Нино Берая, Лали Токадзе

ВВЕДЕНИЕ В СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ

Издательский дом
”Технический университет”

ГРУЗИНСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Нино Берая, Лали Токадзе

ВВЕДЕНИЕ В СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ

Утверждено Редакционно-издательским советом ГТУ в качестве учебника, 24.07.2024, протокол №1

Тбилиси
2024

УДК 004

Учебник «Введение в структуры данных и алгоритмы» представляет собой комплексное руководство для студентов первого курса и закладывает понимание начальных основных понятий и принципов дисциплины «Структуры данных и алгоритмы», что позволит им в дальнейшем выбирать более эффективные и оптимальные подходы к решению задач.

В учебнике рассматривается понятие алгоритма и его роль в решении простых задач. Учебник охватывает описание различных алгоритмов, включая итеративные и рекурсивные, а также рассматривает элементарные и сложные структуры данных, такие как массивы, стеки, очереди, деки, связные списки, графы, префиксные деревья и хеш-таблицы.

Также освещены вопросы, связанные с оценкой и описанием компьютерных алгоритмов, время работы алгоритмов, инварианты циклов и рекурсия.

Большое внимание уделено алгоритмам сортировки и поиска, включая бинарный поиск, сортировку выбором, сортировку вставкой, сортировку слиянием, быструю сортировку и сортировку пузырьком. Учебник также затрагивает вопросы топологической сортировки и нахождения кратчайшего пути в ориентированных графах.

Рассматриваются алгоритмы на строках, задачи классов P и NP, матрицы и действия с ними, основы криптографии, сжатие данных и другие.

Учебник предназначен для студентов факультета информатики и систем управления, а также для читателей, интересующихся информатикой и программированием.

Рецензенты: профессор Лили Петриашвили,
ассоцированный профессор Гулнара Джанелидзе

ISBN 978-9941-512-49-0 (PDF)
http://www.gtu.ge

Verba volant,
scripta manent

Оглавление

Ответ на вопрос, почему же необходимо изучать такой учебный курс, как алгоритмы и структуры данных сегодня очевиден – ни в одной из отраслей информатики, каждая из которых развивается невероятно стремительно, серьезно и эффективно работать сегодня без таких знаний просто невозможно.

Например, в сегодняшних реалиях в условиях информационных войн крайне актуальны вопросы информационной безопасности, включающие в себя, в том числе, и шифрование с использованием открытого ключа, основанного на соответствующих теоретико-числовых алгоритмах.

Если же мы обратимся к протоколам маршрутизации в компьютерных сетях, то обнаружим, что и они опираются на классические алгоритмы поиска кратчайших путей.

Не обходится без геометрических алгоритмов и вычислительные примитивы, используемые в компьютерной графике.

Широко используются алгоритмы динамического программирования в биоинформатике, например, для выявления сходства геномов.

Даже в любой из баз данных индексация, от которой зависит эффективность запросов в ней, неизменно связана со специальными алгоритмами, которые определяют как создается и хранится индекс. Это и B-дерево, и хеш-индекс, и многие другие.

Не обойтись без знаний соответствующих алгоритмов и при решении задач искусственного интеллекта и в машинном обучении.

Невозможно переоценить знание алгоритмов и структур данных разработчиками программного обеспечения, поскольку именно это дает возможность управлять огромными объемами данных и создавать действительно высокоэффективные, оптимизированные, быстрореализуемые приложения.

И это лишь небольшой круг задач и направлений, решение которых невозможно без фундаментального знания алгоритмов и структур данных.

Не обойтись сегодня без знания алгоритмов и структур данных и людям, которые заинтересованы в карьерном росте, поскольку во многих серьезных компаниях эти вопросы неизменно включены в собеседования при приеме на работу. Чем сложнее проекты, реализуемые кампанией, тем серьезнее подходят к знанию подобных вопросов. Предпочтение несомненно отдается кандидатам, которые способны на нахождения максимально оптимального решения задачи.

И, наконец, знание алгоритмов улучшают не только навыки программирования, но и самым эффективным образом способствуют развитию алгоритмического мышления и аналитических способностей.

Постановка задачи. Понятие алгоритма на простых задачах. Описание алгоритма. Примеры итеративных и рекурсивных алгоритмов 1

Постановка задачи. Понятие алгоритма на простых задачах

Основная цель изучения данного учебного курса состоит в освоении общей техники проектирования алгоритмов, которые позволяют решать огромное множество любых алгоритмических, а не только классических задач. В определенной степени алгоритм представляет собой способ мыслить о задаче с целью поиска решения.

Одной из составляющих успеха любого программиста является способность понять чужой алгоритм, адаптировать его под свои нужды и способность на основании известных алгоритмов создать что-то свое, принципиально новое.

Итак, *алгоритм* – это последовательность действий для решения какой-либо задачи. Иными словами, алгоритм представляет собой решение задачи.

Сам процесс создания алгоритмов называется алгоритмизацией и для его выполнения необходимо четкое и всесторонне полное понимание поставленной задачи.

Процесс разработки алгоритма в общем случае заключается в разбиении задачи на последовательно выполняемые этапы. Фактически осуществляется преобразование исходных (входных) данных в результаты (выходные данные) (рис. 1).

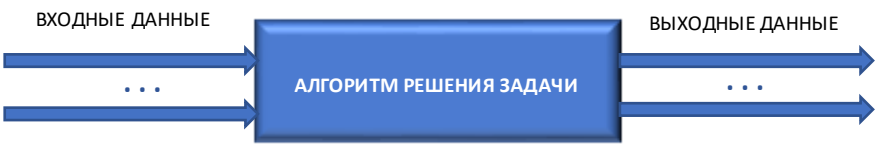


Рис.1. Схема преобразования входных данных в результаты

В повседневной жизни для решения самых простейших бытовых задач нам приходится ежедневно сталкиваться с составлением и реализацией алгоритмов. Более того, любые рецеп-

ты, инструкции, описания процессов схожи с понятием «алгоритм». Однако, алгоритмы в классическом понимании обладают множеством свойств. Остановимся на самых основных:

- ✓ *дискретность* (разрывность) – каждый алгоритм должен состоять из отдельных шагов – действий;
- ✓ *массовость* – алгоритм должен быть применим абсолютно ко всем задачам данного типа вне зависимости от исходных данных;
- ✓ *детерминированность* (*определенность*, *точность*) – каждый шаг алгоритма должен быть строго определен (недопустимы различные толкования) во всех возможных ситуациях;
- ✓ *конечность* (*результативность*) – алгоритм обязательно должен завершиться, пусть даже за большое число шагов;
- ✓ *формальность* – алгоритм должен быть составлен таким образом, чтобы его исполнитель мог, не вникая в содержание поставленной задачи, чисто формально выполнить все инструкции, прописанные в нем.

По своей конструкции алгоритмы могут быть линейными, разветвляющимися, циклическими (с предусловием и постусловием – итерационные[1]), рекурсивными[2].

Линейная конструкция предполагает, что все действия в алгоритме будут выполняться последовательно, одно за другим, причем только один раз (рис.2, *а*).

В *разветвляющейся конструкции* предполагается выбор между двумя альтернативными вариантами в зависимости от входных данных (рис.2, *б*).

Циклическая конструкция, применяемая в тех случаях, когда одно и то же действие повторяется несколько раз, значительно сокращает объем алгоритма, приводя его к макси-

[1] Итерационный цикл – это цикл, для которого число повторений тела цикла заранее неизвестно. В итерационных циклах на каждом шаге вычислений происходят последовательное приближение и проверка условия достижения искомого результата. Выход из итерационного цикла осуществляется в случае выполнения заданного условия.

[2] Рекурсивный алгоритм – это алгоритм, в описании которого прямо или косвенно содержится обращение к самому себе.

мально компактной форме. Однако, следует учитывать несколько моментов – необходимо правильно задавать параметр изменения цикла во избежание «зацикленности», а также предусматривать конструкцию, выполняющую выход из цикла. Классификация циклических конструкций представлена на рис. 3.

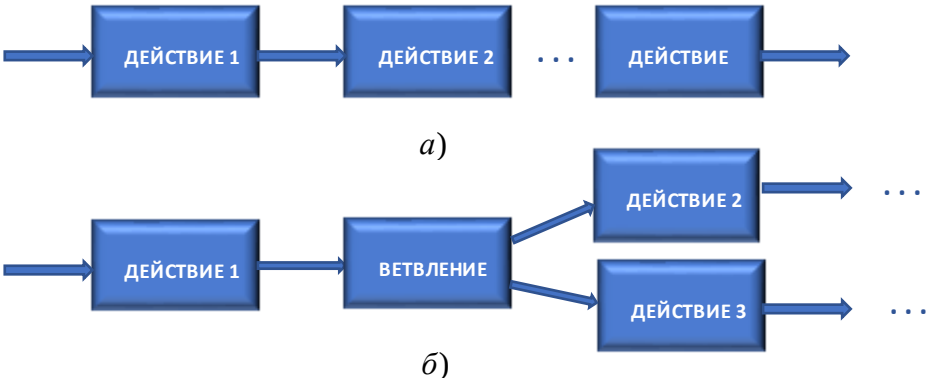


Рис. 2. Конструкция алгоритма: *а*) линейная; *б*) разветвляющаяся

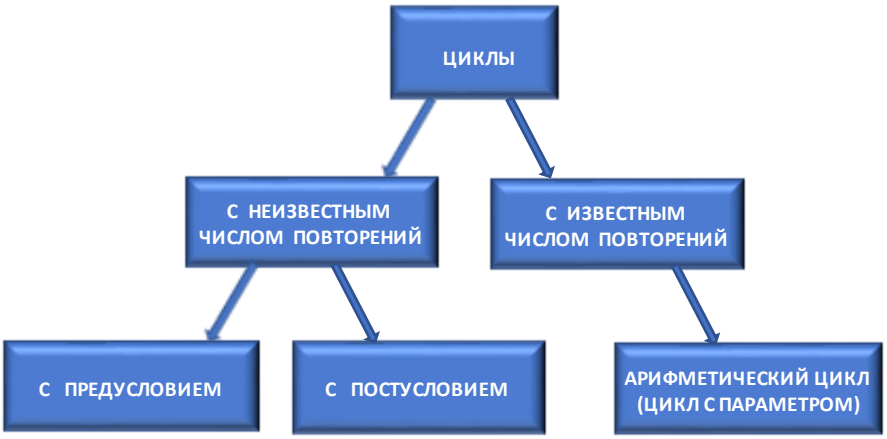


Рис. 3. Классификация циклических конструкций алгоритмов

Рекурсивная конструкция алгоритма предусматривает его обращение (прямо или косвенно) к самому себе. Обычно такие алгоритмы изящны, компактны, просты, за что, впрочем, приходится расплачиваться производительностью, ввиду неэффективного использования оперативной памяти. Построение рекурсивной конструкции алгоритма предусматривает наличие трех этапов:

- ✓ *параметризацию* – выделение параметров, которыми описываются входные данные;
- ✓ *выделение базы* – описание тривиальных (крайне простых) случаев, к которым в конечном итоге и сводится рекурсивный алгоритм;

- ✓ *декомпозицию* – общий случай выполнения алгоритма, который сводится к выполнению более упрощенных вариантов, например, таких, как уменьшение размерности данных, снижение значений алгоритма и т.д.

Описание алгоритма

Для описания алгоритмов существует несколько способов.

• *Словесное описание*. Такой алгоритм представляет собой достаточно подробное описание каких-либо действий на языке его составителя.

Подобные алгоритмы зачастую слишком многословны, не учитывают всех возможных ситуаций, а также иногда могут быть неоднозначно истолкованы.

Примером может служить инструкция по применению какого-либо прибора, кулинарный рецепт и т.п. Классический пример словесного алгоритма – *алгоритм Евклида* для нахождения общего делителя двух чисел, по указаниям которого можно найти наибольший общий делитель[1] для любой пары натуральных чисел.

Некоторые из специалистов в области информатики, такие, как например, всемирно известный Дональд Эрвин Кнут, являющийся автором множества книг по программированию и лауреатом большого количества премий в области информатики, считает алгоритм Евклида первым в истории. Здесь необходимо сделать некоторое уточнение: это утверждение, конечно же, является верным лишь с точки зрения определений, принятых уже в наше время. Тем не менее, несмотря на то, что сам Евклид жил в IV-III вв. до нашей эры, его алгоритм и в настоящее время весьма и весьма актуален, как в практическом, так и теоретическом применении и широко используется в решении ряда частных задач.

Само же слово «алгоритм», как известно, появилось гораздо позднее и неразрывно связано с именем Аль-Хорезми, персидского математика, который жил приблизительно в VIII-IX вв. нашей эры. А уже массовое использование этого слова в современном понимании приш-

[1] *Пример: для чисел 54 и 24 наибольший общий делитель равен 6.*

лось на первые десятилетия XX века нашей эры, когда начался бурный период восхода информационных технологий.

Сам алгоритм Евклида формулируется в виде отдельных шагов следующим образом.

1. Обозревая два числа *a* и *b*, переходи к следующему пункту.
2. Сравни обозреваемые числа (*a* равно *b*, *a* меньше, больше *b*) и переходи к следующему пункту.
3. Если *a* и *b* равны, то прекрати вычисление: каждое из чисел даст искомый результат. Если числа не равны, то переходи к следующему пункту.
4. Если первое число меньше второго, то переставь их местами; переходи к следующему пункту.
5. Вычитай второе число из первого, обозревай два числа: вычитаемое и остаток; переходи к пункту 2.

Еще один классический пример – *решето Эратосфена* для нахождения простых чисел[1] до заданного числа *n*.

1. Выписать подряд все целые числа от двух до *n* (2, 3, 4, …, n).
2. Пусть переменная *p* изначально равна двум — первому простому числу.
3. Считая от *p* шагами по *p*, зачеркнуть в списке все числа от 2*p* до *n* кратные *p* (то есть числа 2*p*, 3*p*, 4*p*, …).
4. Найти первое не зачеркнутое число, большее чем *p*, и присвоить значению переменной *p* это число.
5. Повторять шаги 3 и 4 до тех пор, пока *p* не станет больше, чем *n*.

Все не вычеркнутые числа будут простыми.

[1] *Натуральное число, большее единицы, называют простым, если оно имеет всего два натуральных делителя: единицу и само это число.*

Реализация этого алгоритма представлена ниже на рис. 4.

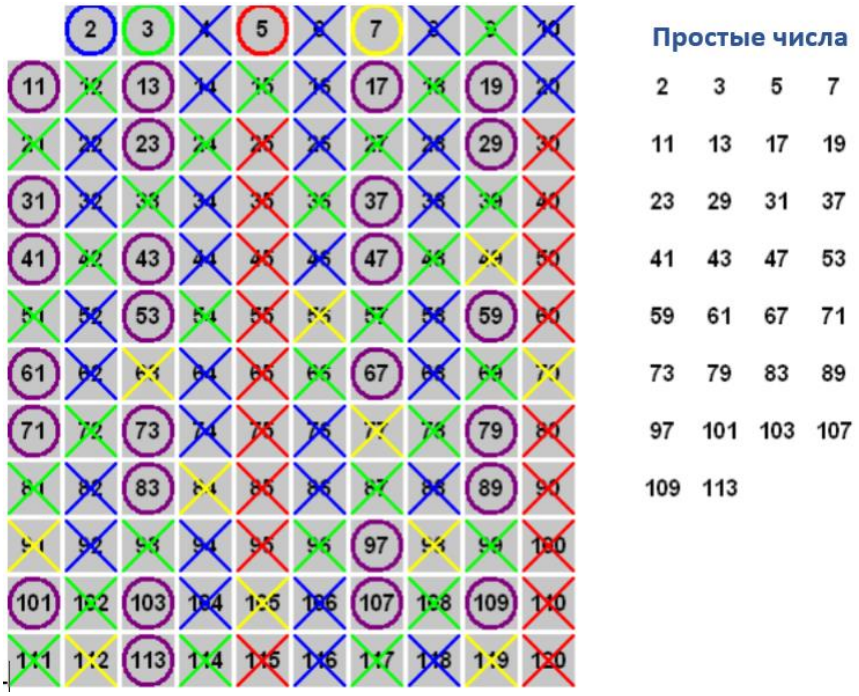


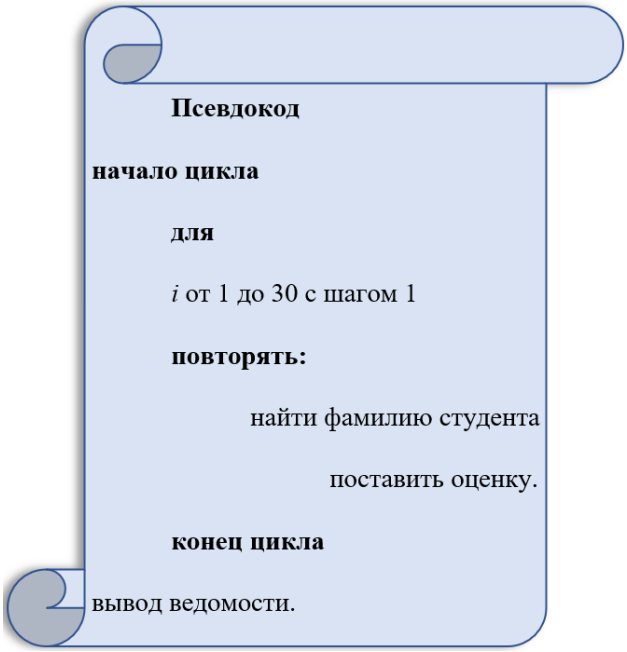
Рис. 4. Реализация алгоритма поиска нахождения простых чисел

К слову сказать, обобщенный алгоритм Евклида, равно, как и простые числа, актуальны и сегодня во многих криптографических системах, в чем мы убедимся в одном из дальнейших разделов.

- *Псевдокод*. Занимает промежуточное положение между словесным алгоритмом и формализованным языком, с помощью которого и пишутся непосредственно программы. При описании структуры алгоритма на псевдокоде используется обычный язык, как и в словесных алгоритмах, но при этом также используются математическая символика и формальные конструкции, служебные слова, приближающие псевдокод к языкам программирования.

Единого или формального определения псевдокода не существует. В связи с этим, при описании алгоритма на псевдокодах возможны различные произвольные варианты, которые отличаются как наборами служебных слов, так и базовыми конструкциями.

Например, алгоритм заполнения оценками ведомости по предмету для 30 студентов, с использованием псевдокода может иметь следующий вид (i – порядковый номер студента):



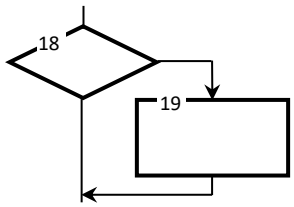
- *Графическая запись (блок-схема).* Представляет собой один из самых наглядных вариантов описания структуры алгоритма. Обладает и другим рядом преимуществ перед вышеописанными способами, такими, как лучшая концентрация на структуре алгоритма, возможности укрупнения или же, наоборот, детализация отдельных частей алгоритма и т.д.

Отдельные функции алгоритмов при этом отображаются в виде условных графических изображений – символов. Перечень условных графических символов, их наименования, форма, размеры и отображаемые функции устанавливаются в соответствии с принятыми стандартами.

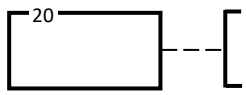
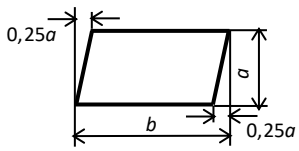
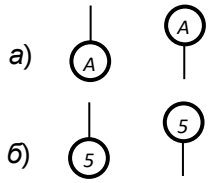
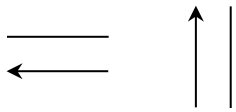
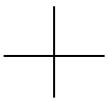
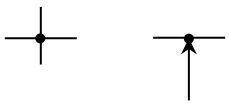
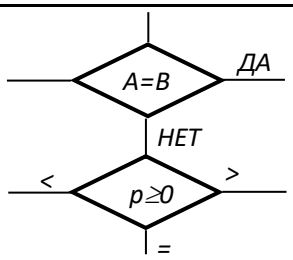
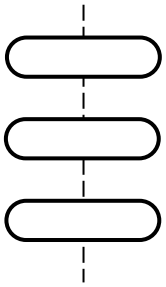
Некоторые примеры применения символов в схемах алгоритмов приведены в табл.1.

Таблица 1

Символы, используемые в графических алгоритмах

Фрагмент схемы	***Содержание изображения***	***Правила применения***
	Обозначение символов в схемах: 18, 19 – порядковые номера символов на схемах	Порядковый номер символа проставляют слева в верхней части символа (при ручном исполнении схемы)

Продолжение табл.1

Фрагмент схемы	*Содержание изображения*	*Правила применения*
	Комментарий	Применяется, если пояснение не помещается внутри символа; комментарий записывают параллельно основной надписи, помещают в свободном месте схемы алгоритма на данном листе и соединяют с поясняемым символом
	Ввод-вывод	Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод)
	Соединитель: *A*, 5 – идентификаторы соединителя в виде: а) буквы; б) цифры	При большой насыщенности схемы символами отдельные линии потока между удаленными друг от друга символами допускается обрывать. При этом в конце (начале) обрыва должен быть помещен символ «соединитель»
	Линии потока	Символ применяют для указания линии потока. Можно без стрелки, если линия направлена слева направо и сверху вниз; со стрелкой – в остальных случаях
	Излом линии потока под углом 90^0	Символ обозначает изменения направлений линий потока
	Пересечение линий потока	Символ применяют в случае пересечения двух несвязанных линий потока
	Слияние линий потока	Символ применяют в случае слияния линий потока, каждая из которых направлена к одному и тому же символу на схеме. Место слияния линий потока обозначают точкой при ручном выполнении схем
	Возможные варианты отображения решения: A=B, $p \geq 0$ – условия решений; A, B, *p* - параметры	При числе исходов не более трех признаки условия решения (например, ДА, НЕТ, =, >, <) проставляются над каждой выходящей линией потока или справа от линии потока
	Начало, прерывание и конец алгоритма или программы	Символы применяются в начале схемы алгоритма или программы, в случае прерывания ее или в конце. Внутри символа «пуск – останов» может указываться наименование действия или идентификатор программы

В качестве примера на рис.5 представлена алгоритм действий студента в зависимости от полученного на экзамене балла в виде блок-схемы.

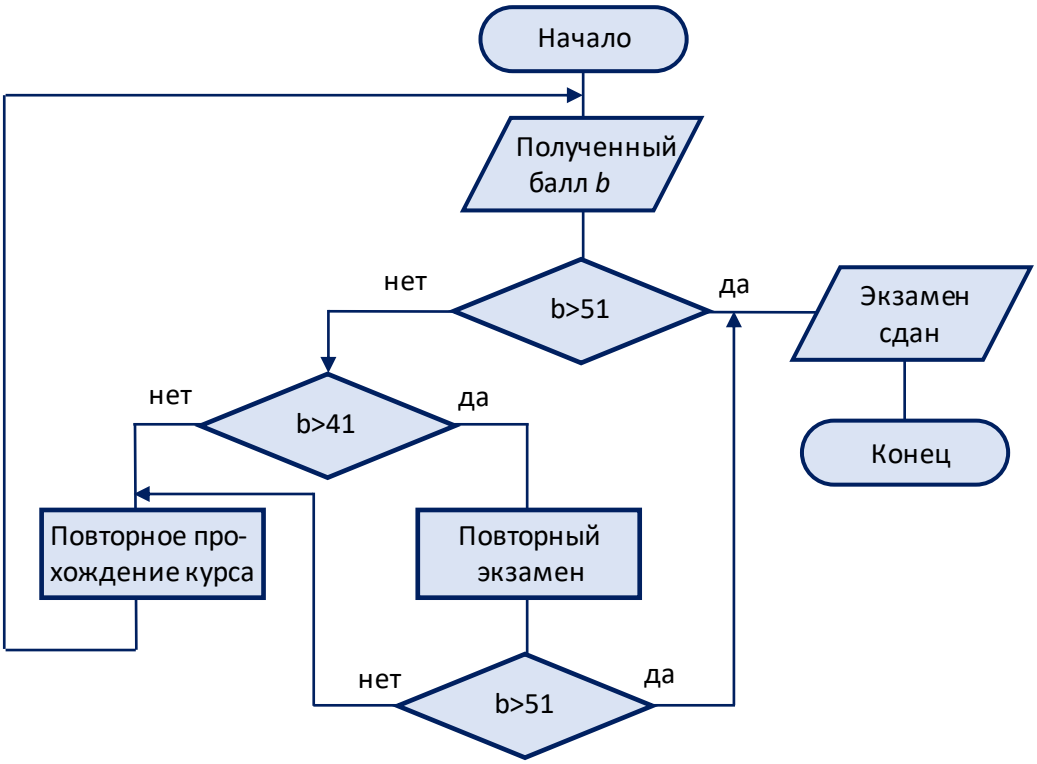


Рис.5. Пример графической записи алгоритма

Примеры итеративных и рекурсивных алгоритмов

Классическим примером самого простого итерационного и рекурсивного алгоритмов является алгоритм вычисления факториала[1].

При реализации рекурсивного варианта следует учитывать несколько условий (как минимум два): условие продолжения рекурсии (шаг) и условие ее окончания.

С учетом этого, рекуррентное соотношение, с помощью которого можно вычислить факториал числа n, будет иметь вид:

$$n! = \begin{cases} f(n) = n * f(n-1) \\ f(0) = 1 \end{cases}, \quad (1)$$

[1] Факториал числа N – это произведение всех целых чисел от 1 до N.

где с помощью первого равенства записан шаг рекурсии, а второе равенство задает условие остановки.

Например, вычисление значения $f(5)$, исходя из (1) можно представить так:

$$f(5)=5\cdot f(4)=5\cdot 4\cdot f(3)=5\cdot 4\cdot 3\cdot f(2)=5\cdot 4\cdot 3\cdot 2\cdot f(1)=5\cdot 4\cdot 3\cdot 2\cdot 1\cdot f(0)=5\cdot 4\cdot 3\cdot 2\cdot 1\cdot 1=120.$$

Фактически при рекурсивном подходе метод вызывает сам себя и каждый раз выполняется с новыми начальными значениями.

В качестве еще одного примера рекурсивно решаемой задачи можно привести классическую головоломку «Ханойские башни». Этот алгоритм, разработанный французским математиком Эдуардом Люка, призван решить проблему перестановки дисков на трех штырях.

Уникальность этой задачи заключается в ее элегантности и простоте. Не смотря на свою видимую простоту, алгоритм Ханойских башен требует от игрока умения размышлять стратегически и предвидеть последствия каждого хода.

Правила игры довольно просты. В начале игры, на одном из трех штырей находится несколько дисков различного диаметра, расположенных в порядке убывания их размеров. Главная цель – переместить все диски на другой штырь, соблюдая следующие правила:

- за один ход разрешается переместить только один диск;
- нельзя класть больший диск на меньший диск.

Эти два простых правила создают сложность и вызывают необходимость применять алгоритм для решения головоломки. Согласно алгоритму Ханойских башен, чтобы решить задачу с n дисками, необходимо выполнить следующие действия:

- переместить $(n-1)$ дисков с исходного штыря на вспомогательный штырь;
- переместить верхний диск с исходного штыря на конечный штырь;
- переместить $(n-1)$ дисков с вспомогательного штыря на конечный штырь.

Следуя этим простым правилам, можно решить головоломку Ханойских башен с любым количеством дисков, а сам алгоритм Ханойских башен, являясь рекурсивным алго-

ритмом, служит примером интеллектуального мышления и способности находить элегантные решения.

Выше уже было отмечено, что рекурсивные методы задействуют при выполнении гораздо больше ресурсов, что в отдельных случаях может привести к переполнению памяти. В основном это происходит в тех случаях, когда глубина рекурсии слишком велика.

В связи с этим рекурсия используется не слишком часто, но при решении некоторых задач является очень эффективной. Примером могут послужить алгоритмы сортировки, алгоритмы, связанные с искусственным интеллектом т. д.

Кроме того, для опытных программистов рекурсия является наиболее читабельной, поскольку в рекурсивном варианте код за счет разбиения большой задачи на маленькие выглядит более лаконично.

Для задач, которые лучше решать последовательно, выбор стоит остановить на использовании итерационных алгоритмов. В таких случаях код может быть более быстрым и экономичным с точки зрения использования памяти. Хотя, возможно, менее читабельным, особенно, если используется большое количество циклических конструкций.

В большинстве случаев же алгоритм, выполненный в виде итеративной конструкции, может быть реализован с помощью рекурсивного алгоритма и наоборот.

На рис.6 реализована один из вариантов блок-схемы итерационного алгоритма для вычисления факториала, а на рис.7 – рекурсивный вариант для той же задачи.

Таким образом, в целом, при выборе рекурсивного или итерационного варианта алгоритма следует опираться на анализ самого условия поставленной задачи, время, которое будет затрачено на ее выполнение и задействованные ресурсы, в первую очередь, память.

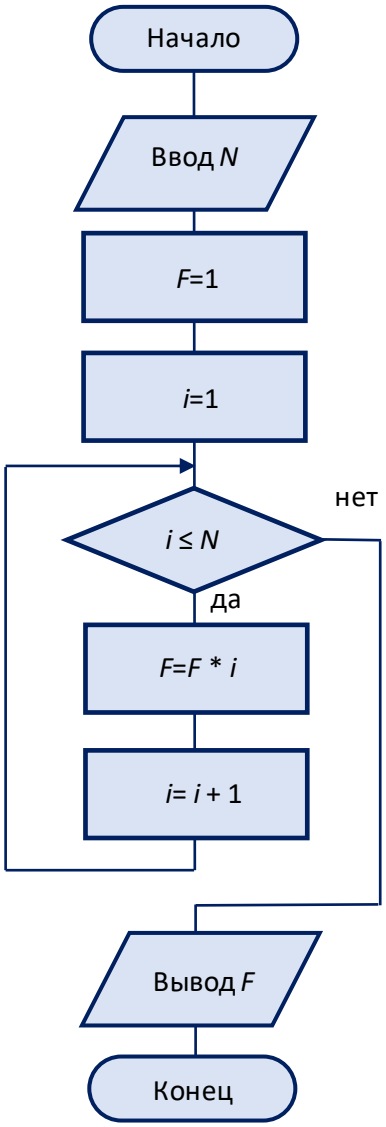


Рис. 6. Итерационная конструкция

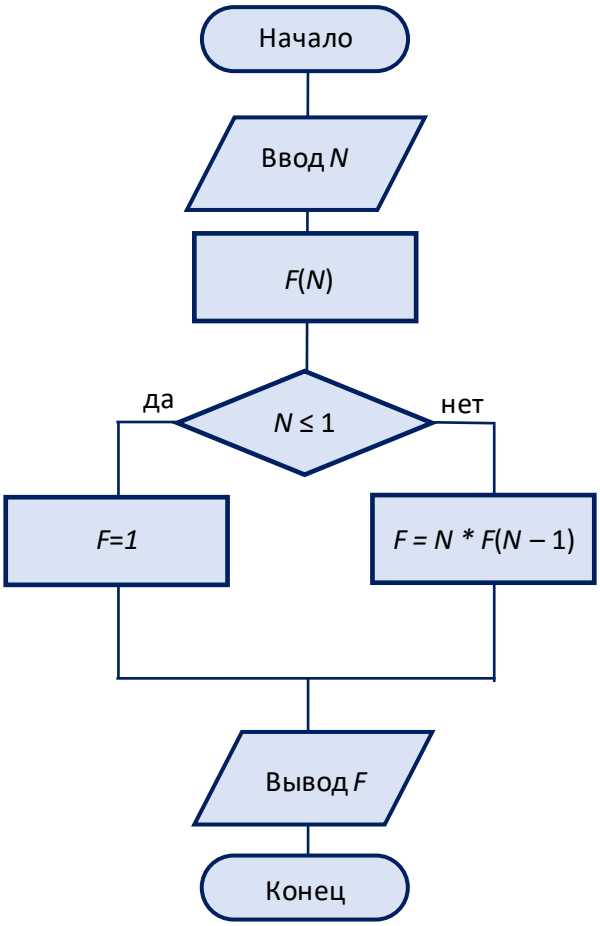


Рис. 7. Рекурсивная конструкция

Одной из важнейших составляющих при написании программ, наряду с алгоритмами и алгоритмическим языком, являются структуры данных.

Информацию, которую мы получаем в виде множества компонентов, может быть объединена в единый элемент – структуру. С помощью этой структуры есть возможность работы с отдельными компонентам, а также со всеми сразу. Иными словами, структура данных представляет собой способ организации данных.

От правильной организации данных во многом зависит возможность быстрого получения необходимой информации, что сказывается в конечном итоге и на быстродействии самого алгоритма и программы.

Существует множество различный структур данных и их классификаций по различным признакам. Один из вариантов классификации приведен на рис.8.

Рис.8. Классификация структур данных

Массив

Одной из простых и известных всем структур данных является массив. Если быть более точными, то это конструкция языка, которую можно интерпретировать как структуру данных.

Массив представляет собой пронумерованную посредством индексов последовательность произвольных чисел (рис.9).

Индексация элементов массива практически во всех языках программирования начинается с нуля, а не единицы. То есть, у первого элемента индекс 0, у второго – 1, у третьего – 2, у последнего – N-1. То есть в массиве, например, содержащем 5 элементов, индексация элементов будет от 0 до 4, в массиве из 10 элементов – от 0 до 9 и т.д.

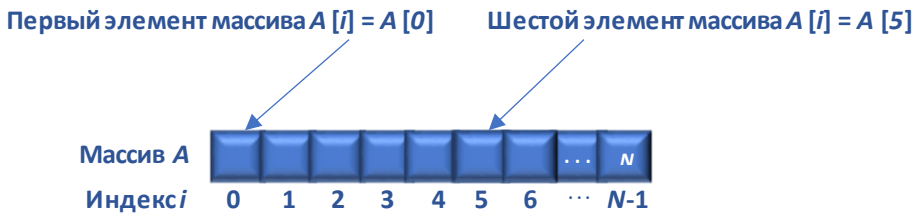


Рис.9. Графическое изображение массива

Такая структура данных как массив широко используется при обработке данных, аналитике, для хранения данных, имен, дат и т.д. Популярность массивов связана с удобством хранения данных и легким доступом по индексу к каждому из его элементов.

Массив является по сути базовой структурой данных, на основании которой можно построить и другие структуры.

Однако, не лишен он и недостатков, поскольку представляет собой непрерывный блок памяти. При добавлении элемента к уже существующему массиву придется копировать все ранее занесенные в него элементы в новую ячейку памяти, что, разумеется, негативным образом скажется на загруженности памяти. «Запасаться» дополнительными нулевыми элементами на случай возможного расширения массива также не является очень хорошей идеей, поскольку есть опасность выделения «лишних мест» и, как следствие, неэффективное использование памяти. Проблемы возникнут и при удалении элемента, если, конечно же, это не последний элемент. Во всех остальных случаях придется осуществлять сдвиг.

И все же, при небольших объемах данных преимущество использования массивов весьма существенно.

Стек

Одной из простейших линейных динамических структур данных является *стек* (*stack*). Стек представляет собой некую коллекцию данных, в которую можно помещать элементы, либо удалять из нее.

Образно стек можно представить в виде какой-то емкости, открытой только с одной стороны, в которую мы можем последовательно помещать данные. При этом реализуется принцип *last-in-first-out* (*LIFO*). В этом принципе как раз и заключается суть работы с данными в стеке – тот элемент, который был помещен последним (вершина стека), будет извлечен (в случае необходимости) и считан из него первым.

Изменить в стеке порядок расположения элементов, либо получить доступ к произвольному элементу невозможно. Если, например, возникнет необходимость достать первый из помещенных в стек элементов (дно стека), то предварительно последовательно надо будет достать все элементы, которые были помещены в стек после него.

Процедура помещения элемента в стек обозначается *push*, а процедура извлечения – *pop*.

Графически стек продемонстрирован на рис. 10.

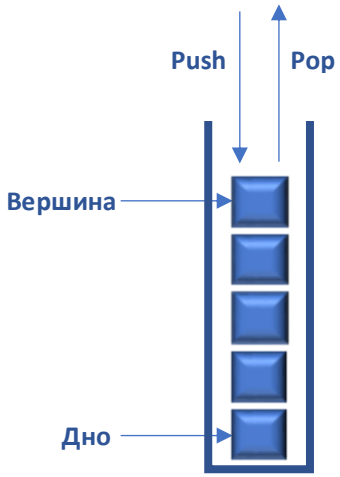


Рис. 10. Графическое изображение стека

Что касается практического применения такой структуры данных как стек, то их используют, например, в калькуляторах, действие которых основано на *обратной польской нотации*[1].

Еще одним примером может служить *отслеживание возврата*, что применяется при поиске, например, правильного пути в лабиринте – точки перемещения последовательно помещаются в стек и при выборе неправильного пути из стека удаляются те, которые привели к неправильному маршруту.

Используется стек и при обходе графа в глубину (об этом речь будет идти в 6 разделе).

Также стеки используют и при проверке правильной скобочной последовательности в сложном выражении, то есть соответствия закрывающихся скобок открывающимся. В этом случае в стек помещается открывающаяся скобка выражения и после того, как встречается закрывающаяся, обе – открывающая и соответствующая ей закрывающаяся скобка – удаляются из стека Затем процесс повторяется. Если в стеке останется одна из скобок «без пары» это будет означать, что выражение записано неправильно. Таким же образом осуществляется проверка правильности записи выражения с использованием открывающихся и закрывающихся квадратных, фигурных скобок и т.д.

Разумеется, это неполный перечень использования таких структур данных, как стек.

Очередь

Очередь (*queue*) также представляет собой элементарную линейную динамическую структуру данных, в основе которой лежит принцип *first-input-first-output* (*FIFO*).

Логически эта структура напоминает обыкновенную очередь, например, перед кассой в магазине – кто первый встал в очередь, тот первый ее и покинет.

[1] В обратной польской нотации операторы следуют за своими операндами. Например, чтобы сложить 5 и 8 вместе, необходимо записать 5 8 +, а не 5 + 8. Оператор сложения удаляет два верхних элемента 5 и 8 из стека, выполняет 5+8 операции и помещает результат, равный 13, в стек. В более сложных выражениях отпадает необходимость использования круглых скобок, что является одной из причин быстроты вычислений при использовании польской нотации в калькуляторах.

Очередь служит в программировании для обработки данных в том порядке, в котором они и поступают.

Взаимодействие с элементами очереди происходит либо с начала очереди, либо с ее конца (рис.11). В отличие от стека, действия *push* и *pop* осуществляются с противоположных концов очереди.

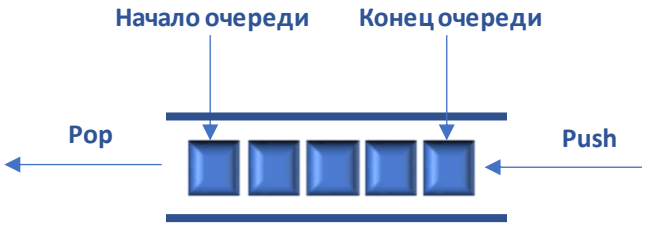


Рис.11. Графическое изображение очереди

Также, как и в случае со стеком доступ к отдельным элементам, расположенным между первым и последним элементом очереди, невозможен. Иными словами, невозможен перебор элементов очереди, добавление и удаление элементов очереди в ее середине также запрещен. Невозможно добавить элемент и в начало очереди – добавление разрешено только с конца очереди. Аналогично, удаление элемента осуществляется исключительно с начала очереди. Для просмотра доступен, как первый, так и последний элемент очереди.

Применение такой структуры данных, как очередь можно встретить при управлении потоками данных и обработке разных задач в приложениях, например, в операционных системах, различных системах обработки данных, сетевых протоколах.

Удобно использовать такую структуру данных и при поэтапном изучении всех узлов графа, в так называемом поиске в ширину (подробно этот алгоритм будет рассмотрен в разделе 6).

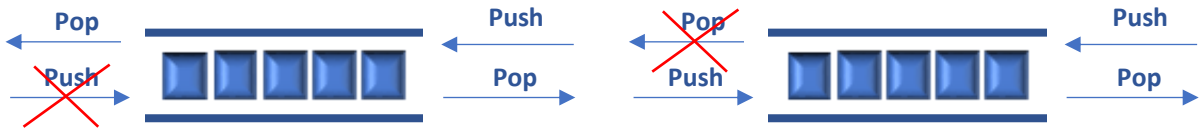
Дек

Дек (*Deque*) является также динамической линейной структурой данных и фактически представляет собой двустороннюю очередь (*Double Ended Queue*). Эта особенность дает возможность добавлять и извлекать элементы с обеих сторон (рис.12).

Добавление и извлечение элементов происходит по правилу очереди. При этом у нас есть возможность добавить элемент справа, удалить справа, добавить слева, удалить слева, а также осуществить чтение крайних элементов справа и слева.

Рис.12. Графическое изображение дека

У такой структуры можно несколько изменить функционал, введением ограничений на добавление (рис.13, а) или извлечение (рис.13, б) элементов с одной из сторон. Такой дек будет называть деком с ограниченным входом и выходом соответственно.



а) *б*)

Рис.13. Дек с ограниченным: *а*) входом; *б*) выходом

Дек представляет собой самую универсальную структуру из всех выше рассмотренных линейных структур, поскольку при помощи дополнительного функционала в виде ограничений на его начало и конец, можно смоделировать так стек, так и очередь.

Связные списки

В отличие от массива данных, к каждому элементу из которого мы можем получить доступ указанием его индекса, списки представляют собой более сложно организованную, но тем не менее, простую в реализации структуру, позволяющую легко изменять размер.

Оптимизированный вариант обыкновенного линейного списка, в котором элементы располагаются в памяти строго друг за другом, представляет собой связанный список, элементы которого в памяти могут располагаться в произвольном порядке.

Каждый отдельный элемент односвязного списка представляет собой в некотором роде контейнер, в котором содержится два поля: первое представляет собой адрес (ссылку) в памяти, второе – непосредственно информация. Адрес отдельным полем дает возможность получить информацию о следующем элементе этого списка, т.е. каждый элемент «знает» адрес следующего элемента. Кроме того, именно благодаря ссылке каждый из элементов может занимать разный объем памяти, что является невозможным в случае использования массива.

У последнего элемента этого списка значение поля «адрес» будет *Null* (рис.14).

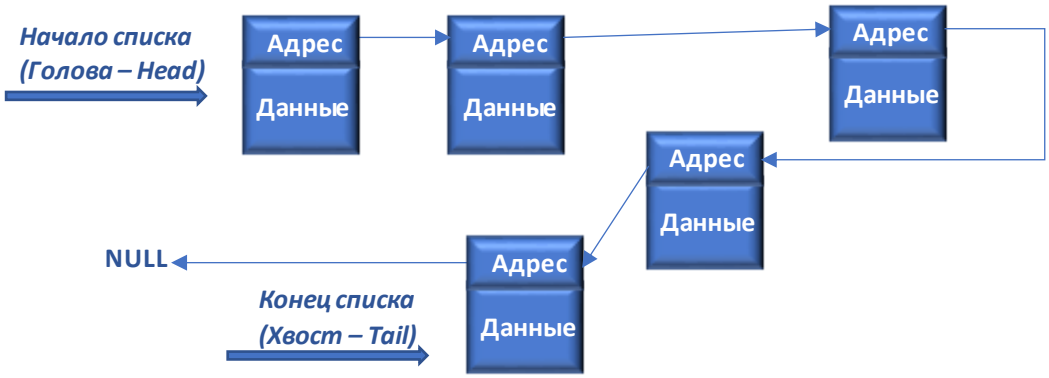


Рис.14. Графическое представление односвязного списка

Для добавления нового элемента в конец списка достаточно изменить значение последнего элемента существующего списка, изменив его с *Null* на адрес добавляемого элемента.

Удаление произвольного элемента из списка на любой из позиций также не представляет особой сложности. Для этого необходимо лишь сменить адрес предшествующего удаляемому элемента, который теперь будет указывать на следующий за удаленным элемент. Это наглядно продемонстрировано на рис.15, где удаляемый элемент перечеркнут, а пунктирная линия со стрелкой исходит из элемента, в котором следует изменить ссылку на следующий элемент (указать адрес следующего элемента). Старые связи ликвидируются – на рисунке они также перечеркнуты.

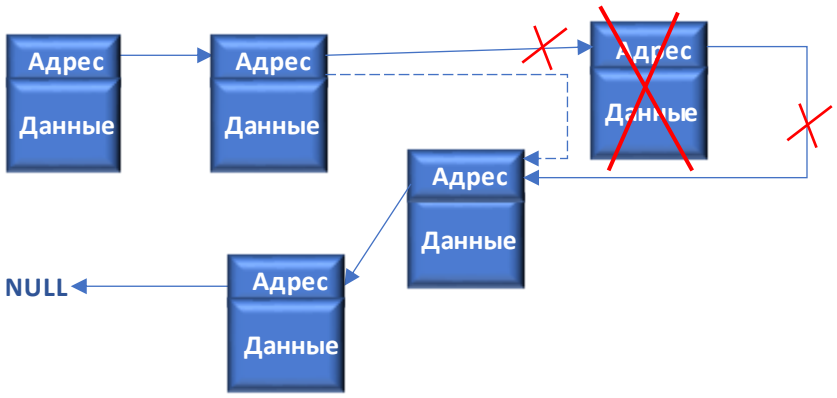


Рис.15. Удаление элемента из односвязного списка

Аналогичным образом осуществляется и вставка нового элемента в середину списка – изменяется адрес у элемента, предшествующего новому. При этом старая связь ликвидируется. У добавляемого элемента адрес будет указывать на следующий за ним элемент (рис.16).

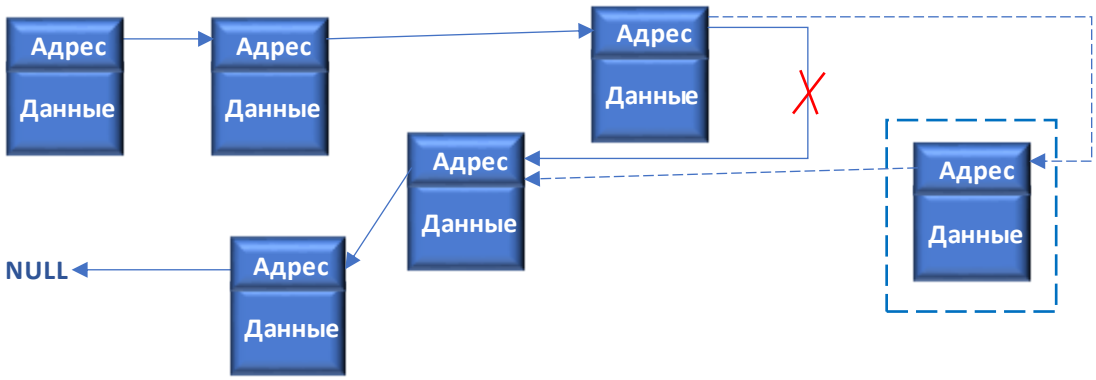


Рис.16. Добавление элемента в односвязный список

Что касается практического применения односвязных списков, то примером могут служить связанные списки состояний при реализации кнопок отмены обычного приложения, нап-

ример, текстового редактора Microsoft Word, графического редактора Paint и других. Картографические данные также могут быть представлены в виде связанных списков, где операциями добавления и удаления узлов осуществляется перенаправление при GPS-навигации.

Несмотря на то, что у списков есть явное преимущество в плане добавления и удаления его элементов, недостатком списков по сравнению с массивами является более сложная итерация при поиске определенного элемента, к тому же это потребует дополнительного ввода счетчика. При этом, чем больше количество элементов в списке, тем дольше времени будет требоваться на поиск нужного.

Частичным решением проблемы быстродействия является использование двусвязных списков (рис.17).

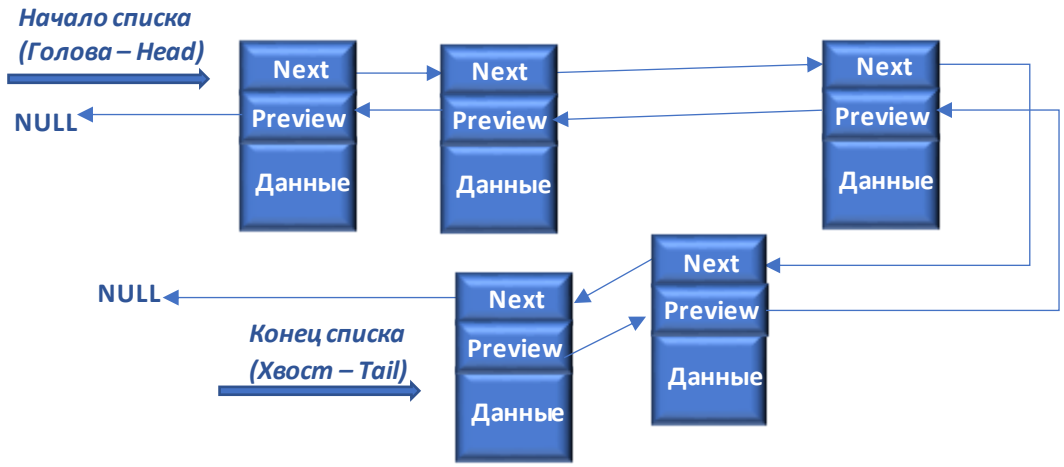


Рис.17. Графическое представление двусвязного списка

С помощью двунаправленной связи между элементами поиск нужного будет осуществляться более быстро оптимальным путем – либо с начала списка, либо с конца. Для этого придется завести поле, в котором будет храниться количество элементов списка.

Однако, если элемент будет находиться, скажем, в середине списка, то его поиск займет достаточно больше времени.

Добавление и удаление элементов списка осуществляется аналогично подобным операциям в односвязных списках с той лишь разницей, что для самого элемента, предшествую-

щего ему и последующего элемента придется изменять данные в обоих полях для каждого – для предыдущего и последующего элементов.

Двусвязные списки URL-адресов адресов используются, например, в реализации предыдущей и следующей кнопок браузера; реализация кнопки последовательного просмотра изображений (следующего и предыдущего) основана на использовании двусвязного списка изображений и т.д.

В последнее время в нашу жизнь прочно вошло такое понятие, как блокчейн. Эта инновационная технология проникает в различные сферы, переплетаясь с обыденностью и преображая мир вокруг. Его применение открывает новые перспективы, преображая каждую область. Так, например, он становится стражем безопасности и эффективности обмена медицинской информацией; в вопросах интеллектуальной собственности блокчейн стал неотъемлемым инструментом в отслеживании и подтверждении прав на интеллектуальные творения; благодаря блокчейн обеспечивается безупречная безопасность и прозрачность на каждом из этапов процесса электронного голосования; в образовании он становится инструментом подтверждения квалификаций и т.д., не говоря уж о том, что он является основой для получивших достаточно широкое распространение криптовалют.

Если рассматривать блокчейн как структуру данных, то его структура очень схожа со структурой связного списка. В блокчейне данные разбиты на блоки, выполняющие роль контейнеров и которые аналогичны узлам связного списка. В каждом из блоков имеется ссылка, которая является хэшем предыдущего блока. Такая ссылка служит связью с предыдущим блоком и помогает поддерживать правильный порядок блоков во всей цепочке.

Общим для связного списка и блокчейна также является то, что данные в них могут быть динамическими и изменяемыми. Таким образом, можно сказать, что блокчейн похож на связный список в том, как он организован и использует ссылки для поддержания порядка

данных. Но, конечно же, в блокчейне есть и уникальные особенности, которые делают его особенным и отличают его от традиционных структур данных.

Абсолютное же различие между технологией блокчейн и связным списком проявляется в степени безопасности. В блокчейне каждая связь оборачивается криптографическим барьером, обеспечивая высший уровень защиты. Внесение новой информации в блокчейн предполагает лишь пошаговое расширение цепи, начиная с её переднего края. Аккуратная проверка корректности защищенных соединений является постоянным этапом. Процедура подтверждения, которая представлена в блокчейне, позволяет уверенно обеспечить защиту данных. Каждый новый блок надстраивается над уже имеющимся, и такая конструкция гарантирует прозрачность изменений. Увеличение количества подтверждений для старых блоков усложняет возможность подделки данных. Каждое подтверждение означает необходимость воссоздания валидного блока с новой валидной ссылкой. В результате, с течением времени, уровень надежности блоков повышается и оставляет мало шансов на внесение каких-либо изменений в данной структуре.

Преимущество блокчейна заключается в том, что невозможно добавить и удалить данные из блока незаметно. Это свойство является фундаментальным для обеспечения надежности и безопасности данных. Любая попытка изменить информацию в блокчейне будет замечена, так как она нарушит целостность цепочки и нарушит ссылки на последующие блоки. Таким образом, благодаря структуре данных и механизму подтверждения, блокчейн является надежным инструментом для хранения и передачи информации. Вся подделка становится явной, а любое изменение данных ставит под угрозу ссылки на все последующие блоки.

Из всего вышесказанного можно сделать вывод, что блокчейн является революционной технологией, которая обеспечивает надежность и безопасность данных, сохраняя их целостность и неизменность. Это дает новые возможности для развития и инноваций во многих отраслях, что делает блокчейн незаменимым инструментом для будущего.

Графы

Граф является абстрактным представлением множества объектов и связей между ними. С помощью графов можно описать организацию различных транспортных систем, загруженность магистралей, организацию телекоммуникационных сетей, социальных сетей и т.п.

В дискретной математике раздел, посвященный изучению графов, называется теорией графов.

Графически граф (обозначается G) можно представить в виде некой совокупности точек, которые соединены между собой линиями. При этом точки, которые обязательно должны быть идентифицированы, называются узлами или вершинами V (*Vertex*), а линии – дугами (в случае ориентированного графа) или ребрами E (*Edge*).

Таким образом, зависимость между компонентами графа выражается формулами:

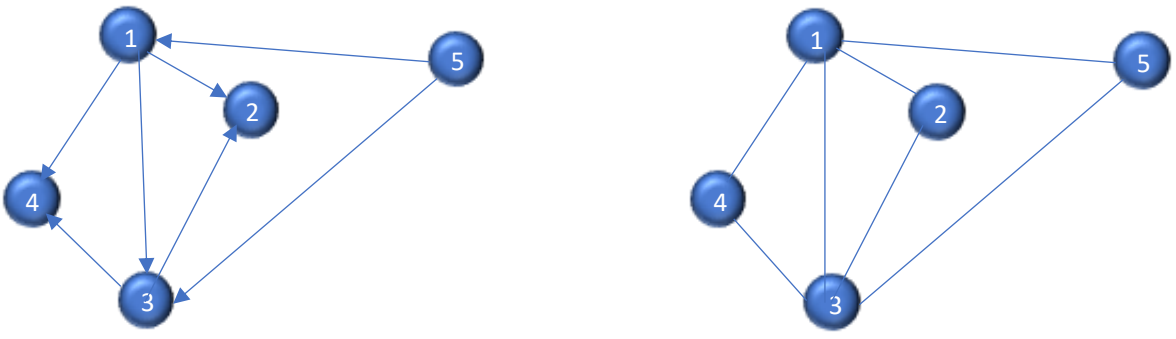
$$V=\{v1, v2, \ldots, vn\}, \text{где } n \text{ – число вершин графа;} \quad (2)$$

$$E=\{e1, e2, \ldots, em\}, \text{где } m \text{ – число ребер графа;} \quad (3)$$

и, соответственно,

$$G=(VE) \quad (4)$$

Среди многообразия графов в первую очередь остановимся на *неориентированных* и *ориентированных* графах. В том случае, если ребра графа представлены стрелками, указывающими направление движения, то граф называется ориентированным (рис.18, *а*); в неориентированном графе можно пройти путь от одной вершины до другой в обоих направлениях (рис.18, *б*).



а) *б*)

Рис.18. Графическое изображение графа: *а*) ориентированного; *б*) неориентированного

Соответственно, и путь графа, представляющий собой конечную последовательность вершин, в которой каждые две вершины идущие подряд соединены ребром, может быть ориентированным, либо неориентированным.

Также бывают графы со *смешанными ребрами*. В таких графах часть ребер не имеет направления, а часть – содержит указывающее направление стрелки (рис.19).

Кроме того, существуют графы, у которых ребра несколько раз соединяют одни и те же вершины, формируя кратные ребра. Подобные графы называют *мульти-графами* (рис.20).

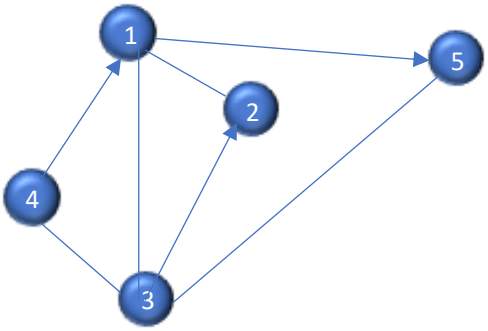


Рис.19. Граф со смешанными ребрами

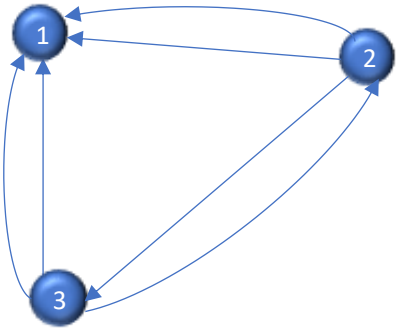


Рис.20. Граф с кратными ребрами

В случае, если у графа отсутствуют ребра (дуги), то он называется *нулевым*, а его вершины – *изолированными* (рис.21).

Если вершина (узел) графа соединена только лишь с одной соседний вершиной, то она будет называться *висячей*. Такой вершиной, например, является вершина 4 (рис.22), поскольку она соединена только лишь с одной вершиной 1.

Если у какой-либо вершины ребро выходит и затем входит в нее же, то такое ребро называют *петлей* (вершины 1 и 2 на рис.22).

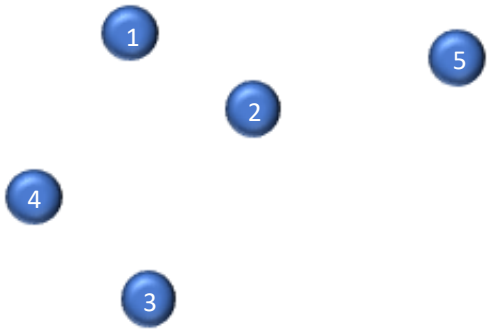


Рис.21. Нулевой граф

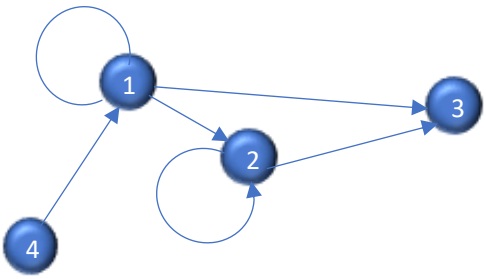


Рис.22. Граф с петлями

Такое понятие, как *степень вершины* (обозначается $deg(v)$), для ориентированных и неориентированных графов различно. В первом случае говорят о степени входа вершины, определяемое по количеству входящих в нее ребер и степени выхода, определяемого по количеству выходящих из нее ребер. В случае же неориентированного графа исходящая степень равна входящей. Так, например, на рис. 18,*а* степень входа вершины 3 равна двум и степень выхода – также двум. Для графа, изображенного на рис.18,*б* степень вершины 3 равна четырем.

Графы могут быть *полными*. Полным называется граф, в котором каждая вершина соединена с каждой другой вершиной графа, то есть каждая пара вершин соединена ребром. Для определения количества всех соединений p_n (ребер) в полном графе с n вершинами пользуются формулой

$$p_n = \frac{n(n-1)}{2}. \quad (5)$$

Еще одно ключевое понятие, связанное с графами – это его *связность*. Граф называется связным, если в нем есть путь от любой вершины до любой, то есть между любой парой вершин этого графа существует как минимум один путь.

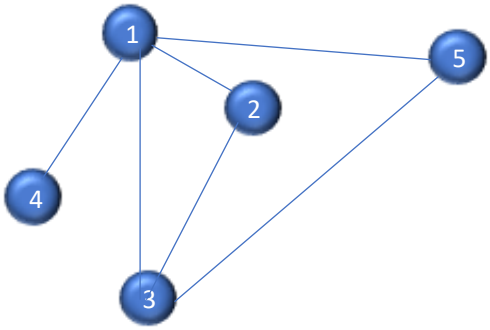
Среди способов представления графов выделяют *матрицы* и *списки*.

Одни из способов представления в виде матриц является *матрица смежности*, в которой заголовки строк и столбцов соответствуют номерам вершин графа, а сами элементы матрицы соответствуют числам 0 и 1, которые выставляются в зависимости от того, есть ли ребро, соединяющее эти вершины. Это правило справедливо для неориентированных графов. В случае же ориентированных графов элементы матрицы заполнятся единицами только лишь в случае наличия исходящей дуги. На рис.23 продемонстрированы матрицы смежности для неориентированного (*а*) и ориентированного (*б*) графов.

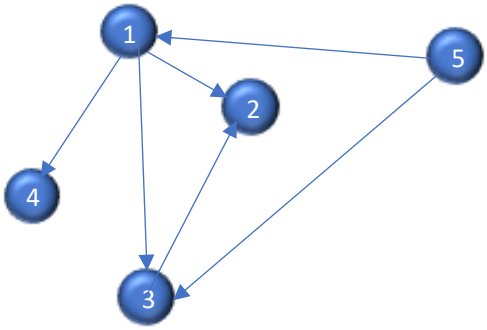
Как видно из рис.23,*а*, матрица смежности для неориентированного графа является симметричной относительно главной оси. Именно эта особенность гарантирует экономию

вдвое используемой памяти, поскольку элементы можно хранить только в верхней части матрицы над главной диагональю.

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



а)



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

б)

Рис.23. Граф и соответствующая матрица смежности для:

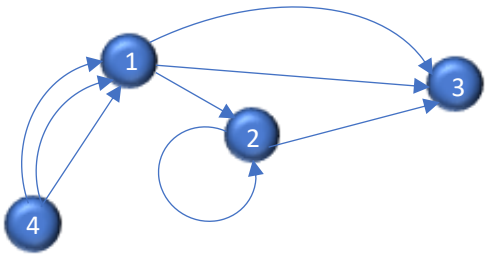
а) неориентированного графа; *б*) ориентированного графа

Наличие петель также отмечается единицей, а в случае кратных дуг вместо единицы в матрицу записывается число, соответствующее количеству дуг (ребер) (рис.24).

Матрицы смежности обычно применяют в случаях, когда число ребер примерно равно числу вершин, в так называемых *плотных* графах.

Вторым способом матричного представления графов является *матрица инцидентности*. Считается, что ребро инцидентно двум вершинам в том случае, если оно их соединяет. Перед заполнением такой матрицы ребра необходимо также пронумеровать (или отметить буквами). Заголовкам строк и столбцов соответствуют вершины и ребра соответственно.

Элементам матрицы соответствуют числа 0 и 1, которые выставляются в зависимости от того, есть ли ребро, соединяющее эти вершины. Это правило справедливо для неориентированных графов. В случае же ориентированных графов элементы матрицы записываются в соответствии с ориентацией ребра: исходящее ребро обозначается 1, а входящее – -1.



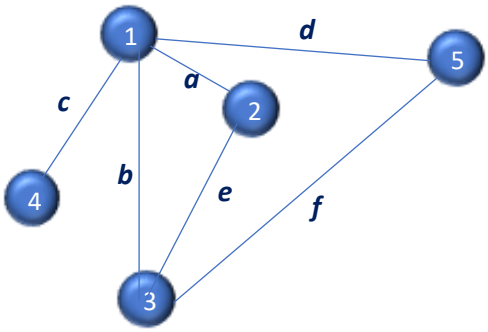
	1	2	3	4
1	0	1	2	0
2	0	1	1	0
3	0	0	0	0
4	3	0	0	0

Рис.24. Граф с наличием кратных дуг и петли и соответствующая матрица смежности

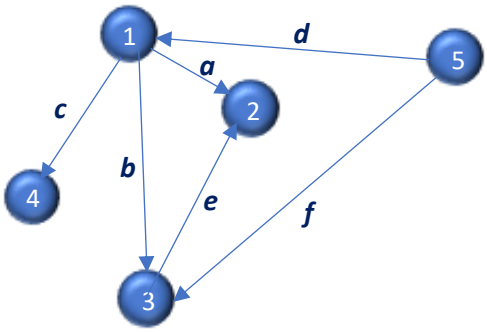
На рис.25 продемонстрированы матрицы инцидентности для неориентированных и ориентированных графов.

соединяющие вершины.

	a	*b*	*c*	*d*	*e*	*f*
1	1	1	1	1	0	0
2	1	0	0	0	1	0
3	0	1	0	0	1	1
4	1	0	0	0	0	0
5	0	0	0	1	0	1



а)



	a	*b*	*c*	*d*	*e*	*f*
1	1	1	1	-1	0	0
2	-1	0	0	0	-1	0
3	0	-1	0	0	1	-1
4	-1	0	0	0	0	0
5	0	0	0	1	0	1

б)

Рис.25. Граф и соответствующая матрица инцидентности для:

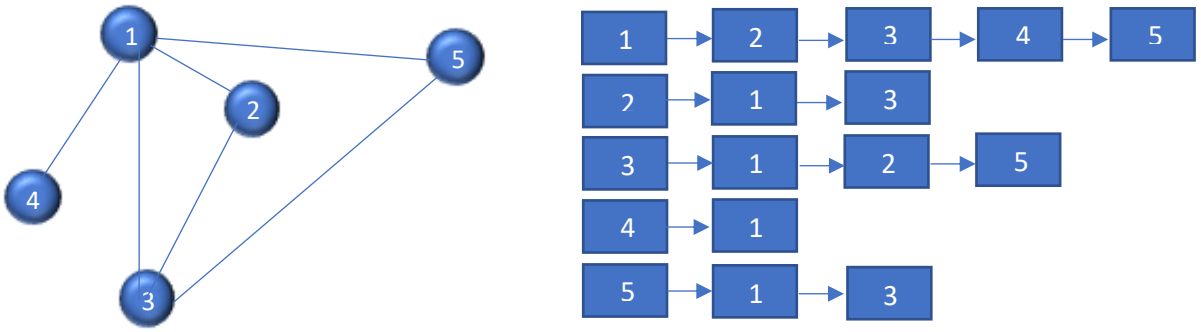
а) неориентированного графа; *б*) ориентированного графа

Нетрудно заметить, что количество единиц в матрицах инцидентности для неориентированных и ориентированных графов совпадает, более того, они находятся в матрицах на одних и тех же позициях. Единственное различие заключается в том, что в случае ориентированных графов учитывается и направление дуг, поэтому в соответствующей матрице вместо единиц в случае входящих дуг появляются единицы со знаком минус.

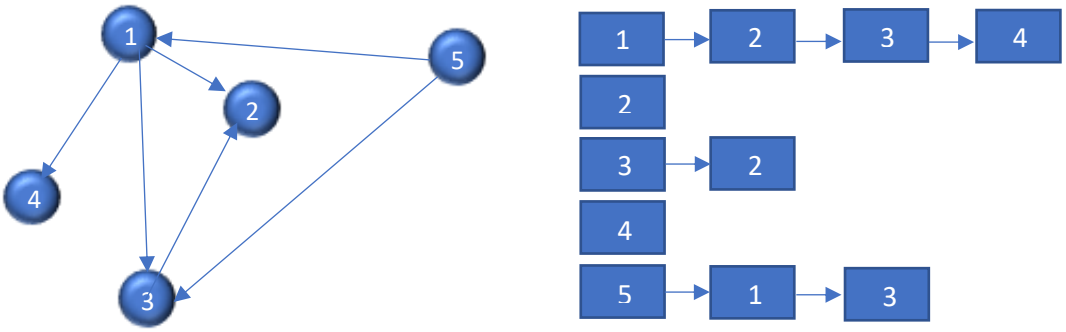
При программной реализации матрицы смежности и инцидентности представляются в виде обыкновенных двумерных массивов.

В том случае, если количество ребер в графе невелико по сравнению с вершинами (разреженные графы), использовать матрицы, в которых большинство элементов будет являться нулями, нецелесообразно. Для таких задач предусмотрено использование списков, которые представлены списками смежности и списками ребер.

Списки смежности для неориентированных и ориентированных графов представлены на рис.26.



а)



б)

Рис.26. Граф и соответствующий список смежности для:

а) неориентированного графа; *б*) ориентированного графа

Если в случае с неориентированным графом наблюдается некоторое дублирование ребер, то в случае ориентированного графа наблюдается значительная экономия памяти.

У списков есть и недостатки. В первую очередь, это касается работы с *насыщенными графами*, имеющими много ребер, а также со *взвешенными графами*[1], требующими хранения двух значений для каждого поля, что значительно усложняет код.

Аналогичным способом составляются и списки ребер. В нем указываются все ребра, соединяющие вершины.

Префиксные деревья

Деревья представляют собой иерархическую структуру данных. Такие структуры используются в основном при работе с данными, имеющую иерархическую структуру, например, генеалогическое древо, служебных должности, биологические виды, географические объекты.

По своему виду они схожи с ациклическими графами. Деревья имеют один узел – вершину, не имеющий предков и называемый *корнем*. Остальные элементы именуют *вершинами* (*узлами*), которые соединены между собой *дугами* (*ветвями*). Вершины, которые ветвями соединены с вышерасположенной вершиной, называют *потомками*, а соответственно, вершину, из которой ветви исходят – *предком*. Потомки со своим предком могут быть представлены как поддерево. Узлы, не имеющие потомков, называют *листьями дерева*.

Глубина дерева определяется длиной пути к корню, а высота – длиной самого дальнего пути к листу.

На рис.27 схематично представлен пример дерева с основной терминологией.

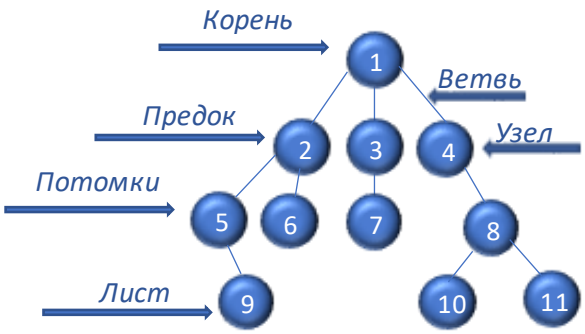


Рис.27. Дерево с основной терминологией

[1] Взвешенный граф – это граф, некоторым элементам которого (вершинам, ребрам или дугам) сопоставлены числа.

Среди многообразия деревьев следует особо выделить *префиксное дерево* (бор, луч, нагруженное дерево), получившее свое название потому, что поиск осуществляется в нем по префиксам. Используются такие структуры при хранении ассоциативных массивов и динамических множеств, где ключом выступают строки.

Хранение данных в виде префиксного дерева значительным образом высвобождает память. К этому заключению можно прийти при анализе конкретного примера, продемонстрированного на рис. 28. Допустим, в нашей базе данных необходимо сохранить данные, представленные в этом примере пятью наборами чисел. Сохранение их в том виде, в котором они записаны, приведет к некоторому дублированию, поскольку даже на первый взгляд видно, что у четырех из пяти наборов числа начинаются с 0, и лишь у последнего – с 1. Следовательно, мы имеем возможность сохранить 0 лишь один раз и перейти к анализу вторых цифр в наборе (на рисунке первое ответвление 0 и 1).

Как показывает анализ, вторая цифра в первых четырех наборах также совпадает (1), а у пятого набора отличается (0). Поэтому в верхней ветке мы указываем 1, а в нижней – 0.

Далее смотрим на третьи цифры в наборах. И снова отмечаем совпадение в первых четырех наборах – в каждом из них третья цифра – это 0. Отмечаем ее на верхней ветке. В нижнюю добавляем отличающееся значение 3.

Переходим к анализу четвертой цифры. И вот здесь отличие наблюдается уже в первых четырех наборах – у первых трех наборов 2, а у четвертого – 1. Поэтому фиксируем ветвление в верхней ветке указанием этих значений. Количество ветвей увеличивается. В самой нижней ветви записываем 0.

Аналогичным образом продолжаем последовательный анализ всех элементов набора, производя ветвления в случае наличия отличающихся цифр.

Подобный подход позволяет избежать избыточности данных, а получить ключ в префиксных деревьях можно выписыванием подряд всех символов, которые отмечены на ребрах по пути от корня до узла. Так, если в нашем примере выписать последовательно все числа,

расположенные на самой верхней ветви (0→1→0→2→4→9→8), то получим первое число набора. Если, например, выписать последовательно все числа, расположенные на самом нижнем разветвлении верхней ветви (0→1→0→1→5→4→7), то получим четвертое число набора и т.д.

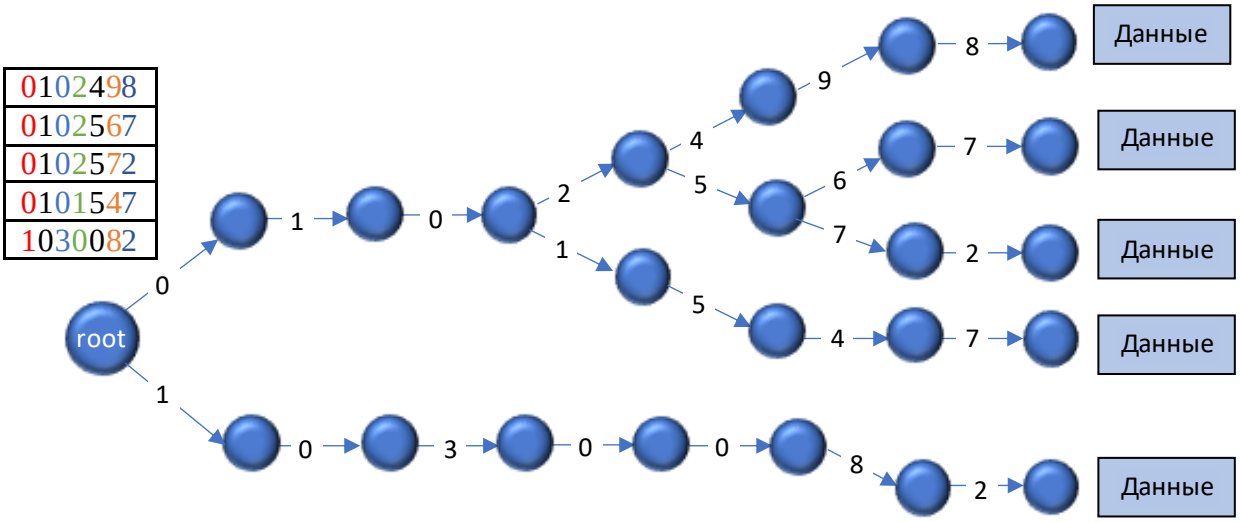


Рис.28. Пример префиксного дерева

Хеш-таблицы

Хеш-таблицы (*Hash-Table, HashMap*) настолько популярны, что используются практически во всех языках программирования. Они применяются в базах данных при построении индексов, при построении кэша, в языковых процессорах, в словарях и т.д. При помощи хеш-таблиц очень легко и быстро осуществляется поиск данных, значительно быстрее, чем при использовании других походов, поскольку не приходится просматривать абсолютно все данные, содержащиеся в таблице. Достаточно быстро происходит добавление и удаление записей.

Хеш-таблица представляет собой массив, который работает в сочетании с *хеш-функцией*. Хеш-функция получает на вход данные, называемые ключом, а на выходе она выдает целое число – хеш-значение. Хеш-значение привязывает ключ к определенному индексу в таблице. В качестве ключа могут использоваться любые типы данных – числа, а также символы.

В качестве хеш-функции применяют различные математические формулы. Часто пользуются для получения хеш-значений остатком от деления ключа на количество строк в хеш-таблице, например, как это продемонстрировано на рис.29. Ключи делятся на число строк табли-

цы с остатком по модулю. В правой части рисунка изображена хеш-таблица, сформированная при помощи полученных хеш-значений, используемых в виде индекса.

Так, например, значение первого ключа в нашем примере 35, делится с остатком по модулю на число строк (в данном конкретном случае строк 8): 35%8=3[1]. Следующие хеш-значения: 4%8=4; 8%8=0; 74%8=2; 15%8=7 и т.д.

Затем составляется хеш-таблица (см. рис.29). В первом столбце этой таблицы располагаются индексы в порядке возрастания, некоторым из которых соответствуют полученные нами по формуле хеш-значения, а во втором столбце – соответствующие хеш-значениям значения ключа. В нашем случае хеш-значению 0 соответствует ключ со значением 8; хеш-значению 1 – ключ 9 и т.д.

Как видно, из хеш-таблицы (см. рис.29), хеш-значению 2 соответствуют два значения ключа – 74 и 50 (на рисунке эта ячейка отмечена). Такая ситуация называется *коллизией*.

Существует два метода разрешения коллизий. Первый – это *метод открытой адресации*, когда значение помещается в другую ячейку, например, в первую свободную. Так, значение 50 на рис.29 будет помещено в свободную ячейку с индексом 5. Этот процесс называется *пробированием*. Пробирование бывает линейным, квадратичным. Иногда используют такой прием, как двойное хеширование и другие способы.

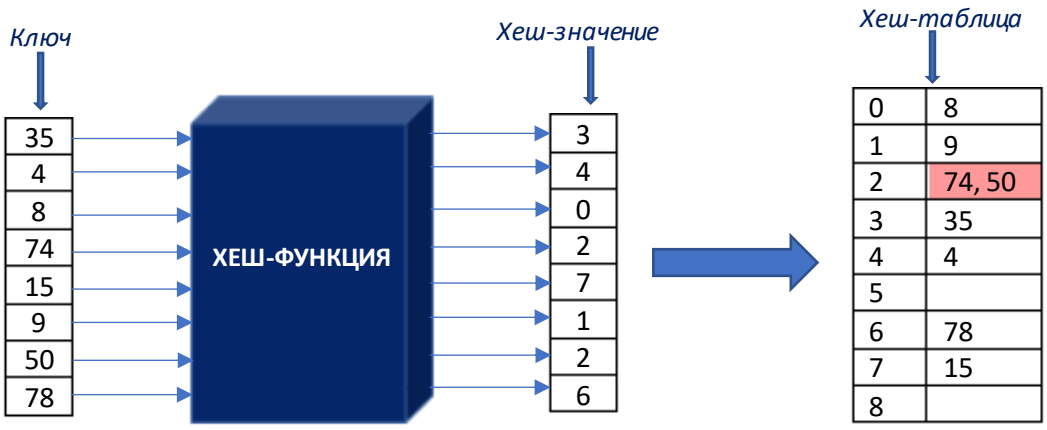


Рис.29. Пример процесса создания хеш-таблицы

[1] Символом % в программировании принято обозначать остаток от деления по модулю.

У метода открытой адресации есть свои недостатки. Во-первых, это зависимость от способа обхода. К примеру, таблица может заполнится не полностью и пути обхода могут стать длинными, затрудняя поиск и добавление информации. Во-вторых, надо внимательно выбирать размер внутреннего массива. Если он мал, то информация может просто не уместиться, а если велик – то память будет расходоваться неэффективно.

Второй метод – *метод цепочек*, который добавляет в конфликтную ячейку ссылку на следующее значение, то есть мы сохраняем это значение не в нашем массиве, а в отдельной области памяти. В этом случае не приходится искать новую ячейку для размещения второго ключа. Первый ключ будет хранить ссылку (адрес) на следующее значение ключа с совпадающим хеш-значением по такому же принципу, как это осуществлялось в связанных списках.

К недостаткам можно отнести расход дополнительной памяти на хранение ссылок, а сами ссылки ведут на разрозненные участки памяти, что затрудняет переход по ним.

Что касается самой хеш-функции, то она должна обладать некоторыми свойствами:

- *детерминизм* – для одного и того же значения ключа функция должна выдавать в разные промежутки времени одно и то же значение;
- *равномерность* – у разных ключей не должно быть одинакового индекса;
- *эффективность* – функция должна вычисляться быстро;
- о*граниченность* – индексы, выдаваемые функцией, должны быть в пределах таблицы.

Описание и оценка компьютерных алгоритмов, описание времени работы алгоритма

При создании и, в особенности, реализации алгоритмов очень важным является оценить их сложность. Например, существует такое понятие, как *описательная сложность алгоритма*, которая определяется непосредственно длиной алгоритма, реализованного на каком-либо из языков программирования. В силу ряда особенностей каждого из них, алгоритм может быть записан разным количеством строк, что и будет определять его описательную сложность.

Однако, чаще всего для оценки сложности алгоритмов используют две другие наиболее важные характеристики. Первая характеристика представляет собой *оценку по времени*, а вторая характеристика – *оценку по памяти.*

Первая характеристика связана с вычислительной сложностью, то есть фактически необходимо оценить, какое количество операций необходимо выполнить алгоритму, чтобы получить желаемый результат.

Оценка же по памяти подразумевает, что необходимо оценить, какое количество дополнительной памяти требуется нашему алгоритму для того, чтобы получить искомый результат.

Рассмотрим сначала оценку по времени на простом примере.

Пусть необходимо найти позицию некоторого числа в массиве случайных целых чисел. Рассмотрим массив из 10 произвольных чисел и определим, сколько необходимо операций, чтобы в данном массиве найти, например, число 55. Скорее всего, придется пройти по каждому элементу, начиная с первого элемента, просмотреть его и сравнить с искомым числом. Если оно найдено, то можем вывести позицию, на котором оно находится. В нашем конкретном случае, искомое число 55 оказалось на 5 позиции (рис.30). Для его поиска нам пришлось 5 операций.

В том случае, если попал наихудший вариант и число 55 оказалось на последнем месте, то для его поиска придется пройтись по всем абсолютно элементам массива от начала до конца, просматривая каждый элемент. А также, если искомого числа нет вообще в этом массиве, опять же необходимо пройтись по каждому элементу массива.

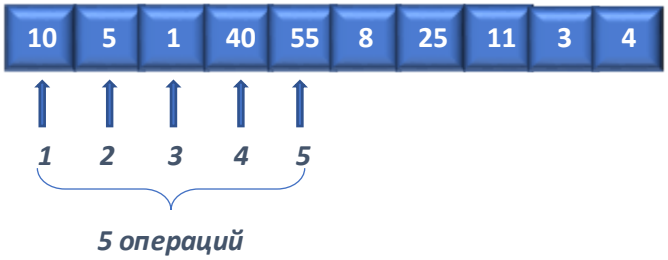


Рис.30. Поиск элемента 55 в массиве данных

То есть, если мы будем рассматривать общий абстрактный случай поиска произвольного целого числа в массиве произвольных целых чисел, то в самом худшем случае для массива длиной n понадобится n операций. Это число n и является характеристикой, описывающей насколько сложным по времени является рассматриваемый алгоритм. Для обозначения данной оценки сложности алгоритма используются специальные обозначения, такое как $O(n)$, описывающее зависимость количества операций от количества исходных данных. В данном примере $O(n)$ показывает линейную зависимость, при которой количество операций будет увеличиваться пропорционально количеству исходных данных (линия 1 на рис.31).

Допустим, необходимо найти наибольшее среднее арифметическое число для двух массивов произвольных целых чисел. Здесь придется сделать цикл для прохода по каждому элементу одного массива с количеством n операций, а для второго – с количеством k операций. Затем необходимо произвести сравнение средних арифметических массивов. Если оценим общую сложность данного алгоритма, то получим $O(n)+O(k)+const$ ($const$ – это время для сравнения средних арифметических). Общая сложность будет выглядеть, как $O(n+k+const)$.

Когда мы рассматриваем общую сложность алгоритма по времени, то самым главным является видение тенденции, роста сложности алгоритма в зависимости от роста количества входных данных. В общем случае, если количество исходных данных устремить к беско неч-

ности, то оценки сложности константы и различных целых коэффициентов в принципе не играют значительной роли. Самую главную роль играет только максимальная степень для переменной, обозначающей исходные данные – максимальная степень для n. В этом случае при игнорировании констант для линейной зависимости придем к такой оценке, как $O(n)$.

Если возьмем квадратичную зависимость, то в ней главную роль играет n^2, так как это именно то слагаемое, которое дает наибольший прирост сложности в алгоритме.

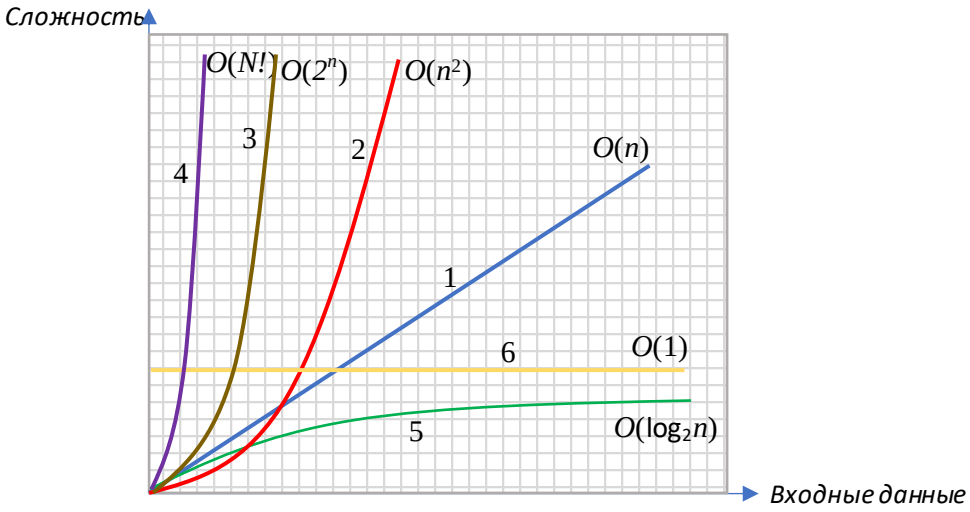


Рис.31. Зависимости, описывающие сложность алгоритма по времени

Итак, $O(n)$ подразумевает линейную зависимость. Если мы возьмем две точки на оси входных данных и посмотрим, насколько между двумя точками изменилось n и насколько изменилась сложность, то увидим, что они растут пропорционально. То есть, если количество элементов увеличилось на 1000, то и количество операций увеличится на 1000.

В случае квадратичной зависимости имеем кривую в виде ветки параболы (линия 2 на рис.31). Можем видеть, что прирост сложности при росте количества элементов очень быстрый. Это значит, что даже при небольшом количестве новых элементов в исходных данных, будет наблюдаться значительный рост сложности алгоритмов. Если например, для 1000 исходных данных мы получаем 1 000 000 операций, то для 10000 исходных данных получаем уже 100 000 000 операций.

Примером может служить двумерный массив, в котором необходимо найти максимальный элемент. Для этого необходимо пройтись по каждой строке и в каждой строке пройтись по каждому столбцу. Таким образом, мы получаем двойной цикл. Внешний цикл будет по строкам, а внутренний – по столбцам. Соответственно, получаем $n*n$ количество операций. Таким образом, получим n^2. В общем случае можем сказать, если видим двойной цикл (вложенный цикл) то зачастую это сложность n^2. Если тройной вложенный цикл, то сложность составит n^3 и т.д.

Экспоненциальная сложность $O(2^n)$ (рис.31, линия 3) встречается в рекурсивных алгоритмах и задачах с возвратом, как, например, поиск элементов последовательности Фибоначчи, где для каждого элемента нужно находить сумму двух предыдущих до тех пор, пока мы не дойдем до первого и до нулевого элементов.

Факториальную сложность (рис.31, линия 4) можно видеть на примере задач с перестановками.

Следующая зависимость – логарифмическая, является самой хорошей из вышерассмотренных. В программировании логарифм чаще всего берется по основанию 2. Как видно из рис.31 (линия 5), логарифмическая кривая растет очень медленно. То есть, при даже большом росте входных данных, получаем очень небольшой рост в сложности алгоритмов. Поэтому логарифмическая зависимость (логарифмическая сложность алгоритма) является одной из самых лучших зависимостей, к которой надо стремиться по возможности при создании алгоритмов.

Примером логарифмической зависимости может служить поиск элементов в массиве, но при этом массив будет не произвольных чисел, а строго упорядоченных чисел. Благодаря этому мы можем применить алгоритм деление пополам – бинарный поиск. Это значит, что каждый раз будем уменьшать просматриваемый массив в два раза. В этом случае понадобится намного меньше операций. Если функция будет вызывать сама себя и благодаря тому, что

количество просматриваемых элементов каждый раз уменьшается в 2 раза, мы получаем сложность $\log_2 n$.

Однако, самой лучшей оценкой сложности алгоритма является O(1). Так обозначают ту сложность, которая вообще не зависит от количества исходных данных. Если алгоритм и на два числа и на 100000 чисел, решает задачу за одинаковое количество операций, то это сложность O(1). На графике видно (рис.31, линия 6), что в независимости от того, какое количество n у нас есть, сложность алгоритма находится на одном уровне. В принципе любая арифметическая операция, которая не зависит от количества элементов, то есть когда нет необходимости просмотра каждого элемента и осуществления итерации для каждого элемента, будет вычисляться за константное время. Таким образом получим сложность алгоритма O(1).

В принципе это основные, наиболее часто встречающие зависимости в программировании, объединяемые нотацией O (омикрон, O-большое, Big O).

Однако, существуют и другие оценки сложности алгоритмов. Если быть точными, то, включая выше рассмотренную нотацию, их пять. Итак:

- Big O – верхняя граница сложности алгоритма. Представляет собой наилучший инструмент для поиска возможного наихудшего случая, то есть $F \in O(g): F \leq g$, где F – реальная функция, а g – асимтотическая[1]. Например, нам необходимо произвести сортировку пузырьком (различные сортировки будут рассмотрены в дальнейших разделах) массива по возрастанию, который уже отсортирован по убыванию;

- Big Ω (большая омега) – нижняя граница сложности, которую зачастую используют при оценивании сложности алгоритма в лучшем случае, то есть $F \in \Omega(g): F \geq g$. Скажем, если потребовалось произвести сортировку в массиве, который уже отсортирован;

[1] Термин *«асимптота»* происходит от древнегреческого *ἀσύμπτωτος* и обозначает в упрощенном виде прямую, которая неограниченно приближается к бесконечной кривой, но не пересекает ее. Таким образом, асимптотический анализ – это метод описания предельного поведения чего-либо, в данном случае алгоритмов.

- Big Θ (большая тета) – располагается между нижней и верхней границами и представляет собой точную сложность, наилучшим образом подходящую для поиска среднего случая, то есть $F \in \Theta(g)$: $F=g$. Например, сортировка в массиве, который не отсортирован в нужной последовательности, но и не отсортирован и в обратном порядке;

- o и ω (маленькая o и маленькая омега) – используются для оценки сложности при свободной (не точной) верхней и нижней границах, соответственно, то есть $F \in o(g)$: $F<g$ и $F \in \omega(g)$: $F>g$, соответственно.

Однако наибольший интерес вызывает все же нотация O, поскольку она показывает как будет расти время в самом наихудшем из вариантов и, тем самым, покрывает все другие ситуации.

Если говорить про *сложность по памяти*, то принцип один и тот же. Единственная разница – оцениваем сколько дополнительной памяти нам надо, а не какое количество операций мы используем для достижения поставленной цели. Например, если возьмем такие алгоритмы как поиск максимума, минимума, суммирование всех элементов массива и т.п., то здесь мы при решении задачи никакой дополнительной памяти не выделяем, то есть сложность O(1). Если же задача состоит в добавлении нового элемента в массив, то для того, чтобы в обыкновенный массив добавить новый хотя бы один элемент, понадобится, как мы уже и говорили ранее, создать новый массив с количеством элементов больше предыдущего на 1, полностью переписать в него все элементы и в дополнительную ячейку внести значение нового элемента. Таким образом, для решение подобной задачи, необходимо выделить память, равную памяти исходного массива, т.е. получаем что сложность по памяти является $O(n)$.

Инварианты циклов

Одной из важнейших задач для программиста является написание корректного кода, который будет работать при любых возможных входных значениях. Тестовые программы, которые, как правило, применяются для оценки правильности кода, в большинстве случаев

могут служить доказательством лишь того, что ошибок нет, что вовсе не доказывает их отсутствие при определенных значениях. Все потому, что тестовые проверки содержат готовые наборы входных данных, для которых известен результат. Однако, вполне возможно, что при других наборах результат не будет корректным.

Гарантией того, что код, а до его написания алгоритм, корректен, может служить использование разнообразных методов доказательного программирования.

Как правило, обычно нет необходимости в проверке всего алгоритма или кода целиком – достаточно ограничиться доказательством корректности работы ключевого блока, например, цикла, или же функции и т.д.

Чаще же всего для проверки корректности работы алгоритмов используют инварианты циклов.

Фактически инвариант цикла представляет собой условие (предикат) среди переменных кода, которое обязательно является верным как непосредственно перед началом цикла, так и сразу после каждого шага (итерации) цикла. Иными словами, инвариант цикла должен быть истинным (хотя иногда он может временно быть ложным во время тела цикла):

1. перед началом цикла;
2. перед каждой итерацией цикла;
3. после завершения цикла.

В качестве примера можно рассмотреть следующую игру, в которой два участника. Перед ними разложены N фишек синего цвета и последняя (под номером N+1) – красного. У игроков есть возможность по очереди забирать из ряда от 1 до 3 фишек и проигравшим окажется тот, кто вынужден будет взять последнюю фишку.

Для своего выигрыша второй игрок должен восстанавливать после каждого хода своего соперника некий инвариант – число остающихся после его хода количество синих фишек должно быть кратным числу 4, т.е. синих фишек должно оставаться 8, 12, 16, 20, … N. Это гарантировано обеспечит второму игроку победу.

Инвариант цикла наглядно можно продемонстрировать и на примере суммирования ряда некоторых элементов, показанного в нижеприведенном фрагменте кода, реализованного на C++.

```
int main()
{
	int i, n, sum;
	cout << "Введите n";
	cin >> n;
	sum = 0;
	for (i=1; i<=n; i++)
	{
		sum = sum + i;
	}
	cout<<sum;
}
```

При каждом проходе цикла в переменную sum последовательно будет добавляться значение переменной i; на первом проходе 1, на втором – 2, на третьем – 3 и т.д. до n. Это и будет инвариантом цикла. На основании сформулированного инварианта можно сразу же сделать верный логический вывод, что в переменной sum после окончания работы цикла окажется сумма всех элементов от 1 до числа, указанного пользователем. Таким образом будет доказана корректная работа кода.

Еще один пример – нахождение максимального элемента в массиве из 5 чисел, указанных в нижеприведенном коде.

```
int main()
{
	int arr[5]{ 5, 8, 6, 4, 3 };
	int max; // переменная, содержащая максимум
	int i; // дополнительная переменная
	// установить максимум как 1-й элемент массива
	max = arr[0];
	for (i = 1; i < 4; i++)
		if (max < arr[i])
		{
			max = arr[i]; // запомнить максимум
		}
	cout << max;
}
```

Как видно, сначала условно за максимум принимается первый элемент массива с индексом 0. Затем в цикле происходит сравнение этого элемента со следующим. И, если следующий элемент окажется больше первого, то максимумом становится именно он. На каждом про-

ходе цикла в переменную max будет записываться максимальное значение из первых i элементов. Это утверждение и будет инвариантом цикла, что будет гарантией того, что при достижении перемененной i последнего значения, в переменную max будет записан максимальный элемент рассматриваемого массива.

Рекурсия

Понятие рекурсивных алгоритмов уже было рассмотрено вкратце в разделе 1. Напомним, что рекурсивный алгоритм – это алгоритм, который на каком-либо шаге прямо или косвенно обращается сам к себе.

В рекурсивном определении должно присутствовать ограничение – граничное условие, при выходе на которое дальнейшая инициация рекурсивных обращений прекращается.

Рассмотрим принцип рекурсии на конкретном примере – вычислении чисел Фибоначчи. Как известно, числа Фибоначчи представляют собой элементы числовой последовательности, в которой первые два числа равны 0 и 1 соответственно, а каждое последующее число равно сумме двух предыдущих (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…). Это определение можно записать в следующем виде:

$$F_n = \begin{cases} 0, & n = 0, \\ 1, & n = 1, \\ F_{n-1} + F_{n-2,} & n > 1. \end{cases} \quad (6)$$

С использованием псевдокода фрагмент вычисления чисел Фибоначчи можно записать следующим образом:

```
Вычисление функции Fibonacci(n);
если n<2
        то возвращаем число n;
в противном случае
        возвращаем Fibonacci(n-1)+Fibonacci(n-2)
```

Рассмотрим, как будет работать рекурсивная функция для вычисления числа Фибоначчи для, например, 5 элемента последовательности. Нумерация последовательности начинается с 0,

поэтому в данном конкретном случае значение 5 элемента совпадает и будет также равно 5, а для 6, например, элемента будет равно 8, для 9 – 13 и т.д.

Обращение к рекурсивной функции продемонстрировано на рис.32.

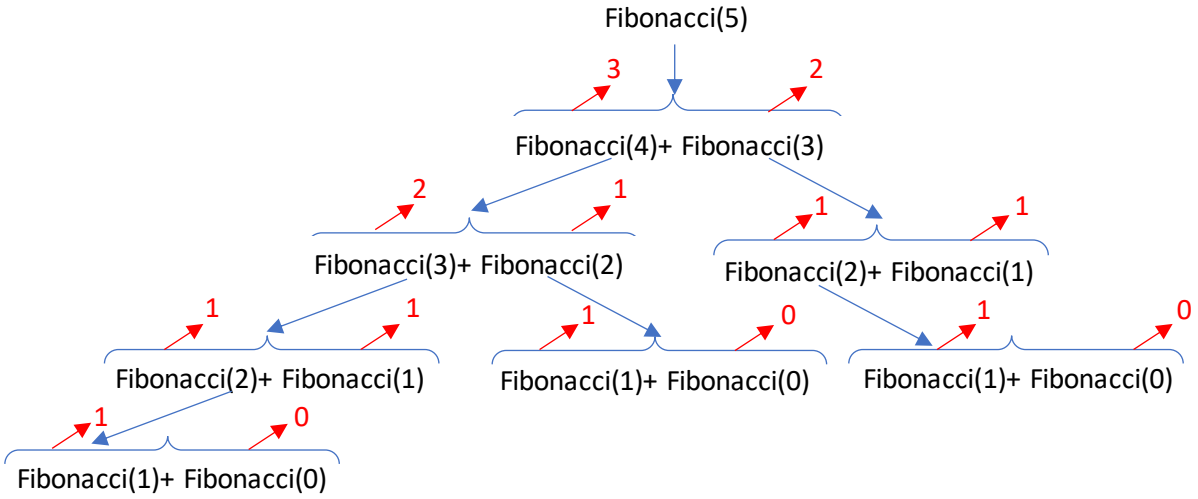


Рис.32. Рекурсивное обращение к функции и нахождение значения элемента последовательности

Как видно из рис.32, последовательное обращение к функции осуществлялось до тех пор, пока мы не дошли до базовых случае, то есть до значений 0 и 1 для *n*.

Красные стрелки показывают направление для расчета конкретного значения пятого элемента последовательности, отслеживая суммы с базовых значений.

Как видно, рекурсивный алгоритмы обладают недостатками. Для одного и того же значения функция вычисляется несколько раз, каждый раз запоминая вычисленное значение. Поэтому даже для такого небольшого значения, как, например, 30, мы можем столкнуться с проблемами крайне нерационального расхода памяти.

Подобного рода задачи лучше решать с использованием других методов, например, используя динамическое программирование.

Алгоритмы сортировки и поиска. *Бинарный поиск. Сортировка выбором. Сортировка вставкой. Сортировка слиянием. Быстрая сортировка. Сортировка пузырьком. Нижняя граница сортировки подсчетом и ее преодоление*

5

Бинарный поиск

Задачи поиска нужного элемента в массиве, нахождение максимального или минимального элементов в массиве, либо, например, первого элемента можно решить линейным поиском, элементарным перебором всех значений с использованием простого цикла. Сложность подобных алгоритмов оценивается как O(n). Однако, есть способы уменьшить сложность таких алгоритмов, сводом ее к нотации $O(\log_2 n)$. Одним из способов является использование бинарного поиска.

Пусть, например, в массиве, состоящем из 10 произвольных чисел следует найти число 89 среди чисел 5, 20, 23, 25, 55, 60, 65, 77, 89, 90. С использованием метода простого перебора придется дойти практически до конца массива, поскольку искомое число расположено на предпоследней позиции. Однако, можно поступить следующим образом. Разбить массив пополам и выяснить, в какой из половин находится искомое число (в нашем случае в правой части). Нужную половину разбить еще раз пополам, перейти в ту половину, в которой находится искомое число и снова ее разбить пополам. Процесс следует продолжать до тех пор, пока искомое число не будет найдено (рис. 33).

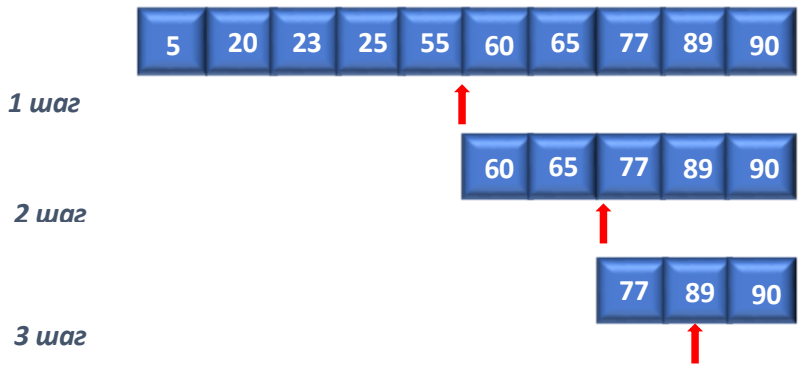


Рис.33. Схема бинарного поиска элемента 89 в массиве данных

За счет того, что на каждом шаге мы уменьшали вдвое исследуемый массив данных, сложность алгоритма сводится к логарифмической.

Разумеется, для корректной работы подобного алгоритма необходимым условием является наличие отсортированного списка. Мы можем брать неотсортированный список и производить его сортировку, либо сортировать список по мере добавления элементов.

Один из способов сохранения отсортированного списка с возможностью постоянного добавления и удаления элементов состоит в использовании особой структуры данных – *бинарного дерева*, обладающем рядом особенностей:

- левое поддерево содержит элементы, меньшие или равные вершине;
- правое поддерево может иметь элементы, большие или равные вершине;
- правые и левые поддеревья всех вершин также являются бинарными деревьями поиска.

Структура бинарного дерева представлена на рис.34. Главным условием является, что каждое поддерево может иметь не более двух потомков.

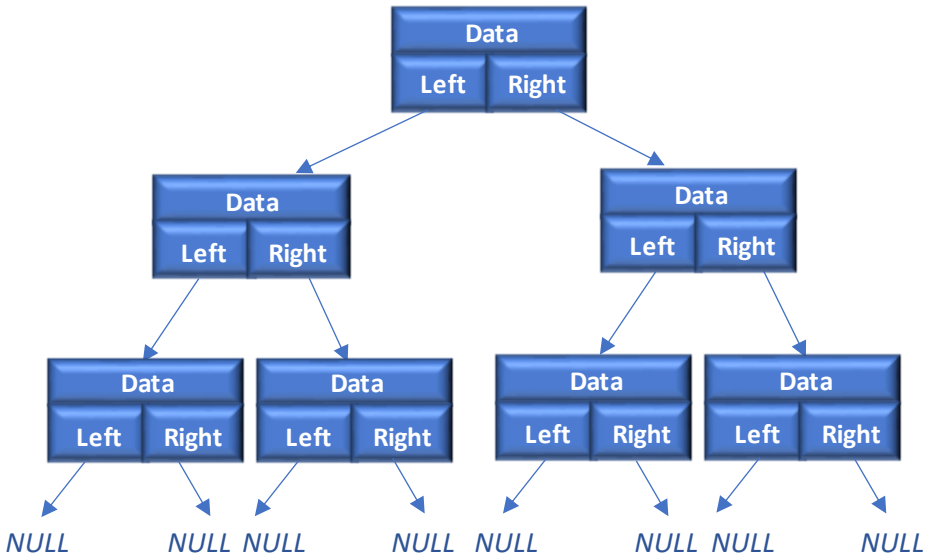


Рис.34. Схема расположения элементов в бинарном дереве

На рис. 35 представлена реализация бинарного дерева для неотсортированного массива [55 65 25 23 24 5 60 77 90 89]. Корнем дерева является число 55. От него отходят две ветви –

слева располагается число 25 (25<55), а справа – число 65 (65>55). После этого числа 25 и 65 сами становятся вершинами поддеревьев. Числа, меньшие 25 будут располагаться левее, а большие – правее (в данном примере слева добавлено число 23, а правая ветвь отсутствует). Затем число 23 становится вершиной поддерева с двумя ветвями (слева 5, а справа – 24). Аналогичным образом структурирована и правая ветвь основного бинарного дерева. В этом случае при поиске, например, числа 89 будет ясно, в каком из направлений следует двигаться. Кроме того, максимально быстро можно определить минимальный и максимальный элементы массива.

В бинарное дерево легко можно добавить новый элемент, а также удалить какой-либо из элементов. Так, на рис. 36 показано добавление элемента 27. Поскольку 27 меньше 55, перемещаемся в левую сторону, к поддереву с вершиной 25. Поскольку правая ветвь у него отсутствует, то мы добавляем элемент 27 справа от его вершины.

Удаление элементов не представляет сложности, если у поддерева нет ветвей, либо имеется одна ветвь. В этом случае последующий элемент просто перемещается на место удаляемого (рис.37, а).

При наличии ветвей процесс несколько усложняется. Например, если необходимо удалить саму вершину (элемент 55), то следует заменить его следующим по величине элементом – в нашем случае 60, который и станет новым корнем бинарного дерева (рис. 37, б).

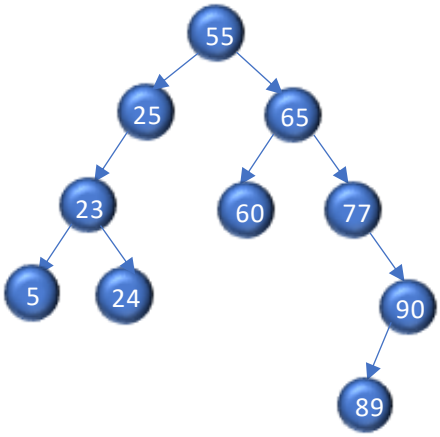


Рис.35. Бинарное дерево

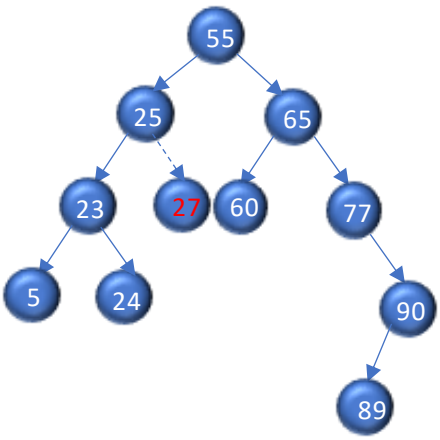
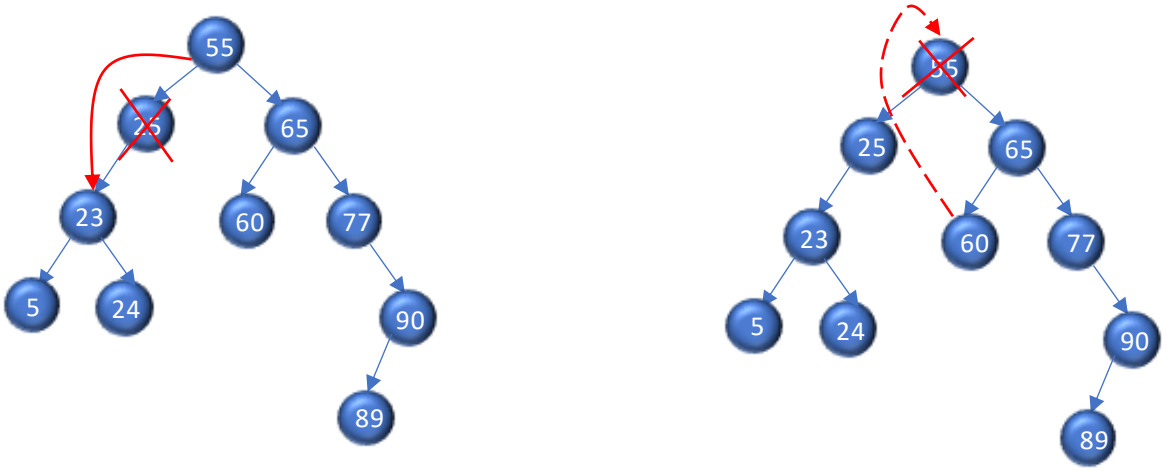


Рис.36. Добавление элемента в бинарное дерево



а) *б*)

Рис.37. Удаление элемента в бинарном дереве: *а*) с одной ветвью; *б*) с двумя ветвями

Сортировка выбором

Сортировка выбором получила свое название благодаря тому, что на каждой из итераций из всей последовательности неотсортированных данных выбирается элемент с наименьшим значением и помещается в самое начало неотсортированной последовательности. В этом случае предполагается сортировка по возрастанию. При этом на каждой итерации увеличивается отсортированная последовательность на один элемент и, соответственно, на один элемент уменьшается неотсортированная последовательность данных. На следующем шаге выбираем минимальный элемент неотсортированной части и помещаем его в начало. Эти действия выполняются до полной сортировки массива.

Аналогично можно производить сортировку и по убыванию. В этом случае следует выбирать максимальный элемент и помещать его на первое место в последовательности, а на его место перенести тот элемент, который до этого находился на первом месте.

На рис. 38 продемонстрирован метод сортировки выбором для последовательности расположенных в произвольном порядке произвольных семи чисел.

На первом шаге нам приходится выполнять n сравнений (мы должны исходить из худшего предположения, что минимальный элемент может находиться на последнем месте), затем

n-1 и так далее. На последнем шаге число сравнений будет равно двум, так как останется только два элемента. Сложность этого алгоритма оценивается как $O(n^2)$.

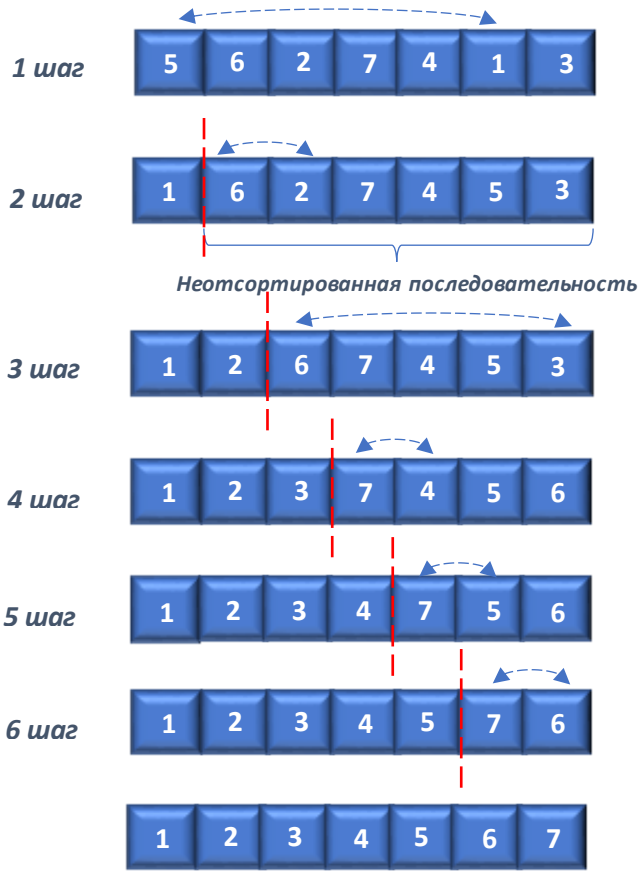


Рис.38. Демонстрация метода сортировки выбором

Сортировка вставкой

В основе идеи сортировки вставкой является разделение списка на две части – первая часть будет отсортированной, вторая – нет. На каждой итерации число будет перемещаться из неотсортированной части списка в отсортированную до тех пор, пока весь массив не будет отсортирован. Изначально, когда еще нет информации обо всех элементах, первое число списка рассматривается как первый и последний элемент отсортированного списка. Затем первое число в неотсортированной части списка сравнивается с единственным пока еще элементом отсортированного списка и переносится в отсортированную часть слева или справа от первого числа в зависимости от результата сравнения. При выполнении последующих итераций число из неотсортированного списка будет перемещаться в отсортированную часть путем сдвига остальных

элементов на позицию, которое занимало это число, не нарушая при этом упорядоченность в отсортированной части последовательности. Принцип такой сортировки продемонстрирован на рис.39.

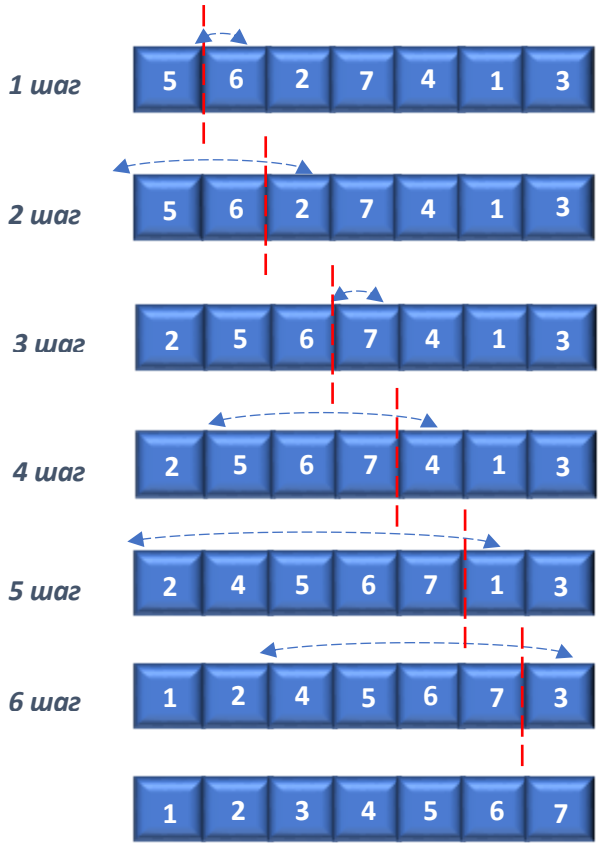


Рис.39. Демонстрация метода сортировки вставкой

Следует помнить, что в отсортированной части мы движемся справа налево, а неотсортированной – слева направо.

Сложность этого алгоритма также оценивается как $O(n^2)$.

Сортировка слиянием

В основе выполнения алгоритма сортировки слиянием положен принцип «разделяй и властвуй», который заключается в разбиение основной задачи на некоторое количество простых подзадач, решаемых рекурсивно. В частности, реализация алгоритма сводится к осуществлению следующих шагов:

- последовательность сортируемых элементов n делится на две части;
- обе полученные последовательности рекурсивно сортируются слиянием;

- полученные две отсортированные последовательности соединяются с целью получения отсортированной последовательности.

Фактически, как и обычно при рекурсивном подходе, следует свести выполнение задачи к тривиальному случаю, то есть разбивать последовательность на части до тех пор, пока в ней не останется n<2 элементов, а затем переходить к слиянию пар последовательностей.

На рис.40 показаны этапы разбиения последовательности на подпоследовательности.

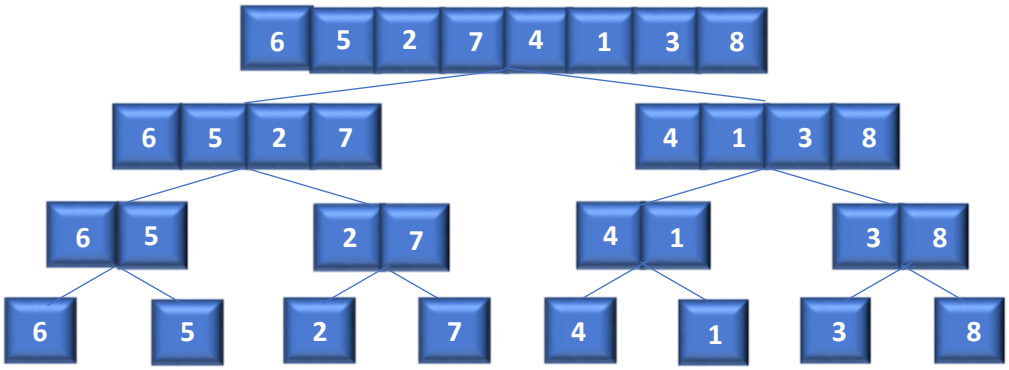


Рис.40. Демонстрация рекурсивных вызовов на этапе разбиения последовательности

Как видно из рис.40, на последнем этапе получены подпоследовательности, состоящие из одного элемента, который, естественно, является уже отсортированным в своей подпоследовательности, а это значит, что можно переходить к слиянию пар подпоследовательности.

При слиянии пар подпоследовательности (начинаем с левой части) следует сравнивать их первые элементы и помещать отсортированные элементы в один список. Таким образом, будут сформированы новые подпоследовательности, каждая из которых состоит уже из двух элементов. Аналогично и они объединяются в список, в каждом из которых уже 4 элемента и т.д. На последнем этапе оставшиеся две подпоследовательности соединяются в один отсортированный список (рис.41).

В том случае, если в списке находится нечетное количество элементов, то при делении пополам можно сделать один из списков больше на один элемент. Это не повлияет на суть выполнения алгоритма.

Подобный подход позволяет свести данный алгоритм к сложности O(n log n). Он является эффективным для списков, в которых более 50 элементов. Однако, за подобное уменьшение вычислительной сложности придется заплатить увеличением используемой памяти, поскольку на формирование каждой отсортированной подпоследовательности приходится выделять дополнительную память.

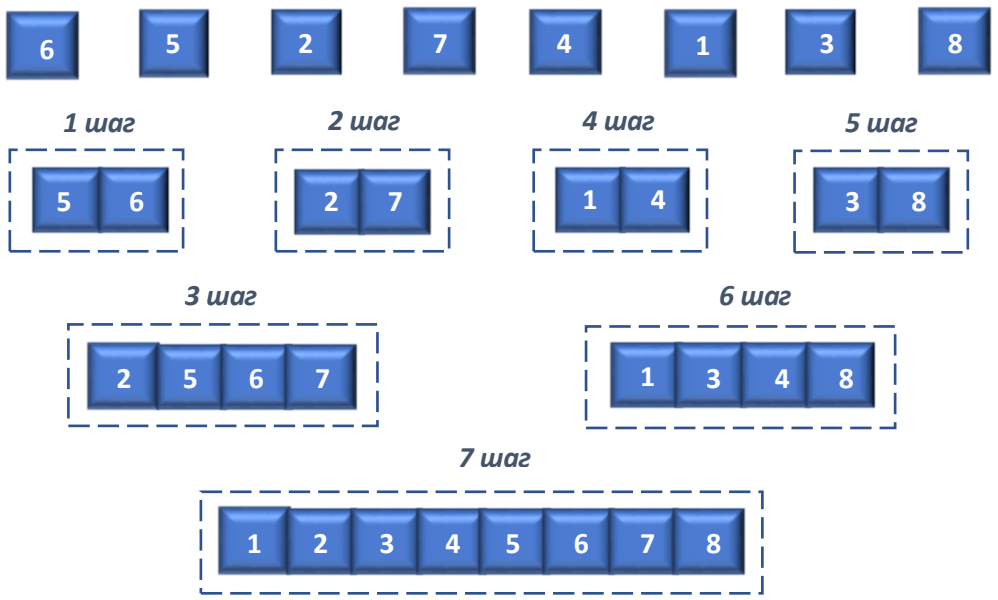


Рис.41. Демонстрация слияния списков при выполнении алгоритма сортировки слиянием

Быстрая сортировка

Алгоритм быстрой сортировки, известный как алгоритма Хоара является рекурсивным и считается одним из самых быстрых и универсальных.

Для начала выбирается некий опорный элемент (pivot), как правило, находящийся в середине сортируемого набора чисел. Тем самым последовательность чисел делится примерно пополам. Затем осуществляются два противоположных процесса. Один из них в левой части определяет элемент, меньший опорного, а второй в правой части – больше отмеченного опорного. При обнаружении этих элементов осуществляется их взаимообмен. Далее поиск продолжается с того места, где процессы остановились. В момент встречи процессов, все элементы в левой части будут меньше опорного, а в правой – больше, то есть сравнивать их больше не надо. Затем такие же операции надо провести по отношению к обеим частям массива и так далее до тех пор, пока

процесс не сведется к одному элементу в каждом списке. Затем, полученные списки, состоящие из одного элемента следует просто сложить.

На рис.42 показан принцип выполнения алгоритма быстрой сортировки. На первом этапе в середине списка выбранный опорным элемент отмечен буквой «Р» (в нашем примере число 7).

Относительно опорного элемента слева и справа сравниваются два крайних элемента. Расположенный слева элемент со значением 9 больше опорного (7), а справа со значением 3 меньше опорного, следовательно, их нужно обменять местами.

На втором шаге сравниваются следующие слева и справа элементы 2 и 1, соответственно. Элемент со значением 2 меньше опорного, поэтому остается на своей позиции, а с числом 1 сравнивается следующее число в левой части – 8. Поскольку 8 больше опорного, а 1 меньше, то их необходимо обменять местами.

Элементы в левой части на этом заканчиваются, а в правой части остается число 4, которое сравнивается с самим опорным элементом. Поскольку опорный элемент 7 больше числа 4, они также меняются местами.

Процессы сошлись, стрелки поменялись местами, и следовательно, в левой части на этом этапе расположены все элементы меньше 7, а в правой – больше 7.

Таким образом, массив можно разбить на две части и произвести сортировку элементов относительно новых опорных элементов. В нашем примере один набор на этом этапе будет представлен числами 3, 2, 1, 4, а второй – 7, 8, 9. В первом из наборов опорным отмечен элемент 2, а во втором – 8.

Поскольку в первом из наборов сравниваемые значения 3 и 1 оказываются «не на своих» местах – 3 больше 2 и находится слева, а 1 меньше 2 и находится справа, – то их необходимо обменять местами.

Оставшаяся в правой части 4 больше опорного элемента, следовательно, находится на нужном месте и процесс снова останавливается. Снова происходит деление набора на два набора – 1, 2 и 3, 4.

Проверка в обоих наборах снова приводит к схождению процессов и это дает возможность разбиения их на еще более мелкие, в состав которых входит всего по одному элементу.

Аналогичным образом действия производятся и в наборе 7, 8, 9, полученном при первом делении массива на две части.

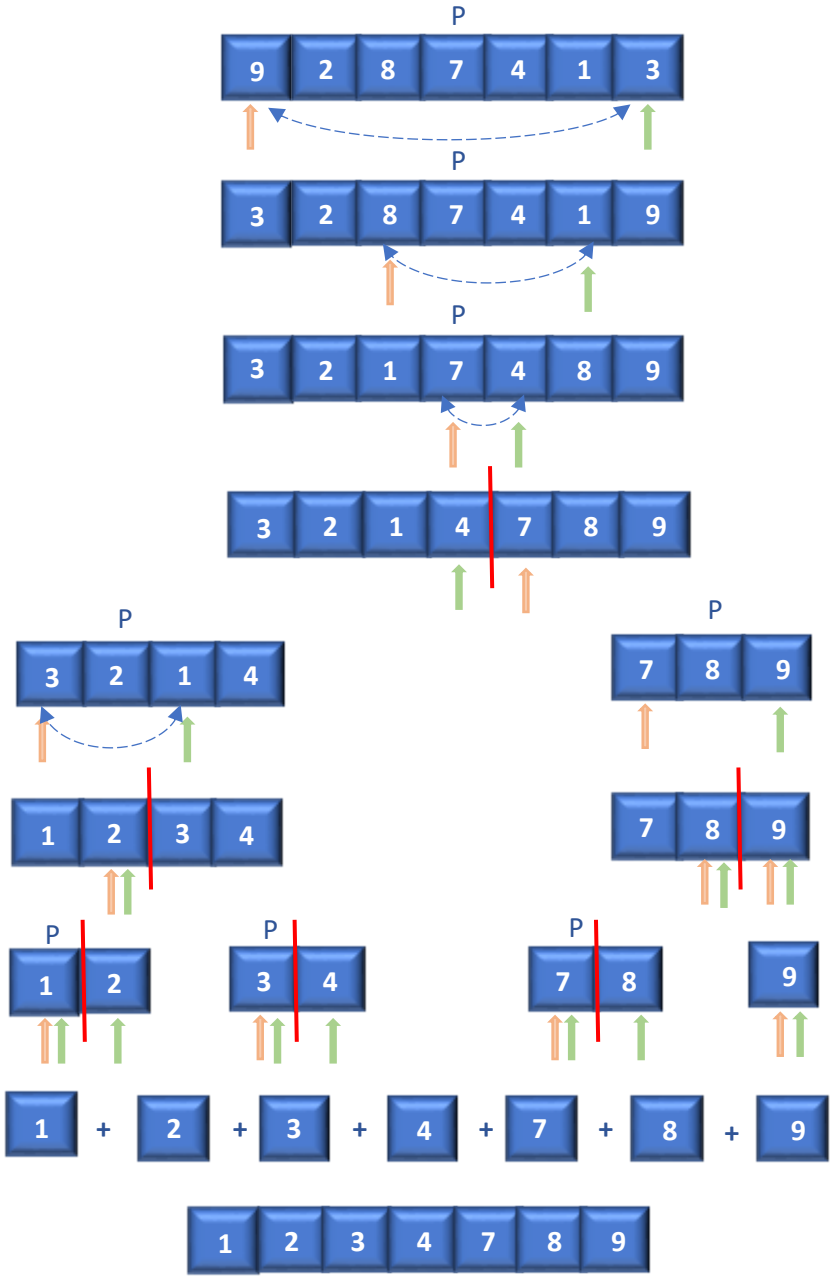


Рис.42. Демонстрация реализации алгоритма быстрой сортировки

В результате, на предпоследнем шаге у нас семь наборов, в каждом из которых по одному числу. Более того, эти числа отсортированы друг относительно друга.

Все, что нам остается сделать на последнем шаге – это объединить все семь наборов в один. Задача выполнена – исходный массив 9, 2, 8, 7, 4, 1, 3 отсортирован по возрастанию.

Сложности подобного алгоритма составляет O(n log n). Однако, есть и некоторые моменты, которые следует учитывать. Время выполнения алгоритма для только что отсортированного массива будет больше, чем для массива со случайными числами. Кроме того, многое зависит и от удачного выбора опорного элемента. Обычно его выбирают по среднему арифметическому индексов первого и последнего элементов. Еще лучший результат даст выбор элемента со средним значением в качестве опорного элемента. И все же, несмотря на некоторые оговорки, алгоритм быстрой сортировки в большинстве случаев считается наиболее простым и быстро исполняемым.

Сортировка пузырьком

Сортировка пузырьком или же простыми обменами является простейшим классическим примером сортировки произвольного списка. Эффективность алгоритма находится в обратно пропорциональной зависимости от количества элементов сортируемого списка.

Идея данной сортировки заключается в сравнении двух соседних элементов массива начиная с конца. В том случае, если последний элемент окажется меньше предпоследнего, они поменяются местами. В результате повторного сравнения наименьший элемент будет постепенно перемещаться к началу списка, подобно пузырьку. На рис.43 показано, как последовательно на первой итерации происходит перемещение наименьшего графическая реализация данной сортировки.

На рис.44 продемонстрирован процесс перемещения остальных элементов списка на следующих итерациях.

Количество итераций в общем случае равно числу элементов. Сложность составляет $O(n^2)$.

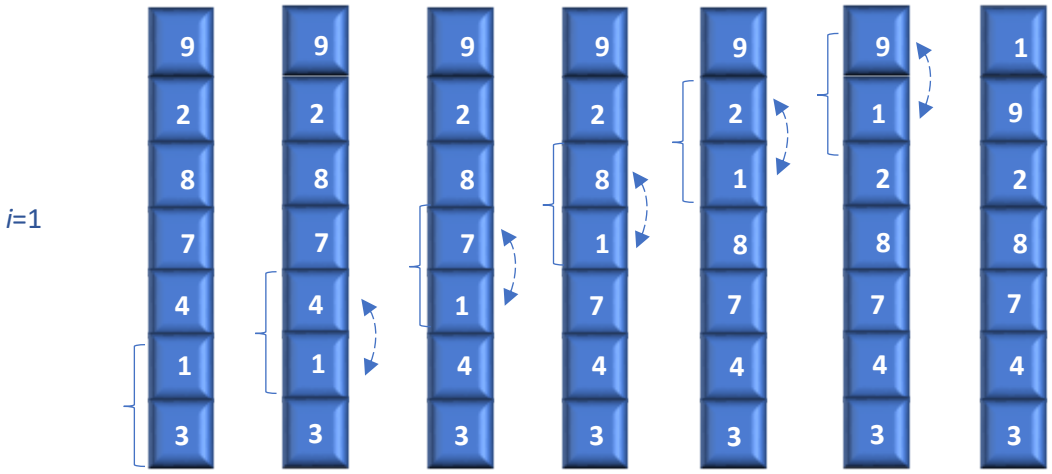


Рис.43. Перемещение наименьшего элемента по списку вверх на первой итерации

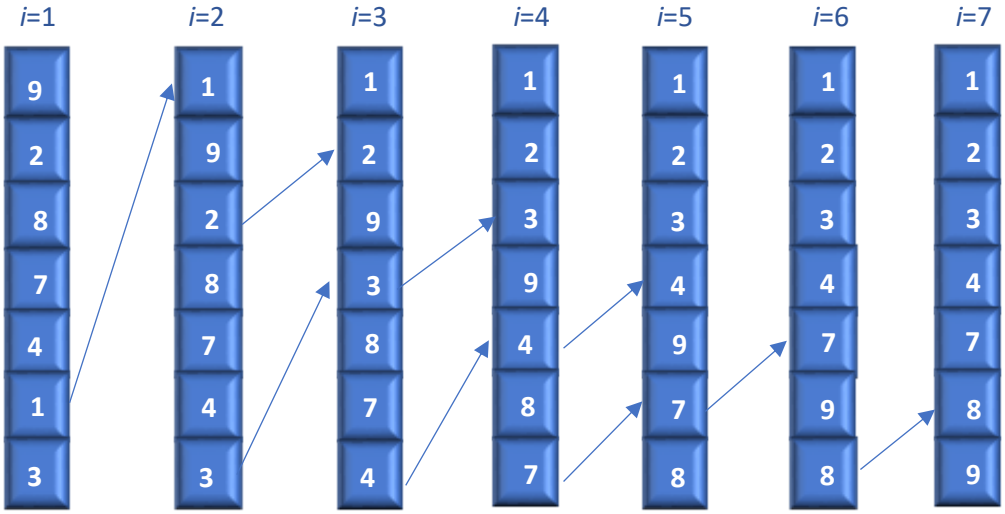


Рис.44. Демонстрация этапов сортировки пузырьком

Иногда возникают ситуации, когда на последних итерациях никакие перестановки уже не происходят, так как все элементы оказываются отсортированными за меньшее количество шагов. В этом случае можно ввести переменную-«флаг», который будет фиксировать наличие перестановок. В том случае, если перестановка на каком-то этапе не была произведена, значит массив уже отсортирован и последующие итерации можно не производить.

Нижняя граница сортировки подсчетом и ее преодоление

Все рассмотренные выше алгоритмы сортировки сравнением, работали в худшем случае за время $O(n^2)$, а в лучшем – за $O(n \log n)$. Наименьшее количество операций для решения

конкретной задачи определяется нижней границей в случаях, когда сортировка производится сравнениями пар элементов, она также составляет Ω(n log n).

Однако, при некоторых ограниченных условиях можно превзойти нижнюю границу. То есть, при ограниченном случае n элементов можно отсортировать за время Ω(n), не прибегая к сравнению пар элементов.

Одним из таких методов является сортировка подсчетом. Это вид сортировки, работающий только с числами. Для его реализации необходимо знать количество отличающихся чисел. В примере, реализованном на рис.45 имеется 4 отличающихся числа – 1, 2, 3, 4. Следует завести новый массив по количеству имеющихся отличающихся чисел. В дальнейшем осуществляется проход по списку и занесения встречающихся чисел в соответствующую ему ячейку в новом массиве-счетчике. После этого останется лишь разместить элементы в подсчитанном количестве. В результате будет сформирован отсортированный массив.

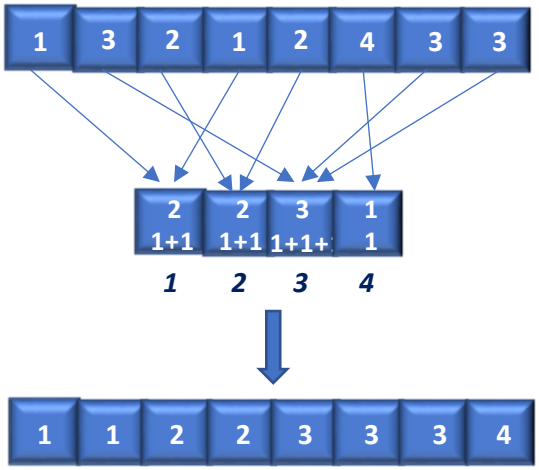


Рис.45. Графическая демонстрация работы алгоритма сортировки подсчетом

На первом этапе будет потрачено n операций (проход по первому массиву), на втором – k операций и на третьем – снова n. То есть сложность алгоритма O(n+k+n)= Ω(n+k+n), что эквивалентно O(n+k)=Ω(n+k), где k – это количество разных чисел. Если k является константой, то ею можно пренебречь и тогда нижняя граница будет определяться как Ω(n).

Ориентированные ациклические графы. *Топологическая сортировка. Представление ориентированных графов. Время работы топологической сортировки. Кратчайший путь в ориентированном ациклическом графе*

6

Топологическая сортировка. Представление ориентированных графов. Время работы топологической сортировки

Топологическая сортировка представляет собой один из основных алгоритмов на графах, применяемых для упорядочивания вершин ориентированного, ациклического графа, согласно частичному порядку, заданному ребрами орграфа на множестве его вершин.

При сортировке графа образуется определенная иерархия с некоторым количеством уровней, что позволяет представить сложный запутанный граф в тривиальном виде.

На рис. 46 представлен произвольный ориентированный граф. Определить по такому представлению графа является ли он ацикличным или в нем присутствуют циклы, а также каким образом соединены вершины довольно затруднительно. Именно поэтому применяют топологическую сортировку.

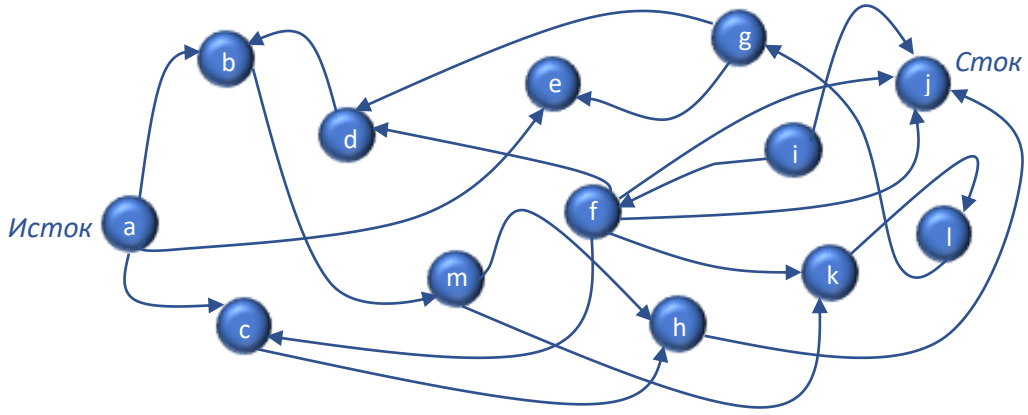
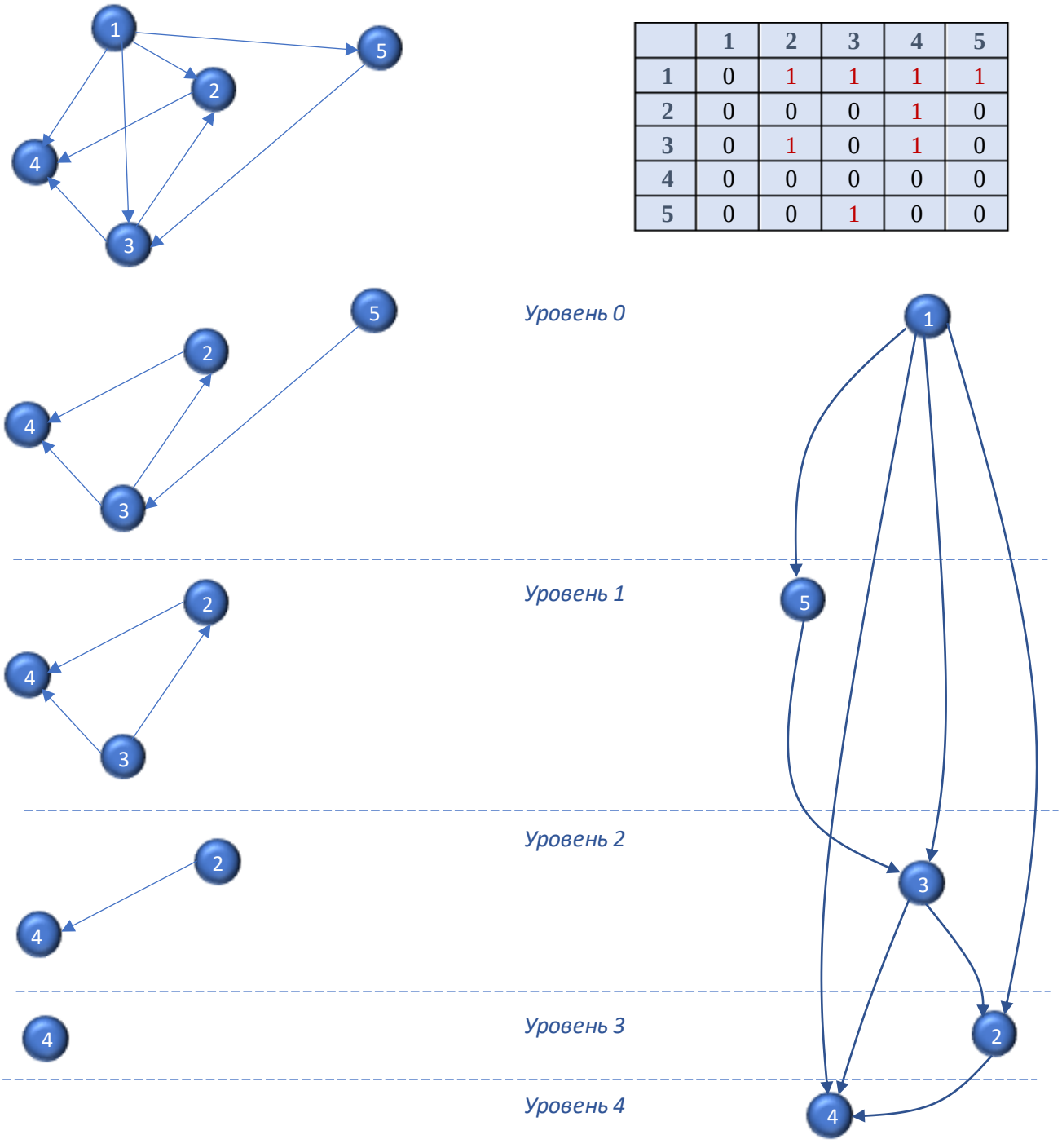


Рис.46. Ориентированный граф

Следует заметить, что каждую вершину можно охарактеризовать числом входящих и выходящих из нее дуг. Если у вершины есть только исходящие дуги и нет ни одной входящей

дуги, то она называется *истоком*, а если у вершины имеются только входящие дуги – *стоком*. Например, вершина «a» является истоком, а вершина «j» – стоком (см. рис. 46).

На рис.47 продемонстрирован принцип, по которому производится топологическая сортировка. При этом используется один из способов представления графов – таблица смежности. Принцип построения таких таблиц, а также другие способы хранения графов подробно были рассмотрены ранее в разделе 3.



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

Рис.47. Топологическая сортировка графа

Как видно из рис.47, топологическая сортировка производится поэтапно. На первом этапе необходимо обнаружить исток и удалить вершину вместе с ее ребрами. На втором этапе следует найти второй исток и также удалить его со своими ребрами. Процесс надо продолжать до тех пор пока все вершины не будут перенесены в правую часть в соответствии с их уровнями. Для получения топологически сортированного графа (правый граф на рис.47) следует соединить вершины, расположенные на разных уровнях в соответствии с матрицей смежности.

Так, в нашем конкретном примере, изображенном на рис.47, сначала определяется исток – это вершина 1, которая удаляется вместе с исходящими из нее ребрами, ведущими к вершинам 2, 3 и 4. Сама вершина 1 помещается на нулевой уровень (см. правую часть рис.47).

Затем определяется вершина, которая после удаления первого истока сама стала истоком. В нашем примере это вершина 5, поскольку только она не содержит входящих ребер. Ее помещаем на уровень 1, а ее саму в графе вместе с ребром, ведущим к третьей вершине – удаляем.

На следующем этапе удаляется вершина 3, так как лишь у нее теперь только исходящие ребра, ведущие к вершинам 2 и 4. Следовательно, вершина 3, как и ее исходящие ребра из графа удаляются, а сама вершина помещается на уровень 2.

Далее у нас остается только одна вершина-исток – это вершина 2 с исходящим ребром к вершине 4. После удаления ее и размещения на уровне 3, остается одна лишь вершина 4, которая помещается на уровень 4.

После этого размещенные по своим уровням вершины соединяются ребрами в соответствии с матрицей смежности. В результате топологическая сортировка графа будет проведена.

Если на каком-либо этапе окажется, что в графе все оставшиеся вершины являются стоками, то это будет свидетельствовать о том, что граф содержит циклы. В этом случае проведение топологической сортировки невозможно.

Сложность подобных алгоритмов достаточно велика. Она составляет $O(n^2)$ в простой реализации, поскольку на каждой из n итераций нужно найти исток. Однако, такой метод поз-

воляет лучше понять другие более эффективные алгоритмы, например, как алгоритм поиска (обхода) в глубину и в алгоритм обхода в ширину.

Алгоритм обхода в ширину (Breadth first search (BFS))

Алгоритм обхода в ширину представляет собой алгоритм, позволяющий пройти по все вершинам графа. Его можно условно представить в виде волны, которая начинается с какой-либо вершины графа и последовательно распространяется на последующие уровни, где расположены соседние с ней вершины. Затем волна последовательно распространяется на следующие соседние вершины и т.д. Графически это изображено на рис. 48.

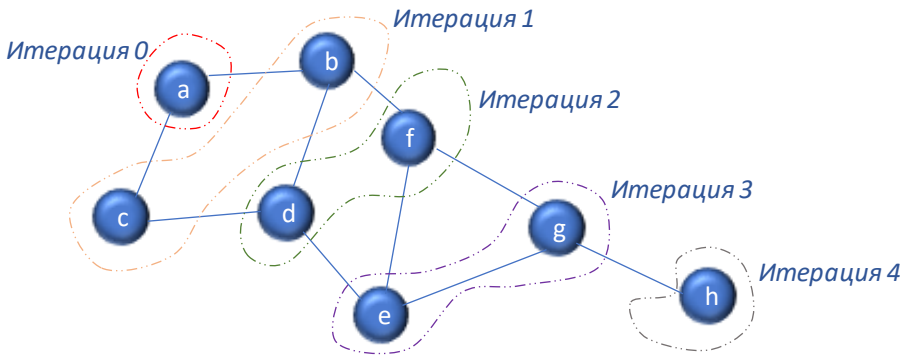


Рис.48. Демонстрация распространения «волн» на каждой итерации

Для реализации такого алгоритма следует воспользоваться такой структурой данных, как очередь. На рис. 49 показан принцип занесения и удаления вершин из очереди q (от англ. queue).

Как видно из рис.49, на 7 шаге в очереди не остается элементов и это является показателем того, что все пути в графе пройдены. Алгоритм обошел все вершины и пометил их метками. Последний уровень помечен 4 и эта метка будет показывать минимальный путь до стартовой вершины, т.е. минимальный путь от h до a будет составлять 4 ребра.

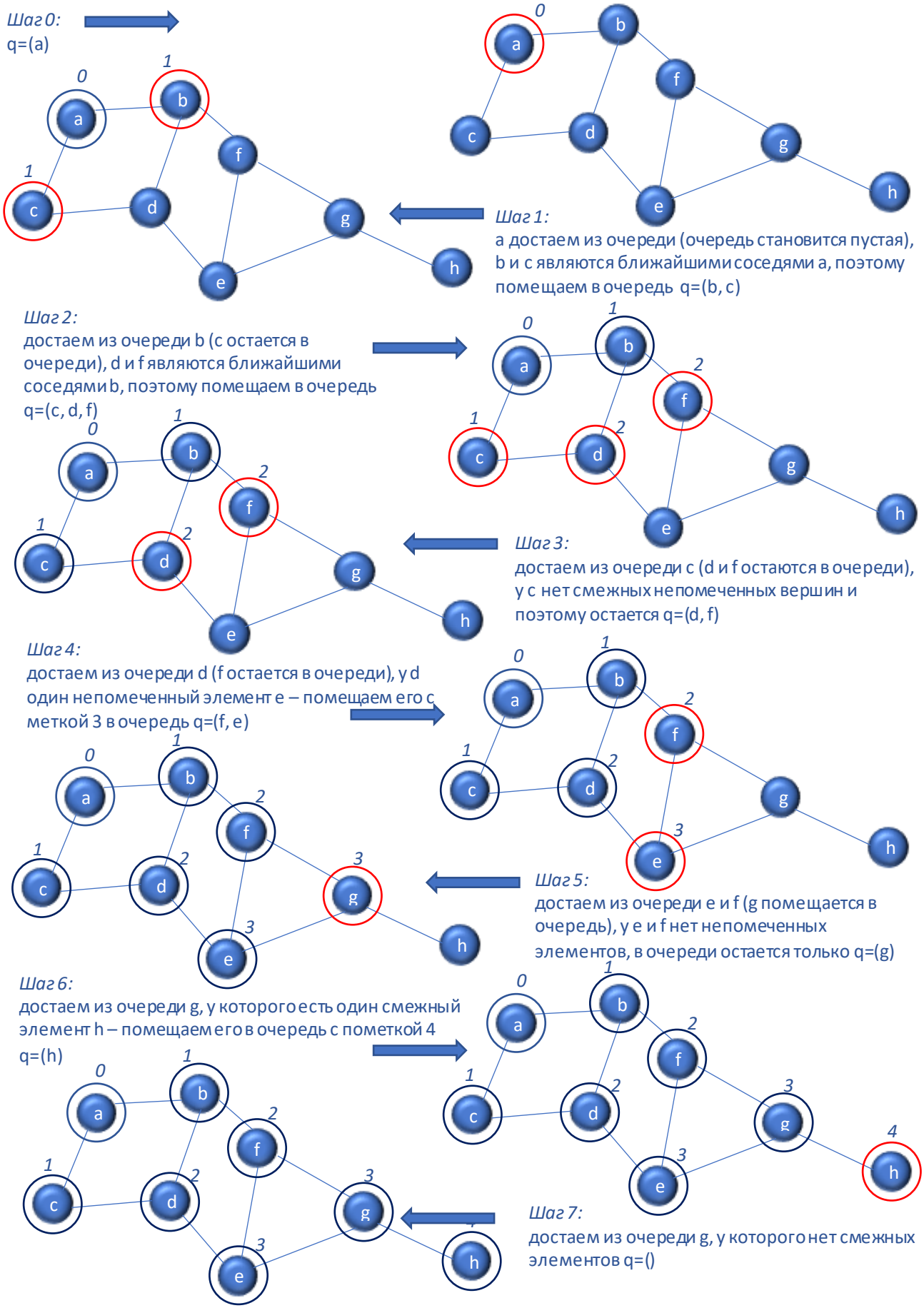


Рис.49. Схема последовательного добавления в очередь и удаления из очереди вершин графа

Что касается сложности такого алгоритма, то надо учесть, что каждый элемент нам нужно положить в очередь, а затем извлечь, т.е. в первую очередь просматривается зависимость от количества вершин V. Кроме того, для каждой вершины мы проверяем смежные с ней элементы. Проверка ребра осуществляется по 2 раза (например, ребро ab проверяется и из вершины a, а затем и из вершины b), что добавляет нам в оценке сложности алгоритма слагаемое 2E. Итоговая сложность алгоритма составит O(V+2E), что эквивалентно сложности O(V+E).

Алгоритм обхода в глубину (Depth first search (DFS))

Алгоритм поиска или же обхода графа в глубину также предоставляет возможность обхода графа по всем его вершинам, однако, несколько иным способом, нежели это было описано выше при реализации алгоритма обхода в ширину.

При поиске в ширину, начиная с одной из вершин графа, мы проходили по его смежным вершинам уровень за уровнем.

Поиск в глубину предполагает передвижение из первой вершины вглубь по последовательно расположенным вершинам до тех пор, пока не окажемся в вершине, у которой помеченными будут все соседние вершины. После этого осуществляется возврат по вершинам в поиске непомеченных соседей какой-либо из вершин.

Реализация алгоритма предполагает использование такой структуры данных, как стек.

На рис.50 продемонстрирована схема реализации алгоритма обхода графа в глубину с последовательным занесением в стек и извлечением из стека соответствующих элементов на каждой из итераций.

При оценке сложности подобного алгоритма следует учесть, что нам дважды приходится обращаться к каждой из вершин (во время помещения ее в стек и удаления из стека), а также дважды проходить по каждому ребру. Следовательно, сложность алгоритма составляет O(2V+2E), что эквивалентно сложности O(V+E).

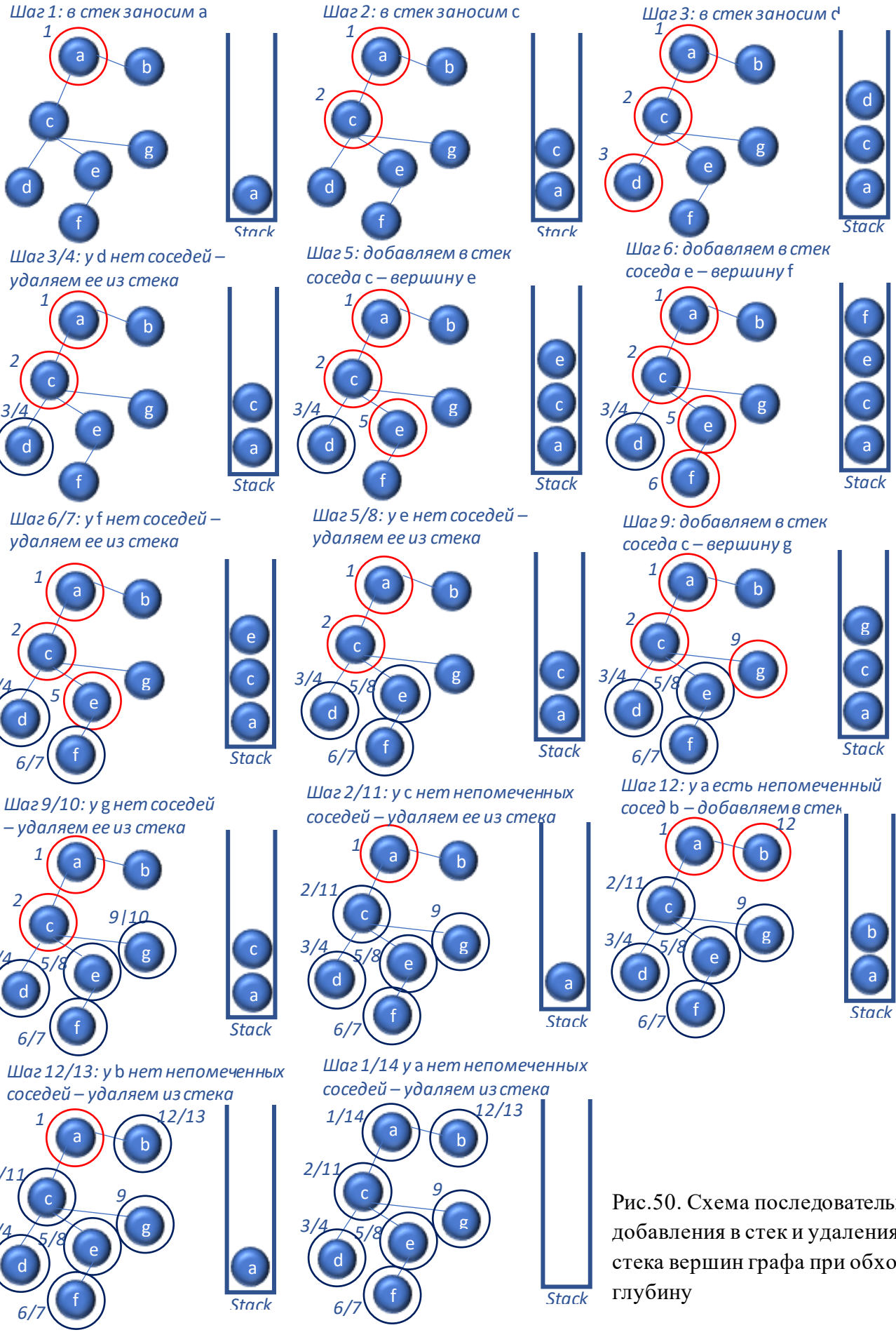


Рис.50. Схема последовательного добавления в стек и удаления из стека вершин графа при обходе в глубину

Кратчайший путь в ориентированном ациклическом графе

Поиск кратчайшего пути легко осуществляется в рамках теории графов. В случае невзвешенных графов под длиной пути будем понимать количество ребер, которое надо пройти в заданном пути.

Частично этот вопрос уже был рассмотрен выше при анализе работы обхода графа в ширину (см. рис.49). Тогда было отмечено, что метка последнего помеченного уровня и будет показывать минимальный путь до стартовой вершины.

Поиск кратчайшего пути с помощью *волнового алгоритма*[1] рассмотрим на примере ориентированного ацикличного графа, изображенного на рис.51. Задача заключается в поиске кратчайшего пути от вершины *a* до другой вершины графа. Пока что под длиной пути будем понимать количество ребер, которое надо пройти в заданном пути. Так например, длина кратчайшего пути из вершины a в вершину g равна двум, поскольку эти две вершины соединены двумя ребрами. Однако, следует заметить, что путь от *a* до *g* не равен пути из *g* в *a*, так как граф ориентированный и его ребра указывают и направление движения. В этом случае говорят, что путь из g в a равен бесконечности (рис.51).

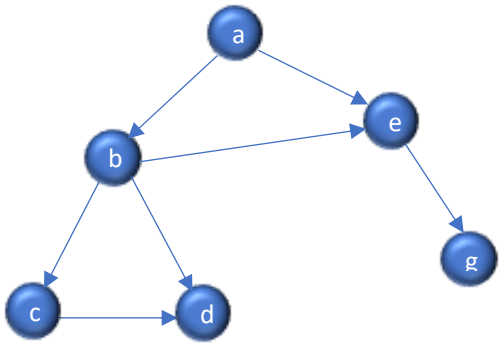


Рис.50. Ориентированный ацикличный граф

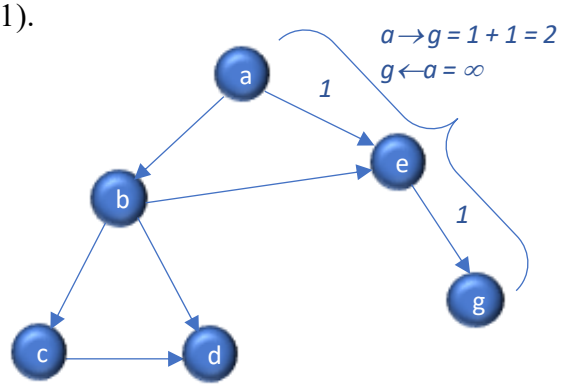


Рис.51. Поиск пути в ориентированном графе

Также из рис.50 и рис.51 следует, что в одну и ту же вершину может существовать несколько путей. Так, например, из вершины *a* в вершину *d* существует два пути: *a*→*b*→*d* и *a*→*b*→*c*→*d*, равные 2 и 3 соответственно. В этом случае кратчайшим является путь *a*→*b*→*d*.

[1] Алгоритм называется волновым, потому что имитирует распространение волны на каждые последующие уровни графа.

Рассмотрим алгоритм, решающий задачу поиска кратчайшего пути более подробно. Первый очевидный факт – кратчайший путь из вершины *a* в нее же равен нулю. Путь между ближайшими соседями составляет единицу. Соответственно, эти соседние вершины помечаются единицей (рис.52). До вершин, соседних с *b* и *e* также можно дойти за единицу. Следовательно, путь до них от вершины *a* составляет два. Следует также уточнить один момент: производится пометка только непомеченных соседей рассматриваемой вершины. Например, вершина *e* была помечена как соседняя с *a*. При поиске соседей *b* мы помечаем двойкой лишь вершины *c* и *d*, игнорируя соседнюю с *b* вершину *e*, поскольку она уже была ранее отмечена единицей. То есть на каждом уровне производится пометка только непомеченных вершин.

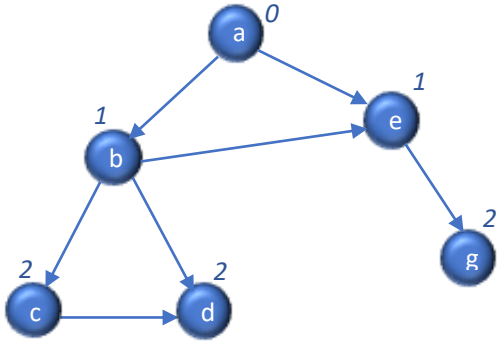


Рис.52. Поиск кратчайшего пути

Итак, кратчайшее расстояние между вершинами не может быть числом, большим, чем количество самих вершин. Более того, путь будет равен количеству вершин минус единица. На кратчайшем пути каждая вершина может встретиться только один раз. Останавливаться следует в том случае, если ни у одной из вершин больше не будет непомеченных соседей.

При поиске кратчайшего пути необходимо искать предшественников конечной вершины. Они должны быть помечены меткой, которая будет на единицу меньше, чем у самой вершины. Так, например, при поиске кратчайшего пути от *a* до *d*, помеченной двойкой, надо посмотреть на соседей *d*. Вершина *c* является неподходящей, поскольку у нее также отметка 2. Следовательно, надо двигаться в направлении *b* с отметкой 1. Затем искать предшественников *b* с отметкой на единицу меньше, то есть с 0, что и является вершиной *a*.

Таким образом, при реализации алгоритма путь восстанавливается с конца до исходной вершины.

Алгоритм Дейкстры

Алгоритм Дейкстры применяется для определения кратчайшего пути между одной из вершин до всех остальных. Рассмотрим работу алгоритма на примере ориентированного взвешенного графа (рис.53). Взвешенным называется граф, у которого ребрам (дугам) сопоставлены какие-либо числа. На практике эти числа могут представлять собой конкретные расстояния между пунктами назначения, вес, стоимость и т.д.

В нашем конкретном примере порядок графа составляет 6, поскольку в нем содержится 6 вершин, а размер графа составляет 11 по числу ребер (дуг).

Реализация алгоритма предполагает создание и заполнение специальной схемы-таблицы. Все этапы выполнения пошагового выполнения алгоритма продемонстрированы на рис. 54. Кратчайшие пути будут определяться от вершины 1. На начальном этапе в таблицу заносятся все вершины. Вершина, с которой производится работа помечается постоянной меткой в виде квадрата.

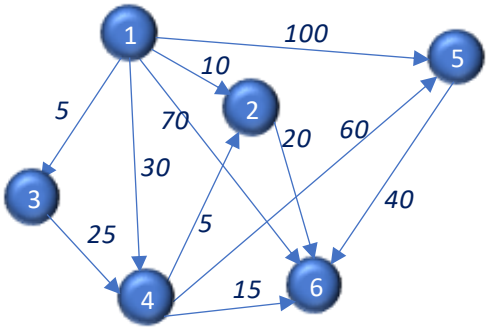


Рис.53. Ориентированный взвешенный граф

На первом шаге заполнения таблицы работа производится с вершиной 1, которой присваивается постоянная метка (отмечена красным кружком) (см. рис.54). Значение нуля, которое присваивается этой вершине (путь из вершины в саму себя равен 0) также помечается постоянной меткой – квадратом в таблице на рис.54. Расстояние до всех остальных вершин принимается равным бесконечности на начальном первом шаге.

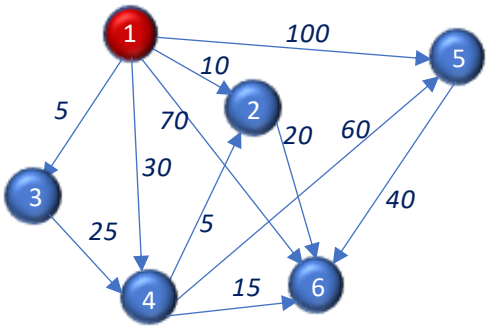
На втором шаге необходимо посмотреть, в какие вершины из первой существует путь и добавить к нулевой метке условно указанные значения на дугах графа. Причем записывать возможно только те значения, которые меньше, чем в предыдущей строке таблицы. Поскольку

все значения, указанные до этого (знак бесконечности) будут больше, чем новые, указанные на дугах, то все они будут записаны. Таким образом, из первой вершины есть пути во вторую, третью, четвертую, пятую и шестую (10, 5, 30, 100 и 70 соответственно). Если бы какого-то из путей не существовало бы, то в соответствующей ему позиции надо было бы проставить предыдущие значения – знак бесконечности в данном конкретном случае. Минимальное значение, полученное во второй строке таблицы (не считая строку заголовка) помечается, как новая постоянная метка. В нашем случае – это число 5. Это же является и кратчайшим путем до вершины 3, которую также помечаем постоянной меткой.

На третьем шаге просматриваем пути из вершины 3 в остальные. Существует только один путь – в вершину 4, составляющий 25 единиц. К этому значению необходимо добавить значение, полученное на предыдущем пути – 25+5=30. По правилам, вносить новое значение можно только в том случае, если оно меньше или равно предыдущему, поэтому вносим полученное значение 30 в таблицу. Остальные значения просто переносим с предыдущей строки.

Как видим, минимальным значением в третьей строке является 10 (шаг 4), соответствующее второй вершине. Следовательно, кратчайший путь до второй вершины составит 10 единиц. Проставляем постоянные метки значению и самой вершине и переходим к поиску путей из вершины 2. Из второй вершины ведет только один путь в вершину 6, равный 20. Прибавляем к 20 значение предыдущей метки (10) и получаем значение 30, которое записываем в таблицу. Остальные значения просто переносим из предыдущей строки.

На пятом шаге оказывается, что из вершины 4 существуют три пути – в вершины 2, 5 и 6. Вершина 2 нас не интересует, поскольку кратчайший путь до нее уже известен. Расстояние до вершины 5 составляет 60 единиц, а с учетом предыдущей метки – 90, а следовательно это новое значение будет занесено в таблицу. Расстояние до вершины 6 составляет 15 единиц, но с учетом предыдущей метки 15+30=45, что больше предыдущего значения, а следовательно, новое значение будет проигнорировано.



Шаг 1

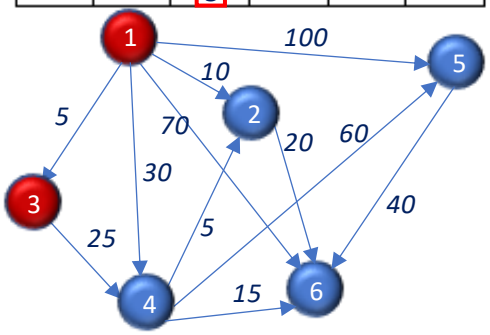
1	2	3	4	5	6
0	∞	∞	∞	∞	∞

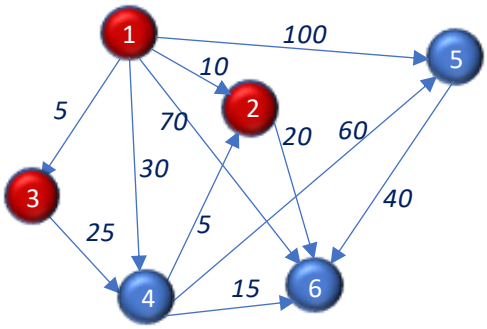
Шаг 2

1	2	3	4	5	6
0	∞	∞	∞	∞	∞
	10	5	30	100	70

Шаг 3

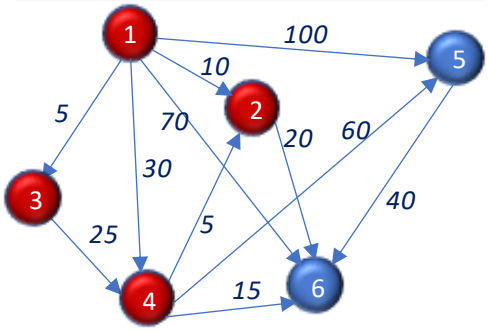
1	2	3	4	5	6
0	∞	∞	∞	∞	∞
	10	5	30	100	70
	10		30	100	70





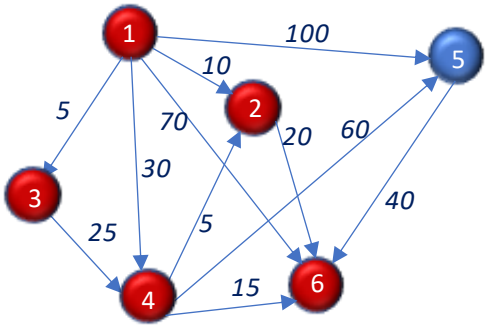
Шаг 4

1	2	3	4	5	6
0	∞	∞	∞	∞	∞
	10	5	30	100	70
	10		30	100	70
			30	100	30



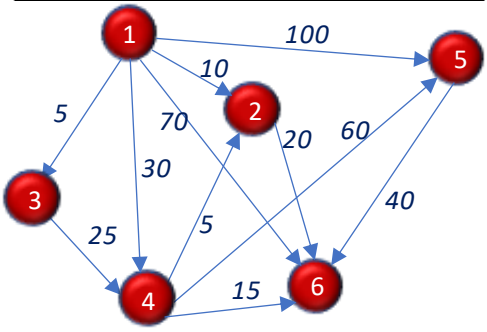
Шаг 5

1	2	3	4	5	6
0	∞	∞	∞	∞	∞
	10	5	30	100	70
	10		30	100	70
			30	100	30
				90	30



Шаг 6

1	2	3	4	5	6
0	∞	∞	∞	∞	∞
	10	5	30	100	70
	10		30	100	70
			30	90	30
				90	30
				90	



Шаг 7

1	2	3	4	5	6
0	∞	∞	∞	∞	∞
	10	5	30	100	70
	10		30	100	70
			30	90	30
				90	30
				90	

Рис.54. Шаги реализации алгоритма Дейкстры

Шестой шаг не вносит изменений в таблицу, поскольку из вершины 6 путей нет. Следовательно, кратчайшим путем до вершины 6 останется полученной на предыдущих шагах значение 30. Метка переходит к вершине 5.

На седьмом шаге рассматриваются пути из вершины 5. Единственный существующий путь ведет к вершине 6 и составляет 40 единиц. Он игнорируется, поскольку на предыдущем шаге был установлен кратчайший путь к вершине 6 в 30 единиц. Соответственно, кратчайшим путем от вершины 1 к вершине 5 остается путь в 90 единиц.

Таким образом, путем реализации алгоритма Дейкстры были установлены оптимальные кратчайшие пути от искомой вершины 1 до всех остальных вершин.

Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда также предназначен для поиска кратчайшего пути в графе и, хотя он и не является лучше алгоритма Дейкстры в плане вычислительной сложности (сложность Дейкстры O(E+VlogV), Беллмана-Форда – O(V*E)), зато обладает по сравнению с ним преимуществом – позволяет искать кратчайший путь при наличии отрицательных чисел на дугах взвешенного графа.

При реализации алгоритма следует учитывать, что кратчайший путь между двумя вершинами при отсутствии циклов отрицательной не длиннее количества вершин в графе.

При поиске кратчайшего пути в соответствии с принципами динамического программирования необходимо разбить большую задачу на подзадачи – искать пути ограниченной пути. С этой целью будет осуществляться поиск сначала путей, в которых одно ребро, затем тех, в котором два ребра, далее три и т.д. Работу алгоритма рассмотрим на конкретном примере для ориентированного взвешенного графа, содержащего дуги с отрицательными весами (рис.55).

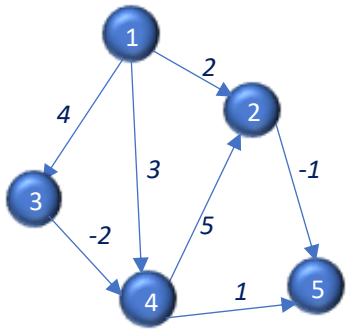


Рис.55. Ориентированный взвешенный граф с отрицательными дугами

На рис.56 продемонстрировано пошаговое выполнение алгоритма Беллмана-Форда.

На нулевом шаге рассматривается путь диной в 0 дуг. Поскольку поиск кратчайшего пути начинается с вершины 1, то путь в соответствии с условиями для нее будет равен нулю, а для остальных вершин – условно бесконечным.

На первом шаге рассматриваются пути от вершины 1 длиной в одну дугу, Соответствующие данные вносятся в таблицу с указанием направления движения от первой вершины.

На втором шаге рассматриваются пути, которые можно пройти от вершины 1 за две дуги. Если при сложении длин путей полученное значения окажется меньше, полученного на предыдущем пути, то оно заменяется на новое. Если полученное значение больше предыдущего, то оно просто игнорируется.

Так, например, с первой вершины до второй за две дуги можно добраться за путь, значение которого составляет 8 (1→4→2=8), но нас это не устраивает, поскольку на предыдущем шаге мы добирались с первой до второй вершины за путь равный 2 (1→2=2). Следовательно, новое значение 8 будет проигнорировано и в третью таблицу рисунка 56 будет перенесено значение 2. А вот путь в две дуги от первой до четвертой вершины оказался более оптимальным, чем на предыдущем первом шаге (1→3→4=2 против 1→4=3), поэтому под обозначением четвертой вершины на втором шаге в таблицу будет занесено новое значение – 2. Как видно, здесь же есть два пути до пятой вершины (1→2→5=1 и 1→4→5=4, соответственно) – выбираем наименьший, равный 1.

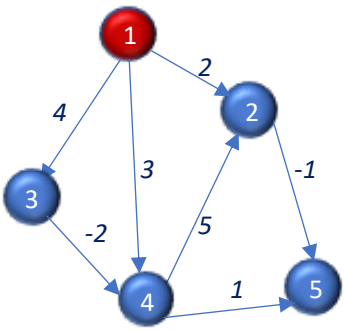
На третьем шаге осуществляем поиск путей за три дуги графа (1→3→4→2)=7 и (1→4→2→5)=7, соответственно, но ни один из них не является наименьшим. Поэтому в следующую строку таблицы переносятся все найденные на предыдущем шаге оптимальные значения.

Аналогично строим пути, состоящие из трех дуг, затем из четырех и т.д.

Конкретно в нашем случае не существует путей, состоящих больше, чем из трех дуг, поэтому после выполнения третьего шага необходимо остановиться.

Осуществив подъем в итоговой таблице по стрелкам вверх, для каждой из вершин можно восстановить кратчайший путь от искомой вершины 1.

В нашем примере путь от вершины 1 до 2, 3, 4, 5 составляет 2, 4, 2, 1, соответственно.



Шаг 0

вершины / шаги	1	2	3	4	5
0	0	∞	∞	∞	∞

Шаг 1:
(1→2)=2
(1→3)=4
(1→4)=3

вершины / шаги	1	2	3	4	5
0	0	∞	∞	∞	∞
1	0	2	4	3	∞

Шаг 2:
(1→2→5)=1
(1→3→4)=2
(1→4→2)=8
(1→4→5)=4

вершины / шаги	1	2	3	4	5
0	0	∞	∞	∞	∞
1	0	2	4	3	∞
2	0	2	4	2	1

Шаг 3:
(1→3→4→2)=7
(1→4→2→5)=7

вершины / шаги	1	2	3	4	5
0	0	∞	∞	∞	∞
1	0	2	4	3	∞
2	0	2	4	2	1
3	0	2	4	2	1

Рис.56. Схема пошагового выполнения алгоритма Беллмана-Форда при поиске кратчайшего пути из вершины 1 в вершины 2, 3, 4, 5

Алгоритм Флойда-Уоршелла

Алгоритм Флойда-Уоршелла также позволяет находить кратчайшие пути в графе между вершинами, причем работает он и с отрицательными числами на своих дугах.

При определение кратчайшего пути по матрице A вычисляется величина $A_{i, j, k}$, где i – начальная вершина, j – конечная вершина, а k – промежуточная вершина с порядковым номером от 1 до k. Так, запись $A_{2, 3, 0}$ означает, что путь отмечен путь с вершины 2 до вершины 3, без учета дополнительных вершин, а запись $A_{2, 3, 1}$ означает, что путь лежит из вершины 2 в 3 через вершину 1. Фактически нахождение кратчайших вершин между вершинами сводится к

построению матриц на основе матрицы смежности (количество матриц будет равно числу вершин) и вычисления элементов для каждой из матриц в соответствии с формулой:

$$A_k[i, j]=\min(A_{k-1}[i, j], A_{k-1}[i, k]+ A_{k-1}[k, j]). \quad (7)$$

Так, например, для первой из матриц значения элементов находятся по формуле

$$A_1[i, j]=\min(A_0[i, j], A_0[i, 1]+ A_0[1, j]); \quad (8)$$

для второй матрицы – по формуле

$$A_2[i, j]=\min(A_1[i, j], A_1[i, 2]+ A_1[2, j]) \quad (9)$$

и т.д.

Для реализации этого алгоритма можно воспользоваться упрощенной схемой, основанной на использовании строк и столбцов стационарности. Суть метода заключается в том, что на каждом их этапов отмечаются соответствующие стационарные строки и столбцы (для первой матрицы отмечается первый столбец и первая строка, для второй – второй столбец и вторая строка и т.д.), которые полностью переписываются в последующую матрицу. Если в них содержатся элементы, содержащие знак бесконечности (показатель отсутствия пути), то в новую матрицу переписываются целиком строки и столбцы, на которых они расположены. Значение же остальных элементов сравниваются с суммой значений, находящихся на пересечении соответствующего элемента с соответствующими, находящимися на пересечении элементами стационарной строки и столбца. В случае, если их сумма окажется меньше, то именно она будет записана на место проверяемого элемента.

Конкретный пример реализации такого алгоритма продемонстрирован на рис.57. Сначала для изображенного на рисунке графа составляется матрица смежности A_0, в которой первый столбец и первая строка будут считаться стационарными и поэтому полностью скопируются в матрицу A_1. Четвертый элемент первой строки является знаком бесконечности, следовательно, в соответствии с вышесказанным, все элементы расположенные под ним (то есть элементы находящиеся на пересечении четвертого столбца и второй строки, четвертого столбца и третьей строки, четвертого столбца и четвертой строки, переносятся также в матри-

цу A_1 без каких-либо изменений. Элементы первого столбца, расположенные на второй и третьей строках также содержат знаки бесконечности, следовательно, и все оставшиеся элементы второй и третьей строк должны быть перенесены в матрицу A_1. Остается вопросом, какие же числа необходимо перенести в оставшиеся два элемента, расположенные на пересечении четвертой строки и второго столбца и четвертой строки и третьего столбца (на рис.57 в матрице A_0 они отмечены контрастным цветом). Для определения этих значений необходимо произвести следующие действия. Сначала от элемента, расположенного на пересечении четвертой строки и второго столбца и равного на этом этапе 5 восстанавливаются перпендикуляры до отмеченных стационарных столбца и строки со значениями. Затем необходимо сложить значения, на которые указывают перпендикуляры (3 и 4, соответственно) и сравнить их со значением 5. Поскольку полученное значение суммы оказывается больше, чем значение ячейки (3+4=7, а 7>5), то в матрицу A_1 на позицию, находящуюся на пересечении четвертой строки и второго столбца переносится старое значение – 5.

Аналогичные действия производятся и для элемента, расположенного на пересечении четвертой строки и третьего столбца (начальное значение бесконечность). Восстанавливаются перпендикуляры от этой ячейки до стационарных столбца и строки; значения, на которые указывают перпендикуляры (3 и 1, соответственно) складываются; полученная сумма сравнивается с текущим значением ячейки. Поскольку любая сумма (в нашем случае 4) будет меньше бесконечности, то в матрицу A_1 на позицию, находящуюся на пересечении четвертой строки и третьего столбца записывается полученное значение – 4.

На втором этапе стационарные столбцы и строки отмечаются в матрице A_1 – это второй столбец и вторая строка. Также, как и было описано ранее, если в стационарной строке и столбце содержатся элементы, равные бесконечности, то все соответствующие им строки и столбцы копируются в следующую матрицу – A_2. В нашем примере оказалось, что в соответствии с этим условием абсолютно все элементы матрицы A_1 переносятся в матрицу A_2.

Следующий шаг предусматривает пометку в матрице A_2 стационарными третьего столбца и третьей строки – они без изменений будут перенесены в матрицу A_3, как и элементы, соответствующие знакам бесконечности в стационарных столбце и строке. Для определения того, какие значения будут записаны в матрицу A_3 в ячейки, находящиеся на оставшихся позициях – пересечении четвертого столбца и первой и четвертой строки, – восстанавливаются соответствующие перпендикуляры. Сумма элементов, на которые указывают перпендикуляры сравниваются с текущими значениями, и то значение, которое окажется наименьшим, запишется в соответствующую ячейку новой матрицы. В нашем примере расчеты показали, что на пересечении четвертого столбца и первой строки запишется новый полученный элемент, равный 3 (1+2=3, а 3<∞), а на пересечении четвертого столбца и четвертой строки в матрицу A_3 будет перенесен старый элемент – число 0 (сумма перпендикуляров 2+4 оказалась больше).

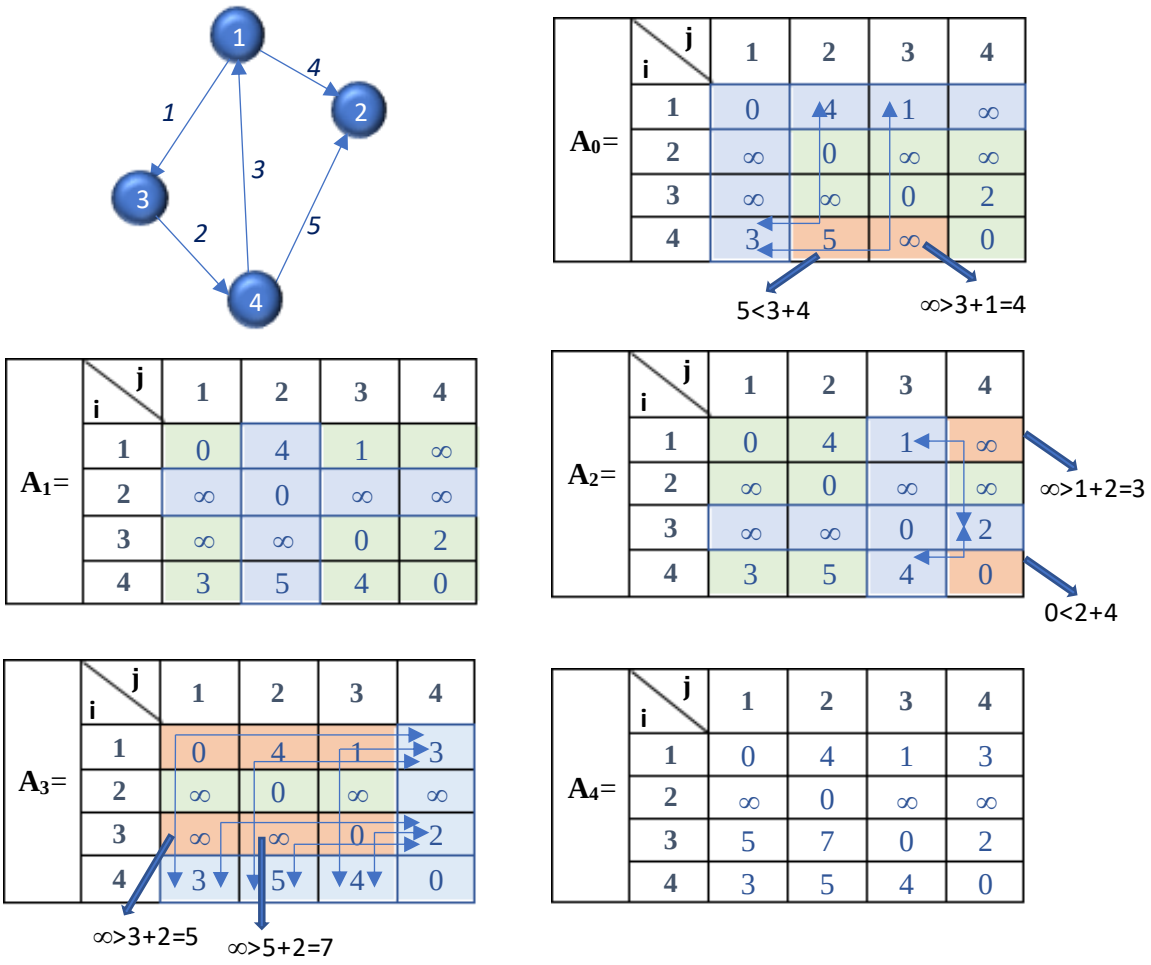


Рис.57. Схема итераций для реализации алгоритма Флойда-Уоршелла

Точно такие же действия производятся для матрицы A_3 с той лишь разницей, что стационарными столбцом и строкой будут помечены четвертый столбец и четвертая строка соответственно. После возведения в случае необходимости соответствующих перпендикуляров, вычисления сумм значений и сравнения с предыдущими значениями будет получена итоговая матрица A_4, содержащая итоговые значения.

Таким образом, с помощью вышеприведенного подхода найдены все кратчайшие пути от любой произвольной вершины до любой другой.

Так, например, в итоговой таблице указан кратчайший путь из вершины 3 в 2 равный 7 (путь через вершины 3-4-2), хотя существует и другой – равный 9 (вершины 3-4-1-2).

Алгоритм Флойда-Уоршелла применяется и в случае наличия отрицательный весов. Сложность алгоритма оценивается как $O(n^3)$.

Наидлиннейшая общая подпоследовательность (Longest Common Subsequence (LCS))

Задача нахождения наибольшей общей подпоследовательности относится к задачам динамического программирования. Она часто встречается в качестве вспомогательной задачи в различных более сложных задачах. Фактически задача сводится к решению двух отдельных подзадач – нахождению длины подпоследовательности и непосредственно самих элементов подпоследовательности.

Допустим, имеются две последовательности A и B: A=2, 3, 5, 1, 9, 7 и B=3, 8, 5, 4, 2, 6, 1. Необходимо найти их наибольшую общую подспоследовательность. Следует учитывать, что элементы в обеих последовательностях нельзя менять местами. Очевидно, что для данного конкретного случая интуитивно она будет представлена элементами 3 5 1. Однако, необходимо найти подход для определения подпоследовательности в общем случае. Можно обратиться к рекурсии. Фрагмент рекурсивного дерева (часть левой ветки) представлен на рис.58.

Поиск осуществляется по последним элементам, которые будут проверяться на одинаковость. Изначально последние элементы у последовательностей, представленных на рис.58, отличаются.

На первом этапе две последовательности разбиваются на две ветви – в первую входит первая последовательность без последнего элемента и вся вторая последовательность, а во вторую – целиком вся первая последовательность и вторая без последнего элемента.

Далее каждая из веток снова разбивается по этому же принципу (у каждой из ветвей появляются по две ветви) и сравниваются последние элементы.

На самой левой ветви в какой-то момент последние элементы совпадут – 2, 3, 5, 1 и 3, 8, 5, 4, 2, 6, 1. В этом случае у обеих последовательностей они отбрасываются (рядом записывается +1, которая указывает на наличие обнаруженного схожего элемента), а процесс продол-

жается по вышеописанному принципу. В результате будет обнаружена наибольшая общая подпоследовательность.

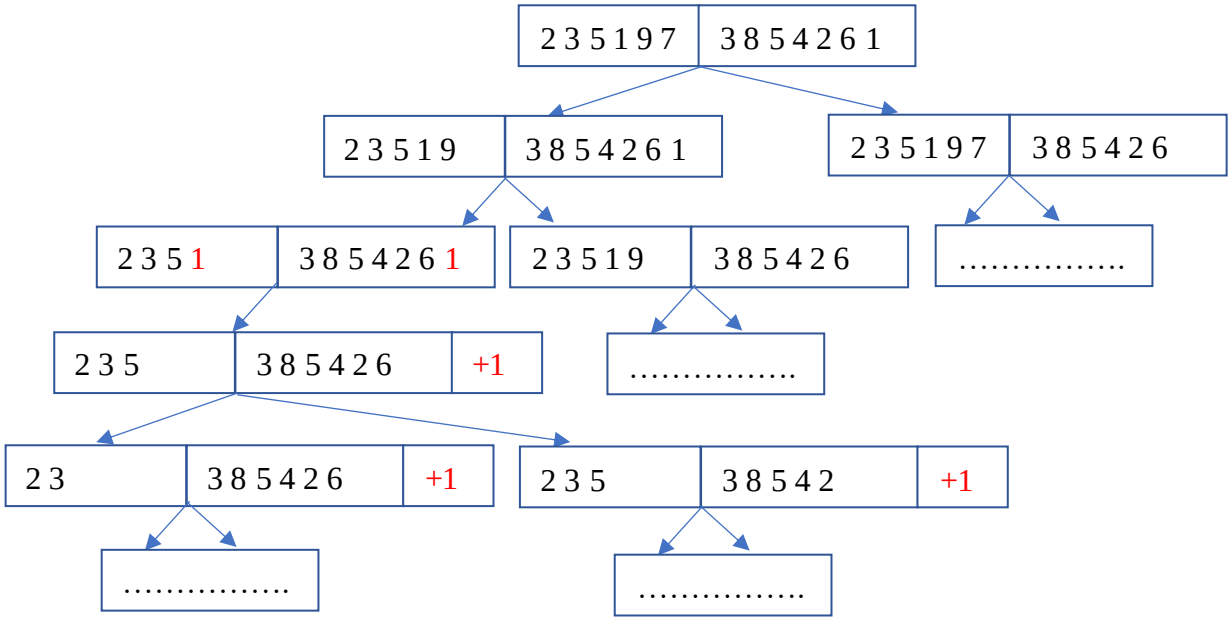


Рис.58. Фрагмент рекурсивного разложения последовательности

Как видно из рис.58, происходит постепенное упрощение задачи, которая в конечном итоге сведется к тривиальному варианту. При этом некоторые из задач будут решаться по нескольку раз, поскольку будут являться перекрывающими задачами. Их можно подсчитать один раз реализацией методов динамического программирования.

В первую очередь для определения длины подпоследовательности необходимо ввести целевую функцию F(i, j), которая будет считать наибольшую общую подпоследовательность для начала двух последовательностей A (число элементов n=6) и B (число элементов m=7):

$$A=2\,3\,5\,1\,9\,\overset{i}{7} \text{ и } B=3\,8\,5\,4\,2\,\overset{j}{6}\,1$$

Целевую функцию можно записать в явном виде: длина НОП для A[: i] и B [: j].

Затем следует установить граничные значения и реккурентное соотношение. Очевидно, что граничными значениями будет принято F(i, 0)=0 (от первой последовательности берем значения, а от второй – нет) и F(i, j)=0 (от первой последовательности не берем значения, а от

второй – берем). Реккурентное соотношение с учетом того, что рассматриваются префиксы последовательностей и последние элементы в них, примет вид:

$$F(i, j)=\begin{cases}(i-1, j-1)+1, & \text{если } A[i-1]=B[j-1];\\ \max\big(F(i-1,j), F(i, j-1)\big), & \text{если } A[i-1] \text{ !=} B[j-1].\end{cases} \quad (10)$$

Работа полученных реккурентных соотношений и граничных условий наглядно продемонстрирована на рис.59.

Выполнение граничных условий приводит к тому, что нулевые столбец и строка таблицы на рис.59 заполнены нулями. Поскольку, если мы возьмем какой-либо элемент с одной последовательности и не возьмем ни один элемент из второй, то длина общей подпоследовательности, естественно, будет равна нулю.

На пересечении единиц также окажется нуль, поскольку, если мы возьмем первый элемент первой последовательности, равный двум и первый элемент второй последовательности, равный трем, то очевидно, что длина их наибольшей общей подпоследовательности будет равна нулю.

Затем рассматривается уже два элемента последовательности m (3 и 8) и все также один элемент последовательности n с индексом 1 (2).

Далее рассматриваются три элемента последовательности m (3 8 5) и один элемент последовательности n с индексом 1 (2) и т.д. до момента, когда в последовательности m рассматриваются пять элементов (3 8 5 4 2), а в последовательности n по прежнему один элемент – 2. Поскольку цифра 2 второй последовательности в этот момент совпадает в цифрой 2, находящейся в конце первой последовательности, то можно сделать заключение, что наибольшая длина их общей подпослеовательности может составлять один элемент, равный двойке и поэтому на пересечении i=1 и j=5 следует записать единицу. Соответственно, единицами будет заполнена далее вся строка.

При заполнении следующей строки следует учитывать, что рассматривается уже два элемента последовательности n (2 3), а для последовательности m рассматриваются последовательно все j от единицы до семи (сначала 3, затем 3 8, затем 3 8 5 и т.д.).

Аналогичным способом можно заполнить всю таблицу. Однако, гораздо легче это сделать с помощью выведенного ранее реккурентного соотношения (см. формулу 10).

			3	8	5	4	2	6	1
	m / n	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	1	1	1
3	2	0	1	1	1	1	1	1	1
5	3	0	1	1	2	2	2	2	2
1	4	0	1	1	2	2	2	2	3
9	5	0	1	1	2	2	2	2	3
7	6	0	1	1	2	2	2	2	**3**

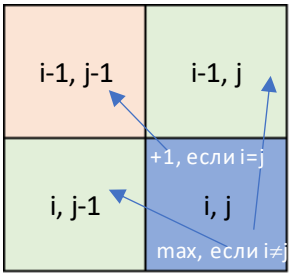


Рис.59. Схема заполнения ячеек при поиске длины наибольшей общей подпоследовательности

Справа на рис.59 продемонстрирована схема заполнения ячеек с помощью реккурентных соотношений в соответствии с формулой 10. То есть, если значения обеих последовательностей совпадают, то в ячейку записывается число, на единицу больше, чем в расположенной по диагонали ячейке. Если же значения отличаются, то в ячейку записывается большее из расположенных слева и сверху от нее.

Последний из полученных элементов, представленных в левой части рис.59 (в нижнем правом углу таблицы), представляет собой длину наибольшей общей подпоследовательности двух последовательностей A и B. В нашем конкретном случае наибольшая общая подпоследовательность состоит из трех элементов.

Для определения самих элементов следует восстановить пройденный путь с конца, фиксируя те из значений, для которых i и j будут одинаковыми. Передвижения от ячейки к ячейке совершаются в соответствии с реккурентным соотношением (10) – переходы влево и

вверх осуществляются в наибольшую ячейку при неравных i и j, а по диагонали – при равных. Значения элементов при равных i и j фиксируются в отдельном списке и представляют собой наибольшую общую подпоследовательность.

Графически это продемонстрировано на рис.60.

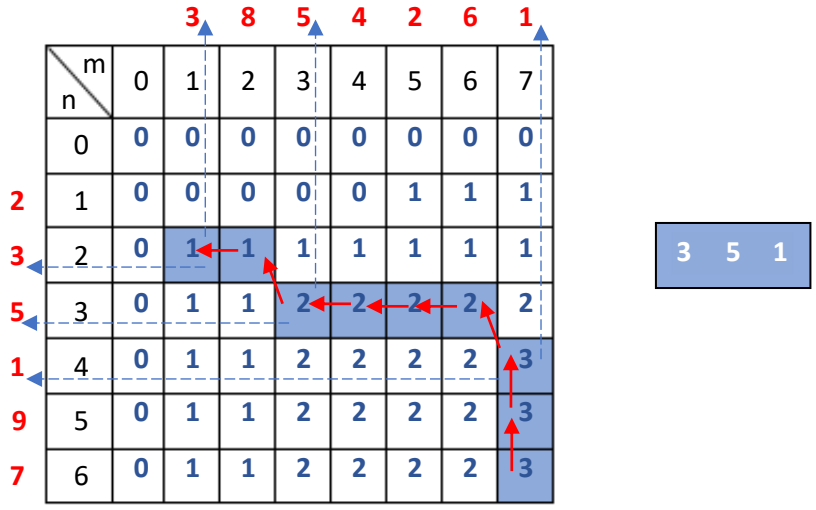


Рис.60. Поиск элементов наибольшей общей подпоследовательности

Как видно из рис.60, в нашем конкретном примере наибольшая общая подпоследовательность представлена элементами 3, 5, 1.

Поиск подстрок. Преобразование одной строки в другую

Одной из классических задач обработки информации является задача поиска фрагмента текста в строке (поиск образа в строке). Поиск можно осуществлять пошаговым сравнением символов образа в строке, однако такой алгоритм не будет эффективным. Для решения подобных задач существует несколько алгоритмов. Рассмотрим работу одного из них – алгоритма Бойера-Мура-Хорспула.

В соответствии с алгоритмом Бойера-Мура-Хорспула сравнение символов будет осуществлять с конца образа, что позволит производить сдвиги на большее количество символов, что и приводит к увеличению эффективности самого алгоритма.

Пусть имеется строка длиной n, в которой необходимо найти образ длиной m. Разумеется, длина m должна быть меньше длины n. В качестве примера рассмотрим поиск слова-образа «структуры» в строке «алгоритмы и структуры данных».

Перед непосредственным поиском образа в строке следует создать массив, в котором будут записаны значения, используемые при сдвиге образа по строке (рис.61).

Рис.61. Формирования массива со значениями для сдвига

Значениями элементов массива является удаление каждого символа от конца образа. Так, первая с конца буква «р» удалена на один символ от конца образа, первая с конца буква «у» удалена на два символа, следующая за ней буква «т» – на три и т.д. Следует обратить внимание на то, что в случае, если встречаются повторяющиеся символы (например, буквы «у», «т» и «р» повторяются дважды в нашем образе), то им надо приписывать значение аналогичного символа, который ближе к концу. При этом неоднократное вхождение символов в образ никак не влияет на удаленность других символов от конца образа, поэтому нумерация остальных символов продолжается по порядку.

В последнюю очередь значение приписывается последнему символу образа (в нашем конкретном примере это буква «ы», которой приписывается значение 9) (см. рис.61). Если последний символ уже встречался в образе, то ему приписывается то же самое значение. Например, если в качестве образа было бы слово «структуру», то последнему символу «у» было бы присвоено значение 2.

Для символов, не входящих в образ, которые на рис.61 обозначены символом «*», будет применяться сдвиг, равный длине образа (в нашем случае – это 9). Если же последнему симво-

лу образа приписано уже встречавшееся значение, то символу «*» будет присвоено следующее после максимального значение. Например, в вышеприведенном примере для образа «структуру», последнему «у» будет присвоена 2, а «*» – 9.

На рис.62 продемонстрирован поиск образа «структуры» путем его сдвига на то количество символов, которое указано в матрице (см. рис.61) для несовпадающих символов.

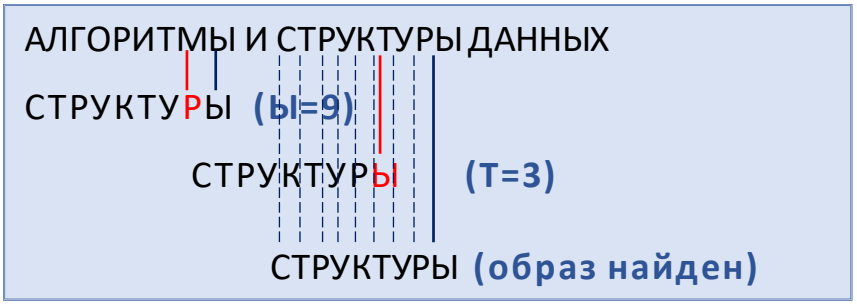


Рис.62. Поиск образа в строке

Как видно из рис.62, сначала происходит сравнение образа с первым словом строки начиная с последнего символа «ы». Он совпадает и происходить дальнейшее сравнение последующего символа. «Р» и «м» не совпадают. Если бы слово заканчивалось на «м», то смещение образа осуществилось бы на девять позиций (*=9). Однако, поскольку ранее был совпадающий символ («ы»), то смещение надо производить в соответствии с ним. В нашем примере смещение для «ы», являющимся последним символом образа также составляет 9. Поэтому образ смещается на 9 позиций с учетом пробелов. Следует заметить, что смещение определяется значением массива, соответствующим символу строки, а не образа. Далее снова осуществляется сравнение символов образа со строкой и первое же сравнение указывает на несоответствие (символы «ы» и «т»). Для «т» смещение составляет 3 позиции. На третьей строке сравниваются последовательно все символы образа и поэтапное сравнение указывает на то, что образ в строке найден (см. рис.62).

Примерно на таких принципах основана и операций преобразования одной строки в другую. На первом этапе проверяется равенство количества символов и строках и о существ-

ляется проверка на совпадение символов, т.е. строки должны быть одной длины и с одинаковым набором символов, расположенных в различном порядке.

Преобразование одной строки в другую начинается с проверки на совпадение символов. При этом проверка осуществляется с конца строк и в случае совпадения последних символов задача сводится к проверки длиной n-1.

Если же символы различаются, то следует найти позицию символа второй строки в первой и осуществить сдвиг символа на определенное количество позиций. Фактически разница между позициями совпадающих символов покажет сколько символов следует поместить перед текущим символом в преобразуемой строке.

Операция повторяется до тех пор, пока вся строка не будет преобразована во вторую.

Аналогичным образом реализуются и задачи замены символов.

Классы P и NP и NP-полнота. Сборник NP-полных задач. Общие стратегии. Неразрешимые задачи

Ранее нами рассматривались достаточно простые задачи, такие как поиск кратчайшего пути, нахождение наибольшей общей последовательности, сортировки множеств и др., которые разрешимы на обычном компьютере либо абстрактном вычислителе за полиномиальное время. Эти алгоритмы детерминированы – каждый следующий шаг алгоритма определяется значениями переменных, входными данными. Программа не делает каких-либо предположений, а четко следует алгоритму с заранее известными шагами.

Сложность таких алгоритмов зависит от входных данных и может составлять n (например, алгоритм сложения двух чисел) или может быть представлена многочленом, содержащим n, n^2 и т.д., то есть представлена каким-либо полиномом. Поэтому подобные задачи относятся к полиномиальным, т.е. решаемым за полиномиальное время. Все они объединены в класс P (Polynomial Time).

Задачи, которые нельзя решить за полиномиальное время относятся к классу NP (Non-deterministic Polynomial) недетерминированных полиномиальных задач.

Примером может служить проверка равенства нулю суммы элементов подмножества какого-либо множества. Можно заметить, что для, например, множества {3 7 25 6 -3 1 -22}, нулю будет равна сумма элементов подмножества {25 -3 -22}. Однако в случае, если множество будет состоять из большого количества элементов n, то перебор всех возможных вариантов, т.е. количество всех возможных подмножеств составит 2^n. В этом случае алгоритм является рабочим, но он работает за не полиномиальное время.

Алгоритм для таких задач делает предположение, а затем проверяет его на истинность. Основной идеей является то, что сколько времени бы не занимало поиск решения подобных

задач, проверить правильность полученного ответа можно за полиномиальное время. Например, если требуется разложить большое число на множители, то сам процесс разложения может занять достаточно много времени, а проверить правильность полученного результата возможно за полиномиальное время.

Классическим примером подобных задач служит задача о коммивояжере, которому необходимо по оптимальному пути посетить все города и вернуться обратно. Перебор всех вариантов займет не полиномиальное время, хотя проверка конкретного выбранного маршрута будет выполнена за полиномиальное время. К слову, задача о коммивояжере на первый взгляд очень похожа на задачу по поиску кратчайшего пути, которая легко решается за полиномиальное время с помощью одного из известных алгоритмов, например, Дейкстры, Беллмана-Форда или же Флойда-Уоршелла, рассмотренных ранее.

К такого рода задачам относится и вариант задачи о рюкзаке, имеющем определенный объем, когда предметы являются неделимыми и необходимо максимально заполнить его ценными вещами. И снова, задача о рюкзаке в варианте, когда предметы можно «распилить», решается достаточно легко за полиномиальное время.

Еще одним примером некой схожести P и NP может служить задача решения системы линейных уравнений или неравенств. Решение легко осуществимо известными алгоритмами линейного программирования, например, методом Крамера, или алгоритмом Гаусса. Но, если наложить некоторое ограничение, например, когда в системе линейных неравенств необходимо найти целое решение, то подобная задача целочисленного линейного программирования уже будет относиться к NP задачам.

Однако, не все задачи возможно легко проверить. Например, может быть дан ответ на то, каким будет победный шаг в шахматной партии, но доказательство этого потребует огромных вычислительных мощностей и, соответственно, времени.

В процессе выполнения алгоритмов задач класса NP предполагают наличие нескольких недетерминированных вычислительных стратегий. Если одна из стратегий будет успешной, то считается, что алгоритм выполнен.

Одним из способов преобразования недетерминированного алгоритма в детерминированный является использование, например, кластеров – большого количества процессоров. Этот способ не является панацеей, поскольку существуют задачи, решение которых не под силу современным суперкомпьютерам.

Второй способ предусматривает наличие какой-либо подсказки – сертификата для выбора вычислительной стратегии. Наличие сертификата помогает выполнить задачу за полиномиальное время, то есть легко проверить полученное решение. Однако, проблема заключается как раз в том, что в большинстве случаев сертификата просто не существует, а следовательно, для решения недетерминированной задачи потребует перебор всех возможных вариантов-сертификатов. Так, например, в случае с подмножеством сертификатом будет являться характеристический вектор – само подмножество с нулевой суммой элементов. Решение можно будет проверить за O(n). Без сертификата придется делать перебор всех возможных вариантов подмножеств за экспоненциальное время.

С NP задачами тесно связан термин NP-полные задачи. Они представляют собой задачи в теории алгоритмов с ответом «да» или «нет», к которым можно свести NP-задачу за полиномиальное время. Предполагается, что, если для такой задачи найдем «быстрый» алгоритм решения, то и для другой задачи из класса NP он будет эффективен. Задача о коммивояжере, например, является NP-полной и, если для нее будет найдено решение, то все задачи подобного типа также будут решены. Однако неизвестно, можно ли решить хоть одну NP-полную задачу не перебором, а за полиномиальное время.

Также к числу NP-полных задач относятся, например, такие как:

- *задача выполнимости булевых формул* (*SAT*, *ВЫП*), когда переменным в формулах, содержащих скобки, операции И, ИЛИ, НЕ необходимо назначить значения «истина» и «ложь» таким образом, чтобы в конечном итоге формула была бы истинной;

- *раскраска графа*, когда элементам графа ставятся метки с учетом определенных ограничений, например, когда две смежные вершины отмечаются разными цветами и т.д.;

- *задача Штейнера о минимальном дереве*, состоящая в том, чтобы заданный набор точек на плоскости был соединен кратчайшей сетью;

- *задача о вершинном покрытии* неориентированного графа, которая сводится к поиску такого множества его вершин S, у которого хотя бы один из концов каждого ребра входил бы в вершину из S. При решении задачи выбирается случайным образом ребро, обе вершины, которого помещаются в S, затем удаляются все ребра, инцидентные этим вершинам. Из оставшихся ребер выбирают следующее, его вершины также добавляют в множество S и удаляют инцидентные этим вершинам ребра. Процесс продолжается до тех пор, пока в графе не останется ребер. В результате будет определено вершинное покрытие S;

- *задача о клике*[1] в неориентированном графе, решение которой простым перебором является крайне неэффективным;

- *задача о независимом множестве*, которая эквивалентна задаче о клике и заключающаяся в поиске множества вершин, никакие две вершины в котором не соединенных друг с другом ребрами. В результате определения такого множества будет получен граф с изолированными вершинами;

- всем известные игры *тетрис*, *судоку*, *пятнашки*, *сапер*.

Разумеется вышеперечисленный список отнюдь не является полным.

[1] Кликой в неориентированном графе называется подмножество вершин, каждые две из которых соединены ребром графа.

И вот главный вопрос тысячелетия – равны ли NP и P (Does P=NP?). Известно лишь то, что P является подмножеством NP (рис.62), поскольку если мы можем решить задачу за полиномиальное время, то, тем более, сможем проверить ее за полиномиальное время. И если когда-то удастся доказать, что NP=P, то есть будет найден хороший полиномиальный алгоритм для решения какой-либо NP-задачи, а, следовательно, и всех NP-задач, то это может привести к коллапсу. Миллиарды пользователей потеряют конфиденциальность своих данных в аккаунтах, произойдут сбои в современных банковских и других системах, использующих криптографические алгоритмы. Дело в том, в настоящее время алгоритмы, шифрующие данные, представляют собой NP-алгоритмы и основаны на том, что проверять их довольно просто и быстро, а вот расшифровать – долго.

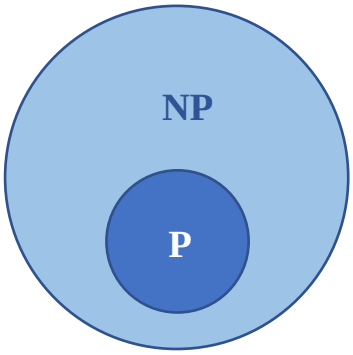


Рис.63. NP и P-классы

На сегодняшний день большинство ученых-математиков склоняется к тому, что NP≠P, меньшая же часть уверена, что NP=P и рано или поздно это будет доказано. Есть и совсем небольшая часть ученых, считающих что доказать равенство или неравенство NP и P невозможно в принципе.

За решения этой задачи тысячелетия математический институт Клэя[1] предлагает премию в размере 1 000 000 $, а также учреждены премии Филдса[2] и Абеля[3].

[1] Институт наиболее известен после объявления 24 мая 2000 года списка Проблем тысячелетия (Millennium Prize Problems). Эти семь проблем определены как «важные классические задачи, решение которых не найдено вот уже в течение многих лет». За решение каждой из задач предложен приз в 1 000 000 долларов США.

[2] Филдсовская премия (англ. Fields Medal) – самая престижная международная премия и медаль в области математики, которые вручаются один раз в 4 года на международном математическом конгрессе 2-4 молодым ученым не старше 40 лет (или достигших 40-летия в год вручения премии).

[3] Абелевская премия (англ. Abel Prize) – премия по математике, названная так в честь норвежского математика Нильса Хенрика Абеля. Основана правительством Норвегии в 2002 году, и, начиная с 2003 года, ежегодно присуждается выдающимся математикам современности. Денежный размер премии сопоставим с размером Нобелевской премии и составляет 6 млн норвежских крон (€ 620 900 или $ 686 300).

Матрицы и их свойства

Матрица представляет собой квадратную либо прямоугольную таблицу с совокупностью строк и столбцов, на пересечении которых располагаются ее элементы, обозначаемые двумя индексами. Первый индекс служит для обозначения строки, а второй – столбца:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{pmatrix}. \quad (11)$$

Множество всех матриц размера m на n записывается следующим образом:

$$A \in M_{m,\,n}(R), \quad (12)$$

где M – множество матриц с числом строк m и столбцов n соответственно, R – поле действительных чисел. Для квадратной матрицы (число строк и столбцов равны) множество записывают как $M_n(R)$.

Элементы квадратной матрицы, расположенные на диагонали, начинающейся в левом верхнем углу матрицы и ведущей к правому нижнему углу образуют главную диагональ, элементы же, расположенные на диагонали, идущей из левого нижнего угла к правому верхнему образуют побочную диагональ:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}. \quad (13)$$

Побочная диагональ

Главная диагональ

Матрицу также записывают в коротком виде:

$$A=(a_{ij}), \quad (14)$$

где i и j ее строки и столбцы соответственно.

В матрице может содержать только один столбец ($A \in M_{m,1}(R)$) или только одна строка ($A \in M_{1,n}(R)$).

Матрицу, содержащую все нули называют нулевой, а квадратную матрицу, содержащую все единицы на главной диагонали (все остальные элементы нули) – единичной.

Для матриц определен ряд операций, таких как сложение, умножение на число и умножение самих матриц, транспонирование, обращение и т.д.

Матрицы можно складывать, но при этом следует учитывать, что эта операция возможна только при совпадении размерности матриц. Сложение для матриц $A,B \in M_{m,n}(R)$ осуществляется по следующему правилу:

$$C=A+B,\ c_{ij}=a_{ij}+b_{ij},\ \text{при } i=1\ldots m,\ j=1\ldots j. \qquad (15)$$

В соответствии с формулой (15) ниже приведено сложение двух матриц A и B.

$$A=\begin{pmatrix}1 & 5 & 7\\ 2 & 5 & 4\\ 7 & 2 & 2\end{pmatrix},\qquad B=\begin{pmatrix}0 & -1 & 5\\ 8 & 9 & 0\\ -3 & 4 & 2\end{pmatrix},\qquad C=A+B=\begin{pmatrix}1 & 4 & 12\\ 10 & 14 & 4\\ 4 & 6 & 4\end{pmatrix},$$

Умножение матрицы на любое произвольное число производится последовательным умножением всех ее элементов на это число. В результате будет получена новая матрица. Например:

$$A=\begin{pmatrix}1 & 5 & 7\\ 2 & 5 & 4\\ 7 & 2 & 2\end{pmatrix},\ \lambda=5,\ B=A*\lambda=\begin{pmatrix}1*5 & 5*5 & 7*5\\ 2*5 & 5*5 & 4*5\\ 7*5 & 2*5 & 2*5\end{pmatrix}=\begin{pmatrix}5 & 25 & 35\\ 10 & 25 & 20\\ 35 & 10 & 10\end{pmatrix}.$$

Часто матрицы приходится транспонировать. Транспонированная матрица – это матрица, которая получается из исходной в результате взаимозамены строк и столбцов:

$$A=\begin{pmatrix}a_{11} & a_{12} & a_{13} & \ldots & a_{1n}\\ a_{21} & a_{22} & a_{23} & \ldots & a_{2n}\\ \ldots & \ldots & \ldots & \ldots & \ldots\\ a_{m1} & a_{m2} & a_{m3} & \ldots & a_{mn}\end{pmatrix}\ \rightarrow\ A^{T}=\begin{pmatrix}a_{11} & a_{21} & a_{31} & \ldots & a_{m1}\\ a_{12} & a_{22} & a_{32} & \ldots & a_{m2}\\ \ldots & \ldots & \ldots & \ldots & \ldots\\ a_{1n} & a_{2n} & a_{3n} & \ldots & a_{mn}\end{pmatrix}. \qquad (16)$$

Так, например, в соответствии с (16) получим:

$$A=\begin{pmatrix}1 & 2 & 4\\ 3 & 5 & 7\end{pmatrix}\ \rightarrow\ A^{T}=\begin{pmatrix}1 & 3\\ 2 & 5\\ 4 & 7\end{pmatrix}.$$

При повторном транспонировании матрица вернется к исходному виду, т.е. $(A^T)^T=A$.

Результат суммирования и последующего транспонирования суммы матриц будет равен сумме транспонированных матриц, т.е. $(A+B)^T= A^T+ B^T$.

Аналогично формулируется и свойство, связанное с умножением скалярного числа на транспонированную матрицу – $(\lambda*A)^T=\lambda(A^T)$.

Одной из основных операций над матрицами является их умножение. Эта операция осуществляется по определенному правилу, в соответствии с которым число элементов строк первой матрицы должно быть равно числу элементов столбца второй матрицы. То есть операция выполнима только для совместимых матриц – $A_{m,n}*B_{n,r}=C_{m,r}$.

Каждый элемент новой матрицы C вычисляется в соответствии с формулой

$$c_{ij}=a_{i1}*b_{1j}+a_{i2}*b_{2j}+a_{i3}*b_{3j}+...+a_{in}*b_{nj}=\sum_{k=1}^{n} a_{ik}b_{kj}. \qquad (17)$$

Так, например,

$$A=\begin{pmatrix}1 & 2 & 4\\ 3 & 5 & 7\end{pmatrix},\quad B=\begin{pmatrix}1 & 6\\ 7 & 9\\ 8 & 4\end{pmatrix},\quad C=\begin{pmatrix}1*1+2*7+4*8 & 1*6+2*9+4*4\\ 3*1+5*7+7*8 & 3*6+5*9+7*4\end{pmatrix}=\begin{pmatrix}47 & 40\\ 94 & 91\end{pmatrix}.$$

Вычислительная сложность при вычислении произведений матриц достаточно велика и составляет $O(n^3)$. Несколько улучшить этот показатель, особенно для плотных матриц, позволяет алгоритм Штрассена, представляющий собой обобщенный вариант метода Карацубы.

Алгоритм Штрассена

Предложенный Штрассеном алгоритм, не будучи самым быстрым, наиболее часто используется на практике, поскольку является легко программируемым и достаточно эффективным при умножении матриц малых размеров.

Алгоритм основан на методе «разделяй и властвуй». В его основу положен метод декомпозиции, состоящий из трех этапов:

- разбиение;
- рекуррентное решение подзадач;

- сборка результатов.

Метод декомпозиции предусматривает представление матриц A и B четырьмя подматрицами, каждая из которых имеет размер n/2 x n/2:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad \text{и} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

Соответственно, при умножении матрицы A на B будет получена матрица C

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{18}$$

элементы которой будут определяться описанным выше способом:

$$C_{11}=A_{11}*B_{11}+ A_{12}*B_{21}; \qquad C_{12}=A_{11}*B_{12}+ A_{12}*B_{22};$$

$$C_{21}=A_{21}*B_{11}+ A_{22}*B_{21}; \qquad C_{22}=A_{21}*B_{12}+ A_{22}*B_{22}. \tag{19}$$

Однако, такая простая декомпозиция хоть и упрощает вычисление матриц, но ничуть не уменьшает вычислительную сложность, которая по-прежнему будет составлять $O(n^3)$.

Штрассен в своем алгоритме предложил уменьшить количество операций умножения путем увеличения более простых операций, таких как сложение и вычитание матриц. С этой целью было введено десять вспомогательных матриц:

$$S_1=B_{12} - B_{22;} \quad S_2=A_{11} + A_{12}; \quad S_3=A_{21} + A_{22}; \quad S_4=B_{21} - B_{11}; \quad S_5=A_{11} + A_{22};$$

$$S_6=B_{11} + B_{22}; \quad S_7=A_{12} - A_{22}; \quad S_8=B_{21} + B_{22}; \quad S_9=A_{11} - A_{21}; \quad S_{10}=B_{11} + B_{12}. \tag{20}$$

Затем вводится еще семь вспомогательных матриц:

$$P_1=A_{11}*S_1= A_{11}*B_{12} - A_{11}*B_{22}; \; P_2=S_2* B_{22}= A_{11}*B_{22}+ A_{12}*B_{22}; \; P_3= S_3*B_{11}= A_{21}*B_{11}+ A_{22}*B_{11};$$

$$P_4= A_{22}*S_4= A_{22}*B_{21} - A_{22}*B_{11}; \; P_5=S_5*S_6= A_{11}*B_{11} + A_{11}*B_{22}+A_{22}* B_{11} + A_{22}*B_{22};$$

$$P_6=S_7*S_8= A_{12}*B_{21} + A_{12}*B_{22} - A_{22}* B_{21} - A_{22}*B_{22};$$

$$P_7=S_9*S_{10}= A_{11}*B_{11} + A_{11}*B_{12} - A_{21}* B_{11} - A_{21}*B_{12}. \tag{21}$$

Как видно из (21) вычисление семи элементов потребует семь операций умножения на каждом этапе рекурсии.

С использованием вспомогательных элементов (20) и (21) могут быть вычислены вспомогательные подматрицы (19) результирующей матрицы (18):

$$C_{11}=P_5+P_4-P_2+P_6; \qquad C_{12}=P_1+P_2;$$

$$C_{21}=P_3+P_4; \qquad C_{22}=P_5+P_1-P_3-P_7. \qquad (22)$$

Как видно из формул (22) для вычисления подматриц используются только операции сложения и вычитания.

Рекурсивный процесс для больших матриц будет продолжаться до тех пор, пока размер матрицы C_{ij} не станет настолько малым, что будет возможно применение правил обычного перемножения матриц.

За счет уменьшения количества умножений (на одну операцию) и выполнения 18 сложений удается немного уменьшить сложность алгоритма, которая в этом случае составляет $O(n^{2,81})$.

В дальнейшем были попытки улучшения предложенного Штрассеном алгоритма. Одна из попыток привела к уменьшению вычислительной сложности до $O(n^{2,3727})$ (алгоритм Копперсмита-Винограда), однако, этот алгоритм не нашел широкого применения ввиду астрономических значений констант в оценке арифметической сложности.

Обращение матриц

Обратной называется матрица, при умножении которой на исходную получается единичная матрица.

Перед обращением матрицы следует убедиться в том, что матрица не является вырожденной, то есть ее определитель не должен быть равен нулю.

Один из способов нахождения обратной матрицы A-1 основан на формуле

$$A^{-1} = \frac{A^*}{|A|}, \qquad (23)$$

где A^* – союзная матрица[1], а $|A|$ – определитель матрицы A.

Пусть дана матрица размером 3x3

[1] Союзная матрица – это матрица, составленная из алгебраических дополнений транспонированной исходной матрицы.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}. \quad (24)$$

Тогда A^T запишется следующим образом

$$A^T = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{pmatrix}. \quad (25)$$

Запишем союзную матрицу к исходной (24), составленную из алгебраических дополнений транспонированной матрицы (25)

$$A^* = \begin{pmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \\ A_{13} & A_{23} & A_{33} \end{pmatrix}. \quad (26)$$

Формула связи алгебраического дополнения через минор[1] имеет вид

$$A_{ij}=(-1)^{i+j}M_{ij}, \quad (27)$$

где M_{ij} – минор элемента ij.

Рассмотрим реализацию первого способа обращения матриц на конкретном примере для следующей матрицы

$$A = \begin{pmatrix} -2 & 2 & 3 \\ 2 & 8 & -3 \\ 1 & -4 & -2 \end{pmatrix}.$$

Определитель матрицы этой матрицы равен 10, следовательно матрица не является вырожденной и для нее может быть найдена обратная матрица.

В первую очередь следует вычислить алгебраические дополнения по исходной матрице для каждого элемента вычеркиванием соответствующих строк и столбцов, на которых расположен элемент и с учетом знаков минора в соответствии с формулой (27).

$$A_{11}=(-1)^{1+1}\begin{vmatrix} 8 & -3 \\ -4 & -2 \end{vmatrix}=(-16-12)=-28; \quad A_{12}=(-1)^{1+2}\begin{vmatrix} 2 & -3 \\ 1 & -2 \end{vmatrix}=-(-4-(-3)=1;$$

$$A_{13}=(-1)^{1+3}\begin{vmatrix} 2 & 8 \\ 1 & -4 \end{vmatrix}=(-8-8)=-16; \quad A_{21}=(-1)^{2+1}\begin{vmatrix} 2 & 3 \\ -4 & -2 \end{vmatrix}=-(-4-(-12))=-8;$$

[1] Минор в линейной алгебре – определитель некоторой меньшей квадратной матрицы, вырезанной из заданной матрицы A путем удаления одной или нескольких ее строк и столбцов.

$$A_{22}=(-1)^{2+2}\begin{vmatrix}-2 & 3\\ 1 & -2\end{vmatrix}=(4-3)=1; \qquad A_{23}=(-1)^{2+3}\begin{vmatrix}-2 & 2\\ 1 & -4\end{vmatrix}=-(8-2)=-6;$$

$$A_{31}=(-1)^{3+1}\begin{vmatrix}2 & 3\\ 8 & -3\end{vmatrix}=(-6-24)=-30; \qquad A_{32}=(-1)^{3+2}\begin{vmatrix}-2 & 3\\ 2 & -3\end{vmatrix}=-(6-6)=0;$$

$$A_{33}=(-1)^{3+3}\begin{vmatrix}-2 & 2\\ 2 & 8\end{vmatrix}=(-16-4)=-20.$$

В соответствии с формулой (26) запишем союзную матрицу:

$$A^{*}=\begin{pmatrix}-28 & -8 & -30\\ 1 & 1 & 0\\ -16 & -6 & -20\end{pmatrix}.$$

Тогда обратная матрица, вычисляемая по формуле (23), будет иметь вид:

$$A^{-1}=\frac{1}{|A|}*A^{*}=\frac{1}{10}*\begin{pmatrix}-28 & -8 & -30\\ 1 & 1 & 0\\ -16 & -6 & -20\end{pmatrix}=\begin{pmatrix}-2.8 & -0.8 & -3\\ 1 & 0.1 & 0\\ -1.6 & -0.6 & -2\end{pmatrix}.$$

Если обратная матрица найдена правильно, то в результате перемножения ее на исходную матрицу должна получиться единичная матрица.

Обратную матрицу можно находить и другим способом, предусматривающем проведение элементарных преобразований исходной матрицы, таких как перестановка строк или столбцов, умножение любой строки на любое число, не равное нулю, прибавление или вычитание к элементам строки элементов другой строки.

Сначала следует записать расширенную матрицу, состоящую из исходной и единичной. В результате элементарных преобразований необходимо добиться, чтобы на месте исходной матрицы была единичная, тогда на месте единичной будет получена обратная матрица, т.е.

$$(A|E)\sim(E|A^{-1}). \qquad (28)$$

Для вышеописанного примера в результате поэтапных элементарных преобразований (мы не будем рассматривать их подробно в рамках этого раздела), осуществляемых для матрицы А, должны дублироваться и для матрицы Е в левой части.

При выполнении преобразований следует придерживаться определенного алгоритма. Сначала следует добиться того, чтобы преобразовать элемент, находящийся на пересечении первой строки и первого столбца к единице. Затем, преобразовать стоящие под ним элементы

к нулю. После этого необходимо добиться того, чтобы на месте элемента, находящегося на пересечении второй строки и второго столбца была бы единица, а под ней – ноль. Далее преобразования к единице производят для элемента, находящегося на пересечении третьей строки и третьего столбца, а затем, для остальных элементов – к нулю.

В результате будет получена запись, представленная в правой части:

$$A = \left(\begin{array}{ccc|ccc} -2 & 2 & 3 & 1 & 0 & 0 \\ 2 & 8 & -3 & 0 & 1 & 0 \\ 1 & -4 & -2 & 0 & 0 & 1 \end{array}\right) \sim \ldots \sim \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & -2.8 & -0.8 & -3 \\ 0 & 1 & 0 & 1 & 0.1 & 0 \\ 0 & 0 & 1 & -1.6 & -0.6 & -2 \end{array}\right).$$

Таким образом, обратная матрица будет найдена.

Основы криптографии. *Простые подстановочные шифры. Криптография с открытым ключом. Криптосистема RSA. Гибридные крипто-системы. Вычисление случайных чисел*

11

Простые подстановочные шифры. Криптография с открытым ключом

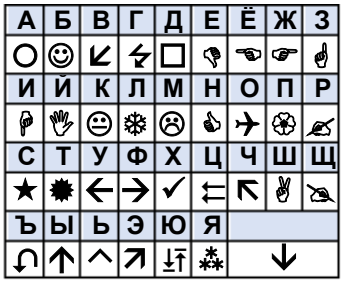
С процессами шифрования в XXI веке приходится сталкиваться ежеминутно, например, при пересылке текстовых сообщений, облачных файлов, хранении банковских реквизитов, личных данных в аккаунтах и т.д.

Шифрование, или же криптографическое кодирование, предполагает некое преобразование исходного текста в набор символов (цифр или/и/ букв), искажающее первоначальный смысл, защищая его от нежелательного просмотра. Сам алгоритм, по которому осуществляется преобразование исходного текста называется шифром. Зашифрованное сообщение также называют криптограммой. Компонент, на основе которого возможно шифрование и дешифрование, носит название ключа.

Люди издавна пытались зашифровать пересылаемые сообщения. Самый простой из способов – это метод простой подстановки. Согласно этому методу, каждая буква алфавита заменяется каким-либо символом, как например, это продемонстрировано на рис.64,а. Фраза «Алгоритмы и структуры данных» при использовании такого шифра будет выглядеть «🌕❄🌩✈✍☝✹☹🡅🡇☝🡇★✹★✹✍🡐😐✹🡐✍🡅🡇☐🌕👌👌🡅✓». Зная ключ, расшифровать такую запись довольно просто.

Еще одним примером простейшего шифра подстановки является азбука Морзе[1], получившая широкое применение в военной радиосвязи, где каждая буква алфавита заменяется набором точек и тире (рис. 64, б).

[1] Код Мо́рзе – способ знакового кодирования, представление букв алфавита, цифр, знаков препинания и других символов последовательностью сигналов: длинных (тире) и коротких (точек). Назван в честь американского изобретателя и художника Сэмюэля Морзе.



а)

А	·–	И	··	Р	·–·	Ш	––––
Б	–···	Й	·–––	С	···	Щ	––·–
В	·––	К	–·–	Т	–	Ъ	·––·–·
Г	––·	Л	·–··	У	··–	Ы	–·––
Д	–··	М	––	Ф	··–·	Ь	–··–
Е	·	Н	–·	Х	····	Э	··–··
Ж	···–	О	–––	Ц	–·–·	Ю	··––
З	––··	П	·––·	Ч	––– ·	Я	·–·–

б)

Рис.64. Шифрование простой подстановкой с использованием а) значков; б) азбуки Морзе

К шифрам простой замены относится и моноалфавитный шифр подстановки. Одним из простейших вариантов такого шифра является шифр Атбаша, получаемый инверсией исходного алфавита (рис.65).

А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
Я	Ю	Э	Ь	Ы	Ъ	Щ	Ш	Ч	Ц	Х	Ф	У	Т	С	Р	П	О	Н	М	Л	К	Й	И	З	Ж	Ё	Е	Д	Г	В	Б	А

Рис. 65. Шифр Атбаша

Один из старейших шифров подстановки, дошедший до нас – это шифр, названный в честь императора Гая Юлия Цезаря, который активно пользовался секретной записью своих документов с помощью сдвига букв алфавита на несколько (обычно 3) позиций (рис.66). В этом случае число букв, на которое сдвигается алфавит называют ключом шифрования.

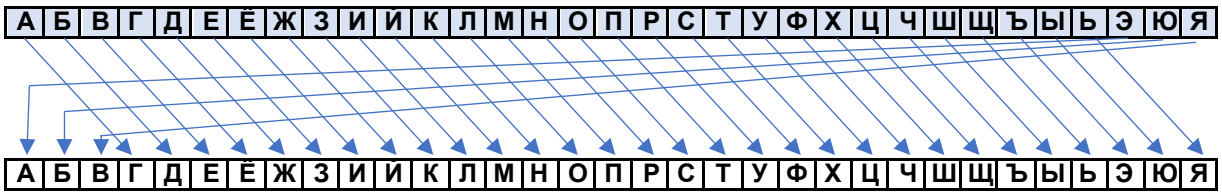


Рис.66. Шифр Цезаря

У шифра Цезаря существует множеством вариантов, зависящих от количества сдвигаемых символов. Одним из современных вариантов является шифр ROT13, осуществляющий сдвиг на 13 позиций и активно использующийся, например, в интернет-форумах.

Подобные моноалфавитные шифры достаточно легко взломать. Так, если в конкретном алфавите содержится k символов, то количество ключей будет составлять (k!-1), поскольку будет существовать k способов записать шифруемое сообщение с помощью исходного алфа-

вита. Конечно, это достаточно много, однако, очень помогает во взломе таких кодов и частотный анализ, определяющий частоту попадания каждой из букв в произвольном тексте. Следует заметить, что он наиболее эффективен при расшифровке текстов больших объемов. В качестве примера на рис.67 приведен частотный анализ для русскоязычных текстов.

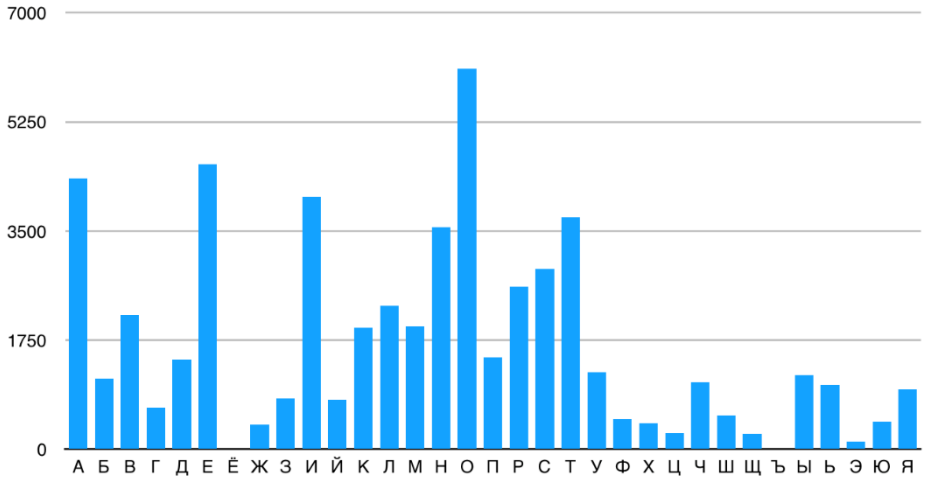


Рис.67. Частотный анализ русскоязычных текстов

Простые моноалфавитные шифры криптографией были признаны практически бесполезными, поэтому в последствии они дорабатывались и усложнялись.

Усложненный вариант шифра Цезаря представляет собой шифр дипломата Блез де Виженера. Шифрование осуществляется при помощи таблицы, в первой строке которой записываются буквы алфавита. В каждой следующей строке алфавит сдвигается на одну букву (рис.68). Ключом в таком методе шифрования является какое-либо слово. Шифруемый текст разбивается на блоки, длина которых соответствует длине ключевого слова. Затем на пересечении строки и столбца букв ключа и букв шифруемого слова последовательно определяются буквы шифра (рис.69).

При реализации шифра Виженера одинаковые буквы в сообщении будут представлены разными буквами в зашифрованном сообщении, что очень значительно усложняет взлом. Для взлома с помощью частотного анализа следует, как минимум, знать длину ключевого слова, либо придется пробовать брать разную длину.

Кроме того, смещение в таблице можно осуществлять не на одну букву, а на произвольное число, что еще более усложнит данный шифр.

	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
А	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
Б	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А
В	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б
Г	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В
Д	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г
Е	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д
Ё	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е
Ж	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё
З	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж
И	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З
Й	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И
К	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й
Л	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К
М	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л
Н	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М
О	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н
П	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О
Р	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Е	Ж	З	И	И	К	Л	М	Н	О	П
С	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р
Т	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С
У	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т
Ф	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У
Х	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф
Ц	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х
Ч	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц
Ш	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч
Щ	Щ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш
Ъ	Ъ	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ
Ы	Ы	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ
Ь	Ь	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы
Э	Э	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь
Ю	Ю	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э
Я	Я	А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю

Рис.68. Таблица Виженера

Ключ:	к	у	р	с	к	у	р	с	к	у	р	с	к	у	р	с	к	у	р	с	к	у	р	с	к
Фраза:	а	л	г	о	р	и	т	м	ы	и	с	т	р	у	к	т	у	р	ы	д	а	н	н	ы	х
Шифр:	к	я	у	а	ы	ь	г	ю	ё	ь	в	д	ы	ж	ы	д	ю	д	л	х	к	б	ю	м	а

Рис.69. Пример использования шифра Виженера

На шифре простой замены, но со значительными доработками, основана и знаменитая машина «Энигма», включающая в себя комбинации электрических и механических схем и активно используемая немцами во время Второй мировой войны. Выходной шифр зависел напрямую от изначальной конфигурации, которая изменялась в течение работы несколько раз.

Ежедневно конфигурация обновлялась, что делало практически невозможным расшифровку создаваемых сообщений. Однако суперкомпьютеру «Бомба», созданному Аланом Тьюрингом, все же удавалось иногда расшифровывать перехваченные сообщения. Наращивание числа дешифраторов «Бомба» до 200 экземпляров привело к возможности обработки до 3000 немецких шифровок в день.

Все рассмотренные выше случаи относятся к симметричным методам шифрования с открытым ключом. Это предполагает наличие ключа у обеих сторон – шифровальщика и дешифровщика. Дешифровка производится тем же самым ключом, каким производилось и шифрование. Существование открытого ключа у обоих сторон легко может привести к его перехвату третьей стороной и достаточно легкому взлому шифра, что делает такой метод крайне ненадежным.

Криптосистема RSA. Гибридные крипто-системы

В 1977 году был придуман алгоритм RSA, получивший свое названия по фамилиям его авторов (Rivest, Shamir, Adleman), представляющий собой криптографический алгоритм с открытым ключом. Этот алгоритм основан на построении односторонних функций, таких математических понятиях, как простые числа, малая теорема Ферма, функция Эйлера и относится к асимметричным методам шифрования.

Если провести аналогию с реальными замками и ключами, то асимметричное шифрование можно представить в виде открытого сейфа с защелкой. Положить в него сообщение можно без ключа, а затем захлопнуть дверцу. Открыть же сейф сможет только человек, имеющий ключ.

В алгоритме RSA используются понятия личного (private key) и открытого ключа (public key). Они формируются при использовании простых чисел.

При формировании открытого ключа используются простые числа, которые обладают свойством делиться только на себя и единицу, например, 2, 3, 5, 7, 11, 13….

Рассмотрим пример создания ключа. Выберем два простых числа – 2 и 7. При их перемножении получим 14. Полученное число 14 и еще одно число, называемое открытой экспонентой и будет открытым ключом – (e, 14). Введем некоторые обозначения. Первое простое число (2) обозначим p, а второе (7) – q, полученный результат умножения 14 – mod.

Затем необходимо вычислить значение функции φ с использованием функции Эйлера для простых чисел

$$\varphi=(p-1)*(q-1). \quad (29)$$

Для нашего примера φ=(p-1)*(q-1)=(2-1)*(7-1)=1*6=6.

Далее следует выбрать число e, которое обязательно должно быть простым числом, кроме того, должно быть меньше, чем функция Эйлера φ (в нашем примере должно быть меньше 6), а также не иметь с φ общих делителей. Простые числа, меньшие φ в нашем примере – это 2, 3 и 5. Однако, 2 и 3 являются делителями φ=6. Следовательно, необходимо остановить выбор на 5.

Таким образом, открытая экспонента e=5 и модуль mod=14 будут открытым ключом – {5, 14}.

Личный ключ будет состоять из модуля (число 14) и числа d, которое является обратным числу e по модулю φ и вычисляется по формуле

$$(d*e)\%\ \varphi=1. \quad (30)$$

Иными словами, остаток от деления произведения d*e на φ должен быть равен 1.

Подставив имеющиеся у нас значения в вышеприведенную формулу, можно определить (подобрать) число d:

$$(d*5)\%\ 6=1.$$

Путем подбора ясно, что подходит число 5: (5*5)%6=1. Однако, чтобы не путать в дальнейшем значения e=5 и d, подберем другой вариант, например: (11*5)%6=1. Итак, d=11.

Таким образом, личный ключ будет равен {d, mod}={11, 14}.

Можно перейти к кодированию конкретного числа x, которое должно быть меньше значения mod=14, например, пусть это будет число 10.

Возведем это число в степень e=5: 10^5=100000, а затем разделим его по модулю (с остатком) на mod=14: 100000 mod 14=7 (14 умещается в число 100000 целиком 7142 раза (14*7142==99988) и 12 остается в остатке).

Полученное число 12 передаем для расшифровки. Для этого следует воспользоваться значением d из личного ключа. В нашем случае оно равно 11. Переданное число 12 следует возвести в степень d=11: 12^{11}= 743008370688. При делении его по модулю на второй элемент личного ключа mod=14, получаем зашифрованное число 10 (число 743008370688 с остатком по модулю делим на число 14. В числе 743008370688 число 14 помещается 5307202647777 раз (5307202647777*14=74300837068878) и 10 остается в остатке, что и является результатом).

Итак, связующим звеном между личным и открытым ключами является значение mod (в нашем примере 14), получаемое изначально путем перемножения двух простых чисел (в нашем примере это были числа 2 и 7).

Схематически описанный выше процесс представлен на рис.70.

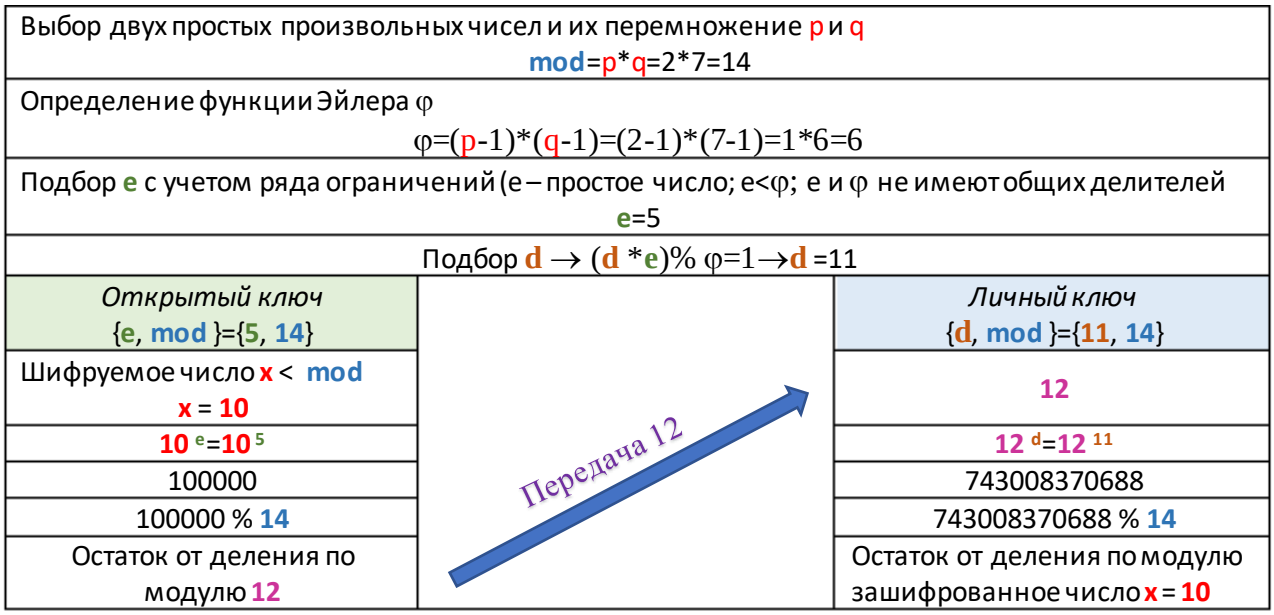


Рис.70. Схема реализации алгоритма RSA

Надежность подобного шифрования обеспечивается тем, что третьему лицу сложно определить личный ключ по открытому. Основной трудностью как раз и является разложение значения mod, представляющему произведение двух простых чисел на простые сомножители. Если значение mod будет представлять произведение двух очень больших многозначных простых чисел, то разложить их на простые сомножители будет непросто, а иногда и невозможно даже самому современному суперкомпьютеру.

Шифрование слов осуществляется по буквам – каждой из них присваивается числовое значение (например, ее порядковый номер в алфавите), а затем каждое число зашифровывается описанным выше способом.

Однако, алгоритмы RSA не являются совершенными. У злоумышленника и без знания ключа есть шанс разгадать шифр. Этому способствуют и частые коды пробелов, и слова, состоящие из одной или двух букв, по которым можно догадаться и о значении других слов и т.д.

Поэтому на практике применяют усложненные алгоритмы шифрования. Зачастую используют такие дополнительные алгоритмы, в которых каждая предыдущая часть будет оказывать влияние на последующую, например b=(b+a) % mod, где b – последующая часть, a – предыдущая, mod – число, на которое будем по модулю делить сумму. Расшифровка будет производиться по формуле b=(b-a) % mod.

Еще более усложнить эту систему шифрования можно дополнительным добавлением к началу последовательности ничего не значащей цифры. Тогда вычисление последующих букв с учетом предыдущих еще более усложнится.

В реальности блоки, на которые разбивается сообщение, длиннее одной буквы, поэтому сначала используют алгоритмы выравнивания, затем алгоритмы разбиения на блоки с перепутыванием (как, например, описано выше), а затем уже применяют алгоритмы RSA. Получатель делает все в обратном порядке – сначала расшифровывает информацию, затем распутывает, а затем отбрасывает ненужные блоки (рис.71).

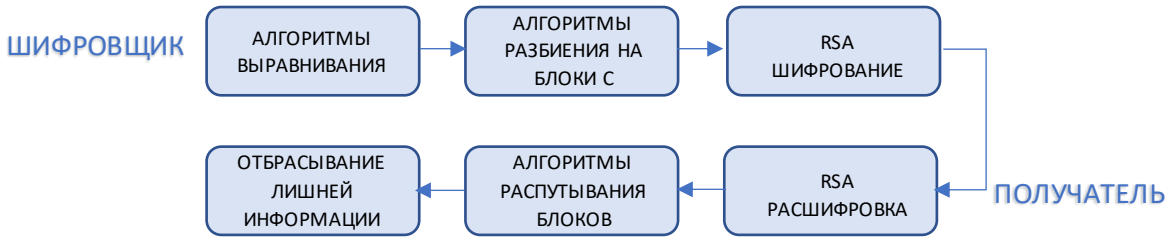


Рис.71. Схема усложнения алгоритма RSA

Также усложняются алгоритмы шифрования использованием гибридных криптосистем. Подобные системы помогают в решении возникающих проблем как при симметричном шифровании, так и при асимметричном.

В общих чертах, подход заключается в зашифровке большого объема данных секретным ключом с помощью алгоритмов симметричного шифрования и дальнейшей передачей самого ключа с помощью методов асимметричного шифрования, как например, такого, как рассмотренный выше алгоритм RSA.

Применение симметричного шифрования дает несомненный временной выигрыш при шифровании большого массива данных, а применение асимметричного шифрования надежно защищает от взлома, а также решает проблему, присущую методам симметричного шифрования – распределение ключей.

В настоящее время именно такие гибридные криптосистемы находят наибольшее применение.

Вычисление случайных чисел

Многие алгоритмы шифрования основаны на разложении простых чисел на множители. В настоящее время спецслужбы могут испытывать трудности в расшифровке сообщений некоторых мессенджеров, поскольку никаких готовых и известных «шифров» просто не существует – генерация простых чисел для создания ключа происходит непосредственно на устройстве, например, на смартфоне пользователя. Причем генерируются числа случайным образом.

В реальности достаточно часто приходится сталкиваемся со случайным порядком. Действительно случайные числа можно получать из различных хаотичных процессов, например, из шума. Слушая такой шум и делая из него выборки, можно получать случайные числа. Например, из телевизионного шума можно получить истинно случайную последовательность. Если такую последовательность представить в виде пути, в котором через каждый шаг направление будет изменяться от значения числа, то можно будет заметить полнейшее отсутствие всякой закономерности. Каждый следующий шаг предсказать будет невозможно.

Случайные процессы являются недетерминированными, то есть непредсказуемыми, в отличии от автоматов, действие которых предсказуемы и воспроизводимы. В 1946 году Джоном фон Нейманом, перед которым встала проблема невозможности сохранения для проведения научных исследований больших случайных чисел на компьютерах того времени, был придуман алгоритм генерации случайных чисел. Алгоритм заключался в выборе первоначального случайного числа (seed), используемое для входных данных, которое затем возводилось в квадрат. Из полученного результата выбирались средние цифры (цифры, находящиеся в середине полученного результата), затем полученное число снова возводилось в квадрат и снова выбирались средние цифры и т.д. этот метод получил название метода серединных квадратов. Он был первым из множества появившихся позднее алгоритмов, генерирующих псевдослучайные числа. Вся последовательность зависит лишь от выбранного начального значения. Один и тот же seed означает одну и ту же последовательность.

Псевдослучайные последовательности отличаются от действительно случайных последовательностей тем, что когда-то обязательно начнет повторяться. Это произойдет, когда некое число встретится и будет использовано в качестве начального значения повторно. Цикл начнется заново. Длину такого цикла называют периодом и он строго ограничен размером seed. Так, если начальное значение представляет собой двухзначное число, то алгоритм выдаст предельно 100 чисел, а затем какое-то начальное значение обязательно снова повторится. Из трехзначного числа нельзя получить более 1000 чисел, поскольку алгоритм за циклится. А из

четырехзначного seed можно получить последовательность не более, чем из 10000 чисел и т.д. Однако, достаточно большое начальное значение может привести к формированию триллионов триллионной последовательности до возникновения повтора.

Следует учитывать, что при псевдослучайной генерации чисел многие последовательности не выпадут никогда, поскольку количество возможных формируемых случайных чисел может быть астрономически огромным.

Использованием вместо случайных чисел псевдослучайных уменьшается пространство ключей до гораздо меньшего пространства начального значения. И чтобы псевдослучайная последовательность была неотличима от действительно случайной нужно, чтобы компьютер не смог перебрать все возможные значения и не смог найти совпадения. Тут всплывает очень важное различие между тем, что возможно сделать теоретически и тем, что возможно сделать за разумное время. Так, у замка с шифром злоумышленник может перебрать все варианты и замок откроется, пусть и за несколько дней или недель. Для псевдослучайных генераторов защита возрастает с длиной начального значения. Если какой-либо мощный компьютер будет перебирать все возможные seed сотни лет, то можно положиться на достаточно секретный шифр вместо совершенно секретного. Но компьютеры становятся быстрее и размер начального значения должен увеличиваться соответственно.

Псевдослучайные алгоритмы избавляет шифровальщика и дешифровщика от необходимости передачи друг другу длинных случайных цепочек. Вместо этого они могут обменяться достаточно коротким случайным начальным значением и получить одну и ту же почти случайную последовательность.

Коды Хаффмана

Алгоритм кодирования, предложенный Хаффманом позволяет эффективно кодировать данные, осуществляя их сжатие, что является весьма ценным при передаче данных по, например, сетям. Сжатие обеспечивается в том числе и тем, что символы, встречающиеся в кодируемом тексте наиболее часто, кодируются меньшим количеством символов, в отличие от тех, которые встречаются реже.

Несмотря на то, что алгоритм является самым оптимальным среди всех алгоритмов именно символьного кодирования, он может сработать плохо, если источник имеет низкую энтропию[1], то есть в сообщении будет много повторяющихся символов и много неповторяющихся. В этом случае энтропия будет меньше 1.

В соответствии с алгоритмом Хаффмана, прежде чем приступить непосредственно к шифрованию букв текста (это могут быть и цифровые, и цветовые последовательности), необходимо определить частоту букв в тексте. В качестве примера возьмем слово «программирование»:

ПРОГРАММИРОВАНИЕ

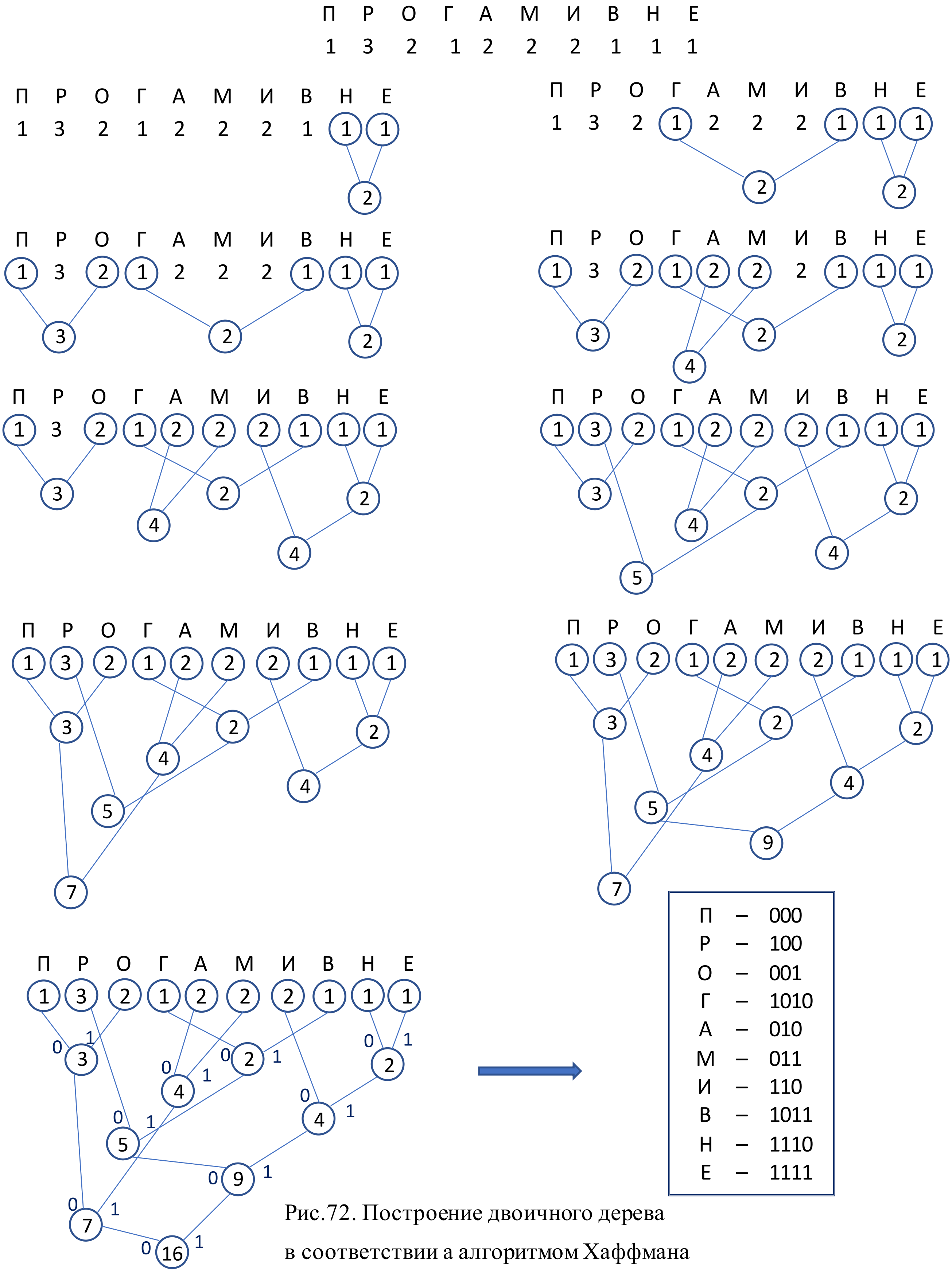
П – 1/16	Р – 3/16	О – 2/16	Г – 1/16	А – 2/16
М – 2/16	И – 2/16	В – 1/16	Н – 1/16	Е – 1/16

Затем следует приступить к построению двоичного дерева (рис.72). Построение дерева начинается с листьев. Согласно алгоритму необходимо попарно поэтапно складывать наименьшие значения. Образующиеся в результате сложения «висячие» вершины (листы) также

[1] Информационная энтропия – в теории информации мера неопределенности источника сообщений, определяемая вероятностями появления тех или иных символов при их передаче.

необходимо сложить по принципу попарного сложения наименьших значений. В результате будет получена одна вершина – корень дерева (дерево будет перевернутым).

На следующем шаге строится непосредственно кодовая таблица.



П	–	000
Р	–	100
О	–	001
Г	–	1010
А	–	010
М	–	011
И	–	110
В	–	1011
Н	–	1110
Е	–	1111

Рис.72. Построение двоичного дерева в соответствии а алгоритмом Хаффмана

Для получения шифра каждой буквы необходимо пройти путь до нее, начиная с корня дерева (снизу вверх). Предварительно левые ветки отмечаются нулями, а правые – единицами (см. рис.72). В результате, букве «П» будет соответствовать код «000», букве «Р» – «100» и т.д., а само слово «Программирование» зашифруется последовательностью чисел 000 100 001 1010 100 010 011 011 110 100 001 1011 010 1110 110 1111.

Специфика кода Хаффмана заключается в его преффиксности, то есть начало одной закодированной буквы не является началом другой. Именно это свойство позволяет избежать ошибок при расшифровке закодированных текстов, даже если зашифрованные слова в нем записаны без единого пробела. Так, если полученную в примере последовательность записать в виде 000100001101010001001101111010000110110101110110 1111, исключая все пробелы, то это никак не повлияет на сложность расшифровки.

Как видно из продемонстрированного выше примера, наиболее часто встречающиеся буквы в результате шифрования получают наименьшее количество знаков, а наиболее редко встречающиеся – наибольшее, что обеспечивает сжатие шифруемых данных. Поэтому рассмотренный алгоритм с некоторыми модификациями крайне широко используется в архиваторах, таких, как zip, графические форматы jpg, некоторых медиа-форматов и др.

Факсимильные аппараты

Еще до появления электронной почты существовал способ передачи различной информации на расстояние с помощью факсимильных аппаратов. Основная функция факсимильной связи заключалась в передаче текстов, чертежей, рисунков, фотографий с листа бумаги отправителя на лист бумаги получателя.

Впервые факсимильный аппарат в том виде, в котором он нам известен, был представлен публике в 20-х годах прошлого века и являлся «детищем» Г. И. Ивса (при непосредственном участии Г. Найквиста).

В 30-х годах прошлого века в СССР были созданы фототелеграфные аппараты, основанные на использовании фотографических методов и материалов для записи фототелеграмм.

Позже появились аппараты, в которых в качестве каналов связи использовались не только телеграфные линии, но и телефонные, а также радиосвязь. В связи с этим появилось новое определение связи – факсимильная связь.

Несмотря на развитие других средств связи с появлением Интернета, факсимильные аппараты и сегодня ни в коей мере не теряют свой популярности.

Реализация такой связи предусматривает наличие трех компонентов: передатчика, линии связи и приемника (рис.73).

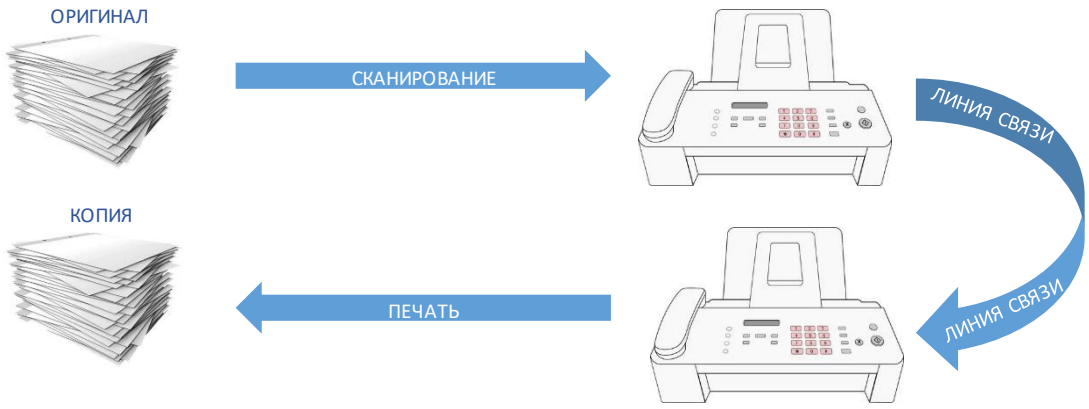


Рис.73. Компоненты факсимильной связи

Все большее распространение в последние годы получает способ записи информации при приеме в виде графических файлов на диск, в облаке и т.д. Впервые же идея использования компьютера для создания интегрированных систем была реализована в 80-х годах прошлого века, кода фирмой GammaLink была выпущена первая из факсимильных плат, позволяющая подключить телефонную линию к компьютеру. Это значительно расширило список функций, выполняемых факсимильный аппаратом, превратив его в телефакс.

Однако, и вопросы безопасной передачи стали актуальнейшей задачей, поскольку при нынешнем технологическом уровне перехват сообщений не представляет собой особо сложную задачу.

В настоящее время выпускается множество факсимильных аппаратов со встроенными устройствами шифрования, а также автономные шифраторы. Зачастую аппараты криптографической защиты подключают в разрыв между самим факсимильным аппаратом и телефонной сетью (рис.74).

Рис.74. Структурная схема подключения аппарата криптографической защиты с факс-модемом под управлением компьютера

Подобные аппараты криптографической защиты осуществляют криптографическое преобразование с помощью соответствующих алгоритмов, что позволяет передавать информацию факсимильными аппаратами по открытым каналам связи. Обычно генерация ключей шифрования госуществляется в таких аппаратах с помощью специальных датчиков случайных чисел.

Структуры больших данных. Проблемы хранения, представления и обработки

В третьем десятилетии двадцать первого века информация представляет собой сырье, инструмент, оружие а также один из самых главных активов, обладать которым стремятся практически все корпорации, страны, государства. За обладание подобными активами разворачиваются настоящие сражения.

Сегодня пришло и осознание правильного использования огромных массивов данных. В наши дни у всех на слуху в любых направлениях бизнеса и технологий термин «большие данные» (Big Data). Этот термин используется при обозначении технологий и инициатив, включающие разнообразные активы данных самых различных категорий. Сами эти активы неизменно и стремительно развиваются и расширяются, становятся слишком многообразными, что значительно осложняет их обработку.

Самыми успешными становятся компании и корпорации, которым удается эффективно хранить и обрабатывать данные.

Тридцать лет назад объем жесткого диска компьютера был ограничен не более чем 10 Гбайт. А любая из нынешних социальных платформ ежедневно обрабатывает полумиллиарда терабайтов данных. Огромную лепту в выработке миллиардов терабайт обновляемой в безостановочном режиме разнообразной информации вносит и повсеместное применение смартфонов, планшетов и других аналогичных устройств. Одной лишь системой учета «кликов» в интернете ежесекундно анализируется многомиллионные действия пользователей сети. Обновление же на разного рода фондовых биржах осуществляется в течение не более нескольких микросекунд. Одновременное подключение миллионов пользователей, каждый из которых осуществляет по несколько действий за секунду, успешно поддерживается игровыми

серверами. И это лишь малая толика реально существующих примеров, демонстрирующих невообразимые темпы генерирования новых данных.

Итак, исходя из этого, Big Data предполагает не только создание больших объемов сложных данных, как структурированных, так и неструктурированных, но и их хранение, извлечение, дальнейший чрезвычайно важный анализ, поскольку на его основании принимаются решения, составляются модели и прогнозы.

Впервые этот термин появился в 2008 году в статье американского ученого Клиффорда Линча. По его мнению большими являлись данные, превышающие объем 150 Гигабайт в сутки. На сегодняшний день этот объем представляется слишком маленьким, однако, новых критериев на сегодняшний день нет.

Набор признаков Big Data (рис.75) изначально был основан на принципе трех V:

• *Volume* (*Объем*) – не менее 150 Гбайт в сутки;

• *Velocity* (*Скорость накопления и анализа*) – большие данные обновляются ежесекундно, соответственно и скорость обработки должна быть максимально высокой. В идеале обработка должна производиться в онлайн режиме;

• *Variety* (*Многообразие*) – данные отличаются большим разнообразием и могут быть представлены в виде таблиц, текста, фотографий и т.д.

В дальнейшем было добавлено еще несколько V (см. рис.75) (количество различается в зависимости от экспертов и корпораций):

• *Veracity* (*Достоверность данных*) – очень важный признак с учетом того, что на основании данных осуществляется принятие решений;

• *Variability* (*Изменчивость*) – данные могут изменяться в зависимости от некоторых факторов, например, продажи в магазинах по сезонам;

• *Value* (*Ценность*) – некоторые из данных могут обладать большей ценность, а некоторые – меньшей;

• *Viability* (*Жизнеспособность*) – фактически представляет собой актуальность собранных данных;

• *Visualization* (*Наглядное представление*) – визуализация данных в виде графика, формулы и т.д.

Кроме вышеперечисленных признаков существует и множество других, таких как исчерпаемость, логическая уникальность, относительность, масштабируемость и др.

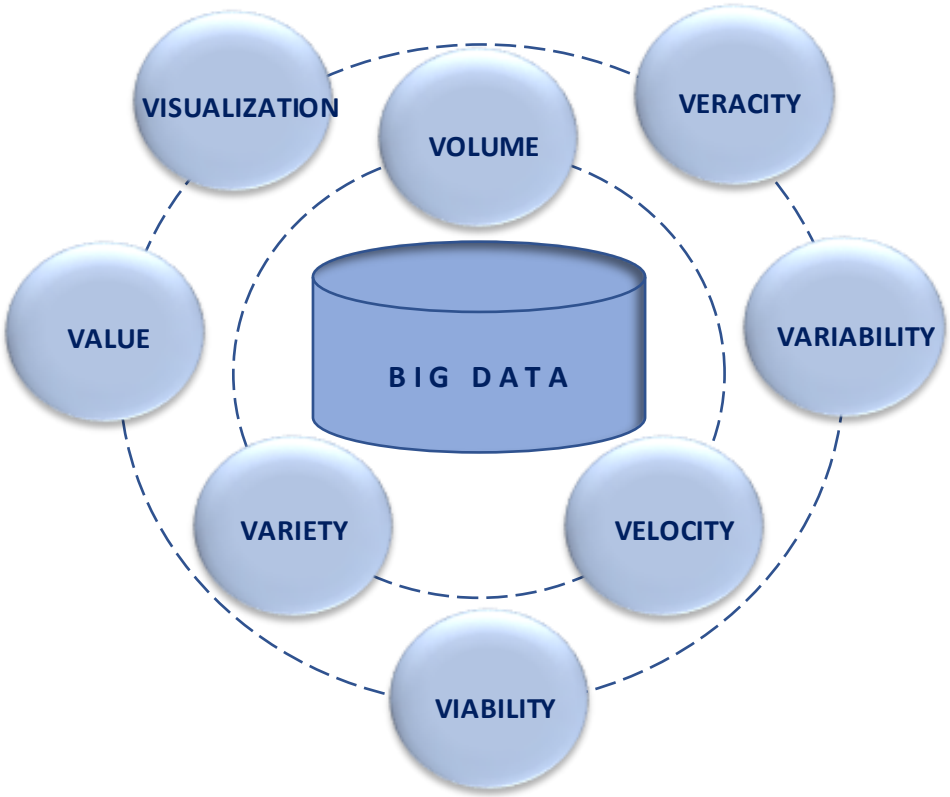


Рис.75. Признаки Big Data

Основными источниками данных являются данные официальной статистики, профили пользователей в социальных сетях, клиентские данных компаний, различные приборы, умные устройства и т.д.

Данные могут быть *структурированными*, то есть представленными в виде таблиц и отношений, как, например, таблицы в базах данных; *полуструктурированными* (или же *слабоструктурированными*), представленными не в виде таблиц, но все же обладающие некоторы-

ми маркерами отделения элементов, как например, информация в файлах журналов; *неструктурированными*, то есть сохраняемыми в виде различных текстов, видео- и аудиофайлов и т.д.

Хранение больших данных осуществляется на серверах, в облачных хранилищах и в озерах данных (Data Lake). Озеро данных представляет собой огромное хранилище разнообразной необработанной неструктурированной информации в ее первоначальном виде из самых разнообразных источников. Такие хранилища обладают достоинствами – они гораздо дешевле привычных баз данных, легко расширяются по мере необходимости, а также и недостатками – огромное количество информации может быть абсолютно бесполезным и может никогда не пригодиться.

Озеро находится в облаке и на сегодняшний день более 70% компаний предпочитают использование облачных хранилищ, а не закупку дорогостоящих серверов, требуемую мощность которых с учетом перспектив довольно сложно предсказать, что может привести как к нехватке вычислительной мощности, так и к простою дорогостоящего оборудования.

При доступе и обработке данных используется несколько подходов. Один из них носит название *Scale-up* (*vertically*) (*Вертикально масштабируемые системы*). Идея заключается в увеличении ресурсов на одном вычислительном узле. То есть при увеличении сложности задач закупается новый сервер (или же компьютер) с более мощными процессорами, большей оперативной памятью и т.д. (рис.76, а).

Второй подход носит название *Scale-out* (*horizontally*) (*Горизонтально масштабируемые системы*). Идея заключается в том, чтобы разбить решаемую задачу на несколько, реализуемых на серверах относительно небольшой мощности, а потом собрать полученные результаты на центральном сервере с целью объединения и возможной дальнейшей обработки (рис.76, б). Огромным плюсом при таком подходе является практически неограниченная расширяемость. По мере роста задач можно не покупать один большой сервер, а докупать необходимое для выполнения задач количество небольших серверов. Финансовая составляющая растет при этом линейно – чем больше данных, тем больше денег надо потратить на приобретение

дополнительных серверов. Однако, есть и недостатки. Необходимо следить за исправностью каждого из серверов и корректно обрабатывать информацию при сбое некоторых из них.

Подход горизонтальной масштабируемости требует и нового подхода к алгоритмам и их сложности. Вводятся новые понятия того, что такое сложность алгоритма и что означает «хороший алгоритм». Ранее мы говорили о том, что при оценке сложности алгоритмов в некоторых случаях коэффициенты являются незначимыми и ими можно пренебречь. В данном же случае, например $O(n^2)/k$, будет означать, что k – это не константа, а то, что алгоритм легко может распараллелиться на k серверов. Естественно, не каждый алгоритм может отвечать подобным требованиям распараллеливания и, если его нельзя переделать соответствующим образом, то считаться хорошим он уже не будет.

а) б)

Рис.76. Масштабируемые системы обработки данных: а) вертикально; б) горизонтально

Для хранения файлов больших размеров и доступа к информации, которая распределена по узлам вычислительного кластера в виде отдельных блоков используется распределенная файловая система Hadoop[1] – HDFS (Hadoop Distributed File System), архитектура которой представлена на рис.77.

[1] Hadoop – проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределенных программ, работающих на кластерах из сотен и тысяч узлов.

Обязательный элемент системы HDFS представлен узлом имен, который также называют управляющим узлом или сервером имен, являющемся единственным в кластере отдельным сервером с программным кодом для управления пространством имен файловой системы. В нем хранится дерево файлов и метаданные файлов и каталогов. Он отвечает за открытие и закрытие файлов, создание и удаление каталогов и т.п.

Узел данных представляет собой один из множества подобных узлов кластера и отвечает за чтение, запись файлов, обработку запросов чтения и записи, выполнение команд от управляющего узла.

Клиент – это пользователь (или приложение), осуществляющий взаимодействие с распределенной файловой системой.

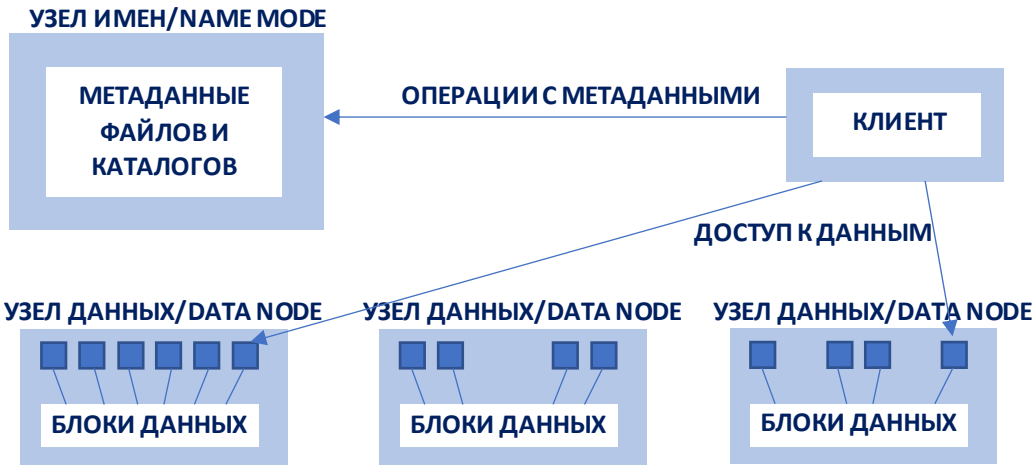


Рис.77. Архитектура HDFS

Распределенная система также хранит копии данных, заботясь о бесперебойной работе отдельных узлов. Основное преимущество заключается в расширяемости на тысячи узлов.

Однако, это всего лишь файловая система, с помощью которой можно хранить файлы, читать их, записывать, но невозможно осуществлять различного рода запросы и операции. Для этого предназначены базы данных. Стандартная классическая Sql пока еще не может работать сверху на распределенной файловой системе. Поэтому наибольшее распространение получила база данных NoSql, являющаяся не реляционной базой данных.

Классические системы баз данных поддерживают целостность данных, что не всегда возможно в распределенных файловых системах. Проблема заключается в том, что в распределенных файловых системах поступающие данные копируются на несколько серверов, что занимает определенное время и при обращении к этим данных в процессе копирования может не быть корректным. После окончания дублирования целостность данных будет соблюдена, но неизвестно через какой промежуток времени это произойдет.

В связи с этой проблемой сформулирована CAP-теорема, определяющая возможный выбор только двух компонентов из трех возможных (рис.78).

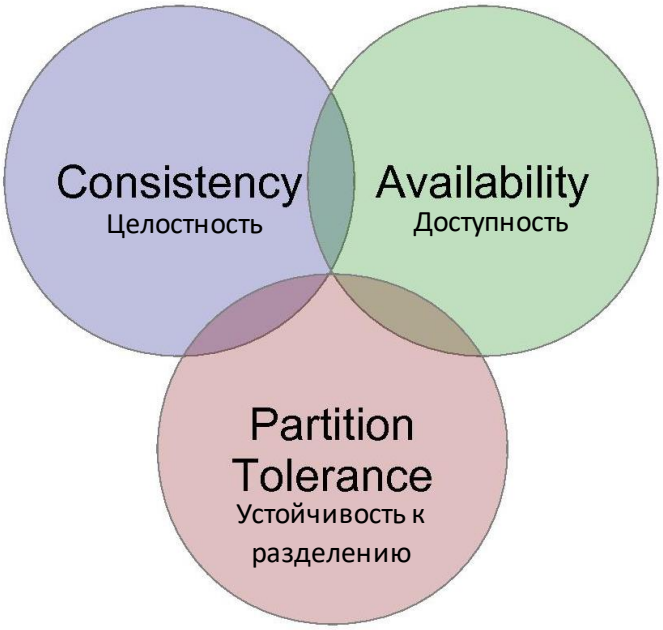


Рис.78. CAP-теорема

Как правило, выбирают возможность распределения на несколько серверов и доступность данных, жертвуя их целостностью.

Несмотря на некоторые проблемы, возникающие при хранении и обработке больших данных, их использование больших данных дает множество преимуществ, давая доступ к необходимой информации, на основе которой принимаются управленческие решения, прогнозируется спрос на товары, разрабатываются грамотные бизнес-стратегии и многое другое.

Алгоритмы параллельных вычислений. *Переходы по указателям. Эффективность по затратам. Эффективная паралельная обработка префиксов. Распараллеливание алгоритмов поиска и сортировки*

14

Переходы по указателям. Эффективность по затратам

Все алгоритмы, рассмотренные ранее в предыдущих разделах в рамках учебного курса «Алгоритмы и структуры данных» являлись последовательными, то есть выполняемые на одном процессоре. Критерием эффективности работы подобных алгоритмов являлось время их работы в худшем случае, то есть фактически подсчитывалось количество выполняемых операций. Решение же сложнейших современных задач требует более эффективных подходов. В частности, одним из таких подходов является использование некоторого количества вычислительных устройств с целью одновременного (в параллельном режиме) выполнения разных частей одной программы. В этом случае помимо времени работы алгоритма (вычислительной сложности) также рассматривается понятие общих затрат алгоритма, представляющее собой произведение времени работы алгоритма и количества процессоров.

Как правило, в многопроцессорной системе процессоры обозначаются буквой «P» и нумеруются от 0 до n-1 (P_o, P_1, … P_{n-1}). Процессоры могут одновременно выполнять логические и арифметические операции.

Доступ к ячейкам памяти осуществляется по их адресу. В зависимости от способности процессоров системы считывать и записывать информацию из одной и той же ячейки принято различать несколько моделей вычислительных устройств:

- *EREW* (*Exclusive Read Exclusive Write*), запрещающая параллельный доступ к одной и той же ячейке как для чтения, так и для записи;
- *CREW* (*Concurrent Read Exclusive Write*), разрешающая параллельный доступ к одной и той же ячейке для чтения, но запрещающая параллельный доступ к одной и той же ячейке для записи;

- *ERCW* (*Exclusive Read Concurrent Write*), запрещающая параллельный доступ к одной и той же ячейке для чтения, но разрешающая параллельный доступ к одной и той же ячейке для записи;
- *CRCW* (*Concurrent Read Concurrent Write*), разрешающая параллельный доступ к одной и той же ячейке как для чтения, так и для записи.

Реализация каждой из этих моделей содержит свои нюансы. Например, разрешение параллельной записи в одну и ту же ячейку может предполагать запись одного и того же значения, либо значения, поступающего от процессора с наименьшим номером, либо среднего значения, поступающих от нескольких процессоров значений и т.д.

Часто при решении многопроцессорных задач следует определить номер элемента в списке. Осуществляется это с помощью специального алгоритма с использованием указателей.

Так, например, пусть имеется односторонний список, состоящий из восьми элементов. Каждый из элементов представляет собой ячейку памяти, содержащую два поля, одно из которых содержит указатель (адрес) следующего элемента списка – next(i), где i – адрес элемента i. Последний элемент справа не будет содержать указатель и будет обозначаться символом ∅, обозначающим то, что указатель пуст. Предполагается, что порядковый номер этого последнего элемента будет нулем. Слева от него элемент будет иметь порядковый номер 1, левее него – 2 и т.д. Порядковый номер i-того элемента обозначим как d(i). Каждому из элементов сопоставлен свой процессор, номер которого может отличаться от порядкового номера элемента. Задача сводится к определению расстояния от каждого элемента до крайнего элемента с нулевым указателем. Для крайнего правого элемента это расстояние будет равно нулю, для расположенного от него слева элемента – единице, далее, для следующего элемента – 2 и т.д. Если бы этот алгоритм реализовывался на однопроцессорном устройстве, то достаточно было бы поставить счетчик и подсчитывать количество элементов справа налево, обратив предварительно указатели. Обращение списка для однопроцессорного алгоритма будет O(1), а число операций O(n).

Шаги, выполняемые при реализации алгоритма определения номеров элементов графически продемонстрированы на рис.79. Первый шаг предусматривает инициализацию. Все элементы d(i) кроме правого крайнего элемента предполагаются равными единице. Для каждого из процессоров фрагмент псевдокода на этом этапе будет иметь вид:

```
Если next(i)= ∅,
    то d(i)=0
    в противном случае d(i)=1.
```

То есть, если поле указателя не является пустым, то величина d(i)=1, а для пустого указателя d(i) указывает на искомую величину.

Эти операции процессорами могут выполняться параллельно, поэтому время, затрачиваемое на подобную операцию будет составлять O(1).

На втором шаге изменяется величина d(i), для каждого элемента i увеличиваясь на величину d от следующего для этого элемента. Изменение касается тех элементов, у которых поле указателя не является пустым. То есть для каждого элемента на втором шаге величина d увеличивается на единицу по сравнению с величиной d следующего за ним элемента. Величина d для второго справа элемента останется прежней (единица), поскольку следующий за ним элемент содержит пустой указатель. Таким образом, искомая величина d будет определена для двух элементов справа.

Процедура будет повторяться до тех пор, пока в списке имеется хоть один элемент с непустым полем указателя. На этом же шаге для каждого элемента следующим объявляется следующий от следующего элемент (показаны стрелками на рис.79 – шаг 2).

Фрагмент псевдокода для каждого процессора этой части алгоритма будет иметь вид:

```
Если next(i)≠ ∅,
    то d(i)= d(i)+ d(next(i)),
        next(i)= next(next(i)).
```

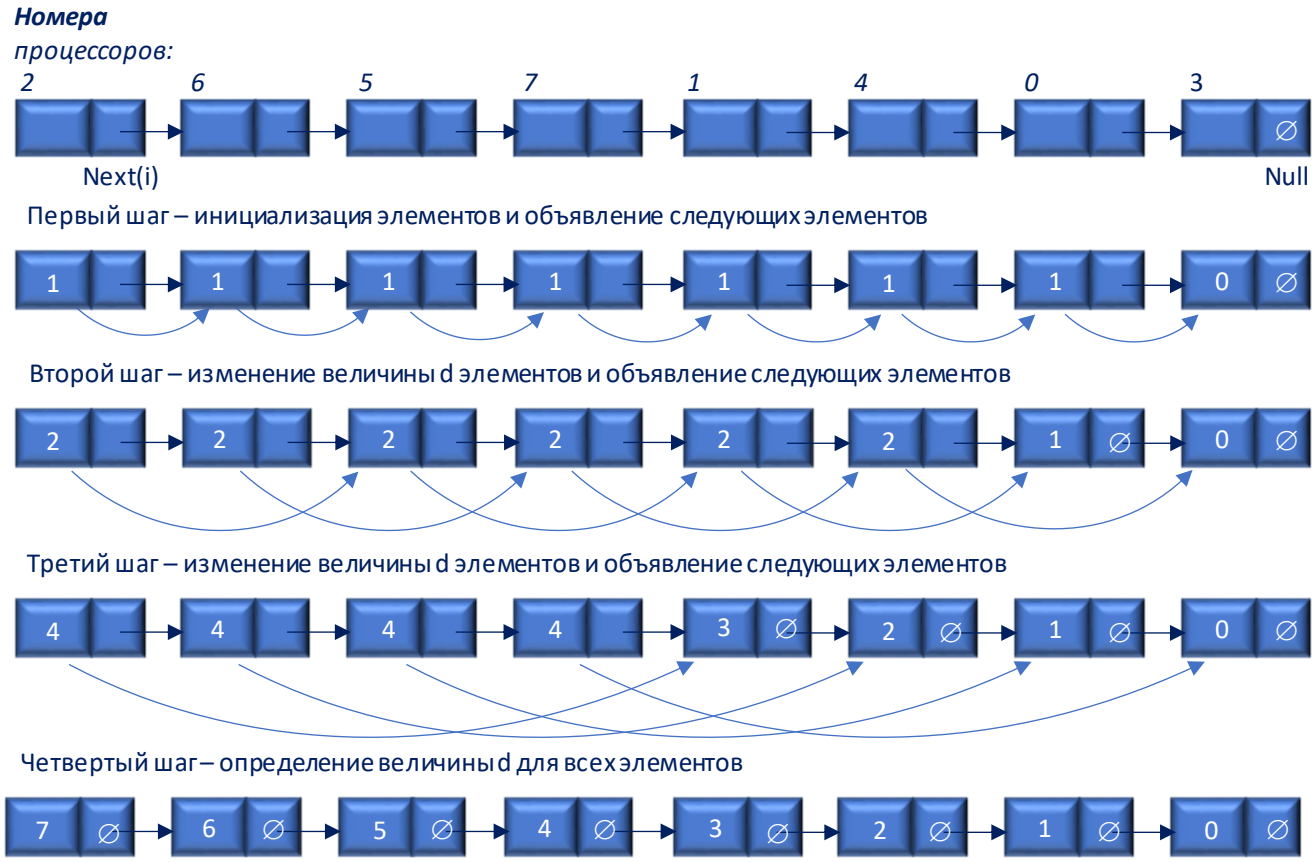


Рис.79. Графическая иллюстрация алгоритма определения номеров элементов

Как видно из рис.79 количество элементов, у которых указатель содержит пустое поле с каждой итерацией увеличивается в два раза. То есть алгоритм будет работать $O(\log_2 n)$. Для вычисления же затрат следует умножить время работы алгоритма на число процессоров n – $O(n\log_2 n)$. Как видим, с точки зрения времени выполнения (сложности) алгоритма, он достаточно эффективен относительно алгоритма для однопроцессорного устройства (как было замечено выше, вычислительная сложность составляет $O(n)$). Однако, с точки зрения затратности подобный алгоритм не является достаточно эффективным по сравнению с алгоритмом для однопроцессорного устройства.

Эффективная паралельная обработка префиксов

Рассмотренный выше алгоритм является частным случаем задачи параллельной обработки префиксов.

В алгоритме обработки префиксов рассматривается бинарная ассоциативная операция, обозначаемая символом «*», для которой (a*b)*c=a*(b*c). Такой операцией может быть сложение или умножение.

Задача заключается в нахождении по входным величинам $x_1, x_2, \ldots, x_n$ следующих величин:

$$y_1=x_1,\ y_k=y_{k-1}*x_k=x_1*x_2*\ \ldots\ *x_k \text{ для } k=2, \ldots, n.$$

Сначала введем некоторые обозначения:

$$[i, j]=x_i*\ \ldots\ *x_j,\ 1\le i<j\le n,$$

$$[i, i]=x_i,$$

$$[i, k]= [i, j]*[j+1, k],\ 1\le i\le j<k\le n.$$

Начинается реализация алгоритма с образования одностороннего списка, каждый элемент (за каждый элемент отвечает один процессор) которого так же, как и в предыдущем примере состоит из двух полей: правое содержит указатель на следующий элемент, а в левом на этапе инициализации находится величина x_i. То есть в каждый элемент на первом шаге помещена входная информация. Псевдокод для этой части алгоритма можно представить в виде следующего фрагмента:

```
Для каждого процессора
    x[i]=y[i].
```

Дальнейшие шаги алгоритма для каждого процессора выполняются в соответствии со следующим фрагментом алгоритма:

```
Если next(i)≠ ∅,
    то y[next(i)]=y[i]*y[next(i)],
        next(i)= next(next(i)).
```

Шаги, выполняемые при реализации алгоритма параллельной обработки префиксов графически продемонстрированы на рис.80.

На втором шаге происходит выполнение операции * сначала между первым и его соседним – вторым элементом. Полученное значение ([1,1]*[2,2]=[1,2]) записывается во вторую ячейку, а значение первой остается неизменным. Затем операция * проводится для второго и третьего элемента, а полученное значение записывается в третью ячейку. Далее операция повторяется для третьей и четвертой ячейки с занесением результата в четвертую ячейку и т.д. Одновременно производится определение следующих (следующие следующих) элементов для каждого из элементов для последующей итерации. Следует учитывать, что в правой части пустой указатель указывается для всех тех элементов, которые ссылались на элементы с пустыми указателями (для которых следующими были элементы с пустыми указателями).

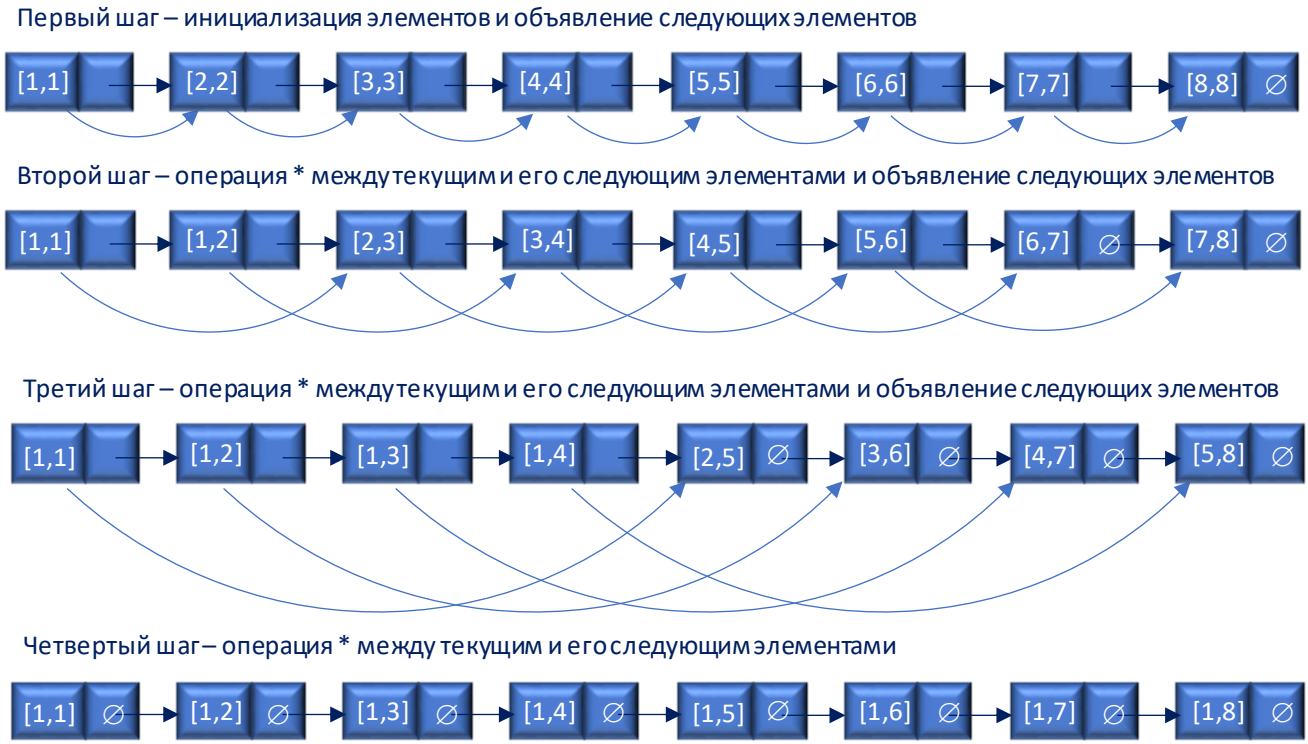


Рис.80. Графическая иллюстрация алгоритма параллельной обработки префиксов

На третьем шаге производится операция * между первым и следующим для него третьим элементом с записью результата в третий элемент. Затем то же действие повторяется для второго и четвертого элемента с записью результата в четвертый элемент и т.д. На этом же шаге определяются следующие элементы для последующей итерации. Количество элементов с пустыми указателями удваивается.

На четвертом шаге операция повторяется для первого и пятого, второго и шестого, третьего и седьмого, четвертого и восьмого элементов. В правой части всех элементов оказываются пустые указатели и параллельная обработка префиксов завершается.

Данный алгоритм не требует того, чтобы процессоры записывали или считывали информацию в одну и ту же ячейку одновременно, что означает возможность его реализации на модели *EREW* устройства, как и в предыдущем примере.

Сложность и затраты алгоритма обработки префиксов также аналогичны сложности и затратам алгоритма определения номеров элементов.

Распараллеливание алгоритмов поиска и сортировки

Распараллеливание некоторых, рассмотренных нами в предыдущих разделах алгоритмов, значительным образом ускоряет их выполнение.

Так, например, как уже упоминалось ранее, бинарный поиск является одним из наиболее эффективных методов для поиска элемента в отсортированном массиве. Однако, с ростом размера массива и количества выполняемых операций, возникает необходимость в оптимизации алгоритма. Именно для этого можно применить распараллеливание методов бинарного поиска.

Одним из подходов к распараллеливанию бинарного поиска является использование многопоточности. Представьте, что у вас имеется массив, который необходимо просмотреть в поисках определенного элемента. Вместо того, чтобы одновременно проверять каждый элемент последовательно, вы можете разделить массив на несколько частей и присвоить каждый фрагмент отдельному потоку для параллельного поиска. Таким образом, время выполнения поиска будет значительно сокращено.

Распараллеливание методов бинарного поиска может также применяться при поиске в нескольких отсортированных массивах одновременно. В этом случае каждый массив может быть разделен на подмассивы, которые параллельно проверяются на наличие нужного элемен-

та. Такой подход может быть полезен в случаях, когда необходимо выполнить поиск по большому количеству массивов одновременно.

Другой подход к распараллеливанию бинарного поиска – это использование параллельных вычислений на графическом процессоре GPU (Graphics Processing Unit), представляющем собой мощное устройство, которое изначально создавалось для обработки и отображения графики, однако, проявило невероятные возможности и в других областях, таких как вычислительная математика и обработка параллельных задач и активно используется, в том числе, в серверных центрах облачных вычислений, где он обрабатывает задачи машинного обучения и искусственного интеллекта. GPU обладает большим количеством ядер и параллельных вычислительных ресурсов, что позволяет выполнять множество операций одновременно. Этим свойством GPU можно воспользоваться и в случае бинарного поиска. Разделяя массив на части и распределяя эти части по ядрам GPU, можно значительно ускорить поиск.

Все эти методы распараллеливания бинарного поиска позволяют достичь более высокой производительности и эффективности алгоритма. Они особенно полезны при работе с большими объемами данных или при поиске элементов в реальном времени. Распараллеливание методов бинарного поиска является достаточно актуальной темой и активно исследуется в научно-исследовательских работах для поиска новых способов оптимизации этого алгоритма.

Распараллеливание значительно улучшает эффективность различных алгоритмов сортировки, среди которых, например:

1. Параллельная сортировка слиянием: одной из наиболее эффективных техник распараллеливания сортировки является параллельная сортировка слиянием. В этом методе массив данных разбивается на несколько подмассивов, которые сортируются независимо друг от друга на разных ядрах или процессорах. Затем отсортированные подмассивы сливаются вместе для получения отсортированного массива данных. Этот метод обеспечивает эффективное использование многопоточности и позволяет значительно сократить время выполнения сортировки, особенно для больших и сложных массивов данных.

2. Параллельная быстрая сортировка: быстрая сортировка - один из самых популярных и эффективных алгоритмов сортировки. При распараллеливании этого метода массив данных разбивается на подмассивы, которые сортируются независимо друг от друга на разных процессорах или ядрах. Затем отсортированные подмассивы объединяются вместе, чтобы получить отсортированный массив данных. В параллельной быстрой сортировке каждый поток выполняет свою собственную "быструю сортировку" для своей части данных. Это позволяет существенно увеличить скорость сортировки и улучшить производительность алгоритма.

3. Параллельная сортировка подсчетом: сортировка подсчетом является простым и эффективным алгоритмом сортировки, который может быть легко распараллелен. При такой сортировке каждый элемент массива подсчитывается и затем суммируется, чтобы получить количество элементов меньше или равных каждому элементу массива. Затем элементы массива перегруппируются на основе этой информации. При распараллеливании каждый поток может быть назначен для подсчета элементов своей части данных, что позволяет существенно ускорить процесс сортировки. Параллельная сортировка подсчетом может быть особенно полезна при сортировке больших массивов данных.

4. Параллельная сортировка пузырьком: сортировка пузырьком известна как один из самых простых и медленных алгоритмов сортировки. Однако, при распараллеливании, этот алгоритм может обрести новую эффективность. Параллельная сортировка пузырьком разбивает массив данных на несколько подмассивов, которые сортируются независимо друг от друга на разных потоках или ядрах. Затем отсортированные подмассивы объединяются вместе, чтобы получить отсортированный массив данных. Хотя параллельная сортировка пузырьком может быть менее эффективна, чем другие алгоритмы сортировки, она все равно может быть полезной в некоторых сценариях, особенно при сортировке малых массивов данных или когда другие методы сортировки недоступны.

15 ***Задачи комбинаторики.*** *Алгоритмы решения полиномиальных и экспоненциальных задач и их эффективность. Математика компьютерной томографии*

Прежде, чем мы перейдем непосредственно к рассмотрению алгоритмов решения полиномиальных и экспоненциальных задач и увидим, насколько они тесно связаны с комбинаторикой, разъясним, что что это за понятие и что оно в себя включает.

В повседневной жизни мы часто сталкиваемся с решением различных задач выбора, оптимального расположения элементов. Тем более часто подобные задачи встречаются при составлении алгоритмов математических задач. И, конечно же, не только математических. Достаточно задуматься о том, что наличие примерно трех десятков букв в алфавите любого из языков дало возможность с помощью всевозможных их комбинаций не только составлять отдельные слова, но и написать огромное количество книг. А на всего лишь сотне химических элементов и вовсе построена вся наша Вселенная.

Для оптимального же просчета всех возможных комбинаций предназначен специальный раздел математики, носящих название комбинаторика или же комбинаторных анализ.

Сегодня знание этой дисциплины помогает справиться с многочисленными задачами в самых различных сферах и областях знаний. В качестве примера можно отметить генетику, лингвистики, статистику и, конечно же информатику, поскольку именно с помощью комбинаторики можно получить ответ на то, сколько различных комбинаций, удовлетворяющих заданным условиям, можно получить из заданных объектов, предметов и т.д.

Кроме того, знание основ комбинаторики дает возможность подсчитать не только количество возможных и оптимальных комбинаций, но и лучше понять саму суть тех или иных явлений, различных статистических закономерностей, но и вырабатывает доказательные навыки.

К числу типичных задач комбинаторики относятся следующие:

- определение количества комбинаторных комбинаций (доказательство или опровержение их существования);
- нахождение практически пригодного алгоритма построения комбинаций;
- определение свойства класса комбинаторных конфигураций.

Первое правило комбинаторики формулируется как правило суммы и гласит: *если объект A можно выбрать m способами, а объект B n способами, то выбор либо A, либо B можно осуществить (m+n) способами*.

В качестве примера можно привести выбор кратного 2 и кратного 5 чисел из множества чисел {2 4 5 8 15 16 17 18 21 23 25}. В этом множестве числа кратные 2 представлены пятью числами – 2, 4, 8, 16, 18. А кратные 5 – тремя числами – 5, 15, 25. Следовательно, количество способов выбора числа кратного либо 2, либо 5, составляет 5+3=8.

При этом следует следить, чтобы одна и та же комбинация не попадала и в класс A, и в класс B. Если совпадения есть и их число k, то правило сложения запишется в виде *(m+n)-k*.

Например, если необходимо произвести выбор кратных 2 или же кратных 5 чисел из множества чисел {2 4 5 8 10 15 16 17 18 20 21 23 25}, то следует учитывать, что число 10 и 20 являются одновременно кратными и для 2, и для 5. Исходя из того, что в этом множестве чисел число, кратное 2 составляет m=7, число чисел, кратное 5 составляет n=5, а число совпадений (т.е. числа кратные и 2, и 5) – k=2, расчет количества способов выбора из множества чисел числа, кратного либо 2, либо 5, сводится к формуле (m+n)-k=(7+5)-2=10.

Второе основное правило комбинаторики – правило произведения – заключается в следующем: *если объект A можно выбрать m способами, а объект B n способами, то выбор пары (A,B) в указанном порядке можно осуществить (m*n) способами*. Это же можно сформулировать на языке произведения множеств: если множество A состоит их m элементов, а множество B состоит из n элементов, то прямое произведение этих множеств A*B состоит из m*n элементов.

Следует учитывать, что как правило суммы, так и правило произведения обобщаются на произвольное количество слагаемых или сомножителей соответственно.

Для формулировки и решения комбинаторных задач используются модели комбинаторных конфигураций, в частности:

- размещения;
- перестановки;
- сочетания;
- композиция числа;
- разбиение числа.

Размещением (из n по k (A_n^k)) называют различные комбинации объектов количеством k из имеющихся n объектов. Порядок размещения элементов учитывается.

Различают размещение без повторений и с повторениями. В первом случае количество возможных размещений рассчитывается по формуле

$$A_n^k = \frac{n!}{(n-k)!}, \quad (31)$$

где k – количество элементов в расстановке, а n – общее количество элементов.

Например, если студенты первого курса изучают различных предметов за семестр, то количество способов составления расписания на день таким образом, чтобы в нем было 4 разных предмета составляет

$$A_8^4 = \frac{8!}{(8-4)!} = \frac{40320}{24} = 1680.$$

На рис.81,а продемонстрирован еще один пример, в котором число размещений трех различных фигур парами рассчитывается в соответствии с формулой (31).

Формула размещения с повторением имеет вид

$$A_n^k = n^k. \quad (32)$$

На рис.81,б продемонстрирован пример, в котором число размещений трех различных фигур парами рассчитывается в соответствии с формулой (32)

$$A_n^k = A_3^2 = 3^2 = 9.$$

Перестановка представляет собой упорядоченный набор без повторений из n элементов по n. Общее количество перестановок из n элементов обозначается как P_n

$$P_n = A_n^n = n! \qquad (33)$$

Например, формула (33) позволяет определить число перестановок трех книг на полке. Графически перестановка продемонстрирована на рис.82.

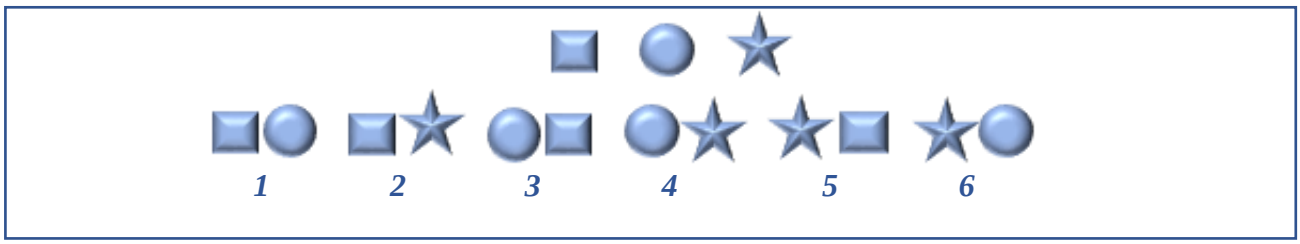
Сочетания представляют собой набор элементов, в котором порядок расстановки не важен, т. е. сочетанием из n по k элементам называются расстановки размера k, составленные из n имеющихся элементов, которые отличаются друг от друга только составом, но не порядком элементов

$$C_n^k = \frac{n!}{k!(n-k)!}, \qquad (34)$$

Например, имеется 4 цифры – {1, 4, 5, 7}. Необходимо определить количество сочетания 2 цифр из этого набора. В соответствии с формулой (34) получим

$$C_4^2 = \frac{4!}{2!(4-2)!} = \frac{4!}{2!*2!} = \frac{24}{2*2} = 6.$$

Графически сочетания продемонстрированы на рис. 83.



а)

б)

Рис.81. Размещение: а) без повторений; б) с повторениями

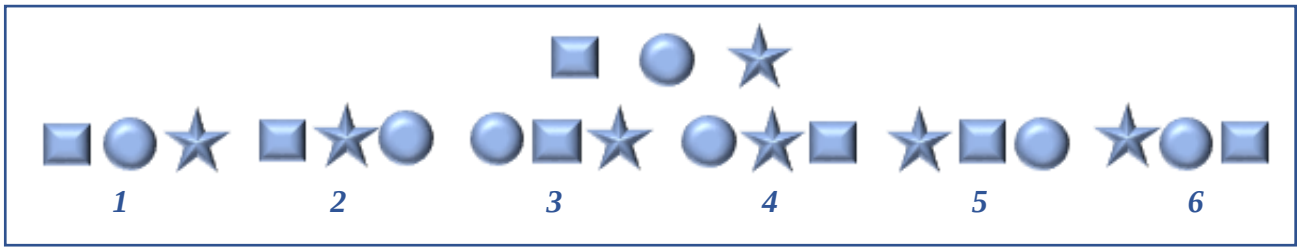


Рис.82. Перестановка

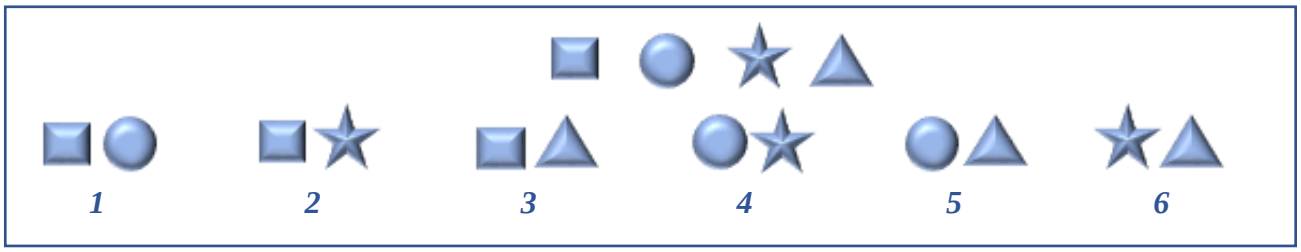


Рис. 83. Сочетания

В теории чисел представление целого числе в виде упорядоченной суммы целых положительных слагаемых носит название композиции числа. Слагаемые, которые входят в композицию именуются частями, а их количество – длиной композиции.

Так, например, для числа 5 существует 16 композиций:

$$
\begin{aligned}
5 &= 5 = \\
&= 4 + 1 = \\
&= 3 + 2 = \\
&= 3 + 1 + 1 = \\
&= 2 + 3 = \\
&= 2 + 2 + 1 = \\
&= 2 + 1 + 2 = \\
&= 2 + 1 + 1 + 1 = \\
&= 1 + 4 = \\
&= 1 + 3 + 1 = \\
&= 1 + 2 + 2 = \\
&= 1 + 2 + 1 + 1 = \\
&= 1 + 1 + 3 = \\
&= 1 + 1 + 2 + 1 = \\
&= 1 + 1 + 1 + 2 = \\
&= 1 + 1 + 1 + 1 + 1.
\end{aligned}
$$

В общем случае существует 2^{n-1} композиций числа n.

Разбиение числа отличается от композиции тем, что не учитывает порядок следования частей. То есть, например, 4+1 и 1+4 считается одинаковыми разбиениями. С учетом этого, разбиение числа 5 будет представлено следующим образом:

5 = 5 =
= 1 + 1 + 1 + 1 + 1 =
= 1 + 1 + 1 + 2 =
= 1 + 1 + 3 =
= 1 + 2 + 2 =
= 1 + 4 =
= 2 + 3.

Как видим, всего существует 7 разбиений числа 5.

Знание основных комбинаторных методов значительно помогает в формулировании и реализации некоторых алгоритмов.

Алгоритмы решения полиномиальных и экмпоненциальных задач и их эффективность

Комбинаторные методы достаточно часто встречаются на практике в различных алгоритмах, некоторые из которых мы уже рассматривали. Однако, среди комбинаторных задач крайне мало таких, которые решаются эффективно за полиномиальное время.

Например, широко известна задача коммивояжера, которая реализуется на графах и о которой мы уже упоминали в разделе 9. В соответствии с этой задачей коммивояжер должен посетить все вершины графа по одному разу и вернуться в исходную. Такой замкнутый контур, по которому происходят перемещения коммивояжера называется *гамильтоновым циклом*. Оптимизационный вариант задачи заключается в выборе минимального пути. В этом случае каждое из ребер имеет длину, выражаемую в числах (вес). Из всех замкнутых контуров (гамильтоновых циклов) выбирается контур минимальной длины.

При поиске всех замкнутых контуров приходится решать множество задач с применением теории перебора и комбинаторики. Предварительно с помощью матрицы смежности следует установить наличие ребер между каждой парой вершин. Это позволит убедиться в наличии гамильтоновых циклов. Количество всевозможных перестановок, то есть количество га-

мильтоновых циклов в полном графе с n вершинами равно $(n-1)!/2$. В худшем случае придется перебрать все гамильтоновы циклы. И, если граф содержит, например, всего пять вершин, то количество гамильтоновых циклов будет равно 12 ((5-1)!/2=4!/2=24/12=12). То есть, существует 12 абстрактных графов, содержащих 5 вершин и имеющих гамильтонов цикл. А вот при наличии, например, 100 вершин придется столкнуться в практически невыполнимой вычислительной сложностью при вычислении факториала числа 99 ((100-1)!/2=99!/2). Соответственно, решить подобную задачу за полиномиальное время невозможно и она относится к числу экспоненциальных задач.

Некоторые из наиболее известных задач такого типа уже упоминались ранее в разделе 9. Все они являются комбинаторными.

Однако, существует ряд задач, в которых есть определенные подходы, позволяющие некоторым, а иногда и существенным образом, сократить количество необходимых вычислительных операций.

Например, в задаче перемножения двух чисел, скажем, n_1 на n_2, по сути необходимо число n_1 просуммировать n_2 количество раз. На первый взгляд, такая задача не представляется сложной. Алгоритм потребует арифметических операций порядка n_1*n_2, Если же таким способом сложения мы попытаемся перемножить достаточно большие числа, например, миллиард на миллиард, то есть $10^9*10^9=10^{18}$, то задача будет экспоненциальной сложности. Но, в то же время, алгоритм умножения обычным методом в столбик представляет собой полиномиальный алгоритм, сложность которого даже в самом худшем случае будет сверху ограничиваться некоторым полиномом.

Также обстоит дело и с возведением числа в степень. Например, использование бинарного алгоритма возведения в степень n позволяет добиться того, чтобы задача была решена за $O(\log n)$ умножений в противовес n умножениям, которые приходится выполнять при обычном возведении в степень.

Таким образом, в некоторых случаях можно с помощью различных приемов добиться перевода задачи из экспоненциальной в полиномиальную сложность.

На самом деле, комбинаторных задач, решаемых за полиномиальное время немного. К таким задачам, например, относится и алгоритм поиска наибольшей общей подпоследовательности, рассмотренный ранее в разделе 8.

Здесь же можно отметить и достаточно широко известную задачу о поиске МОД (минимального остовного дерева), в соответствии с которой во взвешенном неориентированном графе необходимо определить подмножество ребер, отвечающих определенным условиям: в подмножество должны входить все вершины; сумма весов ребер должна быть минимальной. На практике такая задача используется при строительстве, например, дорог с минимизацией расходов непосредственно на их строительство. Для решения подобной задачи известны такие алгоритмы, как Крускала, Прима, Борувки и др.

Однако, и в полиномиальном случае алгоритм может оказаться неэффективным. Это происходит тогда, когда мультипликативная константа перед полиномом, либо сама степень полинома окажется слишком большой по значению. Но такие ситуации крайне редки.

В свою очередь, и экспоненциальные алгоритмы могут оказаться эффективными. Например, в случаях когда числовые параметры, используемые на практике при решении конкретной задачи, оказываются невелики. Однако, и такие случаи крайне редки.

Математика компьютерной томографии

Математика компьютерной томографии представляет собой область, где алгоритмы и методы обработки измерений играют особо важную роль, поскольку в процессе сканирования компьютерным томографом получают данные, которые необходимо анализировать и обрабатывать.

Например, одна из проблем связана с решением обратной задачи томографии, которая заключается в определении внутреннего распределения поглощающего материала в объекте на основе измерений его проекций.

Также важным направлением является реконструкция изображений, где с помощью аналитических и итеративных алгоритмов восстанавливаются 2D или 3D изображения. И здесь особую роль играет комбинаторика, обеспечивая методы для решения различных задач восстановления и анализа трехмерной структуры объектов на основе двумерных проекций. Для этого используется преобразование Радона, которое представляет собой интегральное преобразование вдоль прямых, проходящих через объект. После преобразования получается набор проекций, которые далее используются для восстановления трехмерной структуры. Комбинаторная аналитика здесь применяется для определения наилучшего соответствия между двумерными проекциями и трехмерной моделью объекта.

Еще одним аспектом математики компьютерной томографии является разработка методов снижения дозы излучения при сканировании, сохраняя при этом качество изображений. Для этого используются оптимальное планирование траекторий исследования, а также искусственный интеллект.

Сегментация (разделение изображений на различные области) и визуализация изображений также важные задачи в этой области, где разрабатываются алгоритмы для выделения структур и органов на изображениях, а также их визуализация для удобной интерпретации и анализа.

Кроме того, компьютерная томография может быть использована для анализа динамики процессов внутри организма, и математика компьютерной томографии занимается разработкой алгоритмов для обработки временных данных и изучения динамических свойств.

В целом, математика компьютерной томографии играет важную роль в развитии методов диагностики и лечения, а также в исследованиях в медицинской физике и биомедицинских науках.

Итак, нами были рассмотрены некоторые, связанные с основными алгоритмическими концепциями и способами организации данных вопросы, которые являются неотъемлемой частью образования в сфере информационных технологий и играют крайне важную роль в современном мире.

В заключение хотелось бы еще раз подчеркнуть важность изучения алгоритмов и структур данных в сфере информатики и программирования. Каким бы сложным и запутанным ни казался этот предмет на первый взгляд, он является фундаментальным и неотъемлемым элементом в создании эффективных и оптимальных компьютерных программ и систем.

Прежде всего, изучение алгоритмов позволяет нам разрабатывать решения для различных задач – от простых сортировок до сложных и высоконагруженных алгоритмов обработки графов или криптографии. Кроме того, оно помогает нам разобраться в основах комбинаторики и анализе сложности алгоритмов, что является важной составляющей при выборе оптимального решения для конкретной задачи.

Структуры данных, такие как массивы, деки, стеки, деревья и хеш-таблицы, позволяют нам эффективно хранить и обрабатывать информацию. Изучение этих структур поможет вам научиться выбирать наиболее подходящую для каждой задачи, а также уметь оценивать их производительность и сложность.

Криптографические алгоритмы и основы криптографии – это еще одно важное направление в изучении информатики. Надежная защита информации становится все более актуальной задачей в современном мире, и понимание основных принципов криптографии является ключевым для создания безопасных систем.

Сжатие данных и коды Хаффмана – это область, где можно применить знания о структурах данных и алгоритмах для оптимизации использования памяти и ускорения передачи

информации. Изучение этих тем поможет вам понять, как улучшить эффективность систем обработки и хранения данных.

В заключение, изучение алгоритмов, структур данных, криптографии и других рассмотренных в данном учебнике тем – это инструмент, который позволит вам разработать более эффективные и масштабируемые программы и системы. Овладение этими навыками поможет вам стать успешными и востребованными специалистами в области информационных технологий.

И, наконец, знание основ алгоритмов и структур данных представляет собой фундамент для дальнейшего обучения и успешной карьеры в сфере IT.

Список литературы

1. 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.
2. Фофанов О.Б. Алгоритмы и структуры данных: учебное пособие/ О.Б.Фофанов; Томский политехнический университет. – Томск: Изд-во Томского политехнического университета, 2014. – 126 с.
3. Зачем разработчику знать алгоритмы и структуры данных? https://proglib.io/p/zachem-razrabotchiku-znat-algoritmy-i-struktury-dannyh-2022-06-08
4. Асимптотический анализ: нотации Big-O, Omega и Theta. https://ravesli.com/asymptotic-analysis/
5. Big O Notation: что это такое и как её посчитать. https://skillbox.ru/media/code/big-o-notation-chto-eto-takoe-i-kak-eye-poschitat/
6. Доказательство правильности программ. https://иванов-м.рф/informatika_11_136_pol/informatika_materialy_zanytii_11_136_pol_063.html
7. Собирай рюкзак по алгоритму, если будет NP=P. https://www.youtube.com/watch?v=aB_6ZsLjzzc
8. Генератор псевдослучайных чисел (часть 8) | Криптография | Программирование. https://www.youtube.com/watch?v=mVF8NDM-reg
9. CS50 на русском: Лекции (Гарвард. Основы программирования). https://www.youtube.com/watch?v=SW_UCzFO7X0&list=RDCMUCMcDsSeqS531-HKz6GiJgtA&start_radio=1&rv=SW_UCzFO7X0&t=5
10. Новости высоких технологий. Большие данные. https://hinews.mediasole.ru/trendy__bolshie_dannye
11. Алгоритмы и модели вычислений. Алгоритмы параллельных вычислений. НОУ ИНТУИТ: https://intuit.ru/studies/courses/533/389/lecture/9029
12. Полиномиальные и экспоненциальные алгоритмы. https://studref.com/550123/ekonomika/polinomialnye_eksponentsialnye_algoritmy
13. Полиномиальные алгоритмы и труднорешаемые задачи. https://studfile.net/preview/4056850/
14. Эффективные алгоритмы и сложность вычислений. https://discopal.ispras.ru/img_auth.php/archive/f/f4/20180307142838%21Book-advanced-algorithms.pdf
15. Полиномиальные и экспоненциальные алгоритмы. https://www.youtube.com/watch?v=vcoazkhUiZg
16. Опорные комбинаторные задачи. https://www.youtube.com/watch?v=zw8E6f6D9AY

Печатается в виде предложенным автором

Сдано в производство 10.09.2024 г. Подписано в печать 13.09.2024 г. Формат бумаги 60X84 1/8. Усл. печ. л. 9. N3647.

Издательский дом "Технический университет", Тбилиси, ул. М. Костава 77