერო - ელემენტარული ნაწილაკების ფიზიკა.

ელემენტარული ნაწილაკების ფიზიკა სწავლობს მატერიის უმცირეს ელემენტარულ ნაწილაკებს და მათ შორის არსებულ ურთიერთქმედებებს. იანგ-მილსის თეორია აღწერს ელემენტარული ნაწილაკებისა და მათ შორის ურთიერთქმედების მათემატიკურ მოდელს.

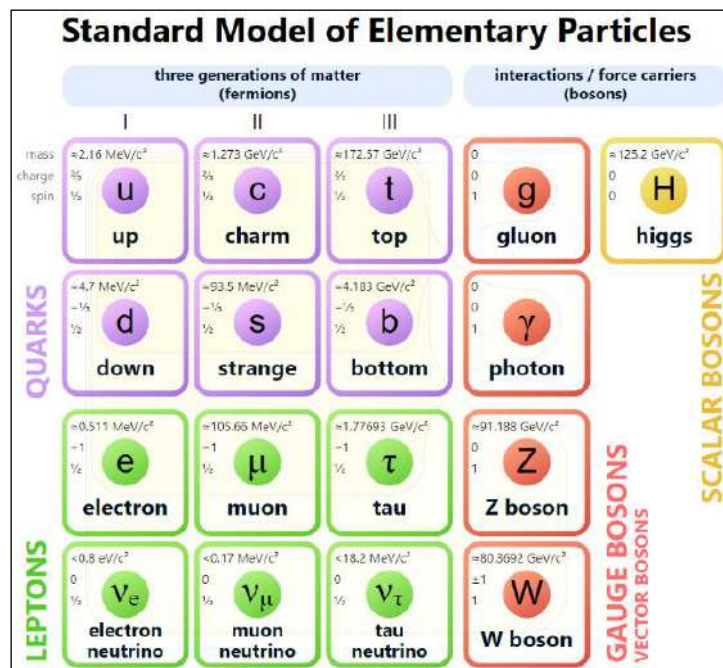
ურთიერთქმედება არის ორი ობიექტის მესამე ნაწილაკით გაცვლის პროცესი. ასეთ ნაწილაკებს ურთიერთქმედების მატარებლებს უწოდებენ.

ვიცით ოთხი ტიპის ურთიერთქმედება, რომელთა არსებობაც უკვე დამტკიცებულია:

1. გრავიტაციული; 2. ელექტრომაგნიტური; 3. ძლიერი; 4. სუსტი.

იანგ-მილსის თეორია ცნობს ბოლო სამ ურთიერთქმედებას: ელექტრომაგნიტურს, ძლიერს და სუსტს. იანგმა და მილსმა მოძებნეს ელექტრომაგნიტური, სუსტი და ძლიერი ურთიერთქმედების თეორიების გაერთიანების გზა. იანგ-მილსის თეორიის დახმარებით, ახალი ნაწილაკების აღმოჩენის პროგნოზირებაც კი მოხერხდა. ნაწილაკების სწორი მასის პროგნოზირება ჯერ არავის გამოსდის.

ჩვენს სამყაროში ყველაფერი შედგება მოლეკულებისგან, მოლეკულები შედგება ატომებისგან, ატომები კი ბირთვისა და ელექტრონებისგან. ატომის ბირთვი ასევე იყოფა პროტონებად და ნეიტრონებად და თუ კვანტური ფიზიკა სწავლობს ელემენტების ურთიერთქმედებას მიკროსკოპულ დონეზე, ელემენტარული ნაწილაკების ფიზიკა კიდევ უფრო შორს მიდის და სწავლობს კვარკებს, ყველაზე უმცირეს ელემენტებს, რომელთა განცალკევებაც შეუძლებელია. სწორედ კვარკებისგან შედგება პროტონები და ნეიტრონები (ნახ. 13.4.).



ნახ. 13.4. ელემენტარული ნაწილაკების სტანდარტული მოდელი ფიზიკაში.

ელემენტარული ნაწილაკების თეორიული კონსტრუქცია, რომელიც აღწერს ყველა ელემენტარული ნაწილაკების ელექტრომაგნიტურ, სუსტ და ძლიერ ურთიერთქმედებას.

თანამედროვე ფორმულირება დასრულდა 2000-იან წლებში კვარკების არსებობის ექსპერიმენტული დადასტურების შემდეგ. t-კვარკის (1995 წ.), b-კვარკის (1977 წ.) და ტაუ-ნეიტრინოს (2000 წ.) აღმოჩენამ დაადასტურა სტანდარტული მოდელის სისწორე.

იანგ-მილსის თეორიის საფუძველზე შეიქმნა ელემენტარული ნაწილაკების ფიზიკის სტანდარტული მოდელი - ჩვენი სამყაროს გარკვეული „კოდი“, რომელიც შედგება კვარკებისგან. კვარკი არის ფუნდამენტური ნაწილაკი, რომელისგანაც შექმნილია სამყარო, ლეფტონებთან და მათ შორის ურთიერთმქმედ ნაწილაკებთან ერთად.

ნახ. 13.4.-დან ჩანს, რომ ელემენტარული ნაწილაკების ცხრილი იყოფა კვარკებად, ლეფტონებად და ურთიერთქმედების გადამტან ბოზონებად.

4. ნავიე-სტოქსის განტოლება (1822 წ.).

კვლევის სფერო ჰიდროდინამიკა.

ნავიე-სტოქსის განტოლება არის მოძრაობის განტოლება, რომელიც აღწერს დინებებს. განტოლება შეესაბამება როგორც სითხეების, ისე აირის ნაკადების მოძრაობას.

ნავიე-სტოქსის განტოლება აღწერს სითხის დინამიკის მოძრაობას და წარმოადგენს ფუნდამენტურ განტოლებებს დინამიკური მექანიკის სფეროში. ეს განტოლება გამოიყენება როგორც არაკომპრესირებადი (წყალი), ისე კომპრესირებადი (ჰაერი) სითხეების დინამიკის შესასწავლად. ნავიე-სტოქსის განტოლების არსი მდგომარეობს იმაში, რომ იგი აღწერს სითხეების მოძრაობას და დინამიკას სივრცესა და დროში, ხოლო ამ დინების მოქმედებას ფიზიკური ძალები აკონტროლებენ. იგი წარმოადგენს მასის, იმპულსისა და ენერგიის შენახვის კანონის გამოყენებას სითხეების მიმართ.

სითხის მოძრაობისას მასის შემცირება ან ზრდა არ ხდება, რაც ნიშნავს, რომ სითხე, რომელიც შედის მოცემულ მოცულობაში, ან მთლიანად ინარჩუნებს მასას, ან უნდა გამოვიდეს მოცულობიდან ისე, რომ მასა მუდმივი დარჩეს. ნავიე-სტოქსის განტოლების ზოგადი ფორმაა:

$$\rho \left(\frac{\partial u}{\partial t} + (u * \nabla)u \right) = -\nabla p + \mu \nabla^2 u + f$$

სადაც მარცხენა მხარე იმპულსის ცვალებადობაა, ხოლო მარჯვენა მხარე - ძალები:

ρ - არის სითხის სიმკვრივე,

u - არის სითხის სიჩქარის ვექტორი (სიჩქარე სხვადასხვა წერტილში),

t - არის დრო,

p - არის წნევა,

μ - არის დინამიკური სიბლანტე (კოეფიციენტი, რომელიც მიუთითებს სითხის წინააღმდეგობას დეფორმაციის მიმართ),

$-\nabla p$ - წნევის ძალა,

$\mu \nabla^2 u$ - სიბლანტის მიერ გამოწვეული ხახუნის ძალა,

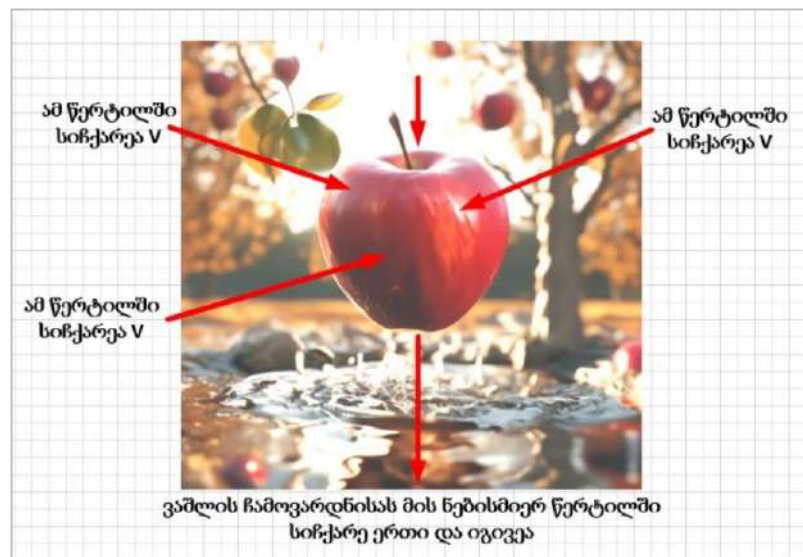
f - არის გარე ძალების ვექტორი (მაგალითად, გრავიტაციის).

უფრო მარტივ შემთხვევაში, თუ დავუშვებთ, რომ დინების ნაკადი მუდმივია და მასა არ იკარგება მაშინ გვექნება:

$$\nabla * \vec{v} = 0, \quad \rho \frac{\partial \vec{v}}{\partial t} = -\nabla p + \mu \nabla^2 \vec{v} + F$$

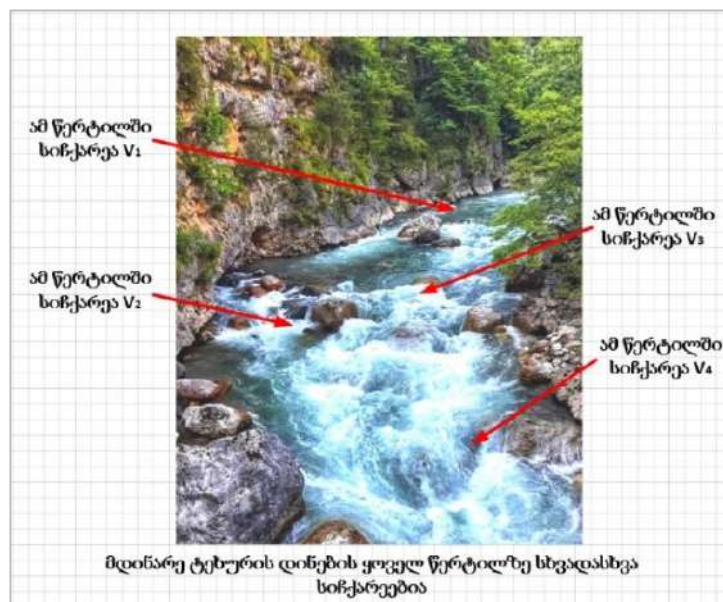
დავუშვათ, რომ პროცესი იზოთერმულია. ასევე ნივთიერება უნდა იყოს ნიუტონური ანუ მასზე მოქმედი წნევა მის სიბლანტეზე არ უნდა ახდენდეს გავლენას.

ვიცით, რომ ხიდან ჩამოვარდნილ ბურთზე ან ვაშლზე მოქმედებს ნიუტონის მეორე კანონი. ამჟამად მნიშვნელოვანია არა ბურთის ან ვაშლის სტრუქტურა, არამედ მხოლოდ მათი მასა. ვაშლის ან ბურთის ვარდნისას მათ ნებისმიერ წერტილში სიჩქარე და აჩქარება ერთი და იგივეა (ნახ. 13.5.).



ნახ. 13.5. ვაშლის ვარდნისას მის ნებისმიერ წერტილში სიჩქარე და აჩქარება არ იცვლება

რაც შეეხება სითხეს, ის მიედინება არაერთგვაროვნად. სითხის დინებისას მის სხვადასხვა წერტილში წნევა და სიჩქარე განსხვავდება ერთმანეთისაგან (ნახ. 13.6.):



ნახ. 13.6. მდინარე ტეხურის სხვადასხვა წერტილში სიჩქარეები იცვლება

ნავიერ-სტოქსის განტოლება აღწერს სითხის მოძრაობას ნებისმიერ პირობებში და საკმაოდ რეალური და რთული ფორმულითაა წარმოდგენილი. ნიუტონის კანონის მარტივი ფორმულის ნაცვლად მივიღეთ დიფერენციალური განტოლება ორი უცნობი ფუნქციით - თითოეულში ერთზე მეტი არგუმენტით. მათემატიკურად სწორია ვუწოდოთ ნაწილობრივი დიფერენციალური განტოლება.

კლასს მათემატიკის ინსტიტუტის ნავიერ-სტოქსის განტოლების გადასაჭრელად, პასუხი უნდა გაეცეს შემდეგ კითხვას - თუ ცნობილია სითხის საწყისი მდგომარეობა, მაშინ არსებობს თუ არა ამონახსნი/ამოხსნა დროის ნებისმიერი მომდევნო მომენტისთვის?

პრობლემა ძალიან აბსტრაქტულია და პასუხი არის: ან არა ან კი (რა თქმა უნდა დამტკიცებით).

ნავიერ-სტოქსის განტოლებას იყენებენ:

1. ფიზიკისა და ინჟინერიის სფეროებში. ნავიერ-სტოქსის განტოლებები გამოიყენება სითხის დინამიკის, განსაკუთრებით ჰიდროდინამიკისა და აეროდინამიკის, შესასწავლად. მათ საფუძველზე არის აგებული დინების ანალიზი, როგორცაა თვითმფრინავების, მანქანების, ხომალდების და სხვა ტექნიკის დიზაინი.

2. ამინდის პროგნოზირებისთვის. ნავიერ-სტოქსის განტოლებები გამოიყენება ატმოსფერული და ოკეანური დინებების შესასწავლად.

3. სქემების მოდელირება. მრავალ პროგრამულ მოდელებში (Computational Fluid Dynamics (CFD)) განტოლებები გამოიყენება კომპიუტერული სიმულაციების შესაქმნელად.

ენერგიის შენახვის კანონის მიხედვით, ენერგია არც წარმოიქმნება და არც ქრება, იგი ერთი სახიდან მეორეზე გადადის. სითხის დინებაში ენერგია ხშირად გადადის კინეტიკურ ენერგიაში (სიჩქარე), პოტენციურ ენერგიაში (წნევა) და სითხოს ენერგიაში (სიბლანტე იწვევს სითხოს გამოყოფას). ნავიერ-სტოქსის განტოლებამ შეიძლება მოგვცეს ინფორმაცია სითხის სიჩქარის, წნევის, და ტემპერატურის შესახებ სივრცის ნებისმიერ წერტილში.

ნავიერ-სტოქსის განტოლება ფუნდამენტური ინსტრუმენტია სითხეების დინამიკის შესასწავლად და გვაძლევს საშუალებას განვსაზღვროთ სითხის მოძრაობის კანონზომიერებები: სიჩქარე და წნევა, მაგრამ მისი გამოყენება მოითხოვს კომპლექსურ მათემატიკურ მეთოდებს ან კომპიუტერულ სიმულაციებს, განსაკუთრებით რთულ, ტურბულენტურ დინებებში. ნავიერ-სტოქსის განტოლების ამოხსნა უმეტესწილად რთულია, რადგან ის არ არის წრფივი, განსაკუთრებით რთული ამოცანაა ტურბულენტური დინებების შესწავლა, ზუსტი ანალიზი ან მოდელირება.

5. ბიორჩ და სვინერთონ-დაიერის ჰიპოთეზა (1960 წ.).

კვლევის სფერო - ალგებრული გეომეტრია.

ბიორჩ და სვინერტონ-დაიერის ჰიპოთეზა დაკავშირებულია რიცხვთა თეორიასთან და შეისწავლის ელიფსურ მრუდებს. ჰიპოთეზის მთავარი იდეა მდგომარეობს იმაში, რომ დავადგინოთ რამდენი რაციონალური წერტილი არსებობს ელიფსურ მრუდეებზე.

ეს ჰიპოთეზა ამყარებს კავშირს ალგებრულ გეომეტრიასა და ანალიზს შორის.

ჰიპოთეზა გვამცნობს, რომ ელიფსური მრუდის რაციონალური წერტილების რიცხვი მჭიდროდ უკავშირდება მრუდის L -ფუნქციის ქვევას, კონკრეტულად, მის მნიშვნელობას $s = 1$ წერტილში. ელიფსური მრუდი გამოისახება ფორმულით:

$$y^2 = x^3 + ax + b$$

სადაც a და b ფიქსირებული მუდმივია, x და y კოორდინატები რაციონალური რიცხვებია (იხ. ნახ. 13.7.).

ბიორჩ და სვინერტონ-დაიერის ჰიპოთეზის ძირითადი არსი:

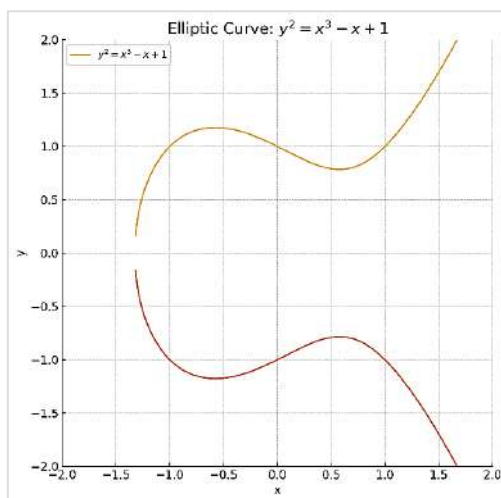
L -ფუნქცია ($L(E, s)$) - არის ელიფსური მრუდის დამახასიათებელი ფუნქცია, რომელიც შეიცავს ინფორმაციას მრუდის შესახებ.

უფრო კონკრეტულად:

თუ L -ფუნქციის მნიშვნელობა $s = 1$ წერტილში არის 0-ის ტოლი, მაშინ ელიფსურ მრუდზე გვაქვს უსასრულოდ ბევრი რაციონალური წერტილი.

თუ L -ფუნქციის მნიშვნელობა $s = 1$ წერტილში არ არის 0-ის ტოლი, მაშინ მრუდზე გვაქვს მხოლოდ შეზღუდული რაოდენობის რაციონალური წერტილი.

ჰიპოთეზა ჯერჯერობით დაუმტკიცებელია, მაგრამ მას მნიშვნელოვანი ადგილი უჭირავს რიცხვთა თეორიაში, განსაკუთრებით პარაბელურ მრუდებსა და მათთან დაკავშირებულ პრობლემებში.



ნახ. 13.7. ელიფსური მრუდი

ნახატ 13.7.-ზე მოცემულია ელიფსური მრუდის გრაფიკი, რომელიც ეფუძნება $y^2 = x^3 + ax + b$ განტოლებას. გრაფიკი შედგება ორი ნაწილისგან: ზედა ნაწილი წარმოადგენს y -ის დადებით მნიშვნელობებს, ხოლო ქვედა - უარყოფით მნიშვნელობებს.

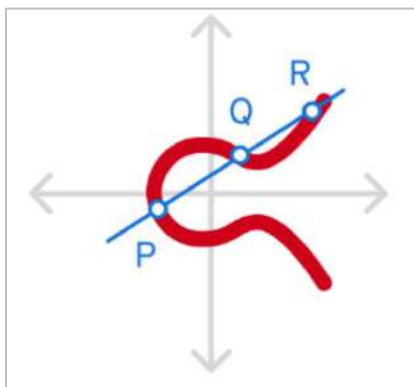
ასეთი მრუდები განეკუთვნებიან რთულ სტრუქტურას და მნიშვნელოვან როლს ასრულებენ რიცხვთა თეორიისა და კრიპტოგრაფიის კვლევებში.

მათემატიკოსებმა შენიშნეს, რომ თუ ელიფსებს გარკვეული კუთხით უყურებთ, წერტილების რაოდენობა მიჰყვება გარკვეულ სქემას. ჰიპოთეზა დადასტურებულია კონკრეტული შემთხვევებისთვის, მაგრამ ზოგადი მტკიცებულება არ არის დადგენილი.

ბირჩისა და სვინერთონ-დაიერის ჰიპოთეზა არის ერთ-ერთი ამოცანა კლასიკური მათემატიკის ინსტიტუტის ათასწლეულის პრიზის შვიდი ამოცანადან.

ელიფსურ მრუდებს აქვს რამდენიმე განსაკუთრებული თვისება, რაც მათ საინტერესოს ხდის როგორც მათემატიკოსებისთვის, ასევე კრიპტოგრაფებისთვის. პირველ რიგში, ელიფსური მრუდები ჰორიზონტალურად სიმეტრიულია x ღერძის მიმართ, მათი ორივე მხარე ერთნაირია, ისევე როგორც სარკისებური გამოსახულება.

გარდა ამისა, ელიფსურ მრუდზე თუ გავავლებთ ნებისმიერ სწორ ხაზს (მხოლოდ, ხაზი არ უნდა იყოს ვერტიკალური), იგი არაუმეტეს სამ წერტილში მაინც გადაკვეთს მრუდს. ქვემოთ მოცემულ ნახატ 13.8-ზე ეს წერტილები აღინიშნა, როგორც P , Q და R .



ნახ. 13.8. ელიფსურ მრუდზე გავლებული წრფე

6. ჰოჯის ჰიპოთეზა (1941 წ.).

კვლევის სფერო - ალგებრული გეომეტრია.

ჰოჯის ჰიპოთეზა (Hodge Conjecture) არის ერთ-ერთი ყველაზე მნიშვნელოვანი ამოუხსნელი ჰიპოთეზა თანამედროვე მათემატიკაში, რომელიც დაკავშირებულია ალგებრული გეომეტრიისა და ტოპოლოგიის რთულ საკითხებთან. ის პირველად წამოაყენა 1940-იან წლებში ბრიტანელმა მათემატიკოსმა უილიამ ჰოჯმა.

ჰოჯის ჰიპოთეზა ეხება ისეთ მათემატიკურ სტრუქტურას, რომელიც გამოიყენება მრავალგანზომილებიანი გეომეტრიული ობიექტების შესასწავლად. ჰოჯის ჰიპოთეზა წარმოადგენს ერთ-ერთ ე. წ. მილენიუმის პრობლემებს, რომლის ამოხსნაც დიდ მიღწევად ჩაითვლება მათემატიკისთვის. თუ ის დადასტურდება, ეს ხელს შეუწყობს ალგებრული გეომეტრიის, ტოპოლოგიისა და რიცხვთა თეორიის განვითარებას.

ჰოჯის ჰიპოთეზა ალგებრული გეომეტრიის ერთ-ერთი უმნიშვნელოვანესი ამოცანაა და შეისწავლის ისეთ გეომეტრიულ ობიექტებს, რომლებიც ალგებრული განტოლებებითაა აღწერილი. ენდრიუ უაილსმა ამ მეთოდებით დაამტკიცა ფერმის თეორემა, მაგრამ დღესდღეობით ზოგადი თეორემა არავის დაუმტკიცებია.

ჩვენს სამყაროში არსებობს მარტივი და რთული ობიექტები. სავსებით ლოგიკურია, რომ რთული ობიექტების აღწერა შესაძლებელია მარტივი და გარკვეული რაოდენობის ობიექტების გამოყენებით.

ჰიპოთეზის მთავარი იდეაა იმის გარკვევა, თუ რამდენად შეგვიძლია მივიღოთ რთული ობიექტის ფორმა მზარდი განზომილების მარტივი ობიექტების ერთმანეთთან შეკვრით.

ჰიპოთეზა ამტკიცებს, რომ „ალგებრულ გეომეტრიაში ჰოჯის კლასი წარმოადგენს ალგებრული ციკლების კლასების რაციონალურ წრფივი კომბინაციას“.

ცნობილია, რომ დღესდღეობით ჰოჯის ჰიპოთეზის დამტკიცება მხოლოდ რამდენიმე განსაკუთრებული შემთხვევისთვის არის შესაძლებელი. უფრო ზოგადი მტკიცებულება ჯერ არ არის ნაპოვნი, ისევე როგორც არ არის ნაპოვნი მტკიცებულება, რომ ჰიპოთეზა არასწორია.

7. პუანკარეს ჰიპოთეზა.

კვლევის სფერო - ტოპოლოგია.

ფრანგმა მათემატიკოსმა ჟიულ ანრი პუანკარემ 1904 წელს გამოთქვა ჰიპოთეზა რომელსაც, სხვანაირად „სამყაროს ფორმულას“-აც უწოდებენ. პუანკარე შეისწავლიდა ისეთი გეომეტრიული ფიგურების საერთო თვისებებს, რომლებიც უცვლელი რჩება ორგანზომილებიან სივრცეში. უფრო დეტალურად კი, ჩვენ შეგვიძლია გარდაქმნათ ერთი ფიგურა მეორეში მხოლოდ დეფორმაციის გზით, „გაჭრის“ და „გაკერვის“ გარეშე.

პუანკარემ დაადგინა, რომ ორგანზომილებიანი სფერო ერთკავშირიანია და ივარაუდა, რომ სამგანზომილებიანი სფეროც იგივეა ანუ გეომეტრიული ფიგურები ინარჩუნებენ თავის თვისებებს სამგანზომილებიან სივრცეშიც, რომელსაც მიეკუთვნება ჩვენი სამყარო.

ეს არის ჰიპოთეზის მთავარი იდეაც, რომლის დამტკიცებასაც ცდილობდნენ კაცობრიობის საუკეთესო მეცნიერები მთელი საუკუნის მანძილზე.

2003 წელს პუანკარეს ჰიპოთეზა ამოხსნა რუსმა მეცნიერმა გრიგორი პერელმანმა, რომელზეც მან რვა წლის განმავლობაში იმუშავა და გადატრიალება მოახდინა მათემატიკურ სამყაროში (იხ. ნახ. 13.8).

გრიგორი იაკობის ძე პერელმანი 1966 წელს დაიბადა ლენინგრადში (ახლანდელი სანქტ-პეტერბურგი). 16 წლის ასაკში გახდა მათემატიკის მსოფლიო ოლიმპიადის გამარ-

ჯვებული, რომელიც ბუდაპეშტში გაიმართა. მისაღები გამოცდების გარეშე მიღებულ იქნა ლენინგრადის სახელმწიფო უნივერსიტეტის მათემატიკურ-მექანიკურ ფაკულტეტზე, დაამთავრა და დაიცვა დისერტაცია.



ნახ. 13.8. გრიგორი პერელმანი. თანამედროვეობის ერთ-ერთი ყველაზე ცნობილი და გენიალური მათემატიკოსი

გრიგორი იაკობის ძე პერელმანი 1966 წელს დაიბადა ლენინგრადში (ახლანდელი სანქტ-პეტერბურგი). 16 წლის ასაკში გახდა მათემატიკის მსოფლიო ოლიმპიადის გამარჯვებული, რომელიც ბუდაპეშტში გაიმართა. მისაღები გამოცდების გარეშე მიღებულ იქნა ლენინგრადის სახელმწიფო უნივერსიტეტის მათემატიკურ-მექანიკურ ფაკულტეტზე, დაამთავრა და დაიცვა დისერტაცია. 1992 წელს პერელმანი ნიუ-იორკის და სტონი ბრუკის უნივერსიტეტებში იყო მიწვეული ლექციების წასაკითხად, მართლაც, 1992 წელს გრიგორი გაემგზავრა სამუშაოდ ნიუ-იორკის უნივერსიტეტში. შერთებულ შტატებში იგი 1996 წლამდე დარჩა და ბოლო პერიოდში ბერკლის უნივერსიტეტში სამეცნიერო მუშაობას ლექციების წაკითხვასაც უთავსებდა.

სამშობლოში დაბრუნებულმა მუშაობა სტეკლოვის სახელობის ინსტიტუტში განაგრძო, სადაც 2003 წლამდე იმუშავა. 2006 წელს ჟურნალმა „Science“ აღიარა მსოფლიოს უდიდეს მეცნიერად. აღსანიშნავია, რომ გრიგორი პერელმანმა 2010 წელს კლემს მათემატიკის ინსტიტუტის მიერ დადგენილი ჯილდოზე, 1 მილიონ დოლარზე, უარი განაცხადა და მისი პასუხი იყო უჩვეულო: „რად მინდა მილიონი დოლარი, როცა მე სამყაროს მართვა შემიძლია!“.

8. მარკოვის ჯაჭვები.

გამოყენების სფერო - სტატისტიკა და ინჟინერია.

ანდრია ანდრიას ძე მარკოვი (1856 - 1922) გამოჩენილი რუსი მათემატიკოსი, რიცხვთა თეორიის, ალბათობის თეორიისა და მათემატიკური ანალიზის სპეციალისტი. იგი სწავლობდა შემთხვევით პროცესებს და დიდი წვლილი შეიტანა ამ დარგის განვითარებაში.

მარკოვის ჯაჭვი არის შემთხვევითი მოვლენების თანმიმდევრობის ერთ-ერთი უმარტივესი შემთხვევა, მაგრამ, მიუხედავად ამისა, ის ხშირად გამოიყენება საკმაოდ რთული პროცესების

აღსაწერად. მარკოვის ჯაჭვი არის შემთხვევითი მოვლენების თანამიმდევრობა, რომელშიც თითოეული მოვლენის ალბათობა დამოკიდებულია მხოლოდ წინა მოვლენაზე და არ არის დამოკიდებული უფრო ადრინდელზე.

მარკოვის ჯაჭვი ეწოდება ცდათა მიმდევრობას, თუ n -ური ცდის შედეგი დამოკიდებულია მხოლოდ წინა $n - 1$ ცდის შედეგზე და არაა დამოკიდებული ადრე ჩატარებული არცერთი ცდის შედეგზე. მარკოვის ჯაჭვები შეგვიძლია განვიხილოთ, როგორც სისტემის ერთი მდგომარეობიდან მეორეში გადასვლის ალბათობების აღწერა. ყველა ალბათობათა ჯამი უნდა იყოს 1-ის ტოლი.

მაგალითად, დავუშვათ, რომ გვაქვს მცირე საქმიანობა (ბიზნესი) და გვჭირდება მისი რეკლამირება. რეკლამირების სქემა შეგვიძლია წარმოვიდგინოთ სამი შესაძლებლობით:

- აქტიური - რეკლამა არის აქტიური;
- არ არის აქტიური - რეკლამა არ არის აქტიური;
- უცნობი - საქმიანობის მდგომარეობა უცნობია ან არ არის გარკვეული.

მარკოვის ჯაჭვის მოდელის აღსაწერად გვჭირდება გადასვლის ალბათობები და შემდეგი სამი მდგომარეობა: 1. S_1 - აქტიურია; 2. S_2 - არ არის აქტიური; 3. S_3 - უცნობია.

მარკოვის ჯაჭვის მდგომარეობების გადასვლის ალბათობები შეგვიძლია წარმოვადგინოთ შემდეგნაირად:

- აქტიურია - $S_1 = 0.6$; $S_2 = 0.3$; $S_3 = 0.1$;
- არ არის აქტიური - $S_1 = 0.4$; $S_2 = 0.5$; $S_3 = 0.1$;
- უცნობია - $S_1 = 0.2$; $S_2 = 0.2$; $S_3 = 0.6$;

ეს ნიშნავს, რომ:

1. თუ საქმიანობის მდგომარეობა **აქტიურია**, მაშინ 60%-იანი ალბათობით ის კვლავ აქტიური დარჩება, 30%-იანი ალბათობით გადავა **არ არის აქტიური** მდგომარეობაში და 10%-იანი ალბათობით მდგომარეობა იქნება **უცნობია**.

2. თუ მდგომარეობა **არ არის აქტიური**, მაშინ 40%-იანი ალბათობით დაბრუნდება **აქტიურ** მდგომარეობაში, 50%-იანი ალბათობით დარჩება **არ არის აქტიური** მდგომარეობაში და 10%-იანი ალბათობით მდგომარეობა იქნება **უცნობია**.

3. თუ მდგომარეობა **უცნობია**, მაშინ 20%-იანი ალბათობით გახდება **აქტიური**, 20%-იანი ალბათობით გადავა **არ არის აქტიური** მდგომარეობაში, ხოლო 60%-იანი ალბათობით მდგომარეობა დარჩება **უცნობია**.

მარკოვის ჯაჭვი შეიძლება გამოყენებულ იქნას ამინდის პროგნოზირებაში, ასევე მარკეტინგული სტრატეგიის შემუშავებაში. მარკოვის ჯაჭვების საფუძველზე შეიქმნა ისეთი აპლიკაციები, როგორიცაა Google PageRank, Subreddit, ტექსტის გენერატორი, სიტყვებისა და წინადადებების ავტომატური შევსება.

დიდი მონაცემები. სტრუქტურები, შენახვის, წარმოდგენისა და დამუშავების პრობლემები. დიდ მონაცემთა სისტემები.

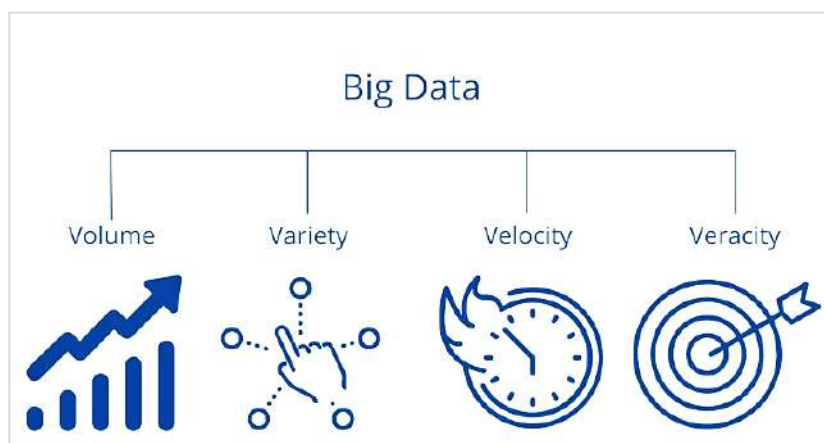
დიდ მონაცემები (Big Data). დიდი მონაცემები არის ძალიან დიდი მოცულობის მონაცემები, რომელთა შეგროვება, შენახვა, დამუშავება, ანალიზი და ვიზუალიზაცია ტრადიციული მონაცემთა მართვის მეთოდებით რთულია ან შეუძლებელი. დიდი მონაცემების ძირითადი მახასიათებლებია (იხ. ნახ. 14.1.):

1. მოცულობა (Volume). მონაცემების უზარმაზარი რაოდენობა, რომლებიც გენერირდება სხვადასხვა წყაროდან, მათ შორის სოციალური ქსელებიდან, სენსორებიდან, მობილური აპლიკაციებიდან და სხ.

2. მრავალფეროვნება (Variety). მონაცემების მრავალფეროვნება ფორმატის მიხედვით. მონაცემები შეიძლება იყოს სტრუქტურირებული (მაგ., ცხრილები), არასტრუქტურირებული (მაგ., ტექსტი, ვიდეო) ან ნახევრად სტრუქტურირებული (მაგ., XML ფაილები).

3. სისწრაფე (Velocity). მონაცემთა წარმოქმნის და მიღების სიჩქარე. Big Data-ს მონაცემები მუდმივად ახლდება, რაც მოითხოვს მონაცემების სწრაფ და მაღალი ხარისხით დამუშავებას.

4. სანდობა/სიზუსტე (Veracity). მონაცემების ხარისხი და სანდობა. დიდი რაოდენობის მონაცემები ხშირად შეიცავს არასწორ ან შეუსაბამო ინფორმაციას, რაც ართულებს და ხელს უშლის სწორი ანალიზის ჩატარებას. მოპოვებული მონაცემების სიზუსტემ შეიძლება მნიშვნელოვანი ზეგავლენა იქონიოს მიღებულ შედეგზე.



ნახ. 14.1. დიდი მონაცემების ძირითადი მახასიათებლები (4V-ს პრინციპი)

დიდი მონაცემების დამუშავება, შენახვა და წარმოდგენა დაკავშირებულია ტექნიკურ გამოწვევებთან და ისეთი სახის პრობლემებთან, რომლებიც წარმოიქმნება მონაცემების მასშტაბის, სიჩქარის, მრავალფეროვნებისა და სისწორის გამო. ეს პრობლემები მოიცავს შემდეგ საკითხებს:

1. მონაცემების სტრუქტურები. დიდი მონაცემების დამუშავებისთვის გამოყენებული სტრუქტურები უნდა იყოს ეფექტური მეხსიერების გამოყენების მხრივ: ჰეშ-ცხრილები (Hash Tables), ბინარული ძიების ხეები (Binary Search Trees) ან B-ხეები (B-Trees) ხშირად გამოიყენება.

2. შენახვის პრობლემები. ერთი მოწყობილობა ხშირად ვერ იტევს დიდ მონაცემებს, ამიტომ მონაცემები ნაწილდება მრავალ ფიზიკურ ან ვირტუალურ სერვერზე. მაგალითად, NoSQL მონაცემთა ბაზა გამოიყენება მონაცემთა ნაწილებად განაწილებისთვის.

დიდი მონაცემების დამუშავებისას ხშირად წარმოიქმნება ჭარბი მონაცემები, რაც იწვევს მეხსიერების გაუმართავ ხარჯვას. ამის თავიდან ასაცილებლად გამოიყენება მონაცემთა შეკუმშვის ტექნოლოგიები: როგორიცაა Snappy ან Zstandard.

3. მონაცემების წარმოდგენა. დიდი მონაცემების დამუშავების შემდეგ რთულია შედეგების სწორად და ეფექტურად წარმოდგენა. ვიზუალიზაციის ტექნოლოგიები (მაგალითად, Tableau, Power BI) ეხმარება მონაცემების ნიმუშების, ტენდენციების და კორელაციების გრაფიკულ წარმოჩენას. ვიზუალიზაციის ინსტრუმენტებმა უნდა უზრუნველყონ ეფექტური მუშაობა დიდი მონაცემების ნაკრებებთან ისე, რომ ვიზუალიზაციის შედეგები იყოს ოპტიმალური და მარტივად გასაგები.

4. მონაცემების დამუშავება. დიდი მონაცემების დამუშავება ხშირად მოითხოვს მასშტაბირებად გამოთვლით რესურსებს: MapReduce, Hadoop, Spark. ისეთი სახის ტექნოლოგიები, როგორიცაა Apache Kafka და Apache Flink, გამოყენებულია რეალურ დროში მონაცემთა ნაკადების ეფექტურად მართვისთვის.

5. მონაცემთა ხარისხი და სისწორე. მონაცემთა დიდი ნაკრებები ხშირად შეიცავს ასლებს ან „ცუდ“ მონაცემებს, რაც იწვევს შეცდომებს ანალიზის ჩატარების დროს.

ალგორითმები და მონაცემთა გასუფთავების ინსტრუმენტები უზრუნველყოფენ მონაცემთა ხარისხის გაზრდას. აუცილებელია მონაცემების სიზუსტის დაცვა და მათი „გატეხილი“ ან არაზუსტი მონაცემების თავიდან აცილება. ამისთვის გამოიყენება მონაცემთა ინტეგრაციის და განახლების ალგორითმები. მონაცემების ეფექტურად მართვისთვის საჭიროა სპეციალური ინსტრუმენტები და მეთოდოლოგიები, რათა უზრუნველყოფილ იქნეს სისტემების მასშტაბირება, გამოთვლითი რესურსების მართვა და შედეგების სწრაფი ანალიზი.

ტერმინი **დიდი მონაცემები (Big Data)** ბოლო ათწლეულში ფართოდ გავრცელდა, მაგრამ მას საკმაოდ შორი ისტორია აქვს. დიდი მონაცემების ისტორია რამდენიმე ფაზად შეიძლება დაიყოს, რაც ასახავს მონაცემთა დამუშავების ტექნოლოგიების განვითარებას და ტერმინის მნიშვნელობის ევოლუციას.

1970-1980 წლები, მონაცემთა ზრდის დასაწყისი. 1970-იანი წლებიდან დაიწყო მონაცემთა ზრდის ეტაპი. მონაცემთა ბაზების მართვის სისტემები, როგორიცაა SQL და რელაციური მონაცემთა ბაზები, შეიქმნა ინფორმაციის შენახვის და დამუშავების ეფექტურად მართვისთვის. თუმცა, იმ დროისთვის მონაცემთა მოცულობები არ იყო ისეთი დიდი, რომ მათ დამუშავებას „Big Data“ დარქმეოდა.

1990-იანი წლები, მონაცემთა ანალიტიკის ზრდა. 1990-იანი წლების ბოლოს, ინტერნეტის და ტექნოლოგიური წინსვლის წყალობით, მონაცემთა რაოდენობა სწრაფად გაიზარდა. კომპანიები და ორგანიზაციები აცნობიერებდნენ, რომ მონაცემებისგან შესაძლებელი იყო მნიშვნელოვანი ინფორმაციის მიღება. Data mining და OLAP (Online Analytical Processing) ტექნოლოგიები განვითარდა, რაც დაეხმარა ორგანიზაციებს განხორციელებინათ დიდი მოცულობის მონაცემების ანალიზი.

2000-იანი წლები, ტერმინი „დიდი მონაცემები (Big Data)-ს“ დამკვიდრება. ტერმინი „დიდი მონაცემები (Big Data)“ პირველად გამოიყენა John Mashey-მა, Silicon Graphics-ის კომპიუტერული მეცნიერის თანამშრომელმა 2000-იანი წლების დასაწყისში. იმ დროს, კომპანიები ხედავდნენ, რომ მონაცემთა რაოდენობა ისე იზრდებოდა, რომ ტრადიციული მეთოდები აღარ იყო საკმარისი მათ დასამუშავებლად. ინტერნეტის, მობილური ტელეფონებისა და სოციალური მედიის სწრაფმა განვითარებამ მონაცემთა რაოდენობა მკვეთრად გაზარდა.

Doug Laney-მა 2001 წელს შემოგვთავაზა „Big Data-ს“ ცნობილი „3Vs“ მოდელი (Volume, Variety, Velocity), რომელიც აღწერს მონაცემების მასშტაბს, მრავალფეროვნებას და წარმოების სიჩქარეს.

2000-იანი წლების შუა პერიოდი, ტექნოლოგიური გარღვევები. Google, Yahoo და სხვა ტექნოლოგიური გიგანტები იმ დროისთვის დიდი მონაცემების მართვის გამოწვევის წინაშე დადგნენ. Google-მა შექმნა MapReduce, სისტემა, რომელიც ახდენდა სერვერებზე მონაცემების პარალელურ დამუშავებას და მისი დამხმარე ინსტრუმენტები, როგორიცაა Hadoop, მალევე გახდა Big Data ტექნოლოგიების სტანდარტი. Apache Hadoop (2006) და Apache Spark (2010) განვითარდა როგორც ღია კოდის პლატფორმები, რომლებიც დაეხმარა დიდ კომპანიებს პარალელური დამუშავების განხორციელებაში და მონაცემთა ანალიზში.

2010-იანი წლები და დღევანდელი, Big Data-ს რევოლუცია. 2010-იან წლებში დიდი მონაცემების გამოყენება ფართოდ გავრცელდა არა მხოლოდ ტექნოლოგიურ კომპანიებში, არამედ სხვადასხვა ინდუსტრიაში, მათ შორის ჯანდაცვაში, ფინანსებში, სამეცნიერო კვლევებში და საზოგადოებრივ სექტორებში. მონაცემთა ბაზების სისტემა განვითარდა, რაც მოიცავდა NoSQL მონაცემთა ბაზებს, როგორიცაა MongoDB და Cassandra, რომლებიც უკეთ

ამუშავებენ არასტრუქტურირებულ მონაცემებს. დიდი მონაცემები (Big Data) ვრცელდება ისეთი სახის სფეროებსა და ინდუსტრიებში, სადაც მონაცემთა მასშტაბები უზარმაზარია და საჭიროა მათი სწრაფი ანალიზი, ეფექტური დამუშავება და მონაცემებთან წვდომა.

ქვემოთ მოცემულია იმ ორგანიზაციებისა და სექტორების ჩამონათვალი, რომლებიც იყენებენ დიდ მონაცემებს:

1. სოციალური მედია. მილიარდობით მომხმარებელი ყოველდღიურად იყენებს პლატფორმებს Facebook, Twitter და Instagram, რის შედეგადაც იქმნება უზარმაზარი რაოდენობის მონაცემები (პოსტები, ლაიქები, კომენტარები, გაზიარებები).

მონაცემები: მომხმარებელთა პროფილები, ურთიერთქმედებები, ფოტოები, ვიდეოები.

გამოყენება: სოციალური ქსელები იყენებენ ამ მონაცემებს მომხმარებლების ინტერესების პროგნოზირებისთვის, რეკლამების პერსონალიზებისთვის და მომხმარებლების ქცევის ანალიზისთვის.

2. ჯანდაცვა. დიდი მოცულობის სამედიცინო მონაცემების - პაციენტების ისტორიები, კვლევების შედეგები, მედიკამენტების მოხმარება - ჩაწერა ხდება ელექტრონული სამედიცინო ჩანაწერების (EMR) სახით.

მონაცემები: პაციენტის ჯანმრთელობის ჩანაწერები, დიაგნოზები, გენეტიკური მონაცემები.

გამოყენება: ანალიზი გამოიყენება დაავადებების პროგნოზირებისთვის, ეპიდემიის გავრცელების პროგნოზირების, ახალი წამლების განვითარებისა და სამედიცინო მომსახურების გაუმჯობესებისთვის.

3. ელექტრონული კომერცია. ონლაინ სავაჭრო პლატფორმები აგროვებენ მომხმარებლების საძიებო შეკითხვებს, ყიდვის ისტორიებს და სხვა ქმედებებს (Amazon, Alibaba, eBay).

მონაცემები: პროდუქტების მოძებნა, ყიდვის ისტორია, მომხმარებლების მიმოხილვები.

გამოყენება: პერსონალიზირებული რეკომენდაციები, ინვენტარის ოპტიმიზაცია, დინამიკური ფასების განსაზღვრა და მარკეტინგული სტრატეგიების გაუმჯობესება.

4. საფინანსო სექტორი. დიდი მონაცემების ანალიზი გამოიყენება ბანკებში და სადაზღვევო კომპანიებში რისკების შეფასებისა და თაღლითობის გამოვლენისთვის.

მონაცემები: ტრანზაქციები, საკრედიტო ქულები, ინვესტიციები.

გამოყენება: თაღლითობის გამოვლენა, საკრედიტო რისკის შეფასება, საინვესტიციო ტენდენციების პროგნოზირება.

5. ენერგეტიკის სექტორი. დიდი მონაცემები გამოიყენება ელექტროენერგიის მოხმარების ანალიზისთვის, ქსელის ოპტიმიზაციისთვის და ალტერნატიული ენერგიის გამოყენების პროგნოზირებისთვის.

მონაცემები: სენსორები, მომხმარებელთა მოხმარების ინფორმაცია, ქსელური მონაცემები.

გამოყენება: ენერგიის მოთხოვნის პროგნოზირება, მოხმარების მაქსიმალური დატვირთვის ოპტიმიზაცია, ეფექტური რესურსების გამოყენება.

6. სატრანსპორტო სექტორი. სატრანსპორტო კომპანიები და ნავიგაციის სისტემები იყენებენ სენსორებიდან მიღებულ მონაცემებს, სატელიტურ სურათებსა და მომხმარებლების ადგილმდებარეობის ინფორმაციას (Google Maps, Uber).

მონაცემები: ტრეფიკის მოძრაობა, მომხმარებლების ადგილმდებარეობა, ტრანსპორტის წვდომა.

გამოყენება: მოძრაობის პროგნოზირება, ოპტიმალური მარშრუტების მოძიება, სერვისის ოპტიმიზაცია.

7. საინფორმაციო ტექნოლოგიები და კიბერუსაფრთხოება. მონაცემთა ანალიზი გამოიყენება ქსელებზე არასწორი ქცევის გამოვლენისა და მავნე პროგრამების მონიტორინგისთვის.

მონაცემები: ქსელური ლოგები, ფაილების ცვლილებები, მომხმარებლის ავტორიზაციის მცდელობები.

გამოყენება: ქსელის დაცვა, მავნე აქტივობების გამოვლენა, დაუცველობების იდენტიფიცირება.

8. სახელმწიფო და კვლევითი ინსტიტუტები. მეცნიერები აგროვებენ უზარმაზარ მონაცემებს კოსმოსური კვლევებისთვის ან ნაწილაკების ფიზიკისთვის (NASA, CERN).

მონაცემები: ასტრონომიული დაკვირვებები, ექსპერიმენტული მონაცემები.

გამოყენება: ფიზიკური კანონების მოდელირება, გლობალური ტენდენციების პროგნოზირება, კლიმატის ცვლილების შესწავლა.

9. კლიმატის კვლევა. დიდი მონაცემთა სისტემები გამოიყენება კლიმატის მონაცემების ანალიზისთვის, მათ შორის გლობალური ტემპერატურის, ატმოსფერული პირობებისა და ოკეანის ნაკადების შესასწავლად.

მონაცემები: მეტეოროლოგიური მონაცემები, ატმოსფერული ჰაერის ხარისხის მონაცემები, ოკეანის მიმოქცევის დინამიკა.

გამოყენება: გლობალური დათბობის პროგნოზირება, ბუნებრივი კატასტროფების მართვა.

10. მონაცემთა ანალიზი სპორტში. სპორტული გუნდები და ლიგები: დიდი მონაცემების გამოყენება ხდება სპორტულ ანალიტიკაში, რათა შეფასდეს მოთამაშეთა მაჩვენებლები და თამაშების ტაქტიკა.

მონაცემები: მოთამაშეთა სტატისტიკა, თამაშის ვიდეოანალიზი, ფიზიკური მახასიათებლები.

გამოყენება: გუნდის სტრატეგიის გაუმჯობესება, მოთამაშეების ფიზიკური მდგომარეობის მონიტორინგი.

დიდი მონაცემების ზოგადი სქემის გრაფიკული ვიზუალიზაცია (Big Data Architecture Diagram) არის პროცესი, რომლის მეშვეობითაც შესაძლებელია მონაცემთა სტრუქტურის უკეთ აღქმა და ანალიზი.

დიდი მონაცემების ზოგადი სქემა შეიძლება მოიცავდეს მონაცემთა შეგროვების, შენახვის, დამუშავების და ანალიზის გრაფიკულ გამოსახვას, რაც გვეხმარება მონაცემთა ნაკადის უკეთ გაგებასა და მართვაში (იხ. ნახ. 14.2.).

ქვემოთ მოცემულია დიდი მონაცემების ზოგადი სქემის ძირითადი ელემენტები:

1. მონაცემთა შეგროვება. მონაცემები შეიძლება შემოვიდეს მრავალი წყაროდან: სოციალური მედია, სენსორები, IoT - ინტერნეტის მოწყობილობები, ბიზნეს ტრანზაქციები, მობილური აპლიკაციები, კლიმატური მონაცემები.

მონაცემები შეიძლება იყოს სხვადასხვა ფორმატის: SQL მონაცემები, ვიდეო და აუდიო ჩანაწერები, ტექსტები, სურათები და მრავალი სხ.

2. მონაცემთა სტრუქტურის გაანალიზება. აუცილებელია წინასწარ განსაზღვროთ მონაცემთა ტიპები: ინდექსები, ურთიერთკავშირები, ცხრილები ან გრაფები.

3. მონაცემთა ურთიერთკავშირის განსაზღვრა. საჭიროა განსაზღვროს კავშირი დიდი მონაცემების სხვადასხვა ელემენტებს შორის. ეს შეიძლება იყოს მონაცემთა ბაზის გასაღები (foreign keys), კავშირები (relationships) ან ურთიერთობები გრაფში.

4. მონაცემთა დამუშავება (Data Processing). რეალურ დროში დამუშავება ან ისეთი სისტემების გამოყენება, როგორიცაა Apache Spark, Flink, Storm ან MapReduce.

5. მონაცემთა შენახვა. დისტრიბუციული ფაილური სისტემები: HDFS (Hadoop Distributed File System), Amazon S3 და სხვა Cloud პლატფორმები; NoSQL მონაცემთა ბაზები: MongoDB, Cassandra, Hbase; Data Lake: ყველა სახის მონაცემების შენახვა ერთ დიდ საცავში.

6. მონაცემთა ნაკადი (Data Ingestion). ETL/ELT პროცესები (Extract, Transform, Load), სადაც ხდება მონაცემების შეგროვება და შემდგომი ანალიზისთვის გამზადდება.

4. მონაცემთა ანალიზი და მოდელირება.

4.1. მონაცემთა ანალიზი: Apache Hive, Presto: SQL-ის გამოყენება დიდი მონაცემების ანალიზისთვის; Data Mining: მონაცემების დამუშავება და შედეგის გამოტანა.

4.2. Machine Learning (ML) მოდელები: მანქანური სწავლების ალგორითმები გამოიყენება დიდი მონაცემების საფუძველზე პროგნოზირებისთვის, კლასიფიკაციისთვის და კლასტერული ანალიზისთვის.

4.3. გრაფის ანალიზი: Neo4j, Giraph სისტემებით ხდება სოციალური ქსელების მონაცემების გრაფის სახით მოდელირება.

5. მონაცემთა დაცვა და კონფიდენციალურობა. მონაცემების უსაფრთხოების უზრუნველსაყოფად გამოიყენება მონაცემთა დაშიფვრა, წვდომის მართვა და კონფიდენციალურობის დაცვის ზომები, როგორიცაა GDPR და სხვა რეგულაციები.

6. გრაფიკული ვიზუალიზაციის ინსტრუმენტები:

გამძლეობა და ხელმისაწვდომობა. მონაცემთა სისტემები განლაგებულია სხვადასხვა სერვერებსა და საცავებში, რათა უზრუნველყოფილი იქნეს მონაცემთა დაცვა და სისტემა ყოველთვის ხელმისაწვდომი იყოს.

დიდი მონაცემთა სისტემები არის აუცილებელი ინსტრუმენტები თანამედროვე ბიზნესებისთვის, მეცნიერებისთვის და მრავალი სხვა სექტორისთვის, რათა ეფექტურად დამუშავდეს და განხორციელდეს უზარმაზარი მოცულობის მონაცემების ანალიზი.

სხვადასხვა სისტემებისა და არქიტექტურების არჩევა დამოკიდებულია იმაზე, თუ რა ტიპის მონაცემების დამუშავება და შენახვაა საჭირო, როგორია დატვირთვა და რა სისწრაფით უნდა მოხდეს მონაცემების ანალიზი.

CAP თეორემა, SQL და NoSQL სისტემები.

CAP თეორემა (Consistency, Availability, Partition tolerance) აღწერს განაწილებული მონაცემთა ბაზის ძირითად თვისებებს: **მთლიანობა** (Consistency), **ხელმისაწვდომობა** (Availability) და **განაწილებადობა** (Partition tolerance).

CAP თეორემა პირველად წამოაყენა ერიკ ბრიუერმა 2000 წელს და ის ერთ-ერთ მთავარ პრინციპად მიიჩნევა განაწილებული სისტემებში.

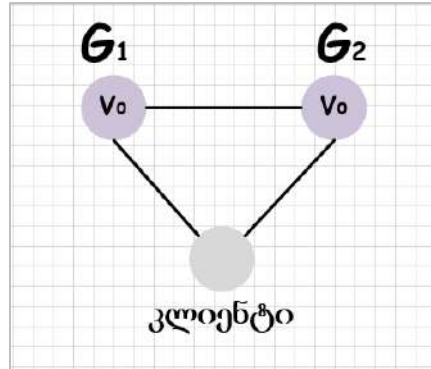
განაწილებად სისტემებში მოცემული სამი თვისებიდან მხოლოდ ორის მიღწევაა შესაძლებელი. იმის მიხედვით თუ როგორი სისტემა გვინდა მივიღოთ საბოლოოდ, უნდა ავირჩიოთ ან მთლიანობა და განაწილებადობა (CP) ან ხელმისაწვდომობა და განაწილებადობა (AP).

თუ სისტემა მხოლოდ ერთ გარემოზე (ერთ მონაცემთა სერვერზე) მუშაობს მაშინ CAP თეორემა აზრს კარგავს ასეთი სისტემებისთვის. რელაციური მონაცემთა ბაზები სწორედ ასეთ სისტემებს მიეკუთვნებიან.

მონაცემების გადანაწილება სხვადასხვა სერვერებზე, აქ რელაციურ მოდელში, თითქმის შეუძლებელია, ამიტომ მონაცემების ზრდასთან ერთად ერთადერთი რაც შეგვიძლია, სერვერის ტექნიკური მახასიათებლების გაუმჯობესებაა. სწორედ ამიტომ რელაციური მოდელი ეფუძნება არა CAP არამედ ACID თეორემას.

მაგალითად: ოპერატიული მეხსიერების გაზრდა, უკეთესი პროცესორის ან/და მყარი დისკის ჩადგმა და ა.შ. ხოლო, რაც შეეხება არარელაციურ მოდელს, აქ მონაცემების სხვადასხვა სერვერებზე გადანაწილება შესაძლებელია, რაც არარელაციური მონაცემთა ბაზების დისტრიბუციულობას ნიშნავს, შესაბამისად ის ასეთი სისტემები CAP თეორემას ეფუძნებიან. დავუბრუნდეთ თეორემას და განვსაზღვროთ CAP - აკრონიმის მნიშვნელობა დისტრიბუციულ სისტემებში. თავდაპირველად, პატარა მაგალითის მეშვეობით განვსაზღვროთ თუ რას ნიშნავს დისტრიბუციული სისტემა.

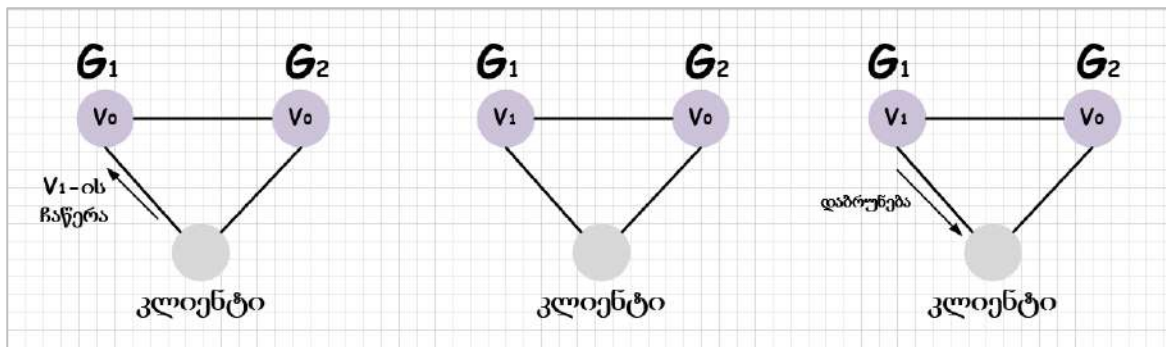
დისტრიბუციული სისტემის მაგალითი. ჩვენი სისტემა შედგება ორი სერვერისგან G_1 და G_2 . ორივე სერვერზე ინახება ერთიდაიგივე მონაცემი - v , რომლის მნიშვნელობაც მიმდინარე მომენტში არის v_0 , ორივე სერვერზე. G_1 და G_2 სერვერებს შეუძლიათ ერთმანეთთან კომუნიკაცია, ხოლო კლიენტს შეუძლია ორივე მათგანთან (ნახ. 14.3.).



ნახ. 14.3. დისტრიბუციული სისტემის მაგალითი

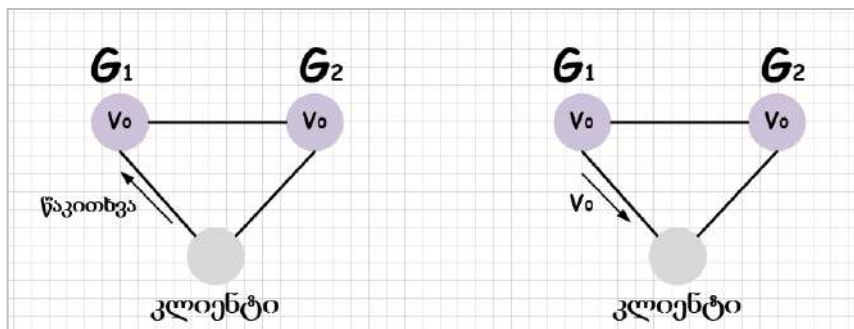
კლიენტს შეუძლია განახორციელოს ჩაწერის ან წაკითხვის ოპერაცია ნებისმიერ სერვერზე, ხოლო, როცა სერვერი მიიღებს მოთხოვნას კლიენტისგან, მან უნდა დაამუშაოს ის და დაუბრუნოს პასუხი. მაგალითად:

ჩაწერის ოპერაცია - კლიენტმა მოითხოვა ჩაწერის ოპერაცია G_1 სერვერზე. მას შემდეგ რაც G_1 -ზე ჩაიწერა v_1 , სერვერმა დაუბრუნა შეტყობინება კლიენტს წარმატებული ჩაწერის შესახებ (ნახ. 14.4.).



ნახ. 14.4. ჩაწერის ოპერაცია - კლიენტმა მოითხოვა ჩაწერის ოპერაცია G_1 სერვერზე

წაკითხვის ოპერაცია - კლიენტმა წაკითხა G_1 სერვერიდან v ცვლადის მნიშვნელობა, სერვერმა დაუბრუნა v_0 ცვლადის მიმდინარე მნიშვნელობა G_1 სერვერზე (ნახ. 14.5.).

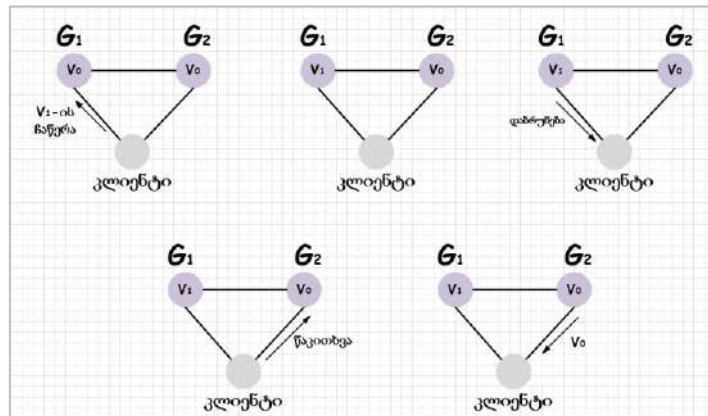


ნახ. 14.5. წაკითხვის ოპერაცია - კლიენტმა წაკითხა G_1 სერვერიდან v ცვლადის მნიშვნელობა

ახლა კი ზემოთ მოყვანილი დისტრიბუციული სისტემის მაგალითზე განვსაზღვროთ, თუ რას ნიშნავს - **Consistency, Availability და Partition tolerance** თვისებები CAP თეორემაში.

Consistency არის თვისება, რომელიც უზრუნველყოფს დაყოფილი სისტემის თანმიმდევრულობას და ერთიანობას. ასეთ სისტემაში მას შემდეგ, რაც კლიენტი შეცვლის/ჩაწერს ახალ მნიშვნელობას v ცვლადში და სერვერიც დადებით პასუხს დაუბრუნებს, კლიენტმა უნდა მიიღოს შეცვლილი მნიშვნელობა ყველა სხვა სერვერიდანაც.

არათანამიმდევრული სისტემის მაგალითი (ნახ. 14.6.):

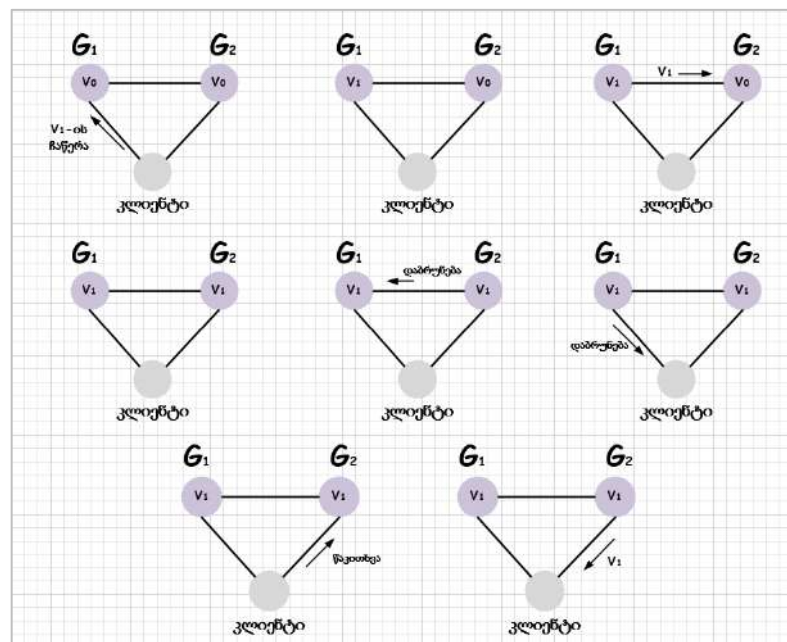


ნახ. 14.6. არათანამიმდევრული სისტემა

კლიენტმა G_1 სერვერზე ჩაწერა v ცვლადის ახალი მნიშვნელობა v_1 , მაგრამ არ მოხდა ცვლადის განახლება G_2 სერვერზე.

აქედან გამომდინარე, როცა კლიენტმა მოითხოვა v ცვლადის მნიშვნელობა G_2 სერვერიდან მან მიიღო ძველი მნიშვნელობა. ასეთ სისტემას არათანამიმდევრული სისტემა ეწოდება.

თანამიმდევრული სისტემის მაგალითი (ნახ. 14.7.):



ნახ. 14.7. თანამიმდევრული სისტემის მაგალითი

ამ შემთხვევაში G_1 სერვერი G_2 სერვერთან კომუნიკაციის შემდეგ უბრუნებს პასუხს მომხმარებელს. კომუნიკაცია კი ნიშნავს G_2 სერვერზე არსებული ცვლადის განახლებას.

შესაბამისად, როდესაც მომხმარებელი G_2 სერვერიდან წაიკითხავს მონაცემს მას განახლებული მნიშვნელობა დაუბრუნდება. ასეთ სისტემას თანამიმდევრული, ერთიანი სისტემა ეწოდება.

Availability თვისების თანახმად სისტემა ყოველთვის უნდა პასუხობდეს კლიენტს, მიუხედავად იმისა მოხდება თუ არა რაიმე ტიპის შეცდომა სისტემაში. ჩვენი მაგალითიდან გამომდინარე, თუ კლიენტი მოითხოვს ახალი მონაცემის ჩაწერას რომელიმე „მუშა“ სერვერზე, სერვერი ვალდებულია ყოველთვის დაუბრუნოს პასუხი მომხმარებელს.

Partition tolerance თვისების თანახმად სისტემამ უნდა იმუშაოს სწორად, მიუხედავად G_1 და G_2 სერვერებს შორის კავშირის ხარვეზებისა. ხარვეზები შეიძლება იყოს, ინტერნეტ კავშირის არ არსებობა კონკრეტულ მომენტში, მონაცემების გადაცემის შენელება და ა.შ.

SQL და NoSQL მონაცემთა ბაზები. SQL და NoSQL მონაცემთა ბაზები განსხვავებულად აგვარებენ მონაცემების შენახვის, დამუშავებისა და ორგანიზების საკითხებს. თითოეულ მათგანს აქვს თავისი ძლიერი და სუსტი მხარეები, რაც მათ სხვადასხვა ტიპის პროექტებისთვის აქცევს ეფექტურ არჩევანად.

1. SQL სისტემები (რელაციური მონაცემთა ბაზები). SQL (Structured Query Language) მონაცემთა ბაზები ეყრდნობიან რელაციურ მოდელს, სადაც მონაცემები ინახება ცხრილებში, რომლებიც დაკავშირებულია ერთმანეთთან გარკვეული წესებით. SQL მონაცემთა ბაზებს მკაცრად განსაზღვრული სქემა აქვთ, რაც მნიშვნელოვნად განსაზღვრავს მონაცემთა სტრუქტურას. SQL-ის ძირითადი მახასიათებლებია:

❖ **სტრუქტურირებული მონაცემები.** მონაცემები ინახება ცხრილებში და არის მკაცრად სტრუქტურირებული. ყოველი ცხრილი შეიცავს სვეტებს (ატრიბუტებს), რომელთა ტიპი წინასწარ განსაზღვრულია.

❖ **სქემაზე დაფუძნებული მონაცემთა მოდელი.** SQL ბაზების სქემა მკაცრია, რაც ნიშნავს, რომ მონაცემთა სტრუქტურა წინასწარ უნდა იყოს დადგენილი. მონაცემთა ცვლილება, მაგალითად, ახალი სვეტის დამატება, მოითხოვს სქემის შეცვლას.

❖ **ACID თვისებები.** SQL მონაცემთა ბაზები ეყრდნობა ACID (Atomicity, Consistency, Isolation, Durability) მახასიათებლებს, რაც უზრუნველყოფს ტრანზაქციების სანდოობას.

❖ **მონაცემთა ურთიერთდაკავშირება.** SQL მონაცემთა ბაზები უზრუნველყოფენ ძლიერი კავშირის უზრუნველყოფას სხვადასხვა ცხრილს შორის. ეს არის ერთ-ერთი მთავარი მიზეზი, რის გამოც ისინი ფართოდ გამოიყენება კომპლექსურ სისტემებში.

SQL მონაცემთა ბაზის მაგალითებია: **MySQL, PostgreSQL, Oracle, Microsoft SQL Server.**

SQL-ის გამოყენების სფეროებია: ა) ფინანსური სისტემები, სადაც საჭიროა მონაცემთა სრული კონსისტენტურობა; ბ) CRM (Customer Relationship Management) სისტემები, სადაც მონაცემთა ურთიერთკავშირი კრიტიკულია; გ) ნებისმიერი სისტემა, სადაც საჭიროა რთული და მრავალგანზომილებიანი მონაცემთა მოდელირება.

2. NoSQL სისტემები (არარელაციური მონაცემთა ბაზები). NoSQL მონაცემთა ბაზები არის არარელაციური სისტემები, რომლებიც განკუთვნილია არასტრუქტურირებული ან ნახევრად სტრუქტურირებული მონაცემების ორგანიზებისთვის. ისინი ჩვეულებრივ არ მოითხოვენ წინასწარ განსაზღვრულ სქემას და გთავაზობენ უფრო მოქნილობისა და მასშტაბირებადობის შესაძლებლობებს.

NoSQL-ის ძირითადი მახასიათებლებია: ა) მოქნილი სქემა - NoSQL მონაცემთა ბაზებს არ სჭირდებათ მკაცრად განსაზღვრული სქემა, რაც ნიშნავს, რომ მონაცემთა სტრუქტურა შეიძლება შეიცვალოს დინამიკურად და მონაცემები შეიძლება ჰქონდეთ განსხვავებული ველები ერთსა და იმავე კოლექციაში ან ცხრილში. ბ) მონაცემთა მოდელები - NoSQL მონაცემთა ბაზებს აქვს სხვადასხვა ტიპის მონაცემთა მოდელები, რომლებიც აძლიერებენ მათ სხვადასხვა გამოყენების შემთხვევაში.

არარელაციური მონაცემთა ბაზების ყველაზე გავრცელებული მოდელებია:

- **დოკუმენტზე ორიენტირებული მონაცემთა ბაზები** (მაგ., MongoDB) - მონაცემები ინახება JSON/ BSON ფორმატში.
- **გრაფის მონაცემთა ბაზები** (მაგ., Neo4j) - განკუთვნილია მონაცემების კავშირებისა და ურთიერთობების მოდელირებისთვის.
- **გასაღების/მნიშვნელობის საცავები** (Key-Value Stores მაგ., Redis) - მონაცემები ინახება გასაღების (key) და მნიშვნელობის (value) წყვილების სახით.
- **სვეტებზე ორიენტირებული მონაცემთა ბაზები** (Column-family databases მაგ., Cassandra) - მონაცემები ინახება სვეტების სახით, რაც ამარტივებს მთლიან შენახვას.

მასშტაბირება. NoSQL მონაცემთა ბაზები ძირითადად ჰორიზონტალურად მასშტაბირებადი სისტემებია. ეს ნიშნავს, რომ შესაძლებელია მათი გაფართოება დამატებითი სერვერების დამატებით, რითაც იზრდება სისტემის გამტარიანობა. NoSQL მონაცემთა ბაზის მაგალითებია: MongoDB (დოკუმენტზე ორიენტირებული), Cassandra (სვეტებზე ორიენტირებული), Redis (გასაღების/მნიშვნელობის მაღაზია), Neo4j (გრაფის მონაცემთა ბაზა).

NoSQL-ის გამოყენების სფეროებია: ა) სოციალური ქსელების ან IoT აპლიკაციები, სადაც მონაცემთა მოდელები ხშირად იცვლება. ბ) მასიური მონაცემების ანალიზი (Big Data), სადაც საჭიროა სწრაფი წვდომა და რეალურ დროში მონაცემების დამუშავება. გ) სესიის და ქეშის მართვის სისტემები, სადაც საჭიროა სწრაფი წვდომა და დამუშავება.

დასკვნა:

SQL მონაცემთა ბაზები იდეალურია, როდესაც საჭიროა მკაცრად განსაზღვრული სქემა, კომპლექსური ურთიერთობები და მონაცემთა კონსისტენტურობა.

NoSQL მონაცემთა ბაზები უფრო მოქნილია და გამოდგება მაშინ, როცა სქემა ხშირად იცვლება ან მონაცემთა მოცულობა არის უზარმაზარი და საჭიროა ჰორიზონტალური მასშტაბირება.

თქვენი პროექტის სპეციფიკისა და მოთხოვნების მიხედვით, შეგიძლიათ აირჩიოთ SQL ან NoSQL ან ორივე მოდელის კომბინაცია, რაც ჰიბრიდულ გადაწყვეტას წარმოქმნის.

პარალელური დაპროგრამების პარადიგმა.

ლაიბნიცის მწკრივს აქვს შემდეგი სახე: $1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 \dots = \pi/4$

ამ წკრივის ერთ-ერთი თვისებაა ძალიან ნელი კრებადობა, რაც განაპირობებს გამოთვლების გაპარალელების დემონსტრირების მოხერხებულობას. მწკრივის ჯამის პირველი n წევრის გამოთვლის მიმდევრობითი პროგრამა ძალიან მარტივია:

```
#define NUM_ITERATIONS 1000000

int main(int argc, char *argv[ ] )
{
    double sum = 0.0;

    for (int i = 0; i < NUM_ITERATIONS; i++)
        sum += ((i & 1) ? - 1.0 : 1.0) / ((i << 1) | 1);

    std::cout << std::setprecision(20) << sum * 4.0 << std::endl;

    return 0; }
```

აქ გვაქვს ერთი ციკლი სადაც მწკრივის ჯამის მნიშვნელობა მიმდევრობით ივსება ყოველი შემდეგი მისი წევრით. ბოლოში მიღებული ჯამი მრავლდება 4-ზე π -ს მისაღებად. ცხადია მწკრივის თითოეული წევრის გამოსათვლელად არ გამოიყენება რეკურენტული დამოკიდებულება და თითოეული იტერაცია არის ერთმანეთისგან დამოუკიდებელი, რაც შესაძლებლობას გვაძლევს თავისუფლად შევასრულოთ იტერაციის გაპარალელება.

ვთქვათ, გვაქვს N ნაკადი და n -მწკრივის ელემენტების რაოდენობა (N -ის ჯერადი). აქ ნაკადებზე გაპარალელება უნდა მოხდეს ერთნაირი ინტერვალებით n/N იტერაციებით და შემდეგ ნაკადებზე მიღებული ჯამების შეკრებით.

პარალელური გამოთვლები მოიცავს საკითხებს, რომელიც ეხება ამოცანის ამოხსნის პროცესში პარალელიზმის რესურსების შექმნას და მის ეფექტურ რეალიზაციას. ნებისმიერი პარალელიზმის მიზანს წარმოადგენს კომპიუტერის მუშაობის ეფექტურობის გაზრდა სხვადასხვა დონეებზე: ამოცანის, მონაცემების, ალგორითმების, ინსტრუქციების და ბიტების.

პარალელობა ამოცანის დონეზე (მრავალნაკადიანობა). პარალელიზმის ეს სახე ეკუთვნის პარალელური პროგრამირების პარადიგმას და გულისხმობს ამოცანის ქვეამოცანებად დაყოფას. ქვეამოცანების რეალიზება ხდება ერთდროულად.

მრავალნაკადიანობა გულისხმობს: ა) პროგრამაში ისეთი გამოთვლების აღმოჩენას, რომელთა რეალიზაცია შეიძლება განხორციელდეს პარალელურად; ბ) მონაცემების შემდგომ განაწილებას პროცესორის ლოკალური მეხსიერების მოდულებში; გ) გამოთვლების პარალელიზმის შემდგომ შეთანხმებასა და მონაცემთა განაწილებას.

პარალელიზმი მონაცემების დონეზე. პარალელიზმის მოცემული სახის რეალიზება ხდება კომპილატორით და მდგომარეობს მონაცემთა სიმრავლეზე ერთნაირი ტიპის ოპერაციების სიმრავლის შეცვლაში ერთი ოპერაციით. დამუშავება ხორციელდება ვექტორულ პროცესორებზე.

მაგალითად, XX საუკუნის 50-ან წლებში აკადემიკოსი ა.ა. სამარსკი დაკავებული იყო ბირთვული აფეთქების მოდელირებით. გამოთვლების განსახორციელებლად ის იყენებდა ერთდროულად რამოდენიმე არითმომეტრს, თითოეულ მათგანზე მუშაობდა ცალკე ოპერატორი. გამოთვლების ასეთი ორგანიზება წარმოადგენდა პარალელური გამოთვლების წინამორბედს და გამოთვლილი იქნა აფეთქების ტალღის ევოლუცია.

პარალელიზმი ალგორითმების დონეზე. პარალელიზმის ეს სახე გულისხმობს ზოგიერთი გამოთვლების მიმდევრობითი ალგორითმების პარალელურით შეცვლას. ეს ეხება ძებნის, დახარისხების და ა.შ. სხვა ალგორითმებს.

გაპარელების პროცესი ხორციელდება პარალელური პროგრამირების საშუალებების გამოყენებით, ისეთების როგორიცაა სპეციალური ბიბლიოთეკები, გარე ცვლადები, კომპილატორის დირექტივები და ა.შ. ამის მაგალითია OpenMP-ს ტექნოლოგია.

პარალელიზმი ინსტრუქციების (ბრძანებების) დონეზე. ინსტრუქციების ქვეშ იგულისხმება პროცესორის მიერ შესრულებული ბრძანებები. აქ გამოყოფილია პარალელიზმი აპარატურულ და პროგრამულ დონეზე.

ბრძანებების დონეზე პარალელიზმი პარალელურ პროგრამირებაზე უარის თქმის შესაძლებლობას იძლევა. ბრძანებების დონეზე პროგრამული პარალელიზმი ხორციელდება კომპილატორის მიერ. კომპილატორი ახდენს შესასრულებელი კოდის ფორმირებას უშუალოდ კონკრეტული პროცესორისათვის.

ჯერ კიდევ გამოიყენება 1966 წლის მაიკ ფლინის კლასიფიკაცია, რომელიც ეხება კომპიუტერების არქიტექტურას:

1. SISD - არქიტექტურა ბრძანებების ერთი ნაკადით და მონაცემთა ერთი ნაკადით;
2. SIMD - არქიტექტურა ბრძანებების ერთი ნაკადით და მონაცემთა მრავალი ნაკადით;

3. MISD - არქიტექტურა ბრძანებების მრავალი ნაკადით და მონაცემთა ერთი ნაკადით;

4. MIMD - არქიტექტურა ბრძანებების მრავალი ნაკადით და მონაცემთა მრავალი ნაკადით.

პარალელიზმი ბიტების დონეზე. პარალელიზმი ბიტების დონეზე გულისხმობს პარალელური გამოთვლების გამოყენებას ოპერაციების ბიტური დონეზე. ბიტების დონეზე პარალელიზმი ამცირებს დროის ხარჯს მათემატიკურ და ლოგიკურ ოპერაციებზე. ასეთ მიდგომაში გამოიყენება რამდენიმე პრინციპი და ტექნიკა:

1. ბიტური ოპერაციების პარალელიზმი. ბიტური ოპერაციები, მაგალითად, AND, OR, XOR, SHIFT და NOT, ჩვეულებრივ, ძალიან სწრაფად სრულდება, რადგან თითოეული ბიტი დამოუკიდებლად შეიძლება დამუშავდეს. ამის გამოყენება შესაძლებელია პარალელურ გამოთვლებში, მაგალითად, ბიტების მასიური რაოდენობის შემთხვევაში ერთდროულად ოპერაციის ჩასატარებლად.

2. SIMD (Single Instruction, Multiple Data). SIMD არქიტექტურა ოპერაციას ერთდროულად რამდენიმე მონაცემზე ასრულებს, რაც საშუალებას იძლევა, რომ ერთმა ინსტრუქციამ ერთდროულად ბევრ ბიტზე იმოქმედოს. მაგალითად, შესაძლებელია ერთი ინსტრუქციით 64 ბიტის (ან მეტი) დამუშავება, თუ თითოეულ ბიტზე მონაცემი დამოუკიდებელია.

3. პარალელიზმი კრიპტოგრაფიაში. კრიპტოგრაფიული ალგორითმები ხშირად იყენებენ ბიტების დონეზე პარალელიზმს. მაგალითად, AES შიფრაციაში ბიტური ოპერაციების პაკეტური გამოყენებით შესაძლებელია ერთდროულად რამდენიმე ბლოკის დაშიფრვა. SHA-256 ჰეშირების ალგორითმშიც ბიტური ოპერაციები პარალელურად სრულდება, რაც აჩქარებს გამოთვლას.

4. ბიტმალტიპლიკაციური ოპერაციები. ბიტურ დონეზე მარტივი ალგორითმები ხშირად იყენებენ ბიტმალტიპლიკაციურ ოპერაციებს (მაგ., გრძელი რიცხვების გამრავლება ბიტების მიხედვით). აქ შეიძლება პარალელური გამოთვლების გამოყენება, სადაც ყოველი ბიტური გამრავლება და გადატანა დამოუკიდებლად ხდება.

ბიტმალტიპლიკაციური ოპერაციები მოიცავს ისეთ გამოთვლით მეთოდებს, რომლებიც ბიტების დონეზე ხდება და იყენებს ოპერაციებს ბიტების დონეზე (მაგალითად, AND, OR, XOR). ბიტმალტიპლიკაცია ფართოდ გამოიყენება მაღალი ეფექტურობის და სისწრაფის მისაღწევად გამრავლების, ბიტური ოპერაციების ოპტიმიზაციისა და დიდი რიცხვების გამრავლების ალგორითმებში.

5. ბიტური პარალელიზმი მანქანურ სწავლებაში. ბიტური პარალელიზმი ასევე გამოიყენება მანქანურ სწავლებაში, განსაკუთრებით ბინარულ მონაცემთა მოდელებში. მაგალითად, ბინარული ნეირონული ქსელები (Binary Neural Networks) იყენებენ ბიტურ ოპერაციებს მატრიცის გამრავლების დასაჩქარებლად, რაც ნაკლები ენერგიის გამოყენებას და უფრო სწრაფ შესრულებას უზრუნველყოფს.

6. ბიტური მონაცემთა დამუშავება GPU-ზე. გრაფიკული პროცესორები (GPU) ხშირად იყენებენ ბიტურ დონეზე პარალელიზმს, განსაკუთრებით მაშინ, როცა საჭიროა ძალიან დიდი რაოდენობის ბიტების ერთდროულად დამუშავება. მაგალითად, გამოსახულების ფილტრაცია, კონვოლუცია (მათემატიკური ოპერაციის ტიპი, რომელიც ჩვეულებრივ გამოიყენება სიგნალის, გამოსახულების დამუშავებასა და ნეირონულ ქსელებში) და სხვა ოპერაციები შეიძლება ჩატარდეს ბიტების დონეზე, სადაც თითოეული ბიტი GPU-ს მძლავრ გამოთვლით საშუალებას იყენებს.

პარალელური ალგორითმების მუშაობა იდეალურ პარალელურ კომპიუტერზე.

განვიხილოთ პარალელური ალგორითმების მუშაობა იდეალურ პარალელურ კომპიუტერზე, რომელიც შედგება პროცესორების სიმრავლისა და საზიარო მეხსიერებისაგან, რომელიც მიმდევრობით მდგრადია (sequentially consistent). მიმდევრობით მდგრადობაში იგულისხმება, რომ საზიარო მეხსიერება, რომელიც სინამდვილეში ასრულებს არა ერთ ოპერაციას პარალელურად, იძლევა იმავე შედეგს, რასაც მივიღებდით, თუ ყოველ ჯერზე მხოლოდ ერთი ბრძანება შესრულდებოდა. იდეალურ პარალელურ კომპიუტერში ყველა პროცესორს ერთნაირი სიმძლავრე აქვს და უგულებელყოფილია დამგეგმავის მუშაობის დრო, რასაც პრაქტიკაში მართლაც მცირე ღირებულება აქვს თუ ალგორითმი არის საკმარისად პარალელური.

შესრულების დროის გამოთვლა. პარალელური ალგორითმების ასიმპტოდური შეფასებისთვის იდეალურ პარალელურ კომპიუტერზე გამოვიყენებთ ორ საზომს:

1. დრო, რომელიც საჭიროა ამ ალგორითმის ერთ პროცესორზე შესასრულებლად. აღინიშნება T_1 -ით.
2. დრო, რომელიც საჭიროა ალგორითმის შესასრულებლად პროცესორების შეუზღუდავი რაოდენობით არსებობისას. აღინიშნება T_∞ -ით.

რეალური მუშაობის დრო დამოკიდებულია არა მარტო ამ ორ საზომზე, არამედ იმაზეც, თუ რამდენი პროცესორი გვაქვს და როგორ მუშაობს დამგეგმავი. დროს, რომელიც საჭიროა ალგორითმის P პროცესორზე შესასრულებლად აღვნიშნავთ T_p -თი.

T_1 და T_∞ გვაძლევს შემდეგ ქვედა ზღვრებს T_p -სთვის: P პროცესორიან იდეალურ პარალელურ კომპიუტერს ერთ ბიჯზე არაუმეტეს P ერთეული სამუშაოს შესრულება შეუძლია. შესაბამისად, T_p დროში - არა უმეტეს $P * T_p$ ერთეულის. რადგანაც სულ T_1 ერთეული სამუშაო შესასრულებელი, გვაქვს შემდეგი: $P * T_p \geq T_1$ (1)

$$\text{უტოლობის ორივე მხარის } P\text{-ზე გაყოფის შედეგად მივიღებთ: } T_p \geq T_1/P \quad (2)$$

P პროცესორიანი იდეალური პარალელური კომპიუტერი ვერ იმუშავებს იმ მანქანაზე სწრაფად, რომელსაც პროცესორები შეუზღუდავი რაოდენობით აქვს: $T_p \geq T_\infty$ (3)

ჩვენ განვსაზღვრავთ გამოთვლის P პროცესორზე შესრულების აჩქარებას (speedup), როგორც შეფარდებას $T1/P * T_p$, რომელიც გვიჩვენებს, თუ რამდენჯერ სწრაფად სრულდება გამოთვლები P რაოდენობის პროცესორზე ერთ პროცესორთან შედარებით. (1)-დან გამომდინარე, $T1/T_p \leq P$ ანუ აჩქარება P პროცესორის გამოყენებისას შესაძლებელია იყოს მაქსიმუმ P -ს ტოლი. როდესაც აჩქარება არის წრფივი პროცესორების რაოდენობის მიმართ, ანუ როდესაც $T1/T_p = \Theta(P)$, ვიტყვით, რომ გვაქვს წრფივი აჩქარება (linear speedup), ხოლო როცა $T1/T_p = P$ - იდეალური წრფივი აჩქარება (perfect linear speedup).

$T1$ -ის შეფარდება T_{∞} -სთან გვაძლევს multithreaded გამოთვლების პარალელურობას (parallelism). როგორც შეფარდება, პარალელურობა განსაზღვრავს სამუშაოს საშუალო რაოდენობას, რომელიც შეიძლება შესრულდეს პარალელურად პარალელური ალგორითმის მუშაობის ყოველ საფეხურზე. როგორც ზედა ზღვარი, პარალელურობა განსაზღვრავს მაქსიმალურ შესაძლო აჩქარებას, რასაც შეიძლება მივაღწიოთ პროცესორების შეუზღუდავი რაოდენობით არსებობის შემთხვევაში.

საბოლოოდ, პარალელურობა შემოსაზღვრავს იდეალური წრფივი აჩქარების მიღწევის შესაძლებლობას. უფრო კონკრეტულად, თუ პროცესორების რიცხვი მეტია პარალელურობის მნიშვნელობაზე, გამოთვლით პროცესს არ შეუძლია მიაღწიოს იდეალურ წრფივ აჩქარებას. ამის დასაწახად დავუშვათ, რომ $P > T1 / T_{\infty}$, რა შემთხვევაშიც (2)-დან გამომდინარეობს, რომ აჩქარება აკმაყოფილებს შემდეგ პირობას:

$$T1/T_p \leq T1/T_{\infty} < P \quad (4)$$

უფრო მეტიც - თუ იდეალურ პარალელურ კომპიუტერში პროცესორების რაოდენობა P ბევრად აღემატება პარალელურობის მნიშვნელობას ანუ თუ $P \gg T1 / T_{\infty}$, მაშინ $T1 / T_p \ll P$, ანუ აჩქარება პროცესორების რაოდენობას ბევრად ჩამოუვარდება. სხვა სიტყვებით რომ ვთქვათ, რაც უფრო მეტად გადააჭარბებს პროცესორების რაოდენობა პარალელურობის მნიშვნელობას, მით ნაკლებ იდეალურ აჩქარებას მივიღებთ.

მრავალ ნაკადიანი გამოთვლის პარალელურ სიჭარბეს (slackness) იდეალურ პარალელურ კომპიუტერზე P პროცესორით განვსაზღვრავთ, როგორც შეფარდებას

$$(T1 / T_{\infty}) / P = T1 / (T_{\infty} * P) \quad (5)$$

პარალელური სიჭარბე გვიჩვენებს, თუ რამდენად აღემატება პარალელურობა პროცესორების რაოდენობას. თუ სიჭარბე ნაკლებია ერთზე, არ შეგვიძლია მივაღწიოთ იდეალურ წრფივ აჩქარებას, რადგან რაც მეტად მცირდება სიჭარბე 1-დან 0-მდე ალგორითმის აჩქარება შორდება იდეალურ მაჩვენებელს. თუმცა, თუ სიჭარბე მეტია 1-ზე, შემზღუდველი ფაქტორი არის ის, თუ რა სამუშაოს შესრულება შეუძლია ერთ პროცესორს დროის ერთეულში. რაც უფრო მეტად აღემატება სიჭარბე ერთს, მით უფრო ახლოსაა აჩქარება იდეალურ წრფივთან კარგი დამგეგმვის არსებობის შემთხვევაში.

დამგეგმვა (Scheduling). შესრულების მაღალი ხარისხის მისაღწევად დიდი მნიშვნელობა აქვს concurrency platform-ის დამგეგმვას. პრაქტიკულად, დამგეგმავი ანაწილებს გამოთვლებს სტატიკურ ნაკადებში (thread-ებში) და ოპერაციული სისტემა ანაწილებს ნაკადებს (thread-ებს) პროცესორებზე. რადგანაც ეს ბოლო ფენა არაა ჩვენთვის მნიშვნელოვანი, ვთვლით, რომ გადანაწილების პროცესი დამოკიდებულია მთლიანად დამგეგმავზე.

თეორემა 1. იდეალურ პარალელურ კომპიუტერზე P პროცესორით ხარბი დამგეგმავი ასრულებს მრავალნაკადიან (multithreaded) გამოთვლას შემდეგ დროში:

$$T_p \leq T_1 / P + T_\infty \quad (6)$$

შედეგი 1. ხარბი დამგეგმავის მიერ შესრულებული მრავალნაკადიანი ალგორითმის მუშაობის დრო $T_p * P$ პროცესორიან იდეალურ კომპიუტერზე ოპტიმალურ დროსთან (T'_p) შემდეგ დამოკიდებულებაშია: $T_p \leq T * p$

შედეგი 2. ზემოთ მოცემული თეორემების პირობებში, თუ $P \ll T_1/T_\infty$, მაშინ $T_p \approx T_1/P$, ანუ აჩქარება არის დაახლოებით P .

შეჯიბრის მდგომარეობა. შეჯიბრის მდგომარეობა პარალელურობის ყველაზე დიდი პრობლემაა. შეჯიბრის მდგომარეობით გამოწვეული შეცდომების აღმოჩენა ძალიან რთულია.

შეჯიბრის მდგომარეობა წარმოიქმნება, როცა ორ ლოგიკურად პარალელურ ინსტრუქციას წვდომა აქვს ერთი და იგივე მეხსიერებაზე და ერთი ბრძანება მაინც ცვლის მონაცემებს.

შეჯიბრის მდგომარეობასთან გამკლავების არა ერთი მეთოდი არსებობს, მათ შორის mutex-ები და სინქრონიზაცია. ჩვენი მიზნებიდან გამომდინარე, აჯობებს დავრწმუნდეთ, რომ ყველა გამოთვლა, რომელიც სრულდება პარალელურად, არის დამოუკიდებელი, და, შესაბამისად, არ გვაქვს შეჯიბრის მდგომარეობა.

პარალელურ for ციკლში ყველა იტერაცია უნდა იყოს დამოუკიდებელი. ასევე, ჩადგმული პარალელიზმის გამოყენებისას, მშობლისა და შვილების მიერ შესასრულებელი კოდი უნდა იყოს დამოუკიდებელი ერთმანეთისაგან.

აღსანიშნავია, რომ ალგორითმმა, რომელიც შეჯიბრის მდგომარეობას შეიცავს, შეიძლება სწორი შედეგიც დააბრუნოს. მაგალითად, იმ შემთხვევაში, თუ ორი ნაკადი (thread) საზიარო მეხსიერებაში ერთსა და იმავე მნიშვნელობას წერს.

$T_1/T_p == \Theta(P)$, ვიტყვი, რომ გვაქვს წრფივი აჩქარება (linear speedup), ხოლო როცა $T_1/T_p = P$ - იდეალური წრფივი აჩქარება (perfect linear speedup).

მატრიცების გადამრავლება. განვიხილოთ მატრიცების გადამრავლების ალგორითმი, რომელიც ეფუძნება ციკლების გაპარალელებას შემდეგ ალგორითმში (იხ. ნახ. 14.8.):

SQUARE-MATRIX-MULTIPLY(A,B)

n = A.rows

let C be a new n*n MATRIX

for i = 1 to n

for j = 1 to n

c_{ij} = 0

for k = 1 to n

c_{ij} = c_{ij} + a_{ik}*b_{kj}

return C

სერიული for ციკლების პარალელური ვერსიებით ჩანაცვლებით მივიღებთ შემდეგ ალგორითმს (იხ. ნახ. 14.9.):

P-SQUARE-MATRIX-MULTIPLY(A,B)

n = A.rows

let C be a new n*n MATRIX

parallel for i = 1 to n

parallel for j = 1 to n

c_{ij} = 0

for k = 1 to n

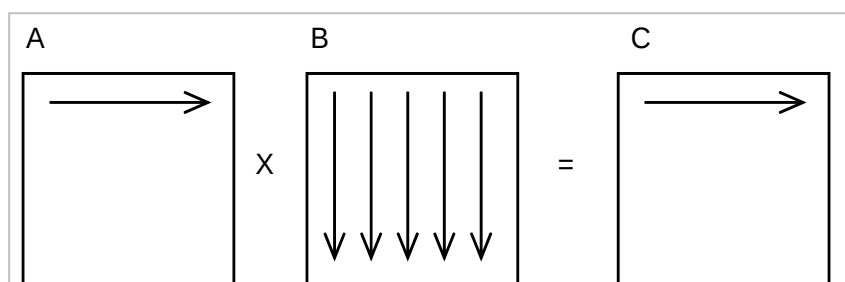
c_{ij} = c_{ij} + a_{ik}*b_{kj}

return C

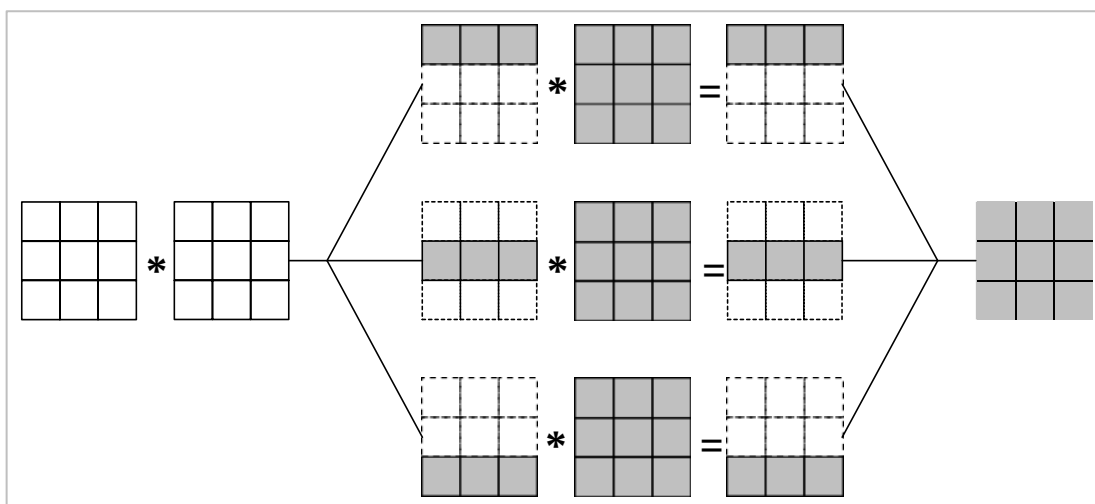
შევნიშნოთ, რომ მესამე for ციკლი არ იცვლება პარალელური ვარიანტით. მისი გაპარალელება გამოიწვევდა შეჯიბრის მდგომარეობას და ალგორითმი არასწორ შედეგს დააბრუნებდა.

რადგანაც მოცემული ალგორითმის სერიალიზაციით ვიღებთ ჩვეულებრივ მატრიცების გადამრავლების ალგორითმს, რომლის მუშაობის დროა $\Theta(n^3)$, იდეალური აჩქარება იქნება $\Theta(n^3)$, ხოლო span $\Theta(n)$.

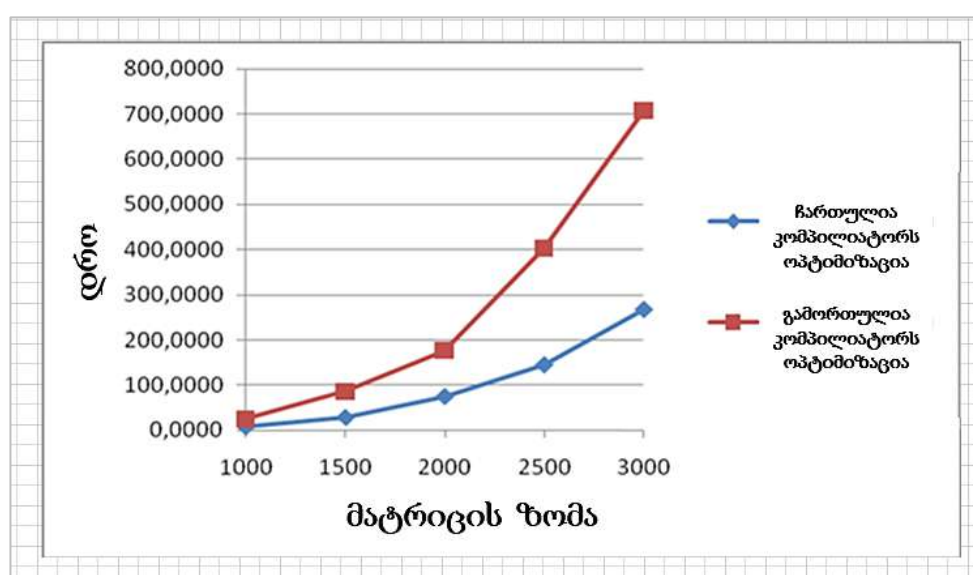
მივიღებთ პარალელურობის მნიშვნელობას, რომელიც უდრის $\Theta(n^3)/\Theta(n) = \Theta(n^2)$ (იხ. ნახ. 14.10., 14.11.).



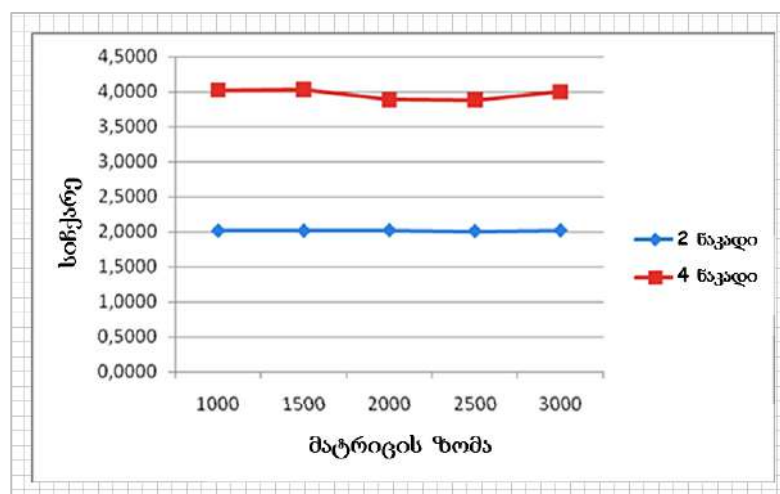
ნახ. 14.8. ორი მატრიცის მიმდევრობითი გადამრავლების ზოგადი სქემა



ნახ. 14.9. ორი მატრიცის პარალელური გადამრავლების ზოგადი სქემა

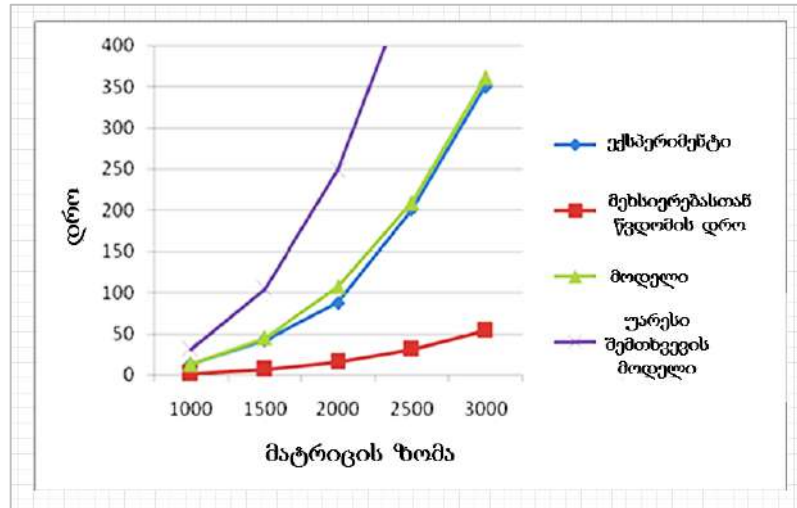


ნახ. 14.10. შედარებითი დიაგრამა. მატრიცების გადამრავლების შესრულების დროის დამოკიდებულება მატრიცის ზომებთან კომპილატორის ჩართვისას და გამორთვისას

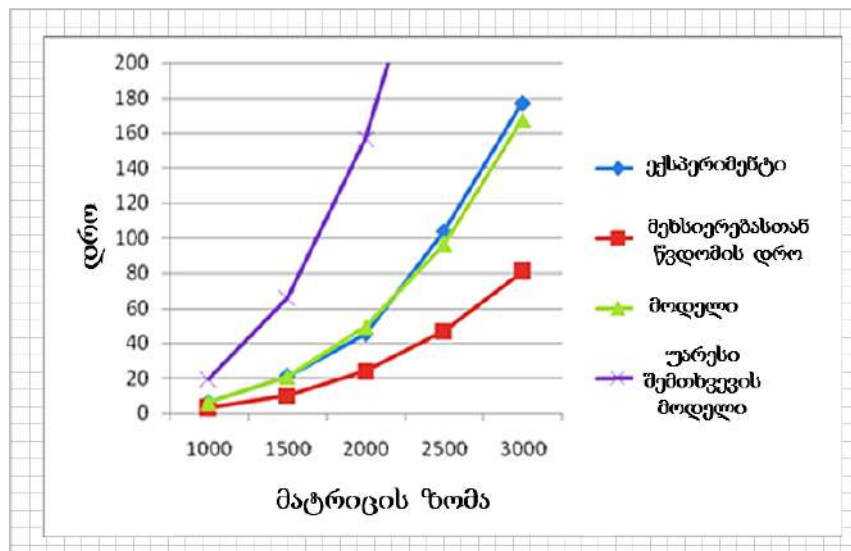


ნახ. 14.11. მატრიცის ზომისა სიჩქარის დამოკიდებულება ორი და ოთხი ნაკადისას

დიაგრამები 2 ნაკადისთვის (ნახ.14.12., 14.13.)



ნახ. 14.12. შედარებითი დიაგრამა მატრიცის ზომასა და დროს შორის ორი ნაკადისას



ნახ. 14.13. შედარებითი დიაგრამა მატრიცის ზომასა და დროს შორის ორი ნაკადისას

დალაგება შერწყმით (merge sort). შერწყმით სორტირების ალგორითმი იყენებს „დაყავი და იბატონე“ პარადიგმას, რის გამოც მისი გაპარალელლება ჩადგმული პარალელულობის გამოყენებით ბუნებრივია. პარალელური merge sort ალგორითმის ფსევდოკოდი, რომელიც ალაგებს A მასივის $[p..r]$ ქვემასივს ($p \leq r$), შემდეგია:

MERGE-SORT'(A,p,r)

if $p < r$

$q = \lceil (p+r)/2 \rceil$

spawn MERGE-SORT'(A,p,q)

MERGE-SORT'(A,q + 1,r)

sync

MERGE(A,p,q,r)

MERGE პროცედურის ფსევდოკოდი შემდეგია:

MERGE(A, p, q, r)

$n1 = q - p + 1$

$n2 = r - q$

Let $L[1..n1 + 1]$ და $R[1..n2 + 1]$ by new arrays

for $i = 1$ to $n1$

$L[i] = A[p + i - 1]$

for $j = 1$ to $n2$

$R[j] = A[q + j]$

$L[n1 + 1] = \infty$

$R[n2 + 1] = \infty$

$i = 1$

$j = 1$

for $k = p$ to r

if $L[i] \leq R[j]$

$A[k] = L[i]$

$i = i + 1$

else

$A[k] = R[j]$

$j = j + 1$

MERGE-SORT'-ის მუშაობის დროის ანალიზისთვის გავიხსენოთ, რომ MERGE პროცედურას n ელემენტის შერწყმისთვის $\theta(n)$ დრო სჭირდება.

რადგანაც MERGE სერიულია, მისი $T1$ და $T\infty$ $\theta(n)$ -ის ტოლია.

$MS1'(n)$ -ით აღვნიშნოთ MERGE-SORT'-ის $T1(n)$, რომელიც შემდეგი რეკურენტული ფორმულით გამოითვლება:

$$MS1'(n) = 2 * MS1'(n/2) + \theta(n) = \theta(n * \lg n) \quad (7)$$

რაც ემთხვევა სერიული ალგორითმის მუშაობის დროს. რადგანაც MERGE-SORT'-ის ორი რეკურსიული გამოძახება შეიძლება ერთმანეთის პარალელურად შესრულდეს, $T\infty$, რომელსაც $MS\infty'(n)$ -ით აღვნიშნავთ, შემდეგი რეკურსიული ფორმულით გამოითვლება:

$$MS\infty'(n) = MS\infty'(n/2) + \theta(n) = \theta(n) \quad (8)$$

გამოდის, რომ MERGE-SORT'-ის პარალელურობის მნიშვნელობაა

$MS1'(n) / MS\infty'(n) = \theta(\lg n)$, რაც არ არის დიდად შთამბეჭდავი მაჩვენებელი.

მაგალითად, 10 მილიონი ელემენტის დასალაგებლად ამ ალგორითმმა შეიძლება მიაღწიოს წრფივ აჩქარებას რამდენიმე პროცესორზე, მაგრამ ასეულობით პროცესორზე მისი მუშაობა არ იქნება ეფექტური.

ადვილი მისახვედრია, რომ ამ ალგორითმის სუსტი მხარე სერიული MERGE პროცედურაა. არსებობს MERGE პროცედურის ვარიანტი, რომელიც იყენებს „გაყავი და იბატონე“ პარადიგმას და რომლის გაპარალელება ადვილად შეიძლება ჩადგმული პარალელიზმის გამოყენებით.

მრავალნაკადიანი MERGE ალგორითმი T მასივის ქვემასივებზე ოპერირებს. წარმოვიდგინოთ, რომ T მასივის ორი დალაგებული ქვემასივი: $T[p1..r1]$, რომლის სიგრძეა $n1 = r1 - p1 + 1$ და $T[p2..r2]$, რომლის სიგრძეა $n2 = r2 - p2 + 1$, უნდა გავაერთიანოთ $A[p3..r3]$ ქვემასივში, რომლის სიგრძეა $n3 = r3 - p3 + 1 = n1 + n2$. სიმარტივისათვის, ზოგადობის შეუზღუდავად დავუშვათ, რომ $n1 \geq n2$.

თავდაპირველად, ვპოულობთ $T[p1..r1]$ ქვემასივის შუა ელემენტს $x = T[q1]$, სადაც $q1 = \lceil (p1 + r1)/2 \rceil$. რადგანაც ქვემასივი დალაგებულია, x არის $T[p1..r1]$ -ის მედიანა: ყოველი ელემენტი $T[p1..q1 - 1]$ -ში არ აღემატება x -ს და ყოველი ელემენტი $T[q1 + 1..r1]$ -ში არის x -ზე მეტი ან მისი ტოლი. შემდეგ, ორობითი ძებნის გამოყენებით, ვპოულობთ $T[p2..r2]$ დალაგებულ ქვემასივში ისეთ $q2$ ინდექსს, რომ x -ის $T[q2 - 1]$ -სა და $T[q2]$ -ს შორის ჩასმის შემდეგ $T[p2..r2]$ ისევ დალაგებული იქნება. ამის შემდეგ, საწყის $T[p1..r1]$ და $T[p2..r2]$ ქვემასივებს ვაერთიანებთ $A[p3..r3]$ -ში შემდეგნაირად:

1. $q3 = p3 + (q1 - p1) + (q2 - p2)$.
2. ჩავწერთ x -ის მნიშვნელობა $A[q3]$ -ში.
3. რეკურსიულად გავაერთიანოთ $T[p1..q1 - 1]$ და $T[p2..q2 - 1]$ და შედეგი ჩავწერთ $A[p3..q3 - 1]$ -ში.
4. რეკურსიულად გავაერთიანოთ $T[q1 + 1..r1]$ და $T[q2..r2]$ და შედეგი ჩავწერთ $A[q3 + 1..r3]$ -ში.

$q3$ -ის გამოთვლისას, $q1 - p1$ არის $T[p1..q1 - 1]$ ქვემასივის ელემენტების რაოდენობა, ხოლო $q2 - p2$ არის $T[p2..q2 - 1]$ ქვემასივის ელემენტების რაოდენობა. მათი ჯამი არის $A[p3..r3]$ ქვემასივის იმ ელემენტების რაოდენობა, რომლებიც ქვემასივში x -მდე უნდა იყოს განთავსებული.

საბაზისო შემთხვევა ხდება, როცა $n1 = n2 = 0$.

ამდროს არანაირი სამუშაო არ არის ჩასატარებელი ორი ცარიელი ქვემასივის გასაერთიანებლად. რადგანაც დავუშვით, რომ ქვემასივი $T[p1..r1]$ არის სულ მცირე იმავე სიგრძის, რა სიგრძისაც $T[p2..r2]$, ანუ $n1 \geq n2$, იმის შესამოწმებლად, გვაქვს თუ არა

საბაზისო შემთხვევა, შეგვიძლია შევამოწმოთ სრულდება თუ არა ტოლობა $n1 = 0$. ასევე, უნდა დავრწმუნდეთ, რომ ალგორითმი სწორად მუშაობს იმ შემთხვევაში, როდესაც $T[p2..r2]$ ცარიელია.

ზემოთ აღწერილი იდეები გადავიტანოთ ფსევდოკოდში. დავიწყოთ ორობითი ძებნით, რომელიც $T[p..r]$ დალაგებულ ქვემასივში ეძებს იმ ინდექსს, რომელზეც შეგვიძლია ჩავწეროთ x ელემენტი ისე, რომ ქვემასივი ისევ დალაგებული იყოს:

BINARY-SEARCH(x, T, p, r)

low = p

high = $\max(p, r + 1)$

while low < high

mid = $\lfloor (\text{low} + \text{high}) / 2 \rfloor$

if $x \leq T[\text{mid}]$

high = mid

else

low = mid + 1

return high

ორობითი ძებნის ზედა ზღვარია $O(\lg n)$, სადაც $n = r - p + 1$ არის იმ მასივის სიგრძე, რომელზეც ის მუშაობს, რადგანაც BINARY-SEARCH სერიული ალგორითმია, $T1 = T\infty = \theta(\lg n)$.

მოვიყვანოთ მრავალნაკადიანი შერწყმის ალგორითმის ფსევდოკოდი.

$P - \text{MERGE}(T, p1, r1, p2, r2, A, p3)$ აერთიანებს ორ დალაგებულ ქვემასივს: $T[p1..r1]$ -სადაც $T[p2..r2]$ -ს $A[p3..r3]$ -ში, სადაც $r3 = p3 + (r1 - p1 + 1) + (r2 - p2 + 1) - 1 = p3 + (r1 - p1) + (r2 - p2) + 1$.

1. $P\text{-MERGE}(T, p1, r1, p2, r2, A, p3)$

2. $n1 = r1 - p1 + 1$

3. $n2 = r2 - p2 + 1$

4. if $n1 < n2$ //check if $n1 \geq n2$

5. exchange $p1$ with $p2$

6. exchange $r1$ with $r2$

7. exchange $n1$ with $n2$

8. if $n1 == 0$ //both empty?

9. return

10. else $q1 = \lfloor (p1 + r1) / 2 \rfloor$

11. $q2 = \text{BINARY-SEARCH}(T[q1], T, p2, r2)$
12. $q3 = p3 + (q1 - p1) + (q2 - p2)$
13. $A[q3] = T[q1]$
14. $\text{spawn } P\text{-MERGE}(T, p1, q1-1, p2, q2-1, A, p3)$
15. $P\text{-MERGE}(T, q1+1, r1, q2, r2, A, q3+1)$
16. sync

P-MERGE პროცედურის მუშაობის დროის ანალიზი. პირველ რიგში, დავითვალოთ T -ის P-MERGE-სთვის, რომელსაც აღვნიშნავთ $PM(n)$ -ით, სადაც ორი ქვემასივის ელემენტების საერთო რაოდენობაა $n = n1 + n2$. რადგანაც პროცედურის 2 გამოძახება მე-13 და მე-14 ხაზებზე ლოგიკურად ერთმანეთის პარალელურად სრულდება, მხოლოდ იმის შეფასებაა საჭირო, რომელიც დროის მეტ რესურსს მოითხოვს.

მთავარი იმის გააზრებაა, რომ ყველაზე უარეს შემთხვევაში, ელემენტების მაქსიმალური რაოდენობა რეკურსიულ გამოძახებებს შორის შეიძლება იყოს მაქსიმუმ $3n / 4$, რასაც აქვე დავამტკიცებთ.

$$n2 \leq n1 \text{ უტოლობიდან გამომდინარეობს, რომ } n2 = 2 * n2/2 \leq (n1 + n2)/2 = n/2$$

უარეს შემთხვევაში, ერთ-ერთი რეკურსიული გამოძახება გააერთიანებს $T[p1..r1]$ ქვემასივის $[n1/2]$ ელემენტს $T[p2..r2]$ ქვემასივის $n2$ ელემენტთან და იმ ელემენტების საერთო რაოდენობა, რომლებიც გამოძახებაში იღებენ მონაწილეობას, არის:

$$[n1/2] + n2 \leq n1/2 + n2/2 + n2/2 = (n1 + n2)/2 + n2/2 \leq n/2 + n/4 = 3n/4$$

ორობითი ძებნის მუშაობის დროის გათვალისწინებით ვიღებთ შემდეგ რეკურენტულ ფორმულას:

$$PM(n) = PM(3n / 4) + \theta(\lg n)$$

$$PM(n) = \theta(\lg^2 n)$$

ახლა გამოვითვალოთ $T1$ P-MERGE-სთვის, რომელსაც $PM1(n)$ -ით აღვნიშნავთ და რომელიც უდრის $\theta(n)$ -ს. რადგანაც n ელემენტიდან თითოეული უნდა გადაიწეროს T მასივიდან A მასივში, გვაქვს $PM1(n) = \Omega(n)$. დასამტკიცებელი გვრჩება მხოლოდ ის, რომ $PM1(n) = O(n)$.

თავდაპირველად დავითვალოთ $PM1(n)$ უარესი შემთხვევისთვის. ორობითი ძებნა მოითხოვს $\theta(\lg n)$ დროს უარეს შემთხვევაში, რაც მეტია იმ დროზე, რასაც დანარჩენი კოდი მოიხმარს გარდა რეკურსიული გამოძახებებისა.

რაც შეეხება რეკურსიულ გამოძახებებს - მიუხედავად იმისა, რომ შესაძლებელია ორი გამოძახება სხვადასხვა რაოდენობის ელემენტების გასაერთიანებლად მუშაობდეს, ერთად ისინი არაუმეტეს n ელემენტზე მუშაობენ. უფრო მეტიც: როგორც T -ის ანალიზისას ვნახეთ, რეკურსიული გამოძახება მუშაობს არა უმეტეს $3n/4$ ელემენტზე.

ამ ყველაფრიდან გამომდინარე ვიღებთ რეკურსიას: $PM1(n) = PM1(\alpha n) + PM1((1 - \alpha)n) + O(\lg n)$, სადაც $1/4 \leq \alpha \leq 3/4$ და სადაც α -ს მნიშვნელობა შეიძლება განსხვავებული იყოს რეკურსიის ყოველ დონეზე.

იმას, რომ რეკურსიის ამონახსნია $PM1 = O(n)$, ვამტკიცებთ ჩანაცვლების მეთოდით. დავუშვათ, რომ $PM1(n) \leq c1n - c2\lg n$ რაიმე დადებითი $c1$ -სა და $c2$ -სთვის. ჩანაცვლების მეთოდით ვიღებთ:

$$\begin{aligned} PM1(N) &\leq (c1\alpha n - c2\lg(\alpha n)) + (c1(1 - \alpha)n - c2\lg((1 - \alpha)n)) + \theta(\lg n) = \\ &= c1(\alpha + (1 - \alpha))n - c2(\lg(\alpha n) + \lg((1 - \alpha)n)) + \theta(\lg n) = \\ &= c1n - c2(\lg \alpha + \lg n + \lg(1 - \alpha) + \lg n) + \theta(\lg n) = \\ &= c1n - c2\lg n - (c2(\lg n + \lg(\alpha(1 - \alpha))) - \theta(\lg n)) \leq c1n - c2\lg n, \end{aligned}$$

რადგან შეგვიძლია ავირჩიოთ საკმარისად დიდი $c2$ იმისათვის, რომ $c2(\lg n + \lg(\alpha(1 - \alpha)))$ აჭარბებს $\theta(\lg n)$ -ს. ასევე შეგვიძლია საკმარისად დიდი $c1$ იმისათვის, რომ რეკურსიის საბაზისო შემთხვევის პირობები დაკმაყოფილდეს. იქიდან გამომდინარე, რომ $PM1(n) = \Omega(n)$ და $PM1(n) = O(n)$, გვაქვს $PM1(n) = \theta(n)$

P-MERGE-ის პარალელურობის მნიშვნელობაა: $PM1(n) / PM\infty(n) = \theta(n / \lg^2 n)$

პარალელური MERGE-SORT. MERGE-SORT' ალგორითმისაგან განსხვავებით, მრავალნაკადიანი (multithreaded) ალგორითმი, რომელიც იყენებს P-MERGE ალგორითმს, არგუმენტად იღებს B მასივს, რომელშიც ალგორითმის მუშაობის შედეგი ჩაიწერება.

უფრო კონკრეტულად, $P - MERGE - SORT(A, p, r, B, s)$ ასორტირებს $A[p..r]$ -ში და ინახავს მათ $B[s..s + r - p]$ -ში.

P-MERGE-SORT(A,p,r,B,s)

n = r-p+1

if n==1 B[s] = A[p]

else T = new Array(n) //create a new array T of size n

q = [floor((p+r)/2)]

q' = q-p+1

spawn P-MERGE-SORT(A,p,q,T,1)

P-MERGE-SORT(A,q+1,r,T,q'+1)

sync

P-MERGE(T,1,q',q'+1,n,B,s)

n არის ელემენტების რაოდენობა $A[p..r]$ -ში.

საბაზისო შემთხვევა გვაქვს, როცა $n = 1$.

q არის ინდექსი, რომელიც $A[p..r]$ -ს ორ ნაწილად ყოფს.

q' არის ელემენტების რაოდენობა $A[p..q]$ -ში და გამოიყენება იმის განსასაზღვრად, თუ რა უნდა დაიწყოს ინდექსიდან $A[q + 1..r]$ -ის დალაგებული ელემენტების შენახვა T -ში.

პარალელური MERGE-SORT-ის ანალიზი. დავიწყეთ P-MERGE-SORT-ის T_1 -ის ანალიზით, რომელსაც $PMS1(n)$ -ით აღვნიშნავთ. $PMS1(n)$ მოიცემა რეკურსიით:

$$PMS1(n) = 2PMS1(n/2) + PM1(n) = 2PMS1(n/2) + \theta(n)$$

ისევე როგორც სერიული დალაგებისთვის შერწყმით, ამ რეკურსიის ამონახსნია: $PMS1(n) = \theta(n \lg n)$. ახლა გამოვითვალოთ T_∞ უარესი შემთხვევისთვის - $PMS_\infty(n)$.

რადგანაც ორი რეკურსიული გამოძახება ლოგიკურად ერთმანეთის პარალელურად სრულდება, შეგვიძლია მხოლოდ ერთი გავითვალისწინოთ, რის შედეგადაც მივიღებთ რეკურენტულ ფორმულას:

$$PMS_\infty(n) = PMS_\infty(n/2) + PMS_\infty(n) = PMS_\infty(n/2) + \theta(\lg 2n)$$

$$\text{რეკურსიის გახსნის შედეგად მივიღებთ: } PMS_\infty(n) = \theta(\lg 3n)$$

P-MERGE-SORT-ის პარალელურობის მნიშვნელობაა $PMS1(n)/PMS_\infty(n) = \theta(n \lg n) / \theta(\lg 3n) = \theta(n / \lg 2n)$, რაც ბევრად უკეთესია, ვიდრე იმ ალგორითმის პარალელურობის მნიშვნელობა, რომელიც სერიულ შერწყმას იყენებს.

პრაქტიკაში კარგი შედეგის მისაღებად მიზანშეწონილია საბაზისო შემთხვევის „გაუხეშება“: პატარა ზომის მასივებისთვის სერიული დახარისხების ალგორითმის გამოძახება.

ფაქტორიალის გამოთვლა. ფაქტორიალის გამოთვლა პარალელური ალგორითმის გამოყენებით უფრო სწრაფია, ვიდრე სერიული, თანამიმდევრული შესრულებით. ჩვეულებრივ, ფაქტორიალი მიიღება რიცხვების გადამრავლების შედეგად, ფაქტორიალი არის 1-დან n -მდე რიცხვების გამრავლების შედეგი ანუ $n! = 1 * 2 * 3 * \dots * n$. ეს პროცესი სერიულია ანუ თითო ბიჯი უნდა დასრულდეს, სანამ მომდევნო დაიწყება, მაგრამ პარალელური ალგორითმით ბიჯები შესაძლებელია დავყოთ ნაწილებად და თითოეული ნაწილი დამოუკიდებლად გავუშვათ შესასრულებლად სხვადასხვა პროცესორზე, შემდეგ კი მიღებული შედეგები გავამრავლოთ. განვიხილოთ ნაწილებად დაყოფა:

დავყოთ 1-დან n -მდე რიცხვები რამდენიმე პატარა ჯგუფად, რომლებსაც შემდეგ პროცესორებს გადავცემთ. მაგალითად, თუ $n = 8$ და ოთხი პროცესორი გვაქვს, თითოეულს გადავცემთ შემდეგი ტიპის ამოცანებს:

პირველი პროცესორი: $1 * 2$

მეორე პროცესორი: $3 * 4$

მესამე პროცესორი: $5 * 6$

მეოთხე პროცესორი: $7 * 8$

გადამრავლება. თითოეული პროცესორი გამრავლების თავის ნაწილს ასრულებს:

პირველი პროცესორი: $1 * 2 = 2$

მეორე პროცესორი: $3 * 4 = 12$

მესამე პროცესორი: $5 * 6 = 30$

მეოთხე პროცესორი: $7 * 8 = 56$

შედეგების გაერთიანება. ბოლოს მიღებულ შედეგებს კვლავ ვამრავლებთ, თუმცა ამჯერად ეს შეიძლება კვლავ პარალელურად ჩავატაროთ:

$2 * 12 = 24$ და $30 * 56 = 1680$ შემდეგ $24 * 1680 = 40320$, რაც არის $8!$.

ამ მეთოდით, თუ ძალიან დიდი n გვაქვს, გამრავლების ოპერაციები პარალელურად ნაწილდება და მთლიანი ფაქტორიალის გამოთვლის დრო მნიშვნელოვნად მცირდება.

ფაქტორიალის გამოსათვლელად პარალელური ალგორითმის გამოყენება განსაკუთრებით სასარგებლოა დიდი რიცხვებისთვის, რადგან თითოეული პროცესორი ამცირებს ერთჯერადი გამოთვლის დროს.

გაუსის მეთოდის პარალელიზაცია. გაუსის მეთოდი არის ერთ-ერთი ყველაზე გავრცელებული ალგორითმი წრფივი განტოლებების სისტემის გადაჭრისთვის, რომელიც გამოიყენება მრავალმხრივ, მათ შორის სამეცნიერო გამოთვლებში. ამ ალგორითმის პარალელურობის უზრუნველყოფა მოითხოვს ოპერაციების სწორად დაყოფას, რათა სხვადასხვა პროცესორმა შეძლონ ერთდროულად მუშაობა. გაუსის მეთოდის პარალელიზაცია საკმაოდ საინტერესოა და დამოკიდებულია პროცესორების არქიტექტურასა და შესრულებაზე. გაუსის მეთოდის პარალელიზაციის ძირითადი ალგორითმის ბიჯები:

ბიჯი 1. მოცემულია A (მატრიცა) და b (ვექტორი), რომლებიდანაც ამოვხსნით $Ax = b$ სისტემას.

ბიჯი 2. მთავარი სტრიქონის არჩევა. ეს ბიჯი შეიძლება პარალელურად განხორციელდეს. ძირითადი იდეა ისაა, რომ მიმდინარე სტრიქონში ამოვარჩიოთ ყველაზე დიდი ელემენტი, რათა თავიდან ავიცილოთ რიცხვითი არასტაბილურობა.

ბიჯი 3. ნულოვანი ელემენტების აღმოფხვრა. გავანაწილოთ ნულების აღმოფხვრის ოპერაციები სხვადასხვა პროცესორზე, ისე, რომ თითოეული მათგანი ასრულებდეს ოპერაციებს სხვადასხვა სტრიქონთან მიმართებით.

ბიჯი 4. განახლების ოპერაცია. ამოცანა მივაწოდოთ თითოეულ პროცესორს ისე, რომ ერთდროულად იმუშაონ სხვადასხვა სტრიქონებზე, თუმცა, აუცილებელია მონაცემების სინქრონიზაცია ყოველი მთავარი სტრიქონის შემდეგ.

ბიჯი 5. უკუსვლითი ამოხსნა. ბოლო ბიჯი ასევე შეიძლება პარალელურად განხორციელდეს, სადაც თითოეულ პროცესორს შეუძლია ამოიღოს განსხვავებული ცვლადი თანამიმდევრულად.

ძირითადი პარალელური სტრატეგიები:

1. სტრიქონების დონეზე პარალელიზაცია. თუ გვაქვს P პროცესორი და n სტრიქონი, მაშინ თითოეული პროცესორი პასუხს აგებს n/P სტრიქონზე.

2. დონეების მიხედვით დისტრიბუცია. პარალელურობის პროცესი მიმდინარეობს დონეების მიხედვით (სტრიქონების დაყოფა ხდება დონეების მიხედვით), ისე რომ ნაკლები კომუნიკაცია იყოს პროცესორებს შორის.

3. შეზღუდული სინქრონიზაცია. ყოველი სტრიქონის შემდეგ საჭიროა სავალდებულო კომუნიკაცია, რათა პროცესი დარჩეს სტაბილური, თუმცა ზოგიერთი ოპერაცია შეიძლება ასინქრონულადაც განხორციელდეს.

ასეთი მიდგომით, გაუსის მეთოდის ალგორითმი უფრო სწრაფად მუშაობს, რადგან გამოთვლები მიმდინარეობს პარალელურად, გაუსის მეთოდის პარალელური ალგორითმი სწრაფად ხორციელდება დიდი მონაცემების მქონე სისტემებზე.

განვიხილოთ მაგალითი. ვთქვათ, მოცემულია გვაქვს სისტემა:

$$\begin{cases} 2x + 3y + z = 9 \\ x + 2y + 3z = 8 \\ 3x + y + 2z = 7 \end{cases}$$

წარმოვადგინოთ მოცემული სისტემა მატრიცული სახით:

$$A = \begin{pmatrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}, \quad b = \begin{pmatrix} 9 \\ 8 \\ 7 \end{pmatrix}$$

გადავიდეთ გაუსის მეთოდის ალგორითმის პარალელურად განხორციელებაზე:

1. პირველი სტრიქონის გარდაქმნა - ორივე მატრიცის პირველი სტრიქონის ელემენტები გავხადოთ 1-ის ტოლი ანუ გავყოთ 2-ზე. მივიღებთ:

$$\begin{pmatrix} 1 & 1.5 & 0.5 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} b = \begin{pmatrix} 4.5 \\ 8 \\ 7 \end{pmatrix}$$

1.1. პირველ სტრიქონს გამოვაკლოთ თითოეულ სტრიქონს ისე, რომ პირველი სვეტის სხვა ელემენტები ნული გახდეს. ეს ოპერაცია შეიძლება პარალელურად შესრულდეს.

1.2. მეორე სტრიქონს გამოვაკლოთ 1-ჯერ გაზრდილი პირველი სტრიქონი:

$$\begin{pmatrix} 1 & 1.5 & 0.5 \\ 0 & 0.5 & 2.5 \\ 3 & 1 & 2 \end{pmatrix} b = \begin{pmatrix} 4.5 \\ 3.5 \\ 7 \end{pmatrix}$$

1.3. მესამე სტრიქონს გამოვაკლოთ 3-ჯერ გაზრდილი პირველი სტრიქონი:

$$\begin{pmatrix} 1 & 1.5 & 0.5 \\ 0 & 0.5 & 2.5 \\ 0 & -3.5 & 0.5 \end{pmatrix} b = \begin{pmatrix} 4.5 \\ 3.5 \\ -6.5 \end{pmatrix}$$

2. მეორე სტრიქონის გარდაქმნა. მეორე სტრიქონის მეორე ელემენტი გავხადოთ 1-ის ტოლი ანუ გავყოთ 0.5-ზე:

$$\begin{pmatrix} 1 & 1.5 & 0.5 \\ 0 & 1 & 5 \\ 0 & -3.5 & 0.5 \end{pmatrix} b = \begin{pmatrix} 4.5 \\ 7 \\ -6.5 \end{pmatrix}$$

2.1. ნულის შექმნა მესამე სტრიქონში მეორე ელემენტის ადგილას. მეორე სტრიქონს გამოვაკლოთ მესამე სტრიქონი ისე, რომ მესამე სტრიქონის მეორე ელემენტი გახდეს ნულის ტოლი:

$$\begin{pmatrix} 1 & 1.5 & 0.5 \\ 0 & 1 & 5 \\ 0 & 0 & -17 \end{pmatrix} b = \begin{pmatrix} 4.5 \\ 7 \\ -24.5 \end{pmatrix}$$

3. უკუსვლით ამოხსნა.

3.1. მესამე სტრიქონის საშუალებით ვპოულობთ z -ის მნიშვნელობას:

$$-17z = -24.5$$

$$z = 1.44$$

3.2. მეორე სტრიქონის საშუალებით ვპოულობთ y -ის მნიშვნელობას:

$$y + (5 * 1.44) = 7$$

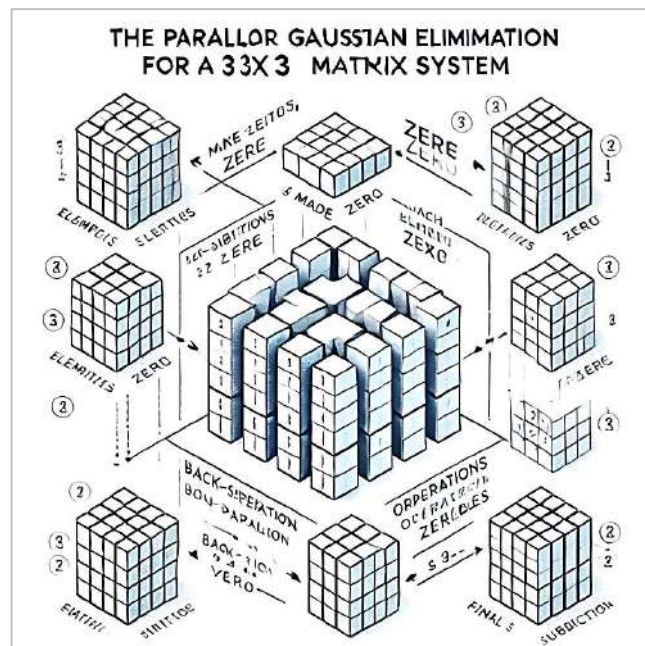
$$y = 7 - 7.2 = -0.2$$

3.3. პირველი სტრიქონის საშუალებით ვპოულობთ x -ის მნიშვნელობას:

$$x + (1.5 * (-0.2)) + (0.5 * 1.44) = 4.5$$

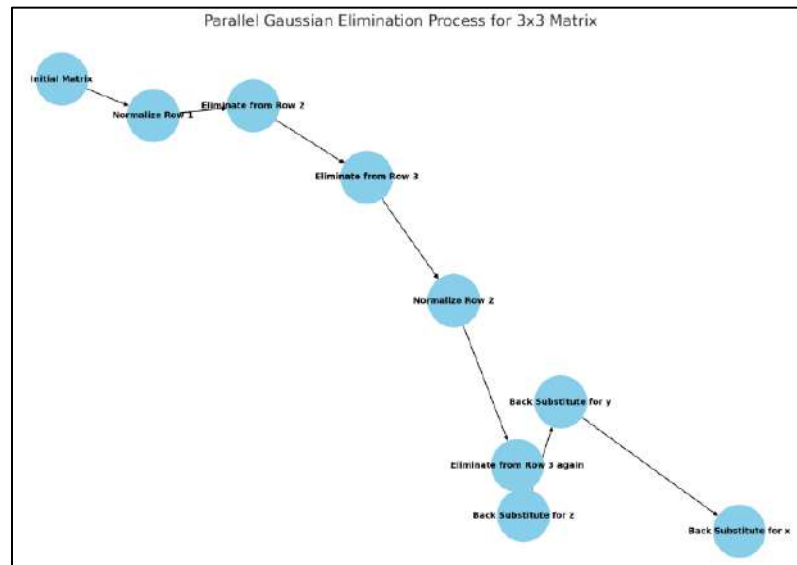
$$x = 2.72$$

ზემოთ აღწერილ მაგალითში პროცესების ერთდროულად გამოყენება შესაძლებელია სხვადასხვა სტრიქონისთვის (იხ. ნახ. 14.14, 14.15).



ნახ. 14.14. დიაგრამა ასახავს პარალელური გაუსის მეთოდს 3×3 მატრიცაზე.

დიაგრამა აჩვენებს გარდაქმნის პროცესის ბიჯებს და იმას, თუ რომელ სტრიქონებზე მუშაობს პროცესორები პარალელურად.



ნახ. 14.15. გრაფიკი ასახავს პარალელური გაუსის ალგორითმის მუშაობის პროცესის ეტაპებს 3×3 მატრიცაზე.

პარკეტის დაგების ამოცანა. პარკეტის დაგების ამოცანა კომბინატორული ამოცანაა, რომლის მიზანია, მოცემული ფართობი (მაგ., ოთახი) დაიფაროს სხვადასხვა ზომის ან ფორმის ფიგურებით ისე, რომ არ დარჩეს ცარიელი ადგილები და ნაწილებმა არ გადაფარონ ერთმანეთი.

პარკეტის დაგების ამოცანა ხშირად ითხოვს ისეთი ალგორითმის შექმნას, რომელიც სივრცეს ოპტიმალურად ავსებს კვადრატული, მართკუთხა ან სხვა ფორმის პარკეტის ფირფიტებით ისე, რომ მაქსიმალურად დაიფაროს არსებული სივრცე.

პარკეტის დაგების ამოცანის ალგორითმის ეტაპები:

1. სივრცის გაზომვა და მოდელირება. დავადგენთ ოთახის ზომებს, პარკეტის ფირფიტების ზომას და ფორმას. ამის შემდეგ შესაძლებელია სივრცის განლაგების მოდელირება ისე, რომ სივრცე მოვიცვათ მაქსიმალურად.

2. თავსებადობის შემოწმება. პარკეტის დაგება მოითხოვს ფირფიტების მჭიდრო შერწყმას. საჭიროა ისეთი ალგორითმის შექმნა, რომელიც შეამოწმებს, თუ რომელი პოზიციებია შესაძლებელი ფირფიტების განლაგებისთვის ისე, რომ სივრცე მაქსიმალურად დაიფაროს და ნაკლები დანაკარგი დარჩეს.

3. გადაწყობის ალგორითმი. პრობლემის გადაჭრისთვის შეიძლება გამოყენებულ იქნას დინამიკური დაგეგმვა ან გრადის ალგორითმი.

მაგალითად, გრადის ალგორითმი ირჩევს მეზობელ პაკეტს, რომელიც ოპტიმალურად შეავსებს დანარჩენ სივრცეს.

4. უკან დაბრუნების მეთოდი. თუ გარკვეული პოზიციები ან ცდები არასწორია, ალგორითმს შეუძლია „უკან დაბრუნდეს“ და სხვა შესაძლო ვარიანტები სცადოს.

პარკეტის დაგების ამოცანის მაგალითი. ვთქვათ მოცემული გვაქვს მართკუთხა ფორმის ოთახი, რომლის ზომებია $10 * 6$ მეტრი, ხოლო პარკეტის თითოეული ფირფიტა არის $2 * 1$ მეტრი ზომის.

ჩვენს მიზანს წარმოადგენს, ისე დავფაროთ ოთახის იატაკი პარკეტით, რომ ზედმეტი ცარიელი სივრცე არ დარჩეს და ფირფიტების მდებარეობა თანაბარი იყოს.

ამოცანის გადაჭრის ალგორითმი ცდილობს, შესაბამისი პოზიციები მიუჩინოს თითოეულ ფირფიტას, მაგალითად მარცხნიდან მარჯვნივ და ზემოდან ქვემოთ მიმართულებით.

ამოხსნა:

1. ოთახის ($10 * 6$ -ზე) სივრცე დაყოფილია უჯრედებად. თითოეული უჯრედის ზომაა $1 * 1$ -ზე მეტრი. მთლიანობაში გვაქვს 60 კვადრატული უჯრედი.

2. **ფირფიტების განლაგების ალგორითმი.** თითოეული ფირფიტა იკავებს $2 * 1$ უჯრედს, ამიტომ ალგორითმს შეუძლია დაათვალიეროს თითოეული უჯრედი (მარცხნიდან მარჯვნივ, ზემოდან ქვემოთ) და თუ უჯრედი თავისუფალია, მისი გვერდით მდებარე უჯრედიც შეამოწმოს.

თუ ორივე თავისუფალია, მაშინ ამ ორ უჯრედზე განათავსებს ფირფიტას.

3. **უკან დაბრუნების მეთოდი (თუ საჭიროა).** თუ ვერ იპოვა ადგილი თანამიმდევრობით ფირფიტების დასადებად, დაბრუნდება უკან და შეცვლის ფირფიტების განლაგების სვლას, რათა ოთახის სივრცე თანაბრად მოიცვას.

4. **საბოლოო განლაგება.** ამ გზით, ოთახი დაიფარება ისე, რომ თითოეული ფირფიტა დაეტევა გვერდიგვერდ თანაბრად და მთლიანი სივრცე ოპტიმალურად დაიფარება.

პარკეტის დაგების ამოცანის **დინამიკური დაგეგმვის (Dynamic Programming)** მეთოდით შესაძლებელია კონკრეტული ფართის ნაწილების დაფარვის კომბინაციების გამოთვლა, ნაწილების სუბსტრუქტურების შემოწმებით და ქეშირებით.

ქვემოთ მოცემულია კოდი, რომელიც ამარტივებს პარკეტის ელემენტების განლაგებას ოთახში:

```
def can_tile_room(m, n, a, b):  
    # თუ პარკეტის ზომა ვერ თავსდება ზუსტად ოთახის ზომაში, ვაბრუნებთ False  
    if (m * n) % (a * b) != 0:  
        return False  
    # ცხრილი, რომლითაც შევინახავთ მოცემულ სივრცეს  
    dp = [[False] * (n + 1) for _ in range(m + 1)]  
    dp[0][0] = True  
    # დავამატოთ პარკეტით შევსება
```

```

for i in range(m + 1):
    for j in range(n + 1):
        if dp[i][j]:
            if i + a <= m and j + b <= n:
                dp[i + a][j + b] = True
            if i + b <= m and j + a <= n:
                dp[i + b][j + a] = True
    return dp[m][n]

```

გამოყენების მაგალითი

m, n = 6, 8 # ოთახის ზომა

a, b = 2, 4 # პარკეტის ზომა

print("შესაძლებელია თუ არა ოთახის შევსება:", can_tile_room(m, n, a, b))

მოცემული კოდის საშუალებით შეიძლება შევამოწმოთ, შესაძლებელია თუ არა მოცემული ზომის პარკეტით ოთახის დაფარვა.

პარკეტის დაგების ამოცანის ამოხსნა საჭიროებს ისეთ ალგორითმებს, რომლებიც ეფექტურად გამოიმუშავენ სხვადასხვა კომბინაციას.

საუკეთესო მიდგომა ხშირად დამოკიდებულია კონკრეტულ მოთხოვნებზე: თუ მნიშვნელოვანია ზუსტი ამოხსნა, შეიძლება გამოვიყენოთ დინამიკური დაგეგმვა, ხოლო ზოგიერთ შემთხვევაში შესაძლებელია როგორც პარალელიზმის, ისე რეკურსიის გამოყენებაც.

პარალელური მონაცემები და პარალელური გამოთვლების ალგორითმები.

ძირითადი განმარტებები, გადასვლები მაჩვენებლებით, დანახარჯების ეფექტურობა, პრეფიქსების ეფექტური პარალელური დამუშავება, სიმეტრიული დარღვევა, პროგნოზირების ამოცანები, პროგნოზირების მოდელების შეჯერების მეთოდი.

პარალელური მონაცემები და პარალელური გამოთვლების ალგორითმები კომპიუტერული მეცნიერების ერთ-ერთი მნიშვნელოვანი მიმართულებაა. პარალელური მონაცემების და პარალელური გამოთვლების ალგორითმში შეგვიძლია ვიგულისხმოთ ამოცანების და მონაცემების პარალელურად (ერთდროულად) დამუშავება, რაც აძლიერებს გამოთვლების სისწრაფეს და ეფექტურობას.

პარალელური მონაცემები (Parallel Data). პარალელური მონაცემები არის ისეთი ტიპის (ფორმატის) მონაცემები, რომლებიც შეიძლება ერთდროულად დამუშავდეს რამდენიმე პროცესორის ან პროცესის მიერ. ეს განსაკუთრებით მნიშვნელოვანია დიდი მოცულობის მონაცემებისთან მიმართებით, რადგან მათი ერთ პროცესორზე დამუშავება, ხშირად არ არის საკმარისი მონაცემთა სწრაფად დასამუშავებლად.

პარალელური მონაცემების ძირითადი ცნებებია:

1. **განაწილებული მონაცემთა ნაკრები.** მონაცემები იყოფა ნაწილებად და ნაწილდება სხვადასხვა პროცესორებს შორის, რათა ყველა მათგანმა იმუშაოს პარალელურად. ეს შეამცირებს ცალკეულ პროცესორზე დატვირთვას.

2. **სინქრონიზაცია.** მონაცემების გადაცემის ან პარალელური მუშაობისას საჭიროა სინქრონიზაცია, რათა პროცესორები ერთმანეთს არ შეეჯახონ და მათი მოქმედება იყოს ეფექტური.

3. **განგარიშების პარალელიზმი.** თითოეული პროცესორი მუშაობს დამოუკიდებელად მონაცემების ნაწილთან, რათა მთელი ამოცანა ერთდროულად გადაწყდეს.

პარალელური გამოთვლების ალგორითმები (Parallel Computing Algorithms). პარალელური გამოთვლების ალგორითმები არის პროგრამების კოდი, რომელიც ნაწილდება ერთდროულად რამდენიმე პროცესორს შორის და მუშაობს პარალელურად. მათი მიზანია გამოთვლითი დროის შემცირება ისეთი ტიპის ამოცანებისთვის, როგორიცაა დიდი მონაცემთა მასივების დამუშავება ან რთული მოდელების გამოთვლა. პარალელური გამოთვლები (პარალელური დამუშავება) არის ერთი და იგივე პროგრამის (პროექტის) სხვადასხვა ნაწილის ერთდროულად შესრულება რამდენიმე ან ბევრი გამოთვლითი მოწყობილობის გამოყენებით.

პარალელური გამოთვლები არის ისეთი გამოთვლები, რომლებიც შეიძლება განხორციელდეს მრავალპროცესორულ სისტემებზე, ერთდროულად მრავალი მოქმედების შესრულების შესაძლებლობის გამოყენებით, რომლებიც წარმოიქმნება ერთ-ერთი ან მრავალი ამოცანის (ერთი პროექტის) გადაწყვეტის პროცესის დროს (კიბერნეტიკის ლექსიკონი). პარალელური გამოთვლების ძირითადი მიზანია ამოცანის გადაწყვეტის დროის (დროითი დანახარჯების) შემცირება.

პრაქტიკული ამოცანების დიდი ნაწილი უნდა გადაწყდეს რეალურ დროში ან მათი გადაწყვეტისთვის საჭიროა ძალიან დიდი მოცულობის გამოთვლები. უნდა აღვნიშნოთ, რომ პროცესორების რაოდენობის გაზრდა ყოველთვის არ უზრუნველყოფს ამოცანის გადაწყვეტის დროის შემცირებას. გამოთვლების სისწრაფის გასაზრდელად მონაცემების პარალელური დამუშავების გამოყენება არ შეიძლება ჩავთვალოთ როგორც ერთადერთი გზა. არსებობს ასევე პროცესორის მოწყობილობების სიმძლავრის გაზრდის მიდგომა.

განვიხილოთ პარალელური გამოთვლების ტერმინოლოგია:

1. სუპერკომპიუტერი. სუპერკომპიუტერი არის გამოთვლითი მანქანა, რომელიც მნიშვნელოვნად აღემატება არსებულ კომპიუტერებს თავისი ტექნიკური პარამეტრებით.

2. კლასტერი. კლასტერი არის კომპიუტერების ერთობლიობა (ჯგუფი), რომელიც გაერთიანებულია ლოკალურ ქსელში და შეუძლია იმუშაოს როგორც ერთი გამოთვლითი რესურსი. კლასტერი უფრო ეფექტურია და მაღალ საიმედოობისაა, ვიდრე უბრალოდ ლოკალური ქსელი. კლასტერი იყენებს სტანდარტულ აპარატურულ და პროგრამულ გადაწყვეტილებებს და შესაბამისად, აქვს უფრო ნაკლები ღირებულება პარალელური გამოთვლითი სისტემების სხვა ტიპებთან შედარებით.

3. განაწილებული გამოთვლა. განაწილებული გამოთვლა არის მონაცემთა დამუშავების ისეთი ტექნოლოგია, როდესაც დიდი ზომის ამოცანები შესასრულებლად გადანაწილდება ბევრ კომპიუტერს შორის, რომლებიც ერთმანეთთან დაკავშირებულია კომპიუტერული ქსელით ან ინტერნეტით.

4. ღრუბლოვანი გამოთვლები. ღრუბლოვანი გამოთვლები არის კომპიუტერული სერვისების მიწოდების მოდელი, რომელიც გულისხმობს რესურსების, მაგალითად, მონაცემთა შენახვის, სერვერების, პროგრამული უზრუნველყოფის და მონაცემთა ბაზების დისტანციური ხელმისაწვდომობის უზრუნველყოფას ინტერნეტის მეშვეობით. ღრუბლოვანი გამოთვლები საშუალებას იძლევა მომხმარებლებმა გამოიყენონ საჭირო რესურსები მათ ხელთ არსებული ინფრასტრუქტურის გარეშე.

პარალელური ამოცანები. პარალელური ამოცანები არის ისეთი სახის ამოცანები, რომელთა შესრულებაც ერთდროულად ხდება (არ არის აუცილებელი დამოუკიდებლად).

პარალელური ალგორითმი. პარალელური ალგორითმი არის ალგორითმი, რომელთა ოპერაციების შესრულებაც ხდება ერთდროულად (არ არის აუცილებელი დამოუკიდებლად).

პარალელური პროგრამა. პარალელური პროგრამა არის ისეთი სახის პროგრამა, რომელიც ერთდროულად ასრულებს რამდენიმე გამოთვლას ან ამოცანას სხვადასხვა პროცესორზე, რათა უფრო სწრაფად და ეფექტურად დაამუშაოს მონაცემები. ეს ხდება რესურსების უკეთ გამოყენების, დროის შემცირების და დიდ მონაცემებთან მუშაობისათვის გამოთვლების დაჩქარების მიზნით.

მონაცემთა პარალელიზმი. ფუნქციური პარალელიზმი. თუ გარკვეული ამოცანის გადაჭრისას პროცესორები გამოთვლების ერთსა და იგივე თანამიმდევრობას ასრულებენ, მაგრამ იყენებენ სხვადასხვა მონაცემებს, მაშინ საუბარია მონაცემთა პარალელიზმზე. მაგალითად, მონაცემთა ბაზაში ძიებისას, თითოეულ პროცესორს შეუძლია იმუშაოს მონაცემთა ბაზის საკუთარ ნაწილთან. თუ პროცესორები ასრულებენ ერთი და იმავე ამოცანის განსხვავებულ დავალებას, ასრულებენ სხვადასხვა ფუნქციებს, მაშინ ვსაუბრობთ ფუნქციურ პარალელიზმზე.

გამოთვლითი სისტემის **რეალური წარმადობა** არის დროის ერთეულში რეალურად შესრულებული ოპერაციების საშუალო რაოდენობა.

გამოთვლითი სისტემის **პიკური წარმადობა** არის სისტემის მიერ დროის ერთეულში ოპერაციების შესრულების მაქსიმალური რაოდენობა.

გამოთვლითი სისტემის **რეალური და პიკური შესრულება** მოიცავს კომპიუტერული ან გამოთვლითი სისტემების წარმადობის შეფასებას, განსაკუთრებით მაღალსიჩქარიანი გამოთვლითი ან პარალელური გამოთვლების სისტემებში.

განმარტებებიდან გამომდინარეობს, რომ სისტემის რეალური და პიკური შესრულება არის შესაბამისად პროცესორული სისტემის რეალური და პიკური შესრულების ჯამები. ერთი პროცესორის პიკური წარმადობა გამოითვლება, როგორც $n * f * k$ ნამრავლი, სადაც n არის პროცესორის ერთი ტაქტის დროს შესრულებული მცოცავმიმდინაო ოპერაციების რაოდენობა, f არის პროცესორის ტაქტის სიხშირე, k არის პროცესორში ბირთვების რაოდენობა. პიკური წარმადობას ასევე უწოდებენ თეორიულ წარმადობას. ტერმინი ხაზს უსვამს იმას, რომ რეალურ პროგრამაში წარმადობა არათუ არ გადააჭარბებს, არამედ ვერასოდეს მიაღწევს ამ ზღურბლს. დროის მოცემულ პერიოდში **პროცესორის დატვირთვა** არის მოცემულ სეგმენტზე პროცესორის რეალური მუშაობის დროის თანაფარდობა სეგმენტის მთელ სიგრძესთან. იდენტური პროცესორების გამოთვლითი სისტემის დატვირთვა არის ყველა პროცესორის დატვირთვის საშუალო არითმეტიკული.

განმარტებებიდან გამომდინარე, დატვირთვა p აკმაყოფილებს $0 \leq p \leq 1$ პირობას.

სინქრონიზაცია პროცესორის მუშაობაში წარმოადგენს მექანიზმს, რომელიც უზრუნველყოფს სხვადასხვა პროცესების ან ნაკადების ისეთ შეთანხმებულ მუშაობას, რომ მონაცემთა პარალელურ დამუშავებასთან არ მოხდეს კონფლიქტი ან არასწორი შედეგების მიღება. სინქრონიზაცია განსაკუთრებით მნიშვნელოვანია **პარალელური პროგრამირების** დროს, როცა რამდენიმე ნაკადი ან პროცესი ერთდროულად მუშაობს საერთო რესურსებთან, მაგალითად, მეხსიერებასთან ან მონაცემთა ბაზებთან.

ბრძანებების ნაკადი (Thread) არის ბრძანებების ნაკადის მინიმალური ერთობლიობა რომლებსაც პროცესორი ასრულებს თანამიმდევრობით სანამ გადაერთვება სხვა ნაკადზე.

Multithreading ნიშნავს როცა პროგრამაში ბრძანებები დაწერილია რამდენიმე ნაკადად და სხვადასხვა ნაკადის ბრძანებები სრულდება ერთდროულად. პროცესორს უნდა ჰქონდეს რამდენიმე ბირთვი ან ბრძანებების რამდენიმე ნაკადად შესრულების უნარი, ასევე პროგრამაშიც ბრძანებები უნდა იყოს რამდენიმე ნაკადად დაწერილი.

გადასვლები მაჩვენებლებით და მათი მნიშვნელობა პარალელურ გამოთვლებში. გადასვლები მაჩვენებლებით ხშირად გამოიყენება პარალელური ალგორითმების შესრულების შესაფასებლად და იმის დასაანხად, თუ როგორ იცვლება ალგორითმის სისწრაფე და ეფექტურობა პროცესორების რაოდენობის მიხედვით.

პარალელური გამოთვლების ალგორითმები ეფუძნება გამოთვლების პარალელურ შესრულებას, რაც საშუალებას გვაძლევს დავაჩქაროთ რთული ამოცანების ამოხსნა.

გადასვლები (transition) პარალელურ გამოთვლებში ეხება იმ პროცესს, როცა ერთი მდგომარეობიდან მეორეში გადასვლას ვახდენთ და მაჩვენებლები (metrics) გამოიყენება ამ გადასვლების ეფექტურობის შესაფასებლად.

პარალელური გამოთვლების ეფექტურობის შეფასება. პარალელურ გამოთვლებში არსებობს რამდენიმე მთავარი მაჩვენებელი, რომლებიც განსაზღვრავენ სისტემის ეფექტურობას და შესრულების ხარისხს. ისინი ძირითადად გამოიყენება იმის გასაზომად, თუ რამდენად კარგად იყენებს სისტემა პარალელურ რესურსებს. პარალელური გამოთვლების ეფექტურობის შეფასება შეიძლება ჩამოვყალიბოთ სამ პუნქტად:

1. პარალელური ალგორითმის ეფექტურობის მაჩვენებლები:

- დაჩქარება (**Speedup**);
- ეფექტურობა (**efficiency**);
- ღირებულება.

2. პარალელურობის მაქსიმალური შეფასების კანონები:

- ამდელის კანონი;
- გუსტაფსონის კანონი.

3. პარალელური ალგორითმის მასშტაბურობის ანალიზი.

ეფექტურობის მაჩვენებელი სიჩქარე. სიჩქარე არის ეფექტურობის ისეთი მაჩვენებელი, რომელიც ზომავს, თუ რამდენად სწრაფად სრულდება პროგრამა პარალელური ალგორითმის გამოყენებით მრავალბირთვიან პროცესორებზე, ვიდრე ერთბირთვიან პროცესორზე გამოთვლების თანმიმდევრული შესრულებისას ანუ ასახავს პარალელური პროგრამის შესრულების სიჩქარის ზრდას პროცესორების მოცემული რაოდენობის პირობებში ერთპროცესორულ სისტემასთან შედარებით.

სიჩქარის ფორმულა გამოისახება შემდეგნაირად:

$$S_{speedup} = \frac{T_{serial}}{T_{parallel}}$$

სადაც T_{serial} არის დრო, რომელიც სჭირდება ამოცანის ერთბირთვიან პროცესორზე თანმიმდევრულ შესრულებას, ხოლო $T_{parallel}$ არის დრო, რომელიც სჭირდება ამოცანის მრავალბირთვიან (მრავალნაკადიან) პროცესორზე პარალელურ შესრულებას.

იდეალური შემთხვევაა, როცა ამოცანა სრულად პარალელდება და შესრულების დრო მცირდება ბირთვების ან პროცესორების რაოდენობის პროპორციულად. მაგალითად, თუ პროცესორი ორჯერ მეტ ბირთვს იყენებს, დრო უნდა შემცირდეს ორჯერ.

იდეალური სიჩქარის ფორმულაა: $S_{speedup} = N$, სადაც N გამოყენებული პროცესორების ან ბირთვების რაოდენობაა.

ეფექტურობის მაჩვენებელი ეფექტურობის შეფასება (Efficiency Evaluation). ეფექტურობა გვიჩვენებს, თუ რამდენად ეფექტურად იყენებს პროგრამა დამატებით რესურსებს სიჩქარის გასაზრდელად.

ეფექტურობა განისაზღვრება შემდეგნაირად:

$$E_{efficiency} = \frac{S_{speedup}}{P}$$

სადაც, $S_{speedup}$ - არის პარალელიზმით მიღებული სიჩქარის ზრდა;

P - გამოყენებული პროცესორების რაოდენობა.

თუ ეფექტურობა 100%-ია, ეს ნიშნავს, რომ ყველა ბირთვი გამოიყენება ოპტიმალურად.

ეფექტურობის მნიშვნელობები. 100%-იანი ეფექტურობა ნიშნავს, რომ სისტემა იდეალურად იყენებს ყველა პროცესორს და ყოველი დამატებითი ბირთვი იწვევს ამოცანის შესრულების დროის პროპორციულ შემცირებას.

100%-ზე ნაკლები ეფექტურობა ნიშნავს, რომ პარალელიზმის შედეგად მიღებული სარგებელი მცირეა და დამატებითი პროცესორები მხოლოდ ნაწილობრივ აუმჯობესებენ შესრულების დროს. ეს მაშინ ხდება, როდესაც ზოგიერთი ამოცანა ვერ პარალელდება ან საჭიროა სინქრონიზაცია პროცესებს შორის.

ეფექტურობის შეფასების მაგალითი. ვთქვათ, ერთი პროგრამის შესრულების დროა 100 წამი, ხოლო იგივე პროგრამა პარალელურ სისტემაში 4 პროცესორის გამოყენებით შესრულდება 40 წამში. ამ შემთხვევაში:

$$\text{სიჩქარე იქნება: } S_{\text{speedup}} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{100}{40} = 2.5$$

$$\text{ეფექტურობა იქნება: } E_{\text{efficiency}} = \frac{S_{\text{speedup}}}{P} = \frac{2.5}{4} = 0.625 = 62.5\%$$

ეს ნიშნავს, რომ სისტემა ეფექტურად იყენებს პროცესორების რესურსების მხოლოდ 62.5%, რაც მიუთითებს იმაზე, რომ დამატებითი პროცესორები სრულად არ გამოიყენება.

სიჩქარე და ეფექტურობა ერთსა და იმავე მონეტის ორი მხარეა: პარალელური გამოთვლებისას ერთი მაჩვენებლის ხარისხის გაუმჯობესების მცდელობამ შეიძლება გამოიწვიოს მეორე მაჩვენებლის ხარისხის გაუარესება.

პროგრამის შესრულების საშუალო სიჩქარე $V(p)$ შეგვიძლია განვსაზღვროთ ამდელის კანონით.

ამდელის კანონი (Amdahl's Law). თუ პროგრამის ნაწილი შეგვიძლია დავაჩქაროთ პარალელიზმით, მაშინ მთლიანად პროგრამის აჩქარების სიჩქარე შეზღუდულია იმ ნაწილის სიდიდით, რომელიც პარალელურად ვერ მუშაობს.

უფრო კონკრეტულად, პროგრამის ნაწილი შეიძლება პარალელურად შესრულდეს (მაგალითად, მრავალპროცესორიან სისტემაში), თუმცა, პროგრამის ზოგიერთ ნაწილს სჭირდება თანამიმდევრული შესრულება (იგი ვერ პარალელდება).

ამდელის კანონი გვეუბნება, რომ რაც უფრო მეტ რესურსს (მაგალითად, მეტი პროცესორი) დავამატებთ, მით უფრო მცირე იქნება საერთო აჩქარება, თუ პროგრამის დიდი ნაწილი მაინც თანამიმდევრულად უნდა შესრულდეს.

ამდელის კანონი განსაზღვრავს, თუ როგორ ზღუდავს თანამიმდევრული ნაწილის ზომა პარალელური გამოთვლების სიჩქარის ზრდას.

ამდელის კანონის ფორმულა გამოიყურება შემდეგი სახით:

$$S(n) = \frac{1}{(1 - P) + \frac{P}{N}}$$

სადაც, $S(N)$ - სიჩქარე, როცა გვაქვს N რაოდენობის პროცესორი;

P - ამოცანის ის ნაწილია, რომელიც პარალელდება;

$(1 - P)$ - ამოცანის ის ნაწილი, რომელიც უნდა შესრულდეს თანამიმდევრულად;

N - გამოყენებული პროცესორების რაოდენობაა.

წარმოვიდგინოთ პროგრამა, რომლის შესრულებასაც სჭირდება 100 წამი. მოცემული პროგრამის 80% შეგვიძლია პარალელურად გავუშვათ (მრავალ პროცესორზე), მაგრამ 20% არ ექვემდებარება დაპარალელებას და თანამიმდევრულად უნდა შესრულდეს.

ვნახოთ, რა მოხდება, თუ დავამატებთ პროცესორების რაოდენობას:

ერთი პროცესორის შემთხვევაში ყველაფერი სრულდება ერთ პროცესორზე. ამ შემთხვევაში, შესრულების დრო იგივე იქნება - 100 წამი.

ორი პროცესორის შემთხვევაში პროგრამის დაპარალელებული ნაწილი (80%) ორად გაიყოფა ანუ 80 წამი შემცირდება 40 წამამდე. თუმცა, 20% (რომელიც არ პარალელდება) მაინც 20 წამის განმავლობაში უნდა გაგრძელდეს.

ასე რომ, ჯამში პროგრამის შესრულებას დასჭირდება: $40 + 20 = 60$ წამი

ოთხი პროცესორის შემთხვევაში დაპარალელებული 80% კიდევ უფრო სწრაფად შესრულდება - 80 წამი გაიყოფა ოთხ ნაწილად და დრო გახდება 20 წამი, მაგრამ 20% ნაწილი ისევ თანმიმდევრულად უნდა შესრულდეს 20 წამში.

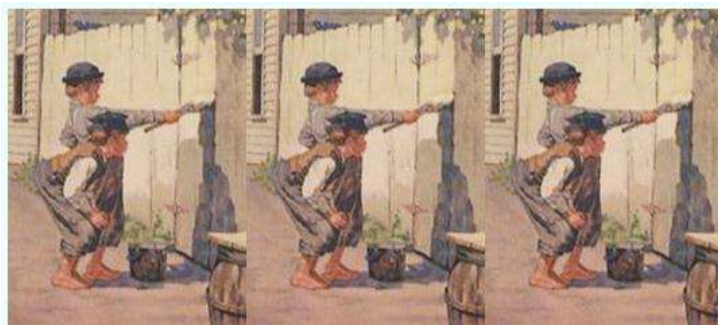
შედეგად, პროგრამის შესრულების დრო იქნება: $20 + 20 = 40$ წამი

ათი პროცესორის შემთხვევაში პარალელური 80% დაიყოფა 10 ნაწილად ანუ ის შესრულდება 8 წამში, თუმცა, პროცესორის ის ნაწილი, რომელიც არ დაექვემდებარა დაპარალელებას - 20% -ს, მაინც დასჭირდება 20 წამი.

შესაბამისად, საერთო დრო იქნება: $8 + 20 = 28$ წამი

მიუხედავად იმისა, თუ რამდენი პროცესორის დამატება მოხდა, პროგრამის მთლიანად შესრულების დრო ყოველთვის შეზღუდული იქნება იმ ნაწილით, რომელიც არ ექვემდებარება დაპარალელებას.

განვიხილოთ კიდევ ერთი ამოცანა ამდალის კანონის გამოყენებით. მოცემულია ღობე, რომელიც შედგება 300 ცალი ფიცრისგან. რამდენი დრო იქნება საჭირო ღობის შესაღებად, თუ ღობეს ღებავს ერთი ან ორი ან 1000000 ადამიანი (ნახ. 15.1).



ნახ. 15.1. ღობის შეღების ილუსტრაცია

ამოცანის შესრულების ეტაპები:

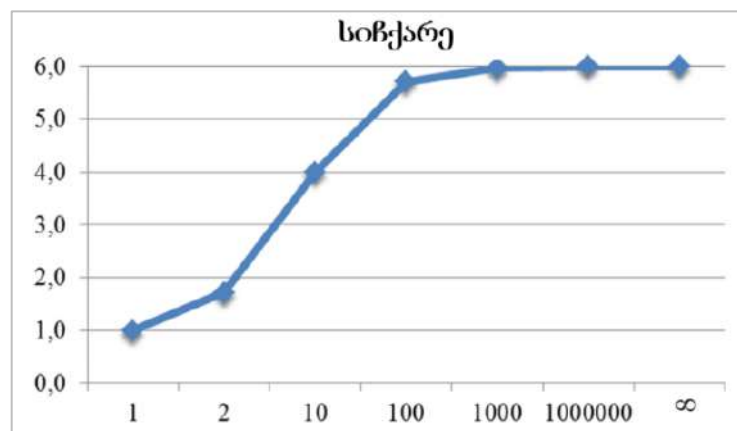
1. ღობის შესაღებად მომზადების ეტაპი - 30 წუთი, არ არის პარალელური პროცესი;
2. ერთი ფიცრის შეღების ეტაპი - 1 წუთი, არის პარალელური პროცესი;
3. შეღების შემდეგ დალაგების ეტაპი - 30 წუთი, არ არის პარალელური პროცესი.

ღობის შეღებისთვის საჭირო დროის მონაცემები მღებავთა სხვადასხვა რაოდენობის შემთხვევაში მოცემულია ცხრილში 15.1.

ცხრილი 15.1. შეღებვისთვის საჭირო დროის მნიშვნელობები

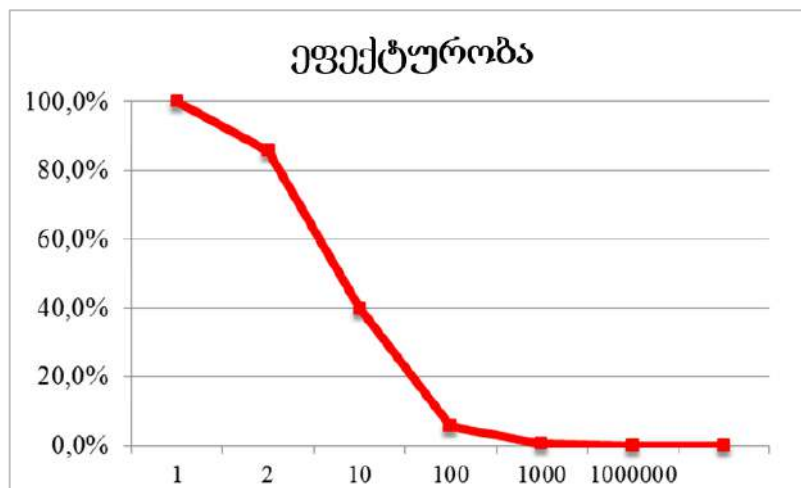
მღებავების რაოდენობა	შეღებვის დრო	
1	$30 + 300/1$	$+ 30 = 360$
2	$30 + 300/2$	$+ 30 = 210$
10	$30 + 300/10$	$+ 30 = 90$
100	$30 + 300/100$	$+ 30 = 63$
1000	$30 + 300/1000$	$+ 30 \approx 60$
1000000	$30 + 300/1000000$	$+ 30 \approx 60$

ცხრილი 15.1-ში მოცემული მონაცემების მიხედვით აგებული სიჩქარის მაჩვენებლის გრაფიკი მოცემულია ნახატზე 15.2.



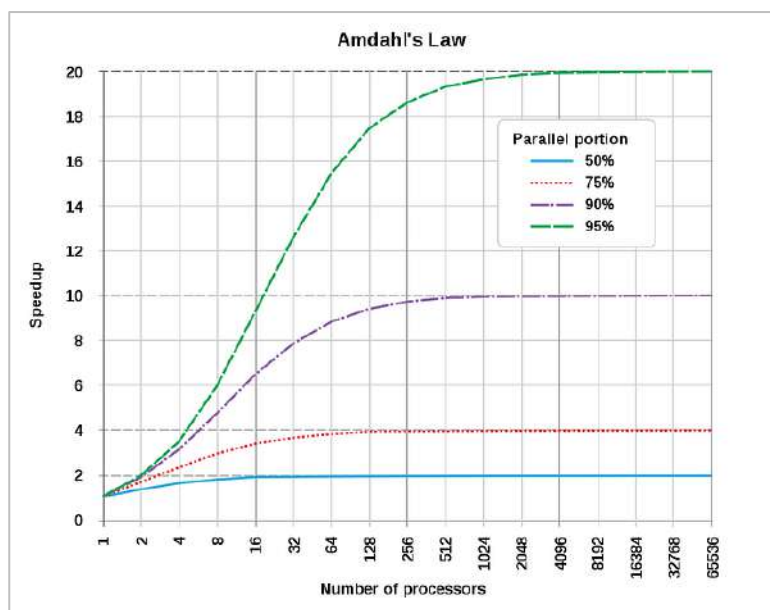
ნახ. 15.2. სიჩქარის გრაფიკი

ცხრილი 15.1-ში მოცემული მონაცემების მიხედვით აგებული ეფექტურობის მაჩვენებლის გრაფიკი მოცემულია ნახატზე 15.3.



ნახ. 15.3. ეფექტურობის გრაფიკი

პროგრამის პარალელური შესრულების სიჩქარე/დაჩქარება დამოკიდებულია არა პროცესორების რაოდენობაზე, არამედ პროგრამის თანამიმდევრული შესრულების ნაწილის ზომაზე (ნახ. 15.4.).



ნახ. 15.4. პარალელური შესრულების სიჩქარის დამოკიდებულება

პროცესორების რაოდენობაზე

თუ ამოცანის 90% (ანუ $P=0.9$) შეიძლება დაექვემდებაროს დაპარელებას, მაშინ, თუნდაც ძალიან დიდი რაოდენობის პროცესორების გამოყენებისას ($N \rightarrow \infty$), მაქსიმალური შესაძლებელი სიჩქარე იქნება:

$$S(\infty) = \frac{1}{(1 - 0.9) + \frac{0.9}{\infty}} = \frac{1}{0.1} = 10$$

ანუ ამოცანის მაქსიმალური აჩქარება 10-ჯერ მეტია, რადგან თანამიმდევრული ნაწილი (10%) ვერ პარალელდება. ამდენის კანონი მიუთითებს, რომ პარალელური გამოთვლების პოტენციალი სერიოზულად შეზღუდულია თანამიმდევრული ნაწილის პროპორციულად, ამიტომ, პარალელური ალგორითმების შემუშავებისას კარგი იქნება ამ ნაწილის მინიმიზაცია, რათა მიღწეული იქნას მაქსიმალური სიჩქარის გაზრდა.

დანახარჯების ეფექტურობა (Cost Efficiency). დანახარჯების ეფექტურობა ნიშნავს რესურსების (ფულის, დროის, ენერგიის) მაქსიმალურად ეკონომიურ გამოყენებას ისე, რომ მიღებული შედეგი იყოს მაქსიმალური და ხარჯები - მინიმალური. ეს კონცეფცია ხშირად გამოიყენება ბიზნესში, პროექტების მართვასა და რესურსების განაწილებაში, რათა უზრუნველ იქნას ოპტიმალური ბალანსი ხარჯებსა და შედეგებს შორის.

დანახარჯების ეფექტურობის მთავარი პრინციპებია:

1. მინიმალური ხარჯები მაქსიმალური სარგებლის მისაღებად. სასურველია მინიმალური რესურსებით მივაღწიოთ მაქსიმალურ შედეგს.
2. გამოსასვლელის ოპტიმიზაცია. გამოთვლები ხორციელდება იმისთვის, რომ შედეგები სრულად შეესაბამებოდეს მიზნებს, ხოლო დანახარჯები დაყვანილ იქნას მინიმუმამდე.

3. ანალიზი და გაუმჯობესება. მიმდინარე ხარჯების მუდმივი მონიტორინგი და მათი ეფექტურობის გაუმჯობესება.

პრეფიქსების ეფექტური პარალელური დამუშავება (Efficient Parallel Processing of Prefixes). მონაცემთა სტრუქტურებში პრეფიქსების ეფექტური პარალელური დამუშავება ეხება პრეფიქსების (პირველი რამდენიმე ელემენტის) დამუშავების ეფექტურ დაპარელებას, რაც მნიშვნელოვნად ზრდის შესრულების სისწრაფეს, განსაკუთრებით დიდი მონაცემების დამუშავებისას.

პრეფიქსების ეფექტური პარალელური დამუშავების ცნობილი ამოცანაა პრეფიქსების ჯამის ამოცანა. პარალელური გამოთვლისას შესაძლებელია შეიქმნას დამოუკიდებელი ნაკადები, რომლის დახმარებითაც ამოცანას გადაწყვეტა მოხდება ბევრად უფრო სწრაფად.

განვიხილოთ პრეფიქსების ჯამის გამოსათვლელი კლასიკური ალგორითმი:

1. **შეტანა.** მოცემული გვაქვს მასივი $A[n]$. ყოველი ინდექსისთვის მისი წინამორბედი ინდექსების ჯამი გამოისახება ფორმულით:

$$B[i] = A[0] + A[1] + A[2] + \dots + A[i]$$

2. **დამუშავება.** ყოველი ინდექსი პარალელურად ამუშავებს მეზობელი ინდექსების კოორდინატებს.

პრეფიქსების ეფექტური დაპარელება შეიძლება განვიხილოთ ორ ეტაპად:

1. Task Parallelism. ყოველი პროცესორი ამუშავებს სხვადასხვა ნაწილს პარალელურად;

2. Prefix Sum Optimization. ხდება ნაწილობრივი ჯამების განახლება, რაც საშუალებას იძლევა, რომ დავაჩქაროთ ალგორითმი.

განვიხილოთ პრეფიქსების პარალელური დამუშავების ალგორითმის მუშაობის მაგალითი:

ვთქვათ მოცემული გვაქვს მასივი $A = [3, 1, 4, 0, 2]$, წარმოვადგინოთ ვიზუალურად, თუ როგორ მიმდინარეობს ალგორითმის თითოეული ნაბიჯის შესრულება პარალელურად:

1. მონაცემების შეტანა. ყველა ელემენტი თავის ადგილზე რჩება.

$$A[0] = 3 \quad A[1] = 1 \quad A[2] = 4 \quad A[3] = 0 \quad A[4] = 2$$

2. პირველი ბიჯი. მეზობელი ინდექსების ჯამი. ყველა ელემენტი პარალელურად იმახსოვრებს ინდექსების ჯამის შედეგს:

$$B[1] = A[0] + A[1] = 3 + 1 = 4$$

$$B[2] = A[1] + A[2] = 1 + 4 = 5$$

$$B[3] = A[2] + A[3] = 4 + 0 = 4$$

$$B[4] = A[3] + A[4] = 0 + 2 = 2$$

2. მეორე ბიჯი. ორი მეზობელი ელემენტის წინასწარი ჯამი. ყველა ელემენტი იყენებს წინა მეზობელ ელემენტებთან მიღებულ ჯამს:

$$B[2] = B[0] + B[2] = 3 + 5 = 8$$

$$B[3] = B[1] + B[3] = 4 + 4 = 8$$

$$B[4] = B[2] + B[4] = 5 + 2 = 7$$

4. მესამე ბიჯი. ოთხი მეზობლის წინასწარი ჯამი. ამ ეტაპზე თითოეული ელემენტი იყენებს ყველა წინამორბედის ჯამს, რომ სრულად გამოთვალოს საბოლოო შედეგი.

$$\text{საბოლოოდ მივიღებთ: } B[0] = 3 \quad B[1] = 4 \quad B[2] = 8 \quad B[3] = 8 \quad B[4] = 7$$

სიმეტრიული დარღვევა (Symmetric Perturbation). სიმეტრიული დარღვევა არის მეთოდი, რომელიც გამოიყენება ოპტიმიზაციის ან სისტემების ანალიზის დროს, როდესაც დარღვევები ხდება სისტემის სხვადასხვა ნაწილში თანაბრად ან სიმეტრიულად. ეს მიდგომა განსაკუთრებით მნიშვნელოვანია ისეთი ამოცანებისთვის, სადაც საჭიროა სტაბილურობისა და ბალანსის შენარჩუნება დარღვევების პირობებში.

პროგნოზირება. პროგნოზი (ბერძნული სიტყვა *πρόγνωση* და განჭვრეტას, წინასწარმეტყველებას ნიშნავს) - არის მეცნიერული, გონივრული, საღი განსჯა მომავალში ობიექტის მდგომარეობების შესახებ ან მისი განხორციელების ალტერნატიულ გზებზე და ვადებზე.

პროგნოზირება - ეს არის პროგნოზის შემუშავება. ვიწრო გაგებით - რომელიმე პროცესის შემდგომი განვითარების კონკრეტულ პერსპექტივების შესახებ სპეციალური სამეცნიერო კვლევა. ნებისმიერი პროგნოზის სიზუსტე განპირობებულია:

- „ჭეშმარიტი“ (ვერიფიცირებული ცნობილი ცდომილებით) საწყისი მონაცემების მოცულობით და მათი შეცვრების პერიოდით;
- არავერიფიცირებული საწყისი მონაცემების მოცულობით და მათი შეცვრების პერიოდით;
- საპროგნოზო ობიექტის თვისებებით და პროგნოზის სუბიექტზე მისი ურთიერთქმედების სისტემით;
- პროგნოზირების მეთოდურობით და მოდელურობით.

პირობითად პროგნოზები იყოფიან:

ვადების მიხედვით: მოკლე, საშუალო, გრძელ და შორეულ ვადიან პროგნოზებად;

მასშტაბის მიხედვით: კერძო, ადგილობრივ, რეგიონალურ, უწყებრივ, ქვეყნის და მსოფლიო(გლობალურ) პროგნოზებად.

პასუხისმგებლობის (ავტორობის) მიხედვით: პირადი, საწარმოს (ორგანიზაციის) დონის და სახელმწიფო ორგანოების დონის.

პროგნოზირების ძირითად მეთოდებად ითვლება:

- სტატისტიკური მეთოდები;
- საექსპერტო შეფასებები (მაგალითად, დელფის მეთოდი);
- მოდელირების მეთოდები;
- ინტუიტიური.

პროგნოზირების ტერმინები.

პროგნოზირებაში გვაქვს შემდეგი სახის ტერმინები: პროგნოზირების ხერხი; პროგნოზირების მოდელი; პროგნოზირების მეთოდი; პროგნოზირების მეთოდოლოგია; საპროგნოზო სისტემა; საპროგნოზო ვარიანტი; პროგნოზირების ობიექტი და მისი მახასიათებლები; პროგნოზირების ობიექტის ცვლადი.

პროგნოზირების სიზუსტე - პროგნოზის სანდოობის ინტერვალის შეფასება (მისი შესრულების მოცემული ალბათობისათვის);

პროგნოზის საიმედოობა - მოცემული სანდოობის ინტერვალში პროგნოზის შესრულების ალბათობა;

პროგნოზის შეცდომა - ეს ობიექტის ნამდვილ მდგომარეობიდან პროგნოზის გადახრის სიდიდეა.

პროგნოზის ვერიფიკაცია - პროგნოზის სანდოობისა და სიზუსტის ან დასაბუთების შეფასება.

პირობითად განვიხილავთ ორი ტიპის პროგნოზებს:

1. სტატიკური - როდესაც გამოსაცნობია როდის, სად და რა მახასიათებლებით მოხდება საძიებო მოვლენა (მიწისძვრა, ღვარცოფები, ზვავები, ომის დაწყების პროგნოზი, ფიზიკური მოვლენები, სოციალური მოვლენები).

2. დინამიკური - როდესაც ყოველ დროით ინტერვალზე საჭიროა გარკვეული სიდიდის პროგნოზირება (ყოველდღიური ვალუტის პროგნოზი, ნავთობის ფასის პროგნოზი, ყოველთვიურად არსებობის მინიმუმის პროგნოზი, ყოველწლიური შემოსავლების პროგნოზი, გრიპის გავრცელების მასშტაბების პროგნოზი და ა.შ.).

სტატიკური პროგნოზებია:

1. დროის მიხედვით. საჭიროა დროის იმ მონაკვეთის პროგნოზირება როდესაც მოსალოდნელია მოცემული მოვლენის მოხდენა გარკვეულ ადგილას (მიწისძვრა, ვირუსის წარმოშობა, საომარი კონფლიქტის დაწყება და ა.შ.). აქ დიდი მნიშვნელობა ენიჭება დროის ინტერვალს. მიწისძვრებისთვის ყველაზე გამოყენებადია მოკლევადიანი პროგნოზი - სად (ეპიცენტრის ან რეგიონის დასახელება) და როდის (გარკვეულ დღეს ან 24 საათიანი ინტერვალში) და რა სიმძლავრის (რიხტერის სკალით => 5) მიწისძვრაა მოსალოდნელი.

2. ანალოგიურად ლოკაციის მიხედვით. საჭიროა იმ ლოკაციის განსაზღვრა (გარკვეული რადიუსით) როდესაც მოცემულ დროის ინტერვალში მოსალოდნელია საძიებო მოვლენის მოხდენა.

3. მახასიათებლების მიხედვით. რა მახასიათებლებითაა მოსალოდნელი მოცემული მოვლენა გარკვეულ დროს და ადგილას (მაგალითად ძლიერი კორონავირუსის პირველი გამოვლინება).

ყოველი საპროგნოზო მოდელი პროგნოზის გასაკეთებლად იყენებს გარკვეულ კანონზომიერებებს დამოკიდებულს გარკვეული მონაცემების მიმდევრობის „ქცევიდან“

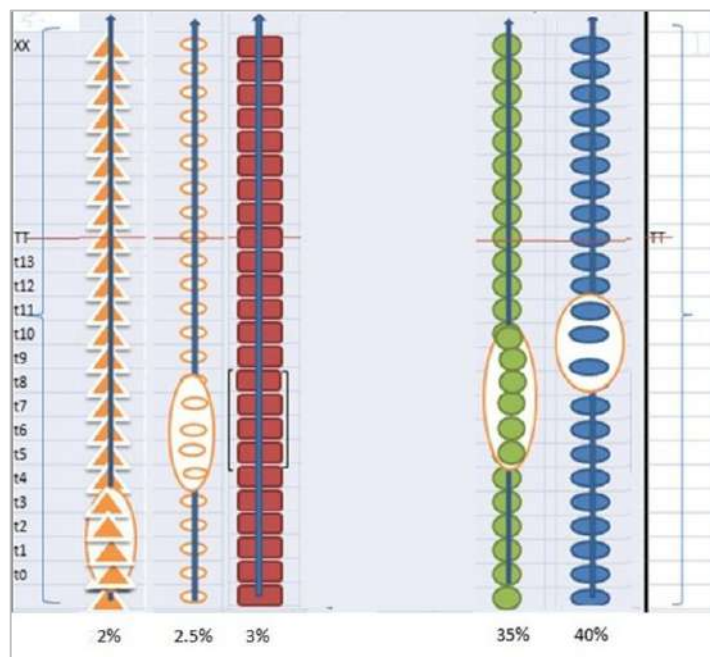
გამომდინარე. მათ წინამაცნეებს (ან წინამორბედებს) უწოდებენ. იქიდან გამომდინარე, რომ მოვლენა არც თუ ისე ხშირად ხდება, ხოლო მოდელები ბევრად უფრო ხშირად იძლევიან პროგნოზს, რომელიც არ სრულდება, შემოდის პროგნოზის გამართლების ალბათობა, რომლის გამოთვლისათვის საჭიროა ზუსტი მონაცემების დიდი რაოდენობა.

მაგალითად, თუ გვაქვს 10 წლის მონაცემები და მოცემული წინამორბედებით პროგნოზირებული იყო მოვლენის K-ჯერ მოხდენა და სინამდვილეში მოხდა M-ჯერ, მაშინ ასეთი მოდელისათვის საპროგნოზო ალბათობა ტოლია M/K .

მაგალითად, საქართველოში ელექტრომაგნიტური რეზონანსების წინამორბედებით 10 წლის განმავლობაში თუ პროგნოზირებული იქნა 100 მიწისძვრა და სინამდვილეში მოხდა 3, ალბათობა ტოლი იქნება $3/100 = 0.03$ ანუ 3%.

როგორც აკადემიკოს ზავიალოვს მიაჩნია, საშუალოდ ყველა არსებული პროგნოზების სიზუსტე წარმოადგენს ასეთივე სიდიდეს.

ამ სიზუსტის გაზრდისათვის საჭიროა ისეთი წყვილების, სამეულების და ა.შ. პოვნა (მოცემული რეგიონებისათვის ამ მიზნისათვის გამოყოფილი თანხიდან გამომდინარე), რომლებიც ბევრად გააუმჯობესებენ ამ ალბათობას (ნახ. 15.5.).

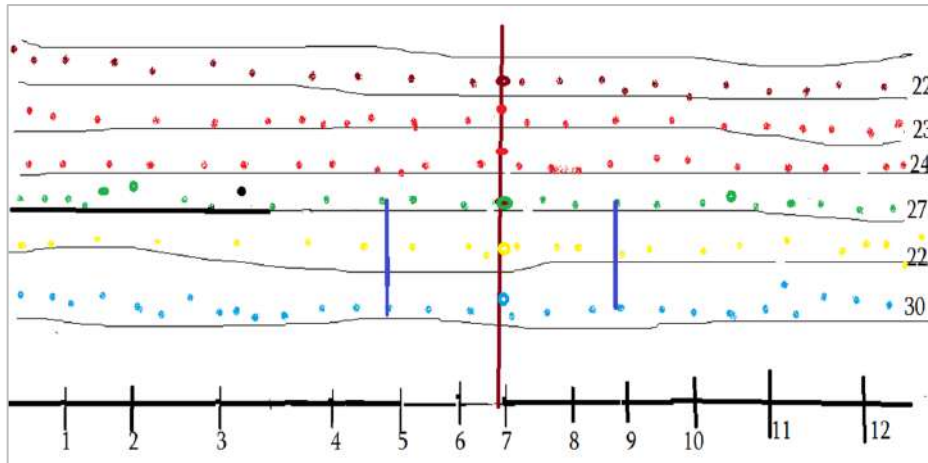


ნახ. 15.5. ალბათობების ზრდის მიხედვით დალაგებული პროგნოზირების სისტემები

არსებული წინამორბედებიდან იღებენ მხოლოდ „აუცილებელ“ წინამორბედებს (თუ მოვლენა მოხდა - ამ წინამორბედებით აუცილებლად იყო პროგნოზირებული).

შემდეგ ეძებენ ამ წინამორბედებიდან ისეთ წყვილებს რომელთა პროგნოზების ერთმანეთზე დამთხვევა მინიმალურია.

ნახატზე 15.6. „მწვანე“ და „ლურჯი“ წინამორბედებით პროგნოზები ცალ-ცალკე ყველაზე დაბალია $1/27$ და $1/30$, მაგრამ წყვილში ეს ალბათობა უკვე $1/3$ -დეა.

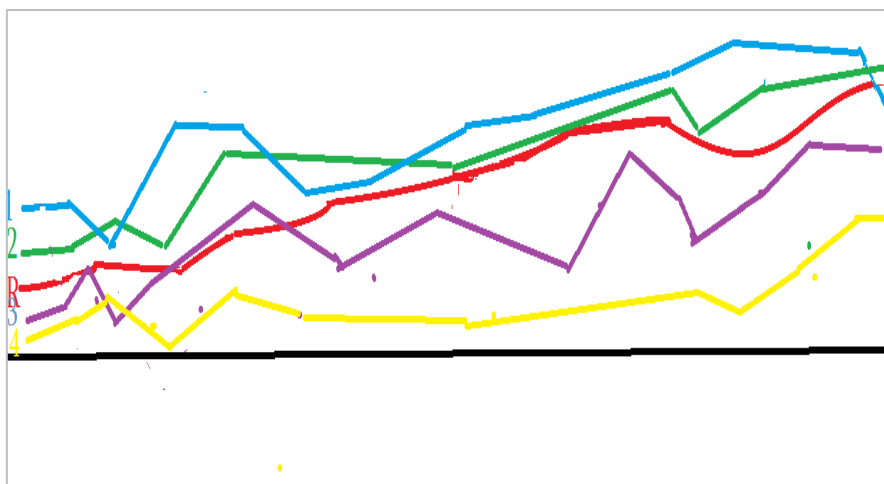


ნახ. 15.6. წყვილები, რომელთა პროგნოზების ერთმანეთზე დამთხვევა მინიმალურია „მწვანე“ და „ლურჯი“ წინამორბედებით პროგნოზები ცალ-ცალკე ყველაზე დაბალია 1/27 და 1/30, მაგრამ წყვილში ეს ალბათობა უკვე 1/3-დე ადის.

დინამიკური პროგნოზირება. დინამიკურია პროგნოზირება, როდესაც ყოველ დროით ინტერვალზე საჭიროა გარკვეული სიდიდის პროგნოზირება (ნახ. 15.7).

მაგალითად, საჭიროა ყოველდღიურად ლარის კურსის პროგნოზირება, ბენზინის კურსის პროგნოზირება, პანდემიის დროს დაავადებულ ადამიანთა რაოდენობის პროგნოზირება, საომარი მოქმედების დროს შესაძლო მსხვერპლის რაოდენობის პროგნოზირება და ა.შ.

აქ არსებული პროგნოზირების სისტემები მუდმივად იძლევიან გარკვეულ რიცხვით ან ტექსტურ შედეგს, რომლებიც შეიძლება ან არ დაემთხვეს რეალურ მნიშვნელობას, მაგრამ ცდომილება (განსხვავება) შეიძლება მეტად მცირე იყოს. თუ არსებობს რამოდენიმე პროგნოზირების სისტემა ცხადია ხდება მათი პროგნოზების საშუალო ცდომილების დათვლა. მაგრამ პერსპექტიულად მიიჩნევა ისეთი პროგნოზირების სისტემების წყვილების (ან სამეულების და ა.შ.) ძებნა, რომელთა მნიშვნელობების საშუალო არითმეტიკული ან საშუალო გეომეტრიული ან რომელიმე სხვა ფუნქციით დათვლილი მეტად ახლოს იქნება რეალურ მნიშვნელობასთან.



ნახ. 15.7. დინამიკური პროგნოზირება

მოცემული ალგორითმების პრობლემები:

სტატისტიკური პროგნოზირებისას საჭიროა დიდი რაოდენობის ზუსტი მონაცემები, რომლებიც მიიღება მრავალწლიანი დაკვირვების შედეგად. სამწუხაროდ ასეთი მონაცემები ძალიან მცირეა, მიღებულია ბოლო ათწლეულებში და ყველა არ არის საჯარო.

ბევრად უკეთ არის საქმე დინამიკური პროგნოზირებისას, აქ ბევრი მონაცემი საჯაროა ან ადვილადაა მოპოვებადი. აქ პრობლემა დადის დიდი რაოდენობის მონაცემების დამუშავებასთან, რომელიც მოითხოვს სუპერკომპიუტერებსა და პარალელურ ალგორითმებს.

ბოლო მიწისძვრები <https://www.emsc-csem.org/#2>

გამოყენებული ლიტერატურა

1. ჯანელიძე გ. ალგორითმიზაციის საფუძვლები, თბილისი, „ტექნიკური უნივერსიტეტი“, 2007. ISBN: 9994095110. ბიბლიოთეკის ინდექსი: 681.142(02)/135
2. ჰ. მელაძე, თ. დავითაშვილი, ვ. კვარაცხელია, მ. მენთეშაშვილი, ზ. წვერაძე. ალგორითმების თეორიის საწყისები, 326 გვ., ISBN 978-9941-20-350-3, „ტექნიკური უნივერსიტეტი“, 2013. კატერი 004.421(02)/1
3. თ. ბახტაძე, ა. კობიაშვილი. მონაცემთა სტრუქტურები და დაპროგრამება. თბილისი, სტუ, 2019, 230 გვერდი, 004.65(02)/9
4. ლ. გაჩეჩილაძე, ალგორითმიზაციის საწყისები, რეც.: გ. სურგულაძე, დ. კაპანაძე. თბილისი. : „ IT-კონსალტინგის სამეცნ. ცენტრი.“, 2019 . გვ. 85, Date: 2019
5. ნინო ბერაია, ლალი ტოკაძე. “Введение в структуры данных и алгоритмы” (შესავალი მონაცემთა სტრუქტურებში და აგორითმებში). სახელმძღვანელო, თბილისი: „ტექნიკური უნივერსიტეტი“, 2024, - გვ. 148, ISBN 978-9941-512-49-0 (PDF).
6. გამეზარდაშვილი ზაზა, 100000000 ამოცანა პროგრამირების შემსწავლელთათვის, ISBN-9789941099199, გვ. 272, თბილისი, 2017
7. თენგიზ ხინიკაძე, კომპიუტერის არითმეტიკული და ლოგიკური საფუძვლები, ბათუმის შოთა რუსთაველის სახელმწიფო უნივერსიტეტი, დამტკიცებულია რსუ აკადემიური საბჭოს მიერ 27.07.2012 დადგენილება №127, ბათუმი - 2012
8. თამაზ ოზგაძე, არჩილ ფრანგიშვილი, ოთარ კემულარია, მარიამ რაშოიანი. ვარიაციული მეთოდები. საგამომცემლო სახლი „ტექნიკური უნივერსიტეტი“, ISBN 978-9942-8-0957-6, 2019
9. ნანა ოდიშელიძე, დისკრეტული სტრუქტურების ამოცანების კომპიუტერული რეალიზაცია, ლექციების კონსპექტი, თბილისის სახელმწიფო უნივერსიტეტი, თბილისი, 2012.
10. ვალერიან კვეკელია, გულნარა კოტრიკაძე, კრიპტოგრაფიის სიმეტრიული სისტემის მეთოდები და მოდელები, საგამომცემლო სახლი „ტექნიკური უნივერსიტეტი“, 2016, ISBN 978-9941-20-653-5 (ყველა ნაწილი) ISBN 978-9941-20-654-2 (პირველი ნაწილი)
11. Jon Kleinberg, Eva Tardos, Algorithm Design. Addison Wesley, 2005
12. გელაშვილი კ. მონაცემთა სტრუქტურები და ალგორითმები. თბილისის სახელმწიფო უნივერსიტეტი, 2010. <https://maglivi.wordpress.com/2010/04/12>.
13. Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. - М. : Издательский дом “Вильямс”, 2011. - 1296 с.
14. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 16.3, pp.
15. <https://ka.khanacademy.org/computing/computer-science/algorithms#merge-sort>
16. <https://kobage.info/Articles/Teaching>
17. Фофанов О.Б. Алгоритмы и структуры данных: учебное пособие/ О.Б. Фофанов; Томский политехнический университет. - Томск: Изд-во Томского политехнического университета, 2014. - 126 с.

18. Гуфельд И. Л., Новоселов О. Н. Сейсмический процесс в зоне субдукции. Мониторинг фонового режима. Москва: Изд-во МГУЛ, 2014. 100 с.
19. Гульельми А. В. Форшоки и афтершоки сильных землетрясений в свете теории катастроф. Успехи физ. наук. 2015. Т. 185. № 4. с. 415 - 429.
20. Шуман В.Н. Сейсмический процесс и современные мониторинговые системы. Геофиз. журн. 2014.Т. 36. № 4. с. 50 - 64.
21. M.K.Kachakhidze, N.K.Kachakhidze. Prediction Capabilities of VLF/LF Emission as the Main Precursor of Earthquake. GESJ: Physics 2015 | No.1(13) ISSN 1512-1461. pp.80-89.
22. Завьялов А.Д. Прогноз землетрясений: состояние проблемы и пути решения, в журнале Земля и вселенная, 2018 № 5, с. 66-79
23. В.Г. Брюков. Как предсказать курс доллара. Библиотека центра исследований платежных систем и расчетов. Москва, 2017.

რედაქტორი ბ. ცხადაძე

გადაეცა წარმოებას 18.07.2025. ხელმოწერილია დასაბეჭდად 20.11.2025. ქალაქის
ზომა 60X84 1/8. პირობითი ნაბეჭდი ქალაქი 22. №3768.

საგამომცემლო სახლი „ტექნიკური უნივერსიტეტი“, თბილისი, კოსტავას 77

