

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ**



Уральский государственный экономический университет

Е. В. Кислицын

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Екатеринбург

2020

УДК 004.051:004.67
ББК 32.973.26
К 44

Кислицын Е.В. Алгоритмы и структуры данных. Екатеринбург, 2020

Учебное пособие разработано в соответствии с рабочей программой дисциплины «Алгоритмы и структуры данных», составленной на основе требований Федерального государственного образовательного стандарта высшего образования. В учебном пособии рассмотрены вопросы построения алгоритмов на основе рекурсии, жадных алгоритмов, алгоритмов поиска и сортировки. Рассматриваются основные положения построения классических структур данных: списков, деревьев, графов. Отдельная глава посвящена решению задач динамического программирования. В каждой главе, помимо объемного теоретического материала представлены примеры реализации алгоритмов и структур данных на языке программирования C#. По каждой главе приведен перечень контрольных вопросов и заданий для самостоятельного выполнения.

Пособие рекомендуется студентам всех форм обучения по направлениям подготовки бакалавриата 09.03.01 «Информатика и вычислительная техника», 09.03.03 «Прикладная информатика», 02.03.03 «Математическое обеспечение и администрирование информационных систем», изучающим курсы «Алгоритмы и структуры данных», «Программирование». Пособие также может быть интересно студентам, магистрантам и аспирантам других направлений подготовки, желающим повысить свой образовательный уровень в области процессов построения алгоритмов и использования сложных структур данных.

© Е.В.Кислицын, 2020

ОГЛАВЛЕНИЕ

Оглавление	3
Введение.....	6
Глава 1. Введение в алгоритмы и структуры данных	9
1.1. Алгоритмы	9
1.2 Исполнитель и инварианты.....	16
1.3 Структуры данных и абстракции	19
1.4 Рекурсия	26
Контрольные вопросы	36
Задания для самостоятельного выполнения.....	37
Глава 2. Жадные алгоритмы.....	41
2.1 Жадные алгоритмы в решении экстремальных задач.....	41
2.2 Алгоритм Хаффмана.....	55
2.3 Префиксное дерево	60
2.4 Строки	69
Контрольные вопросы	74
Задания для самостоятельного выполнения.....	75
Глава 3. Алгоритмы сортировки.....	78
3.1 Задача сортировки.....	78
3.2 Сортировки сравнением	79
3.3 Сортировки с линейным временем выполнения	97
3.4 Внешняя сортировка.....	102
Контрольные вопросы	110
Задания для самостоятельного выполнения.....	111
Глава 4. Списки и деревья.....	115
4.1 Структура данных «Список»	115
4.2 Структура данных «Дерево».....	123
4.3 Бинарная куча	129
Контрольные вопросы	141

Задания для самостоятельного выполнения.....	142
Глава 5. Алгоритмы поиска.....	146
5.1 Последовательный поиск	146
5.2 Бинарный поиск	149
5.3 Распределяющий поиск.....	152
5.4 Поиск по бинарному дереву	154
Контрольные вопросы	161
Задания для самостоятельного выполнения.....	162
Глава 6. Сбалансированные и специальные деревья в задачах поиска ...	166
6.1 Дисбаланс и повороты деревьев.....	166
6.2 Декартовы деревья	168
6.3 Сбалансированные деревья поиска	171
6.4 Внешний поиск и В-деревья	175
Контрольные вопросы	178
Задания для самостоятельного выполнения.....	179
Глава 7. Хеширование	183
7.1 Обобщенный быстрый поиск.....	183
7.2 Хеш-функции.....	185
7.3 Применение хеш-функций	188
7.4 Хеш-таблицы	192
Контрольные вопросы	203
Задания для самостоятельного выполнения.....	203
Глава 8. Динамическое программирование.....	208
8.1 Принцип оптимальности Беллмана.....	208
8.2 Рекурсивное решение задач динамического программирования ...	214
8.3 Восходящее решение задач динамического программирования	222
8.4 Многомерные задачи динамического программирования	229
Контрольные вопросы	235
Задания для самостоятельного выполнения.....	236
Глава 9. Графы.....	240
9.1 Структура данных «Граф»	240
9.2 Обход графов.....	245
9.3 Алгоритмы построения минимального остовного дерева	259

9.4 Поиск кратчайших путей на графах.....	271
Контрольные вопросы	280
Задания для самостоятельного выполнения.....	281
Библиографический список	285

ВВЕДЕНИЕ

Учебное пособие посвящено вопросам построения эффективных алгоритмов и использования линейных и нелинейных структур данных. Данное учебное пособие не претендует на оригинальность, при его создании использованы материалы из сторонних источников. Уникальность данного издания состоит в том, что здесь систематизированы сведения по дисциплине «Алгоритмы и структуры данных» в достаточно кратком виде, что позволяет освоить ее за незначительный промежуток времени. Для более глубокого ознакомления с дисциплиной необходимо ознакомиться с литературой из библиографического списка настоящего учебного пособия.

Учебное пособие состоит из девяти глав. Первая глава является вводной, в ней рассказывается о свойствах алгоритмов и их эффективности. Особое внимание уделяется рекурсивным алгоритмам. Вторая глава посвящена жадным алгоритмам. Правила построения жадных алгоритмов рассматриваются на конкретных примерах. В третьей главе рассказывается об основных алгоритмах сортировки: с использованием сравнений и свойств ключей множеств. Главы 4-7 посвящены алгоритмам поиска с использованием классических структур данных, таких как списков и деревьев. В четвертой главе повествуется о структурах данных «Список» и «Дерево». Пятая глава раскрывает основные классические алгоритмы поиска. Шестая глава посвящена правилам достижения баланса в деревьях. В седьмой главе речь идет об обобщенном быстром поиске и использовании хеш-функций. Восьмая глава раскрывает суть метода динамического программирования при решении задач. В качестве иллюстраций приведены классические задачи динамического программирования и два способа их решения – рекурсивный и восходящий. Девятая глава посвящена структуре данных «Графы». Рассматриваются задачи обхода графов, построения минимального остовного дерева и поиска

кратчайших путей на графах. Структура учебного пособия соответствует структуре рабочей программы дисциплины «Алгоритмы и структуры данных», читаемой в Уральском государственном экономическом университете.

В каждой главе представлен необходимый теоретический материал, а также примеры реализации алгоритмов на языке программирования C#. Также, в каждой главе предложены упражнения, которые студенты должны выполнить для успешного прохождения учебного курса. Упражнения приведены по тексту главы и направлены на закрепление материала. Выполнение упражнений допустимо на любом языке программирования, однако, автор рекомендует использовать языки C#, Java или C++. Кроме того, в конце каждой главы приведены контрольные вопросы, которые преподаватели могут использовать для проведения контрольных работ и коллоквиумов, а студенты – для подготовки к занятиям. Представленные в конце каждой главы задания предполагают их самостоятельное выполнение студентами и использование преподавателем для оценки полученных знаний, умений и навыков в области алгоритмов и структур данных. При выполнении этих заданий обязательно необходимо подготавливать наборы тестовых данных для демонстрации работы программы. Тестовые наборы должны удовлетворять условиям задачи, в том числе на крайних случаях.

В учебном пособии используются следующие обозначения:



– важное определение, теорема или высказывание, которое является основополагающим. Следует запомнить отмеченный текст, так как он будет использоваться во всем учебном пособии. Желательно завести словарь, в который будут записаны все определения и теоремы.



– упражнение. Здесь предлагается выполнить предложенное задание, используя любую IDE и язык программирования C# (или Java, C++). В рамках одной главы все упражнения следует выполнять в одном проекте, так как большинство заданий связано между собой. Важным моментом при выполнении упражнений является тестирование реализованных методов.

Обратите внимание, что реализуемые алгоритмы должны быть эффективны по времени и дополнительной памяти. Для измерения времени выполнения метода на языке C# следует использовать следующую конструкцию:

```
Stopwatch stopWatch = new Stopwatch();  
stopWatch.Start();  
// здесь пишется тестируемый метод  
stopWatch.Stop();  
Console.WriteLine("RunTime: " + stopWatch.Elapsed);
```

Для измерения объема затраченной дополнительной памяти в методе на языке C# можно использовать следующую конструкцию:

```
long before = GC.GetTotalMemory(false);  
// здесь пишется тестируемый метод  
long after = GC.GetTotalMemory(false);  
long consumedInMegabytes = (after - before) / (1024 * 1024);  
Console.WriteLine("Memory is {0} Mb.", consumedInMegabytes);
```

При выполнении упражнений и лабораторных работ необходимо обязательно делать такие измерения.

Автор желает успехов в изучении алгоритмов и структур данных!

ГЛАВА 1. ВВЕДЕНИЕ В АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

1.1. Алгоритмы

1.1.1 Понятие и свойства алгоритма

Данное учебное пособие посвящено алгоритмам и структурам данных. Эти понятия тесно связаны друг с другом. Невозможно создать хороший алгоритм, опираясь на неподходящие структуры данных.



Алгоритм – система формальных правил, четко и однозначно определяющая процесс решения поставленной задачи в виде конечной последовательности действий или операций.

Ниже представлены основные свойства алгоритма как последовательности команд для исполнителя:

1) *полезность (результативность)*, то есть умение решать поставленную задачу. Под результативностью понимается доступность результата решения задачи для пользователя;

2) *детерминированность (определенность)*, то есть каждый шаг алгоритма должен быть строго определен во всех возможных ситуациях. Это свойство означает, что неоднозначность толкования записи алгоритма недопустима;

3) *конечность (финитность)*, то есть способность алгоритма завершиться для любого множества входных данных. Алгоритм должен приводить к решению задачи обязательно за конечное время. Последовательность правил, приведшая к бесконечному циклу, алгоритмом не является;

4) *массовость*, то есть применимость алгоритма к разнообразным входным данным. Это означает, что если правильный результат по алгоритму получен для одних исходных данных, то правильный результат при

использовании этого же алгоритма должен быть получен и для других исходных данных, допустимых в данной задаче;

5) *эффективность*. Каждый алгоритм для своего выполнения (или вычисления) требует от исполнителя некоторых ресурсов. Программа есть запись алгоритма на формальном языке. Одну и ту же задачу зачастую можно решить несколькими способами, несколькими алгоритмами, которые могут отличаться использованием ресурсов, таких, как элементарные действия и элементарные объекты. Например, исполнитель алгоритма «компьютер» использует устройство центральный процессор для исполнения таких элементарных действий, как сложение, умножение, сравнение, переход и других, и устройство «память» как хранителя элементарных объектов целых и вещественных чисел. Способность алгоритма использовать ограниченное количество ресурсов называется эффективностью.

1.1.2 Сложность алгоритма

Выделяют три основных вида сложности алгоритма. Если требуется реализовать алгоритм в виде схемы вычислительного устройства, реализующего конкретную функцию, то *комбинационная сложность* определит минимальное число конструктивных элементов для реализации этого алгоритма. *Описательная сложность* есть длина описания алгоритма на некотором формальном языке. Один и тот же алгоритм на различных языках может иметь различную описательную сложность. *Вычислительная сложность*, определяет количество элементарных операций, исполняемых алгоритмом для каких-то входных данных. Для алгоритмов, не содержащих циклы, описательная сложность примерно коррелирует с вычислительной. Если алгоритмы содержат циклы, то такой корреляции нет и интерес представляет другая корреляция – времени вычисления от входных данных, причем обычно интересна именно асимптотика этой зависимости.

Введем понятие *главный параметр* N , наиболее сильно влияющий на скорость исполнения алгоритма. Это может быть размер массива при его обработке, количество символов в строке, количеством битов в записи числа, наконец. Если есть возможность выделить несколько таких параметров, то можно создать функцию от нескольких таких параметров, определяющую один обобщенный параметр. Для определения вычислительной сложности введена специальная нотация.



Функция $f(N)$ имеет порядок сложности $\Theta(g(N))$, если существуют постоянные c_1 , c_2 , и N_1 такие, что для всех $N > N_1$:

$$0 \leq c_1 g(N) \leq f(N) \leq c_2 g(N)$$

$\Theta(f(N))$ – класс функций, примерно пропорциональных $f(n)$.

На графике это выглядит таким образом: коэффициенты c_1 и c_2 , умноженные на функцию $g(n)$, приводят к тому, что график функции $f(n)$ оказывается зажат между графиками функций $c_1 g(N)$ и $c_2 g(N)$.

Например, если о каком-то алгоритме сказано, что он имеет сложность $\Theta(N^2)$, где-то при больших N функция сложности будет неотличима от функции cN^2 , где c — константа, которую называют *коэффициентом амортизации*.



Функция $f(N)$ имеет порядок $O(g(N))$, если существуют постоянные c_1 и N_1 такие, что для всех $N > N_1$

$$f(N) \leq c_1 g(N)$$

$O(f(n))$ — класс функций, ограниченных сверху $cf(n)$.

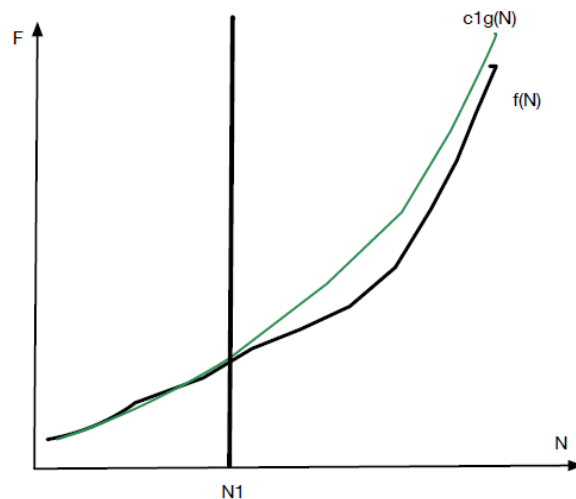


Рис. 1.1. График функции $f(N)$

Функция $g(N)$ имеет порядок $O(f(N))$, если существуют постоянные c_1 и N_1 такие, что

$$g(N) \leq c_1 f(N)$$

для всех $N > N_1$.

Пусть $F(N)$ — функция сложности алгоритма в зависимости от N . Тогда если существует такая функция $G(N)$ (асимптотическая функция) и константа C , что

$$\lim_{N \rightarrow \infty} \frac{F(N)}{G(N)} = C$$

то сложность алгоритма $F(N)$ определяется функцией $G(N)$ с коэффициентом амортизации C .

Говорят, что символ $\Theta(f(n))$ определяет класс функций, примерно пропорциональных $f(n)$, а символ $O(f(n))$ — класс функций, ограниченных сверху $cf(n)$.

Класс сложности определяется по асимптотической зависимости $F(N)$. Существует несколько правил:

- экспонента с любым коэффициентом превосходит любую степень;

- степень с любым коэффициентом, большим единицы, превосходит логарифм по любому основанию, большему единицы;
- логарифм по любому основанию, большему единицы, превосходит 1.

Можно привести несколько примеров:

$$F(N) = N^3 + 7N^2 - 14N = \Theta(N^3)$$

$$F(N) = 1.01^N + N^{10} = \Theta(1.01^N)$$

$$F(N) = N^{1.3} + 10 \log_2 N = \Theta(N^{1.3})$$

Однако, не стоит делать поспешные выводы о том, что алгоритм A_1 с асимптотической сложностью $\Theta(N^2)$ заведомо хуже алгоритма A_2 с асимптотической сложностью $\Theta(N \log N)$. Вполне может оказаться так, что при небольших значениях N количество операций, требуемых для исполнения алгоритма A_1 может оказаться меньше, чем для алгоритма A_2 .

Время исполнения алгоритма, исчисляемое в элементарных операциях, может отличаться для различных входных данных.

1.1.3 Задача поиска элемента в массиве

Пусть имеется массив A длиной N элементов. Сколько операций потребуется, чтобы обнаружить номер первого вхождения элемента со значением P алгоритмом, который заключается в последовательном просмотре элементов массива?

Самый первый элемент массива может оказаться P , следовательно, минимальное количество операций поиска будет $K_{min} = 1$. Элемента в массиве может не быть и для поиска потребуется ровно $K_{max} = N$ операций. А какое среднее значение количества поисков? Предполагая, что количество итераций алгоритма, требуемых для поиска, равномерно распределено по всем числам от 1 до N получаем:

$$K_{avg} = \frac{\sum_{i=1}^N i}{N} = \frac{N \times (N - 1)}{2N} = \frac{N - 1}{2}$$

Можно ли сказать, что алгоритм имеет сложность порядка $\Theta(N)$ в общем случае? Нет, в наилучшем случае $f(n) = 1$ не зависит от N совсем. Можно сказать, что в лучшем случае алгоритм имеет сложность $\Theta(1)$, в среднем и в худшем — сложность $\Theta(N)$. Но для данного алгоритма нам проще будет использовать O -нотацию: $f(N) = O(N)$.

Однако, не все задачи можно решить за полиномиальное время.

1.1.4 Задача о рюкзаке

Задача. Пусть имеется N предметов, каждый из которых имеет объём V_i и стоимость C_i , предметы неделимы. Имеется рюкзак вместимостью V . Требуется поместить в рюкзак набор предметов максимальной стоимости, суммарный объём которых не превышает объёма рюкзака.

Решение задачи. Как оказывается, задача не имеет решения с полиномиальной сложностью. Один из простых в реализации неполиномиальных алгоритмов заключается в следующем:

1. Перенумеруем все предметы
2. Установим максимум достигнутой стоимости в 0.
3. Составим двоичное число с N разрядами, в котором единица в разряде будет означать, что предмет выбран для укладки в рюкзак. Это число однозначно определяет расстановку предметов.
4. Рассмотрим все расстановки, начиная от 000 ... 000 до 111 ... 111, для каждой из них подсчитаем значение суммарного объёма.
 - a. Если суммарный объём расстановки не превосходит объёма рюкзака, то подсчитывается суммарная стоимость и сравнивается с достигнутым ранее максимумом стоимости.
 - b. Если вычисленная суммарная стоимость превосходит максимум, то максимум устанавливается в вычисленную стоимость и запоминается текущая конфигурация.

Алгоритм, как нетрудно убедиться, обладает всеми требуемыми свойствами: он детерминирован, так как его поведение зависит исключительно от входных данных; он конечен, так как его исполнение неизбежно прекратится, как только будут исчерпаны все расстановки; он массовый, так как он решает все задачи этого класса, и он полезный, так как даёт нам решение конкретной задачи.

Его сложность пропорциональна 2^N , так как требуется перебрать все возможные перестановки (не рассматривается тривиальный вариант, когда все N предметов помещаются в рюкзак).

Много ли времени потребуется на решение задачи для $N = 128$? Предположим, на подсчёт одного решения потребуется 10^{-9} секунд, то есть, одна наносекунда. Предположим, задачу будет решать триллион компьютеров (10^{12}). Тогда общее время решения задачи будет составлять:

$$2^{128} \times 10^{-9} / 10^{12} \text{ секунд} \approx 10.8 \times 10^9 \text{ лет}$$

Это пример задачи, которая имеет решение не полиномиальной сложности, но до сих пор не имеет решения полиномиальной сложности. Мало того, не доказано, что она может иметь решение полиномиальной сложности. Однако, проверить, удовлетворяет ли какое-либо предложенное решение условию корректности можно за полиномиальное время. Имеется понятие *сертификат* решения, который явным образом определяет предложенное решение. Например, в рассмотренной задаче о рюкзаке сертификатом может быть значение последовательности из N двоичных цифр. Точное решение подобных задач (а задача о рюкзаке относится к классу *NP-полных* задач) требует времени, превышающего все мыслимые значения. Необходимо понимать, что такие задачи существуют и что для них лучше искать *приближённое* решение, которое может оказаться не таким сложным.

1.2 Исполнитель и инварианты

1.2.1 Понятие исполнителя

В рамках данного учебного пособия в качестве исполнителя будет использоваться язык программирования C#. Этот язык, как и почти все остальные языки программирования, достаточно мощен для того, чтобы быть исполнителем любых алгоритмов.

Под элементарными типами данных будем понимать отображаемые на вычислительную систему типы, такие, как `char`, `int`, `double`. Под элементарными операциями аппаратного исполнителя будем понимать операции над элементарными типами и операции передачи управления. Типы данных языка есть комбинация элементарных типов данных. Операции языка есть комбинация элементарных операций.

Рассмотрим пример цикла `for` как неэлементарной операции языка:

```
int a[] = new int[10];
int s = 0;
for (int i = 0; ((i < 10) && (a[i] % 10 != 5)); i++)
{
    s += a[i];
}
```

Здесь имеется неэлементарный тип *массив*, представителем которого является `a` и элементарный тип `int`, представителем которого является `s`, элементарная операция присваивания (инициализации) `s = 0`, неэлементарная операция `for`, состоящая из операций присваивания `i = 0`, двух операций сравнения, и т. д.

Целые числа на современных компьютерах имеют двоичное представление и этот факт можно использовать для понижения сложности операций. Пока производителям компьютеров не удалось исполнять операции целочисленного деления и умножения так же быстро, как побитовые операции и операции сложения/вычитания. Операторы присвоения значения логической

операции быстрее, чем операторы сравнения. Операции сдвига битов быстрее, чем эквивалентные явные операции умножения и деления на степень двойки.

Побитовые операции можно иногда представить как параллельные операции над массивом битов. В ряде алгоритмов это оказывается полезным.

1.2.2 Модулярная арифметика

Задача. Пусть необходимо найти последнюю цифру значения 3^{7^8} .

Решение задачи. Заметим, что последние цифры степени тройки образуют период.

$$3^0 \pmod{10} = 1$$

$$3^1 \pmod{10} = 3$$

$$3^2 \pmod{10} = 9$$

$$3^3 \pmod{10} = 7$$

$$3^4 \pmod{10} = 1$$

$$3^5 \pmod{10} = 3$$

...

Последняя цифра определяется остатком от деления 7^8 на 4.

Аналогично:

$$7^0 \pmod{4} = 1$$

$$7^1 \pmod{4} = 3$$

$$7^2 \pmod{4} = 1$$

$$7^3 \pmod{4} = 3$$

...

$$7^8 \pmod{4} = 1 \rightarrow 3^{7^8} \pmod{10} = 3$$

Вся компьютерная арифметика основана на тождествах:

$$(a + b) \pmod{m} = (a \pmod{m} + b \pmod{m}) \pmod{m}$$

$$(a - b) \pmod{m} = (a \pmod{m} - b \pmod{m}) \pmod{m}$$

$$(a \times b) \pmod{m} = (a \pmod{m} \times b \pmod{m}) \pmod{m}$$

...

В качестве m при двоичном представлении выступают числа $2^8, 2^{16}, 2^{32}, 2^{64}$.



Создайте новый проект. Реализуйте в нем метод, в который на вход подается три числа: a , b и c . В качестве результата метод должен вернуть цифру, на которую оканчивается значение числа a^{b^c} . Протестируйте реализованный метод.

1.2.3 Индуктивные функции



Пусть имеется множество M . Пусть аргументами функции f будут последовательности элементов множества M , а значениями — элементы множества N . Тогда, если её значение на последовательности x_1, x_2, \dots, x_n можно восстановить по её значению на последовательности x_1, x_2, \dots, x_{n-1} и x_n , то такая функция называется *индуктивной*.

Если нужно найти наибольшее значение из всех элементов последовательности, то функция *maxituit* — индуктивна, так как

$$\text{maxituit}(x_1, x_2, \dots, x_n) = \text{max}(\text{maxituit}(x_1, x_2, \dots, x_{n-1}), x_n)$$

Значение функции для подмножества является *инвариантом*, то есть, предикатом, значение которого всегда истинно.

В алгоритме нахождения наибольшего элемента массива $a[N]$ неявно используются такие предикаты:

```
int m = a[0];
for (int i = 1; i < N; i++)
{
    if (a[i] > m)
    {
        m = a[i];
    }
}
```

Предикат гласит, что для любого $i < N$ переменная m содержит наибольшее значение из элементов $a[0] \dots a[i]$, то есть, m всегда равна значению уже рассмотренной индуктивной функции *maxituit*.

Рассмотрим простейший алгоритм нахождения суммы элементов массива:

```
// Вход: массив a[n]
// Выход: сумма его элементов
int sum = 0;
for (int i = 0; i < n; i++)
{
    sum += a[i];
}
```

Применяемая операция – использование индуктивной функции. Значение функции для подмножества является *инвариантом*, то есть, предикатом, значение которого всегда истинно. Инвариант: значение переменной *sum* в момент времени *i* есть сумма частичного массива от 0 до *i* включительно.

Инвариант является важнейшим понятием при доказательстве корректности алгоритмов. Путь доказательства корректности фрагмента алгоритма следующий:

1. выбираем предикат, значение которого истинно до начала исполнения фрагмента;
2. исполняем фрагмент, наблюдая за поведением предиката;
3. если после исполнения предикат остался истинным при любых путях прохождения фрагмента, алгоритм корректен относительно значения этого предиката.

1.3 Структуры данных и абстракции

1.3.1 Структуры данных: определение и классификация



Под *структурой данных* в общем случае понимают множество элементов данных и множество связей между ними. Такое определение охватывает все возможные подходы к структуризации данных, но в каждой конкретной задаче используются те или иные его аспекты. Поэтому вводится дополнительная классификация структур данных, направления которой соответствуют различным аспектам их рассмотрения.

Первая классификация предполагает деление структур данных на физические и логические. Понятие *физическая структура данных* отражает способ физического представления данных в памяти ЭВМ и называется еще структурой хранения, внутренней структурой или структурой памяти. Рассмотрение структуры данных без учета ее представления в памяти компьютера называется *логической структурой данных*. В общем случае между логической и соответствующей ей физической структурами существует различие, степень которого зависит от самой структуры и особенностей той среды, в которой она должна быть отражена.

Вторая классификация структур данных предполагает их деление на простые и интегрированные. *Простыми* называются такие структуры данных, которые не могут быть расчленены на составные части, большие, чем биты. *Интегрированными* называются такие структуры данных, составными частями которых являются другие структуры данных — простые или, в свою очередь, интегрированные.

Одним из основных критериев классификации структур данных является признак изменчивости. Так, структуры данных делятся на пять видов: базовые, статические, полустатические, динамические и файловые (рис. 1.2).

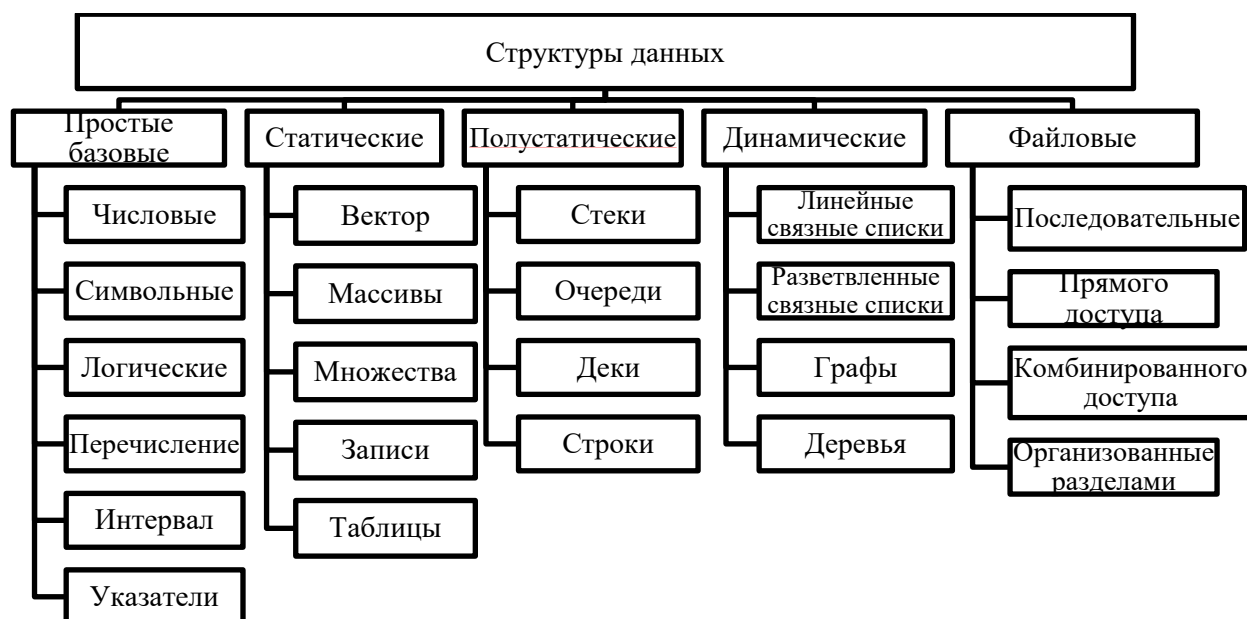


Рис. 1.2. Классификация структур данных по признаку изменчивости

По характеру упорядоченности элементов структуры данных, они делятся на две большие группы: линейные и нелинейные (рис. 1.3).

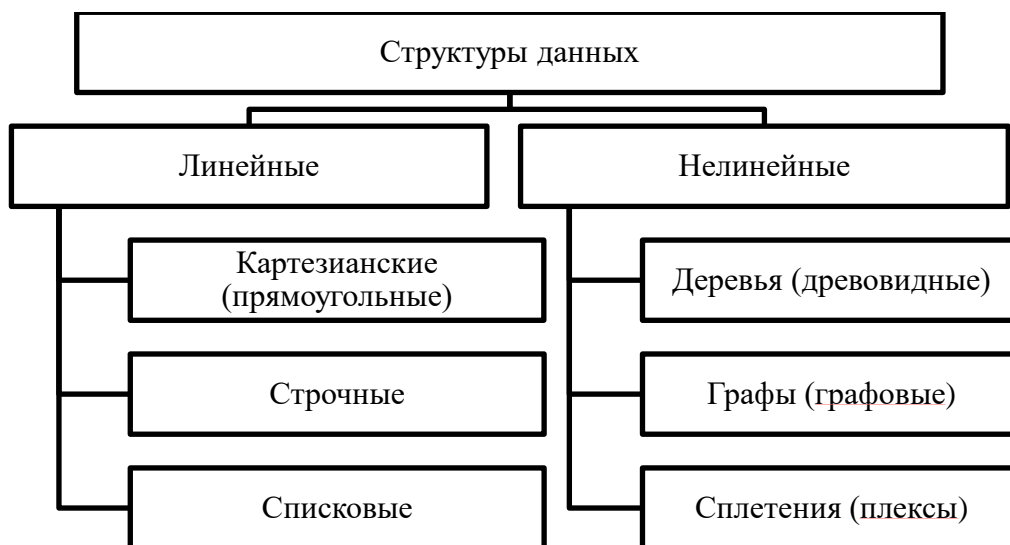


Рис. 1.3. Классификация структур данных по характеру упорядоченности элементов

Линейные структуры данных – это структуры, в которых связи между элементами не зависят от выполнения какого-либо условия. Линейные структуры подразделяются на три типа: картезианские, строчные и списковые.

Картезианские структуры названы так по способу записи данных в виде прямоугольных таблиц. К таким структурам данных относятся:

Вектор – структура данных с фиксированным числом элементов одного и того же типа.

Массив – последовательность элементов одного типа.

Множество – набор неповторяющихся элементов одного и того же типа.

Запись – конечное упорядоченное множество полей, характеризующихся различным типом данных.

Таблица – последовательность записей, имеющая одну и ту же организацию.

Строчные структуры – одномерные, динамически изменяемые структуры данных, различающиеся способами включения и исключения элементов. К строчным структурам данных относятся:

Стек – это последовательность, в которой включение и исключение элемента осуществляется с одной стороны последовательности;

Очередь – последовательность, в которую включают элементы с одной стороны, а исключают — с другой. Структура функционирует по принципу FIFO (первым пришел — первым обслуживается).

Дек – линейная структура (последовательность), в которой операции включения и исключения элементов могут выполняться как с одного, так и с другого конца последовательности.

В *списковых структурах* логический порядок данных определяется указателями. Любая списковая структура представляет собой набор элементов, каждый из которых состоит из двух полей: в одном из них размещен элемент данных или указатель на него, а в другом – указатель на следующий элемент списка.

Нелинейные структуры данных – это структуры данных, у которых связи между элементами зависят от выполнения определенного условия. К таким структурам данных относятся:

Древовидные структуры – это иерархические структуры, состоящие из набора вершин и ребер, каждая вершина содержит определенную информацию и ссылку на вершину нижнего уровня.

Граф – это сложная нелинейная многосвязная динамическая структура, отображающая свойства и связи сложного объекта. Графы представляют собой совокупность двух множеств: вершин и ребер.

Сплетение (многосвязный список, плекс) – это нелинейная структура данных, объединяющая такие понятия, как дерево, граф и списковая структура. Основное свойство сплетений, отличное от других типов структур, – наличие у каждого элемента сплетения нескольких полей с указателями на другие элементы того же сплетения.

1.3.2 Абстракция «Последовательность»

Как только появляются объекты, появляются абстракции – механизм разделения сложных объектов на более простые, без детализации подробностей разделения.

Функциональная абстракция – разделение функций, методов, которые манипулируют с объектами с их реализацией.

Интерфейс абстракции – набор методов, характерных для данной абстракции

Одной из первых абстракций, с которыми можно встретиться на начальном пути освоения алгоритмов – абстракция «последовательность элементов». Первые операции в любом языке, с которых начинается его изучение, это операции чтения с клавиатуры и записи на экран. В простых применениях этих операций нет возможности вернуться назад и пересчитать произвольное число, как и нет возможности вывести какое-то число, потом вернуться назад и вставить что-то перед ним.

Можно сказать, что эта абстракция реализует следующие операции:

1. Создать объект «последовательность». Операционная система обычно предоставляет такие объекты при старте программы – стандартные ввод и вывод. Также, можно создавать такие объекты самим (операция открыть файл), добавляя к ним атрибуты: объект предназначен для чтения или объект предназначен для записи.

2. Удалить объект «последовательность». Это может быть, например, операция закрыть файл.

3. Получить очередной элемент последовательности.

4. Добавить элемент в последовательность.

В классической последовательности уже полученный однажды элемент второй раз получить нельзя. Интересно, что для обработки информации в файлах, содержащих терабайты, иногда вполне хватает единственной операции

получения одного элемента. Алгоритмы, рассчитанные на обработку последовательностей, могут иметь сложность по памяти ($O(1)$) и по времени ($O(N)$).

1.3.3 Абстракция «Массив»

Вторая замечательная абстракция – массив. Она имеется практически во всех языках программирования и реализует следующие операции:

1. Создать массив.

```
int[] a = new [100];
```

2. Удалить массив.

3. Обратиться к элементу массива

```
int q1 = a[i];  
int q2 = b[i];  
int q3 = c[i];
```

Операции копирования обычных массивов, изменения их размеров требуют вызовов соответствующих функций или комбинации более мелких операций, например, цикла и доступа к элементам.

При реализации алгоритмов часто удобно абстрагироваться от того, каким именно образом представлен объект. Например, в случае массива основная операция – это доступ к элементу, и она выглядит одинаково для всех представлений.

1.3.4 Абстракция «Стек»

Ещё одна удобная абстракция — *стек*. Стек должен предоставлять следующие методы:

1. **create** — создание стека. Может быть, потребуется аргумент, определяющий максимальный размер стека;

2. **push** — занесение элемента в стек. Размер стека увеличивается на единицу. Занесённый элемент становится вершиной стека;

3. **pop** — извлечь элемент, являющийся вершиной стека и уменьшить размер стека на единицу. Если стек пуст, то значение операции не определено;

4. **peek** — получить значение элемента, находящегося на вершине стека, не изменяя стека. Если стек пуст, значение операции не определено;

5. **empty** — предикат истинен, когда стек пуст;

6. **destroy** — уничтожить стек.

Стек можно представить в программе в виде пары массив/указатель стека, либо с помощью создания класса:

```
public class Stack<T>
{
    List<T> list = new List<T>();
    public void Push(T value)
    {
        list.Add(value);
    }
    public int Pop()
    {
        if (list.Count == 0) throw new
InvalidOperationException();
        var result = list[list.Count - 1];
        list.RemoveAt(list.Count - 1);
        return result;
    }
}
```



Напишите класс `Stack`, реализующий абстракцию стека. Класс должен реализовывать все методы, описанные в параграфе. Протестируйте возможности класса `Stack`.

1.3.5 Абстракция «Множество»

Ещё очень важная абстракция — множество однотипных уникальных элементов, например, целых чисел. Между элементами должна быть определена операция сравнения на идентичность, чтобы эти элементы можно было различать.

Будем обозначать множество списком значений внутри фигурных скобок. Эта абстракция реализует следующие методы:

1. **insert** — добавление элемента в множество.

$\{1,2,3\}.insert(5) \rightarrow \{1,2,3,5\}$

$\{1,2,3\}.insert(2) \rightarrow \{1,2,3\}$

2. **remove** – удаление элемента из множества.

$\{1,2,3\}.\text{remove}(3) \rightarrow \{1,2\}$

$\{1,2,3\}.\text{remove}(5) \rightarrow \{1,2,3\}$ (или поведение не определено).

3. **in** – определение принадлежности множеству.

$\{1,2,3\}.\text{in}(2) \rightarrow \text{true}$

$\{1,2,3\}.\text{in}(5) \rightarrow \text{false}$

4. **size** — определение количества элементов в множестве.

1.4 Рекурсия

1.4.1 Принцип «Разделяй и властвуй»

Рассмотрим принцип «разделяй и властвуй» на примере вычисления чисел Фибоначчи. Сама последовательность чисел выглядит следующим образом: $\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots\}$. Получение чисел происходит по следующему правилу:

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

Такая способ записи последовательности называется рекуррентной. Часто последовательности задаются только таким способом. Данный алгоритм вычисления элементов последовательности Фибоначчи достаточно легко запрограммировать:

```
int fibo(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibo(n-1) + fibo(n-2);
}
```



Напишите метод `fibo` на языке C# и протестируйте его на разных значениях. Измерьте время работы алгоритма и затраченную память для значений n равных 5, 15, 50, 500, 5000, 500000.

Данный алгоритм является корректным, т.к. фактически реализовано определение числа Фибоначчи. Но, каково время его работы. Данный вопрос уже сложнее. Посмотрим на дерево вызовов этой функции для $N = 5$ (рис. 1.4).

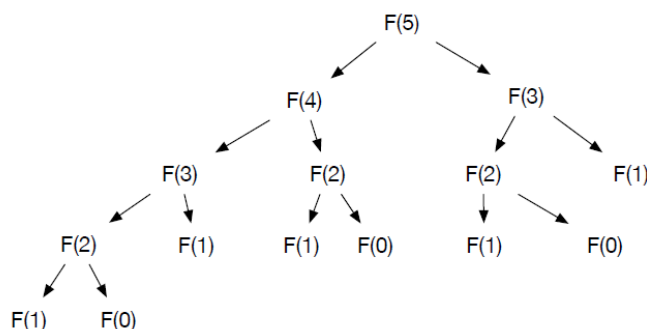


Рис. 1.4.

Попытаемся оценить количество вызовов этой функции. Количество вызовов функции для $n = 0$ и $n = 1$ равно одному. Обозначив количество вызовов через $t(n)$ получаем $t(0) = F(0)$ и $t(1) = F(1)$. Для $n > 1$ $t(n) = t(n - 1) + t(n - 2) > F(n)$. Следовательно, при рекурсивной реализации алгоритма количество вызовов превосходит число Фибоначчи для соответствующего n . Сами же числа Фибоначчи удовлетворяют отношению:

$$\lim_{n \rightarrow \infty} \frac{F_n}{F_{n-1}} = \Phi$$

где $\Phi = \frac{\sqrt{5}+1}{2}$, то есть, $F_n \approx C \times \Phi^n$. По нашей теории сложность этого алгоритма есть $\Theta(\Phi^n)$. Проблема в том, что в реализованной программе повторно вычисляется значение функции от одних и тех же аргументов. Требуемая память для исполнения характеризует *сложность алгоритма по памяти*.

- Каждый вызов функции создаёт новый *контекст функции* или *фрейм вызова*.
- Каждый *фрейм вызова* содержит все аргументы, *локальные переменные* и *служебную информацию*.
- Максимально создаётся количество фреймов, равное глубине рекурсии.

- Сложность алгоритма по занимаемой памяти равна $O(N)$

Можно ли ускорить такой алгоритм? Если ввести добавочный массив для сохранения предыдущих значений функции, уже вычисленные значения функции повторно вычисляться не будут:

```
int newFibo(int n)
{
    const int MAXN = 1000;
    static int c[MAXN];
    if (n == 0) return 0;
    if (n == 1) return 1;
    if (c[n] > 0) return c[n];
    return c[n] = fibo(n-1) + fibo(n-2);
}
```

Алгоритм остаётся рекурсивным, но дерево рекурсии становится вырожденным (рис. 1.5).

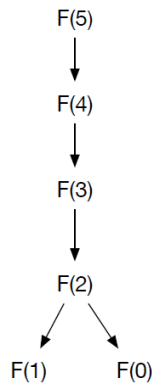


Рис. 1.5.



Напишите усовершенствованный метод `newFibo` и протестируйте его на тех же значениях, что и в предыдущем упражнении. Сравните полученные результаты. Насколько новый метод работает эффективнее?

Необходимо обратить внимание на серьёзную проблему: пока реализована только алгоритмическая часть решения заданной проблемы, то есть установлен верный порядок исполнения операций. При этом, остается другая проблема — значение функции растет слишком быстро и уже при небольших значениях n число выйдет за пределы разрядной сетки.

В реальных программах, исполняющихся на реальных компьютерах существуют ограничения на операнды машинных команд, которые формируются из исполняющейся операционной системы. Популярные 32-битные операционные системы стремительно уступают место 64-битным. В языке C# на архитектурах X86 и X64 стандартный тип данных `int` занимает ровно 32 бита. Чтобы воспользоваться 64-разрядной арифметикой достаточно использовать тип данных `long`. В дальнейшем будем полагать, что основной единицей данных для вычисления является `int` и будем оценивать сложность алгоритмов исходя из этого.

Когда речь идет о числах в алгоритмах, не учитывается тот факт, что при исполнении алгоритма на каком-либо исполнителе некоторые числа, получающиеся при вычислениях, оказываются не представимы на исполнителе. Необходимо ответить на вопрос: *что есть число в алгоритме?*

Значение функции `fibonacci(n)` растёт слишком быстро и уже при небольших значениях `n` число выйдет за пределы разрядной сетки любой архитектуры, то есть, станет непредставимо на исполнителе алгоритма. Поэтому, будем иметь в виду следующее: Алгоритм `fibonacci` оперирует с числами. Программа, реализующая алгоритм `fibonacci` имеет дело с *представлениями* чисел. Любой исполнитель алгоритма имеет дело не с числами, а с их представлениями.

1.4.2 Представление длинных чисел в алгоритмах

Оценим, насколько затратен такой переход, от абстрактных чисел к их представлениям. В реальных программах имеются ограничения на операнды машинных команд. X86, X64 → `int` есть 32 бита, `long` есть 64 бита.

На 32-битной архитектуре сложение двух 64-разрядных → сложение младших разрядов и прибавление бита переноса к сумме старших разрядов. Две или три машинных команды.

X86: сложение: 32-битных ≈ 1 такт; 64-битных ≈ 3 такта

X64: сложение: 32-битных ≈ 1 такт; 64-битных ≈ 1 такт.

X86: умножение: 32-битных ≈ 3 -4 такта; 64-битных ≈ 15 -50 тактов.

X64: умножение: 32-битных ≈ 3 -4 такта; 64-битных ≈ 4 -5 тактов.

Команды деления целых чисел и нахождения целочисленного остатка на современных компьютерах весьма долго исполняются. Будем их использовать только в тех случаях, когда без этого в алгоритме не обойтись.

Для решения проблемы представления всевозможных целых чисел для исполнителя, введём абстрактную структуру данных «*длинное число*», над которой определены все те же операции, что и над элементарно представимыми целыми числами – сложение, вычитание, умножение, деление, нахождение остатка от деления, сравнение.

Представлять такие числа можно многими способами, но для удобства вычислений будем использовать позиционную систему счисления с необычным основанием. Так как длинные числа имеют представление в виде цифр, представляющих удобный тип данных в позиционной системе счисления, то все операции и будут производиться в этой системе счисления. В стандартных задачах используется по одному знаку на десятичную цифру. Однако, аппаратному исполнителю удобнее работать с длинными числами в системе счисления по основаниям, большим 10 (2^8 , 2^{16} , 2^{32} , 2^{64}).



(n)-числа – те числа, которые требуют не более n элементов элементарных типов (цифр) в своём представлении.

Например, если за элементарный тип данных для представления на 32-битной архитектуре выберем тип `int`, то `long` будут (2)-числами.

Представление длинных чисел требует массивов элементарных типов. Основание системы счисления R для каждой из цифр представления должно быть представимо элементарным типом данных аппаратного исполнителя.

Сколько операций потребуется для сложения двух (n) -чисел? Воспользуемся школьным алгоритмом сложения в столбик (рис. 1.6).

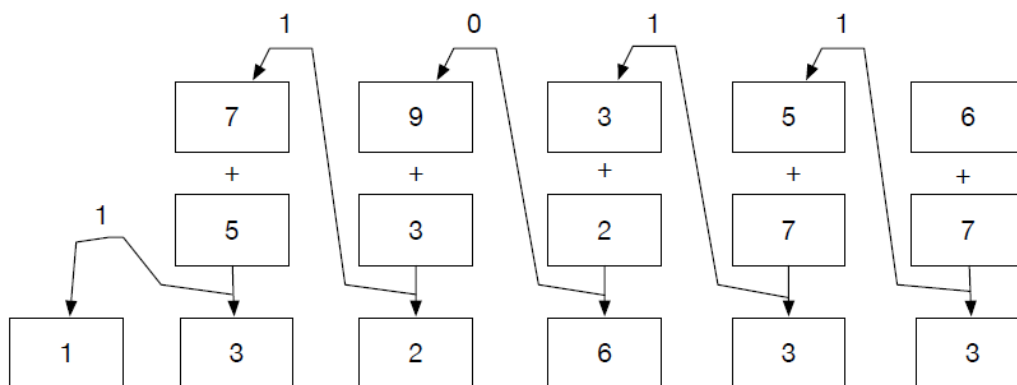


Рис. 1.6.

Каждую цифру первого числа нужно сложить с соответствующей цифрой второго и учесть перенос из соседнего разряда. Сложность алгоритма составляет $O(n)$.

Школьный алгоритм умножения длинных чисел тоже на первый взгляд кажется оптимальным: умножаем первое число на каждую из цифр второго, на что требуется n операций умножения, затем складываем все n промежуточных результатов, что даёт $O(n^2)$ операций умножения и $O(n^2)$ операций сложения (рис. 1.7).

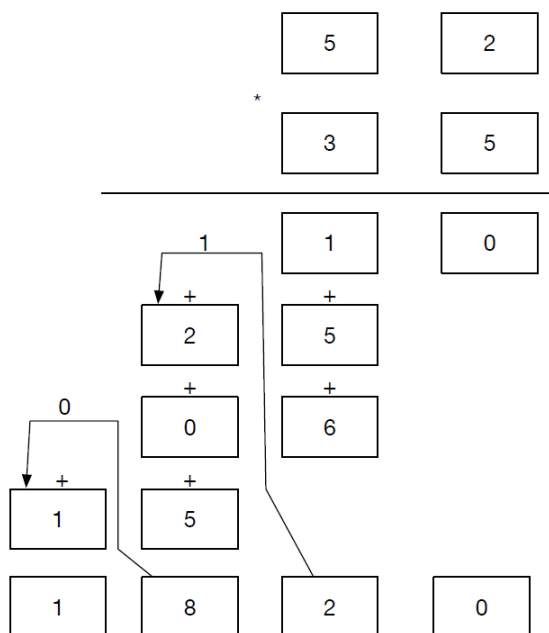


Рис. 1.7.

Можно ли разработать алгоритм, позволяющий перемножить два числа быстрее? Будем полагать, что существует операция умножения двух 32- битных чисел, дающая 64-битное число и это ровно одна операция.

1.4.3 Алгоритм быстрого умножения

Наивный алгоритм умножения длинных чисел (n) имеет сложность $O(N^2)$. К счастью, имеется более быстрый алгоритм. Он изобретён в 1960-х годах аспирантом А. Н. Колмогорова Анатолием Карацубой и с тех пор является неизменным участником любых библиотек работы с большими числами. Он реализует принцип Цезаря – *разделяй и властвуй*. Пусть требуется перемножить два $(2n)$ -числа. Введём константу T , на единицу большую максимального числа, представляемого (n) -числом. Тогда любое $(2n)$ -число X можно представить в виде суммы $T \times x_u + x_l$. Это разложение имеет сложность $O(n)$, так как оно заключается просто в копировании соответствующих разрядов чисел.

$$N_1 = T \times x_1 + y_1$$

$$N_2 = T \times x_2 + y_2$$

При умножении в столбик получается следующая формула:

$$N_1 \times N_2 = T^2 x_1 x_2 + T((x_1 y_2 + x_2 y_1) + y_1 y_2)$$

Это четыре операции умножения и три операции сложения. Число T определяет, сколько нулей нужно добавить к концу числа в соответствующей системе счисления и мы полагаем сложность этой операции равной $O(1)$.

Алгоритм Карацубы находит произведение по другой формуле:

$$N_1 \times N_2 = T^2 x_1 x_2 + T(x_1 + y_1)(x_2 + y_2) - x_1 x_2 - y_1 y_2 + y_1 y_2$$

Рассмотрим алгоритм на примере произведения чисел 56 и 78, приняв T за 10:

$$x_1 = 5, y_1 = 6, x_2 = 7, y_2 = 8.$$

$$x_1 x_2 = 5 \times 7 = 35.$$

$$(x_1 + y_1)(x_2 + y_2) = (5 + 6)(7 + 8) = 11 \times 15 = 165$$

$$y_1 y_2 = 6 \times 8 = 48$$

$$N_1 \times N_2 = 35 \times 100 + (165 - 35 - 48) \times 10 + 48 = 3500 + 920 + 48 = 4368.$$

Таким образом, число операций умножения уменьшено за счет увеличения операций сложения. Насколько это выгодно покажет основная теорема о рекурсии.



Реализуйте алгоритм быстрого умножения двух чисел в отдельном методе. Протестируйте данный метод. Сравните его с операцией стандартного умножения на больших числах.

1.4.4 Основная теорема о рекурсии

Как определить, какой порядок сложности будет иметь рекурсивная функция, не проводя вычислительных экспериментов? Так как рекурсия есть разбиение задачи на подзадачи с последующей консолидацией результата, обозначим количество подзадач, на которые разбивается задача за a , размер каждой подзадачи уменьшается в b раз и становится $\lceil n/b \rceil$.

Пусть сложность консолидации после решения подзадач есть $O(n^d)$. Тогда сложность такого алгоритма, выраженная рекуррентно, есть

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$$

Теорема о рекурсии выглядит следующим образом:



Пусть $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ для некоторых $a > 0$, $b > 1$, $d \geq 0$. Тогда

$$T(n) = \begin{cases} O(n^d), & \text{если } d > \log_b a \\ O(n^d \log n), & \text{если } d = \log_b a \\ O(n^{\log_b a}), & \text{если } d < \log_b a \end{cases}$$

Рассуждая неформально, можно заметить, что общая сложность алгоритма есть сумма членов геометрической прогрессии с знаменателем $q = \frac{a}{b^d}$. При $q < 1$ она сходится и её сумма оценивается через первый член, при $q > 1$

она расходится и её сумма оценивается через её последний член, а при $q = 1$ все члены (а их $O(\log n)$) равны.

Оценим сложность алгоритма Карацубы по этой теореме. Коэффициент a порождения задач здесь равен трём, так как одна первичная операция «большого» умножения требует трёх операций «маленького» умножения. Коэффициент уменьшения размера подзадачи $b = 2$, число делится на две примерно равные части. Консолидация решения производится за время $O(n)$, так как она заключается в операциях сложения и вычитания (которые имеют сложность $O(n)$ и их строго определённое количество), следовательно, $d = 1$. Так как $1 < \log_2 3$, то в действие вступает третий случай теоремы и, следовательно, сложность алгоритма есть $O(N^{3/2})$. Операция умножения чисел (n) при умножении в столбик имеет порядок сложности $O(n^2)$.

Наличие большого количества операций сложения и вычитания говорит о том, что для малых (n) алгоритм может исполняться дольше, чем «школьный» и поэтому в реальных программах рекурсию стоит ограничить. Например, в большинстве библиотек длинной арифметики алгоритм Карацубы используется при достаточно больших значениях n .

1.4.5 Алгоритм быстрого возведения в степень

При вычислении больших чисел Фибоначчи можно воспользоваться аппаратом линейной алгебры. Введем вектор-столбец $\begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$, состоящий из двух элементов последовательности Фибоначчи и умножим его справа на матрицу $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Получим:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_0 + F_1 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$$

Для вектора-столбца из элементов F_{n-1} и F_n умножение на ту же матрицу даст соответствующий результат. Таким образом, вычисление чисел Фибоначчи происходит следующим образом:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \times \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

Это означает, что для нахождения n -го числа Фибоначчи достаточно возвести матрицу $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ в n -ю степень.



В созданном проекте опишите новую структуру матрицы 2 на 2. Создайте новую матрицу согласно представленному материалу выше. Напишите метод вычисления числа Фибоначчи с помощью матрицы. Используйте стандартный метод возведения в степень. Сравните полученные результаты.

Отсюда возникает вопрос, можно ли возвести число в n -ю степень за число операций меньших $n-1$? Возведение числа в квадрат есть умножение числа на себя, отсюда можно предположить, что возвести число в степень можно в 2 раза быстрее. Например, возведение в 16-ю степень можно произвести не за 15 операций умножения, а за 4: $x^{16} = (x^8)^2 = ((x^4)^2)^2 = (((x^2)^2)^2)^2$. С показателями степеней, не равными степеням двойки вычисления будут немного сложнее. Например, возводя число в 18-ю степень получим: $x^{18} = (x^9)^2 = (x^8 \times x)^2 = (((x^2)^2)^2 \times x)^2$.

Таким образом, рекуррентная формула возведения числа в n -ю степень имеет следующий вид:

$$x^n = \begin{cases} 1, & \text{если } n = 0 \\ (x^{n/2})^2, & \text{если } n \neq 0 \wedge n \pmod{2} = 0 \\ x^{n-1} \times x, & \text{если } n \neq 0 \wedge n \pmod{2} \neq 0 \end{cases}$$

По предложенной рекуррентной формуле можно реализовать метод возведения числа в степень n на языке C#:

```
T power(T x, int n)
{
    if (n==0) return (T)1;
    if (n&1) return power(x, n-1);
    T y = power(x, n/2);
    return y*y;
}
```

В данном методе предполагается, что существует тип данных T , для которого определены все необходимые операции.

Оценим сложность данного алгоритма. Представим степень n в виде двоичного числа. Например, 25 в виде 11001. Тогда нечётная степень будет означать, что последний разряд в двоичном представлении степени есть единица и операция $n-1$ есть её обнуление. Чётная же степень будет означать, что последний разряд равен нулю и деление такого числа на два есть вычёркивание этого разряда. Каждую из единиц требуется уничтожить, не изменяя количества разрядов и каждый из разрядов требуется уничтожить, не изменяя количества единиц.



Напишите метод быстрого возведения в степень. Протестируйте метод отдельно, сравнивая со стандартным методом возведения в степень. Используйте новый метод для нахождения чисел Фибоначчи. Сравните полученные результаты.

Контрольные вопросы

1. Что такое алгоритм?
2. Назовите основные свойства алгоритма.
3. Какие виды сложности алгоритмы Вы знаете?
4. Чем отличается асимптотика Θ от O ?
5. Какие классы сложности алгоритма Вы можете назвать?
6. В чем заключается суть задачи о рюкзаке?
7. Что такое исполнитель?
8. Что такое индуктивная функция?
9. Каков путь доказательства корректности фрагмента алгоритма?
10. Что такое структура данных?
11. Какие структуры данных относятся к статическим?
12. Назовите полустатические структуры данных.
13. В чем отличие динамических структур данных?

14. Какие виды линейных структур данных Вы знаете?
15. Дайте определение и перечислите динамические структуры данных.
16. Какие методы реализует абстракция «Последовательность»?
17. В чем особенности абстракции «Массив»?
18. Какие методы реализует абстракция «Стек»?
19. Охарактеризуйте абстракцию «Множество».
20. В чем заключается суть принципа «разделяй и властвуй»?
21. Как улучшить алгоритм поиска числа Фибоначчи?
22. Что такое (n)-числа и для чего они используются?
23. Как работает алгоритм Карацубы?
24. Перескажите основную теорему о рекурсии.
25. Перескажите алгоритм быстрого возведения в степень.

Задания для самостоятельного выполнения

Задание №1. Симметрическая разность

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта.

На вход подается множество чисел в диапазоне от 1 до 20000, разделенных пробелом. Они образуют множество А. Затем идет разделитель – число 0 и на вход подается множество чисел В, разделенных пробелом, 0 – признак конца описания множества (во множество не входит). Необходимо вывести множество $A \Delta B$ – симметрическую разность множеств А и В в порядке возрастания элементов. В качестве разделителя используйте пробел. В случае, если множество пусто, вывести 0. Для вывода можно использовать любой алгоритм сортировки.

Формат входных данных:

1 2 3 4 5 0 1 7 5 8 0

Формат выходных данных:

2 3 4 7 8

Задание №2. Вычисление полинома

Ограничение по времени: 1 секунда. Ограничение по памяти: 16 мегабайт.

Вычисление полинома – необходимая операция для многих алгоритмов.

Нужно вычислить значение полинома

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x^1 + a_0$$

Так как число n может быть достаточно велико, требуется вычислить значение полинома по модулю M . Сделать это предлагается для нескольких значений аргумента.

Формат входных данных:

Первая строка файла содержит три числа – степень полинома $2 \leq N \leq 100000$, количество вычисляемых значений аргумента $1 \leq M \leq 10000$ и модуль $10 \leq \text{MOD} \leq 10^9$.

Следующие $N+1$ строк содержат значения коэффициентов полинома $0 \leq a_i \leq 10^9$

В очередных M строках содержатся значения аргументов $0 \leq x_i \leq 10^9$.

Формат выходных данных:

Выходной файл должен состоять из ровно M строк – значений данного полинома при заданных значениях аргументов по модулю MOD .

Задание №3. Две кучи

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайт.

Имеется $2 \leq N \leq 23$ камня с целочисленными весами W_1, W_2, \dots, W_N .

Требуется разложить их на две кучи таким образом, чтобы разница в весе куч была минимальной. Каждый камень должен принадлежать ровно одной куче.

Формат входных данных:

N

$W_1 W_2 W_3 \dots W_N$

Формат выходных данных:

Минимальная неотрицательная разница в весе куч

Задание №4. Два массива

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

Даны два упорядоченных по неубыванию массива. Требуется найти количество таких элементов, которые присутствуют в обоих массивах. Например, в массивах (0, 0, 1, 1, 2, 3) и (0, 1, 1, 2) имеется четыре общих элемента – (0, 1, 1, 2).

Первая строка содержит размеры массивов N_1 и N_2 . В следующих N_1 строках содержатся элементы первого массива, в следующих за ними N_2 строках – элементы второго массива.

Программа должна вывести ровно одно число – количество общих элементов.

Формат входных данных:

N_a, N_b

a_1

a_2

...

a_{N_a}

b_1

b_2

...

b_{N_b}

Формат выходных данных:

Одно целое число – количество общих элементов

Задание №5. Считаем комментарии

Ограничение по времени: 1 секунда. Ограничение по памяти: 256 мегабайт

Комментарием в языке Object Pascal является любой текст, находящийся между последовательностью символов, начинающих комментарий

определенного вида и последовательностью символов, заканчивающей комментарий этого вида.

Виды комментариев могут быть следующие:

1. Начинающиеся с набора символов (*) и заканчивающиеся набором символов *).
2. Начинающиеся с символа { и заканчивающиеся символом }.
3. Начинающиеся с набора символов // и заканчивающиеся символом новой строки.

Еще в языке Object Pascal имеются литеральные строки, начинающиеся с символа одиночной кавычки ' и заканчивающиеся этим же символом. В корректной программе строки не могут содержать символа перехода на новую строку.

Будьте внимательны, в задаче используются только символы с кодами до 128, то есть, кодировка ASCII. Код одиночной кавычки – 39, двойной – 34.

Формат входных данных:

На вход программы подается набор строк, содержащих фрагмент корректной программы на языке Object Pascal.

Формат выходных данных:

Выходом программы должно быть 4 числа – количество комментариев первого, второго и третьего типов, а также количество литеральных строк.

ГЛАВА 2. ЖАДНЫЕ АЛГОРИТМЫ

2.1 Жадные алгоритмы в решении экстремальных задач

2.1.1 Жадные алгоритмы: основные положения

Очень большое число решаемых с помощью компьютеров задач связано с оптимальным нахождением чего-либо. Это может быть наилучший путь путешественника, желающего посетить определённое число городов и потратить наименьшую сумму денег. Это может быть определение оптимального режима работы светофора, для того, чтобы пропускать наибольшее количество транспортных средств за одно и то же время. Задача о рюкзаке относится к той же области.

Задачи нахождения оптимального значения, максимального или минимального, носят название *экстремальных*. Общий процесс решения таких задач называется оптимизацией. Методы оптимизации – отдельная, до сих пор развивающаяся область. Некоторые задачи можно решить по известным алгоритмам, но для некоторого класса задач возможно подобрать лишь приемлемое решение.



Жадные алгоритмы состоят из итераций и принимают решение на каждом шаге, стараясь найти локально оптимальное решение.

Как пример типичного жадного алгоритма можно привести минимизацию значения функции методом, похожим на метод покомпонентного спуска. Предположим, что имеется непрерывная функция n переменных $f(x_1, x_2, \dots, x_n)$, принимающая действительные значения на области определения. Она определяет поверхность в n -мерном пространстве. Один из простейших алгоритмов минимизации такой функции можно описать следующим образом:

1. выберем начальную точку (x_1, x_2, \dots, x_n) , которая станет текущей точкой алгоритма;

2. обследуя точки вокруг текущей найдем такую, в которой $f(x_1', x_2', \dots, x_n')$ имеет минимальное значение;

3. если найденная точка отлична от текущей, то сделаем его текущей и перейдем ко второму шагу алгоритма;

4. конец алгоритма.

Результаты этого алгоритма хорошо видны при попытке оптимизировать с его помощью одномерную функцию. Если функция унимодальна, то есть имеет единственный оптимум, то алгоритм приводит к решению. Но для функций, имеющих несколько оптимумов, он может привести к неверным результатам.

Код программы, реализующей предложенный алгоритм, на языке C# представлен ниже:

```
static double f(double x, double y)
{
    return (x - 3) * (x - 3) + (y + 2) * (y + 2);
}
public static void Main(String[] args)
{
    double x0 = 0.0, y0 = 0.0, d = 0.1;
    double[] dx = new double[] { d, -d, 0, 0 };
    double[] dy = new double[] { 0, 0, d, -d };
    double newx = 0, newy = 0;
    bool bestfound = false;
    double maxf = f(x0, y0);
    while (!bestfound)
    {
        bestfound = true;
        {
            for (int i = 0; i < 4; i++)
            {
                double newf = f(x0 + dx[i], y0 + dy[i]);
                if (newf < maxf)
                {
                    maxf = newf;
                    bestfound = false;
                    newx = x0 + dx[i];
                    newy = y0 + dy[i];
                }
            }
            if (!bestfound)
            {
                x0 = newx;
```

```

        y0 = newy;
    }
}
Console.WriteLine("Best f({0},{1})={2:0.000}", x0,
y0, maxf);
}
}

```

Стоит обратить внимание, что в предложенном варианте для нахождения точек-кандидатов используются массивы dx и dy , которые содержат приращения координат для четырех возможных направлений попыток. Такая реализация позволяет вычислять координаты внутри цикла.

В качестве примера, сделаем попытку найти с помощью описанного алгоритма минимум функции от двух переменных:

$$f(x, y) = (x - 3)^2 + (y + 2)^2$$

В качестве начальной точки примем точку ($x_0 = 0, y_0 = 0$). Установим шаг поиска равный 0.1. Простейший вариант нахождения минимума «осматривает» окрестности текущей точки в направлении осей координат. Так, значения функции в текущей точке и окрестных равно:

$$f(0, 0) = 3^2 + 2^2 = 13$$

$$f(0 + 0.1, 0) = 2.9^2 + 2^2 = 8.41 + 4 = 12.41$$

$$f(0 - 0.1, 0) = 3.1^2 + 2^2 = 9.61 + 4 = 13.61$$

$$f(0, 0 + 0.1) = 9 + 4.41 = 13.41$$

$$f(0, 0 - 0.1) = 9 + 3.61 = 12.61$$

Так как значение функции в точке (0.1, 0) меньше, чем в остальных кандидатах и меньше, чем в текущей точке, она становится текущей.

На следующем шаге выбираем уже точки из окружения текущей – это точки (0.2, 0), (0,0), (0.1, 0.1) и (0.1, -0.1). Сравнение значений функций в этих точках приводит к следующей точке (0.2, 0). Продолжая выполнять описанные действия, будет найдено верное решение – (3, 2).

Таким образом, можно сделать вывод, что предложенный алгоритм выдает правильное решение. Но действительно ли он всегда выдаст правильное решение? Возьмем другую функцию:

$$f_2(x, y) = (x - 3)^2 + 10\sin x + (y + 2)^2$$

Начиная движение по предложенному алгоритму от точки (0, 0) будет получено решение в точке (-0.7, 2) со значением функции в ней равной 7.24. Но минимум данной функции достигается в другой точке (4.4, -2), где ее значение равно -7.6.

Проблема использования такого алгоритма заключается в том, что первая функция имеет один глобальный минимум и не имеет локальных, тогда как вторая функция имеет не только глобальный, но и локальные минимумы.

Алгоритм сошелся, если бы была взята начальная точка вблизи глобального минимума. Говорят, что данный алгоритм склонен к нахождению локальных экстремумов.



Создайте новый проект. Напишите метод поиска минимума функции по разобранному алгоритму. Протестируйте метод на разных значениях функции. Подумайте над тем, как решить проблему поиска локальных минимумов функции. Попробуйте усовершенствовать метод для функции от нескольких переменных (от произвольного количества элементов). Насколько зависит время выполнения данного метода от размерности функции?

2.1.2 Задача об интервалах

Задача. На прямой дано множество отрезков. Необходимо найти максимальный размер множества непересекающихся отрезков, расположенных на одной прямой.

Альтернативная формулировка задачи: имеется аудитория, на которую существует несколько претендентов. Каждый из претендентов заявляет время, в которое он может эту аудиторию занять и время, в которое он эту аудиторию может освободить. Администрация здания, в котором находится аудитория, должна решить задачу: каким претендентам предоставить аудиторию, а каким отказать, обеспечив себе максимальный доход (считается, что каждый из

претендентов платит одну и ту же сумму за каждое использование аудитории, независимо от времени аренды).

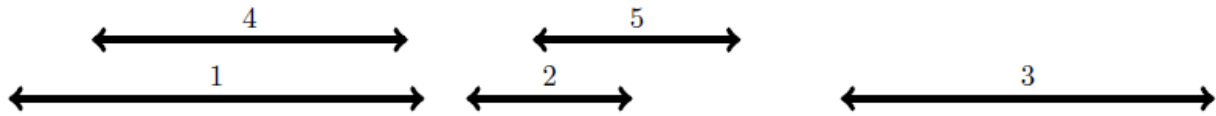


Рисунок 2.1. Исходная расстановка задачи об интервалах (Пример 1)

Решение задачи. Решим данную задачу жадным алгоритмом, выбрав какую-либо стратегию для принятия решения на каждом шаге.

Все варианты жадной стратегии основываются на следующем:

- упорядочить отрезки по какому-либо признаку;
- рассматривать отрезки по одному. Если он не перекрывается с каким-либо из уже внесённым в выходное множество, то добавить его в это выходное множество

Жадность алгоритма здесь заключается в том, что каждый раз, когда рассматривается очередной отрезок, то он сразу добавляется в результирующее множество. Принципов упорядочивания можно выбрать несколько, но, как оказывается, не все из них одинаково полезны.

Жадность алгоритма здесь заключается в том, что каждый раз, когда рассматривается очередной отрезок, то он сразу добавляется в результирующее множество. Принципов упорядочивания можно выбрать несколько, но, как оказывается, не все из них одинаково полезны.

1. Первый вариант – выбирать претендентов с самым коротким временем аренды. Упорядочим отрезки по длительности и выберем сначала самые короткие (рис. 2.2):

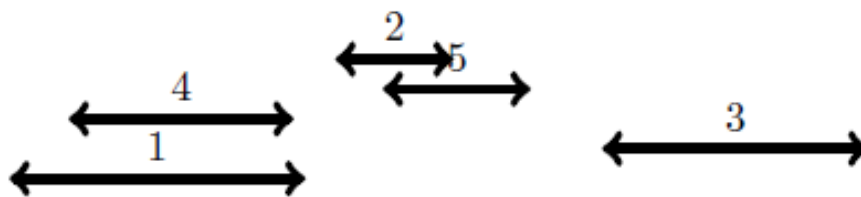


Рис. 2.2.

Такая расстановка выдаст нам ответ в виде трех отрезков: 4, 2 и 3 (рис. 2.3).

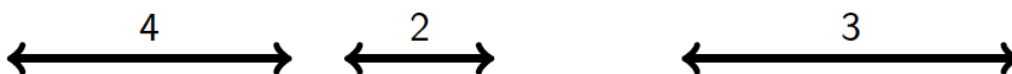


Рис. 2.3.

2. Вторая стратегия – отдавать в аренду аудиторию претенденту, предложившему самое раннее время. Если сформулировать эту стратегию в терминах отрезков, то выберем упорядочивание отрезков по левой границе и выбор самого левого из них. Получим результат, представленный на рис. 2.4.

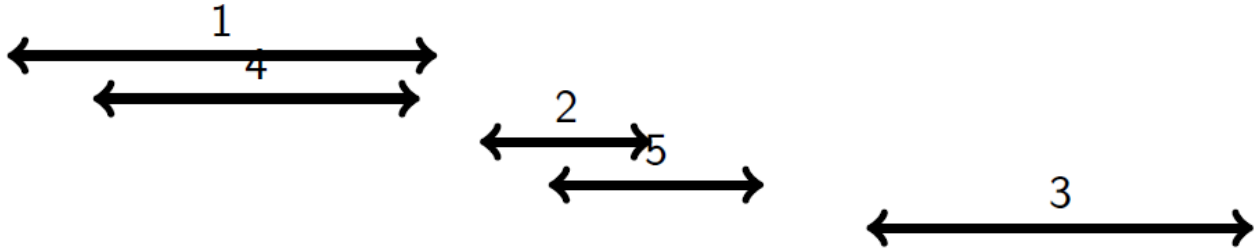


Рис. 2.4.

Ответ представлен на рис. 2.5.

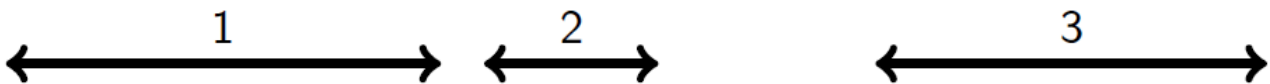


Рис. 2.5.

3. Третья стратегия решения задачи – выбирать тех претендентов, которые раньше всего освободят аудиторию. Другими словами, упорядочим отрезки по правой границе. Получим следующий результат (рис. 2.6).

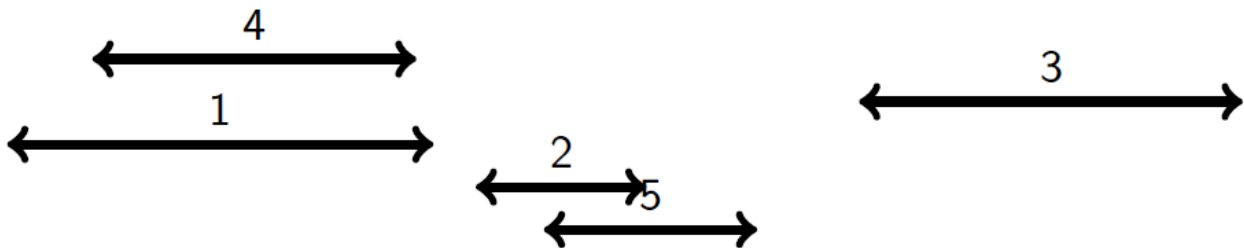


Рис. 2.6.

Тогда, в результате получим вариант расстановки, представленный на рис. 2.7.



Рис. 2.7.

При решении задачи с предложенными исходными данными ответ получается одинаковый, значит все три стратегии являются верными. Однако, не при всех исходных данных получится правильный результат. Например, задача об интервалах с исходными данными, представленными на рис. 2.8 будет верно решена только с применением второй или третьей стратегии (рис. 2.9), тогда как первая стратегия даст абсолютно неверный результат (рис. 2.10).

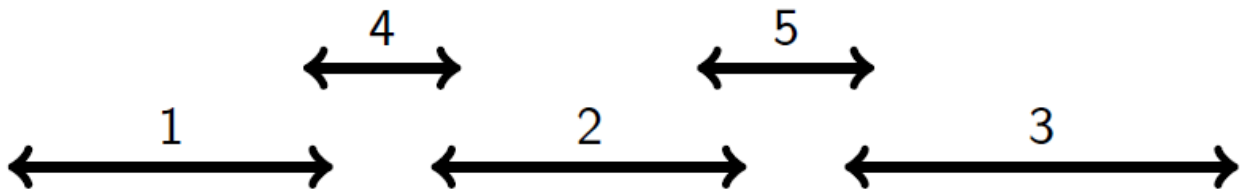


Рис. 2.8. Исходная расстановка задачи об интервалах (Пример 2)

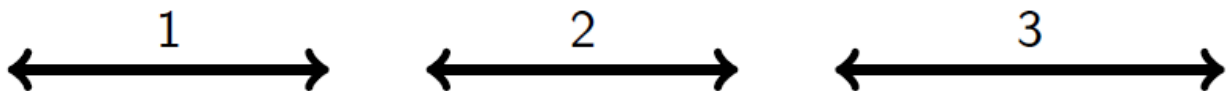


Рис. 2.9. Результат задачи об интервалах по второй и третьей стратегии (Пример 2)



Рис. 2.10. Результат задачи об интервалах по первой стратегии (Пример 2)

Можно сказать, что, выбирая стратегию для жадного алгоритма, устраивается соревнование между алгоритмами-претендентами. На каждой из проверок какой-то алгоритм может показать худшие результаты, тогда в следующем «раунде» соревнования он принимать участие не будет. На данном этапе можно сделать вывод, что первый вариант не даёт нам точного решения во всех случаях и выбывает именно он.

Чтобы такое состязание стратегий когда-то закончилось, необходимо найти такие исходные данные, которые могут опровергнуть какой-либо из вариантов. Можно сказать, что, придумывая стратегии для жадных алгоритмов, придумываются гипотезы, обосновывающие какую-то теорию.

При поиске ситуаций, когда нужно найти опровергающий какую-либо гипотезу вариант, можно наткнуться на следующее расположение отрезков (рис. 2.11):

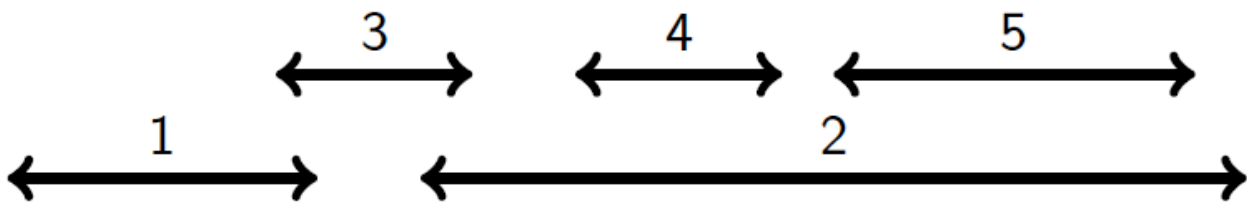


Рис. 2.11.

Упорядочение по началу отрезка даст неверное решение (рис. 2.12), тогда как отсортировав по концу отрезка получим верное (рис. 2.13).



Рис. 2.12.

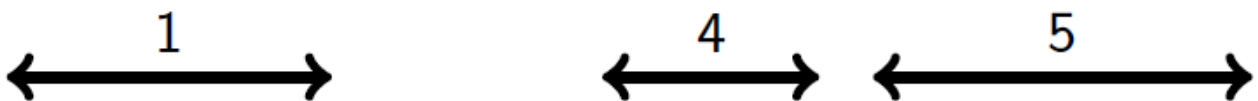


Рис. 2.13.

Предложенное решение – устроить соревнование между стратегиями – к сожалению, доказательной силой не обладает. Можно только показать, что какой-то алгоритм на каком-то наборе тестов ведёт себя лучше, чем другие. Корректность стратегии такой метод доказать не может. Доказательство можно проводить по индукции. Сначала убедимся в том, что оптимальных решений для данной задачи может быть несколько. Это легко проверить.

Первый шаг – доказательство того, что существует оптимальное подмножество отрезков, которое содержит первый отрезок, получившийся при применении нашего алгоритма. Будем рассуждать так: пусть в некотором оптимальном подмножестве поменяем отрезок с минимальным значением конца на первый, выбранный на данном шаге алгоритма. Количество отрезков в

выходном подмножестве не изменится и подмножество останется решением. Это основано на разумном предположении, что функция $f(x)$ оптимального решения на диапазоне от $[x; +\infty)$ невозрастающая. Таким образом, существует оптимальное подмножество, содержащее первый отрезок.

Второй шаг – удаляем из множества отрезков первый отрезок и все отрезки, пересекающиеся с первым.

Третий шаг – повторяем алгоритм для усечённого множества, в котором снова находим первый отрезок.

Применив метод математической индукции, показано, что предложенный жадный алгоритм приводит к одному из оптимальных решений задачи.



Жадные алгоритмы не заглядывают вперёд. Они повторяют локально оптимальные по какому-либо критерию шаги и надеются, что решение будет глобально оптимальным. Возможно, что найдётся такой локально оптимальный критерий и общее решение окажется верным. Это бывает отнюдь не всегда, но тщательный выбор критерия может найти приемлемое решение.



Реализуйте в созданном проекте метод, решающий задачу об интервалах разными стратегиями. В качестве входных параметров должны быть указаны точки начала и конца отрезка, а также выбранная стратегия. В качестве результата – номера отрезков и полученная прибыль. Протестируйте написанный метод.

2.1.3 Задача о резервных копиях

Задача. Имеется распределённая система, состоящая из N хранилищ различной ёмкости, причём в i -м хранилище можно разместить A_i блоков информации. Хранение одного блока считается надёжным, если имеется две его копии в различных хранилищах. Требуется определить наибольшее количество надёжных блоков, которое можно разместить во всех хранилищах.

Для примера возьмём 4 хранилища размерами (8, 7, 4, 3). На рис. 2.14 одинаковыми числами отмечены одинаковые блоки. Число (номер блока) не

может находиться дважды в одном столбце (хранилище). Нетрудно убедиться, что приведённое решение — оптимально, так как удалось распределить все блоки, в каждом из хранилищ размещена его максимальная ёмкость.

8	7	4	3
1	1	2	2
3	3	4	4
5	5	6	6
7	7	-	-
8	8	-	-
9	9	-	-
10	10	-	-
11	-	11	-

Рис. 2.14.

Решение задачи. Попробуем решить задачу жадным алгоритмом, выбрав правильную стратегию.

Для удобства сведём задачу к эквивалентной: имеется N кучек камней. Каждым ходом можно выбрать по одинаковому количеству камней из любой пары кучек. Найти такой порядок игры, при котором останется минимальное количество камней.

Первая стратегия: выбрать две наименьших кучи и взять из них одинаковое наибольшее количество камней. Для приведённой расстановки выберем две кучи, 4 и 3 и возьмем из них по 3 камня. Затем, из двух наименьших куч, 7 и 1, возьмем по одному камню. В результате с двумя оставшимися кучами, по 8 и 6 камней, уже ничего сделать нельзя и получившееся решение оптимальным не является (рис. 2.15).

8	7	4	3
8	7	1	0
8	6	0	0
2	0	0	0

Рис. 2.15.

Вторая стратегия: выбрать наибольшую и наименьшую кучи и взять из них одинаковое наибольшее количество камней (рис. 2.16). Такая стратегия также не приводит к оптимальному решению.

8	7	4	3
5	7	4	0
5	3	0	0
2	0	0	0

Рис. 2.16.

Третья стратегия: выбрать две наибольших кучи и взять из них одинаковое наибольшее количество камней (рис. 2.17). Такое решение уже является оптимальным.

8	7	4	3
0	1	4	3
0	1	1	0
0	0	0	0

Рис. 2.16.

Как обычно в задачах на жадность требуется или доказательство корректности стратегии, или контрпример. Рассмотрим пример, представленный на рис. 2.17:

8	7	7	6	5
0	1	7	6	5
0	1	1	0	5
0	0	1	0	4
0	0	0	0	3

Рис. 2.17.

Для выбранных исходных данных третья стратегия не дает верного результата. Массовые неудачи могут подсказать, что, стоило бы так уменьшать количество камней в кучах, чтобы эти числа были одного порядка.

Четвёртая стратегия: выбрать наибольшую и наименьшую кучи, и взять из них по одному камню. При такой стратегии и в первом (рис. 2.18) и во втором (рис. 2.19) примере получим корректный оптимальный результат.

8	7	4	3
7	7	4	2
7	6	4	1
6	6	4	0
5	6	3	0
5	5	2	0
4	5	1	0
4	4	0	0
0	0	0	0

Рис. 2.18.

8	7	7	6	5
7	7	7	6	4
7	7	6	6	3
7	6	6	6	2
6	6	6	6	1
6	6	6	5	0
6	6	5	4	0
6	5	5	3	0
5	5	5	2	0
5	5	4	1	0
5	4	4	0	0
4	4	3	0	0
4	3	2	0	0
3	3	1	0	0
3	2	0	0	0
1	0	0	0	0

Рис. 2.19.

Создаётся впечатление, что найден если не оптимальный алгоритм, то близкий к нему. Давайте докажем, что найденный алгоритм оптимален:

- Очевидно, что на каждой «большой» итерации наименьшая из куч опустошается, так как из неё по условиям алгоритма всегда производится взятие.
- Когда останется три кучи $A \geq B \geq C$, возможны следующие ситуации:
 - $A > B + C$. Тогда ответ: $A - B - C$. Так как на каждом ходе A оставалось наибольшим, на каждом их ходов, приведшем к позиции происходило вычитание из A , это значит, что A больше суммы всех оставшихся, что, очевидно, даёт верное решение.
 - $A = B + C$. Тогда результат равен нулю.

- $A < B + C$. Тогда, после некоторой, возможно нулевой последовательности ходов достигается ситуация, когда $A' = B' > C'$, которая сведётся к позиции $A' - \lfloor C'/2 \rfloor, A' - \lfloor C'/2 \rfloor$. В зависимости от чётности суммы результат будет равен либо нулю, либо единице.

Таким образом показано, что предложенная стратегия всегда приводит к оптимальному решению.



Реализуйте в созданном проекте метод, решающий задачу о резервных копиях. Протестируйте написанный метод.

2.1.4 Задача о рюкзаке: жадное решение

Вернёмся к задаче о рюкзаке из первой главы. Как известно, одно из точных решений имеет сложность $O(2^N)$. Можно попробовать найти приближённое решение задачи, используя жадный алгоритм. Формализуем условия задачи.

Задача. Пусть имеется N предметов, стоимость i -го предмета v_i , а масса w_i . Найти набор предметов с наибольшей стоимостью и массой, не превосходящей заданного W .

Решение задачи. Попробуем применить следующий локально оптимальный алгоритм:

1. Расположим предметы в порядке убывания отношения v_i/w_i . Пусть они образуют упорядоченное множество B .
2. Установим оставшийся вес $L = W$
3. Установим множество $S = \emptyset$.
4. Выбираем первый предмет I из упорядоченного множества, вес w_I которого не превосходит L .
5. Если такого предмета нет, то алгоритм закончен.
6. Кладём предмет в рюкзак, удаляя его из B : $B \leftarrow B - I$; $L \leftarrow L - w_i$; $S \leftarrow SI$. Переходим к 4-му шагу.

Данный алгоритм конечен и обязательно приведёт к какому-либо решению. Но решение, очевидно, может не быть оптимальным.

Например, при $N = 3$, $W = 40$, $w_1 = 10$, $v_1 = 60$, $w_2 = 20$, $v_2 = 100$, $w_3 = 20$, $v_3 = 100$ алгоритм выберет последовательно первый и второй предметы. Их суммарная стоимость окажется 160. Верное решение – выбрать второй и третий предметы. Их суммарная стоимость будет 200.



Реализуйте в созданном проекте метод, который решает задачу о рюкзаке. Протестируйте данный метод на больших значениях. Проверьте правильность решения.

2.2 Алгоритм Хаффмана

2.2.1 Префиксный код и двоичный деревья

Задача. Имеется исходный текст, состоящий из символов. Поставить в соответствие каждому символу такую последовательность битов (закодировать его таким образом), чтобы:

- каждый из встречающихся символов получил свой двоичный код;
- множество кодов было префиксным;
- после кодирования исходного текста суммарная длина получившейся последовательности была бы минимальной из возможных.

Например, пусть имеется текст, состоящий из множества из четырёх символов:

AAAABAABABABABCSBAAAD

Его длина — 21 символ.

Один из возможных кодов — равномерный код, такой, как ($A \rightarrow 00$), ($B \rightarrow 01$), ($C \rightarrow 10$), ($D \rightarrow 11$), когда каждый символ кодируется одинаковым количеством бит. В приведённом коде на кодирование каждого символа понадобится ровно два бита и общая длина кода составит 42 бита.



Префиксным кодом для набора символов называется код, в котором никакой код символа не начинается с другого кода.

В частности, код $(A \rightarrow 00), (B \rightarrow 10), (C \rightarrow 01), (D \rightarrow 101)$ префиксным не является, так как код символа D начинается с кода символа B. У префиксных кодов есть важное свойство: они допускают однозначное декодирование. Ещё одно название — код, удовлетворяющий условиям Фано. Наша задача — найти минимальный префиксный код для множества символов.

Ещё раз рассмотрим равномерный код из четырёх символов $(A \rightarrow 00), (B \rightarrow 01), (C \rightarrow 10), (D \rightarrow 11)$. Попробуем представить его в виде двоичного дерева. Для определённости пусть левая ветвь дерева будет представлять нулевой бит, правая — единичный. Древесное представление кода представлено на рис. 2.20.

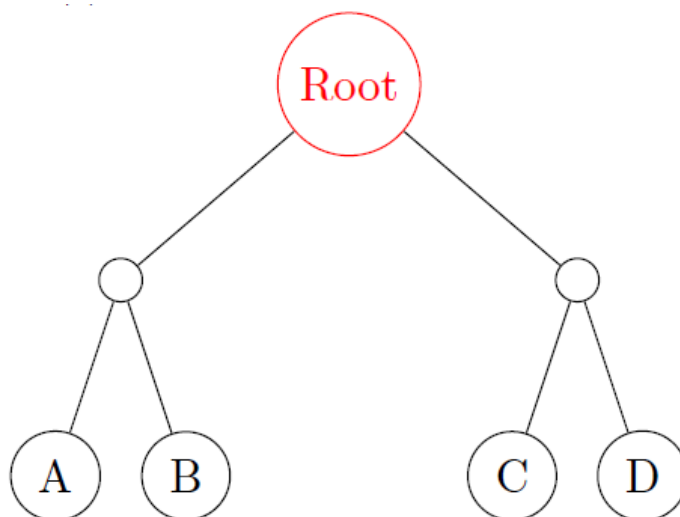


Рис. 2.20.

Всегда имеется корневой узел. Часть узлов будет вершинами или терминальными узлами, а часть узлов понадобится только для того, чтобы связать корень дерева и его вершины. Представление кода в виде дерева показывает, что для соблюдения условия префиксности не должно находиться символов в узлах, ведущих к другим символам.

2.2.2 Сжатие информации: алгоритм Хаффмана

Один из алгоритмов получения оптимальных префиксных кодов был предложен в 1952 году аспирантом MIT Дэвидом Хаффманом в его курсовой работе.

Для иллюстрации алгоритма возьмём следующий исходный текст: AAAABAABABABABCSAAAD.

Первый этап: определим встречаемость символов: $F_A = 12$, $F_B = 6$, $F_C = 2$ и $F_D = 1$. Так как код префиксный, не имеет смысла располагать символы в узлах, отличных от терминальных. Пусть у всех таких терминальных узлов, содержащих символы, вес узла определяется как произведение встречаемости символа на длину пути до корневого узла (глубину узла). Тогда под весом дерева будем понимать сумму всех весов символов. Требуется построить дерево, вес которого минимален из всех возможных, т. е. для которого:

$$\sum_{i=1}^n F_i L_i \rightarrow \min$$

где L_i — глубина i -го символа.

Попробуем понять, какими свойствами должно обладать оптимальное дерево. Во-первых, если из какого-то не узла исходит один путь, то, очевидно, можно поднять вверх это поддерево, сократив на единицу все длины путей до терминальных вершин. Во-вторых, пустых терминальных вершин быть не должно. В-третьих, из первых двух пунктов можно сделать вывод, что самое длинное кодовое слово должно быть парным, то есть, на самой длинной ветке дерева должно висеть два яблока (символа).

1. Перед работой алгоритма создадим (пока пустые) узлы дерева, в которые поместим символы и их встречаемость. Пометим их все как необработанные. Эти узлы пока в дерево не собраны и располагаются отдельно (рис. 2.21).

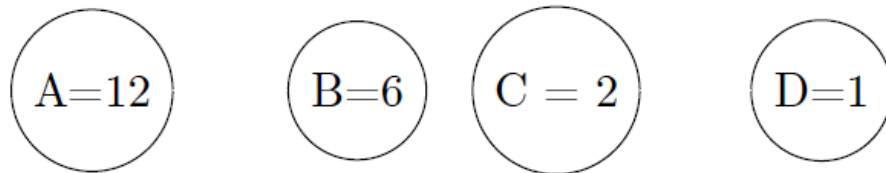


Рис. 2.21.

2. Так как два самых редко встречающихся символа должны иметь наибольшую длину и находиться на одной ветке, найдём их среди необработанных.

3. Для только что найденных узлов создаём новый, объединяющий их узел, детьми которого они будут. В него прописываем сумму встречаемости узлов-детей (рис. 2.22).

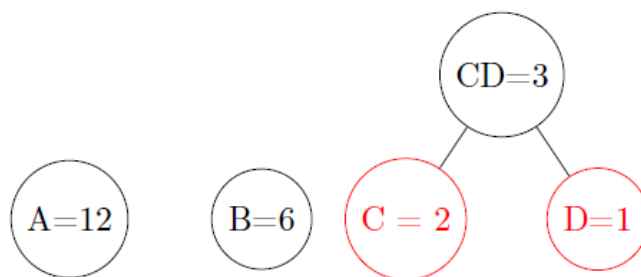


Рис. 2.22

4. После объединения узлов сами по себе они становятся неинтересными. Помечаем узлы или вершины, как уже обработанные (отправляем вниз), а вновь созданный узел – как необработанный (рис. 2.23).

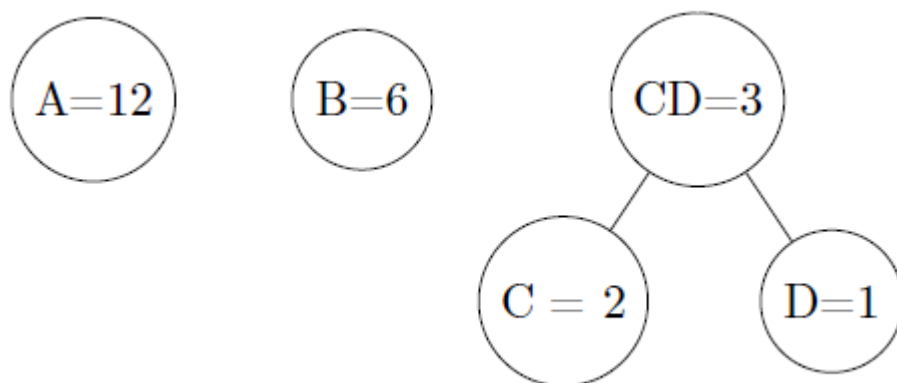


Рис. 2.23.

5. На каждом шаге алгоритма количество необработанных узлов уменьшается на единицу. Если необработанных не осталось, то алгоритм завершён (рис. 2.24, 2.25).

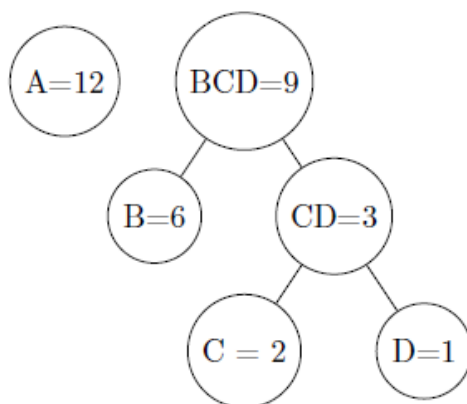


Рис. 2.24.

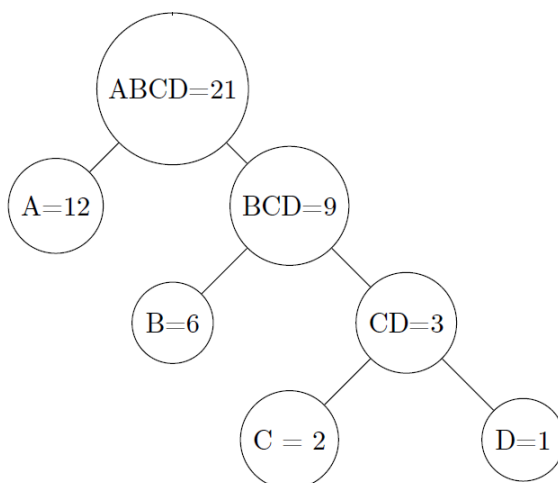


Рис. 2.25.

6. Переходим к шагу 2.

Коды, присвоенные терминальным узлам, содержащим символы, можно получить, обойдя дерево, начиная с вершины и присвоив пути налево бит 0, а пути направо – бит 1:

- $A \rightarrow 0$
- $B \rightarrow 10$
- $C \rightarrow 110$
- $D \rightarrow 111$

Общая длина всех кодовых слов $12 \cdot 1 + 6 \cdot 2 + 2 \cdot 3 + 1 \cdot 3 = 33 < 42$

Для работы данного алгоритма потребовались понятия: «найти два наименьших элемента в множестве», «построить дерево», «обойти дерево». Несмотря на то, что алгоритм по нашей классификации относится к классу жадных, для его реализации требуются новые структуры данных (упорядоченный массив, дерево) и новые алгоритмы (поиск наименьшего элемента, построение и обход деревьев), речь о которых пойдет в последующих главах настоящего учебного пособия.

2.3 Префиксное дерево

2.3.1 Задача о покрытии строки

Задача. Имеется набор «образцов» — s_i , $i = 1 \dots N$ — «слов», ни одно из которых не начинается с другого, то есть, образующих префиксный код.

Имеется строка p — «предложение».

Требуется определить, можно ли составить предложение p из слов s_i .

Например, если имеется набор слов $s_i = \{ab, ca, ra, dab\}$, то предложение *abracadabra* из них составить можно $ab + ra + ca + dab + ra$, а вот предложения *barca*, *abracadabraa* — нет.

Решение задачи. Для начала, необходимо определить, что является главными параметрами алгоритма. Одним из них точно должна являться длина предложения p — N . Пока алгоритм не составлен, больше информации о нём нет. Поэтому пусть, например, вторым главным параметром будет M — сумма длин слов s_i .

Решить задачу можно следующим жадным алгоритмом:

1. Устанавливаем указатель позиции на начало предложения.
2. Из списка слов выбираем то, которое полностью совпадает с подстрокой, начинающейся с указателя в предложении.

3. Если такого слова не найдено, выводим «нет», алгоритм завершён, ответ получен.

4. Если такое слово есть, перемещаем указатель в предложении на длину слова.

5. Если все слова закончились, а совпадения нет, то алгоритм завершён с ответом «нет».

6. Если предложение закончилось, то выводим «да» и завершаем алгоритм.

7. Возвращаемся к пункту 2.

Рассмотрим пример, представленный на рис. 2.26.

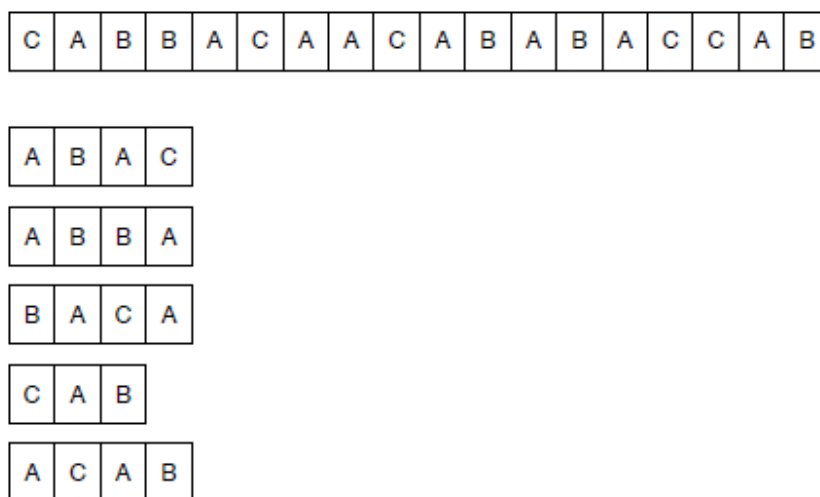


Рис. 2.26.

Указатель предложения установлен на первый его символ – букву С. Первые три слова в списке успеха не принесли, а вот четвёртое – совпало (см. рис. 2.27).

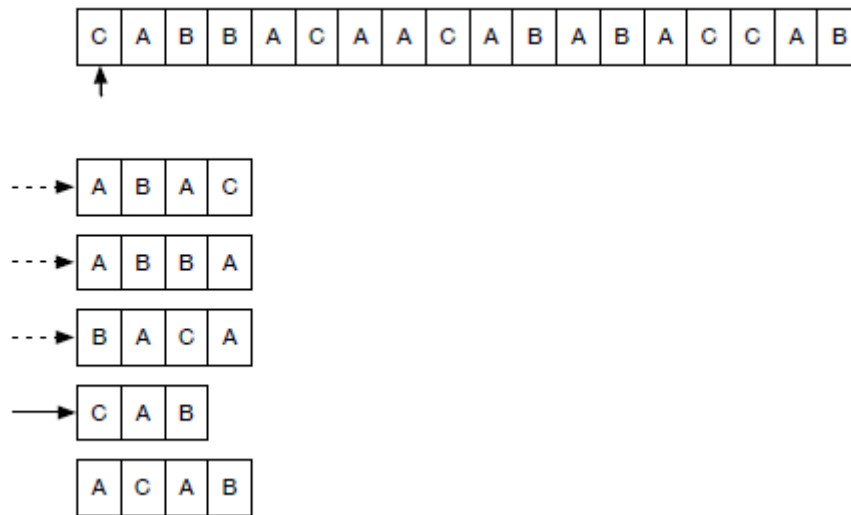


Рис. 2.27.

После этого необходимо переместить указатель предложения на длину совпавшего слова и запустить следующую итерацию (рис. 2.28).

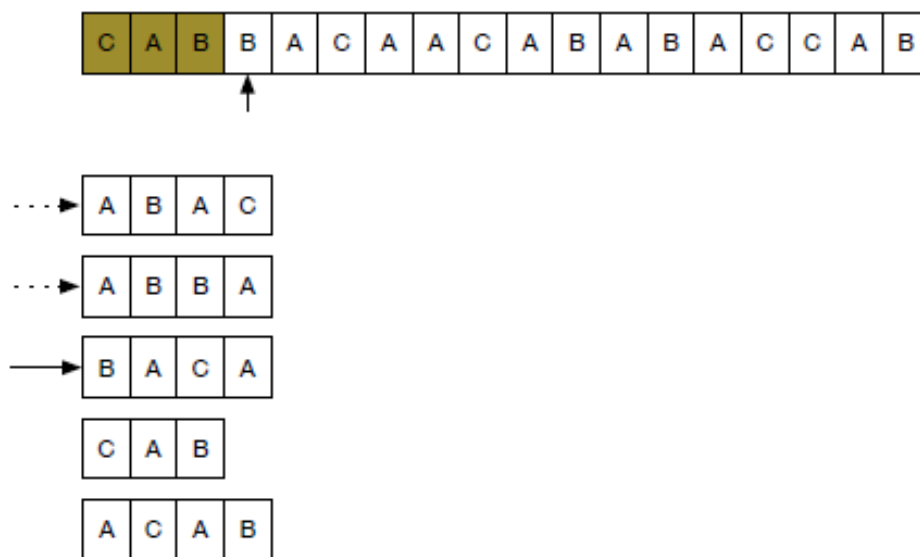


Рис. 2.28.

После нахождения еще одного слова, итерации продолжают (рис. 2.29).

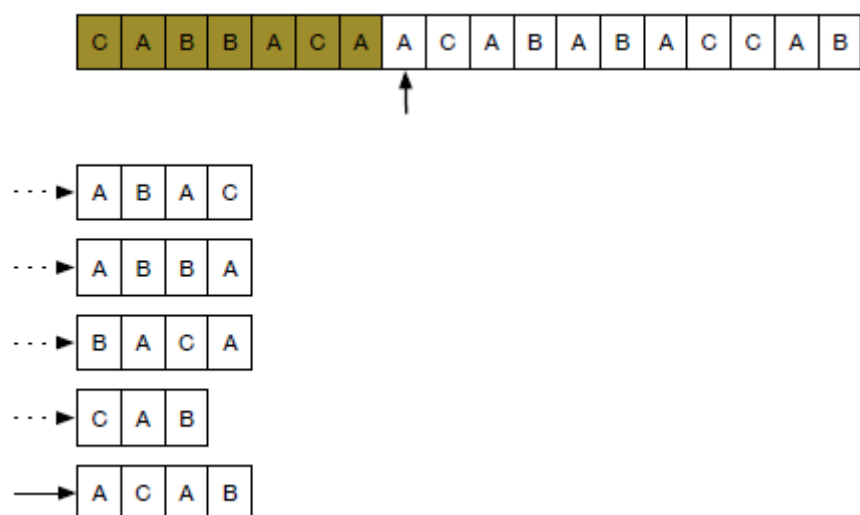


Рис. 2.29.

Оценим сложность предложенного алгоритма. Попытка оценить сложность алгоритма вызывает ряд вопросов, на которые нужно как-то ответить. Например, определить, что очередное слово из списка подошло, можно только просмотрев все слово. Но, при этом, определить, что слово не подошло, можно даже с первого символа слова. Отсюда возникает вопрос, сколько попыток поиска в слове будет происходить в среднем? Можно ли считать, что в среднем на каждое слово длины L придётся $L/2$ попыток? Точный ответ на этот вопрос зависит от функции распределения слов и букв в словах и является достаточно сложной самой по себе комбинаторной задачей. Нам требуется общая оценка сложности алгоритма, поэтому грубо оценим количество попыток на слово длины L как $L/2$

Еще один вопрос – что такое «средняя длина слова»? Оценим её в $L = M/K$, где K — число слов. Так, понадобился ещё один главный параметр.

На одном этапе поиска в среднем перебирается $K/2$ слов, каждое из которых имеет среднюю длину L , при этом один этап продвигает в среднем на L позиций в предложении. Исходя из этого можно оценить среднее количество этапов $T = N/L$ (рис. 2.30).

Итого общую сложность алгоритма в «среднем» сценарии можно оценить как $F = N/L \times K/2 \times L/2 = NK/4 = O(NK)$.

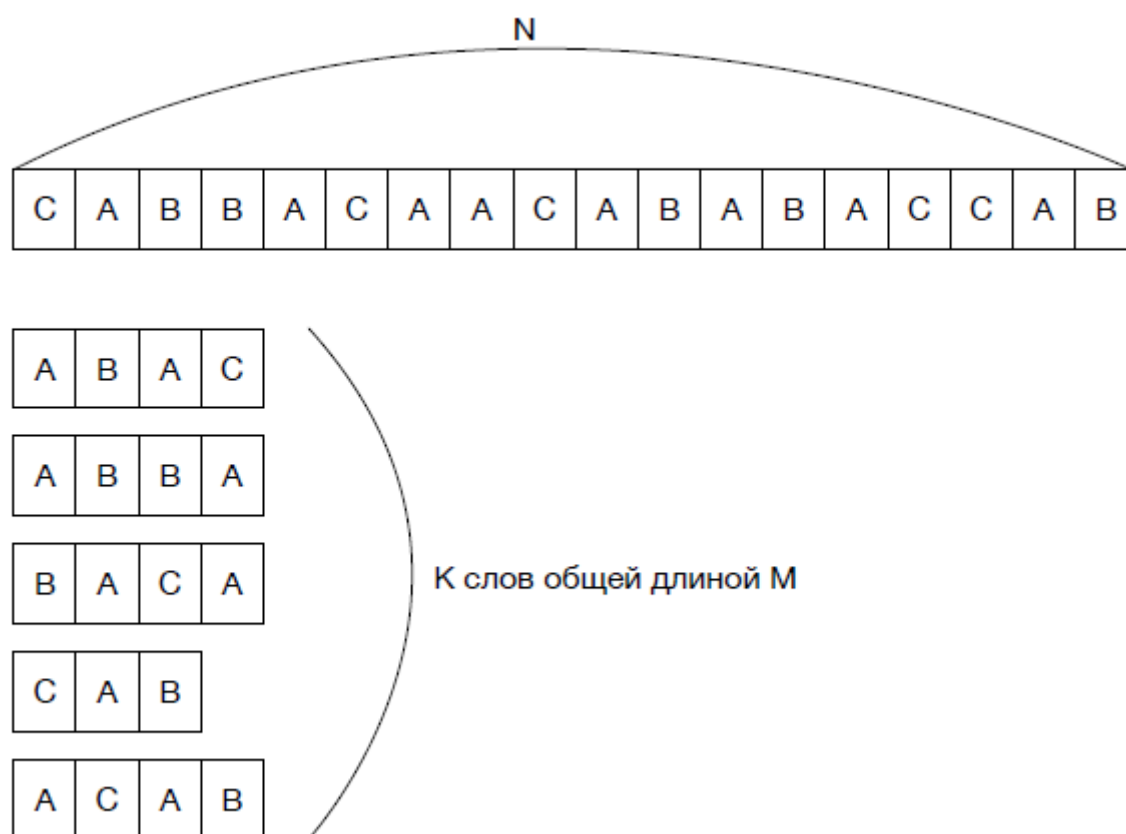


Рис. 2.30.

На первый взгляд может показаться, что результат является неверным, ведь сложность алгоритма не зависит от общей длины всех слов. Для доказательства верности оценки, рассмотрим крайний сценарии. Например, для $K = 1$, то есть, при поиске покрытия ровно одним словом длины M (пусть N делится нацело на M), каждый успешный поиск продвигает по предложению на M позиций. Наибольшее количество поисков составляет N / M , в каждом из которых сравнивается M символов, что даёт сложность в $F = O(N)$.

2.3.2 Использование префиксного дерева

Исследование сложности алгоритма дало понять, что для большого K эффективность алгоритма падает. Имеется ли более эффективное решение? Неэффективность только что разработанного алгоритма состоит в том, что, обнаружив несовпадение с одной подстрокой, ничего не получается для следующих.

Попробуем изменить структуру данных, усложнив представление тех строк, которые необходимо найти. Это даст возможность проводить поиск параллельно по всем словам. Расширим дерево, используемое в алгоритме Хаффмана, обобщив его до префиксного дерева. Алгоритм останется жадным, но использование другой структуры данных поменяет его сложность.

Дерево, которое было построено в предыдущей задаче (см. п. 2.2) уже было префиксным, но оно позволяло кодировать только последовательностями из нулей и единиц, так как каждый узел имел не более двух потомков. Идея о том, что каждая ветка однозначно определяет код, позволяет иногда уменьшить количество данных, требуемых для представления. Не умаляя общности положим, что из каждого узла всегда выходит ровно три ветви, А, В и С. Каждая из веток приходит или в узел, либо в вершину, либо в никуда. При этом, нет необходимости рисовать ветви, уходящие в никуда (см. рис. 2.31).

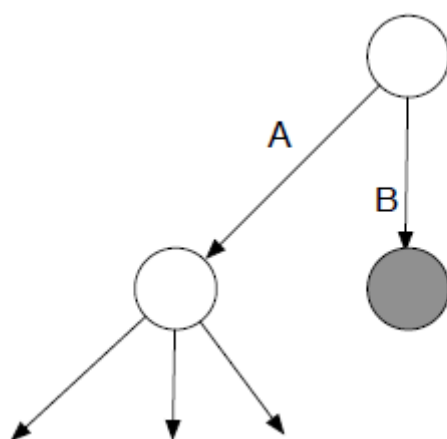


Рис. 2.31.

Пусть предоставлен следующий набор слов: АВАС, АВВА, ВАСА, САВ, АСАВ. Построим дерево для каждого слова посимвольно. Для первого слова, АВАС дерево будет таким, как показано на рис. 2.32.

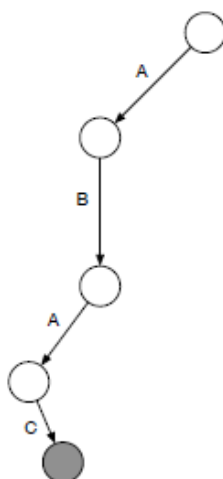


Рис. 2.32.

После добавления слов АВВА и ВАСА префиксное дерево изменит свой вид, как показано на рис. 2.33.

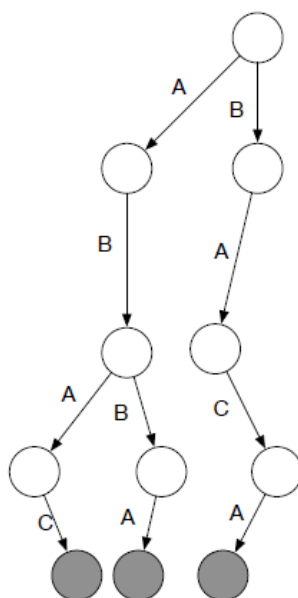


Рис. 2.33.

Заключительное дерево после добавления всех слов из словаря выглядит так, как показано на рис. 2.34.

Теперь применим новую структуру данных для поиска (рис. 2.35).

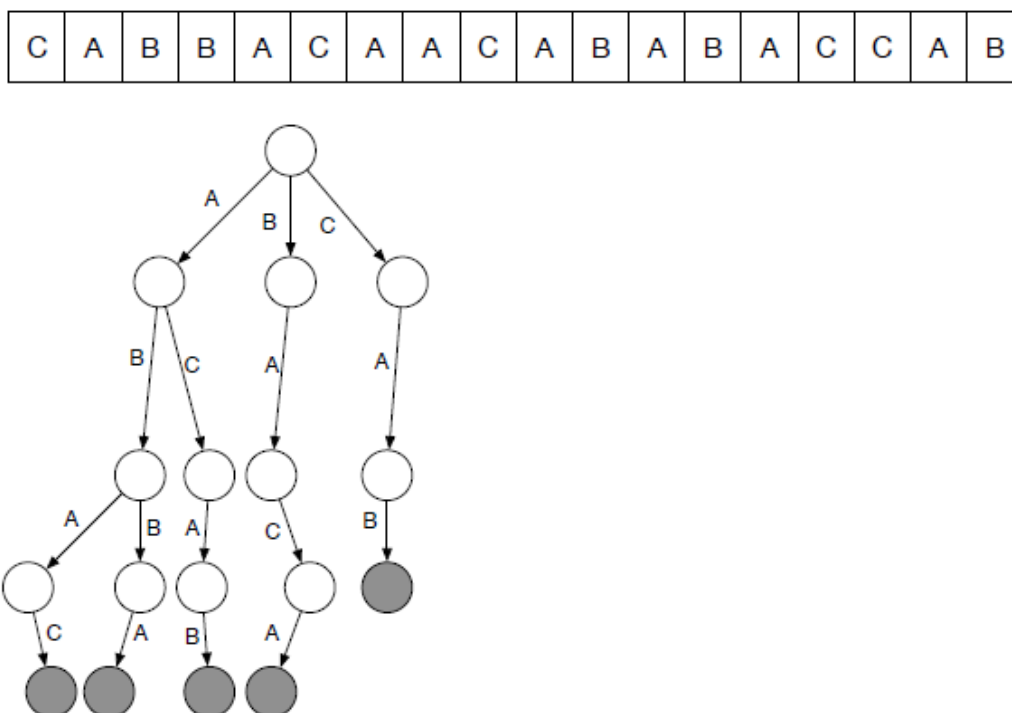


Рис. 2.35.

Алгоритм следующий:

1. Устанавливаем указатели на начало строки и на вершину дерева.
2. Если указатель стоит за последним символом в строке, то:
 - а) Если указатель в дереве находится в корне — алгоритм завершён с успехом;
 - б) Иначе алгоритм завершён с неудачей.
3. Считываем очередной символ строки и передвигаем указатель.
4. Если такой ветви дерева нет — завершаем алгоритм с неудачей.
5. Переходим по дереву по соответствующей ветке.
6. Если мы попали в серый узел — возвращаемся в корень дерева.
7. Переходим к пункту 2.

Каждое перемещение символа в предложении приводит к перемещению указателя в дереве. Сложность каждого перемещения постоянна и равна $O(1)$. Количество перемещений в случае успеха равно N , где N — длина строки. Итого: сложность алгоритма поиска — $O(N)$.

Общий алгоритм состоит из двух этапов – построения дерева и поиска по дереву, следовательно, общая сложность алгоритма – $(N + M)$. Вспомнив, что сложность первой версии алгоритма была $O(N \times K)$, убеждаемся, что, используя более сложную структуру данных, получен более эффективный алгоритм.

2.4 Строки

2.4.1 Структура данных «Строка»

Здесь и далее под строкой понимается структура данных, применяемая для хранения текстовой информации. В языке C# под строкой понимают любую последовательность байтов, заканчивающуюся нулевым. Это с одной стороны позволяет весьма эффективно реализовывать некоторые алгоритмы, но часто небезопасно. Самым большим недостатком такого представления является то, что для определения длины строки требуется её полный просмотр, а эта операция весьма востребована. Например, для того, чтобы в языке C# получить строку *s3*, состоящую из конкатенации (слияния) строк *s1* и *s2*, требуется:


1. Определить длину строки *s1*: `size_t l1 = strlen(s1);`
2. Определить длину строки *s2*: `size_t l2 = strlen(s2);`
3. Выделить участок памяти, достаточный для того, чтобы туда поместились обе строки: `char *s3 = malloc(l1+l2+1);`. Мы не должны забывать, что нужно зарезервировать в том числе и место под заключительный ограничительный нулевой байт
4. Скопировать первую строку в результирующую: `strcpy(s3,s1);`
5. Дописать вторую строку к концу первой: `strcpy(s3+l1,s2);`. Можно, конечно, воспользоваться функцией `strcat`: `strcat(s3,s2);`, но `strcat` опять должен был бы определить место, куда разрешено дописывать, пробегаая по строке *s3* с самого начала.


Нельзя сказать, что это всё очень эффективно, ведь неоднократно осуществляется пробег по строкам s_1 и s_2 . Эффективность можно было бы повысить, добавив счётчик длины для строки и организовав бы всё вместе в одну структуру. К счастью, в стандарте C# это уже сделано в виде класса `string`. Операция конкатенации строк теперь смотрится так: `string s3 = s1 + s2;`.

2.4.2 Z-функция

Строки применяются в том числе и для того, чтобы определять, содержится ли одна строка в другой, то есть, для операций поиска. Z-функция, рассмотренная в данном параграфе, поможет в ряде алгоритмов обработки строк.

Дадим несколько определений. Пусть имеется строка $s[0..n)$

 Подстрокой $\text{sub}(s, p, l)$ строки s называется строка, состоящая из символов $s[p..p+l)$. Префиксом длины l строки s называется подстрока $s[0..l)$. Суффиксом длины l строки $s[0..k)$ называется подстрока $s[k-l..k)$. Собственным префиксом строки $s[0..k)$ называется префикс длины $l < k$. Собственным суффиксом строки $s[0..k)$ называется суффикс длины $l < k$.

 Z-функция от строки s и позиции p есть длина наибольшей подстроки строки s , начинающейся в позиции p , совпадающей с собственным префиксом строки s .

a	b	r	a	s	h	v	a	b	r	a	c	a	d	a	b	r	a
0	0	0	1	0	0	0	4	0	0	1	0	1	0	4	0	0	1

Рис. 2.36. Значения Z-функции для строки `abrashvabracadabra`

Цель – разработать эффективный алгоритм решения задачи построения Z-функции от строки.

Обычная практика при разработке алгоритмов – вначале реализовать простейший, но не обязательно самый эффективный вариант. Обычно доказать корректность такого решения существенно проще, чем более сложного. При написании программы, реализующей алгоритм, простейшее решение может

выступать как эталонное. Алгоритм не считается реализованным корректно до тех пор, пока хотя бы в одном случае, хотя бы на одном наборе входных данных, результат его работы отличается от эталонного.

Корректность следующего алгоритма не вызывает сомнений, так как он просто реализует определение Z-функции:

1. Обнулить выходной массив `ret`.
2. Для всех значений `j` от 1 до размера строки делать:
 - а) Установить `ret[j]` равным длине максимальных совпадающих подстрок, начинающихся с 0 и с `j`.

Попробуем исполнить этот алгоритм для различных входных данных. Стрелка над строкой на рисунках 2.37–2.39 показывает совпавший префикс, стрелка под строкой – совпавшую часть подстроки. Начала и концы стрелок показывают позиции сопоставления.

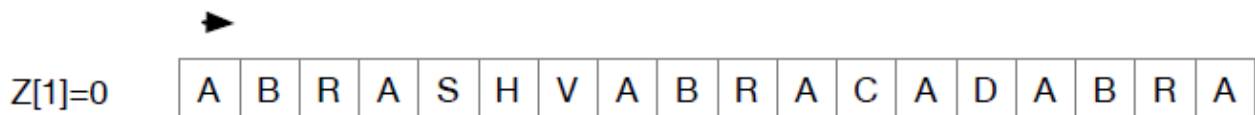


Рис. 2.37. Сопоставление для $p=1$

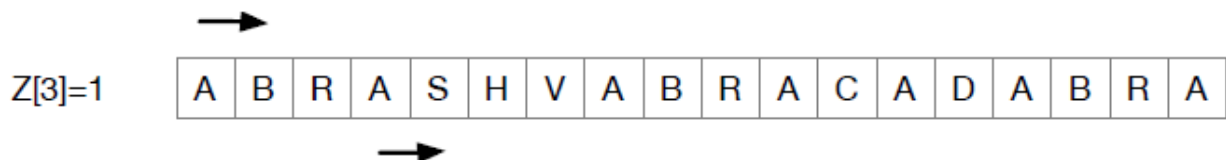


Рис. 2.38. Сопоставление для $p=3$

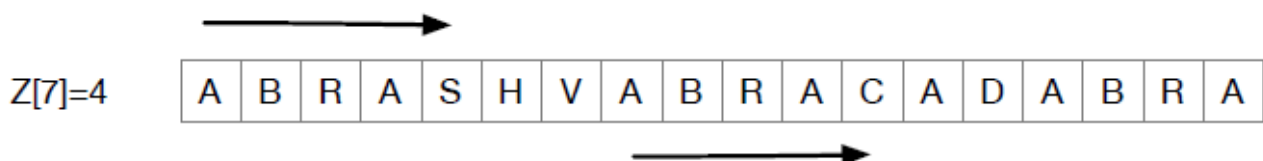


Рис. 2.39. Сопоставление для $p=7$

Эталонная программа, реализующая алгоритм проста, функция `z` возвращает вектор Z-функции для строки-входного аргумента.

```
public int[] z(string s) {
    int[] ret = new int[s.Length];
    for (int i = 1; i < s.Length; i++) {
        int p = i;
```

```

        while (p < s.Length && s[p] == s[i - ret[i]])
            p++;
        ret[i] = p - i;
    }
    return ret;
}

```

Опять возникает три ключевых вопроса: Корректен ли алгоритм? Какова его сложность? Можно ли его улучшить?

Ответ на первый вопрос – да, данная функция просто использует определение Z-функции.

Для ответа на второй вопрос можно оценить сложность, выявив наихудший случай, именно он даст нам оценку сверху по количеству требуемых операций. Ряд попыток прогона алгоритма показывает, что наихудший случай – строка, состоящая из одинаковых символов.

Сопоставление для $p=1$ требует $N - 1$ сравнение символов, для $p=2$ требует $N-2$ сравнения. Общая сложность алгоритма составляет:

$$T(N) = (N - 1) + (N - 2) + \dots + 1 = O(N^2)$$

Возможно, для данного алгоритма это и есть предельная сложность? Чтобы понять, можно ли алгоритм улучшить, требуется понаблюдать за его поведением. Вернёмся к первой строке, `abrashvabracadabra` и рассмотрим случай поиска значения для $p=4$ (рис. 2.40).

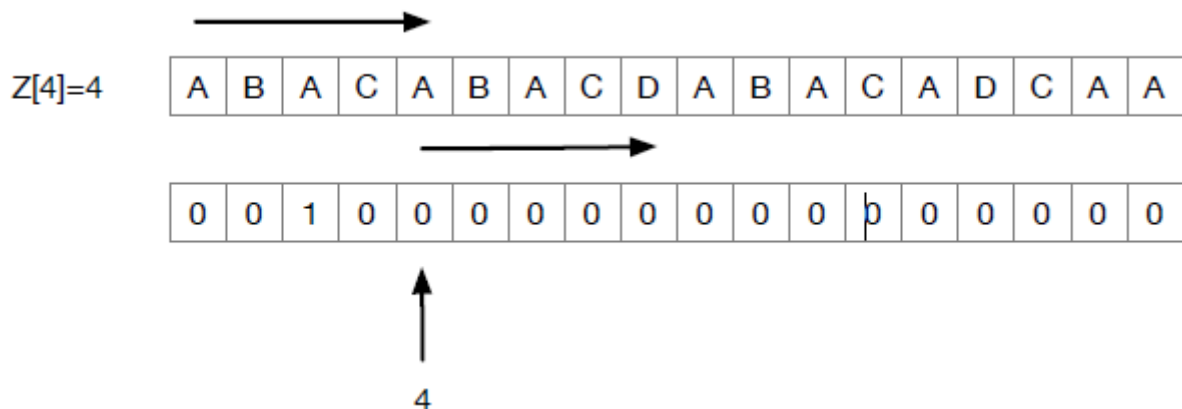


Рис. 2.40.

Введём понятие *последняя сопоставленная подстрока*. Значение Z-функции для $p=4$ оказалось равно 4. Почему же сопоставление для $p=5$ даёт 0? (рис. 2.41)

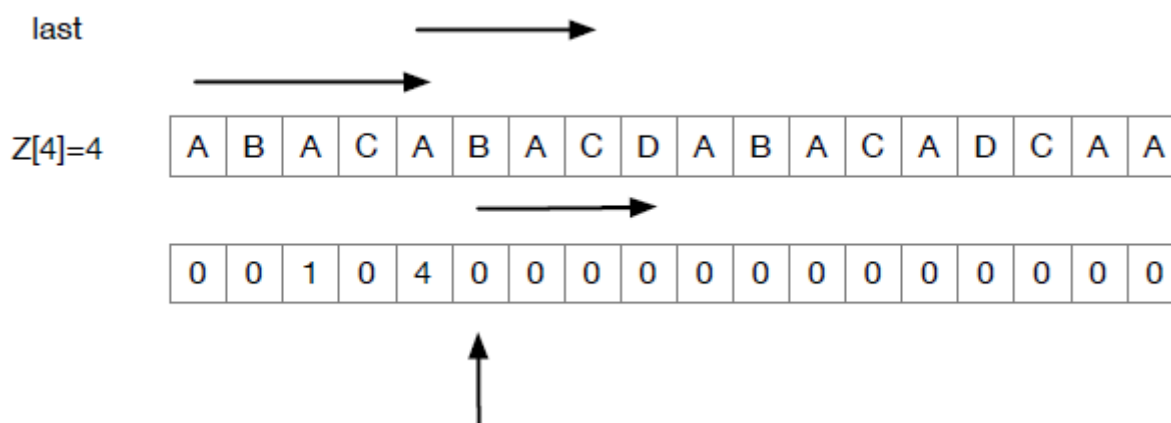


Рис. 2.41. Случай поиска значения для $p=5$

Получается, что если p попадает внутрь подстроки сопоставления, то он сдвигается сразу за правую границу. Если p не попадает, то ищем как обычно.

В эталонный алгоритм внесем два изменения:

1. начало поиска теперь может сдвинуться сразу на несколько позиций;
2. при выходе p за границу подстроки сопоставления меняем эту подстроку на новую.

```

public int[] z(string s) {
    int[] ret = new int[s.Length];
    for (int i = 1, l=0, r=0; i < s.Length; i++) {
        int p = i > r ? i : i + Math.Min(r - i + 1, ret[i
- 1]);

        while (p < s.Length && s[p] == s[p-i])
            p++;
        ret[i] = p - i;
        if (p > r)
        {
            l = i;
            r = p - 1;
        }
    }
    return ret;
}

```

Сложность алгоритма теперь посчитать труднее. Инвариант: положение конца подстроки сопоставления изменяется на её длину L . С другой стороны, поиск внутри строки сопоставления тоже составляет $O(L)$. Это означает, что

каждый из символов строки обрабатывается за время $O(1)$, что и даёт нам сложность всего алгоритма:

$$T(N) = O(N).$$

От $O(N^2)$ сложность уменьшена до $O(N)$. Время исполнения программы для разных N показывает, что чем больше входная строка, тем больше выигрыш во времени исполнения.

Можно ли улучшить этот алгоритм? Вероятно, нет. Так как длина строки равна N и для построения Z -функции потребуется заполнить не менее N элементов выходного массива, сложность не может быть меньше, чем $O(N)$.



Реализуйте в созданном проекте метод, который находит Z -функцию для введенной строки. Протестируйте данный метод на больших строках. Сравните первый и второй вариант построения Z -функции для строк разной длины. По результатам сравнения составьте таблицу.

Контрольные вопросы

1. Какие задачи называются экстремальными?
2. В чем особенность жадных алгоритмов?
3. Как найти локально оптимальный минимум функции с помощью жадного алгоритма?
4. В чем суть задачи об интервалах? Как можно интерпретировать данную задачу?
5. Какие стратегии можно предложить для жадного решения задачи об интервалах?
6. Как решить задачу о резервных копиях с помощью жадного алгоритма?
7. Решить задачу о рюкзаке с помощью жадного алгоритма.
8. Какой код называется префиксным?
9. Что такое двоичное дерево?
10. Для чего используется алгоритм Хаффмана?

11. Объясните, как работает алгоритм Хаффмана.
12. Что такое префиксное дерево? Как построить префиксное дерево?
13. Какое жадное решение можно предложить для задачи о покрытии строки?
14. Как можно улучшить решение задачи о покрытии строки с помощью префиксного дерева?
15. Чему равна сложность алгоритма поиска с использованием префиксного дерева?
16. Что такое подстрока строки?
17. Что такое префикс длины l строки? Что такое суффикс длины l строки?
18. Что такое собственный префикс и собственный суффикс строки?
19. Что такое Z-функция от строки?
20. Как найти Z-функцию строки?

Задания для самостоятельного выполнения

Задание №1. Сумма элементов подмассива

Ограничение по времени: 1 секунда. Ограничение по памяти: 256 мегабайт

Имеется массив V целых чисел, состоящий из $1 \leq N \leq 10^8$ элементов, $-2 \times 10^9 \leq V_i \leq 2 \times 10^9$.

Подмассивом называют непрерывное подмножество элементов массива, возможно, включающее в себя и полный массив.

Требуется найти наибольшую из возможных сумм всех подмассивов.

Формат входных данных:

N

V_1

V_2

...

V_N

Формат выходных данных:

Задание №2. Длинное сложение и вычитание

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

На вход подается три строки. Первая содержит представление длинного десятичного числа (первый операнд), вторая – представление операции, строки + и -, третья – представление второго операнда.

Длина первой и третьей строки ограничены 1000 символами. Вторая строка содержит ровно один символ.

Требуется исполнить операцию и вывести результат в десятичном представлении.

Формат входных данных:

123

+

999

Формат выходных данных:

1122

Задание №3. Периодическая дробь

Ограничение по времени: 1 секунда. Ограничение по памяти: 256 мегабайт

Выведите десятичное представление рациональной правильной дроби. Если в представлении присутствует период, то нужно вывести первое его вхождение в круглых скобках.

Формат входных данных:

Два целых числа, введенных через пробел: $1 \leq N < M \leq 150000000$

Формат выходных данных:

Десятичное представление числа N/M

Задание №4. Танец точек

Ограничение по времени: 1 секунда. Ограничение по памяти: 256 мегабайт

На прямой располагается $1 \leq N \leq 10000$ точек с целочисленными координатами $-10^9 \leq V_i \leq 10^9$. Каждой из точек разрешается сделать ровно одно движение (танцевальное па) в любом направлении на расстояние не больше $0 \leq L \leq 10^8$ и остановиться на другой позиции. Какое минимальное количество точек может остаться на прямой после окончания танца (все точки после танца, оказывающиеся на одной позиции, сливаются в одну)?

Формат входных данных:

L N

V1 V2 ... VN

Формат выходных данных:

MinimalNumberOfPoints

Задание №5. Ровно M простых

Ограничение по времени: 2 секунды. Ограничение по памяти: 24 мегабайта

Требуется найти такое наименьшее натуральное число $2 \leq K \leq 2 \times 10^7$, что, начиная с этого числа, среди N натуральных чисел имеется ровно M простых.

Если такого числа не существует или оно больше 2×10^7 , вывести -1.

Формат входных данных:

M N

Формат выходных данных:

K или -1

ГЛАВА 3. АЛГОРИТМЫ СОРТИРОВКИ

3.1 Задача сортировки

Имеется последовательность из n ключей.

$$k_1, k_2, \dots, k_n.$$

Требуется: упорядочить ключи по не убыванию или не возрастанию. Это означает: найти перестановку ключей

$$p_1, p_2, \dots, p_n$$

такую, что

$$k_{p_1} \leq k_{p_2} \leq \dots \leq k_{p_n}$$

или

$$k_{p_1} \geq k_{p_2} \geq \dots \geq k_{p_n}$$

Элементами сортируемой последовательности могут иметь любые типы данных. Обязательное условие – наличие ключа.

Возьмём, к примеру последовательность:

(Москва, 10000000), (Нью-Йорк, 12000000), (Париж, 9000000), (Токио, 20000000), (Лондон, 10000000), (Дели, 9000000)

Пусть ключом будет число жителей. Можно упорядочить эту последовательность несколькими способами, так как имеются одинаковые ключи. Но, как оказывается, не все способы одинаково полезны и иногда важно обеспечить устойчивость сортировки.



Алгоритм сортировки *устойчивый*, если он сохраняет относительный порядок элементов с одинаковыми ключами. Например, если в начальной последовательности Москва (имеющая одинаковое количество жителей с Лондоном) находилась перед Лондоном, то в устойчивой сортировке она тоже должна остаться перед ним:

(Токио, 20000000), (Нью-Йорк, 12000000), (Москва, 10000000), (Лондон, 10000000), (Париж, 9000000), (Дели, 9000000)

В неустойчивой сортировке они могут поменяться местами.

Можно отметить ещё одно важное для теории алгоритмов свойство: при наличии одинаковых ключей возможны несколько вариантов отсортированных последовательностей, удовлетворяющих свойству отсортированности, но только устойчиво отсортированная последовательность ровно одна.

3.2 Сортировки сравнением


Необходимо понять, каким образом определить, что один элемент должен находиться ранее другого в отсортированной последовательности. Самым популярным способом является сравнение этих двух элементов. Сортировки, которые сравнивают пары элементов для их упорядочения, называются *сортировками сравнением*. Для исполнения алгоритма сортировки сравнением необходимо определить операцию сравнения ключей:

$$a < b$$

Полагается, что

$$\text{not}(a < b) \wedge \text{not}(b < a) \rightarrow a = b$$

Это необходимое условие для соблюдения закона *трихотомии*: для любых a, b либо $a < b$ либо $a = b$, либо $a > b$.

 *Инверсия* — пара ключей с нарушенным порядком следования. В последовательности {4, 15, 6, 99, 3, 15, 1, 8} имеются следующие инверсии: (4,3), (4,1), (15,6), (15,3), (15,1), (15,8), (6,3), (6,1), (99,3), (99,15), (99,1), (99,8), (3,1), (15,1), (15,8).

Перестановка соседних элементов, расположенных в ненадлежащем порядке, уменьшает инверсию ровно на 1. Количество инверсий в любом множестве конечно, в отсортированном — равно нулю. Следовательно, количество обменов для сортировки конечно и не превосходит числа инверсий. Наблюдение за уменьшением инверсий даёт алгоритм сортировки пузырьком.

3.2.1 Сортировка пузырьком

Основная идея алгоритма пузырьковой сортировки заключается в следующем: до тех пор, пока найдётся пара соседних элементов, расположенных не в надлежащем порядке, меняем их местами.

Корректность алгоритма сортировки пузырьком прямо следует из определения инверсии:

- Если инверсий в последовательности нет, то алгоритм завершён.

- Если хотя бы одна инверсия в последовательности есть, то на очередном проходе по массиву количество инверсий уменьшается хотя бы на 1.

Рассчитаем сложность данного алгоритма. Так как один проход по массиву имеет сложность $O(N)$, быстрее установить, что массив уже отсортирован невозможно, поэтому в лучшем случае сложность составляет именно $O(N)$, например, для последовательности $\{1, 2, 3, 4, 5, 6\}$.

случае — последовательность в противоположном порядке, такая, как $\{6, 5, 4, 3, 2, 1\}$. Худший случай возникает тогда, когда на каждом проходе хотя бы одна пара элементов переставляется. Тогда количество проходов есть $O(N)$, количество сравнений будет $(N - 1) + (N - 2) + \dots + 1 = O(N^2)$. Самый худший, $\{4, 3, 2, 1\}$. К $O(N^2)$ сравнениям добавляются достаточно дорогие перестановки. Количество инверсий у этой последовательности максимально, $(N \cdot (N - 1))/2$, количество перестановок соответствует этому числу. Сложность алгоритма не изменилась — $O(N^2)$, но заметно возрос коэффициент амортизации.

А чему равно количество инверсий случайного массива в среднем? Для его нахождения потребуется привлекать относительно сложный математический аппарат (см. подробнее в [1]).

Код алгоритма представлен ниже:

```
static void SortBubble (int[] a) {
    bool sorted = false;
    int n = a.Length;
    while (!sorted) {
        sorted = true;
        for (int i = 0; i < n-1; i++) {
```



```

        if (a[i] > a[i+1]) {
            int tmp = a[i];
            a[i] = a[i+1];
            a[i+1] = tmp;
            sorted = false;
        }
    }
}

```

Для этого и других алгоритмов сортировки будем находить инвариант, соблюдающийся при исполнении алгоритма. Это позволит более подробно исследовать алгоритм и определить его применимость. Инвариант: после i -го прохода на верных местах находится не менее i элементов «справа» (как показано на рис. 3.1):

$\{ \boxed{5, 3}, 15, 7, 6, 2, 11, 13 \}$
 $\{ 3, \boxed{5, 15}, 7, 6, 2, 11, 13 \}$
 $\{ 3, 5, \boxed{15, 7}, 6, 2, 11, 13 \}$
 $\{ 3, 5, 7, \boxed{15, 6}, 2, 11, 13 \}$
 $\{ 3, 5, 7, 6, \boxed{15, 2}, 11, 13 \}$
 $\{ 3, 5, 7, 6, 2, \boxed{15, 11}, 13 \}$
 $\{ 3, 5, 7, 6, 2, 11, \boxed{15, 13} \}$
 $\{ 3, 5, 7, 6, 2, 11, 13, \underline{15} \}$
 $\{ 3, 5, 6, 2, 7, 11, \underline{13, 15} \}$
 $\{ 3, 5, 2, 6, 7, \underline{11, 13, 15} \}$
 $\{ 3, 2, 5, 6, 7, \underline{11, 13, 15} \}$
 $\{ 2, 3, 5, 6, 7, 11, 13, \underline{15} \}$

Рис. 3.1. Иллюстрация инварианта при выполнении пузырьковой сортировки

Особенности алгоритма сортировки пузырьков:

- Крайне проста в реализации и понимании.
- Устойчива. Действительно, так как сортировка оперирует только соседними элементами, она при соседних элементах с одинаковыми ключами не способна перебросить один элемент через голову другого.

- Сложность в наилучшем случае $O(N)$.
- Сложность в наихудшем случае $O(N^2)$.
- Сортирует на месте (*in-place*). Это означает, что для сортировки не требуется создавать другой массив размера, сравнимого с исходным. Для реализации требуется всего 3-4 переменных.



Создайте новый проект. Напишите метод, сортирующий заданный целочисленный массив с помощью алгоритма сортировки пузырьком.

Протестируйте метод для массивов разной размерности.

3.2.2 Сортировка вставками

Идея алгоритма сортировки вставками заключается в том, что необходимо поддерживать следующий инвариант: после i -го прохода по массиву в первых i позициях будут находиться отсортированные i элементов первоначального массива. Алгоритм неформально можно описать так:

- Первый проход — нужно поместить самый лёгкий элемент на первую;
- В i -м проходе ищется, куда поместить очередной a_i внутри левых i элементов;
- Элемент a_i помещается на место, сдвигая вправо остальные внутри области $0 \dots i$.

Инвариант сортировки вставками представлен на рис. 3.2.

$$\underbrace{a_1, a_2, \dots, a_{i-1}}_{a_1 \leq a_2 \leq \dots \leq a_{i-1}}, a_i, \dots, a_n$$

Рис. 3.2. Инвариант сортировки вставками

На шаге i имеется упорядоченный подмассив a_1, a_2, \dots, a_{i-1} и элемент a_i , который надо вставить в подмассив без потери упорядоченности. После первого шага инвариант соблюдается. Соблюдение инварианта после остальных шагов показывает нам корректность алгоритма. Метод, реализующий данный алгоритм на языке C#, представлен ниже:

```

static void SortInsertion(int[] a, int n) {
    for (int i = n-1; i > 0; i--) {
        if (a[i-1] > a[i]) {
            int tmp = a[i-1]; a[i-1] = a[i]; a[i] = tmp;
        }
    }
    for (int i = 2; i < n; i++) {
        int j = i;
        int tmp = a[i];
        while (tmp < a[j-1]) {
            a[j] = a[j-1]; j--;
        }
        a[j] = tmp;
    }
}

```

Оценим сложность алгоритма сортировки вставками.

– Худший случай – упорядоченный по убыванию массив. Тогда цикл вставки на i -й итерации каждый раз будет доходить до 1-го элемента, что даст нам сложность $O(i)$ на итерацию.

– Для вставки элемента a_i потребуется $i - 1$ итерация.

– Позиции ищутся для $N - 1$ элемента. Общее время равно

$$T(N) = \sum_{i=2}^N Nc(i-1) = \frac{cn(n-1)}{2} = O(N^2)$$

– Лучший случай — упорядоченный по возрастанию массив. $T(N) = O(N)$.

Так как каждый раз приходит элемент a_{i+1} , больший элемента a_i , то операция перемещения не производится ни разу и сложность вставки каждого из элементов кроме первого оказывается $O(1)$.

Особенности сортировки вставками:

– Сортировка упорядоченного массива требует $O(N)$. Если массив частично упорядочен, то большое количество перемещений элементов будет проводиться за время $O(1)$, что уменьшит общую сложность алгоритма;

- Сложность в худшем случае $O(N^2)$;

– Алгоритм устойчив. При перестановке элементов не существует ситуаций, когда элементы, имеющие одинаковые ключи, могут перескакивать друг через друга;

– Число дополнительных переменных не зависит от размера (*in-place*);

– Позволяет упорядочивать массив при динамическом добавлении новых элементов — *online*-алгоритм. Если массив уже отсортирован, то добавление нового элемента будет происходить со сложностью $O(N)$. Это свойство иногда оказывается очень важным. Отнюдь не все эффективные алгоритмы могут похвастаться таким поведением.



В созданном проекте напишите метод, сортирующий данный целочисленный массив с помощью алгоритма сортировки вставками.

Протестируйте метод для массивов разной размерности.

3.2.3 Сортировка Шелла

Одной из модификаций алгоритма пузырьковой сортировки является алгоритм Шелла. Прежде чем перейти к рассмотрению данного алгоритма, отметим некоторые моменты в сортировке методом пузырька. Как известно, что один обмен неупорядоченных соседних элементов уменьшает инверсию на 1. Наихудшим случаем для этого алгоритма будет упорядоченная в противоположном порядке последовательность:

$$I(\{8, 7, 6, 5, 4, 3, 2, 1\}) = (8 \cdot 7) / 2 = 28.$$

Однако, обмен местами именно соседних элементов не является панацеей. Может быть попробовать обменивать элементы с расстоянием $d > 1$? Так рассуждал американский математик Дональд Шелл в 1959 году.

Рассмотрим ситуацию, когда происходит обмен в сортировке методом пузырька не соседних элементов, а элементов, находящихся на расстоянии $d=4$.

Обмен первого элемента с пятым даёт последовательность, инверсия которой

$$I(\{4, 7, 6, 5, 8, 3, 2, 1\}) = 21.$$

За один шаг инверсия уменьшилась с 28 до 21, то есть на 7.

Продельвая эту же операцию с вторым и шестым элементами, с третьим и седьмым и так далее, получаем новую последовательность:

{8, 7, 6, 5, 4, 3, 2, 1}

{4, 7, 6, 5, 8, 3, 2, 1}

{4, 3, 6, 5, 8, 7, 2, 1}

{4, 3, 2, 5, 8, 7, 6, 1}

{4, 3, 2, 1, 8, 7, 6, 5}

Проделаем то же самое, уменьшив шаг d до двух:

{4, 3, 2, 1, 8, 7, 6, 5}

{2, 3, 4, 1, 8, 7, 6, 5}

{2, 1, 4, 3, 8, 7, 6, 5}

{2, 1, 4, 3, 6, 7, 8, 5}

{2, 1, 4, 3, 6, 5, 8, 7}

Обратим внимание на то, что при классической сортировке пузырьком самый маленький из элементов, 1, за два прохода переместился бы на две единицы влево, а если он стоял на крайней правой позиции, то количеством проходов, меньшим, чем $N - 1$ обойтись было бы невозможно. Здесь же единица после второго прохода уже оказалась на второй позиции.

Третий проход при $d = 1$ даёт следующие шаги:

{2, 1, 4, 3, 6, 5, 8, 7}

{1, 2, 4, 3, 6, 5, 8, 7}

{1, 2, 3, 4, 6, 5, 8, 7}

{1, 2, 3, 4, 5, 6, 8, 7}

{1, 2, 3, 4, 6, 5, 7, 8}

После всего трёх проходов удалось отсортировать самый неудобный для сортировки пузырьком массив. Сам Шелл предложил выбирать шаги сортировки в виде убывающей геометрической прогрессии с шагами $N/2, N/4, \dots, 1$. Как вследствие оказалось, предложенный Шеллом вариант оказался не лучшим из возможных. Да, в среднем сложность алгоритма уменьшилась, но в худшем случае (для специально сформированной последовательности) она осталась $O(N^2)$.

Было предложено множество различных вариантов последовательностей шагов. Стоит отметить, что от выбора последовательности зависит сложность алгоритма. Например, для последовательности $a = \{1, 4, 13, \dots, 3a_{n-1}+1, \dots\}$ сложность в среднем составляет $O(N^{4/3})$, а в худшем — $O(N^{3/2})$. Между тем, для последовательности $d = \{1, 8, 23, 77, \dots, 4^{i+1} + 3 \cdot 2^i + 1, \dots\}$ сложность в наихудшем случае составляет $O(N^{4/3})$.

Реализация алгоритма сортировки Шелла представлена ниже:

```
public static void SortShell(int[] a)
{
    int h;
    int n = a.Length;
    bool flag = false;
    while(!flag)
    {
        for (h = 1; h <= n / 9; h = 3 * h + 1);
        for (; h > 0; h /= 3)
        {
            for (int i = h; i < n; i++)
            {
                int j = i;
                int tmp = a[i];
                while (j >= h && tmp < a[j - h])
                {
                    a[j] = a[j - h];
                    j -= h;
                    flag = true;
                }
                a[j] = tmp;
            }
        }
    }
}
```

В первых строках выбирается начальный шаг, который затем уменьшается по приведённому закону. Логическая переменная flag служит для проверки условия Айверсона.

Выделим особенности сортировки Шелла:

- Сортировка упорядоченного массива требует $O(N)$.
- Алгоритм неустойчив. Действительно, при начальных, больших шагах сравнения, возможен обмен далеко отстоящих друг от друга элементов.

Алгоритм не видит элементов в промежутке между обмениваемыми, поэтому может поменять порядок элементов с одинаковыми ключами.

– Число дополнительных переменных не зависит от размера (*in-place*).

– Низкий коэффициент амортизации и простота реализации делает этот алгоритм конкурентом популярным алгоритмам при не очень больших N .



В созданном проекте напишите метод, сортирующий данный целочисленный массив с помощью алгоритма сортировки Шелла.

Протестируйте метод для массивов разных размеров.

3.2.4 Сортировка выбором

Если попытаться уменьшить количество обменов между собой в предыдущем алгоритме, то можно получить алгоритма сортировки выбором. Идея этой сортировки также интуитивно понятна: на первом шаге находим наименьший элемент массива и меняем его с первым. Появляется упорядоченный набор из одного элемента. Каждый очередной шаг будет добавлять к этому набору по наименьшему из тех элементов, которые пока не входят в набор. На втором шаге находится наименьший элемент из подмассива, начинающегося с позиции 2 и меняется местами с элементом, находящимся на позиции 2 и так далее.

Инвариант: после i итераций упорядочены первые i элементов (рис. 3.3).

$\{5, 3, 15, 7, 6, \boxed{2}, 11, 13\}$
 $\{\underline{2}, \boxed{3}, 15, 7, 6, 5, 11, 13\}$
 $\{\underline{2}, \underline{3}, 15, 7, 6, \boxed{5}, 11, 13\}$
 $\{\underline{2}, \underline{3}, \underline{5}, 7, \boxed{6}, 15, 11, 13\}$
 $\{\underline{2}, \underline{3}, \underline{5}, \underline{6}, \boxed{7}, 15, 11, 13\}$
 $\{\underline{2}, \underline{3}, \underline{5}, \underline{6}, \underline{7}, 15, \boxed{11}, 13\}$
 $\{\underline{2}, \underline{3}, \underline{5}, \underline{6}, \underline{7}, \underline{11}, 15, \boxed{13}\}$
 $\{\underline{2}, \underline{3}, \underline{5}, \underline{6}, \underline{7}, \underline{11}, \underline{13}, \boxed{15}\}$
 $\{\underline{2}, \underline{3}, \underline{5}, \underline{6}, \underline{7}, \underline{11}, \underline{13}, \underline{15}\}$

Рис. 3.3.

Реализация алгоритма сортировки выбором на языке C# выглядит следующим образом:

```
public void SortSelection()
{
    for (int i = 0; i < A.Length - 1; i++)
    {
        int imin = i;
        for (int j = i + 1; j < A.Length; j++) // в этом
цикле ищем минимальный элемент
            if (A[j] < A[imin]) imin = j;
        if (i != imin)
        {
            int tmp = A[imin]; // обмен местами мин.
элемента с первым
            A[imin] = A[i]; // из оставшейся - не
отсортированной
            A[i] = tmp; // части массива
        }
    }
}
```

Особенности сортировки выбором следующие:

- Во всех случаях сложность $O(N^2)$. Да, к сожалению, найдя минимальный элемент в подмассиве мы ничего не можем сказать о расположении остальных элементов. На каждой итерации мы должны производить поиск снова. Количество операций в поиске на первой итерации N , на второй — $N - 1$ и так далее. Сумма этих значений и даёт $O(N^2)$.

- Алгоритм можно реализовать и в устойчивом, и в неустойчивом режиме. Если при поиске минимума мы остановимся на самом первом из минимальных элементов, то, очевидно, обменов, нарушающих устойчивость, производиться не будет. В противном случае алгоритм окажется неустойчивым.

- Число дополнительных переменных не зависит от размера (*in-place*)

- Рекордно маленькое по сравнению с другими алгоритмами сортировки количество операций обмена $O(N)$. Это может пригодиться, в частности, для сортировок массивов с очень большими элементами. Если данные, которые мы сортируем, можно сравнивать очень быстро, но для их перемещения требуется большое время, то этот алгоритм оказывается одним из кандидатов.



В созданном проекте напишите метод, сортирующий данный целочисленный массив с помощью алгоритма сортировки выбором.

Протестируйте метод для массивов разных размеров.

3.2.5 Нахождение k -й порядковой статистики



k -й порядковой статистикой массива называется k -й по упорядочиванию величины элемент массива.

Операция упорядочивания в этом определении явным образом не определяется, поэтому упорядочивать мы можем по любой удобной нам операции. Минимальный элемент массива — 1-я порядковая статистика при использовании традиционной операции $<$. Максимальный элемент при использовании этой же операции будет иметь порядковую статистику N .

Медиана — «средний» по величине элемент. Примерно половина элементов не больше медианы, примерно половина не меньше, точное значение и точное определение зависит от чётности множества. Обратите внимание, что медиана — это не среднее значение! Например, для множества $\{1, 1, 1, 1, 1, 10\}$ медиана равна 1, а среднее значение — 2.5.

Легко ли найти k -ю порядковую статистику? Рассмотрим частные случаи.

$k=1$ Нахождение минимума — очевидно, что сложность $O(N)$.

$k=2$ Нахождение второго по величине элемента. Простой способ: хранить значения двух элементов, минимального и второго по величине. Тогда при обработке следующего элемента возможны три варианта: он не больше минимального, он больше минимального, но не больше второго элемента, он больше второго элемента. В первом случае он становится минимальным, а старое значение минимума отправляется во второй элемент. Во втором случае заменяется только значение второго элемента. Ну а в третьем случае ничего делать не надо. Каждая итерация требует до двух сравнений.

$k=3$ Неужели придётся повторять алгоритм случая $k=2$, расширив его на три сравнения? Тогда потребуется три переменные.

Произвольное k . Требуется ли использовать $O(k)$ памяти и тратить $O(k)$ операций на одну итерацию?

Оказывается, наивный алгоритм не столь уж хорош и алгоритм нахождения k -й порядковой статистики методом разделяй и властвуй значительно эффективнее:

1. Выбираем случайным образом элемент v массива S .
2. Разобьём массив на три подмассива S_l , элементы которого меньше, чем v ; S_v , элементы которого равны v и S_r , элементы которого больше, чем v .
3. Введём функцию $\text{Selection}(S, k)$, где S — массив, а k — номер порядковой статистики.

$$\text{Selection}(S, k) = \begin{cases} \text{Selection}(S_l, k), & \text{если } k \leq |S_l| \\ v, & \text{если } |S_l| < k \leq |S_l| + |S_v| \\ \text{Selection}(S_r, k - |S_l| - |S_v|), & \text{если } k > |S_l| + |S_v| \end{cases}$$

Под операцией $|S|$ имеется в виду операция получения числа элементов множества S .

Проиллюстрируем алгоритм на конкретных значениях.

Пусть нужно найти $k = 6$ порядковую статистику в массиве $S = \{10, 6, 14, 7, 3, 2, 18, 4, 5, 13, 6, 8\}$. Выбираем произвольный элемент. Пусть генератор случайных чисел указал на элемент со значением 8. Этот элемент будем называть ведущим (pivot). Ведущий элемент разбил исходный массив на три подмассива: $S_l = \{6, 7, 3, 2, 4, 5, 6\}$ с размером $|S_l| = 7$, $S_v = \{8\}$ с размером $|S_v| = 1$ и $S_r = \{10, 14, 18, 13\}$ с размером $|S_r| = 4$.

Так как осуществляется поиск порядковой статистики 6, а размер первого массива (множества, элементы которого меньше v) равен 7, то очевидно, что искомое находится в первом множестве; $k < |S_l| \rightarrow$ первый случай.

Уменьшаем размер массива, в котором производим поиск и выполняем вторую итерацию. $S = \{6, 7, 3, 2, 4, 5, 6\}$. Пусть случайным образом выбран

элемент 5. $S_l = \{3, 2, 4\} \mid S_l \mid = 3$. $S_v = \{5\} \mid S_v \mid = 1$. $S_r = \{6, 7, 6\} \mid S_r \mid = 3$. $k > \mid S_l \mid + \mid S_v \mid \rightarrow$ третий случай.

Третий проход: $S = \{7, 6\}$. Выбран произвольный элемент 6. $S_l = \{\} \mid S_l \mid = 0$. $S_v = \{6\} \mid S_v \mid = 1$. $S_r = \{7\} \mid S_r \mid = 1$. $\mid S_l \mid < k \leq \mid S_l \mid + \mid S_v \mid \rightarrow$ второй случай.

Ответ: 6

Разобранный алгоритм относится к классу алгоритмов «разделяй и властвуй», следовательно, к нему можно применить основную теорему о рекурсии.

Вначале зададимся вопросом, а что же будет, если каждый раз выбор элемента даст уменьшение подзадачи в 2 раза? Тогда

$$T(N) = T\left(\frac{N}{2}\right) + O(N)$$

Ясно, что после разбиения на подзадачи только одна из оставшихся будет интересна, поэтому количество подзадач $a = 1$. В идеальном варианте каждая новая подзадача меньше задачи в 2 раза, то есть $b = 2$. Операция «перетасовки» массива, отправляющая все элементы, меньше ведущего слева от него, а все элементы, больше ведущего, справа от него, имеет сложность $O(N)$. Коэффициент $d = 1$.

Так как $\log_b a = \log_2 1 = 0 < 1$, то в действие вступает первая ветвь основной теоремы о рекурсии, следовательно $T(N) = O(N)$. Возможно, результат покажется довольно неожиданным, но вспомним, что сумма геометрической прогрессии с множителем $1/2$ стремится к удвоенному первому элементу.

Худший случай наступает, если при выборе элемента, каждый раз оказывается, что либо $\mid S_l \mid = 0$, либо $\mid S_r \mid = 0$, то есть, если ведущим элементом станет наименьший или наибольший элемент массива. Тогда

$$T(N) = N + (N - 1) + \dots = \theta(N^2)$$

Таким образом, появился алгоритм, который в среднем способен находить k -ю порядковую статистику за время $O(N)$, или, как говорят, за

линейное время, но в маловероятных худших случаях его сложность может составлять $O(N^2)$.



В созданном проекте напишите метод, находящий в заданном целочисленном массиве k -порядковую статистику по имеющемуся k .

3.2.6 Быстрая сортировка

Самым популярным, на сегодняшний день, является алгоритм быстрой сортировки (или алгоритм сортировки Хоара). По сути, он практически полностью повторяет алгоритм поиска k -й порядковой статистики. В его основе лежит то же самое разбиение массива ведущим элементом на два подмассива и рекурсивная сортировка левой и правой частей. Хотелось бы ведущий элемент сделать как можно ближе к медианному, но такой элемент так просто не найти. Рассмотрим алгоритм на примере массива $S = \{10, 5, 14, 7, 3, 2, 18, 4, 5, 13, 6, 8\}$.

Разделение 1. Ведущий элемент равен 8 (рис. 3.4).

$$S_1l = \{5, 7, 3, 2, 4, 5, 6, 8\}, S_1r = \{10, 14, 18, 13\}$$

$$S_1 = \underbrace{\{5, 7, 3, 2, 4, 5, 6, 8\}}_{\text{left part}} \underbrace{\{10, 14, 18, 13\}}_{\text{right part}}$$

Рис. 3.4.

Разделение рекурсивное 2. Пусть ведущий элемент равен 5, $S_1 = \{5, 7, 3, 2, 4, 5, 6, 8\}$ (рис. 3.5).

$$S_2l = \{5, 3, 2, 4, 5\}, S_2r = \{7, 6, 8\}$$

$$S_2 = \underbrace{\{5, 3, 2, 4, 5\}}_{\text{left part}} \underbrace{\{7, 6, 8\}}_{\text{right part}}$$

Рис. 3.5.

И левая и правая части нового массива в конце концов оказались достаточно малы для того, чтобы избежать рекурсии. Пусть в нашем примере такая граница проходит для массивов длиной 5 или меньших. На этом этапе в

качестве «обычной» сортировки популярен алгоритм сортировки обменом (рис. 3.6).

$$S_2 = \{ \underbrace{2, 3, 4, 5, 6}, \underbrace{6, 7, 8} \}$$

Рис. 3.6.

S_{lr} после обычной сортировки станет равным $S_{lr} = \{10, 14, 14, 18\}$. Так как и левая и правая части массива на каждом из этапов сортируются на месте (а ведущий элемент на месте уже находился), то массив оказывается отсортирован.

С одной стороны, алгоритм напоминает алгоритм поиска k -й порядковой статистики, за тем исключением, что теперь интересны обе подзадачи и, следовательно, коэффициент порождения задач b в лучшем из случаев будет достигать двух. Следовательно, при выборе медианного элемента при следующих значениях параметров основной теоремы о рекурсии (количество подзадач $a = 2$, размер подзадачи $b = 2$, коэффициент консолидации $d = 1$) сложность будет следующей:

$$T(N) = T\left(\left\lceil \frac{N}{2} \right\rceil\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + \theta(N)$$

$$\log_b a = \log_2 2 = 1 \rightarrow T(N) = O(N \log N).$$

Если каждый раз при выборе ведущего элемента будет выбираться минимальный или максимальный элемент, то сложность алгоритма попадёт в зону, занятую примитивными алгоритмами с квадратичной сложностью.

Особенности алгоритма быстрой сортировки:

- Все операции могут производиться без требования дополнительных массивов, поэтому сортировка может проводиться на месте.

- Сложность в наихудшем случае $O(N^2)$, но вероятность такого события весьма мала. Если требуется обязательно использовать именно этот алгоритм и абсолютно неприемлема даже минимальная вероятность квадратичного времени, стоит больше уделить внимание вопросу выбора ведущего элемента.

– В прямолинейной реализации глубина рекурсивных вызовов может достигать N , что потребует $O(N)$ для стека вызовов. Это тоже часто бывает неприемлемо.

– Сложность в среднем $O(N \log N)$.

Реализация алгоритма быстрой сортировки на языке C# представлена ниже:

```
static void HoareSort(int[] array, int start, int end)
{
    if (end == start) return;
    var pivot = array[end];
    var storeIndex = start;
    for (int i = start; i <= end - 1; i++)
        if (array[i] <= pivot)
        {
            var t = array[i];
            array[i] = array[storeIndex];
            array[storeIndex] = t;
            storeIndex++;
        }

    var n = array[storeIndex];
    array[storeIndex] = array[end];
    array[end] = n;
    if (storeIndex > start) HoareSort(array, start, storeIndex - 1);
    if (storeIndex < end) HoareSort(array, storeIndex + 1, end);
}

static void HoareSort(int[] array)
{
    HoareSort(array, 0, array.Length - 1);
}
```



В созданном проекте напишите метод, реализующий алгоритм быстрой сортировки.

3.2.7 Сортировка слиянием

Алгоритмы нахождения k -статистики и быстрой сортировки основаны на операции *выборки* – нахождения k -го минимального элемента в массиве. Ещё один алгоритм сортировки основан на операции, противоположной операции *выборки* – объединении уже отсортированных массивов. Нетрудно убедиться,

что при наличии двух отсортированных массивов примерно одинакового размера их объединение в отсортированный массив можно произвести за $O(N)$, где N – длина образовавшегося массива. Объединение отсортированных массивов называют *двухпутевым* слиянием. *Декомпозиция* — разделение массива на подмассивы, весьма простая операция. По сути она может заключаться в выборе границы между подмассивами.

Имея эти операции, можно начинать сортировку массива $S = \{10, 5, 14, 7, 3, 2, 18, 4, 5, 13, 6, 8\}$. Первая декомпозиция (проводимая за $O(1)$) даёт подмассивы $S_l = \{10, 5, 14, 7, 3, 2\}$ и $S_r = \{18, 4, 5, 13, 6, 8\}$. Декомпозиция для левой половины даёт подмассивы $S_{l_l} = \{10, 5, 14\}$ и $S_{l_r} = \{7, 3, 2\}$. Установив порог перехода на обычную сортировку в три элемента, после данной декомпозиции его достигаем, и после сортировки подмассивы становятся равными $S_{l_{ll}} = \{5, 10, 14\}$ и $S_{l_{lr}} = \{2, 3, 7\}$. Их слияние даст полностью отсортированный массив $S_l = \{2, 3, 5, 7, 10, 14\}$. Проведя подобные операции с правой частью массива, получим отсортированную правую половину $S_r = \{4, 5, 6, 8, 13, 18\}$, после чего очередная операция слияния даст полностью отсортированный массив $S = \{2, 3, 4, 5, 6, 7, 8, 10, 13, 14, 18\}$.

Реализация алгоритма на языке C# представлена ниже:

```
static int[] temporaryArray;

static void Merge(int[] array, int start, int middle, int end)
{
    var leftPtr = start;
    var rightPtr = middle + 1;
    var length = end - start + 1;
    for (int i = 0; i < length; i++)
    {
        if (rightPtr > end || (leftPtr <= middle && array[leftPtr]
< array[rightPtr]))
        {
            temporaryArray[i] = array[leftPtr];
            leftPtr++;
        }
        else
        {
            temporaryArray[i] = array[rightPtr];
            rightPtr++;
        }
    }
}
```

```

        }
    }
    for (int i = 0; i < length; i++)
        array[i + start] = temporaryArray[i];
}

static void MergeSort(int[] array, int start, int end)
{
    if (start == end) return;
    var middle = (start + end) / 2;
    MergeSort(array, start, middle);
    MergeSort(array, middle + 1, end);
    Merge(array, start, middle, end);
}

static void MergeSort(int[] array)
{
    temporaryArray = new int[array.Length];
    MergeSort(array, 0, array.Length - 1);
}

static Random random = new Random();

```

В данном алгоритме наблюдается классический пример работы принципа разделяй и властвуй – обработка различных частей массива совершенно независима, следовательно, работает основная теорема о рекурсии.

$$T(N) = T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + \Theta(N)$$

- Количество подзадач всегда равно двум $a = 2$.
- Размер подзадачи всегда близок к половине основной $b = 2$
- Операция слияния имеет сложность $O(N)$, коэффициент степени при операции $d = 1$. Мы попадаем на ветку $\log_b a = \log_2 2 = 1$, следовательно, $T(N) = O(N \log N)$.

Особенности алгоритма сортировки слиянием:

- Операция декомпозиции может быть простой, но не имеется простых алгоритмов слияния двух отсортированных массивов в один без использования дополнительной памяти. Обычно требует добавочно $O(N)$ памяти.

– Интересный и полезный факт: сложность не зависит от входных данных и всегда равна $O(N \log N)$.



В созданном проекте напишите метод, реализующий алгоритм сортировки слиянием. Протестируйте все реализованные методы сортировки на разных по длине массивах и сравните результаты. Составьте таблицу по использованию времени и памяти каждым алгоритмом в зависимости от размера массива.

3.3 Сортировки с линейным временем выполнения

3.3.1 Сортировка подсчетом

Существуют ли алгоритмы, выполняющие сортировку со сложностью меньше $O(N \log N)$? Для общего случая нет. Но для отдельных типов данных используя свойства ключей можно использовать более эффективные алгоритмы.

Пусть множество значений ключей дискретно и ограничено

$$D(K) = \{K_{min}, \dots, K_{max}\}.$$

Тогда, при наличии добавочной памяти количеством $|D(K)|$ ячеек, сортировку можно произвести за $O(N)$.

Продemonстрируем это, для примера отсортировав массив $S = \{10, 5, 14, 7, 3, 2, 18, 4, 5, 13, 6, 8\}$. Пусть заранее известно, что значения массива – натуральные числа, которые не превосходят 20.

Этап 1. Создаем массив $F[1..20]$, содержащий вначале нулевые значения (рис. 3.7). Каждый элемент массива F в будущем будет содержать количество использований элемента с таким значением в исходном массиве.

$$F = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Рис. 3.7.

Этап 2. Проходим по массиву S . $S_1 = 10$; $F_{10} \leftarrow F_{10} + 1$. $S_2 = 5$; $F_5 \leftarrow F_5 + 1$ $S_{12} = 5$; $F_5 \leftarrow F_5 + 1$. Получим следующий массив (рис. 3.8).

$$F = \begin{bmatrix} 0 & 1 & 1 & 1 & 2 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Рис. 3.8.

Этап 3. Проходим по ненулевым элементам F . Так как второй элемент массива F содержит число 1, то это число — количество элементов, равных 2, в отсортированном массиве. 5-й элемент массива F содержит число 2, следовательно, в отсортированный массив мы добавляем два таких элемента.

$$S = \{2, 3, 4, 5, 5, 6, 7, 8, 10, 13, 14, 18\}$$

Обратите внимание, что в этом алгоритме для размещения элементов отсортированного массива не использованы операций сравнения, а просто подсчитано количество необходимых элементов, используя их значение в качестве *ключа* в другом массиве. К сожалению, данный алгоритм работает только с целыми числами.

Особенности сортировки подсчетом:

- Ключи должны быть перечислимы и могут использоваться в качестве индекса для вспомогательного массива счётчиков.
- Пространство значений ключей должно быть ограниченным.
- Требуется дополнительная память $O(|D(K)|)$ для размещения счётчиков.
- Сложность алгоритма определяется, исходя из сложности этапов: на первом этапе обнуление массива в $|D(K)|$ элементов требует $O(|D(K)|)$; на втором этапе, подсчёта, требуется $O(N)$ операций; третий этап — повторный проход по массиву счётчиков, что даёт нам $O(|D(K)|)$. Итого:

$$T(N) = O(|D(K)|) + O(N) + O(|D(K)|) = O(|D(K)|) + O(N).$$

3.3.2 Цифровая сортировка

Цифровая (или поразрядная) сортировка является первой автоматизированной промышленной сортировкой. Её история идёт от обработки результатов переписи населения конца 19-го начала 20-го веков. Основным хранителем информации были перфокарты, на которых цифры

пробивались в определённых позициях. Электромеханическое устройство могло распределять поступающие перфокарты на несколько выходных карманов, в зависимости от значения пробивки в заданной позиции.

Давайте воспользуемся их примером и попробуем отсортировать множество трёхзначных чисел без использования операций сравнения и без чрезмерных затрат памяти на массив счётчиков (этот массив для выбранных нами чисел должен был бы содержать 1000 элементов). Воспользуемся усложнённым вариантом сортировки подсчётом — поразрядной сортировкой.

Разобьём ключ на фрагменты — разряды и представим его как массив фрагментов. Все ключи должны иметь одинаковое количество фрагментов. Например: ключ 375 можно разбить на 3 фрагмента {3, 7, 5}, и ключ 5 — тоже на 3 {0, 0, 5}.

В качестве примера попробуем отсортировать массив $S = \{153, 266, 323, 614, 344, 993, 23\}$ при разбиении на 3 фрагмента.

Этап 1. { {1, 5, 3}, {2, 6, 6}, {3, 2, 3}, {6, 1, 4}, {3, 4, 4}, {9, 9, 3}, {0, 2, 3} }

Рассматривая последний фрагмент, как ключ, устойчиво отсортируем фрагменты методом подсчёта.

{ {1, 5, 3}, {3, 2, 3}, {9, 9, 3}, {0, 2, 3}, {3, 4, 4}, {6, 1, 4}, {2, 6, 6} }

Этап 2. Теперь устойчиво отсортируем по второму фрагменту. { {6, 1, 4}, {3, 2, 3}, {0, 2, 3}, {3, 4, 4}, {1, 5, 3}, {2, 6, 6}, {9, 9, 3} }

Этап 3. И, наконец, по первому фрагменту. { {0, 2, 3}, {1, 5, 3}, {2, 6, 6}, {3, 2, 3}, {3, 4, 4}, {6, 1, 4}, {9, 9, 3} }

При размере исходного массива N и размере вспомогательного массива 10 нам потребовалось провести всего три сортировки подсчётом, что даёт сложность в $3O(N) = O(N)$.

Возвращаясь к нашим предкам и перфокартам, посмотрим, что же они делали. Необходимым условием было размещение сортируемых ключей в одних и тех же позициях перфокарты (а перфокарта могла содержать от 45 до

80 позиций). Стопку сортируемых перфокарт помещали во входной карман, установив переключатель на последнюю цифру ключа. После нажатия на кнопку начиналась операция сортировки: перфокарты, содержащие в заданной позиции ноль, попадали в один приёмный карман, цифру один — в другой и так далее. Затем работник, не меняя порядка внутри перфокарт в кармане (устойчивая сортировка), собирал перфокарты в нужном порядке, получая тем самым частично отсортированную по ключу стопку. Затем переключатель устанавливался на предпоследнюю цифру ключа, ... После последнего прохода собранная стопка содержала полностью отсортированную последовательность.

Особенности поразрядной сортировки:

- Требуется ключи, которые можно трактовать как множество перечислимых фрагментов.
- Требуется дополнительной памяти $O(|D(K_i)|)$ на сортировку фрагментов.
- Сложность постоянна и равна $O(N \cdot |D(K_i)|)$.

Для практической реализации можно сделать замечание: операция разделения на фрагменты в виде десятичных цифр числа на современных компьютерах чрезвычайно медленная, так как требует операций целочисленного деления и нахождения остатков. Гораздо быстрее использовать разделение на фрагменты по набору битов или байтов. В этом случае для выделения фрагментов будет достаточно побитовых операций.

3.3.3 Карманная сортировка

Карманная (блочная) сортировка – это алгоритм сортировки, в котором сортируемые элементы распределяются между конечным числом отдельных карманов так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше), чем в предыдущем. Каждый блок затем сортируется отдельно, либо рекурсивно тем же методом, либо другим. Затем элементы помещаются обратно в массив. Этот тип сортировки обладает линейным временем исполнения.

Данный алгоритм требует знаний о природе сортируемых данных, выходящих за рамки функций «сравнить» и «поменять местами». Основными недостатками данного алгоритма является то, что он сильно деградирует при большом количестве мало отличных элементов, или же на неудачной функции получения номера корзины по содержимому элемента.

Если входные элементы подчиняются равномерному закону распределения, то математическое ожидание времени работы алгоритма карманной сортировки является линейным. Это возможно благодаря определенным предположениям о входных данных. При карманной сортировке предполагается, что входные данные равномерно распределены на отрезке $[0, 1)$.

Идея алгоритма заключается в том, чтобы разбить отрезок $[0, 1)$ на n одинаковых карманов, и разделить по этим карманам n входных величин. Поскольку входные числа равномерно распределены, предполагается, что в каждый карман попадет небольшое количество чисел (рис. 3.9). Затем последовательно сортируются числа в карманах. Отсортированный массив получается путём последовательного перечисления элементов каждого кармана.

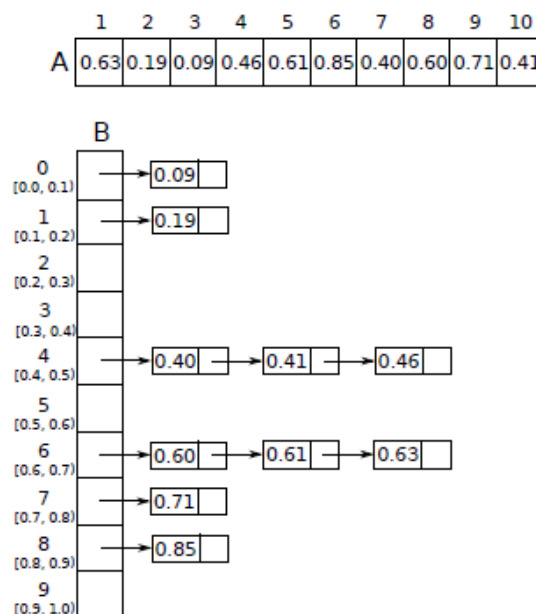


Рис. 3.9.

```

function bucket-sort(A, n) is
  buckets ← новый массив из n пустых элементов
  for i = 0 to (length(A)-1) do
    вставить A[i] в конец массива buckets[msbits(A[i], k)]
  for i = 0 to n - 1 do
    next-sort(buckets[i])
  return Конкатенация массивов buckets[0], ..., buckets[n-1]

```

На вход функции `bucket-sort` подаются сортируемый массив (список, коллекция и т.п.) `A` и количество блоков - `n`. Массив `buckets` представляет собой массив массивов, подходящих по природе к элементам `A`.

Функция `msbits(x,k)` тесно связана с количеством блоков - `n`, и, в общем случае, возвращает `k` наиболее значимых битов из `x` ($\text{floor}(x/2^{(\text{size}(x)-k)})$). В качестве `msbits(x,k)` могут быть использованы разнообразные функции, подходящие по природе сортируемым данным и позволяющие разбить массив. Функция `next-sort` также реализует алгоритм сортировки для каждого созданного на первом этапе блока. Рекурсивное использование `bucket-sort` в качестве `next-sort` превращает данный алгоритм в поразрядную сортировку. В случае `n = 2` соответствует быстрой сортировке (хотя и с потенциально плохим выбором опорного элемента).



В созданном проекте напишите методы, реализующие алгоритмы сортировки подсчетом, цифровой и карманной сортировки.

Протестируйте методы.

3.4 Внешняя сортировка

Сортировать и упорядочивать данные по какому-либо критерию — задача очень распространённая. Для данных, к которым идёт много запросов от различных клиентов, придумали удобный механизм — базы данных. У баз данных свои способы хранения, свои языки запросов и для наших задач их функциональность пока избыточна. Мы пока ограничимся существенно более простой задачей — сортировкой большого количества данных. Для таких данных можно выделить две основных проблемы:

– Для сортируемых данных недостаточно быстрой оперативной памяти и приходится пользоваться существенно более медленной внешней — HDD и SSD.

– Время сортировки превосходит приемлемые границы. Отсортировать миллиарды записей даже при очень хорошо реализованных алгоритмах сортировки — тяжёлая задача. Сосредоточимся на первой — сортировке при недостатке оперативной памяти. Такая сортировка носит название внешней.

3.4.1 Внешняя сортировка слиянием

В ситуации, когда помещение всех данных для обработки в быструю оперативную память невозможно, приходится использовать внешнюю память. Для сортировки достаточно использовать абстракцию «лента», предоставляющую следующие методы:

- create — создать новую пустую ленту и открыть её с возможностью записи.
- open — открыть существующую ленту исключительно для чтения.
- close — перестать использовать ленту.
- getdata — прочитать информацию с ленты.
- putdata — записать информацию на ленту.

Как известно, в алгоритме сортировки слиянием в оперативной памяти на фазе слияния каждый из входных массивов последовательно просматривается от начала до конца и выходной массив пишется строго последовательно. Это наталкивает на мысль, что подобный алгоритм мог бы быть полезным и для операций работы с лентами. Рассмотрим операцию *слияние* двух лент — *двухпутевое слияние*. Входными данными *двухпутевого слияния* являются две отсортированные ленты, выходными — другая отсортированная лента. Введём ещё один термин — *чанк* (*chunk*) — фрагмент данных, помещающихся в оперативной памяти.

Попробуем решить задачу прямолинейно, используя всю доступную оперативную память для обработки за раз максимально возможных фрагментов. Пусть исходная лента содержит 8 чанков (рис. 3.10).

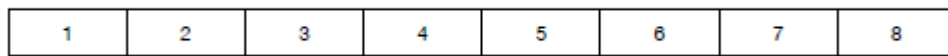


Рис. 3.10.

На первом этапе считывается первый чанк, сортируется внутренней сортировкой и отправляется на первую временную ленту (рис. 3.11).

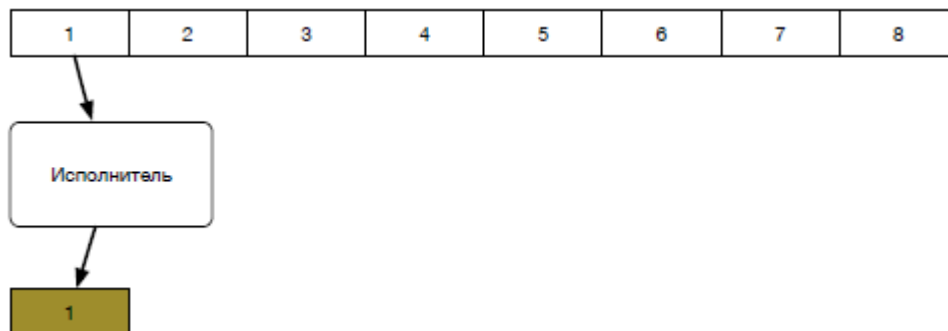


Рис. 3.11.

На втором этапе второй чанк сортируется и отправляется на вторую временную ленту (рис. 3.12).

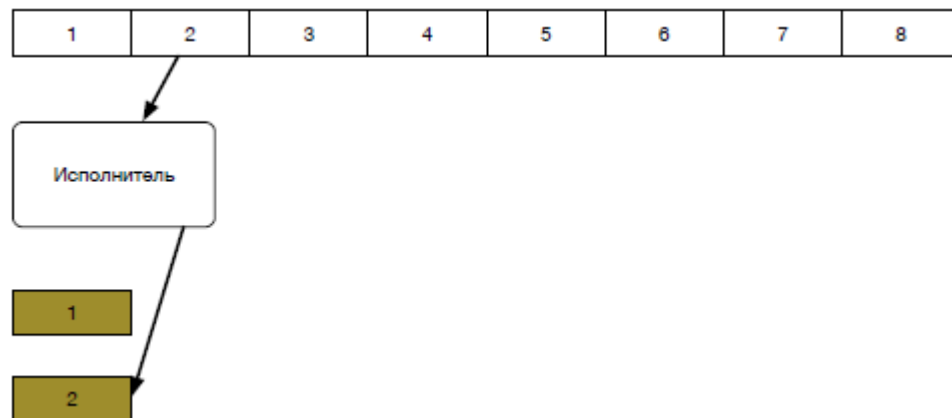


Рис. 3.12.

Третий этап – слияние. Сливаются первая и вторая временные ленты (рис. 3.13).

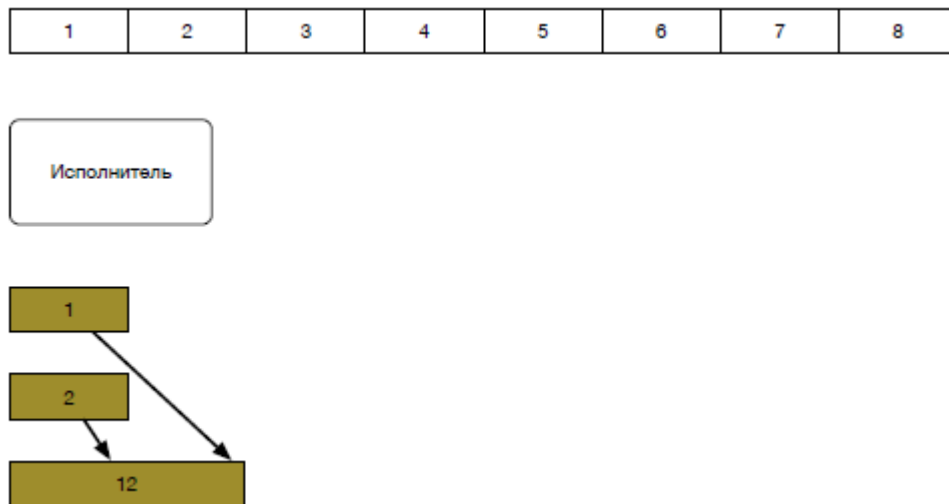


Рис. 3.13.

По аналогии, сортируются и выводятся на временные ленты 3-й и 4-й чанки (рис. 3.14).

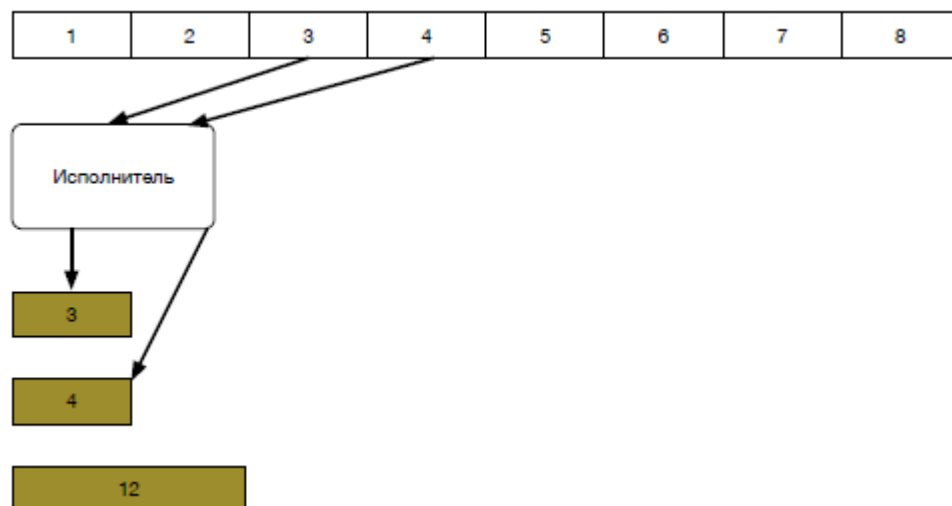


Рис. 3.14.

Аналогично временные ленты сливаются в ещё одну, четвёртую (рис. 3.15). Использовать меньшее количество лент здесь невозможно.

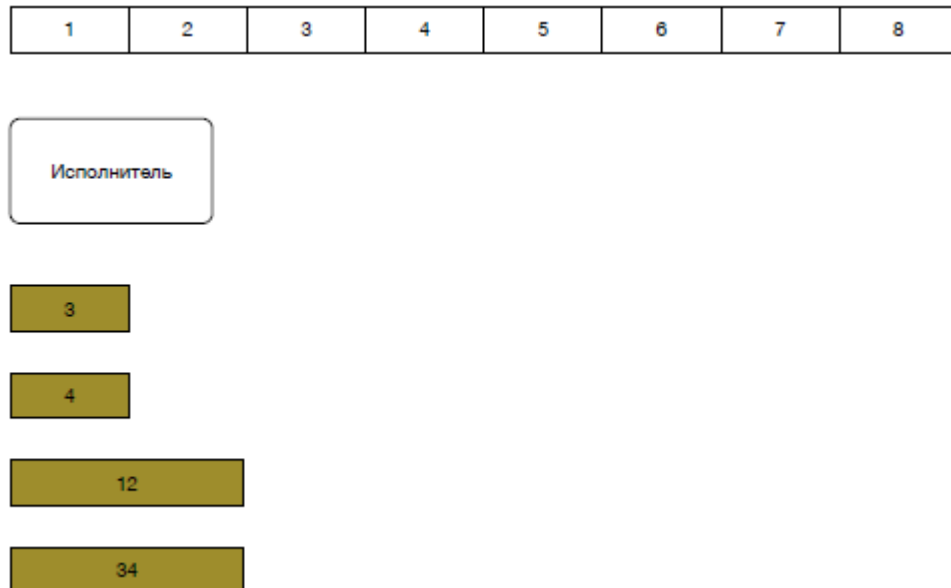


Рис. 3.15. Слияние

Сливаем ленты, содержащие чанки 12 и 34, в результате чего получаем ленту 1234. Для получения ленты 5678 требуется 4 временные ленты. Плюс лента 1234. Итого — 5 лент.

Нельзя сказать, что данный алгоритм оказался очень простым. При его реализации приходится тщательно отслеживать состояние каждой ленты и фиксировать, что на ней находится. Тем не менее, оценим его сложность.

- Внутренних сортировок в этом алгоритме столько же, сколько было чанков в исходных данных, K .

- Общая сложность внутренней сортировки

$$O((N/K) \times \log (N/K)) = O(N \log N).$$

- Сложность каждой из операций слияния — $O(N)$.

- Количество операций слияния $O(\log K)$.

- Общая сложность $O(N \log N)$.

- Сложность по количеству временных лент $O(\log K)$.

Сложность алгоритма с использованием внешней памяти не увеличилась и это обнадеживает. А можно ли упростить алгоритм до такой степени, чтобы он не использовал большое количество лент?

Попытаемся воспользоваться наблюдением, что при операции слияния дополнительной памяти не требуется, достаточно памяти для двух элементов. Попробуем произвести внешнюю сортировку без использования большого буфера и отказаться от использования чанков. Вместо этого будем использовать серии – неубывающие последовательности на ленте.

3.4.2 Сортировка сериями

Пусть имеется лента, представленная на рис. 3.16.

14, 4, 2, 7, 5, 9, 6, 11, 3, 1, 8, 10, 12, 13.

Рис. 3.16.

Заводим две вспомогательных ленты, в каждую из которых помещаем очередную серию длины 1 из входной ленты (рис. 3.17).

$\left\{ \begin{array}{cccccccc} 14, & 2, & 5, & 6, & 3, & 8, & 12 \\ 4, & 7, & 9, & 11, & 1, & 10, & 13 \end{array} \right.$

Рис. 3.17.

Формируем пары из первого элемента первой серии вместе с первым элементов второй серии, второго элемента первой серии и второго элемента второй серии и так далее (рис. 3.18). Каждую из пар упорядочиваем и результат попарно сливаем в исходный файл. Эту операцию можно совершить, имея в памяти ровно по одному элементу из каждой последовательности.

$\left\{ \begin{array}{cccccccc} 14, & 2, & 5, & 6, & 3, & 8, & 12 \\ 4, & 7, & 9, & 11, & 1, & 10, & 13 \end{array} \right.$

Рис. 3.18.

Инвариант операции слияния серий: совокупная последовательность является серией удвоенной длины (рис. 3.19).

4, 14, 2, 7, 5, 9, 6, 11, 1, 3, 8, 10, 12, 13.

Рис. 3.19.

Серии длины 2 попеременно помещаем на выходные ленты (рис. 3.20).

$$\underbrace{4, 14}, \underbrace{2, 7}, \underbrace{5, 9}, \underbrace{6, 11}, \underbrace{1, 3}, \underbrace{8, 10}, \underbrace{12, 13}.$$

$$\left\{ \begin{array}{l} \underbrace{4, 14}, \underbrace{5, 9}, \underbrace{1, 3}, \underbrace{12, 13} \\ \underbrace{2, 7}, \underbrace{6, 11}, \underbrace{8, 10}. \end{array} \right.$$

Рис. 3.20.

Снова каждую из серий попарно сливаем в исходную ленту (рис. 3.21).

$$\left\{ \begin{array}{l} \underbrace{4, 14}, \underbrace{5, 9}, \underbrace{1, 3}, \underbrace{12, 13} \\ \underbrace{2, 7}, \underbrace{6, 11}, \underbrace{8, 10}. \end{array} \right.$$

Рис. 3.21.

Длина полных серий в выходной ленте не меньше 4 (рис. 3.22). Только последняя серия может иметь меньшую длину.

$$\underbrace{2, 4, 7, 14}, \underbrace{5, 6, 9, 11}, \underbrace{1, 3, 8, 10}, \underbrace{12, 13}$$

Рис. 3.22.

Третий этап: формируем временные ленты сериями по 4 (рис. 3.23). И здесь, и в других подобных случаях только последняя серия на ленте может иметь длину меньше 4.

$$\underbrace{2, 4, 7, 14}, \underbrace{5, 6, 9, 11}, \underbrace{1, 3, 8, 10}, \underbrace{12, 13}$$

$$\left\{ \begin{array}{ll} \underbrace{2, 4, 7, 14}, & \underbrace{1, 3, 8, 10} \\ \underbrace{5, 6, 9, 11}, & \underbrace{12, 13} \end{array} \right.$$

Рис. 3.23.

Сливаем серии длины 4, что обеспечивает длину серий 8 (рис. 3.24).

$$\underbrace{2, 4, 5, 6, 7, 9, 11, 14}, \underbrace{1, 3, 8, 10, 12, 13}$$

Рис. 3.24.

Последний этап: разбивка на серии длины 8 с последующим слиянием (рис. 3.25, 3.26).

$$\left\{ \begin{array}{l} \underbrace{2, 4, 5, 6, 7, 9, 11, 14} \\ \underbrace{1, 3, 8, 10, 12, 13} \end{array} \right.$$

Рис. 3.25.

$$\underbrace{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}$$

Рис. 3.26.

Оценка сложности алгоритма сортировки сериями:

- За один проход примем операцию разбивки с последующим слиянием.
- На каждом проходе участвуют все элементы лент по два раза.
- Инвариант: длина серии после прохода k равна 2^k .
- Алгоритм завершается, когда длина серии оказывается не меньше N .
- Итого требуется $\log N$ проходов по файлам.
- Сложность алгоритма: $O(N \log N)$.
- Сложность по памяти: $O(1)$.
- Сложность по ресурсам: две временные ленты.

Несмотря на изящество алгоритма, его анализ показывает, что длина серии начинается от 1 и для полной сортировки требуется ровно $\lceil \log_2 N \rceil$ итераций, но первые итерации при этом производятся с небольшой длиной серии.

Но, есть возможность сократить число итераций, используя больше, чем 1 элемент памяти. Вспомним, что, хотя нас и ограничили по памяти, но какой-то её объём мы использовать можем. Тогда, улучшенный вариант алгоритма имеет следующий вид:

- Подбираем такое число k_0 , при котором серия длиной 2^{k_0} помещается в доступную память.

– Разбиваем исходную ленту на серии: считывается первый чанк длиной 2^{k_0} , сортируется внутренней сортировкой, пишется на первую ленту. Второй чанк после сортировки пишется на вторую ленту.

– После подготовки возвращаемся к алгоритму сортировки сериями.

Мы заменили k_0 проходов алгоритма одним. Следовательно, количество итераций сокращается на $k_0 - 1$, при этом сложность алгоритма не меняется, оставшись $O(N \log N)$, меняется лишь коэффициент амортизации. Тем не менее для практического применения улучшенный вариант может дать значительную выгоду.

Контрольные вопросы

1. В чем сущность задачи сортировки?
2. Что понимается под устойчивостью сортировки?
3. Что такое инверсия?
4. В чем состоит идея сортировки пузырьком?
5. Какова сложность алгоритма сортировки пузырьком?
6. Каковы особенности сортировки пузырьком?
7. В чем состоит идея сортировки вставками?
8. Как работает сортировка вставками? Какова ее сложность?
9. Каковы особенности сортировки вставками?
10. Поясните, почему сортировка Шелла в некоторых случаях будет работать быстрее пузырьковой сортировки? В каких случаях, сложность сортировки Шелла возрастает?
11. Как работает сортировка выбором?
12. Что такое k -порядковая статистика?
13. Как найти k -порядковую статистику?
14. Какова сложность нахождения k -порядковой статистики?
15. В чем состоит идея быстрой сортировки?
16. Назовите особенности быстрой сортировки? Какова ее сложность?

17. Как работает сортировка слиянием?
18. Какова сложность и особенности сортировки слиянием?
19. Что необходимо знать о наборе данных для применения сортировки подсчетом?
20. Как работает сортировка подсчетом?
21. Как работает поразрядная сортировка?
22. Какова сложность сортировки подсчетом и поразрядной сортировки?
23. Зачем нужна внешняя сортировка?
24. Поясните суть алгоритма внешней сортировки слиянием?
25. В чем особенность сортировки сериями?

Задания для самостоятельного выполнения

Задание №1. Максимальная тройка

Ограничение по времени: 1.5 секунд. Ограничение по памяти: 8 мегабайт

Имеется не более 1000000 целых чисел, каждое из которых лежит в диапазоне от -1000000 до 1000000. Найти максимально возможное значение произведений любых трех различных по номерам элементов массива.

Формат входных данных:

N

A1

A2

...

AN

Формат выходных данных:

MaxPossibleProduct

Задание №2. Сортировка по многим полям

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

В базе данных хранится N записей, вида $(Name, a_1, a_2, \dots, a_k)$ – во всех записях одинаковое число параметров. На вход задачи подается приоритет полей – перестановка на числах $1, \dots, k$ – записи нужно вывести по невозрастанию в соответствии с этим приоритетом. В случае, если приоритет полей таков: $3\ 4\ 2\ 1$, то это следует воспринимать так: приоритет значений из 3 колонки самый высокий, приоритет значений из колонки 4 ниже, приоритет значений из колонки 2 еще ниже, а приоритет значений из колонки 1 самый низкий.

Формат входных данных:

$N \leq 1000$

$k: 1 \leq k \leq 10$

$p_1\ p_2\ \dots\ p_k$ – перестановка на k числах, разделитель – пробел

N строк вида

$Name\ a_1\ a_2\ \dots\ a_k$

Формат выходных данных:

N строк с именами в порядке, согласно приоритету

Пример:

Ввод	Вывод
3	В
3	А
2 1 3	С
А 1 2 3	
В 3 2 1	
С 3 1 2	

Так как колонка под номером 2 самая приоритетная, то переставить записи можно только двумя способами: (A, B, C) и (B, A, C) . Следующий по приоритетности столбец – первый, и он позволяет выбрать из возможных перестановок только (B, A, C) . Так как осталась ровно одна перестановка, третий приоритет не имеет значения.

Задание №3. Оболочка

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

Имеется массив из N целочисленных точек на плоскости.

Требуется найти периметр наименьшего охватывающего многоугольника, содержащего все точки.

Формат входных данных:

N

$x_1 y_1$

$x_2 y_2$

...

$x_n y_n$

$5 \leq N \leq 500000$

$-10000 \leq x_i, y_i \leq 10000$

Формат выходных данных:

Одно вещественное число – периметр требуемого многоугольника с двумя знаками после запятой.

Задание №4. Очень быстрая сортировка

Ограничение по времени: 1.5 секунд. Ограничение по памяти: 512 мегабайт

Имеется рекуррентная последовательность A_1, A_2, \dots, A_N , строящаяся по следующему правилу:

$A_1 = K$

$A_{i+1} = (A_i \times M) \% (2^{32} - 1) \% L$

Требуется найти сумму всех нечетных по порядку элементов в отсортированной по неубыванию последовательности по модулю L .

Для входных данных

5 7 13 100

последовательность будет такой:

$\{7; 7 \times 13\%100 = 91; 91 \times 13\%100 = 83; 83 \times 13\%100 = 79; 79 \times 13\%100 = 27\}$, то есть $\{10; 91; 83; 79; 27\}$.

Отсортированная последовательность $\{7; 27; 79; 83; 91\}$.

Сумма элементов на нечетных местах = $(7 + 79 + 91) \% 100 = 77$.

Формат входных данных:

N K M L

$5000000 \leq N \leq 600000000, 0 \leq K, L, M \leq 2^{32} - 1$

Формат выходных данных:

RESULT

Задание №5. Внешняя сортировка

Ограничение по времени: 2 секунды. Ограничение по памяти: 2 мегабайта

В файле «input.txt» содержатся строки символов, длина каждой строки не превышает 10000 байт. Файл нужно отсортировать в лексикографическом порядке и вывести результат в файл «output.txt». Вот беда, файл занимает много мегабайт, а в Вашем распоряжении оказывается вычислительная система с очень маленькой оперативной памятью. Но файл должен быть отсортирован.

ГЛАВА 4. СПИСКИ И ДЕРЕВЬЯ

4.1 Структура данных «Список»

4.1.1 Понятие и виды списков

Список – структура данных, которая реализует абстракции:

- insertAfter — добавление элемента за текущим.
- insertBefore — добавление элемента перед текущим.
- insertToFront — добавление элемента в начало списка.
- insertToBack — добавление элемента в конец списка.
- find — поиск элемента.
- size — определение количества элементов.

Для реализации списков обычно требуется явное использование указателей.

```
class linkedListNode<T>
{ // Одна из реализаций
    T data; // Таких полей – произвольное число
    linkedListNode<T> next; // Связь со следующим
}
```

Внутренние операции создания элементов – через new.

```
linkedListNode<int> item = new linkedListNode<int>();
item.data = 5;
```

Существуют различные варианты представлений списков:

1. В линейном виде (рис. 4.1).

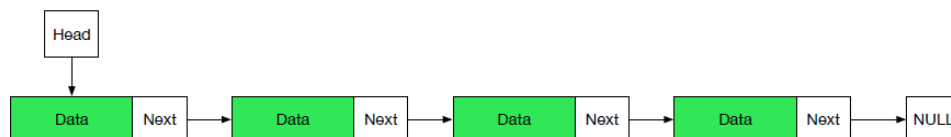


Рис. 4.1.

2. В виде кольца (рис. 4.2).

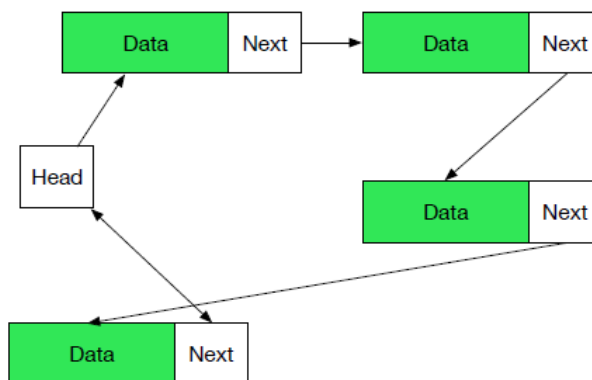


Рис. 4.2.

Стоимость основных операций представлена в табл. 4.1.

Табл. 4.1.

Операция	Время	Память
Вставка после	$O(1)$	$O(1)$
Вставка перед	$O(N)$	$O(1)$
Вставка в начало списка	$O(1)$	$O(1)$
Вставка в конец списка	$O(N)$	$O(1)$
Поиск элемента в списке	$O(N)$	$O(1)$
Определение размера списка	$O(N)$	$O(1)$



Создайте новый проект. В нем реализуйте класс `LinkedListNode`, реализующий структуру данных «Список».

4.1.2 Основные операции со списками

Рассмотрим основные операции со списками.

1. Создание списка из одного элемента. Для этого опишем метод создания узла списка:

```

static linkedListNode<T> list_createNode(T data)
{
    linkedListNode<T> result = new linkedListNode<T>();
    result.data = data;
    result.next = null;
    return result;
}
  
```

Создание списка выглядит следующим образом (рис. 4.3):

```

        linkedListNode<double> head =
        linkedListNode<double>.list_createNode(555.666);

```



Рис. 4.3.

2. Добавление элемента в конец списка.

```

        linkedListNode<double> oth =
        linkedListNode<double>.list_createNode(123.45);
        head.next = oth;

```

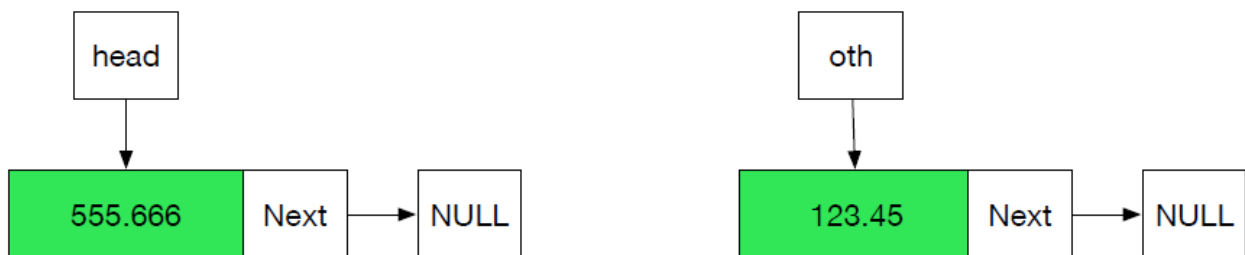


Рис. 4.4. Добавление элемента в конец списка (первый этап)

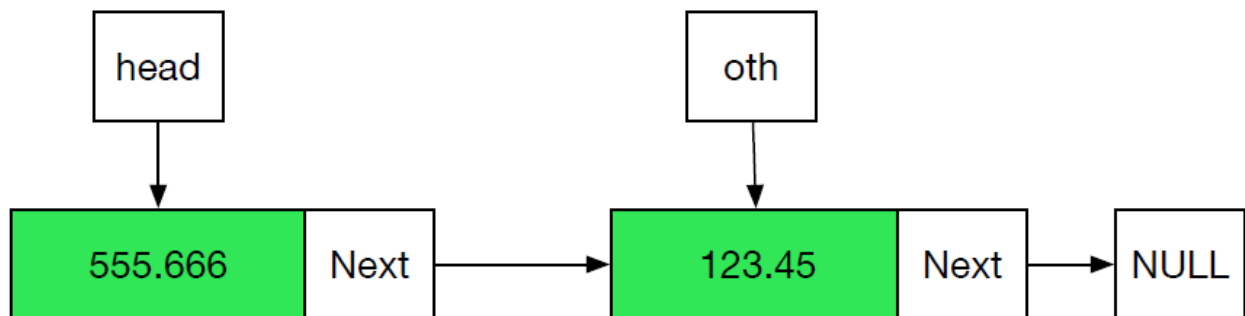


Рис. 4.5. Добавление элемента в конец списка (второй этап)

3. Добавление элемента в конец списка, состоящего из нескольких элементов. Сложность данной операции равна $O(N)$.

```

        linkedListNode<double> ptr = head;
        while (ptr.next != null)
        {
            ptr = ptr.next;
        }
        ptr.next = oth;

```

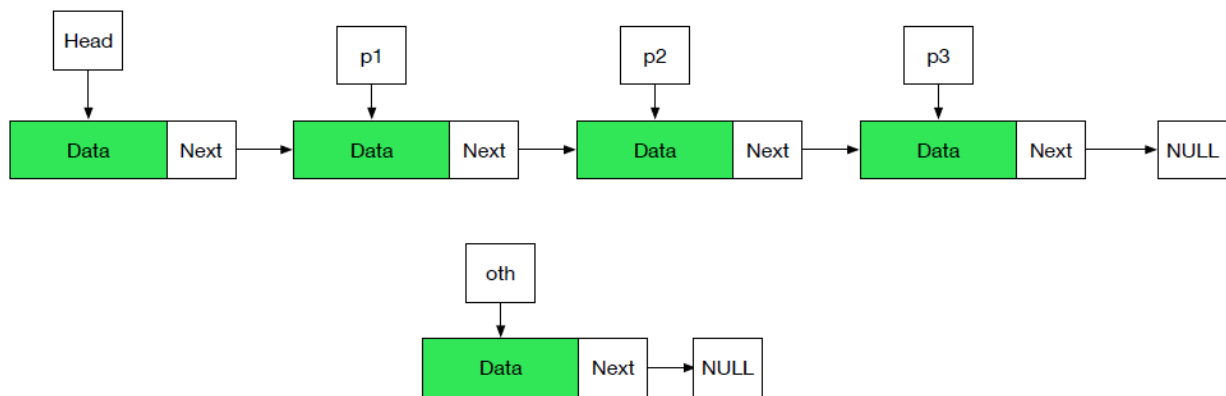


Рис. 4.6. Добавление элемента в конец списка (первый этап)

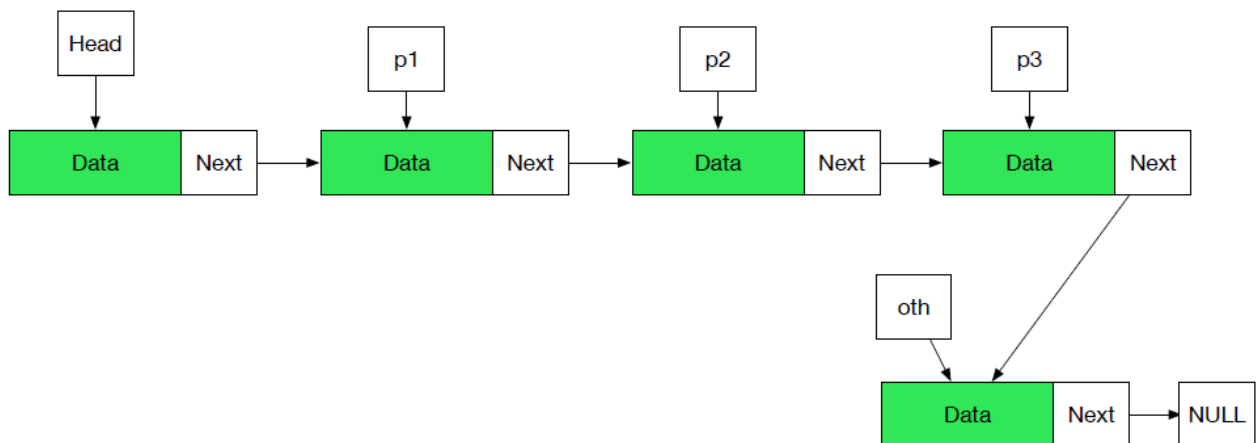


Рис. 4.7. Добавление элемента в конец списка (заключительное состояние после вставки)

4. Вставка за конкретным элементом (рис. 4.8).

```

linkedListNode<double> p1 =
linkedListNode<double>.list_createNode(76.24);
oth.next = p1.next;
p1.next = oth;

```

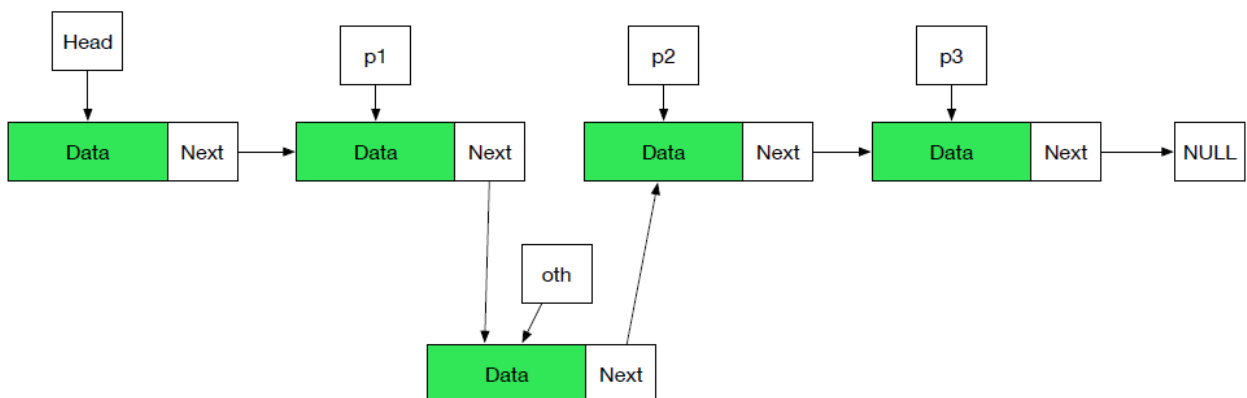


Рис. 4.8.

5. Вставка перед конкретным элементом.

```

linkedListNode<double> prt = head;
while (prt.next != p2)
{
    prt = prt.next;
}
oth.next = p2;
prt.next = oth;

```

6. Удаление элемента – является более сложной операцией, так как необходимо найти как удаляемый элемент, так и его предшественника (рис. 4.9), а затем переместить ссылку от предшественника к следующему за удаляемым элементу (рис. 4.10).

```

// поиск элемента p2
linkedListNode<double> ptr = head;
while (ptr.next != p2)
{
    ptr = ptr.next;
}
// ptr - предшественник p2
ptr.next = p2.next; // удаление p2

```

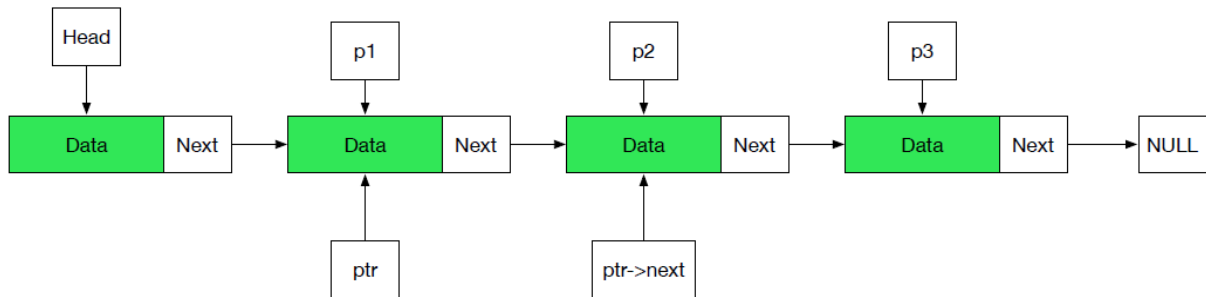


Рис. 4.9.

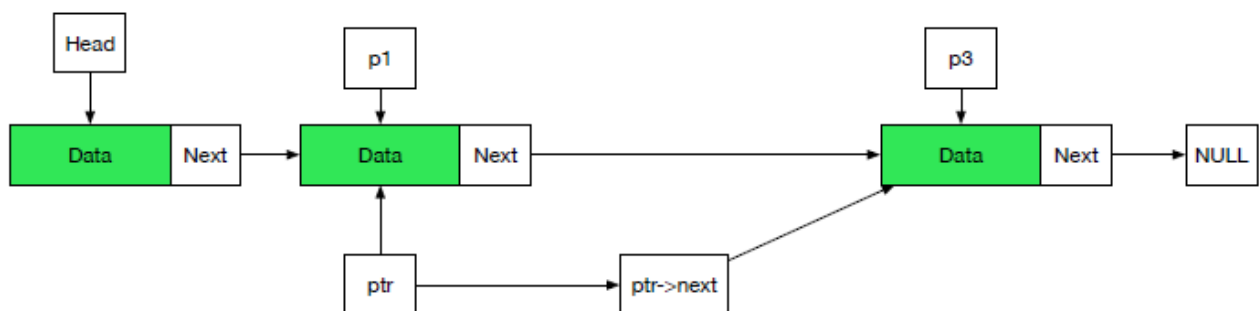


Рис. 4.10.

7. Определение размера списка. Такую операцию можно реализовать двумя способами: медленно через операцию walk до ссылки null, или быстро, поддерживая инвариант размера списка в структуре данных, что повлечет за собой изменение всех методов вставки и удаления.



Продолжайте работать в созданном проекте. Реализуйте все основные операции работы со списком. Реализуйте два варианта определения размера списка.

Выбранное представление списка удобно не для всех применений. Чтобы не потерять указатели, требуется всегда различать, работаем ли мы с головой списка или с другим элементом. Если в программе где-то сохраняется копия головы списка в отдельной переменной, то при операциях изменения головы списка копия становится неактуальной. Чтобы избежать этого, удобно иметь список с неизменной головой (рис. 4.11, 4.12). Такое представление упрощает реализацию за счёт одного дополнительного элемента.

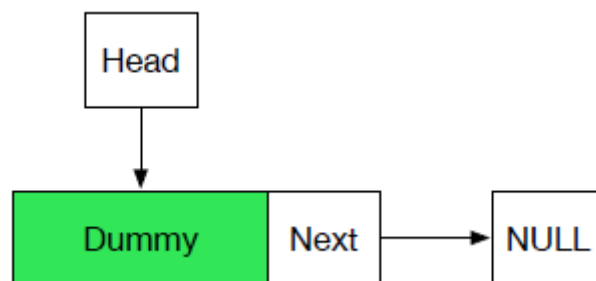


Рис. 4.11. Пустой список

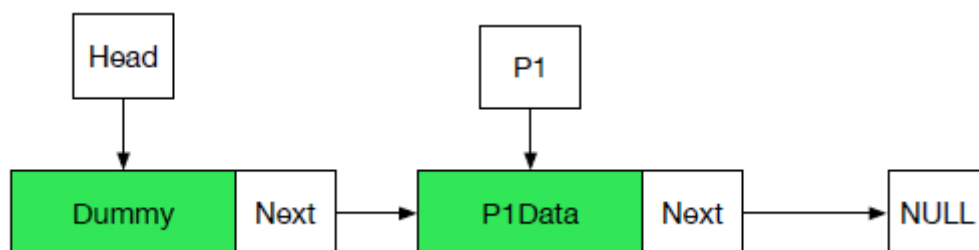


Рис. 4.12. Список, состоящий из одного элемента

Таким образом, систематизируем информацию по сложности основных операций со списками:

- Вставка элемента в голову списка — $O(1)$
- Вставка элемента в хвост списка — $O(N)$
- Поиск элемента — $O(N)$
- Удаление известного элемента — $O(N)$
- Вставка элемента ЗА заданным — $O(1)$

– Вставка элемента ПЕРЕД заданным — $O(N)$

4.1.3 Двусвязные списки

Есть ли возможность улучшить худшие случаи? На самом деле, их можно улучшить, если использовать двусвязные списки, т.е. такие списки, в которых каждый элемент содержит две ссылки: на предыдущий и следующий элемент (рис. 4.13).

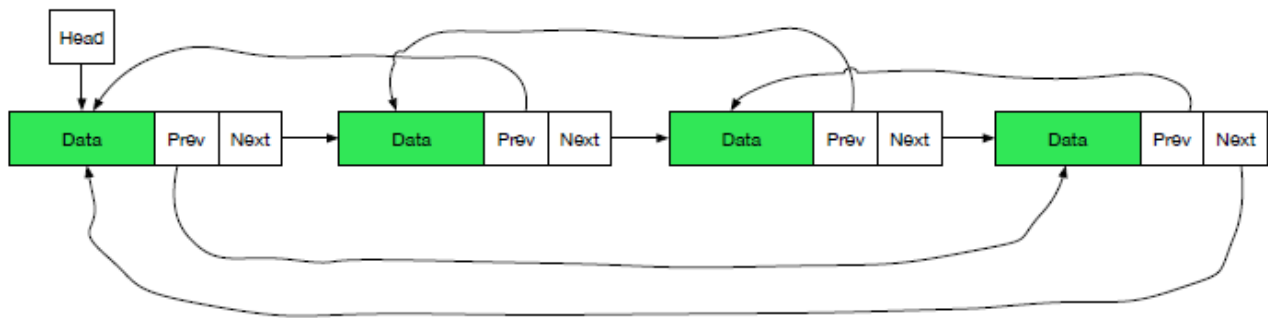


Рис. 4.13.

Для двусвязного списка сложность основных операций следующая:

- Вставка элемента в голову списка — $O(1)$
- Вставка элемента в хвост списка — $O(1)$
- Поиск элемента — $O(N)$
- Удаление заданного элемента — $O(1)$
- Вставка элемента ЗА заданным — $O(1)$
- Вставка элемента ПЕРЕД заданным — $O(1)$

Однако, при использовании двусвязных списков усложняются операции вставки и удаления. Для вставки элемента *oth* после *p1* необходимо выполнить следующие действия:

1. подготавливаем вставляемый элемент;
2. сохраняем указатель `s = p1.next;`
3. `oth.prev = p1;`
4. `oth.next = s;`
5. `s.prev = oth;`

```
6. p1.next = oth;
```

Для удаления элемента `p1` из списка:

```
1. сохраняем указатель s = p1.next;
```

```
2. s.prev = p1.prev;
```

```
3. p1.prev.next = s;
```

```
4. Освобождаем память элемента p1;
```

Списки в практическом применении обычно используют для представления быстро изменяющегося множества объектов.

Пример из математического моделирования: множество машин при моделировании автодороги. Они:

- появляются на дороге (вставка в начало списка)
- покидают дорогу (удаление из конца списка)
- перестраиваются с полосы на полосу (удаление из одного списка и вставка в другой).

Пример из системного программирования: представление множества исполняющихся процессов, претендующих на процессор. Представление множества запросов ввода/вывода. Важная особенность: лёгкий одновременный доступ от различных процессорных ядер.

Пример из системного программирования: одна из реализаций выделения/освобождения динамической памяти (`calloc/new/free/delete`).

- Вначале свободная память описывается пустым списком.
- Память в операционной системе выделяется страницами.
- При заказе памяти:
 - если есть достаточный свободный блок памяти, то он разбивается на два подблока, один из которых помечается занятым и возвращается в программу;
 - если нет достаточной свободной памяти, запрашивается несколько страниц у системы и создаётся новый элемент в конце списка (или изменяется старый).



Продолжайте работать в созданном проекте. Создайте новый класс, реализующий двусвязный список. Определите основные операции со списками для этого класса.

4.1.4 Абстракция «очередь»

С помощью списков легко реализовать абстракцию очередь, реализующую (помимо неизбежных операций создания и удаления) следующий набор операций:

- enqueue: добавить элемент в конец очереди;
- dequeue: извлечь с удалением элемент из начала очереди;
- empty: определить, пуста ли очередь.

Ещё одно популярное название этой абстракции — FIFO, First In First Out.

Эта абстракция, наряду с похожей на неё абстракцией стека (LIFO), будет широко использоваться в алгоритмах на графах.



Продолжайте работать в созданном проекте. Создайте новый класс, реализующий очередь. Определите операции enqueue, dequeue и empty для созданной очереди.

4.2 Структура данных «Дерево»



Древовидная структура (дерево) с базовым типом T – это либо пустая структура, либо узел типа T , с которыми связано конечное число древовидных структур, называемых поддеревьями.

Отличие деревьев от списков — наличие нескольких наследников. Если с узлом связано только два поддерева, то дерево называется двоичным или бинарным. Остальные обычно называются N -ричными.

4.2.1 Представление деревьев

Компьютерные представления деревьев обычно достаточно прямолинейны и подчиняются следующим соглашениям:

- Любое N-ричное дерево может представлять деревья меньшего порядка.
- Если потомка нет, соответствующий указатель равен NULL.
- Деревья 1-ричного порядка существуют и являются списками.

```
class Tree<T>
{
    Tree<T>[] children;
    T data;
    //...
}
```

Пример троичного дерева или дерева 3-порядка представлен на рис. 4.14.

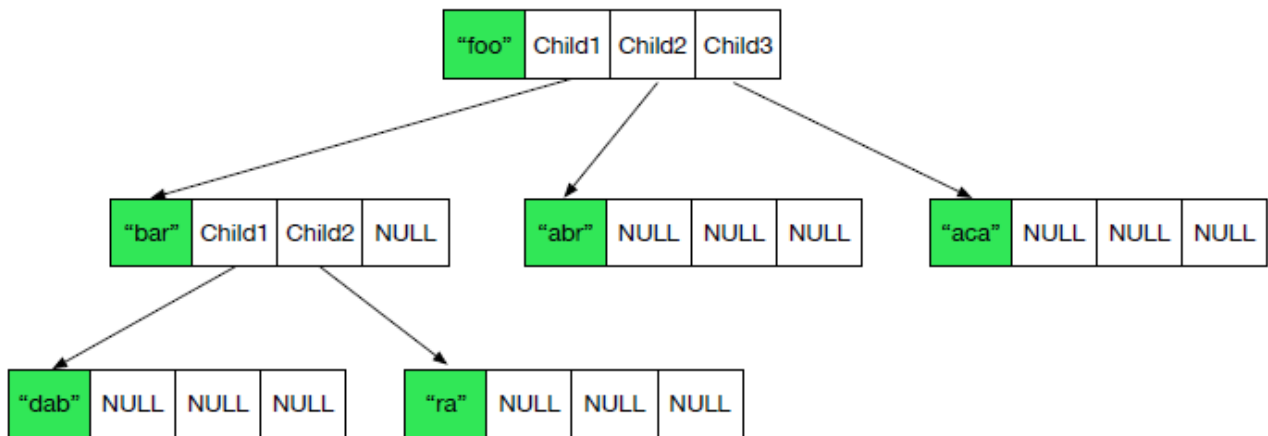


Рис. 4.14.

Узлы в деревьях часто неравнозначны и их делят на две группы:

- *Вершины*, не содержащие связей с потомками. Другое название — терминальные вершины или терминальные узлы.

- *Узлы*, содержащие связи с потомками.

Приведем еще несколько определений:

Родитель (parent) — узел, потомком которого является данный.

Дети (children) — потомки данного узла.

Братья (sibs) — узлы, имеющие одного родителя.

Глубина (depth) — длина пути от корня до узла $D_{\text{node}} = D_{\text{parent}} + 1$.

Создание конструктора для дерева выглядит следующим образом:

```

class Tree<T>
{
    Tree<T>[] children;
    T data;
    Tree(T init, int count)
    {
        children = new Tree<T>[count];
        for (int i=0; i<count; i++)
        {
            children[i] = null;
        }
        data = init;
    }
    //...
}

```

Чтобы построить дерево, показанное на рисунке, достаточно исполнить следующий код:

```

Tree<string> root = new Tree<string>("foo", 3);
root.children[0] = new Tree<string>("bar", 3);
root.children[1] = new Tree<string>("abr", 3);
root.children[2] = new Tree<string>("aca", 3);
root.children[0].children[0] = new Tree<string>("dab", 3);
root.children[0].children[1] = new Tree<string>("ra", 3);

```

Деревья — одна из популярнейших структур данных и их использование можно наблюдать в различных разделах компьютерных наук.

Указатели — это хорошо и красиво. Однако, их использование подразумевает заказ динамической памяти и её освобождение, что не является очень быстрой операцией. В некоторых применениях можно обойтись и без указателей, храня узлы в заранее заказанном массиве. Для этого все возможные узлы нумеруются, например, начиная с 0. Тогда для N-дерева для узла с номером K номера детей будут $K \cdot N + 1 \dots K \cdot N + N$. Например, для 2-дерева корневой узел будет иметь номер 0, узлы первого уровня ($D = 2$) — номера 1 и 2, второго ($D = 3$) — от 3 до 7.

Для двоичных деревьев узлы часто удобнее нумеровать с 1. Тогда нумерация и детей и родителей упрощается. Например, для родителя под номером 10 дети будут иметь номера 20 и 21 ($2K, 2K + 1$), а для ребёнка номер 35 родителем будет узел под номером 17 ($K/2$). Это особенно удобно, если

учесть, что операции умножения на 2 и деления на два можно производить с помощью побитовых операций сдвига.

Конечно, представление бинарного дерева в виде массива эффективно только для тех деревьев, которые всегда или почти всегда имеют двух потомков у каждого родителя. Мы будем называть такие деревья плотными.

Приведём пример неудачного выбора представления. Все современные компиляторы в какой-то момент времени представляют выражения языков программирования в виде деревьев. Оказывается, это представление удобно для проведения каких-либо преобразований программы, например, распространения констант. Разберем дерево для следующей операции:

$$x = 35 \times y - \sin(x + z)$$

Если дерево представлено в виде указателей, то расход памяти минимален (рис. 4.15).

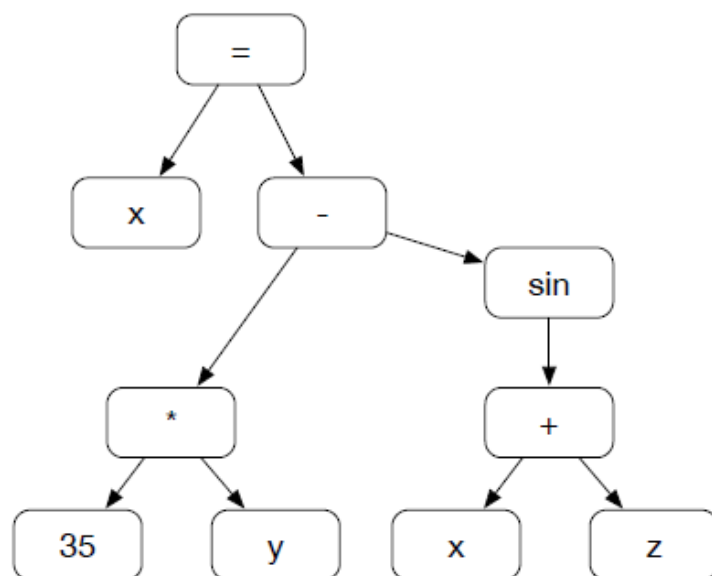


Рис. 4.15.

Если выбрать представление в виде массива, то для данного выражения нумерация узлов будет такой, как показано на рис. 4.16.

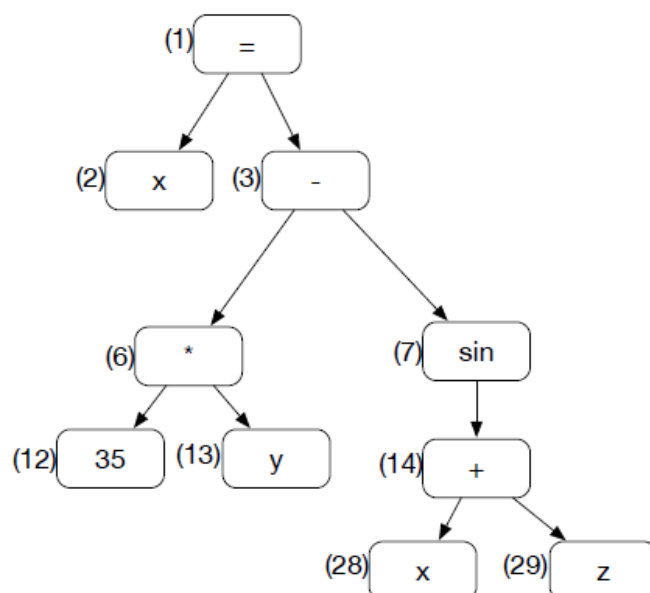


Рис. 4.16.

Представление в виде массива потребует значительного количества памяти. Количество требуемой памяти в таких массивах есть $O(2^{D_{\max}})$, что при разреженном дереве крайне невыгодно.



Продолжайте работать в созданном проекте. Создайте новый класс, реализующий дерево. Создайте класс-наследник «Бинарное дерево», у которого каждый узел имеет только два потомка. Создайте конструктор для класса двоичного дерева. Создайте произвольное бинарное дерево.

4.2.2 Обход деревьев

Дерево – рекурсивная по своей сущности структура данных и алгоритмы работы с деревьями часто рекурсивны. Одна из распространённых задач – посетить все узлы дерева (обойти его).

Для бинарного дерева порядок обхода определяется порядком обработки трёх сущностей – обработки левого потомка, правого потомка и самого узла. Таким образом существует $3!$ способов обхода бинарного дерева. На практике чаще всего применяют четыре основных варианта рекурсивного обхода: прямой, симметричный, обратный и обратно симметричный.

При прямом способе обхода сначала обрабатывают сам узел, затем рекурсивно посещают левого и правого потомков именно в этом порядке (рис. 4.17).

```
static void walk(BinaryTree<T> t)
{
    work(t);
    if (t.left != null) walk(t.left);
    if (t.right != null) walk(t.right);
}
```

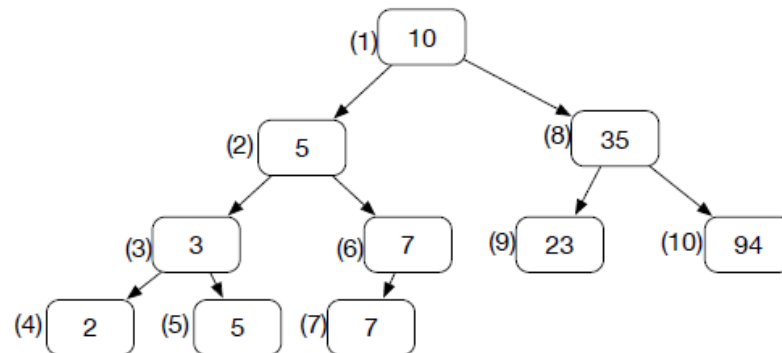


Рис. 4.17.

Симметричный способ обхода инициирует посещение сначала левого потомка, затем обработку самого узла, после чего посещается правый потомок (рис. 4.18).

```
static void walkSymmetry(BinaryTree<T> t)
{
    if (t.left != null) walkSymmetry(t.left);
    work(t);
    if (t.right != null) walkSymmetry(t.right);
}
```

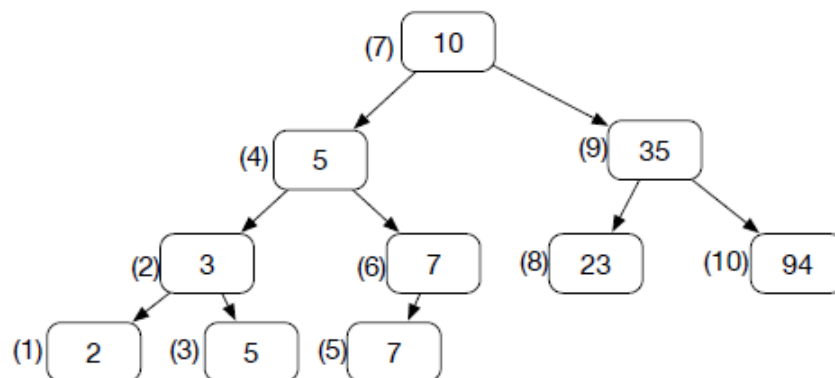


Рис. 4.18.

При обратном способ обхода сначала посещаются потомки, левый и правый, и только потом очередь доходит до самого узла (рис. 4.19).

```
static void walkReverse(BinaryTree<T> t)
{
    if (t.left != null) walkReverse(t.left);
    if (t.right != null) walkReverse(t.right);
    work(t);
}
```

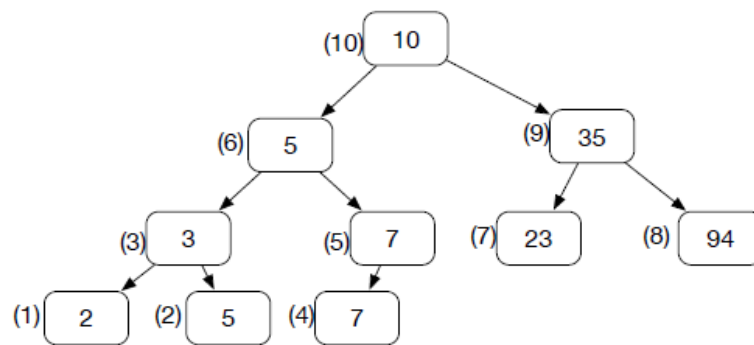


Рис. 4.19.



Продолжайте работать в созданном проекте. Реализуйте в классе бинарного дерева четыре метода обхода дерева (включая обратный симметричный). Самостоятельно напишите метод `work`, который выполняет какую-либо операцию (например, печать на консоль) с узлом дерева. Проверьте результаты.

4.3 Бинарная куча

4.3.1 Бинарная куча и абстракция приоритетная очередь



Полное бинарное дерево T_H высоты H есть бинарное дерево, у которого путь от корня до любой вершины содержит ровно H рёбер, при этом у всех узлов дерева, не являющимися листьями, есть и правый, и левый потомок (рис. 4.20).



Полное бинарное дерево T_H высоты H есть бинарное дерево, у которого к корню прикреплены левое и правое полные бинарные поддеревья T_{H-1}

высоты $H - 1$. По этому определению число узлов в дереве T_H есть $N = 2^{H+1} - 1$,
 $H = \log_2(N + 1)$,

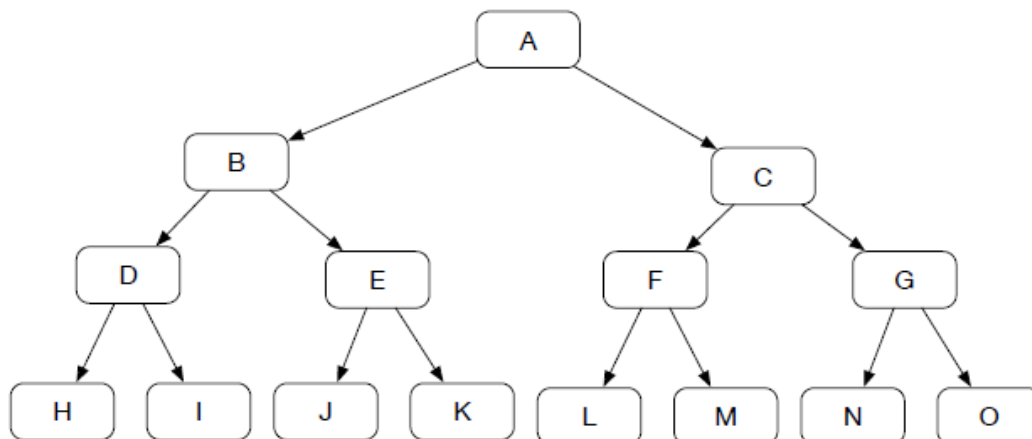


Рис. 4.20.

Мы уже знаем про абстракцию обычной очереди, когда первый поступивший в очередь элемент будет первым же извлечён. В ряде случаев такой стратегии недостаточно и требуется как-то упорядочивать поступающих в очередь, например, по их приоритету. Классический пример — очередь к врачу. Посетителя с сильной зубной болью могут принять вне очереди. А если таких больных несколько, то из них тоже образуется очередь. Проще считать, что очередь — одна, но у каждого посетителя свой приоритет.

ⓘ *Приоритетная очередь* (priority queue) — очередь, элементы которой имеют приоритет, влияющий на порядок извлечения. Первым извлекается наиболее приоритетный элемент.

На первый взгляд простейшим способом поддержания приоритетной очереди мог бы быть полностью упорядоченный массив. Однако, стоимость поддержания упорядоченности в массиве весьма велика. Например, обнаружить самый приоритетный элемент легко — достаточно отсортировать массив в нужном порядке, и он будет находиться в самом конце. Извлечение этого элемента займёт $O(1)$. А вот со вставкой будет намного хуже. Бинарным поиском мы обнаружим место вставки (это потребует сложности $O(\log N)$), а

что затем? А затем придётся сдвигать все элементы правее точки вставки вправо, что даст сложность $O(N)$.

Перед тем, как реализовывать приоритетную очередь, определимся с её абстракцией, то есть, с теми методами, которые необходимо реализовать.

Интерфейс абстракции *приоритетная очередь*:

- insert — добавляет элемент в очередь;
- fetchPriorityElement — получает самый приоритетный элемент, но не извлекает его из очереди;
- extractPriorityElement — извлекает самый приоритетный элемент из очереди;
- increasePriority — изменяет приоритет элемента;
- merge — сливает очереди.

С точки зрения использования элементы будут последовательно извлекаться в порядке от самого приоритетного. Например, следующий список городов с их населением может быть представлен в виде приоритетной по убыванию населения очереди (табл. 4.2).

Табл. 4.2.

Значение	Приоритет
Москва	12 692 466
Екатеринбург	1 483 119
Нижний Тагил	352 135
Реж	36 843



Бинарная куча – это бинарное дерево, удовлетворяющее следующим условиям:

- Приоритет любой вершины не меньше приоритета потомков.
- Дерево является правильным подмножеством полного бинарного, допускающим плотное хранение узлов в массиве.

Другое название этой структуры данных — *пирамида* (heap). В невозрастающей пирамиде приоритет каждого родителя не меньше приоритета потомков (рис. 4.21).

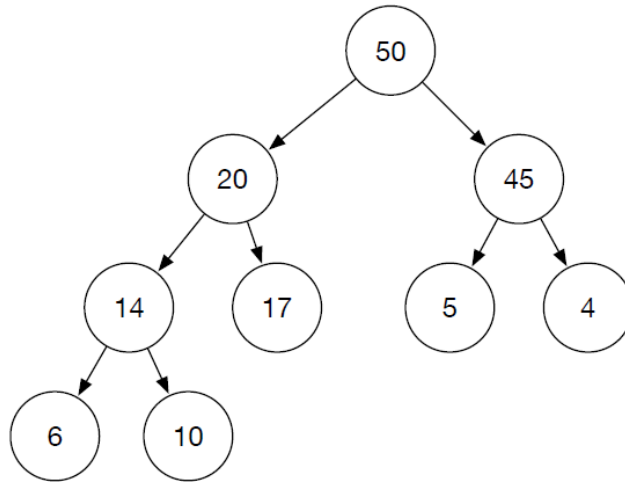


Рис. 4.21. Невозрастающая пирамида

Так как дерево с узлами — плотное, удобно хранить его в виде массива с индексами от 1 до N:

50	20	45	14	17	5	4	6	10
----	----	----	----	----	---	---	---	----

Удобство такого хранения трудно переоценить:

- Индекс корня дерева всегда равен 1 — самый приоритетный элемент
- Индекс родителя узла i всегда равен $\lfloor i/2 \rfloor$
- Индекс левого потомка узла i всегда равен $2i$
- Индекс правого потомка узла i всегда равен $2i + 1$

```

struct BNode<T>
{ // узел
    T data;
    int priority;
};
class BinaryHeap<T>
{
    BNode<T>[] body;
    int bodysize;
    int numnodes;
public BinaryHeap (int maxsize)
    {
        //...
    }
}
  
```

Операция создания бинарной кучи определённого размера заключается в простом выделении памяти под массив, хранящий элементы. Нулевой элемент массива мы использовать не будем. Будем фиксировать количество помещённых в кучу элементов в переменной `numnodes`. Ещё нам понадобится операция обмена элементов кучи по их индексам.

```
public BinaryHeap (int maxsize)
{
    body = new BNode<T>[maxsize + 1];
    bodysize = maxsize;
    numnodes = 0;
}

void swap(int a, int b)
{
    //...
}
```

Сложность операции создания бинарной кучи — $T_{\text{create}} = O(N)$.

Операция поиска самого приоритетного элемента тривиальна. Её сложность — $T_{\text{fetchMin}} = O(1)$.

```
BNode<T> fetchPriorityElement()
{
    if (numnodes == 0)
        throw new NullReferenceException();
    return body[1];
}
```

Операция добавления элемента требует небольшой эквилибристики. Правильная бинарная куча должна поддерживать два инварианта – структурной целостности, то есть представимости в виде бинарного дерева, и упорядоченной целостности, то есть свойства «потомки узла не могут иметь приоритет больший, чем у родителя». Структурную целостность поддерживать сложнее, поэтому мы с неё и начнём.

Этап 1. Вставка в конец кучи (рис. 4.22).

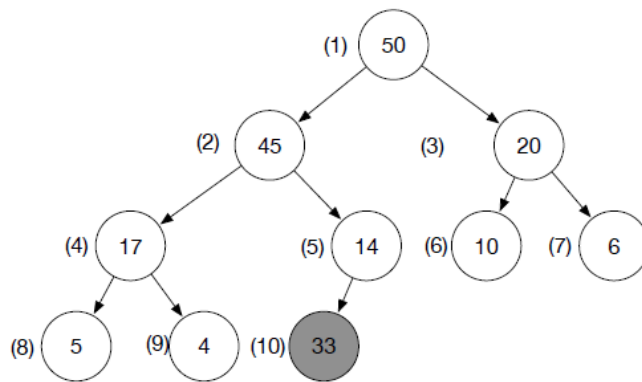


Рис. 4.22. Вставка элемента 33.

Нетрудно заметить, что структура кучи не испортилась, однако не выдержана упорядоченность.

Этап 2. Корректировка значений.

Только что вставленный элемент может оказаться более приоритетным, чем его родитель. В таком случае необходимо поменять их местами (рис. 4.23).

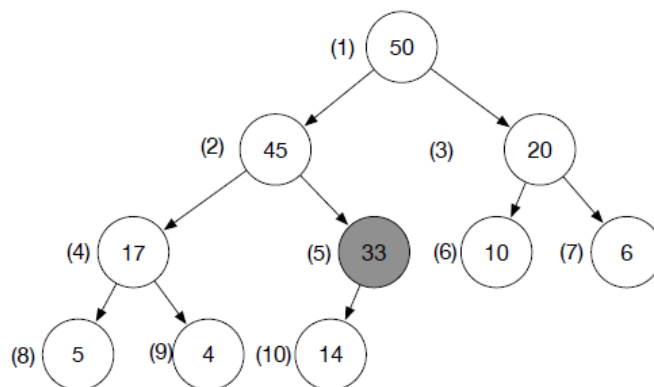


Рис. 4.23.

После таких манипуляций куча удовлетворяет всем условиям. Если вставляемый элемент имеет приоритет больше, чем все элементы в куче, необходимо менять его местами с родительскими узлами до тех пор, пока он не окажется в корне дерева.

Корректность алгоритма базируется на двух фактах:

- Инвариант структурной целостности не нарушен ни в один момент времени;

– После каждого шага перемещения вставленного элемента поддереву с корнем в текущем элементе поддерживает инвариант упорядоченной целостности.

Сложность алгоритма определяется высотой дерева и составляет $T_{\text{Insert}} = O(\log N)$.

```
void Insert(BHNode<T> node)
{
    if (numnodes > bodysize)
    {
        throw new IndexOutOfRangeException(); // или
        // необходимо расширить дерево
    }
    body[++numnodes] = node;
    for (int i = numnodes; i > 1 && body[i].priority >
body[i / 2].priority; i /= 2)
    {
        swap(i, i / 2);
    }
}
```

Операция удаления самого приоритетного элемента кажется более сложной – ведь после удаления корневого элемента нарушается структурная целостность, а восстанавливать структурную целостность, сохраняя упорядоченность — задача не из простых. Выход здесь заключается в повторном использовании принципа: делать сначала то, после чего существует простой путь к исправлению. Как мы выяснили при операции вставки, упорядоченную целостность восстанавливать легче, поэтому первый шаг при удалении корневого элемента должен сохранять структурную целостность. Так как после удаления корня количество элементов уменьшится на один, то отправляем самый последний элемент кучи в корень, уменьшая при этом numnodes (рис. 4.24).

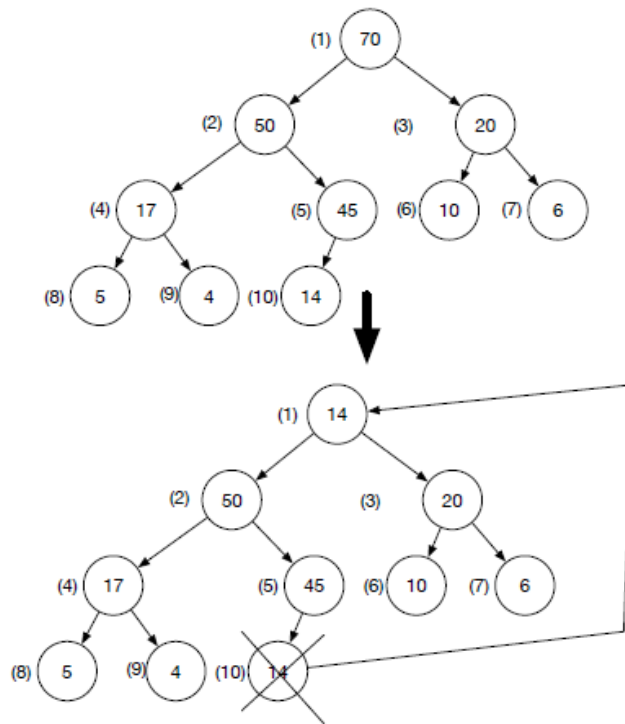


Рис. 4.24. Удаление максимального элемента

Структурная целостность кучи не изменилась, но могла измениться упорядоченная целостность. Поэтому, требуется ее восстановить с помощью метода `heapify`.

```
void Heapify(int index)
{
    for (; ; )
    {
        int left = index + index;
        int right = left + 1;
        // Кто больше, [index], [left], [right]?
        int largest = index;
        if (left <= numnodes && body[left].priority >
            body[index].priority)
            largest = left;
        if (right <= numnodes && body[right].priority >
            body[largest].priority)
            largest = right;
        if (largest == index) break;
        swap(index, largest);
        index = largest;
    }
}
```

Идея метода проста: начиная с корневого элемента мы проводим соревнование между тремя кандидатами – теми, кто может занять это место.

Если кандидаты (левый и правый потомки) менее приоритетны, чем текущий претендент, то алгоритм завершён. Иначе претендент меняется местами с самым приоритетным из потомков, и операция повторяется уже на уровне ниже. Так как каждая операция обмена передвигает претендента на один уровень вниз, процесс обязательно завершается не более, чем за $O(\log N)$ шагов, что и составляет сложность данного алгоритма.

Проиллюстрируем алгоритм на примере. После перемещения последнего узла (14) в корень он становится претендентом, а кандидатами оказываются узлы (50) и (20). Индекс восстановления (номер претендента) пока равен 1 (рис. 4.25).

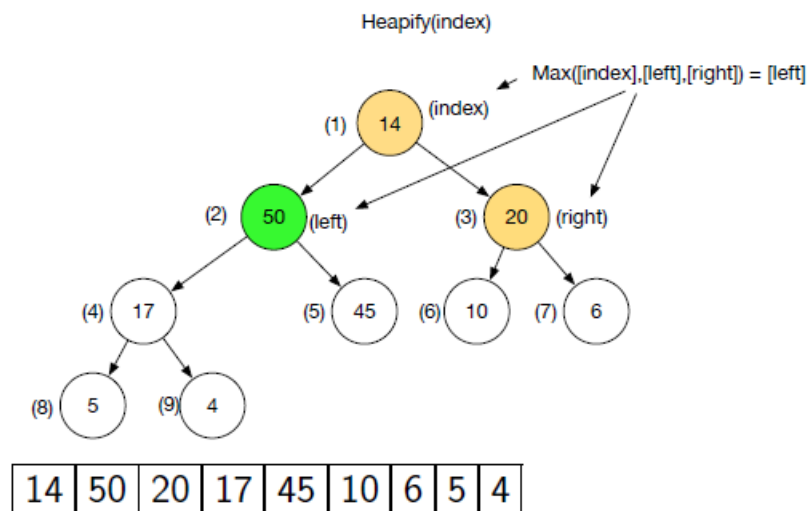


Рис. 4.25.

Претендента обменяли на кандидата (50) с индексом 2. Теперь элемент под этим индексом — новый претендент (рис. 4.26).

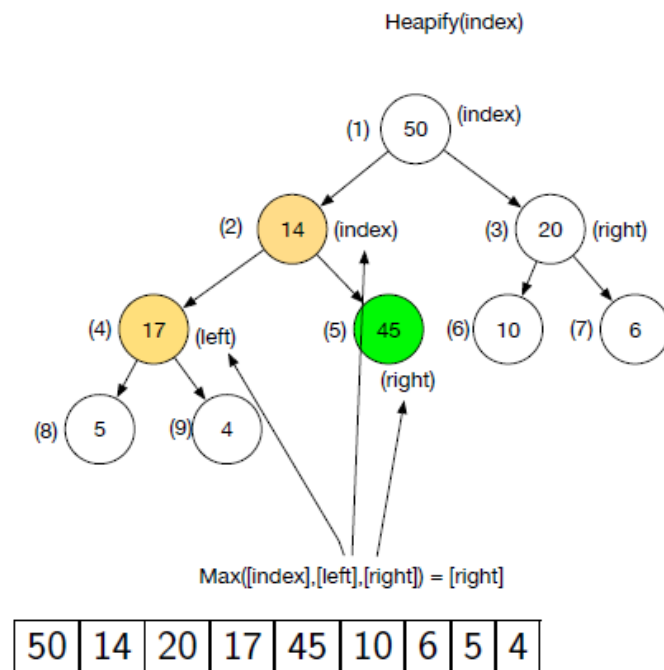


Рис. 4.26.

После очередного обмена претендентом становится элемент с индексом 5 (рис. 4.27).

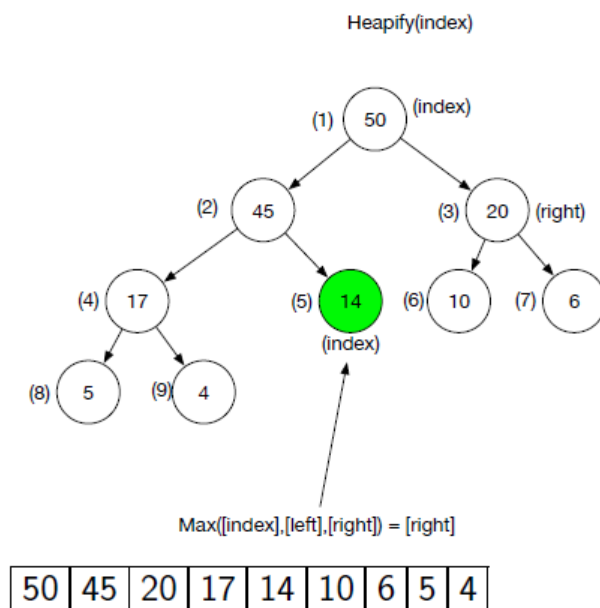


Рис. 4.27.

Восстановление закончено.



Продолжайте работать в созданном проекте. Реализуйте структуру данных «Бинарная куча» и основные методы работы со структурой данных.

4.3.2 Пирамидальная сортировка

Возможность получать из бинарной кучи самый приоритетный элемент за $O(\log N)$ и добавлять элементы в бинарную кучу за $O(\log N)$ вызывают желание реализовать ещё один алгоритм сортировки. Что интересно, этот алгоритм будет иметь сложность $O(N \log N)$ в худшем случае.

Для доказательства этого рассмотрим сначала простой метод реализации сортировки массива размером N .

1. Создать бинарную кучу размером N . Это потребует сложности $O(N)$.

2. Поочерёдно вставить в неё все N элементов массива. Сложность этого этапа есть

$$O(\log 1) + O(\log 2) + \dots + O(\log N) < O(\log N) + O(\log N) + \dots + O(\log N) = N \log N.$$

3. Поочерёдно извлекать с удалением самый приоритетный элемент из бинарной кучи с помещением их в последовательные N позиций исходного массива. Сложность этого этапа есть

$$O(\log N) + \dots + O(\log 2) + O(\log 1) < O(\log N) + \dots + O(\log N) + O(\log N) = N \log N.$$

Общая сложность всех этапов составляет $O(N \log N)$. Такая прямолинейная организация не особенно хороша: потребуется добавочная память на бинарную кучу размером N элементов. Небольшая хитрость – и добавочной памяти можно избежать.

Модифицируем функцию `heapify` для того, чтобы она могла работать с произвольным массивом, адресуемым с нуля:

```
void Heapify(int[] a, int i, int n)
{
    int curr = a[i];
    int index = i;
    for (; ; )
    {
        int left = index + index + 1;
        int right = left + 1;
        if (left < n && a[left] > curr)
```

```

        index = left;
    if (right < n && a[right] > a[index])
        index = right;
    if (index == i) break;
    a[i] = a[index];
    a[index] = curr;
    i = index;
}
}

```

Теперь сортировка заключается в том, что мы создаём бинарную кучу размером n на месте исходного массива, переставляя его элементы. Затем на шаге i мы обмениваем самый приоритетный элемент кучи (он всегда располагается на позиции 0) с элементом под номером $n - i - 1$. Размер кучи при этом уменьшается на единицу, а самый приоритетный элемент занимает теперь положенное ему по рангу место.

```

void HeapSort(int[] a, int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
    {
        Heapify(a, i, n);
    }
    while (n > 1)
    {
        n--;
        swap(a[0], a[n]);
        Heapify(a, 0, n);
    }
}

```

Возникает вопрос: если эта сортировка гарантирует нам сложность $O(N \log N)$ даже в самом худшем случае, а быстрая сортировка не гарантирует, то почему не использовать только эту сортировку?

Первая причина в том, что в быстрой сортировке используется меньшее количество операций обмена с памятью, а излишней работы с памятью на современных компьютерах стоит избегать. Вторая причина тоже связана с памятью, точнее с тем фактом, что N обращений к последовательным ячейкам памяти выполняется до 10-15 раз быстрее, чем столько же обращений к случайным ячейкам памяти. Это связано с наличием ограниченного количества аппаратной кэш-памяти на современных процессорах. При наличии различных

алгоритмов, исполняющих одну и ту же задачу, некоторые из них могут быть дружелюбны к кэшу (cache-friendly), а некоторые – нет. Поэтому наилучшие по времени исполнения алгоритмы могут быть разными в разное время и на различных вычислительных системах.



Продолжайте работать в созданном проекте. Реализуйте метод пирамидальной сортировки любого массива. Сравните быструю и пирамидальную сортировку на разных наборах данных.

Контрольные вопросы

1. Что такое список?
2. Какие абстракции реализует структура данных список?
3. Чем отличаются списки линейные и кольцевые?
4. Как происходит добавление элемента в хвост списка? В начало списка?
5. Как происходит вставка элемента перед и после указанного элемента?
6. Как происходит удаление элемента из списка?
7. В чем особенность двусвязного списка?
8. Приведите примеры использования списков?
9. Что такое очередь? Какие операции реализует эта структура данных?
10. Что такое дерево?
11. Что такое бинарное дерево?
12. Чем отличаются вершины и узлы дерева?
13. Какие виды обхода деревьев вы можете назвать?
14. Как представляется дерево в программной реализации? В чем различие таких представлений? Когда какой способ целесообразнее использовать?
15. Что такое приоритетная очередь?
16. Что такое полное бинарное дерево?
17. Что такое бинарная куча?

18. Как реализуются операции добавления элемента в очередь и извлечения самого приоритетного элемента из очереди?
19. Опишите алгоритм пирамидальной сортировки.
20. Почему пирамидальную сортировку не всегда целесообразно использовать?

Задания для самостоятельного выполнения

Задание №1. Скобочная последовательность

Последовательность A , состоящую из символов из множества «(», «)», «[» и «]», назовем *правильной скобочной последовательностью*, если выполняется одно из следующих утверждений:

A — пустая последовательность;

первый символ последовательности A — это «(», и в этой последовательности существует такой символ «)», что последовательность можно представить как $A=(B)C$, где B и C — правильные скобочные последовательности;

первый символ последовательности A — это «[», и в этой последовательности существует такой символ «]», что последовательность можно представить как $A=[B]C$, где B и C — правильные скобочные последовательности.

Так, например, последовательности «(())» и «()[]» являются правильными скобочными последовательностями, а последовательности «[]» и «((» таковыми не являются.

Входной файл содержит несколько строк, каждая из которых содержит последовательность символов «(», «)», «[» и «]». Для каждой из этих строк выясните, является ли она правильной скобочной последовательностью.

Формат входных данных:

Первая строка входного файла содержит число N ($1 \leq N \leq 500$) - число скобочных последовательностей, которые необходимо проверить. Каждая из следующих N строк содержит скобочную последовательность длиной от 1 до 104 включительно. В каждой из последовательностей присутствуют только скобки указанных выше видов.

Формат выходных данных:

Для каждой строки входного файла выведите в выходной файл «YES», если соответствующая последовательность является правильной скобочной последовательностью, или «NO», если не является.

Задание №2. Очередь с минимумом

Реализуйте работу очереди. В дополнение к стандартным операциям очереди, необходимо также отвечать на запрос о минимальном элементе из тех, которые сейчас находятся в очереди. Для каждой операции запроса минимального элемента выведите ее результат.

На вход программе подаются строки, содержащие команды. Каждая строка содержит одну команду. Команда — это либо «+ N », либо «-», либо «?». Команда «+ N » означает добавление в очередь числа N , по модулю не превышающего 109. Команда «-» означает изъятие элемента из очереди. Команда «?» означает запрос на поиск минимального элемента в очереди.

Формат входных данных:

В первой строке содержится M ($1 \leq M \leq 106$) — число команд. В последующих строках содержатся команды, по одной в каждой строке.

Формат выходных данных:

Для каждой операции поиска минимума в очереди выведите её результат. Результаты должны быть выведены в том порядке, в котором эти операции встречаются во входном файле. Гарантируется, что операций извлечения или поиска минимума для пустой очереди не производится.

Задание №3. Куча ли?

Структуру данных «куча», или, более конкретно, «неубывающая пирамида», можно реализовать на основе массива. Для этого должно выполняться основное свойство неубывающей пирамиды, которое заключается в том, что для каждого $1 \leq i \leq n$ выполняются условия:

- если $2i \leq n$, то $a[i] \leq a[2i]$;
- если $2i+1 \leq n$, то $a[i] \leq a[2i+1]$.

Дан массив целых чисел. Определите, является ли он неубывающей пирамидой.

Формат входных данных:

Первая строка входного файла содержит целое число n ($1 \leq n \leq 106$). Вторая строка содержит n целых чисел, по модулю не превосходящих $2 \cdot 10^9$.

Формат выходных данных:

Выведите «YES», если массив является неубывающей пирамидой, и «NO» в противном случае.

Задание №4. Очередь с приоритетами

Реализуйте очередь с приоритетами. Ваша очередь должна поддерживать следующие операции: добавить элемент, извлечь минимальный элемент, уменьшить элемент, добавленный во время одной из операций.

Формат входных данных:

В первой строке входного файла содержится число n ($1 \leq n \leq 106$) - число операций с очередью.

Следующие n строк содержат описание операций с очередью, по одному описанию в строке. Операции могут быть следующими:

A x — требуется добавить элемент x в очередь.

X — требуется удалить из очереди минимальный элемент и вывести его в выходной файл. Если очередь пуста, в выходной файл требуется вывести звездочку «*».

Д х у — требуется заменить значение элемента, добавленного в очередь операцией А в строке входного файла номер $x+1$, на у. Гарантируется, что в строке $x+1$ действительно находится операция А, что этот элемент не был ранее удален операцией Х, и что у меньше, чем предыдущее значение этого элемента.

В очередь помещаются и извлекаются только целые числа, не превышающие по модулю 109.

Формат выходных данных:

Выведите последовательно результат выполнения всех операций Х, по одному в каждой строке выходного файла. Если перед очередной операцией Х очередь пуста, выведите вместо числа звездочку «*».

Задание №5. Постфиксная запись

В постфиксной записи (или обратной польской записи) операция записывается после двух операндов. Например, сумма двух чисел А и В записывается как А В +. Запись В С + D * обозначает привычное нам $(В + С) * D$, а запись А В С + D * + означает $A + (B + C) * D$. Достоинство постфиксной записи в том, что она не требует скобок и дополнительных соглашений о приоритете операторов для своего чтения.

Дано выражение в обратной польской записи. Определите его значение.

Формат входных данных:

В первой строке входного файла дано число N ($1 \leq N \leq 106$) - число элементов выражения. Во второй строке содержится выражение в постфиксной записи, состоящее из N элементов. В выражении могут содержаться неотрицательные однозначные числа и операции +, -, *. Каждые два соседних элемента выражения разделены ровно одним пробелом.

Формат выходных данных:

Необходимо вывести значение записанного выражения. Гарантируется, что результат выражения, а также результаты всех промежуточных вычислений, по модулю будут меньше, чем 231.

ГЛАВА 5. АЛГОРИТМЫ ПОИСКА

5.1 Последовательный поиск

Информация нужна для того, чтобы ей пользоваться. Дорогостоящая операция сортировки проводится именно для того, чтобы пользоваться информацией стало удобнее. Сортировка сама по себе обычно никого не интересует, интересны её результаты. Представим себе словарь с переводом иностранных слов, в котором нет порядка. Поиск в таком словаре занимал бы чудовищное время. Упорядоченный по алфавиту словарь – другое дело.

Сформулируем расширенную задачу поиска:

1. Этап первый. Сбор информации. Её накопление.
2. Этап второй. Организация информации (переупорядочивание, сортировка).
3. Этап третий. Извлечение информации (собственно поиск).

При построении словаря новые карточки могут сразу вноситься таким образом, чтобы не нарушать упорядоченность, то есть, второй и третий этапы могут смешиваться.

Более обобщённая задача: абстракция *хранилище*.

Задача: построение эффективного хранилища данных.

Требования:

- Поддержка больших объёмов информации.
- Возможность быстро находить данные.
- Возможность быстро модифицировать, в том числе и удалять данные.

Методы абстракции:

- Create — добавление новой записи к хранилищу.
- Read — поиск данных по ключу.
- Update — обновление данных по ключу.
- Delete — удаление данных.

Самая первая подзадача – задача, которую требуется решить для реализации хранилища – Read. Формализуем её частный случай, который назовём find.

Пусть имеется множество ключей a_1, a_2, \dots, a_n . Требуется определить индекс ключа, совпадающего с заданным значением key.

```
bunch a;
index = a.find(key);
```

Здесь bunch — некое абстрактное хранилище элементов, содержащих ключи. *Хорошая организация хранилища входит в расширенную задачу поиска.*

Самая простая ситуация: ключи не упорядочены, они находятся в простом массиве (рис. 5.1).

Индекс	0	1	2	3	4	5	6	7	8	9
Ключ	132	612	232	890	161	222	123	861	120	330
Данные	AB	CA	ЯФ	AB	AA	НД	ОР	ОС	ЗЛ	УГ

Рис. 5.1.

Операция $\text{find}(a, 222)$ возвратит индекс, равный 5, а вот, например, элемента 999 в хранилище нет, поэтому операция поиска должна вернуть нечто, которое заведомо не может служить индексом при доступе к данному массиву. Например, таким индексом может быть число -1 или номер элемента за границей массива. Остановимся на последнем варианте. $\text{find}(a, 999) = 10$ (элемент за границей поиска).

Листинг алгоритма последовательного поиска:

```
static int DummySearch(int[] array, int element)
{
    for (int i = 0; i < array.Length; i++)
        if (array[i] == element) return i;
    return array.Length;
}
```

При случайных значениях ключей в массиве и при поиске присутствующего в нём ключа вероятность найти ключ в i -м элементе $P_i = 1 / N$. Математическое ожидание числа поисков в этом случае $E = N / 2$. Число операций сравнения $2N$ в худшем случае $T(N) = O(N)$.

Обратите внимание, что при использовании исполнителя в виде языка C# на каждой итерации цикла происходит две операции сравнения? Сравнивается искомый ключ с очередным элементом массива и, кроме того, номер элемента массива с количеством элементов.

Прделаем небольшую подготовку: положим искомый элемент в элемент, находящийся за последним элементом массива (рис. 5.2).

Индекс	0	1	2	3	4	5	6	7	8	9	10
Ключ	132	612	232	890	161	222	123	861	120	330	999
Данные	AB	CA	ЯФ	AB	AA	НД	ОР	ОС	ЗЛ	УГ	??

Рис. 5.2.

Результаты поисков не изменились. `find(a, 222)` по-прежнему возвращает 5, а `find(a, 999)` — 10.

```
static int CleverSearch(int[] array, int element)
{
    n = array.Length;
    array[n] = element;
    for (int i = 0; array[i] != element; i++)
        if (array[i] == element) return i;
    return array.Length;
}
```

Сложность операций для неупорядоченного массива:

- Create — $O(1)$. Мы просто добавляем ключ к концу массива, расширяя его.
- Read — $O(N)$.
- Update — $O(N)$. Элемент нужно сначала найти.
- Delete — $O(N)$. После поиска удаляемого элемента массив требуется сжать.



Создайте новый проект. В созданном проекте напишите метод, реализующий последовательный поиск.

5.2 Бинарный поиск

К сожалению, неупорядоченные данные обрабатывать достаточно трудно. Если в зоне поиска имеется упорядочивание — всё становится значительно лучше.

Сортировка даёт нам возможность упорядочить по какому-либо отношению. Тогда задачу поиска можно переформулировать следующим образом:

– Имеется множество ключей

$$a_1 \leq a_2 \leq \dots \leq a_n$$

– Требуется определить индекс ключа, совпадающего с заданным значением `key` или указать, что такого значения в множестве не существует.

Если мысленно разбить множество на два примерно равных подмножества, то принцип «разделяй и властвуй» здесь будет работать идеально.

1. Искомый элемент равен центральному? Да — нашли.

2. Искомый элемент меньше центрального? Да — рекурсивный поиск в левой половине.

3. Искомый элемент больше центрального? Да — рекурсивный поиск в правой половине.

– Вход алгоритма: упорядоченный по возрастанию массив, левая граница поиска, правая граница поиска.

– Выход алгоритма: номер найденного элемента или -1. Формализуя алгоритм, можно получить следующее решение:

```
int BinarySearch(int array[], int value, int left, int right)
{
    if (left >= right) return array[left] == value? left : -1;
    int mid = (left+right)/2; if (array[mid] == value) return mid;
    if (array[mid] < value) {
        return BinarySearch(array, value, left, mid-1);
    }
    else {
        return BinarySearch(array, value, mid+1, right);
    }
}
```

}

Глубину рекурсии оценить легко: каждый раз интервал поиска уменьшается примерно в два раза, поэтому количество рекурсивных вызовов здесь не превзойдёт $O(\log N)$, где N — размер множества.

Рассмотрим пример нахождения ключа 313 в массиве {1, 4, 23, 45, 67, 68, 89, 105, 144, 279, 313, 388} (рис. 5.3).

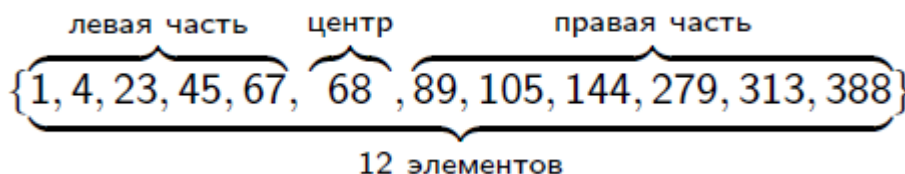


Рис. 5.3.

Так как $313 > 68$, то ключ находится в правом подмассиве (рис. 5.4).



Рис. 5.4.

На следующем этапе сравниваем 313 и 144, и обнаруживаем, что ключ находится снова в правом подмассиве (рис. 5.5). На следующем шаге он будет найден.

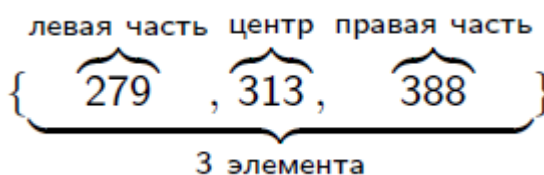


Рис. 5.5.

Сложность алгоритма здесь достаточно очевидна:

- Количество подзадач $a = 1$.
- Каждая подзадача уменьшается в $b = 2$ раза.
- Сложность консолидации $O(1) = O(N^0) \rightarrow d = 0$

Так как $d = \log_b a$ то $T(N) = \log N$.

Использование рекурсии, конечно, изящно, но итеративные алгоритмы обычно практичнее.

```
static int BinarySearch(int[] array, int element)
{
    var left = 0;
    var right = array.Length - 1;
    while (left < right)
    {
        var middle = (right + left) / 2;
        if (element <= array[middle])
            right = middle;
        else left = middle + 1;
    }
    if (array[right] == element)
        return right;
    return -1;
}
```

В худшем случае на каждой итерации в данном варианте алгоритма нужно провести три операции сравнения. Как уже было сказано ранее, операция сравнения — не самая быстрая из всех операций на современных компьютерах и неплохо бы уменьшить количество таких операций.

Имеется много вариантов бинарного поиска с сужением интервала, каждый из которых может иметь свои плюсы для решения конкретной задачи. Однако, асимптотическая сложность алгоритма остаётся логарифмической.

Сложность операций для упорядоченного массива:

- Create — $O(N)$. Для добавления элемента требуется расширить массив.
- Read — $O(\log N)$.
- Update — $O(N)$. Элемент нужно сначала найти за $O(\log N)$, затем передвинуть его налево или направо, в зависимости от величины изменения. Увы, но эта операция занимает $O(N)$.
- Delete — $O(N)$. После поиска удаляемого элемента массив требуется сжать.



В созданном проекте напишите метод, реализующий бинарный поиск.

5.3 Распределяющий поиск

Рассмотрим поиск с использованием свойств ключа. Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$? Без вспомогательных данных – нет.

Какова сложность нахождения $M \approx N$ значений в неотсортированном массиве? Вариант ответа: если $M > \log N$, то предварительной сортировкой можно добиться того, что сложность составит $O(N \log N) + M \cdot O(\log N) = O(N \log N)$. Можно ли быстрее? Оказывается, что в некоторых случаях можно. Если $|D(\text{Key})|$ невелико, то имеется способ, похожий на сортировку подсчётом.

Создаётся инвертированный массив (рис. 5.6).

$a = \{2, 7, 5, 3, 8, 6, 3, 9, 12\}$. $|D(a)| = 12 - 2 + 1 = 11$.

$a_{\text{inv}}[2..12] = \{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1\}$

$a[0] = 2 \rightarrow a_{\text{inv}}[2] = 0$

$a[1] = 7 \rightarrow a_{\text{inv}}[7] = 1$

$a[2] = 5 \rightarrow a_{\text{inv}}[5] = 2$

$a_{\text{inv}}[2..12] = \{0, 6, -1, -1, 5, -1, 4, 7, -1, -1, 8\}$

index	0	1	2	3	4	5	6	7	8
key	2	7	5	3	8	6	3	9	12

key	2	7	5	3	8	6	3	9	12
index	0	1	2	3	4	5	6	7	8

Рис. 5.6.

Алгоритм состоит из двух этапов — подготовки данных и собственно поиска. Подготовка данных состоит в создании инвертированного массива:

```
int[] prepare(int[] array, int N, out int min, out int max)
{
    min = max = array[0];
    for (int i = 1; i < N; i++)
```



```

    {
        if (array[i] > max)
            max = array[i];
        if (array[i] < min)
            min = array[i];
    }
    int[] result = new int[max - min + 1];
    for (int i = min; i <= max; i++)
    {
        result[i] = -1;
    }
    for (int i = 0; i < N; i++)
    {
        result[array[i] - min] = i;
    }
    return result;
}

```

Второй этап поиска реализуется следующим образом:

```

// Подготовка
int min, max;
int[] ainv = prepare(a, N, out min, out max);
// Поиск ключа key
int result = -1;
if (key >= min && key <= max) result = ainv[key - min];

```

Особенности данного алгоритма следующие:

$O(N)$ на подготовку.

$O(M)$ на поиск M элементов.

$T(N, M) = O(N) + O(M) = O(N)$

Сложность операций:

– find — $O(1)$

– insert — $O(1)$

– remove — $O(1)$

Основным недостатком данного алгоритма является жесткие ограничения на множество ключей. При наличии $f(\text{key})$ сводится к хеш-поиску.



В созданном проекте напишите метод, реализующий распределяющий поиск. Протестируйте все три реализованных метода поиска. Сравните их результативность с разными наборами данных.

5.4 Поиск по бинарному дереву

5.4.1 Абстракция «Отображение»

Абстракция «массив» устанавливает соответствие между номерами элементов (или, как говорят, индексами) и их значениями. Важное свойство массивов — возможность «пробежать» по всем возможным индексам, получив все значения по ним. Второе важное свойство — сложность операций извлечения по индексу и установки значения по индексу очень мала и составляет $O(1)$. Можно сказать, что абстракция массив устанавливает соответствие между подмножеством множества неотрицательных целых чисел в заданном диапазоне (множества ключей) и множеством значений по этим ключам. Если расширить множество ключей до произвольных величин, то мы приходим к понятию абстракции отображение



Отображение — это абстракция, устанавливающая направленное соответствие между двумя множествами (множеством ключей и множеством данных) и реализующая определённые операции над ними (рис. 5.7).



Рис. 5.7.

Абстракция «отображение» есть аналог дискретной функции. В языке C# данная абстракция реализуется с помощью класса Dictionary.

```
Dictionary<string, int> dict = new Dictionary<string, int>;  
dict["Шанхай"] = 24150000;
```

```
dict["Карачи"] = 23500000;
dict["Пекин"] = 21150000;
dict["Дели"] = 17830000;
int BeijingPopulation = dict["Пекин"];
foreach(var x in dict)
{
    Console.WriteLine("Население города {0} составляет {1}",
x.Key, x.Value);
}
```

Интерфейс абстракции отображение есть частный случай абстракции хранилища CRUD, и реализует следующие функции:

- *insert(key, value)* — добавить элемент с ключом *key* и значением *value*;
- *Item find(key)* — найти элемент с ключом *key* и вернуть его;
- *erase(key)* — удалить элемент с ключом *key*;
- *walk* — получить все ключи (или все пары ключ/значение) в каком-либо порядке.

В дальнейшем под термином ключ будем понимать пару «ключ+значение», в которой определена операция сравнения по ключу.

Абстракцию «множество» можно рассматривать как частный случай абстракции отображение. Например, одна из возможных реализация отображения – множество с прикрепленными данными. С другой стороны – можно абстрагировать понятие множество как отображение набора ключей на логические переменные, все присутствующие в множестве элементы отображаются на логическую истину, остальные – на логическую ложь. Одна из удобных и эффективных структур данных для представления и множеств, и отображений – бинарное дерево поиска.

5.4.2 Бинарные деревья поиска



Бинарным деревом поиска называется бинарное дерево, в котором все узлы, находящиеся справа от родителя, имеют значения, не меньшие значения в родительском узле, а слева — не большие этого значения.

Задача: на вход подаётся последовательность чисел. Например, {10, 5, 35, 7, 3, 23, 94, 2, 5, 7}. Выходом должно быть двоичное дерево поиска (рис. 5.8).

Бинарные деревья поиска часто обозначаются аббревиатурой BST — Binary Search Tree.

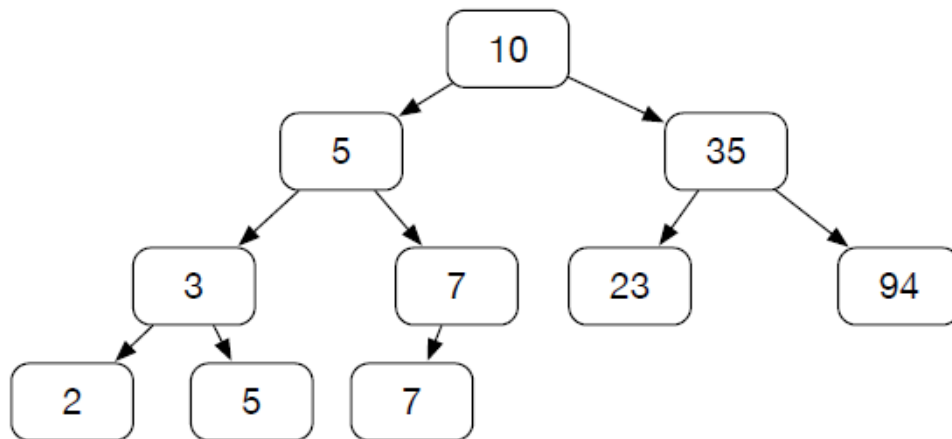


Рис. 5.8.

Алгоритм поиска элемента в BST, содержащего ключ X

1. Делаем текущий узел корневым.
2. Переходим в текущий узел C .
3. Если $X = C.\text{Key}$, то алгоритм завершён.
4. Если $X > C.\text{Key}$ и C имеет потомка справа, то делаем текущим узлом потомка справа. Переходим к п. 2.
5. Если $X < C.\text{Key}$ и C имеет потомка слева, то делаем текущим узлом потомка слева. Переходим к п. 2.
6. Ключ не найден. Конец алгоритма.

Алгоритм прост и эффективен и его сложность определяется только наибольшей высотой дерева.

Наивное построение бинарных деревьев поиска по последовательности ключей заключается в том, что первый поступивший элемент последовательности формирует корневой узел дерева и каждый последующий элемент занимает соответствующее место после неудачного поиска (если поиск удачен, то дерево уже содержит ключ). Неудачный поиск всегда заканчивается на каком-то узле, у которого в нужном направлении нет потомка.

Алгоритм вставки элемента с ключом X в BST.

1. Делаем текущий узел корневым.
2. Переходим в текущий узел C .
3. Если $X = C.Key$, то алгоритм завершён, вставка невозможна.
4. Если $X > C.Key$ и C имеет потомка справа, то делаем текущим узлом потомка справа. Переходим к п. 2.
5. Если $X > C.Key$ и C не имеет потомка справа, то создаём правого потомка с ключом X и завершаем алгоритм.
6. Если $X < C.Key$ и C имеет потомка слева, то делаем текущим узлом потомка слева. Переходим к п. 2.
7. Если $X < C.Key$ и C не имеет потомка слева, то создаём левого потомка с ключом X и завершаем алгоритм.

На рис. 5.9. представлен пример хорошего дерева поиска.

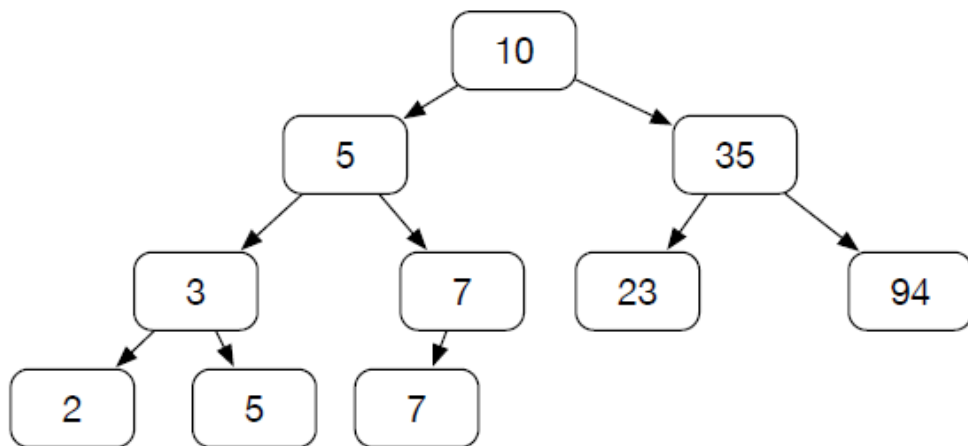


Рис. 5.9.

Однако, не всегда деревья получаются такими (рис. 5.10). Поиск по дереву на следующем рисунке имеет сложность $O(N)$.

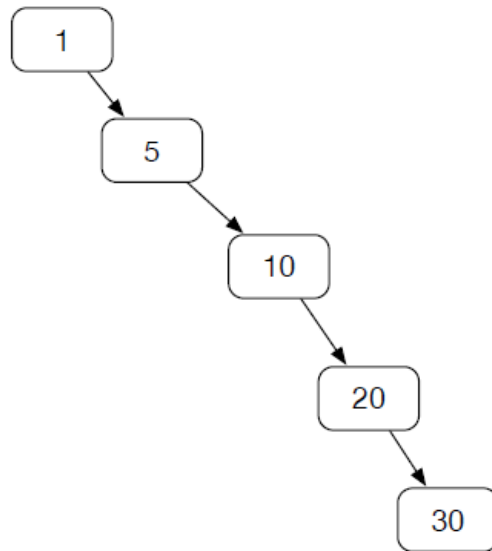


Рис. 5.10.

Интересно узнать, насколько часто будут встречаться такие деревья при случайном поступлении элементов из последовательности?



Случайное бинарное дерево T размера n – это дерево, получающееся из пустого бинарного дерева поиска после добавления в него n узлов с различными ключами в случайном порядке, при условии, что все $n!$ возможных последовательностей добавления равновероятны.

Определим его среднюю глубину. Пусть $\bar{d}(N + 1)$ – средняя глубина всех узлов случайного дерева с $N + 1$ узлами. Пусть k – узел, добавленный первым. Вероятность добавления узла k есть $p_k = 1/(N + 1)$.

Остальные узлы разобьются на группы, каждая из которых начнётся с высоты 1. В левую группу войдут элементы $\{0, \dots, k - 1\}$, в правую – $\{k + 1, \dots, N\}$.

$$\bar{d}(N + 1) = \sum_{k=0}^N \frac{1}{N + 1} \left(1 + \frac{k}{N} \times \bar{d}(k) + \frac{N - k}{N} \times \bar{d}(N - k) \right)$$

$$\bar{d}(N + 1) = \frac{2}{N(N + 1)} \sum_{k=0}^N k \times \bar{d}(k)$$

Используя предел

$$\lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k} - \ln n \right) = \gamma = 0.577$$

получаем

$$\lim_{N \rightarrow \infty} (\bar{d}(N) - 2 \ln N) \rightarrow \text{const}$$

Отсюда можно сделать вывод, что и средняя глубина узлов случайного бинарного дерева, и средние времена выполнения операций вставки, удаления и поиска в случайном бинарном дереве есть $\Theta(\log_2 N)$.



В созданном проекте реализуйте класс бинарного дерева, используемый в предыдущей главе. Преобразуйте его в бинарное дерево поиска. Реализуйте методы вставки нового элемента в бинарное дерево поиска и поиска заданного элемента в дереве.

5.4.3 Свойства бинарного дерева поиска

Так как все потомки слева от узла имеют значения ключей, меньшие значения ключа самого узла, то наименьший элемент всегда находится в самом низу левого поддерева. Аналогично, наибольший элемент всегда находится в самом низу правого поддерева.

```
BinaryTree<T> minNode(BinaryTree<T> t)
{
    if (t == null) return null;
    while (t.left != null)
    {
        t = t.left;
    }
    return t;
}
```

Методы поиска и вставки, описанные ранее, достаточно просты.

```
BinaryTree<T> SearchNode(BinaryTree<T> t, T key)
{
    BinaryTree<T> p = t;
    while (t != null)
    {
        p = t;
        if (t.data.Equals(key)) return t;
        t = key.Equals(t.data) ? t.right : t.left;
    }
}
```

```

        return p;
    }

    BinaryTree<T> InsertNode(BinaryTree<T> t, T key, T
value)
    {
        BinaryTree<T> parent = t;
        while (t != null)
        {
            parent = t;
            if (t.data.Equals(key))
                throw new ArgumentException();// если
такой элемент уже есть
            t = key.Equals(t.data) ? t.right : t.left;
        }
        BinaryTree<T> node = new BinaryTree<T>(key,
value);

        if (key < parent.data) parent.left = node;
        else parent.right = node;
    }

```

Процедура удаления сложнее, здесь требуется рассмотреть три случая:

- 1) у удаляемого узла нет потомков – достаточно удалить этот узел у родителя (рис. 5.11);
- 2) имеется один потомок – переставляем узел у родителя на потомка (рис. 5.12);
- 3) имеется два потомка – находим самый левый лист в правом поддереве и им замещаем удаляемый (рис. 5.13).

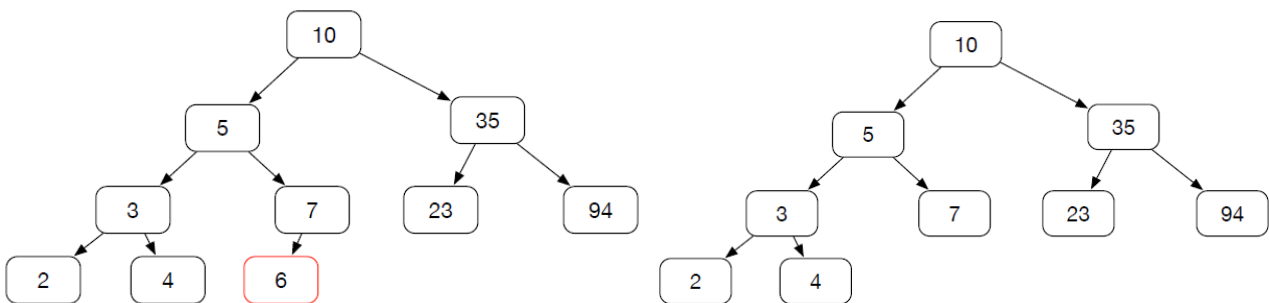


Рис. 5.11.

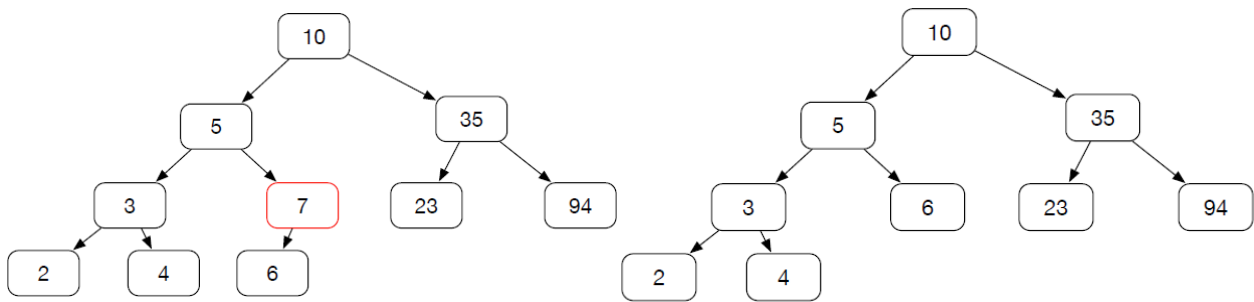


Рис. 5.12.

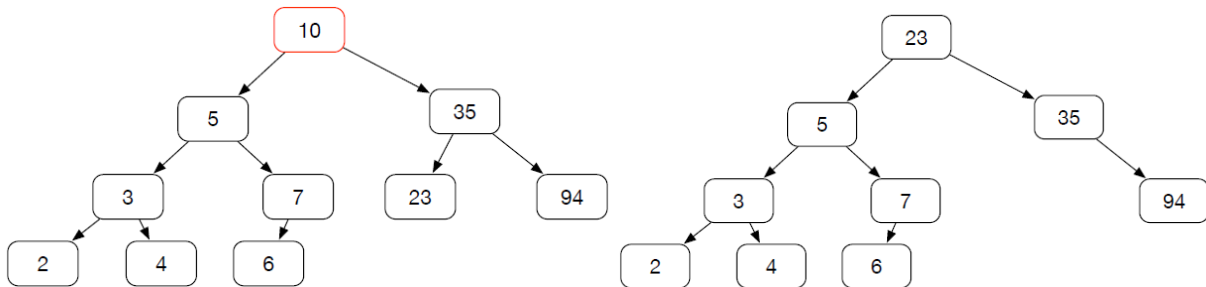


Рис. 5.13.



В созданном проекте реализуйте метод, который удаляет элемент из бинарного дерева поиска.

Контрольные вопросы

1. Сформулируйте расширенную задачу поиска.
2. Опишите абстракцию хранилище.
3. Как работает алгоритм последовательного поиска?
4. Опишите алгоритм бинарного поиска.
5. В чем разница рекурсивной и нерекурсивной реализации алгоритма бинарного поиска?
6. Как реализуется алгоритм распределяющего поиска?
7. Какова сложность каждого из рассмотренных алгоритмов поиска в этой главе?
8. Что из себя представляет абстракция «отображение»?
9. Чем похожи абстракции отображение, массив и множество?
10. Назовите основные функции, реализуемые абстракцией отображение.
11. Что такое бинарное дерево поиска?

12. Что такое случайное бинарное дерево?
13. Назовите основные свойства бинарного дерева поиска.
14. Как удалить узел из бинарного дерева поиска?
15. Как вставить элемент в бинарное дерево поиска? Как найти элемент в бинарном дереве поиска?

Задания для самостоятельного выполнения

Задание №1. Провода

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

На складе есть провода различной целочисленной длины. Их можно разрезать на части. Необходимо получить K кусочков одинаковой целочисленной и как можно большей длины. Найти максимальную длину M , при которой можно получить по меньшей мере K кусочков этой длины. Все оставшиеся на складе куски проводов длиной меньше M в подсчете не участвуют.

Формат входных данных:

В первой строке – количество проводов на складе N и необходимое количество кусочков K . В следующих N строках – длины проводов.

$$1 \leq N \leq 100000$$

$$1 \leq K \leq 100000$$

Формат выходных данных:

M – максимальная длина, на которую можно разрезать все провода так, чтобы получилось не менее K кусочков.

Задание № 2. Брокеры

Ограничение по времени: 1 секунда. Ограничение по памяти: 64 мегабайта

В королевстве Шарп фирма «Джава» имеет много отделений, работающих сравнительно автономно. После неких экономических преобразований такая форма функционирования оказалась невыгодной и фирма

решила сливать капиталы отделений, образуя укрупненные департаменты, отвечающие за несколько отделений сразу. Цель фирмы – слить все отделения в один громадный департамент, владеющий всеми капиталами. Проблема заключается в том, что по законам королевства Шарп операция слияния капиталов должна проводиться государственной брокерской службой, которая не может производить более одной операции слияния в одной фирме одновременно. Вторая проблема состоит в том, что брокерская служба берет за свои услуги один процент всех средств, получившихся в результате слияния двух подразделений. Важно спланировать порядок операций слияния таким образом, чтобы фирма потратила на слияние как можно меньшую сумму.

Формат входных данных:

На вход программы подается число отделение $2 \leq N \leq 1000000$, за которым следует N капиталов отделений $1 \leq C_i \leq 1000000$.

Формат выходных данных:

T – минимальная сумма из возможных, которую должна заплатить брокерам фирма «Джава», с двумя знаками после запятой.

Задание № 3. Пересечение множеств

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

Задано $2 \leq N \leq 1000$ множеств из $3 \leq M \leq$ элементов, значения которых могут находиться в диапазоне от -2000000000 до 2000000000. Значения каждого множества задаются в произвольном порядке. Гарантируется, что для одного множества все задаваемые значения различные.

Требуется найти наибольший размер множества, получаемого при пересечении какой-либо из пар из заданных множеств.

Формат входных данных:

$N \ M$

$A_1[1] \ A_1[2] \ \dots \ A_1[M]$

$A_2[1] \ A_2[2] \ \dots \ A_2[M]$

...

$A_N[1] A_N[2] \dots A_N[M]$

Формат выходных данных:

Одно целое число.

Задание № 4. Составные строки

Ограничение по времени: 0.5 секунд. Ограничение по памяти: 16 мегабайт

В первой строке входного файла содержится число $4 \leq N \leq 1000$, последующие N строк состоят только из заглавных букв латинского алфавита и образуют множество, то есть, равных элементов среди них нет. Длина строки не превышает 1000 букв. Некоторые из этих строк можно составить, прописав друг за другом каких-то два элемента множества, возможно, совпадающие. То есть, если исходное множество содержит элементы A, B, AB, AA, C, ABC , то элемент AB можно составить из элементов A и B , а строку AA – из элементов A и A .

Ваша задача – вывести все элементы множества, которые можно составить из пары элементов множества по одному на строку в лексикографически возрастающем порядке.

Задание №5. Длинное деление

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

На вход подается три строки. Первая содержит представление длинного десятичного числа (первый операнд), вторая – всегда строка $/$, третья – десятичное представление второго операнда.

Длина первой и третьей строки ограничены 100000 символами. Вторая строка содержит ровно один символ $/$.

Требуется исполнить операцию и вывести результат в десятичном представлении.

Формат входных данных:

999

/

9

Формат выходных данных:

111

ГЛАВА 6. СБАЛАНСИРОВАННЫЕ И СПЕЦИАЛЬНЫЕ ДЕРЕВЬЯ В ЗАДАЧАХ ПОИСКА

6.1 Дисбаланс и повороты деревьев

Сложность всех алгоритмов в бинарных деревьях поиска определяется средневзвешенной глубиной. Операции вставки/удаления могут привести к дисбалансу и ухудшению средних показателей. Для борьбы с дисбалансом применяют рандомизацию и балансировку.

Создание BST из упорядоченной последовательности чревато крайне несбалансированным деревом. Возникает вопрос: а что будет, если вставлять новые элементы не в терминальные узлы дерева, а заменять ими корень? Последствия: если вставляемый элемент больше корня, то старый корень сделаем левым поддеревом, а его правое поддерево — нашим правым поддеревом. Аналогично рассуждаем для случая, когда вставляемый элемент меньше корня. К сожалению, и в том, и в другом случае может нарушиться упорядоченность. Чтобы нарушений не происходило, требуется сохранять инвариант упорядоченности. Для этого введём понятие поворота, не изменяющего упорядоченные свойства дерева, но меняющего его структуру, то есть, высоту поддеревьев.

Пусть дерево перед поворотом выглядит так, как показано на рис. 6.1. После поворота дерево выглядит так, как показано на рис. 6.2.

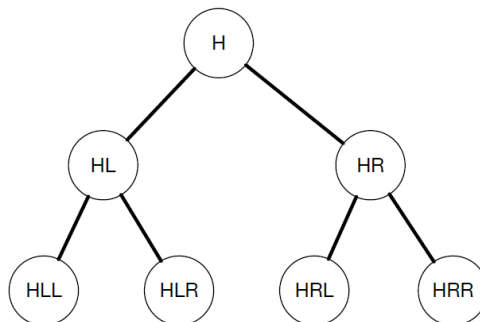


Рис. 6.1.

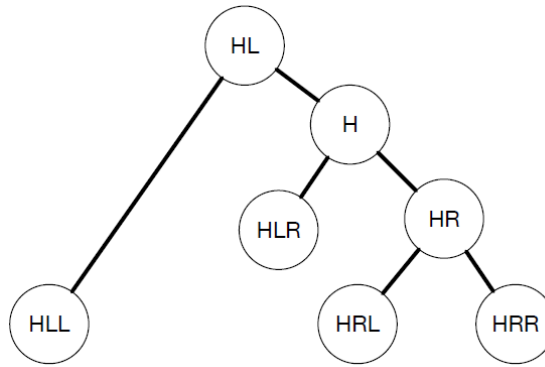


Рис. 6.2.

Обратите внимание на то, что упорядоченность узлов сохранена, меньшие значения ключей всё ещё находятся в левых поддеревьях, а большие — в правых.

При реализации кода, связанного с деревьями, рекомендуется всегда рисовать где-нибудь на листке бумаги состояния до операции и после операции. После такой подготовки требуемый код прост.

```
static void rotateRight(ref BinaryTree<T> head)
{
    BinaryTree<T> temp = head.left;
    head.left = temp.right;
    temp.right = head;
    head = temp;
}

static void rotateLeft(ref BinaryTree<T> head)
{
    BinaryTree<T> temp = head.right;
    head.right = temp.left;
    temp.left = head;
    head = temp;
}
```

Каждая из функций поворота изменяет свой аргумент значением нового корня поддерева. Функция вставки нового узла в корень дерева рекурсивна.

```
void insert(BinaryTree<T> head, T x)
{
    if (head == null)
    {
        head = new BinaryTree<T>(x);
    }
    if (x.key < head.item.key)
    {

```

```

        insert(head.left, x);
        rotateRight(head);
    }
    else
    {
        insert(head.right, x);
        rotateLeft(head);
    }
}

```



Создайте новый проект. Скопируйте в него класс бинарного дерева, созданный в упражнениях прошлых двух глав. Реализуйте методы левого и правого поворота, а также вставки.

6.2 Декартовы деревья

Случайные бинарные деревья поиска близки к идеальным по сложности ($H = O(\log N)$). Можно внести ещё более серьёзный элемент случайности, добавив второй ключ, генерируемый случайно.



Декартово дерево (treap) – это комбинация бинарного дерева поиска (BST) и бинарной кучи (BH). При поиске информации декартово дерево – BST. Вставка и упорядочивание узлов происходит по отношениям BH (рис. 6.3).

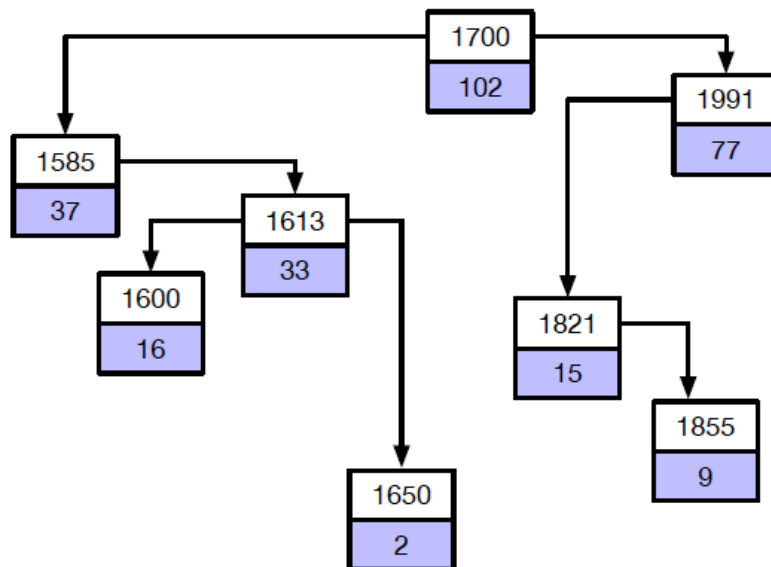


Рис. 6.3. Декартово дерево

При вставке в BST можно получить определяемое комбинаторикой количество различных деревьев, содержащих те же самые элементы. При вставке в BST с вторичным упорядочиванием по отношениям ВН получается единственное дерево со свойствами случайного BST.

Перечислим основные операции над Декартовыми деревьями.

1. find — Декартово дерево есть BST. Сложность поиска равна $O(\log N)$.

2. insert — Декартово дерево есть BST + ВН (рис. 6.4):

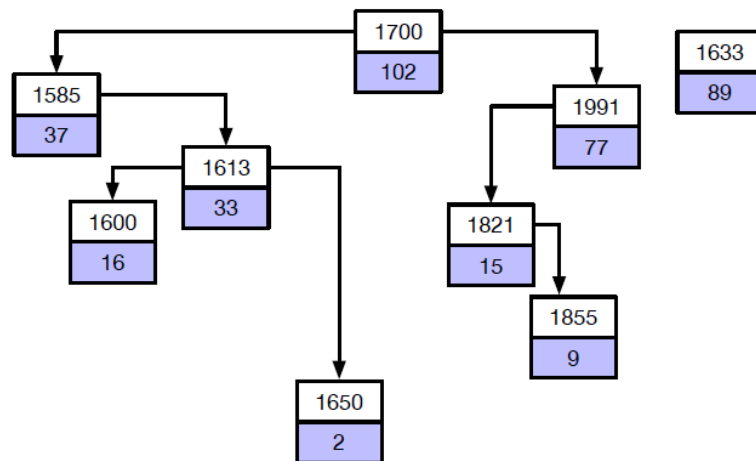


Рис. 6.4.

— первичная вставка проводится в BST. При этом может быть нарушены свойства ВН (рис. 6.5);

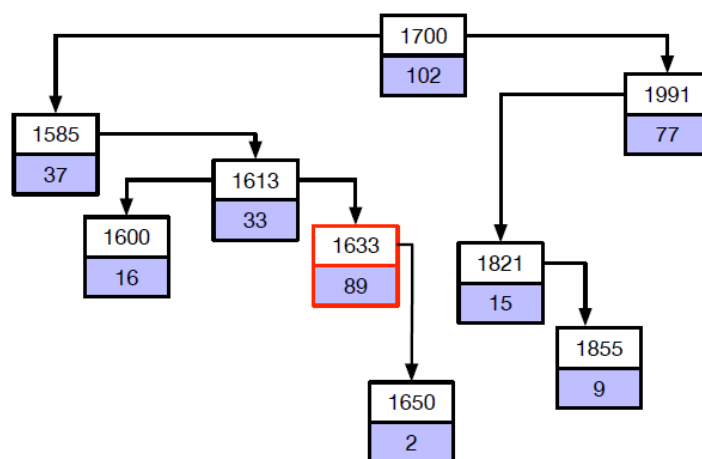


Рис. 6.5.

— если вставленный элемент не нарушает свойства (ВН), то вставка завершена;

– если свойства ВН нарушаются, проводится вращение, поднимающее вставленный элемент (рис. 6.6);

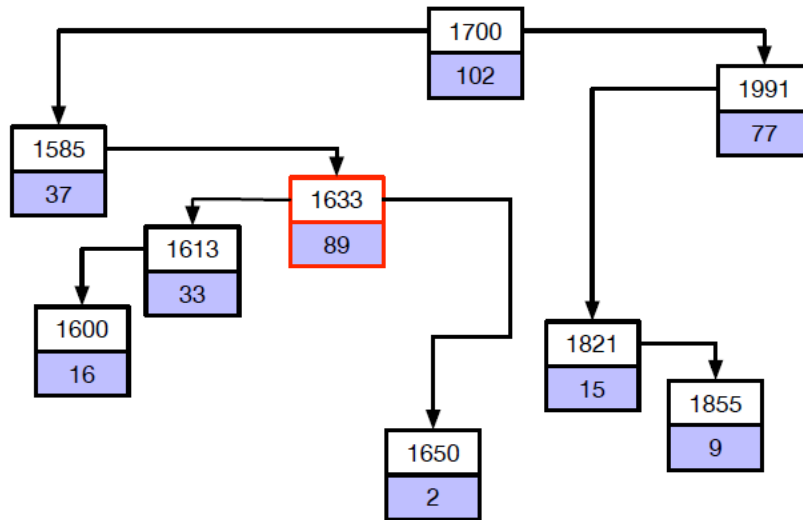


Рис. 6.6.

– подъём происходит до тех пор, пока нарушены свойства (ВН) (рис. 6.7).

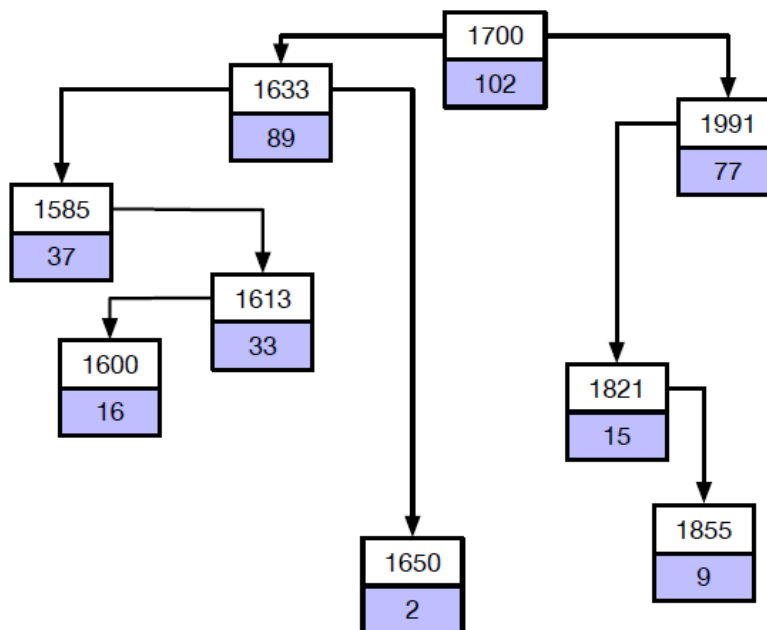


Рис. 6.7.

Здесь и далее будем применять выражение для какого-то узла «вращение в сторону родителя». Если искомый узел – левый подузел у родителя, то вращение в сторону родителя есть правый поворот с корнем в родителе. Если искомый узел – правый подузел у родителя, то вращение в сторону родителя есть левый поворот с корнем в родителе.

3. **remove** – Декартово дерево есть BST + ВН.

Так как удаление узлов, отличных от вершин, нетривиально, а удаление вершин – тривиально, задача – сделать удаляемый узел терминальным. Для этого на каждом шаге вращаем удаляемый узел с его ребёнком, имеющим наибольшее значение у до тех пор, пока он не станет терминальной вершиной. На этапе спуска мы не обращаем внимания на сохранение свойства ВН, нас интересуют только значения u .



Продолжайте работу в созданном проекте. Создайте класс декартова дерева. Реализуйте в нем три метода: поиска, вставки и удаления элемента.

6.3 Сбалансированные деревья поиска

6.3.1 Два вида сбалансированных деревьев поиска

Добиться хороших оценок времени исполнения операций с BST можно и без использования случайности. Для этого требуется при операциях избегать тех преобразований структуры деревьев, которые приводят к их вырождению. Это можно сделать, измеряя высоты поддеревьев и делая повороты при необходимых условиях, балансировать деревья.

Поставим следующую задачу: реализовать операции с деревьями, имеющие время в худшем $\Theta(\log N)$. Для этого требуется сохранять высоту дерева H в определённых границах. Чтобы сравнить различные стратегии балансировки, будем сравнивать высоту H , выраженную формулой $H < A \cdot \log_2 N + B$, где A и B – некоторые фиксированные константы. Если обозначить за H_{ideal} высоту идеально сбалансированного дерева, а через H_{algo} – наибольшую из возможных высот при реализации выбранной стратегии балансировки, то в первую очередь нас будет интересовать константа $A = H_{algo} / H_{ideal}$ — отношение этих высот.

Рассмотрим несколько критериев сбалансированности и вычислим для них коэффициенты А и В.

Сбалансированное дерево №1.

Для любого узла количество узлов в левом и правом поддереве N_l , N_r отличаются не более, чем на 1. $N_r \leq N_l + 1$, $N_l \leq N_r + 1$. Это — идеально сбалансированное дерево.

Пусть $H_{ideal}(N)$ — максимальная высота идеально сбалансированного дерева. Если N — нечётно и равно $2M + 1$, тогда левое и правое поддерева должны содержать ровно по M вершин.

$$H_{ideal}(2M + 1) = 1 + H_{ideal}(M).$$

Если N — чётно и равно $2M$. Тогда

$$H_{ideal}(2M) = 2 + \max(H_{ideal}(M - 1), H_{ideal}(M)).$$

Так как $H_{ideal}(M)$ — неубывающая функция, то

$$H_{ideal}(2M) = 1 + H_{ideal}(M),$$

$$H_{ideal}(N) \leq \log_2 N.$$

Это означает, что ключевой коэффициент А равен 1.

Сбалансированное дерево №2.

Для любого узла количество подузлов в левом и правом поддеревах удовлетворяют условиям $N_r \leq 2N_l + 1$, $N_l \leq 2N_r + 1$.

Примерная сбалансированность количества узлов. Пусть $H(M)$ — максимальная высота сбалансированного дерева со свойством 2. Тогда $H(1) = 0$, $H(2) = H(3) = 1$.

При добавлении узла один из узлов будет корнем, остальные $N - 1$ распределятся в отношении $N_l : N_r$, где $N_l + N_r = N - 1$.

Не умаляя общности, предположим, что $N_r > N_l$, тогда $N_r \leq 2N_l + 1$.

$$H(N) = \max_{N_l, N_r} (1 + \max(H(N_l), H(N_r)))$$

Функция $H(N)$ — неубывающая, поэтому

$$H(N) = 1 + H(\max(N_r, N_l)).$$

При ограничениях $N_r \leq 2N_l + 1$ и $N_l + N_r = N + 1$ получаем

$$H(N) = 1 + H\left(\left\lceil \frac{2N-1}{3} \right\rceil\right),$$

$$H(N) > 1 + H\left(\left\lfloor \frac{2N}{3} \right\rfloor\right),$$

$$H(N) > \log_{3/2} N + 1 \approx 1.71 \log_2 N + 1$$

6.3.2 AVL-деревья

Для любого узла высоты левого и правого поддеревьев H_l , H_r удовлетворяют условиям $H_r \leq H_l + 1$, $H_l \leq H_r + 1$

Это — примерная сбалансированность высот. Название этих деревьев — AVL-деревья (взято из первых буквы фамилий их изобретателей Георгия Максимовича Адельсона-Вельского и Евгения Михайловича Ландиса).

Пусть $N(H)$ — минимальное число узлов в AVL-дереве с высотой H (минимальное AVL-дерево) и левое дерево имеет высоту $H-1$. Тогда правое дерево будет иметь высоту $H-1$ или $H-2$. Так как $N(H)$ — неубывающая, то для минимального AVL-дерева высота правого равна $H-2$.

Число узлов в минимальном AVL-дереве: $N(H) = 1 + N(H-1) + N(H-2)$

$$\lim_{h \rightarrow \infty} \frac{N(h+1)}{N(h)} = \Phi = \frac{\sqrt{5} + 1}{2},$$

$$H(N) \approx \log_{\Phi}(N-1) + 1 \approx 1.44 \log_2 N + 1$$

6.3.3 Красно-черные деревья

Красно-чёрное дерево — это сбалансированное бинарное дерево поиска, которое в качестве критерия балансировки использует цвет узлов (рис. 6.8).

- Вершины разделены на красные и чёрные.
- Каждая вершина хранит поля ключ и значение.
- Каждая вершина имеет указатель left, right, parent.
- Отсутствующие указатели помечаются указателями на фиктивный узел

nil.

- Каждый лист `nil` — чёрный.
- Если вершина — красная, то её потомки — чёрные.
- Все пути от корня `root` к листьям содержат одинаковое число чёрных вершин. Это число называется чёрной высотой дерева, `black height`, `bh(root)`.

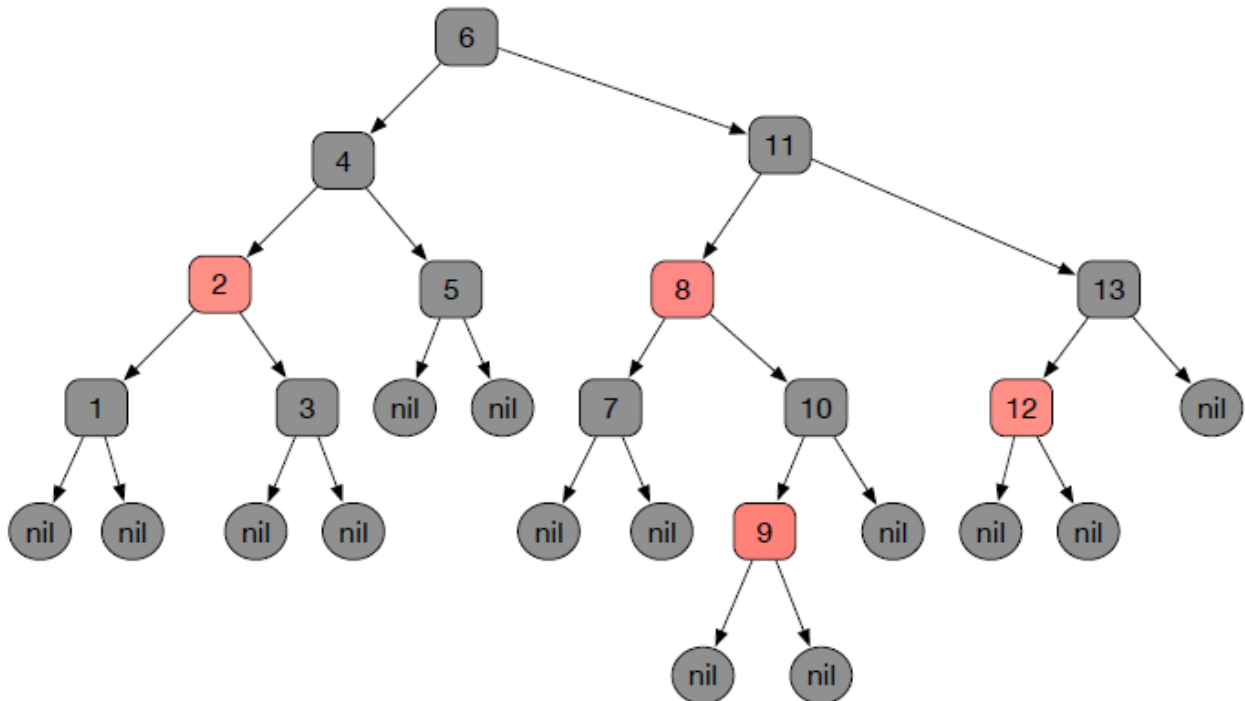


Рис. 6.8.

Найдём коэффициенты A и B для формулы высоты этого дерева в зависимости от числа узлов.

Для листьев чёрная высота равна нулю, $bh(x) = 0$. Обозначим через $|T_x|$ количество узлов в дереве, с чёрной высотой, равной x

Докажем, что $|T_x| > 2^{bh(x)}$.

– База индукции: Пусть вершина x является листом. Тогда $bh(x) = 0$ и $|T(x)| = 1 > 2^0$.

– Пусть вершина x не является листом и $bh(x) = k$. Тогда для обоих потомков $bh(l) > k-1$, $bh(r) > k-1$, т. к. красный будет иметь высоту k , чёрный — $k-1$.

– По предположению индукции $|T_l|, |T_r| > 2^{k-1} \rightarrow |T_k| = |T_l| + |T_r| > 2^k$

По свойству (3) не менее половины узлов составляют чёрные вершины, $bh(t) > H_{rb}/2$. Отсюда $N > 2^{H_{rb}/2} - 1$ и значит

$$H_{rb} \leq 2 \times \log_2 N + 1$$



Продолжайте работу в созданном проекте. Создайте классы для четырех рассмотренных типов сбалансированных деревьев.

6.4 Внешний поиск и В-деревья

Современные компьютеры в качестве носителей информации в основном используют два типа устройств — накопители, использующие магнитные пластины, HDD, и накопители, использующие флэш-память, SSD.

На HDD информация располагается в секторах, которые логически расположены на дорожках. Типичный размер сектора 512/2048/4096 байт. Информация считывается и записывается головками чтения/записи. Для чтения/записи информации требуется подвести головку чтения записи к нужной дорожке и дождаться подхода нужного сектора. Типичные скорости вращения HDD — 5400/7200/10033/15000 оборотов в минуту. Один оборот совершается за время от 1/90 до 1/250 секунды. Операция перехода на соседнюю дорожку составляет примерно 1/1000 секунды.

Внешние сортировки используют многократный последовательный проход по данным, расположенным на носителях информации. Последовательное считывание информации с HDD типично 100-250 МБ/сек.

Смена позиции в файле часто требует:

- ожидания подвода головки на нужную дорожку;
- ожидания подхода нужного сектора к головкам чтения/записи.

Операция последовательного чтения 4096 байт занимает $4096/100 \times 10^6 \approx 40 \times 10^{-6}$ секунд.

Операция случайного чтения 4096 байт занимает не менее $5 - 10 \times 10^{-3}$ секунд.

На SSD информация хранится в энергонезависимой памяти на микросхемах. Операции производятся блоками размером 64-1024 КБ. Время доступ к блоку $\approx 10^{-6}$ секунд. HDD и SSD используют буферизацию для ускорения работы.

Алгоритмы поиска во внешней памяти должны минимизировать число обращений к внешней памяти.

На логическом уровне обращения происходят блоками любого размера, кратного 512 байт. На физическом уровне всё сложнее. Размер физического блока от 64 до 1024 КБ. При операции частичной записи 512 байт:

1. Считывается полный блок (всегда).
2. Заменяется 512 байт в требуемом месте.
3. Записывается полный блок (всегда).

При выровненной записи целого блока никаких добавочных действий не производится

А так ли необходимы структуры данных, хранящиеся на внешних носителях? Оценим, какое количество записей можно обработать, имея 16 Гб оперативной памяти, при использовании бинарного дерева поиска, состоящего из данных размером 64 байта, ключа размером 8 байт и указателей left и right размером 8 байт. Общий размер узла — 88 байт (не считая многочисленных накладных расходов на системные структуры данных, используемых в менеджере памяти для операций malloc/new). При заданных условиях можно будет обработать $16 \times 2^{30} / 88 \approx 195 \times 10^6$ узлов, что не так уж и много.



В-дерево — сбалансированное дерево поиска, узлы которого хранятся во внешней памяти.

1. Каждый узел содержит:
 - количество ключей n , хранящихся в узле.
 - индикатор листа final.
 - n ключей в порядке возрастания.
 - $n+1$ указатель на детей, если узел не корневой.

2. Ключи есть границы диапазонов ключей в поддеревьях.
3. Все листья расположены на одинаковой глубине h .
4. Вводится показатель t — минимальная степень дерева.
5. В корневом узле от 1 до $2t-1$ ключей.
6. Во внутренних узлах минимум $t-1$ ключей.
7. Во внешних узлах максимум $2t-1$ ключей.
8. Заполненный узел имеет $2t-1$ ключ.

Такие хранилища называют персистентными. Из соображений эффективности часть информации хранится в оперативной памяти (рис. 6.9).

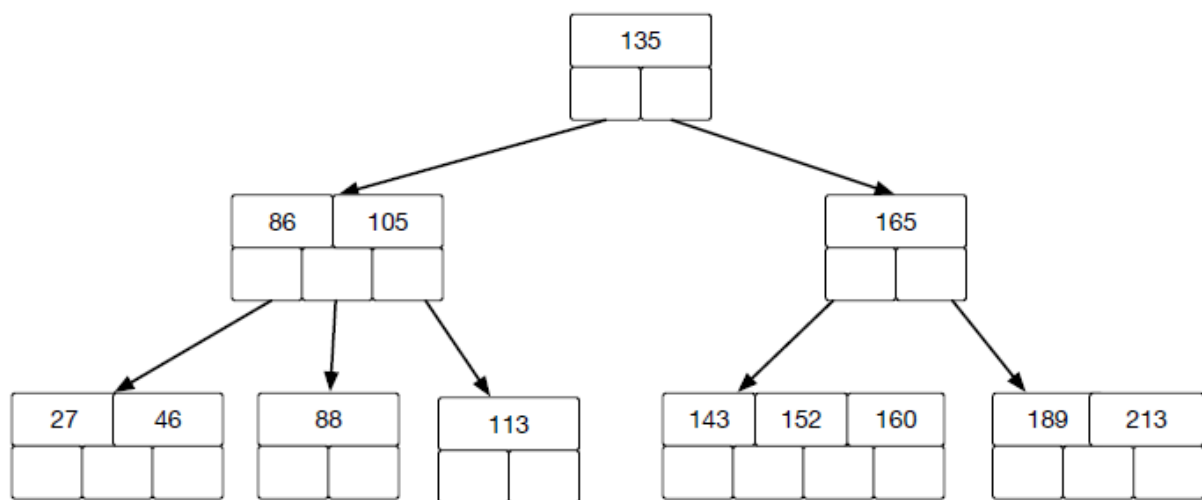


Рис. 6.9.

Высота В-дерева с $n > 1$ ключами и минимальной степенью $t > 2$ в худшем случае не превышает $\log_t((n + 1)/2)$

Для операций с внешней памятью используем абстракцию storage с операциями Load и Store. Корневой узел сохраняем в оперативной памяти. Наша цель — минимизировать количество операций обмена с внешней памятью. Размер блоков в операциях Load и Store можно подобрать в соответствии с физическим размером блока на носителе.

Операция find в В-дереве:

1. Операцией бинарного поиска ищем самый левый ключ $key_i \geq k$.
2. Если $key_i = k$, то узел найден.

3. Исполняем Load для дочернего узла и рекурсивно повторяем операцию. 4. Если $final = true$, то ключ не найден.

Количество операций с носителем $T_{load} = O(h) = O(\log_t N)$.

Операция вставки в B-дерево:

1. Операцией find находим узел для вставки.
2. Если лист не заполнен, сохраняя упорядоченность вставляем ключ.
3. Если лист заполнен (содержит $2t-1$ ключей), разбиваем его на два листа по $t-1$ ключу поиском медианы.
4. Медиана рекурсивно вставляется в родительский узел.

Сложность в худшем случае: почти заполненный узел на каждом уровне каждый раз разбивается на два. Тем не менее, количество таких операций не превосходит высоты дерева

Количество операций: $T_{ext} = O(h) = O(\log_t n)$.

B^+ -дерево содержит информацию только в листьях, а ключи присутствуют только во внутренних узлах. Это — очень популярная структура данных, используемая в файловых системах XFS, JFS, NTFS, Btrfs, HFS, APFS и многих других. Ещё одна область использования этой структуры данных — хранение индексов в базах данных Oracle, Microsoft SQL, IBM DB2, Informix и других.

Контрольные вопросы

1. Что означает дисбаланс дерева?
2. Как можно повернуть дерево?
3. Что такое Декартово дерево?
4. Какие операции можно совершить с Декартовым деревом?
5. Назовите известные вам виды сбалансированных деревьев поиска.
6. В чем особенность АВЛ-дерева?
7. По какому принципу закрашиваются вершины в красно-черном дереве?

8. Для чего нужен внешний поиск?
9. Что такое В-дерево?
10. Какие хранилища называются персистентными?
11. Как реализуется операция поиска в В-дереве?
12. Как реализуется операция вставки в В-дереве?

Задания для самостоятельного выполнения

Задание № 1. Запросы сумм

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

В первой строке файла содержатся два числа: количество элементов в массиве V : $10 \leq N \leq 500000$ и количество запросов $1 \leq M \leq 500000$. Каждый элемент массива лежит в интервале $[0 \dots 2^{32})$.

Каждый запрос – отдельная строка, состоящая из кода запроса, который может быть равен 1 или 2 и аргументов запроса.

Запрос с кодом один содержит два аргумента, начало L и конец отрезка R массива. В ответ на этот запрос программа должна вывести значения суммы элементов массива от $V[L]$ до $V[R]$ включительно.

Запрос с кодом два содержит тоже два аргумента, первый из которых есть номер элемента массива V , а второй – его новое значение.

Количество выведенных строк должно совпадать с количеством запросов первого типа.

Задание № 2. Поиск множеств

Ограничение по времени: 1 секунда. Ограничение по памяти: 64 мегабайта

В первой строке файла содержится три числа: N – количество эталонных множеств, M – размер каждого из множеств и K – количество пробных множеств.

Каждое из множеств содержит целые числа от 0 до 10^9 , числа могут повторяться.

Требуется для каждого из пробных множеств вывести в отдельной строке цифру '1', если это множество в точности совпадает с каким-либо из эталонных множеств и цифру '0', если оно ни с одним не совпадает, то есть выведено должно быть в точности K строк.

$$5 \leq N \leq 50000$$

$$3 \leq M \leq 1000$$

$$5 \leq K \leq 50000$$

Задание №3. Телефонная книга

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

Необходимо разработать программу, которая является промежуточным звеном в реализации телефонной книги. На вход подается $N \leq 1000$ команд вида

ADD User Number

DELETE User

EDITPHONE User Number

PRINT User

Согласно этим командам нужно соответственно добавить пользователя в телефонную книгу, удалить пользователя, изменить его номер и вывести на экран его данные. В случае невозможности выполнить действие, необходимо вывести **ERROR**. Добавлять пользователя, уже существующего в телефонной книге, нельзя.

Необходимо вывести протокол работы телефонной книги

Формат входных данных:

```
9
ADD IVAN 1178927
PRINT PETER
ADD EGOR 123412
PRINT IVAN
EDITPHONE IVAN 112358
PRINT IVAN
PRINT EGOR
DELETE EGOR
EDITPHONE EGOR 123456
```

Формат выходных данных:

В случае невозможности выполнения действия требуется вывести **ERROR**

В случае команды **PRINT** User: User Number

Задание №4. Анаграммы

Ограничение по времени: 0.5 секунд. Ограничение по памяти: 256 мегабайт

Как известно, анаграммами называются слова, которые могут получиться друг из друга путем перестановки букв, например LOOP, POOL, POLO. Будем называть все слова такого рода *комплект*ом.

На вход программы подается число слов $1 \leq N \leq 100000$. В каждой из очередных N строк присутствует одно слово, состоящее из заглавных букв латинского алфавита. Все слова имеют одинаковую длину $3 \leq L \leq 10000$.

Требуется определить число комплектов во входном множестве.

Формат входных данных:

N

W1

W2

...

WN

Формат выходных данных:

NumberOfComplects

Задание №5. Кеширующий сервер

Ограничение по времени: 3 секунды. Ограничение по памяти: 64 мегабайта

Игорь работает в компании, которая занимается обработкой больших данных. Обработываемые данные находятся где-то в распределенной системе. Количество различных данных в системе ограничено и каждое данное имеет свой номер. Эти данные регулярно требуются различным клиентам и,

поскольку время обращения к ним достаточно велико, для ускорения обработки информации Игорю поручено написать часть middleware – сервер-посредник, к которому и обращаются теперь клиенты за данными. Так как система – распределенная, а сервер – нет, все требуемые данные на сервер не помещаются, но он имеет возможность запоминать результаты своих запросов к распределенной системе. Для этого на сервере выделена ограниченная память на N запросов. Важно, что клиент не имеет возможности обращаться к распределенной системе и результат запроса к распределенной системе всегда должны оказаться на сервере.

К большой радости Игоря оказалось, что самые крупные и значимые клиенты всегда обращаются за одними и теми же данными в одном и том же порядке, так что у него есть последовательность запросов. Игорь придумал такой алгоритм, что как можно большее количество запросов исполняется из кеша сервера, без обращения к распределенной системе. Придумаете ли вы что-то подобное?

Формат входных данных:

На вход программы подается размер памяти под кеширование запросов $1 \leq N \leq 100000$, количество запросов $1 \leq M \leq 100000$ и ровно M запросов с номерами $0 \leq R_i \leq 10^{18}$. Количество различных номеров запросов ограничено и не превосходит 100000.

Формат выходных данных:

Требуется вывести одно число: сколько раз пришлось обратиться к распределенной системе за данными, отсутствующими в кеше. В начале работы кеш пуст.

ГЛАВА 7. ХЕШИРОВАНИЕ

7.1 Обобщенный быстрый поиск

Структуры данных, рассмотренные в предыдущей главе настоящего учебного пособия, имеют логарифмическую сложность всех основных операций. Естественно, возникает желание эту сложность уменьшить. Двоичный поиск является достаточно хорошим и быстрым алгоритмом. Но возникает вопрос, можно ли осуществить поиск быстрее? Если посмотреть на толстый бумажный словарь, то можно увидеть подсказку – в некоторых словарях на торце книги имеются отметки для букв. Последуем их примеру.

Для более быстрого поиска в словаре, содержащем названия городов и их население, можно разбить общее хранилище на 33 более маленьких, по одному связному списку на букву (рис. 7.1).

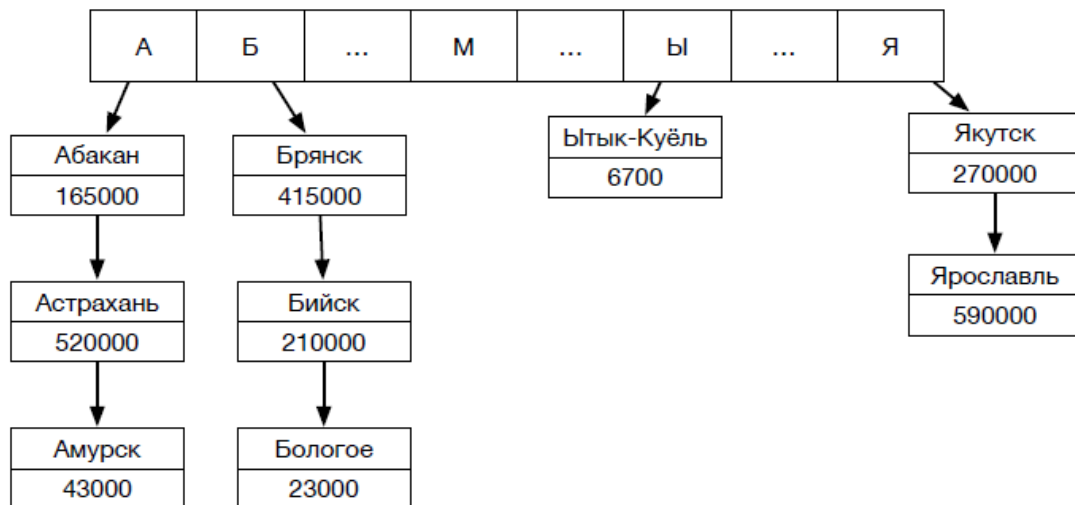


Рис. 7.1.

Попробуем теперь заменить связные списки на изученные в предыдущих главах деревья поиска (рис. 7.2).

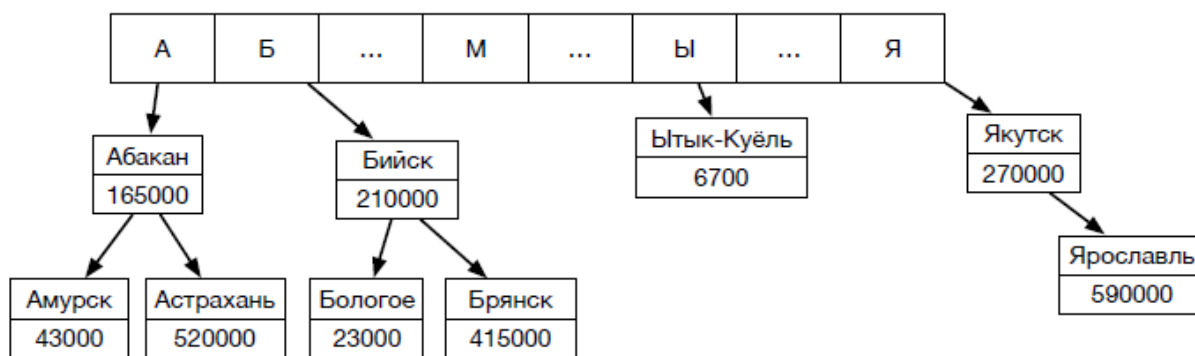


Рис 7.2.

В итоге получается 33 сбалансированных дерева. Очевидно, что и в том и в другом случае поиск потребует меньшую сложность. Но такое прямолинейное разбиение не вполне хорошо — на мягкий и твёрдый знак названий нет, для некоторых букв названий совсем мало, некоторые буквы очень популярны.

Основная идея обобщенного поиска — разбиение пространства ключей на независимые подпространства (partitioning). При независимом разбиении на M подпространств сложность поиска уменьшается.

Для разбиения множества N ключей на примерно равные M подмножеств сложность вычисляется по главной теореме о рекурсии при числе подзадач M , коэффициенте размножения 1 и консолидации $O(1)$.

$$C \times O(N) \rightarrow \frac{C}{M} O(N)$$

$$C \times O(N \log N) \rightarrow \frac{C}{M} O(N \log N)$$

При увеличении M время поиска уменьшается? а требуемая память увеличивается. При $M \approx N$ имеется зона оптимальности — поиск уже будет проводиться за $O(1)$, а вот потребная память ещё не столь велика — $O(N)$.

Примитивное разбиение пространства ключей по первым буквам — не очень хороший вариант. Требуется иметь детерминированный способ разбиения пространства ключей на M независимых подпространств. Условия разбиения — мощность множеств ключей, принадлежащих каждому

подпространству, должна быть примерно равна. Отсюда, возникает потребность в создании функции $H(K)$, удовлетворяющую некоторым условиям.

7.2 Хеш-функции



Хеш-функция есть функция преобразования множество ключей K на множество V мощностью M .

$$H(K) \rightarrow V$$

$$|D(V)| = M$$

Введём понятие *соперника*, то есть того, кто предоставляет нам ключи. Цель соперника – предоставлять ключи таким образом, чтобы значения функции оказались не равновероятными. Соперник знает хеш-функцию и может выбирать ключи. Чтобы быть независимыми от соперника и для удобства практического применения, для функции $H(K)$ необходимо обеспечить следующие свойства:

1. *Эффективность*.

$$T(H(K)) \leq O(L(K)),$$

где $L(K)$ — мера длины ключа K . Время вычисления хеш-функции не должно быть велико.

2. *Равномерность*. Каждое выходное значение равновероятно.

$$pH(K_1) = pH(K_2) = \dots = pH(K_M)$$

3. *Лавинность*. При незначительном изменении входной последовательности выходное значение должно меняться значительно, иначе соперник может просто подобрать ключ.

4. *Необратимость*, то есть невозможность восстановления ключа по значению его функции.

Отсюда вытекают следующие следствия:

– Функция не должна быть близка к непрерывной. Неплохо было бы, если для близких значений аргумента получаются сильно различающиеся результаты.

– В значениях функции не должно образовываться кластеров, множеств близко стоящих точек.

Примеры плохих функций:

– $H = K^2 \bmod 10000$ для $K < 100$. Функция монотонно возрастает.

Пространство значений ключа слишком велико и часть значений недостижима.

– $H = \sum_{i=0}^{s.size()-1} s[i]$ для строки s . Функция даёт одинаковые значения для строк $abcd$ и $abdc$ и отличающиеся на единицу для строк $abcd$ и $abde$. Сопернику легко найти ключи, которые дают равные значения функции.



Совпадение значений функции для разных значений ключа называется *коллизией*. Большое количество коллизий для данного множества ключей (больше $1/|D(M)|$) говорит о том, что хеш-функция плохая.

Введём H^* – множество хеш-функций, которые отображают пространство ключей в $m = |D(M)|$ различных значений



Множество хеш-функций *универсально*, если для каждой пары ключей $K_i, K_j, i \neq j$ количество хеш-функций, для которых $H^*(K_i) = H^*(K_j)$ не более $|H^*| / m$.

При наличии такого универсального множества борьба с соперником закончится победой, если каждый раз мы будем выбирать случайным образом функцию из этого множества. Ведь, если случайным образом выбирается функция из множества H^* , то для случайной пары ключей $K_i, K_j, i \neq j$ вероятность коллизии не должна превышать $1 / m$.



Пусть множество $Z_p = \{0, 1, \dots, p-1\}$, множество $Z_p^* = \{1, 2, \dots, p-1\}$, p – простое число, $a \in Z_p^*, b \in Z_p$. Тогда множество

$$H^*(p, m) = \{H(a, b, K) = ((aK + b) \bmod p) \bmod m\}$$

есть универсальное множество хеш-функций.

Обратим внимание на то, что для хорошей универсальной функции множество Z_p есть множество неотрицательных целых чисел, что не всегда совпадает с нашими запросами: часто требуется получить значение хешфункции (часто говорят: получить хеш) от строк, объектов и прочих сущностей. Вполне достаточно рассматривать значение ключа как последовательность битов, только трактовать эту последовательность придётся как целое число соответствующей разрядности, то есть, перейти к операциям над (N)-числами. Для ключей с большой длиной это будет весьма неэффективно. Поэтому на практике часто применяют специальные, не универсальные хеш-функции.

Для строки можно использовать вариант полиномиальной хеш-функции:

$$h = \sum_{i=0}^n s_i \times q^i \pmod{HASHSIZE}$$

Как и все полиномы, её быстрее вычислить по схеме Горнера.

```
int HashSum(string s, int q, int HASHSIZE)
{
    int sum = 0;
    for (int i = 0; i < s.Length; i++)
    {
        sum = sum * q + s[i];
    }
    return sum % HASHSIZE;
}
```

В книге [6] предлагается хеш-функция, основанная на идеях из генерации случайных чисел:

```
int HashSedgwick(string s, int HASHSIZE)
{
    int h, i, a = 31415, b = 27183;
    for (h = 0, i = 0; i < s.Length; i++, a = a * b %
(HASHSIZE - 1))
    {
        h = (a * h + s[i]) % HASHSIZE;
    }
    return h;
}
```

Лучшие по статистическим показателям функции – хеш-функции, применяемые в криптографии. К сожалению, у них есть и свои недостатки: у них длинный код и они достаточно медленные.

Одна из хороших и быстрых функций основана на полях Галуа. Это функция CRC, она очень популярна в архиваторах. Есть варианты с различным количеством бит.



Создайте новый проект. Опишите методы расчета хеш-функций: полиномиальную, Седжвика и CRC. Сравните их на наборах больших текстов.

7.3 Применение хеш-функций

7.3.1 Вероятностные множества

Надёжны ли современные вычислительные системы? Неужели они не отказывают совсем? Увы, сбои при обработке информации происходят, их вероятность мала, но всё же отлична от нуля.

Какая вероятность отказа при сравнении двух блоков памяти во 4096 байт? Вероятность получения неверного ответа при их равенстве есть $4096 / 10^{20} \approx 2.5 \cdot 10^{-16}$.

Если взять хорошую хеш-функцию, выдающую равномерно значения из множества в 2^{64} элементов, то вероятность совпадения значений этой хеш-функции для двух блоков данных размером в 4096 байт есть $4096 / 2^{64} = 2^{-52} \approx 10^{-17.1}$, то есть меньше. Для хеш-функции, выдающей значения из множества 2^{128} элементов вероятность коллизии будет порядка 10^{-35} , что несравнимо меньше вероятности аппаратного отказа. Таким образом с вероятностью, максимально близкой к достоверности, можно сказать, что для если для двух блоков данных хорошая «длинная» хеш-функция дала одно и то же значение, то эти блоки равны. В дальнейшем будем использовать этот факт.

Вероятностное множество – структура данных, реализующая функциональность абстракции множество, имеющая операции insert и find с отсутствием гарантии точности результата поиска в этом множества. Результаты поиска могут быть ложноположительными, если элемент отсутствует, но операция find вернула истину. Отсутствие элемента всегда определяется точно, то есть, ложноотрицательных результатов быть не может.

7.3.2 Алгоритм Карна-Рабина

Хеш-функции могут использоваться для ускорения поиска подстрок в строке. Задача формулируется так: имеется исходная строка и образец. Определить позицию в исходной строке, содержащую образец.

Пусть множество символов, из которых могут состоять исходная строка и образец ограничено символами A, B, C, D. Отобразим их в 1, 2, 3, 4.

Пусть строка-образец — pat=ABAC или 1213.

Строка-источник — src=ACABAACABACAABCA (рис. 7.3).

A	B	A	C
1	2	1	3

A	C	A	B	A	A	C	A	B	A	C	A	A	B	C	A
1	3	1	2	1	1	3	1	2	1	3	1	1	2	3	1

Рис. 7.3.

Выберем простое число, немного превышающее мощность алфавита $P = 5$. Составим таблицу (рис. 7.4) T степеней числа P по модулю 2^{32} .

0	1	2	3	4	5	6	7	8	9
1	5	25	125	625	3125	15625	78125	390625	1953125

10	11	12	13	14	15
9765625	48828125	244140625	1220703125	1808548329	452807053

Рис. 7.4.

Предлагаемая хеш-функция от строки S в поддиапазоне $[k...r]$ вычисляется следующим образом:

$$H(S_{[k,r]}) = \sum_{i=k}^r S_{i-k} \times P^{i-k} = \sum_{i=k}^r S_{i-k} \times T_{[i-k]}$$

Чтобы найти, имеется ли образец длиной 4 в исходной строке, сначала вычислим значение хеш-функции от образца, а затем от всех подстрок строки src длиной 4:

$$H(pat_{[0,3]}) = H(ABAC) = 0 \cdot 5^0 + 1 \cdot 5^1 + 0 \cdot 5^2 + 2 \cdot 5^3 = 411$$

Для подстрок:

$$H(src_{[0,3]}) = 291, H(src_{[1,4]}) = 183,$$

$$H(src_{[2,5]}) = 161, H(src_{[3,6]}) = 407,$$

$$H(src_{[4,7]}) = 206, H(src_{[5,8]}) = 291,$$

$$H(src_{[6,9]}) = 183, H(src_{[7,10]}) = 411,$$

$$H(src_{[8,11]}) = 207, H(src_{[9,12]}) = 166,$$

$$H(src_{[10,13]}) = 283, H(src_{[11,14]}) = 431.$$

Хеш-функция для выбранных строк выглядит следующим образом:

```
int Hash(string s, int l, int r, int[] ptab)
{
    int sum = 0;
    for (int i = l; i < r; i++)
    {
        sum += (s[i] - 'A' + 1) * ptab[i - l];
    }
    return sum;
}
```

Наивный поиск подстроки:

```
string s1 = ""; string s2 = "";
int[] ptab = new int[] { 1, 3, 5 };
int hs1 = Hash(s1, 0, s1.Length, ptab);
for (int i = 0; i < s2.Length - s1.Length; i++)
{
    int hs2 = Hash(s2, i, i + s1.Length, ptab);
    if (hs2 == hs1)
    {
        bool ok = true;
        for (int j = 0; ok && j < s1.Length; j++)
        {
```

```

        if (s1[j] != s2[i + j])
        {
            ok = false;
            break;
        }
    }
    if (ok)
    {
        Console.WriteLine("match at: {0}", i);
    }
}

```

Примем, что $N = \text{src.Length}$, $M = \text{pat.Length}$. Сложность алгоритма тогда составит $O(NM)$, что слишком много. Такой неэффективный результат обусловлен большим количеством избыточных действий. Очевидно, что:

$$H(s_{[0,4]}) = s_0 + s_1 \cdot p_1 + s_2 \cdot p_2 + s_3 \cdot p_3.$$

Попробуем применить индукцию и вычислить $H(s_{[1,4]})$. $H(s_{[1,5]}) = s_1 + s_2 \cdot p_1 + s_3 \cdot p_2 + s_4 \cdot p_3.$

$$\text{Умножим на } p^1: H(s_{[1,5]}) \cdot p_1 = s_1 \cdot p_1 + s_2 \cdot p_2 + s_3 \cdot p_3 + s_4 \cdot p_4.$$

$$\text{Сравним с } H(s_{[0,5]}) = s_0 + s_1 \cdot p_1 + s_2 \cdot p_2 + s_3 \cdot p_3 + s_4 \cdot p_4.$$

$$H(s_{[k,l]}) \cdot p^k = H(s_{[0,l]}) - H(s_{[0,k]}).$$

Выражение показывает нам, что достаточно вычислить значения хеш-функции от всех собственных префиксов строки src.

$$H(\text{src}_{[0,0]}) = 0, H(\text{src}_{[0,1]}) = 1$$

$$H(\text{src}_{[0,2]}) = 16, H(\text{src}_{[0,3]}) = 41$$

$$H(\text{src}_{[0,4]}) = 291, H(\text{src}_{[0,5]}) = 916$$

$$H(\text{src}_{[0,6]}) = 4041, H(\text{src}_{[0,7]}) = 50916$$

$$H(\text{src}_{[0,8]}) = 129041, H(\text{src}_{[0,9]}) = 910291$$

$$H(\text{src}_{[0,10]}) = 2863416, H(\text{src}_{[0,11]}) = 32160291$$

```

static int KarpRabin(string s1, string s2, int[] ptab)
{
    int hs1 = Hash(s1, 0, s1.Length, ptab);
    int[] htab = new int[s2.Length];
    for (int i = 1; i < s2.Length; i++)
        htab[i] = htab[i - 1] + (s2[i - 1] - 'A'
+1)*ptab[i - 1];
    for (int i = 0; i < s2.Length - s1.Length; i++)
    {
        int hs2 = htab[i + s1.Length] - htab[i];

```

```

        if (hs2 == hs1)
        {
            bool ok = true;
            for (int j = 0; j < s1.Length; j++)
                if (s1[j] != s2[i + j])
                {
                    ok = false;
                    break;
                }
            if (ok) return i;
        }
        hs1 *= 5;
    }
    return -1;
}

```

Важно, что при совпадении хеш-функций для фрагментов, необходимо сравнить эти фрагменты, то есть, применение хеш-функций даёт подсказки, где можно было бы найти совпадающие подстроки. Приведённый алгоритм может показаться избыточно сложным: ведь есть более простые алгоритмы Z- и префикс-функций, которые помогают быстро найти подстроку в строке. Оказывается, применённая здесь функция имеет свойство rolling-hash, то есть, достаточно дёшево обходятся операции добавления единичных элементов строки в её конец и удаления единичных элементов из начала строки. Ни Z-, ни префикс-функции этим свойством не обладают.



Реализуйте и рассмотрите работу алгоритма Карпа–Рабина для разных наборов входных данных.

7.4 Хеш-таблицы

7.4.1 Понятие хеш-таблиц

Простая хеш-таблица есть массив пар {ключ, значение}. Пусть размер этого массива будет HASHSIZE и хеш-функция от ключа будет давать числа в диапазоне [0..HASHSIZE). Тогда применение хеш-функции к ключу даст индекс в этом массиве, который поможет найти пару {ключ, значение}. Каким образом будет произведен поиск этой пары зависит от организации хеш-таблицы (рис. 7.5).

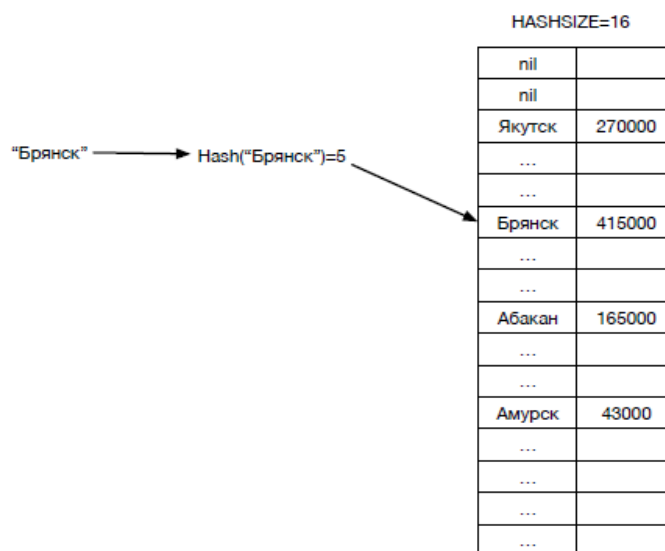


Рис. 7.5.

В простейшем варианте можно сразу найти необходимую пару. Например, для приведённой таблицы, если $\text{Hash}(\text{"Брянск"})=5$ и в 5-м элементе таблицы находится пара с ключом "Брянск", то поиск завершился успехом после первой же попытки.

Часто в таблице находятся не сами пары {ключ,значение}, а указатели на них, или на голову связного списка, содержащего пары. NULL в элементе таблицы означает, что элемент пуст (рис. 7.6).

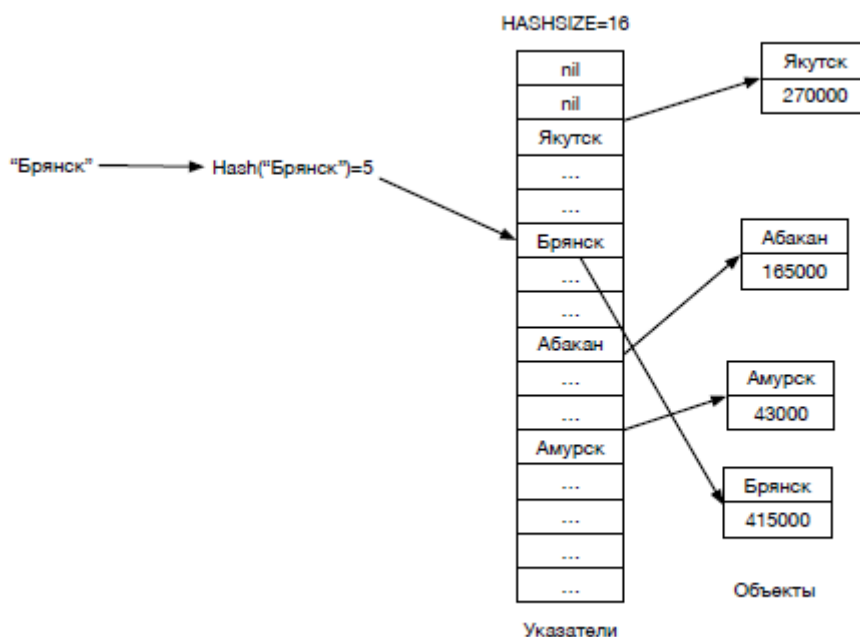


Рис. 7.6.

Введём несколько терминов: Если известно количество элементов в контейнере C и размер массива M , то $\alpha = C / M$ — *коэффициент заполнения*, fill-factor или load-factor. α — главный показатель хеш-таблицы.

Операция create для хеш-таблицы часто имеет аргумент, равный начальному количеству элементов массива. Массив заполняется либо NULL, либо парами с невозможным значением ключей — всегда необходимо знать, свободен ли слот для размещения пары.

Добавление элементов (операция insert) требует поиска (операции find). Если после нахождения значения хеш-функции от вновь прибывшего ключа оказывается, что запись с таким значением хеш-функции уже есть (например, $\text{Hash}(\text{"Якутск"}) = 2$ и $\text{Hash}(\text{"Мышкин"}) = 2$), то говорят, что произошла коллизия. Коллизий хотелось бы избежать, так как без них операции поиска и вставки имели бы сложность $O(1)$.

Для борьбы с коллизиями есть несколько способов организации хеш-таблиц. Это — прямая (закрытая) адресация и открытая адресация.

7.4.2 Хеш-таблицы с прямой адресацией

При коллизии во время создания элемента создаётся связный список конфликтующих. Технически можно создать любую поисковую структуру данных. Обычно стараются, чтобы количество элементов во вторичной структуре данных оставалось небольшим, поэтому простой односвязный список является хорошим выбором (рис. 7.7).

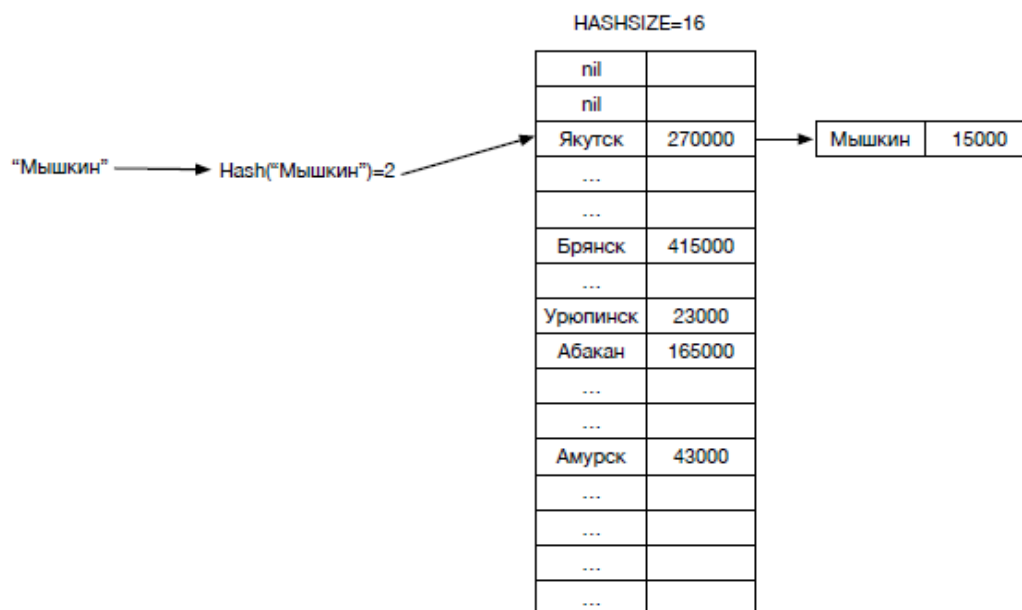


Рис. 7.7.

Операции поиска и вставки.

1. При поиске вычисляется значение хеш-функции от ключа.
2. По этому значению определяется место поиска — вторичная поисковая структура данных.
3. Если вторичной структуры нет, то нет и элемента, который мы ищем. Теперь, если это требуется, то создаётся вторичная структура данных и элемент вставляется в неё.
4. Иначе элемент ищется во вторичной структуре и вставляется туда при необходимости.

Операция удаления.

1. При удалении вычисляется хеш-функция от ключа.
2. Определяется место нахождения ключа — вторичная поисковая структура данных.
3. Если вторичной структуры нет, то нет и элемента.
4. Иначе элемент удаляется из вторичной структуры.
5. Если вторичная структура пуста, удаляется сама структура и точка входа в неё.

7.4.3 Хеш-таблицы с открытой адресацией

При другой организации хеш-таблиц вторичные структуры данных не используются и все пары ключ/значение хранятся в самой таблице. Это означает, что требуется удобный способ разрешения коллизий (рис. 7.8).

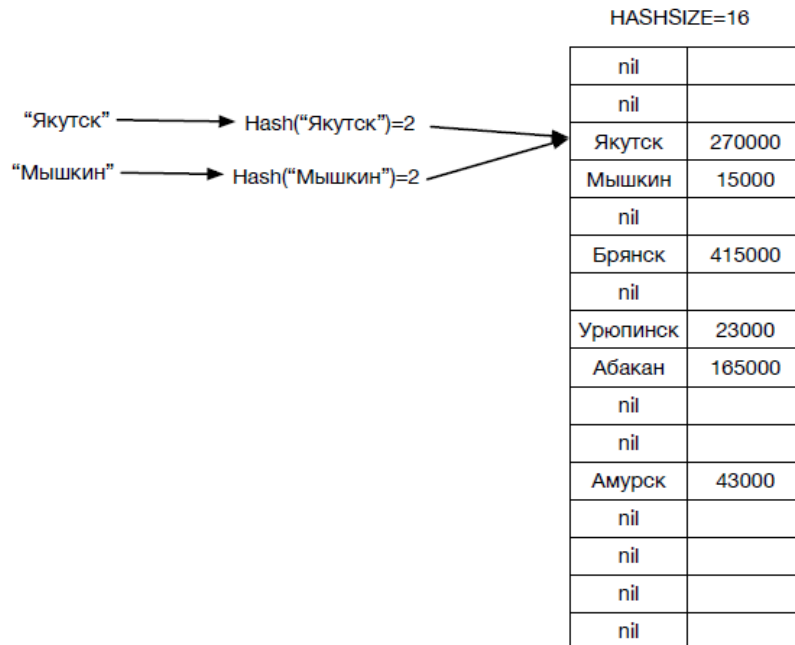


Рис. 7.8.

Процедуры поиска по ключу и вставки здесь немного различаются. Дело в том, что требуется понимать, что делать с удалённым элементом.

Операция поиска по ключу.

1. При поиске существующего элемента вычисляется хеш-функция от его ключа.
2. По значению хеш-функции определяется место поиска – индекс в хеш-таблице.
3. Если по индексу ничего нет, то нет и элемента, алгоритм завершён.
4. Иначе по индексу находится элемент с нашим ключом (требуется операция сравнения ключей) – элемент найден.
5. Если по индексу находится элемент с другим ключом или элемент помечен удалённым, увеличиваем индекс на единицу (возвращаясь в начало таблицы при необходимости) и переходим к пункту 3.

6. Следующий индекс вычисляется по формуле $(\text{index} + 1) \bmod M$.

Операция вставки по ключу.

1. При вставке нового элемента вычисляется хеш-функция.

2. По значению хеш-функции определяется место поиска — индекс в хеш-таблице.

3. Если по индексу находится пустой элемент или имеется элемент, помеченный как удалённый, то мы нашли подходящее место, вставляем по индексу элемент

4. Если по индексу уже присутствует элемент с искомым ключом — не трогая ключа меняем данные и выходим.

5. Если по индексу элемент с другим ключом то индекс увеличиваем на единицу и переходим к пункту 3.

6. Следующий индекс вычисляется по формуле $(\text{index} + 1) \bmod M$.

Рисунок 7.9 иллюстрирует, почему требуются свойства равномерности от хеш-функции: малейшая неравномерность при генерации значений приводит к большим кластерам коллизий.

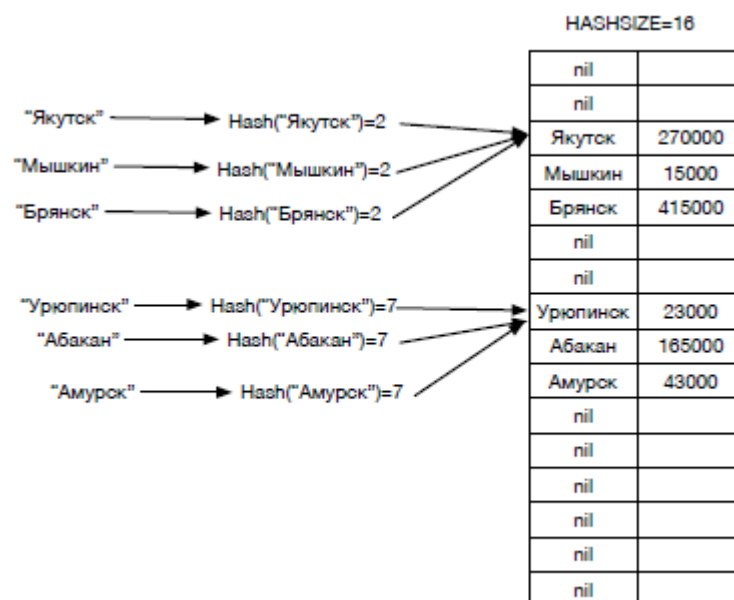


Рис. 7.9.

Операция удаления.

1. При удалении вычисляется хеш-функция от ключа.

2. Определяется место поиска — индекс в хеш-таблице.

3. Если по индексу ничего нет, то нет и элемента.

4. Иначе если по индексу расположен элемент с требуемым ключом, то элемент найден. Помечаем его удалённым и заканчиваем алгоритм.

5. Если по индексу расположен элемент с другим ключом, индекс увеличивается на единицу и переходим к пункту 3.

6. Следующий индекс вычисляется по формуле $(\text{index} + 1) \bmod M$.

Когда fill-factor начинает превосходить 0.7-0.8, отношение количества сравнения ключей к количеству производимых запросов становится слишком велико, поэтому какой бы начальный размер таблицы не был выбран, при регулярных операциях вставки возникает момент, когда таблицу необходимо расширить.

Расширение хеш-таблицы:

- Создаётся другой массив пустых элементов либо указателей на них нужного размера.

- Из оригинального массива в порядке увеличения индексов извлекаются элементы и вставляются в новый массив (таблицу).

- Старый массив удаляется.

Очевидно, что расширение таблицы – дорогая операция, которая должна произвести $O(N)$ операций вставок в новое хранилище для таблицы.

Можно посчитать амортизированную сложность N операций вставки в таблицу, если положить, что каждый раз новое хранилище будет в $k > 1$ раз больше предыдущего. Примем, что начальный размер хеш-таблицы был M , а количество операций сравнения ключей есть $O(1)$ для операций вставки, а операция расширения таблицы будет проводиться при достижении фактора заполнения в 0.5.

Тогда до первого расширения будет проведено $M / 2$ операций вставки, после чего будет создана новая копия хранилища за $k \cdot M$ операций, после чего

будет произведено $M/2$ вставок их старого хранилища. Итого — $M/2 + k \cdot M + M/2$ операций, что для вставленных M элементов даст $O(1)$ на одну операцию.

Итак, рассмотрен простой способ нахождения пустого элемента, увеличивая номер позиции-кандидата каждый раз на единицу, $K = 1$ (вспомните формулу $(\text{index} + 1) \bmod M$). При не очень удачной хеш-функции в таблице быстро образуются кластера из элементов, ключи которых оказались в коллизии. Если обобщить эту формулу до $(\text{index} + K) \bmod M$, то, казалось бы, ничего не изменится. Но K можно сделать функцией от ключа, то есть, $K = H_2(\text{key})$. Оказывается, в этом случае кластеризация уменьшается и коэффициент амортизации уменьшается вместе с ней. Это – рехеширование.

Итак, рассмотрено два основных способа организации хеш-таблиц – с открытой и с закрытой адресациями. Выбор организации для конкретного применения зависит от многих факторов – размера ключа и данных, сложности операции сравнения ключей, архитектуры конкретной вычислительной системы. Практика показывает, что при закрытой адресации ухудшается локальность обращения к кэш-памяти, что часто приводит к потере производительности. Важность хорошей хеш-функции трудно переоценить. Последняя рекомендация – не допускать фактора заполнения, большего, чем 0.5-0.6. При превышении этого значения количество сравнений ключей на одну операцию резко увеличивается.

7.4.4 Хеш-таблицы во внешней памяти

Хеш-таблицы, как оказывается, очень удобны и для работы с данными, для хранения которых не хватает оперативной памяти.

Рассмотрим следующую задачу: имеется $5 \cdot 10^9$ записей, состоящих из уникального ключа размером 1000 байтов и данных размером 10000 байтов. В настоящее время данные располагаются в 5000000 файлах, в каждом из которых по 1000 строк.

Требуется организовать данные так, чтобы обеспечить быстрый поиск по ключу. Ожидается небольшое количество операций добавления и удаления после формирования структур данных этой поисковой системы.

Общий размер превышает 2000GB. Количество операций поиска будет велико, но много меньше общего числа записей. Допустимо хранение результатов преобразования данных на устройстве с произвольным доступом.

Для решения задачи можно использовать В-деревья. Но лучшим выбором будут хеш-таблицы во внешней памяти. Одним проходом по исходным данным эта таблица формируется, после чего файл, содержащий хеш-таблицу может использоваться для многократного поиска, возможно, в другом приложении.

Один из возможных вариантов хранения данных в файле приведён на рисунке 7.10.

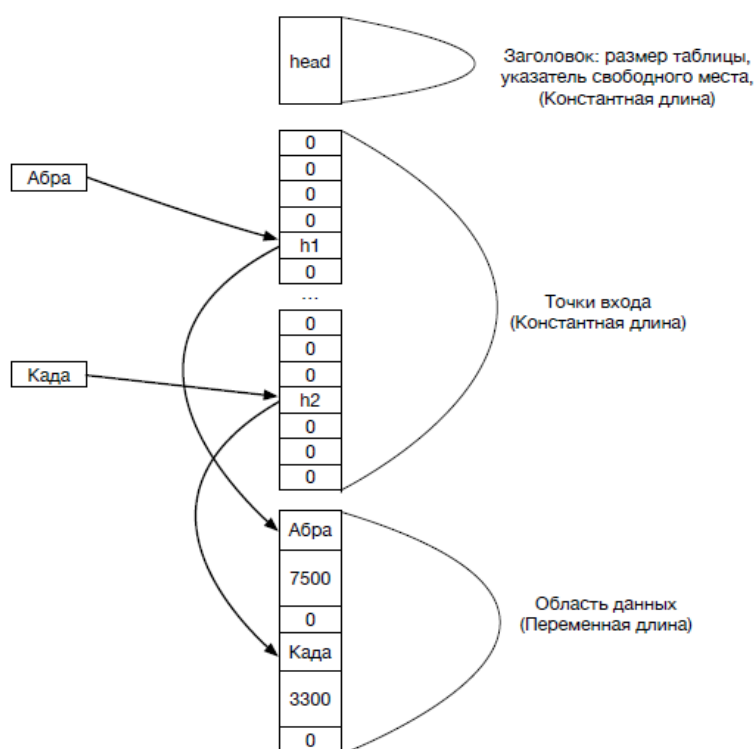


Рис. 7.10.

Весь файл разбит на три зоны.

Первая зона — фиксированного размера, заголовок. В нём можно хранить такую информацию, как размер хеш-таблицы HASHSIZE, начало и размер

второй зоны, количество записей в таблице, смещение в файле, по которому можно записывать новые данные и прочее.

Вторая зона содержит точки входа — смещение в файле, по которому начинается поиск требуемой записи. В этой зоне находится HASHSIZE входов

Третья зона содержит сами записи. Каждая запись содержит три поля — ключ, данные и точку продолжения поиска при коллизии.

Для эффективности ввода/вывода все зоны начинаются с позиций, кратных размеру блока на диске или размеру страницы виртуальной памяти, это число всегда степень двойки и одно из типичных значений составляет 4096 байт. Возможный размер адреса поиска определяется максимальным размером файла с хеш-таблицей. Он может составлять, например, 8 байт для удобства.

Операция поиска в такой хеш-таблице может производиться, например, таким образом:

1. Находится значение хеш-функции от ключа H . Это число — номер записи во второй зоне
2. Считывается запись под номером H из второй зоны как число L .
3. Полагается, что, начиная со смещения L в файле, находится запись R . Если сравнение ключа записи $R.key$ и ключа поиска завершилось успехом, запись найдена и можно вернуть поле данных $R.data$.
4. Если ключи не сравнились, то анализируется поле $R.overflow$. Пустое значение этого поля означает, что такой записи в таблице нет.
5. Если поле $R.overflow$ не пусто, оно содержит новый адрес L , после чего переходим к пункту 3 (рис. 7.11).

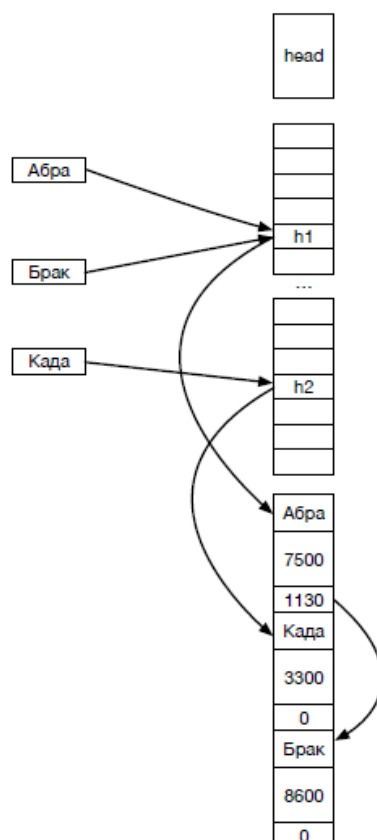


Рис. 7.11.

Алгоритм вставки в хеш-таблицу должен вести учёт свободного места, куда можно разместить очередную запись. Если при вставке записи коллизий не произошло, то всё достаточно просто, требуется изменить запись во второй зоне. При коллизии потребуется пробежать по цепочке коллизий и в последней изменить значение поля `R.overflow` на указатель свободного места, куда и будет вставлена новая запись.



Рассмотрим работу с хеш-таблицей на примере простейшего телефонного справочника. В каждом элементе хеш-таблицы храните структуру, содержащую два строковых поля: номер телефона и фамилию абонента. В качестве хеш-функции выберите функцию для телефонных номеров, в которой суммируются коды цифр с весами, равными порядковому номеру цифры. Реализуйте операции с хеш-таблицей: добавление элемента в таблицу, удаление элемента из таблицы, поиск элемента в таблице. Для разрешения коллизии используйте метод открытой адресации.

Контрольные вопросы

1. В чем состоит основная идея обобщенного поиска?
2. Что такое хеш-функция?
3. Назовите основные свойства хеш-функции.
4. Приведите примеры плохих хеш-функций.
5. Что такое коллизия?
6. Какое множество хеш-функций называется универсальным?
7. Приведите пример полиномиальной хеш-функции.
8. Приведите пример хеш-функции, основанной на идее генерации случайных чисел.
9. В чем заключается суть фильтра Блума?
10. Как реализуется алгоритм Карпа–Рабина?
11. Что из себя представляет хеш-таблица?
12. Какие методы разрешения коллизии вы знаете?
13. Как строится хеш-таблица с прямой адресацией?
14. Как строится хеш-таблица с открытой адресацией?
15. Как организуются хеш-таблицы во внешней памяти?

Задания для самостоятельного выполнения

Задание № 1. Подстроки

Ограничение по времени: 2 секунды

Ограничение по памяти: 64 мегабайта

Входной файл состоит из одной строки I , содержащей малые буквы английского алфавита.

Назовем подстроковой длиной L с началом S множество непрерывно следующих символов строки.

Например, строка

abca**b**

содержит подстроки:

длины 1: a, b, c, a, b

длины 2: ab, bc, ca, ab

длины 3: abc, bca, cab

длины 4: abca, bcab

длины 5: abcab

В строках длины 1 есть два повторяющихся элемента — a и b. Назовем весом подстрок длины L произведение максимального количества повторяющихся подстрок этой длины на длину L.

В нашем случае вес длины 1 есть 2 ($2*1$), длины 2 есть 4 ($2*2$), длины 3 — 3 ($1*3$), длины 4 — 4 и длины 5 — 5.

Требуется найти наибольший из всех весов различных длин.

Задание № 2. Большая книжка

Ограничение по времени: 5 секунды. Ограничение по памяти: 4 мегабайта

Заказчику понравилось решение нашей задачи по созданию записной книжки, и он предложил нам более сложную задачу: создать простую базу данных, которая хранит много записей вида ключ: значение. Для работы с книжкой предусмотрены 4 команды:

ADD KEY VALUE — добавить в базу запись с ключом KEY и значением VALUE. Если такая запись уже есть, вывести ERROR.

DDELETE KEY — удалить из базы данных запись с ключом KEY. Если такой записи нет — вывести ERROR.

UPDATE KEY VALUE — заменить в записи с ключом KEY значение на VALUE. Если такой записи нет — вывести ERROR.

PRINT KEY — вывести ключ записи и значение через пробел. Если такой записи нет — вывести ERROR.

Количество входных строк в файле с данными не превышает 300000, количество первоначальных записей равно половине количества строк (первые $N/2$ команд есть команды ADD).

Длины ключей и данных не превосходят 4096. Ключи и данные содержат только буквы латинского алфавита и цифры и не содержат пробелов.

Особенность задачи: все данные не поместятся в оперативной памяти и поэтому придется использовать внешнюю.

Формат входных данных:

```
10
ADD JW SJXO
ADD RZBR YMW
ADD ADX LVT
ADD LKFLG UWM
PRINT ADX
UPDATE HNTP JQPVG
PRINT QURWB
DELETE MB
```

Формат выходных данных:

```
ADX LVT
ERROR
QURWB MEGW
ERROR
```

Задание №3. Сопоставление по образцу

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

Известно, что при работе с файлами можно указывать метасимволы * и ? для отбора нужной группы файлов, причем знак * соответствует любому множеству, даже пустому, в имени файла, а символ ? Соответствует ровно одному символу в имени.

Первая строка программы содержит имя файла, состоящее только из заглавных букв латинского языка (A-Z), а вторая — образец, содержащий только заглавные буквы латинского алфавита и, возможно, символы * и ?. Строки не превышают по длине 700 символов. Требуется вывести слова YES или NO в зависимости от того, сопоставляется ли имя файла указанному образцу.

Формат входных данных:

```
SOMETEXT
PATTERN
```

Формат выходных данных:

YES или NO

Задание №4. Точные квадраты

Ограничение по времени: 1 секунды. Ограничение по памяти: 256 мегабайта

Можете ли вы по десятичному представлению натурального числа определить, является ли это число полным квадратом? А если в числе много десятичных знаков?

Формат входных данных:

Первая строка содержит $5 \leq N \leq 10^6$ — количество тех чисел, которые нужно проверить. В последующих N строках — десятичные представления натуральных чисел количеством десятичных цифр в представлении не более 100.

Формат выходных данных:

Для каждого из чисел, являющихся полным квадратом, вывести его номер. Нумерация начинается с единицы.

Задание №5. Такси

Ограничение по времени: 2 секунды. Ограничение по памяти: 32 мегабайта

В некотором очень большом городе руководство осознало, что в автономные такси, то есть, такси без водителя — большое благо и решило открыть 10 станций по аренде таких такси. Были получены данные о том, откуда клиенты могут заказывать машины. Было замечено, что если станция находится от клиента на расстоянии, не большем, чем некоторое число R , то клиент будет арендовать машину именно на этой станции, причем, если таких станций несколько — клиент может выбрать любую. Для экономии, станции решено строить только в местах возможного расположения клиентов. Задача заключается в том, чтобы определить места наилучшей постройки, то есть такие, которые могут обслужить наибольшее количество клиентов.

Формат входных данных:

В первой строке входного файла – два числа, количество клиентов и значение параметра R . В каждом из последующих N – два числа, координаты X_i и Y_i i -го клиента (нумерация ведется с нуля).

Формат выходных данных:

В выходном файле должно присутствовать не более 10 строк. Каждая строка должна содержать номер клиента, у которого выгоднее всего строить станцию, и количество обслуживаемых этой станцией клиентов, отличное от нуля. Выводимые строки должны быть упорядочены от наибольшего количества обслуживаемых клиентов к наименьшему. Если две и более станции могут обслужить одинаковое количество клиентов, то выше в списке должна находиться станция с меньшим номером.

ГЛАВА 8. ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

8.1 Принцип оптимальности Беллмана

8.1.1 Задача о количестве маршрутов

Задача. Имеется несколько городов и между некоторыми из них проведены односторонние дороги так, что, выехав из города, вернуться туда невозможно. Требуется найти количество маршрутов из одного пункта в другой (рис. 8.1).

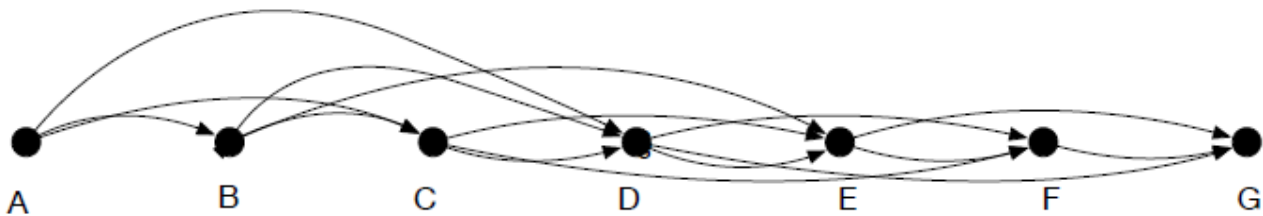


Рис. 8.1.

В этой задаче нужно найти количество маршрутов из пункта A в пункт G. Введём функцию от узла: Пусть $F(i)$ будет равно количеству маршрутов из A до i . Тогда $F(G) = F(F) + F(E) + F(D)$.

Аналогично:

$$F(F) = F(E) + F(D) + F(C),$$

$$F(E) = F(D) + F(C) + F(B),$$

$$F(D) = F(C) + F(B) + F(A),$$

$$F(C) = F(B) + F(A),$$

$$F(B) = F(A),$$

$$F(A) = 1.$$

Данная задача является рекурсивной. Каждая задача разбивается на подзадачи. После решения подзадачи требуется консолидировать результаты. Каждая из подзадач решается аналогично основной, но сама задача несколько проще, чем главная, в данном случае города подзадач находятся немного ближе

к начальному пункту, чем города задач. Имеются подзадачи, не требующие рекурсии, например, подзадача для начального пункта уже имеет решение – имеется ровно один маршрут из города в него же.

Так как задача – рекурсивная, для неё можно построить дерево рекурсии. Однако, даже для такой небольшой задачи это дерево очень велико (рис. 8.2).

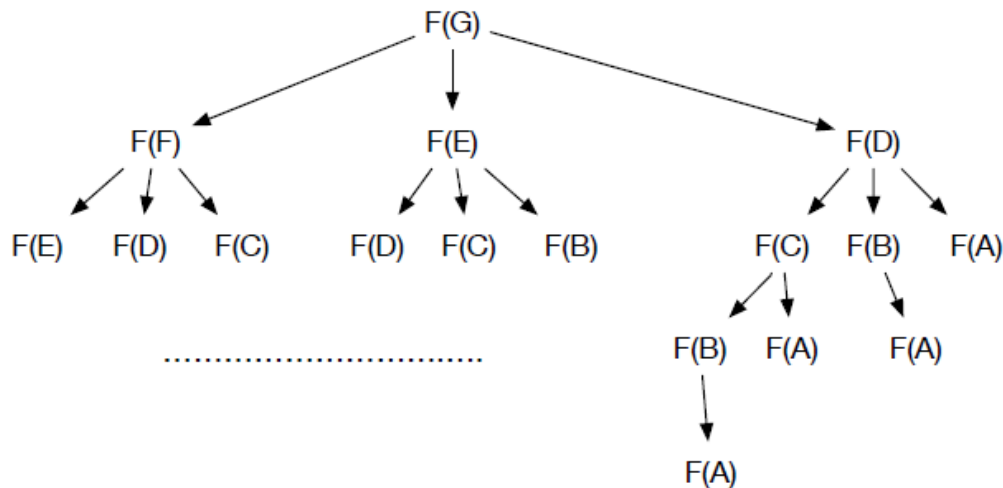


Рис. 8.2.

Однако, стоит обратить внимание на тот факт, что подзадачи в данном дереве частично совпадают. Такой факт говорит о том, что исходная задача является задачей динамического программирования.

Условия появления задачи динамического программирования:

- Задача может быть разбита на произвольное количество подзадач.
- Решение полной задачи зависит только от решения подзадач.
- Каждая из подзадач по какому-либо критерию немного проще основной задачи.
- Часть подзадач и подзадач у подзадач совпадает.

Задачу динамического программирования можно решить двумя способами: рекурсивно с запоминанием результатов или с помощью восходящего решения в правильном порядке.

Возникает еще один вопрос, можно ли решить данную задачу методом «разделяй и властвуй»? Для применения данного метода, задача должна удовлетворять следующим условиям:

- Задача может быть разбита на произвольное количество b подзадач.
- Решение полной задачи зависит только от решения подзадач.
- Каждая из подзадач проще основной задачи не менее, чем в b раз.

Основной способ решения задач методом разделяй и властвуй – рекурсия или эквивалентная ей итерация. Методы динамического программирования и «разделяй и властвуй» очень похожи, но, вместе с тем, отличаются на одно важное и фундаментальное условие – подзадачи метода «разделяй и властвуй» всегда проще основной задачи не менее чем в b раз, тогда как в динамическом программировании такое условие отсутствует, а простота подзадач обозначена как немного проще по какому-либо критерию.

8.1.2 Принцип и уравнение Беллмана

Термин динамическое программирование появился в 1940-х годах и был связан с задачами управления. Сам Беллман сформировал принцип оптимальности так: «Каковы бы ни были первоначальное состояние и решение, последующие решения должны составлять оптимальное поведение относительно уже решённого состояния».

Принцип Беллмана для динамических управляемых систем таков:

- Пусть имеется управляемая система.
- S – её текущее состояние.
- $W_i = f_i(S, x_i)$ – функция выигрыша (стоимости) при использовании управления x на i -м шаге.
- $S' = \phi_i(S, x_i)$ – состояние, в которое переходит система при воздействии x .

Независимо от значения S нужно выбрать управление на этом шаге так, чтобы выигрыш на данном шаге плюс оптимальный выигрыш на всех последующих шагах был максимальным.

$$W_i(S) = \max_{x_i} \{f_i(S, x_i) + W_{i+1}(\varphi_i(S, x_i))\}$$

Ключом к решению задач динамического программирования является составление уравнения оптимальности Беллмана. Рассмотрим его на примере задачи о количестве маршрутов.

Пусть A – матрица смежности для городов,

$$A[i, j] = \begin{cases} 1, & \text{если имеется прямой путь из } i \text{ в } j \\ 0, & \text{если не имеется прямого пути из } i \text{ в } j \end{cases}$$

Тогда:

$$F(x) = \sum_{i=1}^k F(i) \times A[i, x]$$

Для дорожного графа, представленного на рис.8.1 матрица смежности имеет следующий вид:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Построив такую матрицу теперь можно без труда решить поставленную задачу рекурсивно:

```
int func(int x, int[,] a, int k)
{
    int result = 0;
    for (int i = 0; i < k; i++)
    {
        result += f(i, a, k) + a[i, x];
    }
    return result;
}
```

На самом деле, разобранный пример – это только первый этап решения задачи. Рассмотрим еще один пример.

8.1.3 Задача о возрастающей подпоследовательности наибольшей длины

Задача. Имеется последовательность чисел a_1, a_2, \dots, a_n .

Подпоследовательность $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ называется возрастающей, если

$$1 \leq i_1 < i_2 < \dots < i_k \leq n$$

и

$$a_{i_1} < a_{i_2} < \dots < a_{i_k}$$

Требуется найти максимальную длину возрастающей подпоследовательности. Например, для последовательности $a_i = \{10, 4, 13, 7, 3, 6, 17, 33\}$ одна из возрастающих подпоследовательностей есть $\{4, 7, 17, 33\}$.

Нарисуем граф задачи и соединим направленными рёбрами элементы, которые могут быть друг за другом в подпоследовательности (рис. 8.3).

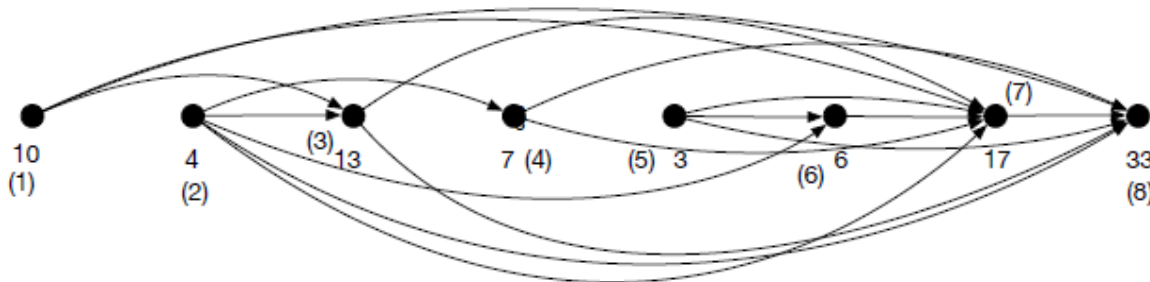


Рис. 8.3.

Задача оказалась похожей на предыдущую, только теперь требуется найти не количество путей, а длину наибольшего пути. В задаче на количество путей консолидация подзадач имела вид:

$$R_i = \sum_{j=1}^{N_{R_{i-1}}} R_{i-1}$$

В задаче о возрастающей подпоследовательности длина наибольшего пути к вершине i (уравнение Беллмана) есть максимум из наибольших путей к предыдущим вершинам плюс единица:

$$L_i = 1 + \max(L_{i-1}, j = 1, N_{L_{i-1}})$$

Задача и подзадачи выглядят так:

$$L8 = 1 + \max(L1, L2, L3, L4, L5, L6, L7),$$

$$L7 = 1 + \max(L1, L2, L3, L4, L5, L6),$$

$$L6 = 1 + \max(L2, L5),$$

...

Рекурсивно эта задача решается следующей программой:

```
int func(int[] a, int N, int k)
{ // k - номер элемента
  int m = 0;
  for (int i = 0; i < k - 1; i++)
  { // для всех слева
    if (a[i] < a[k])
    { // есть путь?
      int p = f(a, N, i); // его длина
      if (p > m) m = p; // m = max(m, p)
    }
  }
  return m+1;
}
```

Для последовательности {1, 4, 2, 5, 3} решение будет таким:

$$f_5 = 1 + \max(f_1, f_3)$$

$$f_4 = 1 + \max(f_1, f_2, f_3)$$

$$f_3 = 1 + \max(f_1)$$

$$f_2 = 1 + \max(f_1)$$

$$f_1 = 1$$

Возвращаемся назад:

$$f_2 = 1 + \max(f_1) = 2$$

$$f_3 = 1 + \max(f_1) = 2$$

$$f_4 = 1 + \max(f_1, f_2, f_3) = 3$$

$$f_5 = 1 + \max(f_1, f_3) = 3$$

$$\text{Решение найдено: } \max(f_1, f_2, f_3, f_4, f_5) = 3$$

Чтобы повторно не решать решённые подзадачи необходимо ввести массив размером N, который хранит значения вычисленных функций. Начальные значения элементов массива равны нулю, что позволит осуществить проверку, была ли данная подзадача решена ранее:

```
int func(int[] a, int N, int k, int[] c)
```

```

{
    if (c[k] != 0) return c[k]; // Уже решали для k
    int m = 0;
    for (int i = 0; i < k - 1; i++)
    { // для всех слева
        if (a[i] < a[k])
        { // есть путь?
            int p = func(a, N, i); // его длина
            if (p > m) m = p; // m = max(m, p)
        }
    }
    return c[k] = m + 1; // заносим в кэш и возвращаем
}

```

Такое запоминание результатов промежуточных подзадач часто называют меморизацией, хотя такой метод по сути есть кеширование значений функции. В дальнейшем будем называть массив, помогающий нам избежать повторных вычислений, кеш-таблицей.



Создайте новый проект. Реализуйте в нем методы, решающие задачу о возрастающей подпоследовательности и о количестве маршрутов. Организуйте ввод данных через консоль или через формы.

8.2 Рекурсивное решение задач динамического программирования

8.2.1 Рекурсия как база динамического программирования

Вспомним задачу о банкомате, рассмотренную во второй главе настоящего учебного пособия.

Задача. В банкомате имеется неограниченное количество банкнот (b_1, b_2, \dots, b_n) заданных номиналов. Нужно выдать требуемую сумму денег наименьшим количеством банкнот.

Как уже выяснено ранее, жадное решение возможно не для всех наборов входных данных. Например, при $b = \{1, 6, 10\}$ и $x = 12$ жадное решение даст ответ 3 ($10 + 1 + 1$), хотя существует более оптимальное решение 2 ($6 + 6$). Сведем данную задачу к задаче о количестве маршрутов (рис. 8.4).

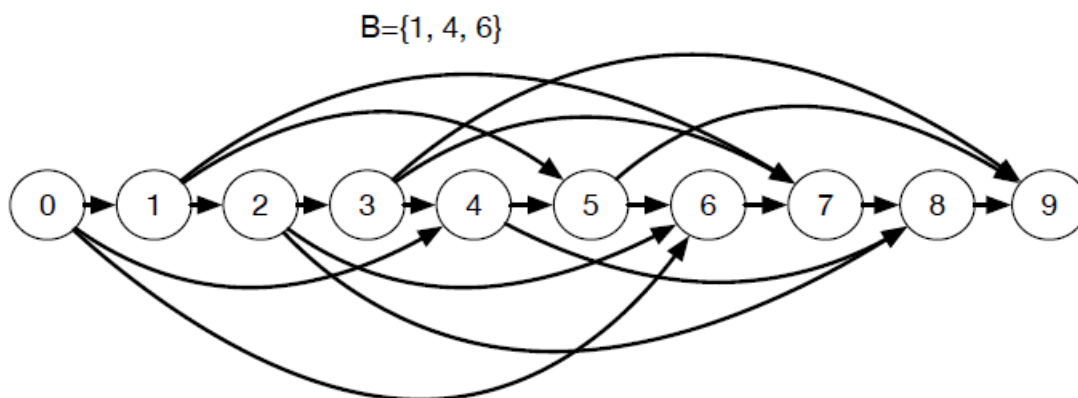


Рис. 8.4.

Для каждой суммы проведём стрелку к узлу с новой разрешённой суммой. Уравнение Беллмана для задачи о банкомате имеет следующий вид:

$$f(x) = \begin{cases} \min_{i=1,n} \{f(x - b_i) + 1, \text{если } x > 0 \\ 0, \text{если } x = 0 \\ \infty, \text{если } x < 0 \end{cases}$$

Составим по уравнению Беллмана код для решения задачи о банкомате:

```
const int MAXVALUE = 999999999;
int func(int x, int[] b, int n)
{
    if (x < 0) return MAXVALUE;
    if (x == 0) return 0;
    int min = MAXVALUE;
    for (int i = 0; i < n; i++)
    {
        int r = func(x - b[i], b, n);
        if (r < min) min = r;
    }
    return min + 1;
}
```

Как и в предыдущем случае, постоянный вызов рекурсивной функции влечет за собой разрастание дерева рекурсии. Несмотря на то, что глубина рекурсии не превосходит $\frac{x}{\min_{i=1,n} b_i}$, количество рекурсивных вызовов экспоненциально растет. Решить данную проблему можно также с помощью введения дополнительного массива, начальные значения элементов которого равны -1 (так как 0 может быть решением конкретной подзадачи):

```
const int MAXVALUE = 999999999;
int func(int x, int[] b, int n, int[] cache)
```

```

{
    if (x < 0) return MAXVALUE;
    if (x == 0) return 0;
    if (cache[x] >= 0) return cache[x];
    int min = MAXVALUE;
    for (int i = 0; i < n; i++)
    {
        int r = func(x - b[i], b, n, cache);
        if (r < min) min = r;
    }
    return cache[x] = min + 1;
}

```



Напишите метод, решающий задачу о банкомате с помощью метода динамического программирования. Сравните данную реализацию с решением этой же задачи методом «разделяй и властвуй». Проанализируйте результаты.

8.2.2 Декомпозиция задачи

Должное разбиение на подзадачи, декомпозиция, – вероятно, главная сложность при решении задачи методом динамического программирования. Для задачи о количестве путей до точки i подзадачей было определение количества путей до точек, находящихся в одном шаге от i . При условии отсутствия замкнутых маршрутов размер подзадачи всегда был несколько меньше размера задачи. Подзадача решалась тем же методом, что и задача. Наличие таких условий натолкнуло нас на мысль, что эту задачу можно решить методом динамического программирования

Давайте сформулируем общий план определения, подходит ли для решения задачи метод динамического программирования. Задача напрашивается на решение методом динамического программирования если:

1. можно выделить множество подзадач;
2. имеется порядок на подзадачах, то есть можно показать, что задача с одними аргументами заведомо должна быть решена раньше задачи с другими аргументами; можно сказать, что один набор аргументов меньше другого набора;

3. имеется рекуррентное соотношение решения задачи через решения подзадач;

4. рекуррентное соотношение есть рекурсивная функция с целочисленными аргументами, или с аргументами, сводящимися к целочисленному.

При динамическом программировании:

1. принципиально исключаются повторные вычисления в любой рекурсивной функции если есть возможность запоминать значения функции для аргументов, меньших текущего;

2. снижается время выполнения рекурсивной функции до времени, порядок которого равен сумме времён выполнения всех функций с аргументом, меньших текущего, если затраты на рекурсивный вызов постоянны.

Правильная декомпозиция задачи — ключ к её решению. Вернёмся к задаче о рюкзаке, но немного упростим её таким образом, чтобы её можно было решать методом динамического программирования.

Задача. Имеется N предметов, каждый из которых имеет вес $H_i \in \mathbb{N}$ и стоимость $C_i \in \mathbb{N}$. Найти комбинацию предметов, имеющую наибольшую суммарную стоимость, суммарный вес которых не превышает V .

Решение задачи. Декомпозиция должна состоять в уменьшении сложности задачи, то есть, уменьшении какого-либо параметра. Что есть подзадача меньшего размера? Наполнение рюкзака меньшего размера? Наполнение рюкзака меньшим количеством предметов?

При любой выбранной декомпозиции требуется ответ на вопросы:

1. Какие ресурсы потребуются для запоминания результатов подзадач?
2. Если имеется решение подзадач, можно ли на основе этого получить решение задачи?

Рассмотрим подзадачу «рюкзак меньшего размера». Размер памяти для результатов есть размер рюкзака. Если мы знаем результаты для всех меньших рюкзаков V_k , то поможет ли это решить задачу?

Для подзадачи «рюкзак с меньшим количеством предметов» размер памяти для результатов есть количество предметов. Если мы знаем результаты для всех подзадач с меньшим количеством предметов, то поможет ли это решить задачу?

Задача «рюкзак меньшего размера» напоминает нам задачу о банкомате. Но, увы, это всё же различные задачи. Важно, что в задаче про банкомат имеется неограниченный запасом купюр каждого номинала, поэтому выбор каждой купюры не влияет на множество всех возможных ходов. А ведь если количество предметов ограничено, то часть входных данных задачи — множество предметов, которые можно взять. Поэтому успех при решении задачи о банкомате не помогает при решении этой задачи — после выбора одного из предметов для укладывания в рюкзак множество для выбора изменяется.

Но может быть, это не так страшно? Пусть аргументами подзадачи будут оставшееся множество предметов S и оставшееся место в рюкзаке L . H_i — веса предметов, C_i — их стоимости.

Тогда уравнение Беллмана принимает следующий вид:

$$F(S, L) = \begin{cases} \max_{e \in S} (F(S - e, L - H(e)) + C(e)), & \text{если } L > 0 \\ 0, & \text{если } L = 0 \\ -\infty, & \text{если } L < 0 \end{cases}$$

Чтобы задача решалась методом динамического программирования, размер пространства аргументов должен быть невелик. В данном случае он составляет

$$D = O(2^N \cdot L_0) = O(2^N)$$

что слишком много.

Попробуем произвести декомпозицию по количеству предметов. Мы берёмся за решение задачи с $K + 1$ предметами, зная решения всех задач с K предметами. Аргументами подзадачи являются количество предметов K и оставшееся место в рюкзаке L . Чтобы составить уравнение Беллмана, мы

должны сделать индуктивный переход от множества из K предметов к множеству из $K + 1$ предмета. Если у нас есть решение задачи для K предметов, то $(K + 1)$ -й предмет мы можем либо взять, либо его не брать:

$$F(K, L) = \begin{cases} \max(f(F - H_K, K - 1) + C_K, F(L, K - 1)), & \text{если } L > 0 \\ 0, & \text{если } L = 0 \\ -\infty, & \text{если } L < 0 \end{cases}$$

Размер пространства аргументов этой задачи оказывается существенно меньше, чем при другом способе декомпозиции:

$$D = O(N \cdot L_0)$$

Время решения задачи пропорционально пространству решений. Оказывается, и задачу о рюкзаке при ряде условий можно решить за неполиномиальное время.

8.2.3 Восстановление решения в задаче о банкомате

Решая задачу о банкомате, мы получили то, что хотели, а именно, минимальное количество банкнот. Однако, если бы это была задача для реального банкомата, то решение осталось бы неполным, ведь не была получена информация о том, какие именно банкноты требуется выдать.

Находя решение подзадачи, мы всегда имеем возможность запомнить не только ответ, но и оптимальный шаг. Нетрудно, например, просто запоминать историю получения решения.

Давайте сохранять цепочку вызовов рекурсивной функции, имея список банкнот для каждого промежуточного решения. Так как промежуточных решений может быть N и каждое из решений имеет различную длину, то для их хранения потребуется N векторов:

```
const int MAXVALUE = 999999999;
int f(int x, int[] b, int n, int[] cache, ref
List<List<int>> solution)
{
    if (x < 0) return MAXVALUE;
    if (x == 0) return 0;
    if (cache[x] >= 0) return cache[x];
```

```

int min = MAXVALUE, best = -1;
for (int i = 0; i < n; i++)
{
    int r = f(x - b[i], b, n, cache, ref solution);
    if (r < min)
    {
        min = r; best = b[i];
    }
}
solution[x] = solution[x - best];
solution[x].Add(best);
return cache[x] = min + 1;
}

```

Данный код действительно решает задачу, но является не самым оптимальным ее решением. Рассмотрим другой вариант решения.

Доведя решение задачи до ответа и имея кэш-таблицу, можно восстановить решение без знания истории. Предположим, что мы знаем, что точный ответ при заданных начальных значениях есть 7. Тогда возникает вопрос: какой предыдущий ход мы сделали, чтобы попасть в заключительную позицию? Введём термин *ранг* для обозначения наименьшего числа ходов, требуемого для достижения текущей позиции из начальной. Тогда каждый ход решения всегда переходит в позицию с рангом, большим строго на единицу и это позволяет нам разработать следующий алгоритм:

1. Решение основной задачи дало в ответе k , это — ранг заключительной позиции.
2. Делаем позицию текущей.
3. Если текущая позиция имеет ранг 0, то это — начальная позиция и алгоритм завершён.
4. Рассматриваем все позиции, ведущие в текущую и выбираем из них произвольную с рангом $k - 1$.
5. Запоминаем ход, который привёл из позиции ранга $k - 1$ в ранг k .
6. Понижаем ранг — $k \rightarrow k - 1$ и переходим к 2.

```

List<int> buildSolution(int x, int[] b, int n, int[]
cache)
{
    List<int> result = new List<int>();

```

```

for (int k = cache[x]; k >= 0; k--)
{
    for (int i = 0; i < n; i++)
    {
        int r = x - b[i];
        if (r >= 0 && cache[r] == k - 1)
        {
            x = r; result.Add(b[i]);
            break;
        }
    }
}
return result;
}

```

Функция `buildSolution` возвращает вектор, содержащий номиналы купюр, требуемых для выдачи необходимой суммы. Хотя вектор, содержащий номиналы купюр, формируется в обратном порядке, именно для этой задачи это оказывается несущественным.

Третья возможность восстановить решение — добавить ещё один кеш, `bestnote`, сохраняющий номинал лучшей банкноты при лучшем решении.

```

const int MAXVALUE = 999999999;
int f(int x, int[] b, int n, int[] cache, int[] bestnote)
{
    if (x < 0) return MAXVALUE;
    if (x == 0) return 0;
    if (cache[x] >= 0) return cache[x];
    int min = MAXVALUE, best = -1;
    for (int i = 0; i < n; i++)
    {
        int r = f(x - b[i], b, n, cache, bestnote);
        if (r < min)
        {
            min = r;
            bestnote[x] = r;
        }
    }
    return cache[x] = min + 1;
}

```

Теперь для того, чтобы получить нужные для размена банкноты для суммы в `x`, достаточно пробежаться по кешу банкнот.



Продолжайте работу с задачей о банкомате. Реализуйте различные вариации восстановления решения в задаче о банкомате. Сравните их.

8.3 Восходящее решение задач динамического программирования

8.3.1 Восходящее решение

Вернёмся к задаче о возрастающей подпоследовательности. При нисходящем решении нам требуется находить максимум из всех значений функций от $F(1)$ до $F(N)$, для чего рекурсивный вызов должен опуститься от $F(N)$ до $F(1)$. Для больших N уровень рекурсивных вызовов может превысить разумные рамки (размер стека в программах ограничен). Можно или нет обойтись без рекурсии при решении задачи динамического программирования, определяется структурой хеш-таблицы.

При восходящем решении мы пробуем решать подзадачи до того, как они будут поставлены. Решая задачи в возрастающем порядке, мы достигаем следующих целей:

1. Гарантируется, что все задачи, решаемые позже, будут зависеть от ранее решённых.
2. Не потребуются рекурсивные вызовы.

Обязательное условие: монотонность аргументов при решении подзадач, если $f(k)$ — подзадача для $f(n)$, то $M(k) < M(n)$, где $M(x)$ есть некая метрика сложности решения.

Для задачи о максимальной возрастающей подпоследовательности порядок решения будет таким:

$$F(1) = 1$$

$$F(2) = 1$$

$$F(3) = \max(F(1), F(2)) + 1 = 2$$

...

$$F(8) = \max(F(1), F(2), F(3), F(4), F(5), F(6), F(7)) + 1$$

После получения значений $F(i)$ не требуется вызывать функцию, можно использовать уже вычисленное значение.

Восходящее решение — отнюдь не панацея. Например, пусть решение задачи определяется таким образом:

$$F(n) = \max(F((n + 1)/2), F(n/3)) + 1$$

$$F(0) = F(1) = 1$$

Это описывает какую-то последовательность. Нисходящий способ для $F(8)$ требует следующих вызовов:

$$F(8) = \max(F(4), F(2)) + 1$$

$$F(4) = \max(F(2), F(1)) + 1$$

$$F(2) = \max(F(1), F(0)) + 1 = 2$$

$$F(4) = 3$$

$$F(8) = 4$$

Восходящее решение здесь существенно более трудоёмко:

$$F(0) = F(1) = 1$$

$$F(2) = \max(F(1), F(0)) + 1 = 2$$

$$F(3) = \max(F(2), F(1)) + 1 = 3$$

$$F(4) = \max(F(2), F(1)) + 1 = 3$$

$$F(5) = \max(F(3), F(1)) + 1 = 3$$

$$F(6) = \max(F(3), F(2)) + 1 = 3$$

$$F(7) = \max(F(4), F(2)) + 1 = 4$$

$$F(8) = \max(F(4), F(3)) + 1 = 4$$

Значения $F(3), F(5), F(6), F(7)$ можно было бы и не вычислять, так как они не используются в дальнейшем. Но ведь мы этого заранее не знаем.

Поставленная задача более подходила под рекурсивный алгоритм с запоминанием промежуточных результатов (memoizing). Нисходящее решение $F(N)$ имеет сложность $O(\log N)$, а восходящее — сложность $O(N)$.

С другой стороны, рассмотрим задачу, которая может быть решена следующим уравнением Беллмана:

$$F(x) = \begin{cases} 1, & \text{если } x \in \{0, 1, 2\} \\ 0, & \text{если } x < 0 \\ F(x - 1) + F(x - 3), & \text{если } x > 2 \end{cases}$$

При вычислении $F(100000)$ в нисходящем порядке размер стека будет равен 100000, что, возможно, приведёт к аварийному завершению программы. Здесь мы различаем понятия «алгоритм», который корректен, и «программа», которая исполняется «исполнителем». Для хранения аргументов и локальных переменных каждого рекурсивного вызова потребуется всё возрастающее количество памяти.

Последовательность будет выглядеть так:

{1, 1, 1, 2, 3, 4, 6, 9, 13, 19, 28, 41, 60, 88, 129}

$$F(998) \approx 2.89273 \cdot 10^{165}$$

$$F(999) \approx 4.23951 \cdot 10^{165}$$

Для хранения 999-го элемента $F(999)$ потребуется по меньшей мере $\log_2 4.23951 \cdot 10^{165} \approx 559$ бит ≈ 69 байтов, не считая управляющей информации. Для восходящего решения задачи достаточно небольшого числа локальных переменных.

Восходящее решение по корректности эквивалентно нисходящему, но при его построении необходимо обеспечить вычисление в соответствующем порядке. При простом целочисленном аргументе вычисления можно проводить в порядке увеличения аргумента. Восходящее решение требует меньшего размера стека, но может решать задачи, ответ на которых не понадобится при решении главной задачи.



Вернитесь к задаче о возрастающей подпоследовательности и попытайтесь реализовать ее с помощью восходящего решения. Сравните полученную реализацию с выполненной ранее. Сравните производительность двух методов.

8.3.2 Задача о покрытии

Рассмотрим пример задачи, в которой требуется более двух целочисленных аргументов.

Задача. Имеется прямоугольник размером 5×6 . Сколькими способами его можно замостить фигурами 1×2 и 1×3 ? Симметрии и повороты различаются (рис. 8.5).

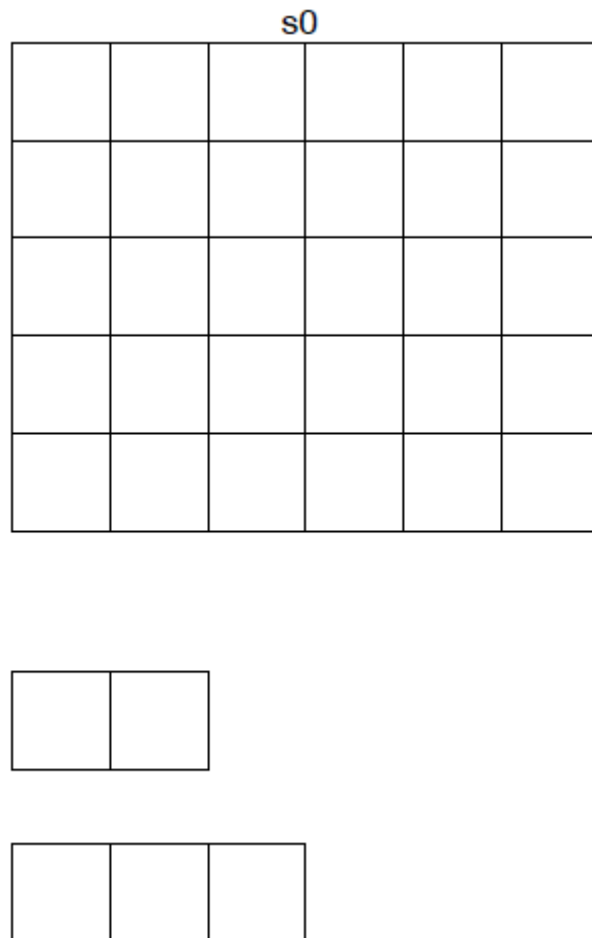


Рис. 8.5.

Решение задачи. Эту задачу можно решить методами комбинаторики, но в данном параграфе попробуем решить ее методом динамического программирования. Обозначим через $f(s)$ количество разбиений фигуры s на требуемые фрагменты. Тогда $f(s_0) = f(s_1) + f(s_2) + f(s_3) + f(s_4)$, где s_i — подфигуры, получающиеся вычитанием одного из фрагментов, содержащих крайний левый верхний квадрат — так как все разбиения фигуры должны содержать этот квадрат, изменим порядок таким образом, чтобы включить его первым ходом. Возможные подзадачи представлены на рис. 8.6.

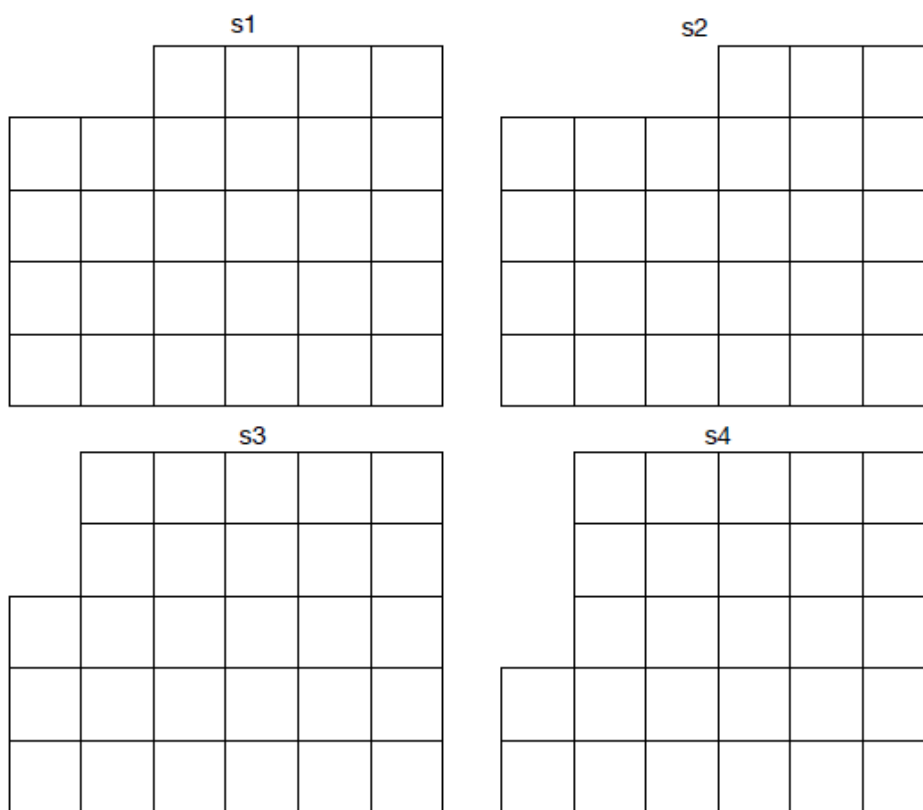


Рис. 8.6.

$f(s1) = f(s11) + f(s12) + f(s13) + f(s14)$. Данное уравнение продемонстрировано на рис. 8.7.

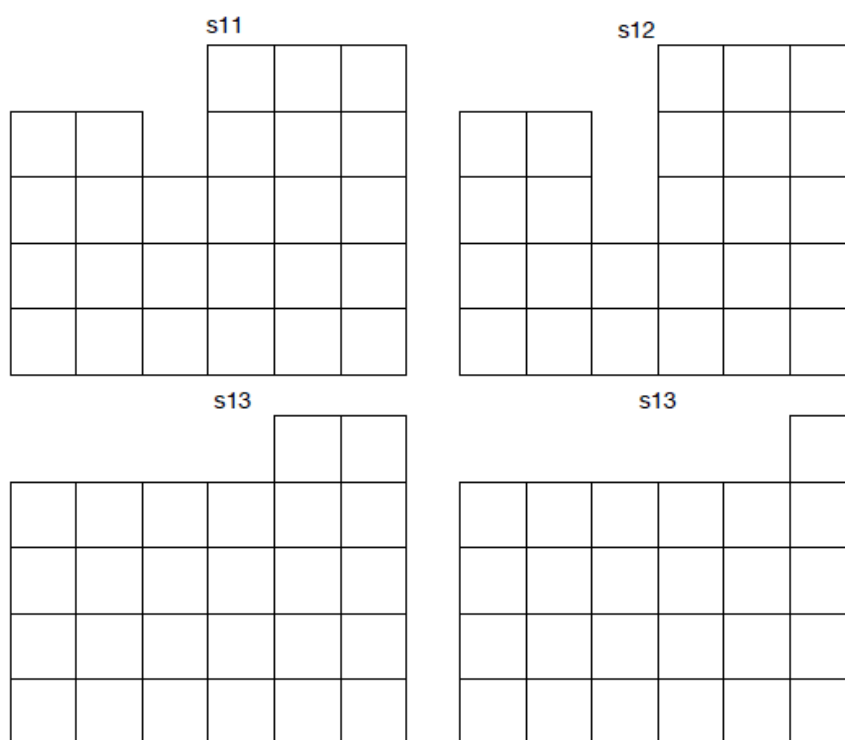


Рис. 8.7.

Является ли данная задача задачей динамического программирования?

При предложенном алгоритме решения одна подзадача может возникнуть при различных путях (рис. 8.8).

s'					
1	1	2	2	3	3
4	4	5	5	6	6

s''					
1	2	3	4	5	6
1	2	3	4	5	6

s'''					
1	1	1	2	2	2
3	3	3	4	4	4

Рис. 8.8.

Решение таких подзадач не зависит от истории их получения. Если имеется отображение аргументов подзадачи (позиции) на результат, то задача может быть решена методом динамического программирования. Аргументом в функции, определяющей уравнение Беллмана, в данном случае является абстрактный тип «фигура», над которым определены операция «вырезать». Что есть $f(s)$, если s не является числом? s — это объект, который мы должны использовать в виде ключа key в отображении. $value$ в отображении есть значение, хранимое по этому ключу.

Теперь задача конкретизируется — нужно создать взаимно однозначное соответствие объекта s какому-то набору битов. Объект «фигура» в данной задаче хорошо подходит под такое преобразование. Можно пронумеровать все 30 квадратиков и каждому из номеров присвоить 1, если он присутствует и 0, если отсутствует. Результатом является строка в 30 битов. Это хорошо

подходит для ключа, но размер множества возможных ключей есть 2^{30} , что слишком много для восходящего метода.

Воспользуемся изученной абстракцией «отображение». Каждая из позиций является ключом в отображении, задача решается нисходящим динамическим программированием с меморизацией. При решении задачи из таблицы решений по ключу, соответствующему текущей позиции, извлекается значение. Если такого ключа нет, то производится полное решение и в отображение добавляется пара (ключ/полученное значение). Если ключ имеется, то результатом подзадачи будет значение по ключу. Выбор способа реализации отображения остаётся за нами. Мы можем использовать и деревья поиска, и хеш-таблицы.

8.3.3 Этапы решения задачи методом динамического программирования

Сформулируем общий протокол действий при подозрении на то, что задача может быть решена методом динамического программирования.

1. Определяется необходимость именно в динамическом программировании. При быстром уменьшении подзадач задача решается методом «разделяй и властвуй».

2. Определяется максимальный уровень рекурсии в главной задаче. Например, в задаче на покрытие прямоугольника максимальный уровень равен 15.

3. Для нетривиальных задач всегда вначале разрабатывается рекурсивный вариант решения задачи.

4. Разрабатывается метод отображения аргументов задачи в результат.

5. Если выбирается нисходящий вариант, то реализуется процесс меморизации.

6. Если максимальный уровень слишком большой, то нисходящий метод неприменим, но по результатам исследования реализуется восходящий метод.

8.4 Многомерные задачи динамического программирования

8.4.1 Задача о преобразовании слов

Если мы сделали опisku в слове, исправить её можно несколькими способами – например, заменив неверную букву на верную, удалив лишнюю букву, или добавив новую букву. Чем больше таких операций мы делаем, тем слова (с нашей точки зрения — строки) менее похожи друг на друга. Количество операций для превращения одного слова в другое называется расстоянием редактирования или расстоянием. Это расстояние – мера различия между двумя строками. Именно этой мерой измеряют сходство или различие двух генных последовательностей.

Задача. Определить минимальное количество операций для преобразования одного слова в другое, при следующих допустимых операциях:

- замена одной буквы на другую;
- вставка одной буквы;
- замена одной буквы.

Например, сколько нужно операций, чтобы превратить слово СЛОН в слово ОГОНЬ? Например, это можно сделать следующей последовательностью:

СЛОН → СГОН → СГОНЬ → ОГОНЬ

Решение задачи. Данная задача решается с помощью метода динамического программирования, значит необходимо составить уравнение Беллмана. В данном случае пока неясно, что является рекурсивной функцией, решающей данную задачу, а точнее, что есть аргументы функции?

Давайте зафиксируем входные строки – исходную строку и строку назначения как s и d . В процессе изменения строка s должна превратиться в d . Если данная задача является задачей динамического программирования, то как найти более простые подзадачи и что является мерой простоты? Похоже на то, что если бы мы знали решения подзадач для более коротких строк, то из них можно было бы вывести и решение полной задачи. Например, в приведённом

выше примере, если было бы известно решение задач, как перевести СЛОН в ОГОН, то, добавив одну букву, мы бы получили решение и основной задачи. Пусть i и j — номера последних символов строк-префиксов s и d соответственно. Если оперировать только последними символами строк, то мы получаем три варианта:

- заменить последний символ строки s на последний символ строки d ;
- добавить символ к концу строки s ;
- удалить последний символ строки s .

Последовательность переходов, которую мы предложили ранее, не обладает нужными свойствами (последовательным увеличением сложности задачи). А есть ли нужная нам последовательность, в которой каждая подзадача решается на подстроке всё более возрастающей длины? Да, имеется.

1. В префиксах строк длины 1 C и O меняем букву C на O . СЛОН \rightarrow ОЛОН.
2. В префиксах длины 2 меняем L на G . ОЛОН \rightarrow ОГОН.
3. В префиксах длины 4 добавляем букву B . ОГОН \rightarrow ОГОНЬ.

Теперь осталось это всё формализовать.

Введём двумерную матрицу D , элемент $D[i, j]$, в которой есть минимальное из всех количеств значений различий между всеми s длиной от 1 до i и префиксом строки d длиной j :

$$D_{i,j} = \min \left(\begin{array}{l} D_{i-1,j-1}, \text{ если } s_i = t_j \text{ или } D_{i-1,j-1} + 1 \text{ если } s_i \neq t_j \\ D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \end{array} \right)$$

Если последние символы префиксов совпадают, тогда нам ничего не надо делать, иначе требуется замена, что даст нам штраф в единицу.

Вторая строка показывает, что нам необходимо вставить символ для соответствия со строкой d .

Третье число — случай удаления лишнего символа.

Чисто рекурсивный вариант решения этой задачи, очевидно, будет слишком медленным. К счастью для нас, плотность значений аргументов и их дискретность позволяют использовать динамическое программирование, сохраняя результаты решённых задач. Более того, эту задачу можно решить без рекурсии, просто заполняя таблицу по строкам.

В качестве примера рассмотрим задачу преобразования строки ARROGANT в строку SURROGATE. Для удобства добавим один лишний левый столбец и одну лишнюю верхнюю строку к таблице результатов (рис. 8.9). Заполним их последовательно возрастающими числами. Эти числа означают, что пустая строка может превратиться в образец за число операций, равное длине образца.

		S	U	R	R	O	G	A	T	E
	0	1	2	3	4	5	6	7	8	9
A	1									
R	2									
R	3									
O	4									
G	5									
A	6									
N	7									
T	8									

Рис. 8.9.

Заполнение таблицы происходит по строкам. Значение в первой свободной ячейке зависит от трех элементов таблицы (рис. 8.10).

		S	U	R	R	O	G	A	T	E
	0	1	2	3	4	5	6	7	8	9
A	1	1								
R	2									
R	3									
O	4									
G	5									
A	6									
N	7									
T	8									

Рис. 8.10.

Так как буквы (S столбца и A строки) не совпадают, то значение, полученное по диагональной стрелке, увеличивается на 1. По горизонтальной и вертикальной стрелке в ячейку приходят увеличенные на 1 числа из тех ячеек. А вот при совпадении букв штраф за замену отсутствует и в клеточку (рис. 8.11) записывается число 6, значение, полученное по диагональной стрелке.

		S	U	R	R	O	G	A	T	E
	0	1	2	3	4	5	6	7	8	9
A	1	1	2	3	4	5	6	6		
R	2									
R	3									
O	4									
G	5									
A	6									
N	7									
T	8									

Рис. 8.11.

Итого решением задачи является правый нижний элемент таблицы (рис. 8.12).

		S	U	R	R	O	G	A	T	E
	0	1	2	3	4	5	6	7	8	9
A	1	1	2	3	4	5	6	6	7	8
R	2	2	2	2	3	4	5	6	7	8
R	3	3	3	2	2	3	4	5	6	7
O	4	4	4	3	3	2	3	4	5	6
G	5	5	5	4	4	3	2	2	3	5
A	6	6	6	5	5	4	3	2	3	4
N	7	7	7	6	6	5	4	3	3	4
T	8	8	8	7	7	6	5	4	3	4

Рис. 8.12.

Нетрудно убедиться, что сложность алгоритма есть $|s| \times |d|$ по времени. Нетрудно убедиться, что хранения всей таблицы не требуется, достаточно двух строк. Для больших строк, длиной в миллионы и десятки миллионов символов (сравнение геномных последовательностей), алгоритм становится трудноразрешимым и его модификации — одна из задач вычислительной биологии.



Напишите реализацию задачи о преобразовании слов. Проанализируйте результаты выполнения программы для разных значений (в том числе, рассматривая крайние случаи — когда в словах нет ни одной одинаковой буквы и когда слова идентичны).

8.4.2 Задача о счастливых билетах

Задача. Билет состоит N цифр от 0 до 9 и является счастливым, если сумма первой половины его цифр равна сумме второй половины. N — чётное число. Найти количество счастливых билетов.

Решение задачи. Сведём задачу к другой. Пусть, например, $N = 6$ (рис. 8.13).

3	3	5	6	4	1
3	3	5	3	5	8

Рис. 8.13.

Заменим во второй половине числа все цифры на их дополнение до девяти. Количество таких чисел в точности равно количеству счастливых, так как отображение *биективное*. Исходный инвариант: $x_1 + x_2 + x_3 = x_4 + x_5 + x_6$. Инвариант после отображения: $x_1 + x_2 + x_3 + (9 - x_1) + (9 - x_2) + (9 - x_3) = 3 \cdot 9$. Требуется найти количество N -значных чисел, сумма которых равна $9 \cdot N / 2$

Привычное решение задачи наталкивается на проблему: нам нужно найти не просто количество любых чисел от 0 до 9, сумма которых 27, нужно, чтобы таких чисел было именно 6. Количество таких чисел есть сумма количеств чисел:

- первая цифра которых 0 и сумма пяти остальных равна 27;
- первая цифра которых равна 1 и сумма пяти остальных равна 26;
- ...
- первая цифра которых равна 9 и сумма остальных пяти равна 18

Вот и необходимая для применения динамического программирования декомпозиция. Нам удалось разбить задачу подзадачи меньшего ранга. Обозначим за $f(n, \text{left})$ количество чисел, имеющих n знаков, сумма которых left . Тогда

$$f(6, 27) = f(5, 27) + f(5, 26) + \dots + f(5, 19) + f(5, 18).$$

Доопределим функцию $f(n, \text{left})$ таким образом, что при $n > 0$ и $\text{left} < 0$ она возвращала 0 и $f(1, \text{left}) = 1$, если $0 \leq \text{left} \leq 9$ и $f(1, \text{left}) = 0$ в противном случае.

Тогда $f(n, \text{left}) = \sum_{i=0}^9 f(n-1, \text{left} - i)$. Мы свели задачу к задаче динамического программирования, но в двухмерном варианте. Если задача двухмерная, то и таблица решений — двухмерная. Первый размер определяется размерностью задачи n . Второй размер определяется максимальным значением

$\text{left} = 9 \cdot n / 2$. Нерешённые подзадачи в таблице помечаются числом -1, так как ни одна из подзадач не может вернуть отрицательное число.

Значения в таблице решений занимают последовательные ячейки, она не разрежена, следовательно, задача допускает восходящее решение. Таблица заполняется, начиная от значения $n = 1$ и всех возможных left от 0 до 9. Максимальный уровень рекурсии равен n , это немного, поэтому вполне возможно решить задачу и нисходящим способом.



Напишите реализацию задачи о счастливых билетах. Проанализируйте результаты выполнения программы для разных значений N .

Контрольные вопросы

1. Опишите задачу о количестве маршрутов.
2. Сформулируйте условия появления задачи динамического программирования.
3. Чем отличаются методы динамического программирования и «разделяй и властвуй»?
4. Назовите два способа решения задачи методом динамического программирования.
5. В чем заключается принцип Беллмана и как выглядит уравнение Беллмана?
6. Составьте уравнение Беллмана для задачи о количестве маршрутов. Поясните используемые обозначения.
7. Сформулируйте задачу о возрастающей подпоследовательности наибольшей длины.
8. Что такое меморизация?
9. Как решить задачу о банкомате методом динамического программирования?
10. Что такое восстановление решения в задаче динамического программирования?

11. Что означает восходящее решение задачи динамического программирования?
12. Как можно применить абстракцию «отображение» в динамическом программировании?
13. Сформулируйте этапы решения задачи методом динамического программирования.
14. Как решить задачу о преобразовании слов методом динамического программирования?
15. Сформулируйте алгоритм решения задачи о счастливых билетах методом динамического программирования.

Задания для самостоятельного выполнения

Задание №1. Игра в фишки

Ограничение по времени: 1 секунда. Ограничение по памяти: 16 мегабайта

На столе лежит куча из $1 \leq N \leq 10^6$ фишек. Игроки First и Second ходят строго по очереди, первый ход за игроком First. Каждый ход игрок может взять из кучи любое количество фишек, не превосходящее целой части квадратного корня из оставшегося на столе количества фишек. Например, при 28 фишках на столе он может взять от одной до пяти фишек. Игра заканчивается, когда на столе не остается ни одной фишки и победителем объявляется тот, кто совершил последний ход.

Требуется вывести имя игрока, который побеждает при обоюдной лучшей игре.

Формат входных данных:

N

Формат выходных данных:

First или Second

Задание № 2. Путешествие продавца

Ограничение по времени: 1 секунда. Ограничение по памяти: 64 мегабайта

Как обычно, кому-то надо продать что-то во многих городах. Имеются города, представленные как M множеств (столбцов) по N городов (строк) в каждом.

Продавец должен посетить ровно по одному городу из каждого множества, затратив на это как можно меньшую сумму денег. Он должен посетить сначала город из первого множества, затем из второго и так далее, строго по порядку. Он может выбирать начало своего путешествия. Число, которое находится в i -й строке и j -м столбце означает стоимость перемещения из предыдущего места в этот город. Однако, имеется ограничение на перемещения: он может перемещаться из города в i -й строке только в города следующего столбца, находящиеся в одной из строк $i-1, i, i+1$, если такие строки существуют.

Иногда, чтобы заставить посетить продавца какой-то город, ему доплачивают, то есть, стоимость перемещения может быть отрицательной.

Требуется определить наименьшую стоимость маршрута и сам маршрут.

Формат входных данных:

$N\ M$

$C_{11}\ C_{12}\ \dots\ C_{1M}$

$C_{21}\ C_{22}\ \dots\ C_{2M}$

$\dots\ \dots\ \dots\ \dots$

$C_{N1}\ C_{N2}\ \dots\ C_{NM}$

$3 \leq N \leq 150$

$3 \leq M \leq 1000$

$-1000 \leq C_{ij} \leq 1000$

Формат выходных данных:

Первая строка — список через пробел номеров строк (начиная с 1) из M посещенных городов.

Вторая строка — общая стоимость поездки.

Если имеется несколько маршрутов с одной стоимостью, требуется вывести маршрут, наименьший в лексикографическом порядке.

Начинать и заканчивать маршрут можно в любой строке.

Задание №3. Счастливые билеты

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

Билет состоит из четного числа N цифр в M -ричной системе счисления. Счастливым билетом называется билет, сумма первой половины цифр которого равна сумме второй половины цифр.

Найти количество счастливых билетов. Учтите: их число может быть велико.

Формат входных данных:

N M

$2 \leq N \leq 150$

$N \bmod 2 = 0$

$2 \leq M \leq 26$

Формат выходных данных:

Количество счастливых билетов

Задание №4. Наибольшая общая подстрока

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

Во входном файле находятся две строки, длиной до 30000 символов, состоящих из цифр и прописных и строчных букв латинского алфавита, каждая в отдельной строке файла.

Необходимо найти общую подстроку наибольшей длины. Если таких подстрок несколько, то следует вывести ту из них, которая лексикографически меньше.

Задание №5. Самая тяжелая подтаблица

Ограничение по времени: 3 секунды. Ограничение по памяти: 256 мегабайта

В каждой ячейке прямоугольной таблицы размером $N \times M$ состоит из чисел от -10^9 до 10^9 .

Назовем подтаблицей любую часть таблицы, включая целую, образующую прямоугольник, а ее весом — сумму всех ее чисел.

Найти вес самой тяжелой из всех возможных подтаблиц, которые можно построить на основе оригинальной.

Формат входных данных:

$N \ M$

$C_{11} \ C_{12} \ \dots \ C_{1M}$

$C_{21} \ C_{22} \ \dots \ C_{2M}$

$\dots \quad \dots \quad \dots \quad \dots$

$C_{N1} \ C_{N2} \ \dots \ C_{NM}$

$5 \leq N, M \leq 500$

Формат выходных данных:


MaximalPossibleWeight


ГЛАВА 9. ГРАФЫ


9.1 Структура данных «Граф»

9.1.1 Основные понятия и определения теории графов

На самом деле, в предыдущих главах уже не раз упоминалась такая структура данных, как граф. Приведем основные определения теории графов.

 Пусть V – непустое конечное множество. Через $V(2)$ обозначим множество всех двуэлементных подмножеств из V . *Графом* G называется пара множеств (V, E) , где E – произвольное подмножество из $V(2)$. Элементы множеств V и E соответственно называются вершинами и ребрами графа.

 Если v_1, v_2 – вершины, а $e = (v_1, v_2)$ – соединяющее их ребро, тогда вершина v_1 и ребро e *инцидентны*, вершина v_2 и ребро e инцидентны. Две вершины, инцидентные одному ребру, называются *смежными*. Два ребра, инцидентные одной вершине также называются смежными. *Степень (валентность) вершины* $deg(v)$ – это количество ребер, инцидентных вершине v . Вершина называется *висячей*, если ее степень равна 1 и *изолированной*, если ее степень равна 0.

 Граф G называется *ориентированным графом (орграфом)*, если все его ребра являются ориентированными (рис. 9.1). Ориентированные ребра называются дугами. Дуга описывается как упорядоченная пара вершин (v, w) , где вершину v называют началом, а w – концом дуги.

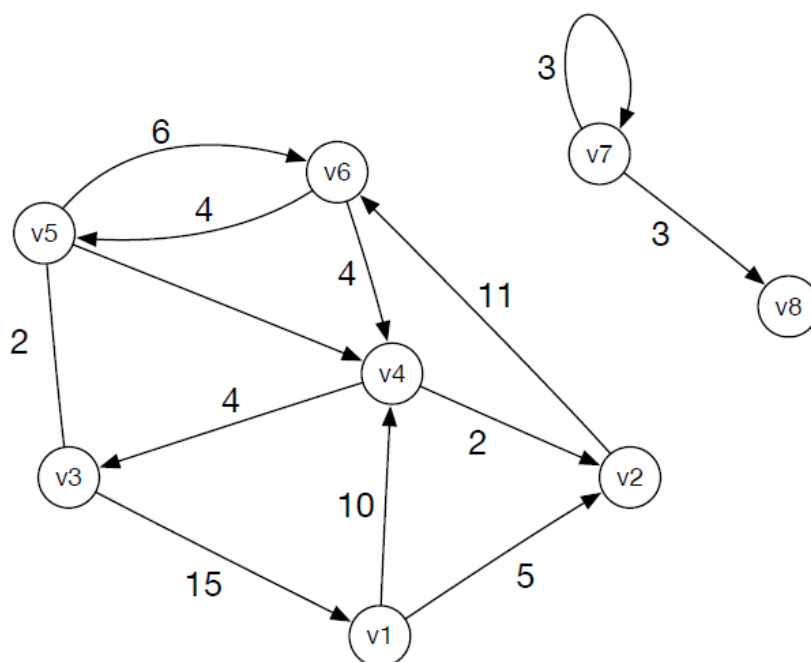


Рис. 9.1. Орграфы



Полным графом называется граф, в котором для каждой пары вершин (v_1, v_2) существует ребро, инцидентное v_1 и инцидентное v_2 . Иначе говоря, граф $G(V, E)$ называется полным, если любая пара его вершин соединена хотя бы в одном направлении.

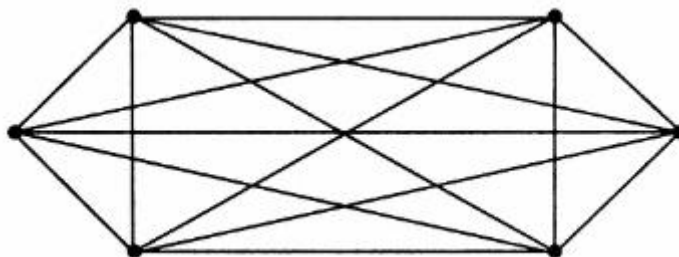


Рис. 9.2. Полный граф



Псевдографом называется граф, содержащий петли, а *мультиграфом* – граф, в котором существует пара вершин, соединенная более чем одним ненаправленным ребром, либо более чем двумя дугами противоположных направлений. *Пустой граф* – это граф без ребер.



Двудольным называется граф, множество V вершин которого разбито на два непересекающихся подмножества V_1 и V_2 , причем каждое ребро

графа соединяет вершину из V_1 с вершиной из V_2 . Множества V_1 и V_2 называются долями двудольного графа.



Маршрут в графе G – это чередующаяся последовательность вершин и ребер $v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k$, в которой любые два соседних элемента инцидентны. Если $v_0 = v_k$, то маршрут замкнут, иначе открыт. Длина маршрута – это количество содержащихся в нем ребер с возможными повторениями.



Цепью в графе называется маршрут, все ребра которого различны. Если при этом все вершины различны, то такая цепь называется простой.

Цикл в графе – это цепь, которая начинается и заканчивается в одной и той же вершине. Цикл, в котором нет повторяющихся вершин, кроме совпадающих начальной и конечной, называется простым циклом. Простой цикл орграфов называется контуром. *Гамильтоновым циклом* называется простой цикл, который проходит через все вершины графа. Граф, содержащий такой цикл, называется гамильтоновым графом.



Два графа называются *изоморфными*, если существует такая перестановка вершин, при которой они совпадают. *Планарным* графом называется граф, который может быть изображен на плоскости без пересечения ребер.



Подграф исходного графа – это граф, содержащий некое подмножество вершин данного графа и все ребра, инцидентные данному подмножеству. *Регулярным* называется граф, степени всех вершин которого одинаковы. Степень регулярности является инвариантом графа G и обозначается $r(G)$. Для нерегулярных графов $r(G)$ не определено.



Цикломатическое число графа – это число, равное увеличенной на единицу разности между количеством ребер и количеством вершин графа: $\gamma = n - m + 1$, где n – количество ребер, m – количество вершин. Цикломатическое число графа показывает, сколько ребер надо удалить из графа, чтобы в нем не осталось ни одного цикла.



Взвешенный граф – это граф, ребрам или дугам которого поставлены в соответствие действительные числа или величины, принимающие действительные значения.

9.1.2 Представление графа в памяти ЭВМ

Имеется несколько популярных способов представления графов в памяти ЭВМ: в виде матрицы смежности, матрицы инцидентности, множеств смежности и множества рёбер.

Представление графа в памяти в виде матрицы смежности.

Матрица смежности представляет собой квадратную матрицу Adj размером $|V| \times |V|$, где элемент Adj_{ij} есть вес ребра от узла V_i до V_j . Если граф невзвешенный, то обычно наличие связи обозначается единицей (или логической истиной), отсутствие – нулём или логической ложью (рис. 9.3). Это представление не позволяет описывать графы, содержащие кратные рёбра (мультирёбра).

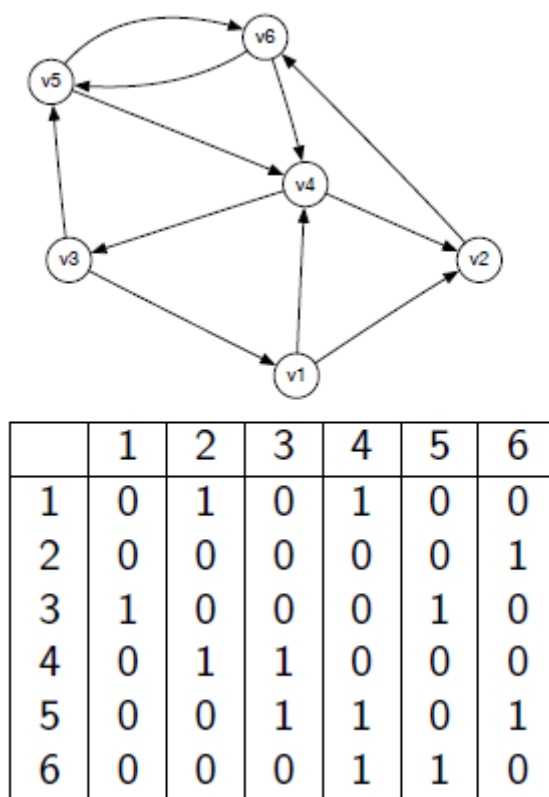


Рис. 9.3. Представление графа в виде матрицы смежности

Представление графа в памяти в виде матрицы инцидентности.

Матрица инцидентности представляет собой двумерный массив B размерности $|V| \times |E|$, где элемент B_{ij} отражает инцидентность вершины V_i ребру E_j .

Представление графа в памяти в виде множеств смежности.

Для каждого узла имеется множество смежных с ним узлов или соседей.

$v1 : \{2, 4\}$

$v2 : \{6\}$

$v3 : \{1, 5\}$

$v4 : \{2, 3\}$

$v5 : \{3, 4, 6\}$

$v6 : \{4, 5\}$

Для взвешенного графа элементы множеств смежности есть пары, один элемент которых есть номер смежного узла, а другой — вес связи. Мультирёбра в этом представлении, хотя и возможны, но они ограничивают гибкость представления.

$v1 : \{(2, 5), (4, 10)\}$

$v2 : \{(6, 11)\}$

$v3 : \{(1, 15), (5, 2)\}$

$v4 : \{(2, 2), (3, 4)\}$

$v5 : \{(4, 7), (6, 6)\}$

$v6 : \{(4, 4), (5, 4)\}$

Представление взвешенного графа в памяти в виде списка рёбер.

Для некоторых алгоритмов оказывается достаточным, если граф представлен массивом троек: {откуда, куда, стоимость}. Мультирёбра в этом представлении вполне возможны.

$\{\{1, 2, 5\}, \{1, 4, 10\}, \{2, 6, 11\}, \{3, 1, 15\}, \{3, 5, 2\},$

$\{4, 2, 2\}, \{5, 4, 7\}, \{5, 6, 6\}, \{6, 4, 4\}, \{6, 5, 4\}\}$

9.2 Обход графов

9.2.1 Поиск в ширину. Алгоритм BFS



Предшественник $\pi(u)$ на пути от s : предпоследняя вершина в кратчайшем пути из s в u .

Поиск в ширину от вершины s — просмотр вершин графа в порядке возрастания расстояния от s . Это позволяет, например, для невзвешенных графов решить задачу определения кратчайшего расстояния от одной вершины для других. Побочным эффектом данного алгоритма является установление достижимости одной вершины из другой.

В качестве вспомогательной структуры данных используется абстракция «Очередь» (Queue), очередь вершин, которые ещё не посещены алгоритмом. Сам алгоритм достаточно прост. Для каждой вершины u определим её цвет $c[u]$: белый будет обозначать, что вершина ещё не обработана, серый — что она обрабатывается, а чёрный — что обработка завершена. Второй атрибут вершины $d[u]$ — её расстояние до начальной. Алгоритм начинается выкрашиванием начальной вершины в серый цвет и установлением всех расстояний (кроме начальной вершины) в бесконечность. Начальная вершина отправляется в очередь и начинается главный цикл. На каждой итерации из очереди забирается очередная серая вершина и все её необработанные соседи помещаются в очередь, предварительно будучи выкрашены в серый цвет. После того, как все смежные вершины просмотрены и часть из них отправлена в очередь, текущая вершина выкрашивается в чёрный цвет.

```
procedure BFS( $G$  : Graph,  $s$  : Vertex)
  for all  $u \in V[G] \setminus \{s\}$  do
     $d[u] \leftarrow \infty$ ;  $c[u] \leftarrow \text{white}$ ;  $\pi[u] \leftarrow \text{nil}$ 
  end for
   $d[s] \leftarrow 0$ 
   $c[s] \leftarrow \text{grey}$ 
   $Q.\text{enqueue}(s)$ 
  while  $Q \neq \emptyset$  do
     $u \leftarrow Q.\text{dequeue}()$ 
    for all  $v \in \text{Adj}[u]$  do
```

```

        if c[v] = white then
            d[v] ← d[u] + 1
            π[v] = u
            Q.enqueue(v)
            c[v] = grey
        end if
    end for
    c[u] ← black
end while
end procedure

```

Проиллюстрируем работу данного алгоритма. На первом шаге все вершины являются белыми (кроме начальной), а очередь содержит начальную вершину.

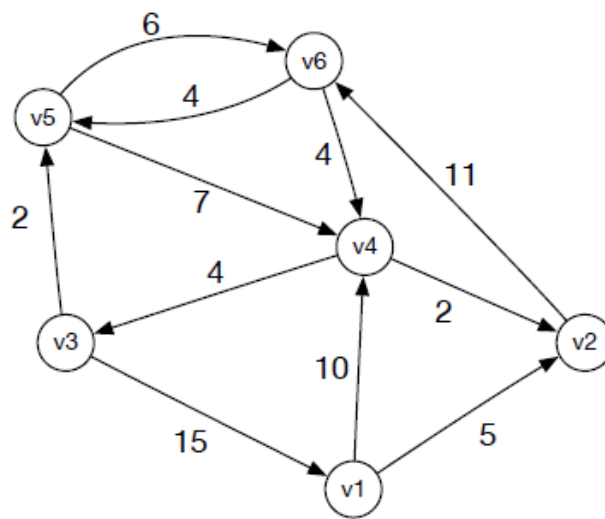


Рис. 9.4.

$d = \{0, \infty, \infty, \infty, \infty, \infty\}$

$\pi = \{\text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}\}$

$Q = \{v_1\}$

После первой итерации цикла While в очередь попали все смежные с v_1 вершины, так как они были белыми (рис. 9.5).

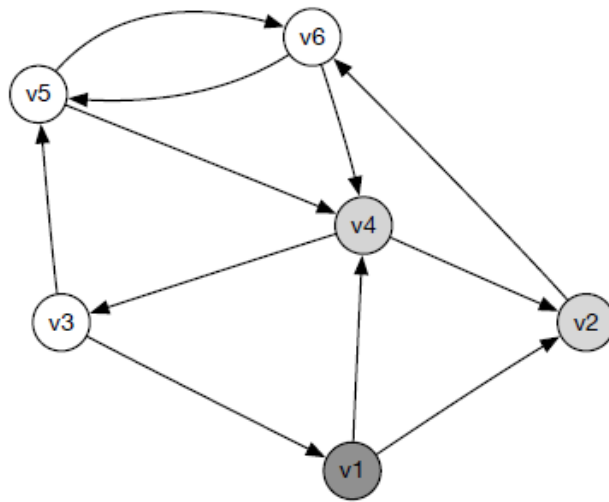


Рис. 9.5.

$$d = \{0, 1, \infty, 1, \infty, \infty\}$$

$$\pi = \{\text{nil}, v1, \text{nil}, v1, \text{nil}, \text{nil}\}$$

$$Q = \{v4, v2\}$$

При втором прохождении цикла While из очереди извлечена вершина v_4 и туда добавлена вершина v_3 . Соседняя с v_4 вершина уже имеет серый цвет и поэтому она игнорирована (рис. 9.6).

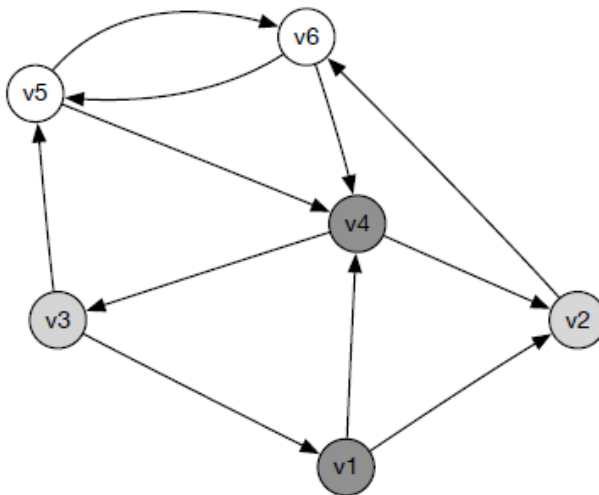


Рис. 9.6.

$$d = \{0, 1, 2, 1, \infty, \infty\}$$

$$\pi = \{\text{nil}, v1, v4, v1, \text{nil}, \text{nil}\}$$

$$Q = \{v2, v3\}$$

Третье прохождение цикла While уберёт из очереди v_2 и добавит в конец очереди v_6 (рис. 9.7).

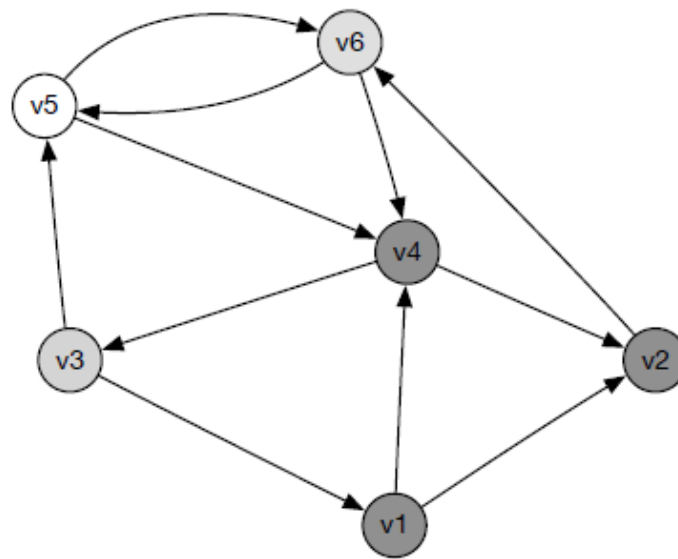


Рис. 9.7.

$$d = \{0, 1, 2, 1, \infty, 2\}$$

$$\pi = \{\text{nil}, v_1, v_4, v_1, \text{nil}, v_2\}$$

$$Q = \{v_3, v_6\}$$

После четвёртого прохождения цикла While очередь покинет вершина v_3 и в неё добавится вершина v_5 . После этого все вершины выкрашены в цвета, отличные от белого, и в очередь ничего добавляться больше не будет (рис. 9.8).

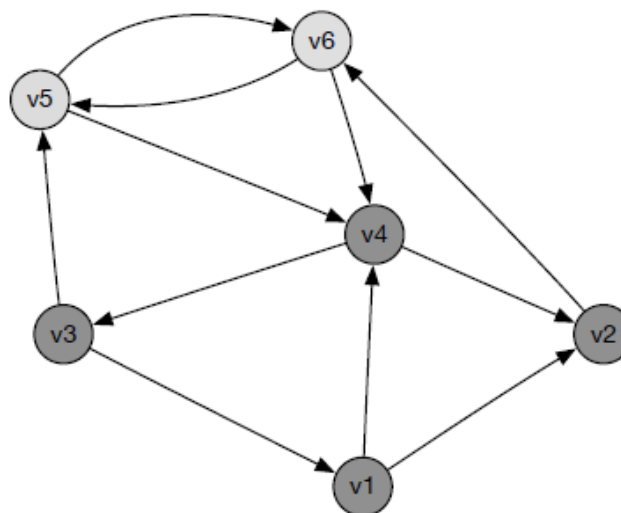


Рис. 9.8.

$$d = \{0, 1, 2, 1, 3, 2\}$$

$$\pi = \{\text{nil}, v1, v4, v1, v3, v2\}$$

$$Q = \{v6, v5\}$$

Завершение алгоритма даёт нам заполненные массивы расстояний от начальной вершины и предшественников.

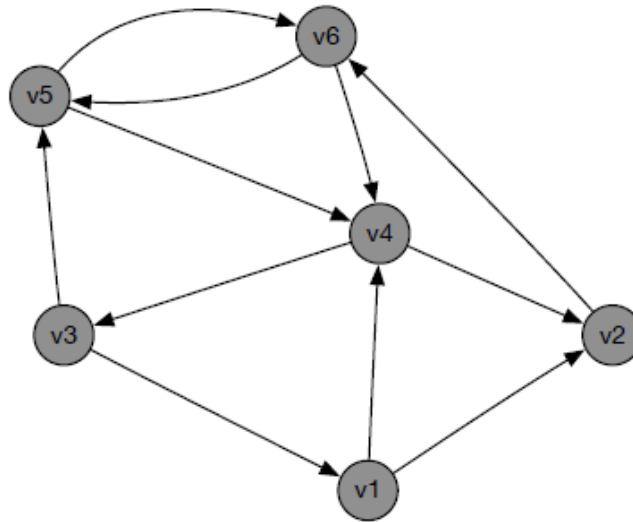


Рис. 9.9.

$$d = \{0, 1, 2, 1, 3, 2\}$$

$$\pi = \{\text{nil}, v1, v4, v1, v3, v2\}$$

$$Q = \{\}$$

Представление в виде множеств смежности для этого алгоритма наиболее удобно, так как позволяет легко узнать все смежные вершины. Сложность алгоритма посчитать несложно.

Фаза инициализации имеет сложность $O(|V|)$. Каждая вершина попадает в очередь не более одного раза. Для каждой попавшей в очередь вершины проверяются все смежные вершины, что имеет сложность:

$$\sum_{v \in V} |Adj(v)| = O(|E|)$$

Итоговая сложность алгоритма — $T = O(|V| + |E|)$.

Все окрашенные по завершении обхода вершины составляют компоненту связности для начальной вершины, то есть, множество достижимых из начальных вершин. К сожалению, является ли эта компонента связности

сильной, когда все вершины компоненты достижимы друг из друга, или слабой, этот алгоритм ответа не даёт.



Создайте новый проект. Опишите класс графа. Реализуйте алгоритм поиска в ширину.

9.2.2 Поиск в глубину. Алгоритм DFS

Этот алгоритм пытается идти вглубь, пока это возможно. Обнаружив вершину, алгоритм не возвращается, пока не обработает всех её потомков. Цвета, в которые окрашиваются вершины, те же самые: белый для непросмотренных вершин, серый для обрабатываемых вершин и чёрный для обработанных вершин.

Используются переменные

- $time$ — глобальные часы.
- $d[u]$ — время начала обработки вершины u .
- $f[u]$ — время окончания обработки вершины u .
- $\pi[u]$ — предшественник вершины u

Сам алгоритм часто используется для получения компонент связности всего графа, поэтому его делят на две части — нерекурсивную и рекурсивную. Нерекурсивная часть просто инициализирует переменные и запускает рекурсивный обход DFS-visit для каждой из необработанных вершин.

```
procedure DFS(G : Graph)
  for all  $u \in V[G]$  do
     $c[u] \leftarrow \text{white}; \pi[u] \leftarrow \text{nil}$ 
  end for
   $time \leftarrow 0$ 
  for all  $u \in V[G]$  do
    if  $c[u] = \text{white}$  then
      DFS-visit( $u$ )
    end if
  end for
end procedure
```

Рекурсивная часть на время выкрашивает вершину в серый цвет, устанавливает время входа $d[u]$ и рекурсивно вызывает себя для всех

необработанных смежных вершин. После завершения обработки, фиксируется время выхода $f[u]$ и вершина окрашивается в чёрный цвет.

```

procedure DFS-vizit(u : V ertex)
  c[u] ← grey
  time ← time + 1
  d[u] ← time
  for all v ∈ Adj[u] do
    if c[v] = white then
      π[v] ← u
      DFS-vizit(v)
    end if
  end for
  c[u] ← black
  time ← time + 1
  f[u] ← time
end procedure

```

Прогон алгоритма DFS начинается обход с вершины v_1 (рис. 9.10).

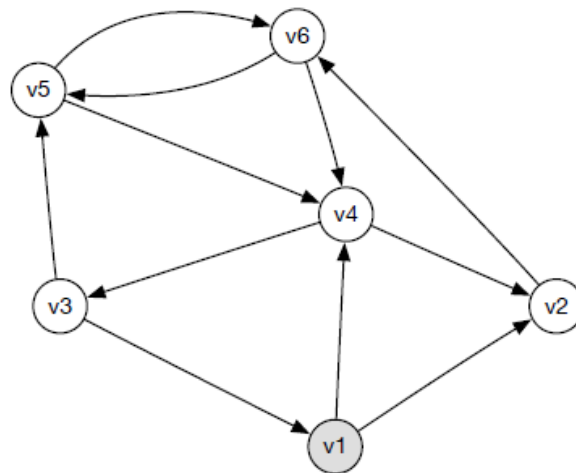


Рис. 9.10.

$d = \{1, -1, -1, -1, -1, -1\}$

$f = \{-1, -1, -1, -1, -1, -1\}$

$\pi = \{\text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}\}$

Первый рекурсивный вызов $\text{DFS-vizit}(v_2)$ (рис. 9.11).

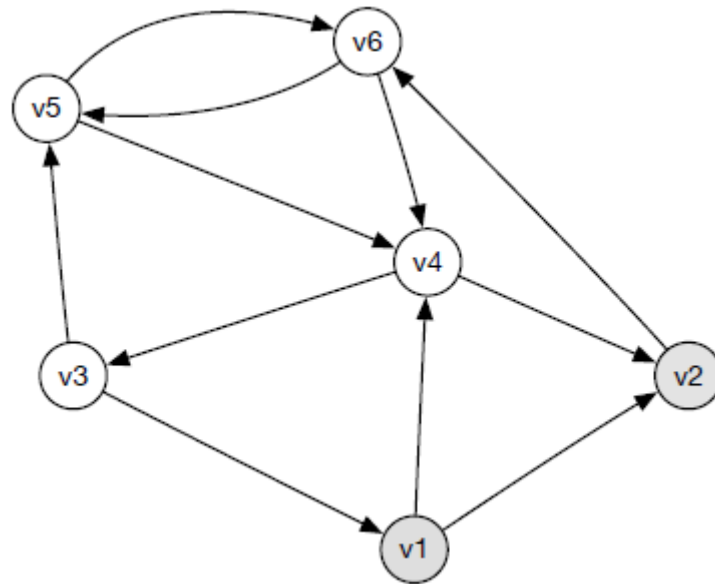


Рис. 9.11.

$D = \{1, 2, -1, -1, -1, -1\}$

$f = \{-1, -1, -1, -1, -1, -1\}$

$\pi = \{\text{nil}, v1, \text{nil}, \text{nil}, \text{nil}, \text{nil}\}$

Второй рекурсивный вызов DFS-vizit(v_6) (рис. 9.12).

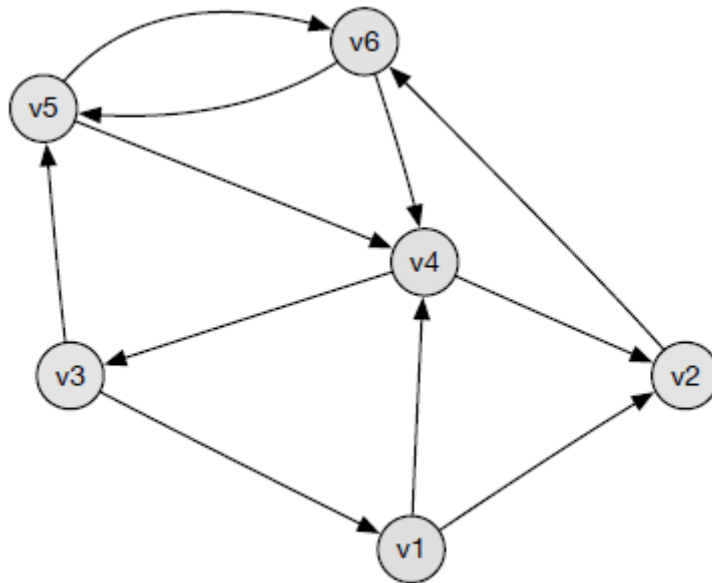


Рис. 9.12

$d = \{1, 2, -1, -1, -1, 3\}$

$f = \{-1, -1, -1, -1, -1, -1\}$

$\pi = \{\text{nil}, v1, \text{nil}, \text{nil}, \text{nil}, v2\}$

Далее рекурсия идёт по вершинам v_4 , v_3 и v_5 , после чего оказывается, что у вершины v_5 нет необработанных соседей, следовательно, из рекурсии на v_5 нужно выйти (рис. 9.13).

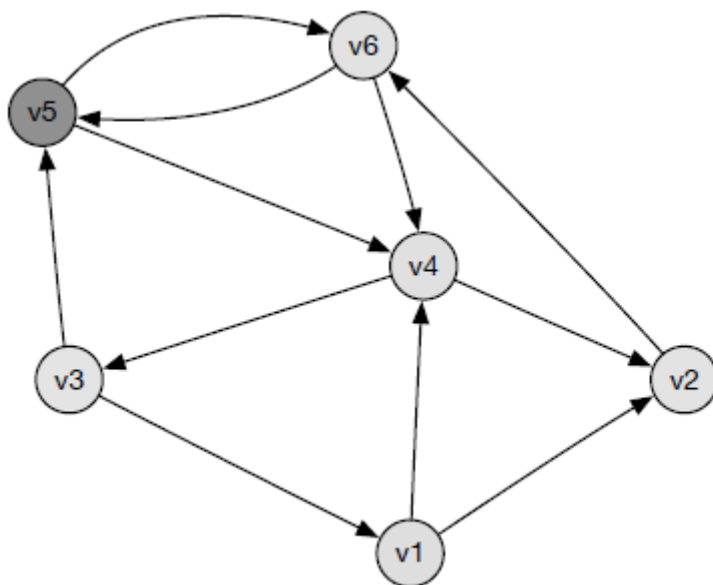


Рис. 9.13.

$d = \{1, 2, 5, 4, 6, 3\}$

$f = \{-1, -1, -1, -1, 7, -1\}$

$\pi = \{\text{nil}, v1, v4, v6, v3, v2\}$

Для всех остальных вершин рекурсия тоже завершается (рис. 9.14).

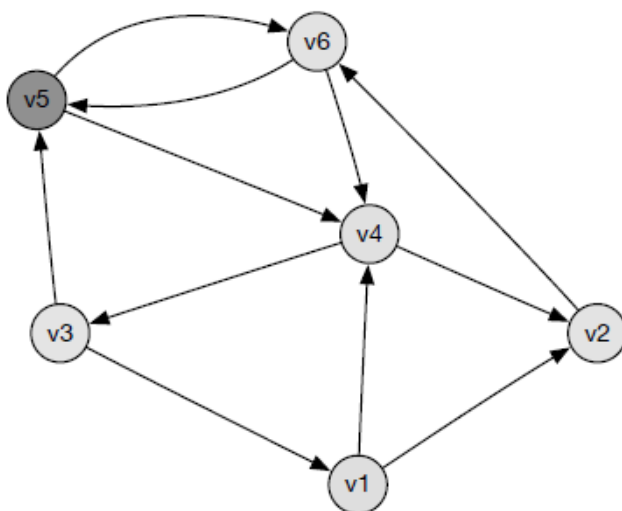


Рис. 9.14.

$d = \{1, 2, 5, 4, 6, 3\}$

$$f = \{12, 11, 8, 9, 7, 10\}$$

$$\pi = \{\text{nil}, v1, v4, v6, v3, v2\}$$

Чтобы определить сложность алгоритма будем полагать, что граф представлен множествами смежности. Тогда инициализация потребует $O(|V|)$, посещение каждой вершины ровно по одному разу — $O(|V|)$ и проверка каждого ребра — $O(|E|)$, итого — $O(|V| + |E|)$.

Данный алгоритм хорошо применим для нахождения компонент связности или топологической сортировки.



Продолжайте работу в созданном проекте. Реализуйте алгоритм поиска в глубину.

9.2.3 Топологическая сортировка

Задача. Имеется ориентированный граф $G = (V, E)$ без циклов. Требуется указать такой порядок вершин на множестве V , что любое ребро ведёт из меньшей вершины к большей.

Нам потребуется небольшая модификация алгоритма DFS и структура данных Queue. Времена входов и выходов наряду с предшественниками не понадобятся.

```

procedure TopoSort(G : Graph)
    L ← ∅
    for all u ∈ V [G] do
        c[u] ← white;
    end for
    for all u ∈ V [G] do
        if c[u] = white then
            DFS-vizit(u)
        end if
    end for
end procedure

procedure DFS-vizit(u : V ertex)
    c[u] ← grey
    for all v ∈ Adj[u] do
        if c[v] = white then
            DFS-vizit(u)
        end if
    end for
end procedure

```

```

c[u] ← black
L.insert(u)
end procedure

```

Пусть обход следующего графа начнётся с вершины v_1 (рис. 9.15)

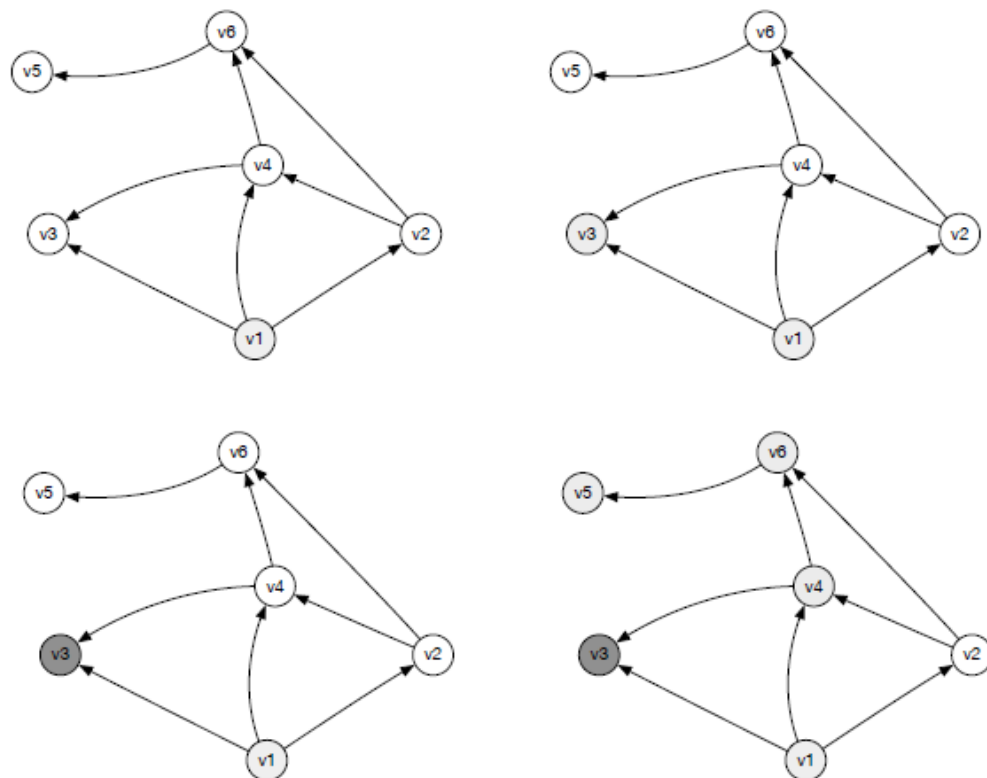


Рис. 9.15.

В результате обхода около каждой вершины написан её порядковый номер (рис. 9.16).

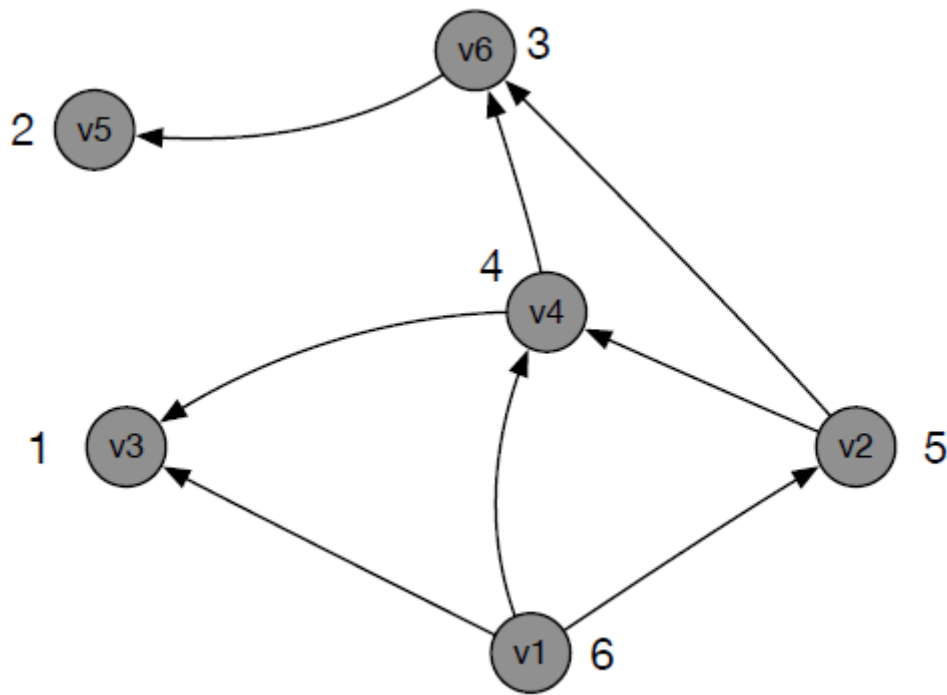


Рис. 9.16.

Порядок вершин (добавляем в начало по номерам): $V_1, V_2, V_4, V_6, V_5, V_3$. А теперь вспомним метод динамического программирования. Мы использовали термин «порядок на подзадачах». Теперь всё это можно сказать более строго: подзадачи должны образовывать ациклический направленный граф. Процедура топологической сортировки на этом графе нам и даст порядок решения этих подзадач, так как будет гарантировано, что все подзадачи решаются раньше самой задачи.



Продолжайте работу в созданном проекте. Реализуйте алгоритм топологической сортировки.

9.2.4 Поиск компонент связности: алгоритм Косарайю

Для неориентированных графов найти компоненты связности можно, например, запустив поиск BFS или DFS. Все выкрашенные по завершении поиска вершины образуют компоненту связности. После этого выбирается произвольным образом необработанная вершина и алгоритм повторяется,

формируя другую компоненту связности. Алгоритм заканчивается, когда не остаётся необработанных вершин.

Если требуется найти компоненты сильной связности, то для ориентированных графов эта процедура не приведёт к правильному решению. Требуется другой алгоритм, например, прост в реализации алгоритм Косарайю (Kosaraju).

Попробуем найти все компоненты сильной связности для следующего графа, представленного на рис. 9.17.

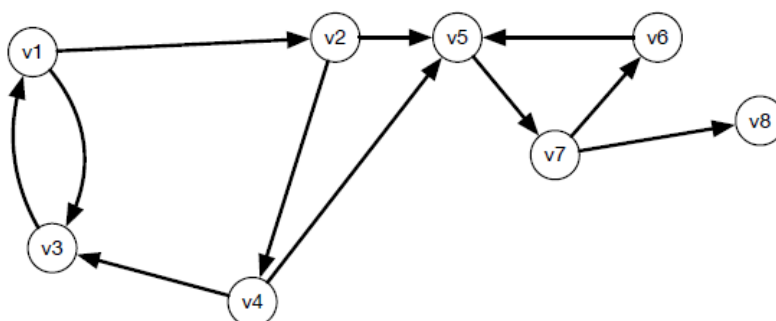


Рис. 9.17.

Для данного графа проведём полный DFS поиск. Так как в алгоритме полного DFS не специфицировано, с какой вершины начинается поиск, можно выбрать произвольную. Для наблюдения за алгоритмом около каждой вершины будем писать два числа: время входа в вершину и время выхода из вершины.

Начав обход с вершины v_2 , мы можем получить результат, представленный на рис. 9.18.

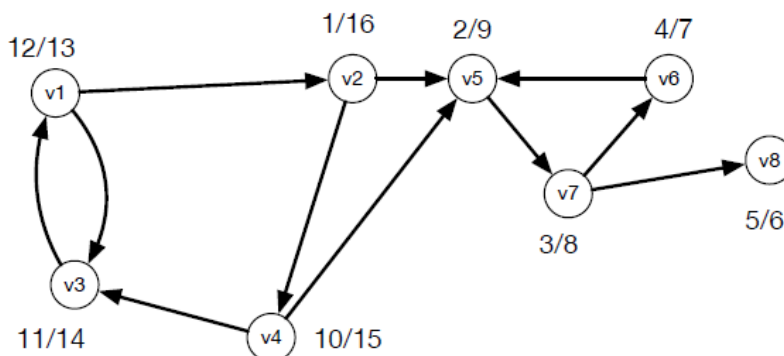


Рис. 9.18.

Составим таблицу времени входа/выхода для всех вершин (табл. 9.1).

Табл. 9.1.

Номер вершины	1	2	3	4	5	6	7	8
Время входа/выхода	12/13	1/16	11/14	10/15	2/9	4/7	3/8	5/6

Следующая операция – заменить направления всех связей (перевернуть все стрелки). Если из вершины u была связь с вершиной v , то после переворота вершина v получит связь с вершиной u (рис. 9.19).

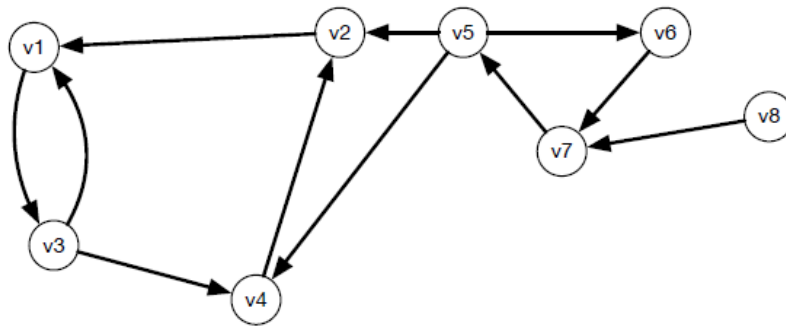


Рис. 9.19.

Этот граф нужно обойти ещё раз. Но теперь порядок обхода уже не произволен. Мы должны выбрать в качестве начальной вершины ту из необработанных, у которой наибольшее значение времени выхода. В нашем случае это вершина 5. Обход покрасил вершины v_1 , v_2 , v_3 и v_4 . После завершения обхода, остались непокрашенные вершины v_5 , v_6 , v_7 и v_8 . Из них снова выбираем вершину с наибольшим временем выхода — и так до тех пор, пока останутся непокрашенные вершины.

Каждый «малый» проход алгоритма DFS даст нам вершины, которые принадлежат одной компоненте сильной связности (рис. 9.20).

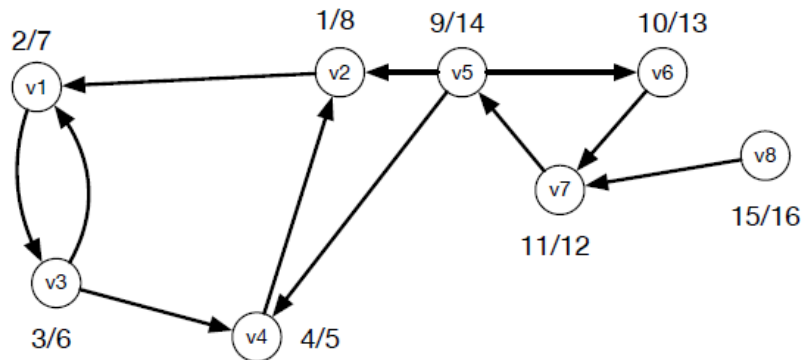


Рис. 9.20.

Рассматривая компоненту сильной связности как единую мета-вершину, мы получаем новый граф, который называется конденсацией исходного графа или конденсированным графом (рис. 9.21).

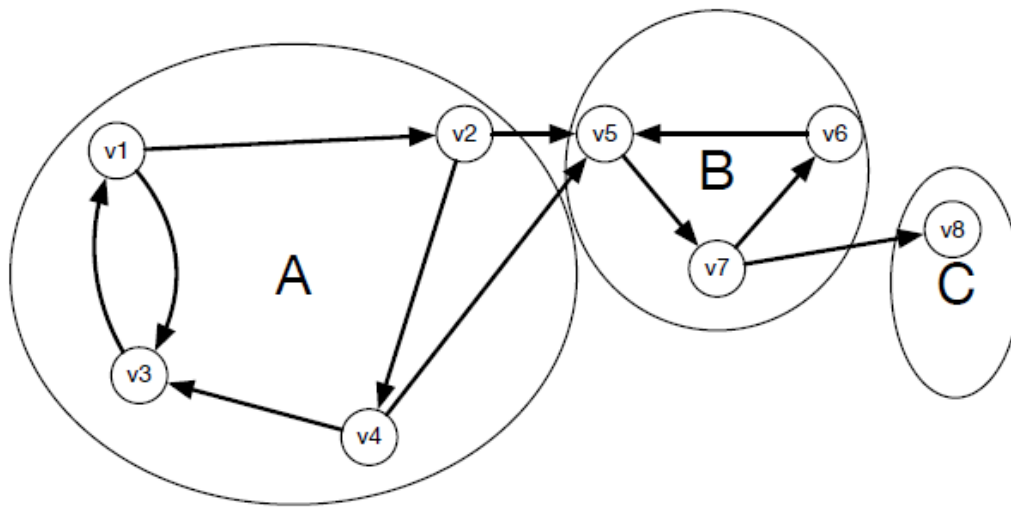


Рис. 9.21.



Продолжайте работу в созданном проекте. Реализуйте алгоритм Косарайю.

9.3 Алгоритмы построения минимального остовного дерева

9.3.1 Остовное дерево: основные понятия и свойства



С точки зрения теории графов *дерево* есть ациклический связный граф. Множество деревьев называется *лесом* (*forest*) или бором. *Остовное дерево* связного графа — подграф, который содержит все вершины графа и представляет собой полное дерево. *Остовный лес* графа — лес, содержащий все вершины графа.

Построение остовных деревьев — одна из основных задач в компьютерных сетях. Решение таких задачи позволит оптимально спланировать маршрут от одного узла сети до других. Проблема состоит в том, что для некоторого типа узлов в передаче сообщений недопустимо иметь несколько возможных маршрутов. Например, если компьютер соединён с маршрутизатором по Wi-Fi и Ethernet одновременно, то в некоторых

операционных системах сообщения от компьютера до маршрутизатора не будут доходить из-за наличия цикла. Построение остовного дерева — избавление от циклов в графе.

Остовных деревьев может быть много. Например, для графа, представленного на рис. 9.22, возможные остовные деревья изображены на рис. 9.23, 9.24.

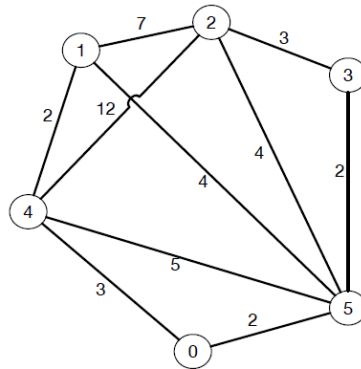


Рис. 9.22.

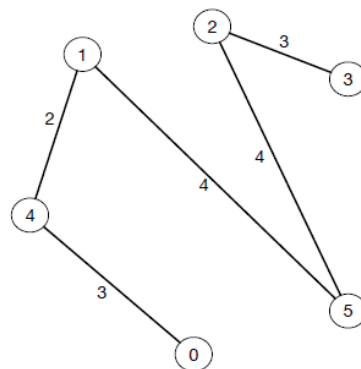


Рис. 9.23.

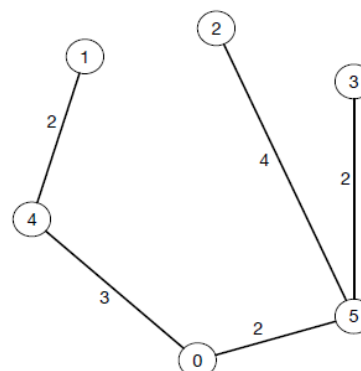





Рис. 9.24.


Чаще всего задача формулируется как нахождение минимального остовного дерева.

 *MST – Minimal Spanning Tree – минимальное остовное дерево* взвешенного графа – есть остовное дерево, вес которого (сумма его всех рёбер) не превосходит вес любого другого остовного дерева.


Именно минимальные остовные деревья больше всего интересуют проектировщиков сетей.

 *Сечение графа* — разбиение множества вершин графа на два непересекающихся подмножества. *Перекрёстное ребро* — ребро, соединяющее вершину одного множества с вершиной другого множества.

 Если T — произвольное остовное дерево, то добавление любого ребра e между двумя вершинами u и v создаёт цикл, содержащий вершины u , v и ребро e .

 При любом сечении графа каждое минимальное перекрёстное ребро принадлежит некоторому MST-дереву и каждое MST-дерево содержит перекрёстное ребро

Каждое ребро дерева MST есть минимальное перекрёстное ребро, определяемое вершинами поддеревьев, соединённых этим ребром.

 Пусть имеется граф G и ребро e . Пусть граф G' есть граф, полученный добавлением ребра e к графу G . Результатом добавления ребра e в MST графа G и последующего удаления максимального ребра из полученного цикла будет MST графа G' . Эта лемма выявляет рёбра, которые не должны входить в MST.

9.3.2 Алгоритм Прима

Алгоритм Прима строит MST, используя в каждый момент времени сечение графа на два подграфа. Один подграф, вначале состоящий из одной корневой вершины, содержит те вершины, которые входят в уже построенный фрагмент MST. Другой подграф содержит все оставшиеся вершины. Алгоритм

шаг за шагом добавляет по одной вершины в первое множество, убирая её из второго. Как только новой подходящей вершины найти не удаётся, алгоритм завершается.

1. Выбираем произвольную вершину. Это – MST дерево, состоящее из одной древесной вершины.

2. Выбираем минимальное перекрёстное ребро между MST вершиной и недревесным множеством.

3. Повторяем операцию до тех пор, пока все вершины не окажутся в дереве.

Рассмотрим данный алгоритм на примере графа, представленного на рис. 9.25.

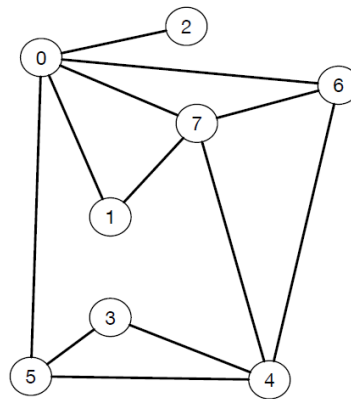


Рис. 9.25.

Определяем вершину 0 как корневую и переводим её в MST. Проверяем все веса из MST в не MST (рис. 9.26). В качестве весов в данной задаче рассматривается длина ребер.

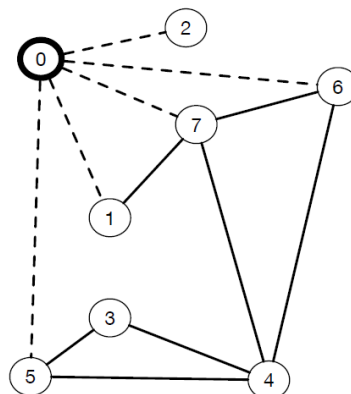


Рис. 9.26.

Самым легким ребром является ребро $(0, 2)$. Поэтому, переводит вершину 2 и ребро $(0, 2)$ в MST (рис. 9.27).

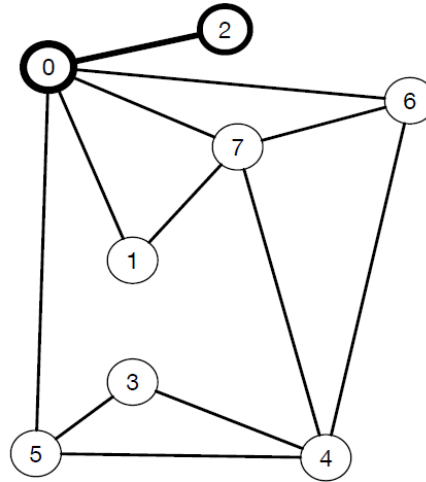


Рис. 9.27.

Следующим шагом в минимальное остовное дерево попадает вершина 7 и ребро $(1, 7)$. Еще раз стоит отметить, что на каждом шаге рассматриваются все ребра, не входящие в MST и инцидентные вершинам, входящим в MST. Четвертый шаг определяет новую вершину – 1 – для вхождения в MST (рис. 9.28).

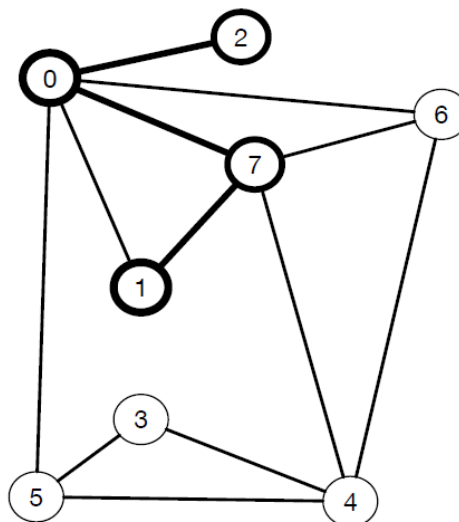


Рис. 9.28.

Затем, в MST попадут вершины 6, 4, 3 и завершится алгоритм на вершине 5 (рис. 9.29).

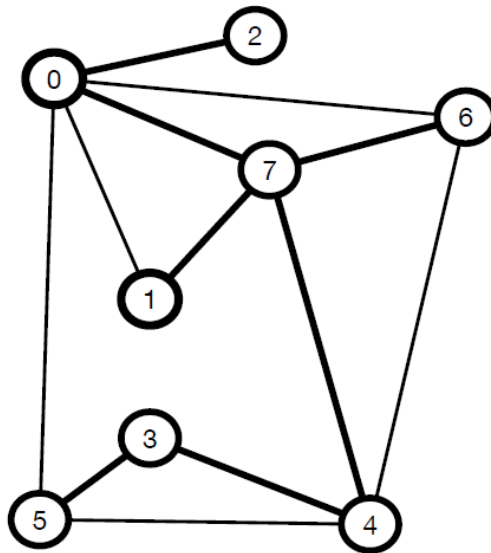


Рис. 9.29.

При каждой итерации в MST включается самое лёгкое ребро. Сложность алгоритма $O(|V|^2)$. В данном виде алгоритм не очень эффективен. При поиске очередного ребра из MST в не-MST мы забываем про те рёбра, который уже проверяли. Задача упростится, если мы введём накопитель.

Накопитель – структура данных, которая содержит множество рёберкандидатов

Более эффективная реализация алгоритма Прима, использующая накопитель, выглядит следующим образом:

1. Выбираем произвольную вершину. Это – MST дерево, состоящее из одной вершины. Делаем вершину текущей.
2. Помещаем в накопитель все рёбра, которые ведут из этой вершины в не MST узлы. Если в какой-либо из узлов уже ведёт ребро с большей длиной, заменяем его ребром с меньшей длиной.
3. Выбираем ребро с минимальным весом из накопителя.
4. Повторяем операцию до тех пор, пока все вершины не окажутся в дереве.

Алгоритм Прима – обобщение поиска на графе. Если представить накопитель как приоритетную очередь с операциями «добавить элемент»,

«извлечь минимальное» и «увеличить приоритет», то сложность алгоритма составит $O(|E| \log |V|)$. Действительно, операции с приоритетной очередью выполняются за $O(\log N)$, а в нашем случае N не превосходит $|V|$.

Такое обобщение поиска на графе носит название PFS — поиск по приоритету.



Продолжайте работу в созданном проекте. Реализуйте алгоритм Прима.

9.3.3 Алгоритм Краскала

Один из самых старых алгоритмов на графах — это алгоритм Краскала, который известен с 1956 года, но затем был забыт как непрактичный до момента изобретения подходящей структуры данных и связанного с ней вспомогательного алгоритма

Сам алгоритм — один из наиболее изящных в теории графов. Предварительным

условием алгоритма является связность графа. Заключается он в следующем

1. Создаётся число непересекающихся множеств по количеству вершин и каждая вершина составляет своё множество.
2. Множество MST вначале пусто.
3. Среди всех рёбер, не принадлежащих MST выбирается самое короткое из всех рёбер, которые не образуют цикл. Это значит, что вершины ребра должны принадлежать различным множествам.
4. Выбранное ребро добавляется к множеству MST
5. Множества, которым принадлежат вершины выбранного ребра, сливаются в единое.
6. Если размер множества MST стал равен $|V| - 1$, то алгоритм завершён, иначе отправляемся к пункту 3.

Проиллюстрируем алгоритм на том же графе, который применялся в алгоритме Прима. Для удобства добавим веса рёбер (рис. 9.30).

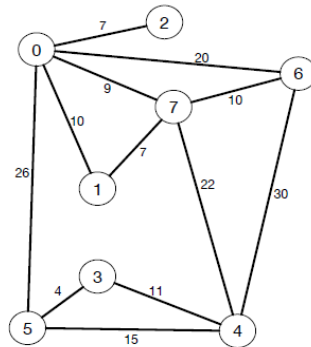


Рис. 9.30.

Создадим список ребер, упорядоченный по возрастанию (табл. 9.2) и таблицу, показывающую, какому множеству принадлежит вершина (табл. 9.3).

Табл. 9.2.

i	3	0	1	0	0	6	3	4	0	0	4
j	5	2	7	7	1	7	4	5	6	5	6
W_{ij}	4	7	7	9	10	10	11	15	22	26	30

Табл. 9.3.

V_i	0	1	2	3	4	5	6	7
p	0	1	2	3	4	5	6	7

На первой итерации выбирается самое короткое ребро (3, 5), имеющее длину 4 (рис. 9.31). Так как вершины с номерами 3 и 5 принадлежат разным множествам, отправляем выбранное ребро в множество MST и объединяем множества (табл. 9.4). Для простоты будем полагать, что при объединении множеств они получают меньший из номеров.

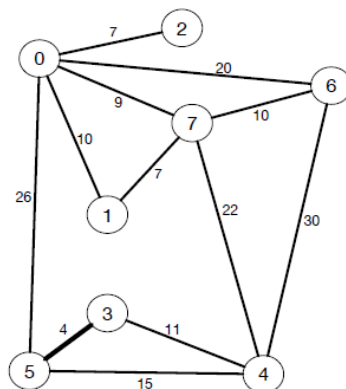


Рис. 9.31.

Табл. 9.4.

V_i	0	1	2	3	4	5	6	7
p	0	1	2	3	4	3	6	7

На второй итерации среди оставшихся рёбер имеются два подходящих с одинаковым весом. Мы можем из них выбрать произвольное. Если добавление первого ребра не мешает добавлению второго, то, очевидно всё в порядке. Если же оно мешает, то есть, после добавления первого ребра добавление второго создаст цикл, то, очевидно, удаление любого из рёбер решит проблему и общий вес дерева останется неизменным.

Выберем произвольным образом ребро $(0,2)$ и поместим вершину 2 в множество номер 0 (рис. 9.32, табл. 9.5).

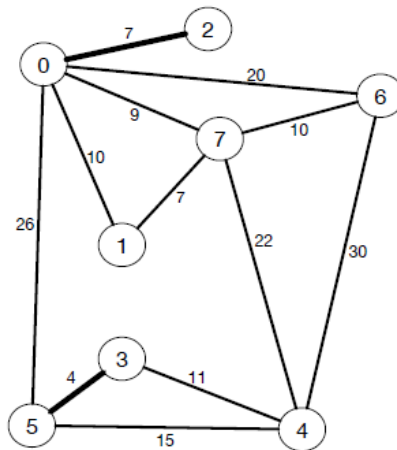


Рис. 9.32.

Табл. 9.5.

V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	7

На следующей итерации выбирается ребро $(1, 7)$, что приводит к слиянию первого и седьмого множеств. Далее, самым коротким ребром является ребро $(0, 7)$, что приводит к объединению множеств, содержащих вершины $\{0, 2\}$ и $\{1, 7\}$. Предположим, что новое множество получит номер 0. Что теперь делать с массивом p , в котором записаны номера множеств для вершин? Прямолинейная реализация алгоритма говорит нам, что нужно найти в массиве p все единицы (номер второго множества) и заменить их на нули (номер того

множества, куда переходят элементы первого). К счастью, для эффективного решения этой задачи существует подходящая структура данных: система непересекающихся множеств (Disjoint Set Union – DSU). Немного отступим от разбора примера, чтобы её пояснить.

Абстракция DSU реализует три операции:

1. **create(n)** — создать набор множеств из n элементов. Это мы и сделали, когда создали массив p при решении задачи.

2. **find_root(x)**. Вместо помещения номера множества в каждую из вершин множества (что, как мы видели, требует поиска по всему массиву и достаточно неэффективно), давайте введём понятие представителя множества. В нашем случае массив p пока имеет вид, представленный в табл. 9.6.

Табл. 9.6.

V _i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	1

Для вершины 7, например, массив p хранит число 1 — номер множества, он же и представитель множества 1. Если мы, не глядя, для слияния множеств 0,2 и 1,7 поместим в p[7] число 0, то в дальнейшем мы получим проблемы: для вершины 1 представителем останется 1, что, очевидно, неверно, так как после слияния вершина 1 должна принадлежать множеству 0. Сейчас p[7]=1 и это означает, что и у седьмой и у первой вершины представители одинаковые. Если номер вершины совпадает с номером представителя, то, очевидно, что для этой вершины в массив p ничего не было записано в процессе прохождения алгоритма, то есть, эта вершина есть корень дерева. Таким образом, после слияния нам нужно просто заменить всех родителей вершины на нового представителя. Это делается изящным рекурсивным алгоритмом:

```
int find_root(int r)
{
    if (p[r] == r) return r; // Тривиальный случай
    return p[r] = find_root(p[r]); //Рекурсивный случай
}
```

3. **merge(l,r)** — сливает два множества. Для сохранения корректности алгоритма вполне достаточно любого из присвоений: $p[l] = r$ или $p[r] = l$. Всю дальнейшую корректировку родителей в дальнейшем сделает метод `find_root`.

На практике для повышения эффективности используются несколько приёмов. Одним из них является использование ещё одного массива, хранящего длины деревьев: слияние производится к более короткому дереву. Вторым — случайный выбор дерева-приёмника.

```
void merge(int l, int r)
{
    l = find_root(l);
    r = find_root(r);
    if (rand() % 1)
    {
        p[l] = r;
    }
    else
    {
        p[r] = l;
    }
}
```

Обратите внимание на то, что внутри операции слияния имеется операция поиска, которая заменяет аргументы значениями корней их деревьев

Вернёмся к нашей задаче. Для определения, каким деревьям принадлежат концы ребра (0,7) мы дважды вызовем `find_root`. `find_root(0)` проверит, что в элементе $p[0]$ находится 0 и вернёт его, как номер множества.

А вот `find_root(7)` сначала убедится, что в $p[7]$ лежит 1 и вызовет `find_root(1)`, после чего, возможно, заменит $p[7]$ на 1.

Теперь мы убедились, что концы ребра 0 принадлежат разным множествам и сливаем множества, вызвав `merge(0,7)`. Первое, что сделает операция `merge` — заменит свои аргументы, 0 и 7, корнями деревьев, которым принадлежат 0 и 7, то есть, 0 и 1 соответственно. После этого в $p[1]$ помещается 0 и деревья слиты. Обратите внимание на то, что в $p[7]$ всё ещё находится 1.

На следующей итерации алгоритма выбирается ребро (6, 7) (рис. 9.33).

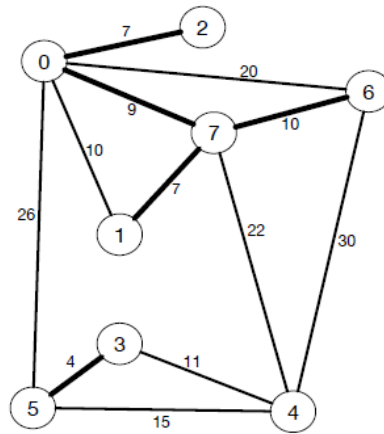


Рис. 9.33.

Поиск корня дерева для седьмой вершины приведёт к тому, что для вершины 7 будет установлен корень 0 и поэтому $p[7]$ станет равным 0. Это же значение будет присвоено и $p[6]$ (табл. 9.7).

Табл. 9.7.

V_i	0	1	2	3	4	5	6	7
p	0	0	0	3	4	3	0	0

На следующем этапе ребро (3,4) окажется самым коротким. Поиск множеств, которым принадлежат концы следующих рёбер (4,5) и (0,6) покажет, что эти рёбра принадлежат одному множеству, а вот ребро (4,7) подойдет (рис. 9.34).

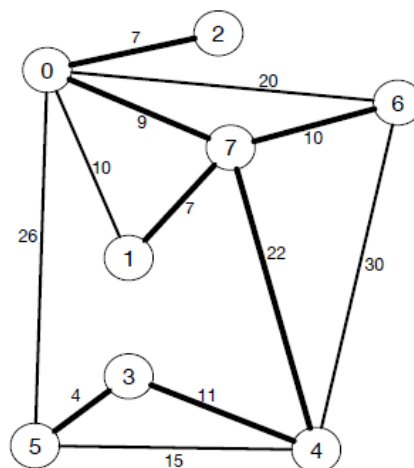


Рис. 9.34.

На этом алгоритм завершается, так как количество рёбер в множестве MST достигло $7 = N - 1$.

Как оценить сложность алгоритма? Первая часть алгоритма – сортировка рёбер. Сложность этой операции $O(|E| \log |E|)$. А как оценить сложность операции поиска и слияния? В 1984 году Tarjan, доказал, используя функцию Аккермана, что операция поиска в DSU имеет амортизированную сложность $O(1)$. Таким образом, сложность всего алгоритма Краскала и есть $O(|E| \log |E|)$. Для достаточно разреженных графов он обычно быстрее алгоритма Прима, для заполненных — наоборот.



Продолжайте работу в созданном проекте. Реализуйте алгоритм Краскала. Сравните его работу с реализованным ранее алгоритмом Прима для разных наборов графов.

9.4 Поиск кратчайших путей на графах

9.4.1 Алгоритм Дейкстры



Дерево кратчайших путей (SPT) для s – это подграф, содержащий s и все вершины, достижимые из s , образующий направленное поддерево с корнем в s , где каждый путь от вершины s до вершины u является кратчайшим из всех возможных путей.

Задача. Пусть задан граф G и вершина s . Построить SPT для s .

Решение задачи. Алгоритм Дейкстры строит SPT, определяя длины кратчайших путей от заданной вершины до остальных. Веса рёбер могут принимать произвольные неотрицательные значения. Если в графе есть отрицательные рёбра, но нет отрицательных циклов, то можно применить алгоритм Форда-Беллмана.

Алгоритм Дейкстры – ещё один вариант класса жадных алгоритмов PFS (как мы помним, к этому классу алгоритмов относится и алгоритм Прима). Логика алгоритма такова:

1. В SPT заносится корневой узел (исток).

2. На каждом шаге в SPT добавляется одно ребро, которое формирует кратчайший путь из истока в не-SPT.

3. Вершины заносятся в SPT в порядке их расстояния по SPT от начальной вершины.

Обозначим за U множество вершин, принадлежащих уже найденному оптимальному множеству. На каждом шаге мы будем исследовать вершины, не принадлежащие U и одну из них добавлять в U . Жадная стратегия заключается в том, что после того, как вершина попала в оптимальное множество, она его не покидает.

В начале алгоритма оптимальное множество U состоит из одного элемента — вершины s .

Длины кратчайших путей до вершин множества обозначим, как $d(s, v)$, $v \in U$.

Основной шаг — нахождение среди вершин, смежных с U , такой вершины u , $u \notin U$ что достигается минимум:

$$\min_{v \in U, u \notin U} d(s, v) + w(v, u)$$

После нахождения этой вершины она добавляется в множество U : $U \leftarrow U \cup \{u\}$. Эта операция повторяется до тех пор, пока множество U может пополняться.

В реализации алгоритма используются переменные:

- $d[u]$ — длина кратчайшего пути из вершины s до вершины u ;
- $\pi[u]$ — предшественник u в кратчайшем пути от s ;
- $w(u, v)$ — вес пути из u в v (длина ребра, вес ребра, метрика пути);
- Q — приоритетная по значению d очередь узлов на обработку;
- U — множество вершин с уже известным финальным расстоянием.

Сам алгоритм формализуется следующим образом:

```
procedure Dijkstra( $G$  : Graph;  $w$  : weights;  $s$  : Vertex)
  for all  $v \in V$  do
     $d[v] \leftarrow \infty$ 
     $\pi[v] \leftarrow \text{nil}$ 
```



```

end for
d[s] ← 0
U ← ∅
Q ← V
while Q ≠ ∅ do
    u ← Q.extractMin()
    U ← U ∪ {u}
    for all v ∈ Adj[u], v ∉ U do
        Relax(u, v)
    end for
end while
end procedure
procedure Relax(u, v : Vertex)
    if d[v] > d[u] + w(u, v) then
        d[v] = d[u] + w(u, v)
        π[v] ← u
    end if
end procedure

```

Здесь мы сталкиваемся с новым понятием — релаксацией. В данном алгоритме *релаксация* есть корректировка кратчайшего пути между двумя вершинами, если находится путь через какую-либо промежуточную вершину.

Попробуем посмотреть, как изменяются данные при прохождении алгоритма для какого-то исходного графа (рис. 9.35).

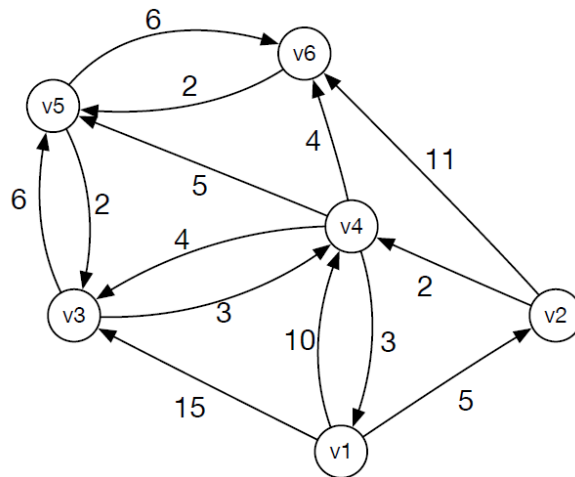


Рис. 9.35.

В начале алгоритма SPT содержит одну вершину, v_1 . Просмотр смежных вершин добавляет в накопитель три вершины — v_2 , v_3 и v_4 . Соответственно, элементы массива d_2 , d_3 и d_4 становятся равны 5, 15 и 10 (рис. 9.36). Обратите внимание, как это происходит, например, для d_2 : так как d_2 пока равно ∞ , и $d_1 + w_{12} < d_2$, то происходит релаксация пути и d_2 становится равно $d_1 + w_{12}$.

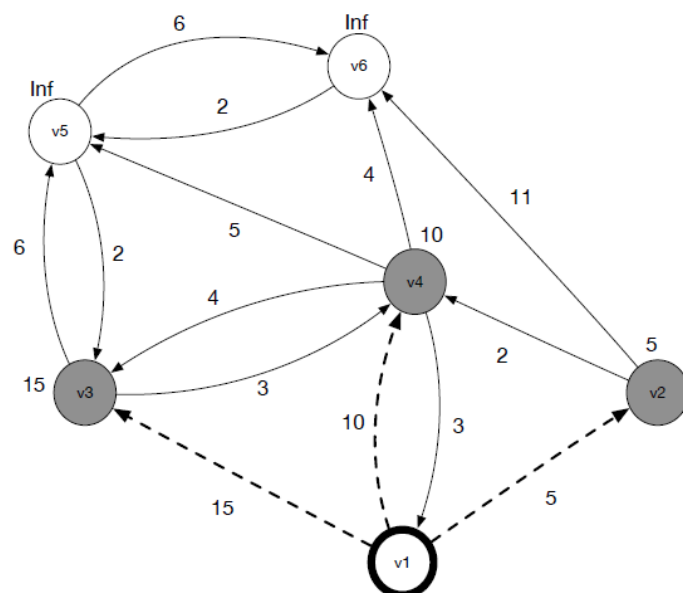


Рис. 9.36.

Из накопителя выбирается элемент с наименьшим значением d , это – второй элемент. Для всех исходящих связей проверяется возможность релаксации. Так как $d_2 + w_{24} < d_4$, то d_4 становится равным $d_2 + w_{24}$, а это значит, что старый наилучший маршрут ($1 \rightarrow 4$) заменён на ($1 \rightarrow 2 \rightarrow 4$) (рис. 9.37).

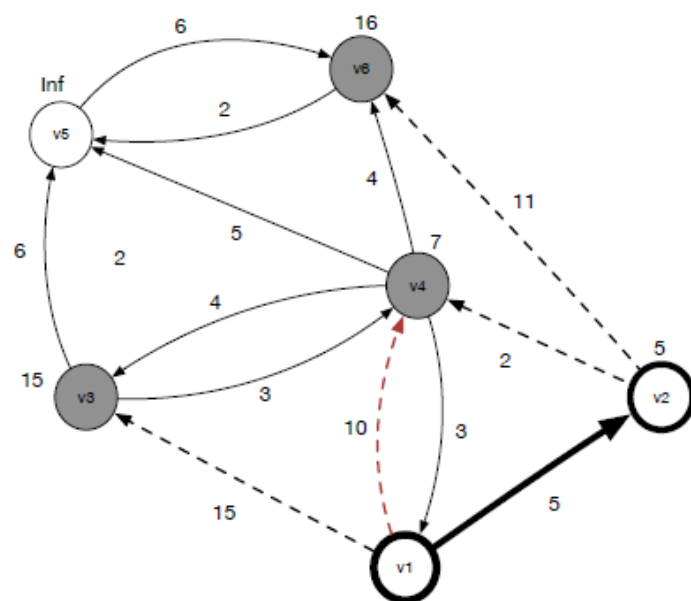


Рис. 9.37.

Далее, выбран третий узел и все смежные вершины корректируют значения d (рис. 9.38).

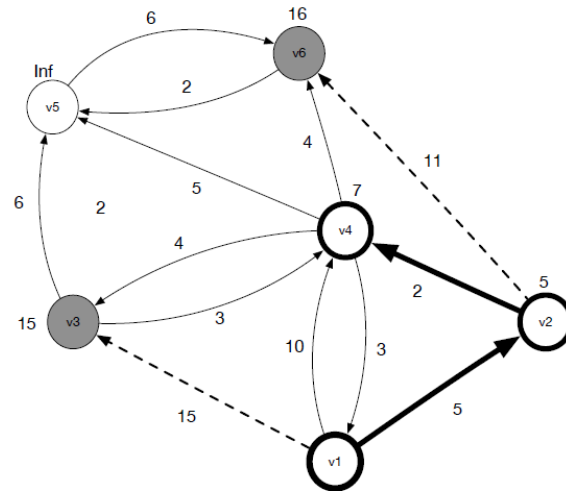


Рис. 9.38.

Следующим в накопитель отправляется узел v_5 . Операция релаксации заменяет $(1 \rightarrow 2 \rightarrow 6)$ на $(1 \rightarrow 2 \rightarrow 4 \rightarrow 6)$, а $(1 \rightarrow 3)$ на $(1 \rightarrow 2 \rightarrow 4 \rightarrow 3)$.

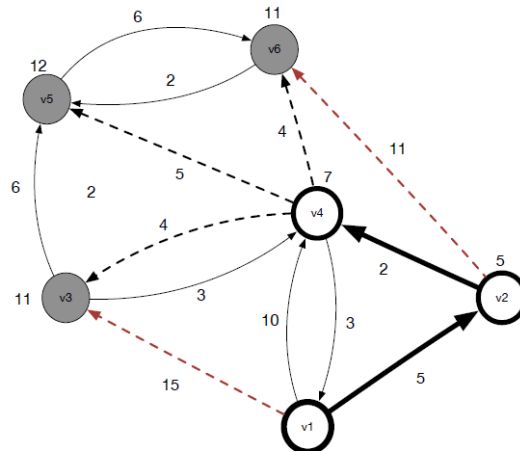


Рис. 9.39.

Итоговый результат представлен на рис. 9.40.

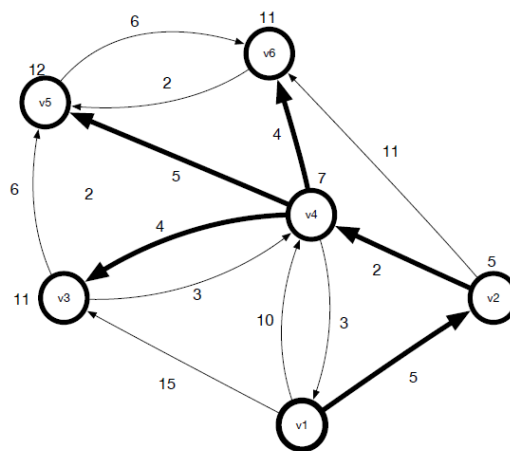


Рис. 9.40.

Для того, чтобы поместить во множество U все вершины, требуется сделать $|V| - 1$ шаг. На каждом шаге происходит корректировка расстояния до смежных вершин $O(|E_i|)$ и выбор минимального элемента из накопителя $O(\log |V|)$. Общая сложность алгоритма сильно зависит от структур данных, применяемых при работе с накопителем. Для изученных нами структур данных она составляет:

$$T = \sum_{i=1, |V|} |E_i| + |E| \times \log |V| = O(|E| \times \log |V|)$$

С помощью алгоритма Дейкстры можно найти расстояния от конкретной вершины до всех достижимых. Применив его для всех вершин, можно составить таблицу кратчайших расстояний между каждой парой вершин. Для насыщенного графа классический вариант алгоритма Дейкстры, который мы рассматривали, будет иметь сложность $O(|V|^2 \log |V|)$. Соответственно, построение таблицы всех кратчайших расстояний между парами будет иметь сложность $O(|V|^3 \log |V|)$. Много ли это? Оказывается, можно и быстрее, существует более быстрый алгоритм, имеющий сложность $O(|V|^3)$.



Продолжайте работу в созданном проекте. Реализуйте алгоритм Дейкстры для нахождения кратчайшего пути.

9.4.2 Алгоритм Флойда – Уоршалла

Этот алгоритм наряду с таблицей кратчайших расстояний между любыми парами вершин позволяет обнаруживать для каждой пары (u, v) ту вершину p , смежную с u , через которую проходит кратчайший путь из u в v .

Сам алгоритм известен с 1962 года. Для его исполнения наиболее удобно использовать представление графа матрицей смежности, в которой число, находящееся в i -й строке и j -м столбце есть вес связи между вершинами i и j . Для корректной работы алгоритма модифицируем матрицу C таким образом, что если вершина v не является соседней для u , то $C_{uv} = +\infty$. После исполнения алгоритма матрица D будет содержать в элементе C_{uv} вес кратчайшего пути из u

в v. Хорошим свойством алгоритма является то, что допустимы пути с отрицательным весом, а вот циклы с отрицательным весом недопустимы, но могут быть обнаружены

Сам алгоритм может быть описан в рекуррентной форме:

$$D_{ij}^{(k)} = \begin{cases} C_{ij}, & \text{если } k = 0 \\ \min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}), & \text{если } k \geq 1 \end{cases}$$

Это — задача динамического программирования, решаемая восходящим методом/ Код этого алгоритма исключительно прост.

```
void FloydWarshall(long [,] d, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int s = 0; s < n; s++)
        {
            for (int t = 0; t < n; t++)
            {
                if (s != t)
                { // Пропустим петлю
                    if (d[s,t] > d[s,i] + d[i,t])
                    {
                        d[s,t] = d[s,i] + d[i,t];
                    }
                }
            }
        }
    }
}
```

Нулевая итерация состоит в том, что заключительная матрица D заполняется элементами матрицы C. Инвариант алгоритма: после итерации n каждый элемент D_{ij} матрицы D содержит значение кратчайшего маршрута из i в j с промежуточными маршрутами, проходящими через вершину n. Это утверждение верно и после нулевой итерации, так как нумерация вершин в алгоритме начинается с единицы.

Продemonстрируем пример работы алгоритм Флойда-Уоршалла для графа, представленного на рис. 9.35 из предыдущего параграфа. Матрица D после нулевой итерации имеет вид, представленный на рис. 9.41.

$$D^{(0)} =$$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	10	∞	∞
V_2	∞	0	∞	2	∞	11
V_3	∞	∞	0	3	6	∞
V_4	3	∞	4	0	5	4
V_5	∞	∞	2	∞	0	6
V_6	∞	∞	∞	∞	5	0

Рис. 9.41.

Начальная матрица $D^{(0)}$ содержит метрики всех наилучших маршрутов единичной длины. Каждая следующая итерация алгоритма добавляет в матрицу $D^{(i)}$ элементы, связанные с маршрутами, проходящими через вершину i . После первой итерации произошла одна релаксация: элемент D_{42} получил значение 8. Жирным шрифтом помечены изменившиеся элементы таблицы (рис. 9.42).

$$D^{(1)} =$$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	10	∞	∞
V_2	∞	0	∞	2	∞	11
V_3	∞	∞	0	3	6	∞
V_4	3	8	4	0	5	4
V_5	∞	∞	2	∞	0	6
V_6	∞	∞	∞	∞	5	0

Рис. 9.42.

Вторая итерация пробует провести операцию релаксации с использованием вершины V_2 в качестве промежуточной. После неё изменился элемент D_{14} . Он стал равен 7. И действительно, произошла релаксация: $(1 \rightarrow 4)$ заменено на $(1 \rightarrow 2 \rightarrow 4)$. Появился маршрут $(1 \rightarrow 2 \rightarrow 6)$ (рис. 9.43).

$$D^{(2)} =$$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	7	∞	16
V_2	∞	0	∞	2	∞	11
V_3	∞	∞	0	3	6	∞
V_4	3	8	4	0	5	4
V_5	∞	∞	2	∞	0	6
V_6	∞	∞	∞	∞	5	0

Рис. 9.43.

Третья итерация добавила маршруты ($1 \rightarrow 3 \rightarrow 5$), вес которого равен 21 и ($5 \rightarrow 3 \rightarrow 4$) с весом 5 (рис. 9.44).

$$D^{(3)} =$$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	7	21	16
V_2	∞	0	∞	2	∞	11
V_3	∞	∞	0	3	6	∞
V_4	3	8	4	0	5	4
V_5	∞	∞	2	5	0	6
V_6	∞	∞	∞	∞	5	0

Рис. 9.44.

Четвёртая итерация добавила много новых маршрутов, так как она проводила релаксацию через оживлённый перекрёсток v_4 (рис. 9.45).

$$D^{(4)} =$$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	7	12	16
V_2	5	0	6	2	7	11
V_3	6	11	0	3	6	7
V_4	3	8	4	0	5	4
V_5	8	13	2	5	0	6
V_6	∞	∞	∞	∞	5	0

Рис. 9.45.

Пятая, последняя итерация, завершает алгоритм. Заключительная матрица D содержит кратчайшие расстояния между вершинами (рис. 9.46). Если элемент D_{ij} матрицы равен бесконечности, то вершина j недостижима из вершины i .

$$D^{(4)} =$$

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	5	15	7	12	16
V_2	5	0	6	2	7	11
V_3	6	11	0	3	6	7
V_4	3	8	4	0	5	4
V_5	8	13	2	5	0	6
V_6	13	18	7	10	5	0

Рис. 9.46.

Для определения того, имеет ли граф отрицательный цикл, достаточно повторить алгоритм, используя в качестве начальной матрицы D матрицу, полученную после $|V| - 1$ итерации. Если в какой-то момент операция релаксации завершится успехом, то это даст нам два факта: граф содержит отрицательный цикл и вершины графа, участвующие в успешной операции релаксации, содержатся в этом цикле.



Продолжайте работу в созданном проекте. Реализуйте алгоритм Флойда-Уоршалла для нахождения кратчайшего пути. Сравните работу двух алгоритмов: Дейкстры и Флойда-Уоршалла для разных графов. Какой алгоритм в каких случаях эффективнее?

Контрольные вопросы

1. Что такое граф?
2. Когда вводится понятие инцидентности? Что может быть инцидентно?
3. Как рассчитывается степень вершины?
4. Какая вершина называется висячей, а какая изолированной?
5. Какой граф называется ориентированным?
6. Какой граф называется взвешенным?
7. Какой граф называется полным?
8. Какой граф называется псевдографом, а какой мультиграфом?
9. Что такое маршрут в графе? Какие бывают маршруты?
10. Когда говорят, что в графе есть цикл? Какие виды циклов вы можете назвать?
11. Что является подграфом исходного графа?
12. Как можно представить граф в памяти ЭВМ?
13. Как реализуется обход графа BFS?
14. В чем основная идея поиска в глубину?
15. Как реализуется топологическая сортировка?
16. Что такое компонент связности?

17. Какое дерево называется остовным?
18. Опишите алгоритм Прима для поиска минимального остовного дерева.
19. Опишите алгоритм Краскала для поиска минимального остовного дерева.
20. Что такое дерево кратчайших путей?
21. Как работает алгоритм Дейкстры?
22. Какова сложность алгоритма Дейкстры?
23. Как работает алгоритм Флойда-Уоршалла?
24. Что такое релаксация?
25. Какой из алгоритмов нахождения кратчайшего пути в графе является жадным, а какой реализован методом динамического программирования?

Задания для самостоятельного выполнения

Задание №1. Зелье

Ограничение по времени: 1 секунда. Ограничение по памяти: 256 мегабайт

Злой маг Крокобобр варит зелье. У него есть большая колба, которую можно ставить на огонь и две колбы поменьше, которые огня не выдержат. В большой колбе налита компонента зелья, которую нужно подогреть на огне, маленькие колбы пусты. Емкость большой колбы магу Крокобобру известна — N миллилитров, маленьких — M и K миллилитров. Смесь можно переливать из любой колбы в любую, если выполняется одно из условий: либо после переливания одна из колб становится пустой, либо одна из колб становится полной, частичные переливания недопустимы.

Требуется ровно L миллилитров смеси в большой колбе. Помогите Крокобобру определить, сколько переливаний он должен сделать для этого.

Формат входных данных:

$N\ M\ K\ L$

$1 \leq N, M, K, L \leq 2000$

Формат выходных данных:

Одно число, равное количеству необходимых переливаний или слово OOPS, если это невозможно.

Задание №2. Скутер

Ограничение по времени: 1 секунда. Ограничение по памяти: 256 мегабайта

У Еремея есть электросамокат и он хочет доехать от дома до института, затратив как можно меньше энергии. Весь город расположен на холмистой местности и разделен на квадраты. Для каждого перекрестка известна его высота в метрах над уровнем моря. Если ехать от перекрестка с большей высотой до смежного с ним перекрестка с меньшей высотой, то электроэнергию можно не тратить., а если наоборот, то расход энергии равен разнице высот между перекрестками.

Помогите ему спланировать маршрут, чтобы он затратил наименьшее возможное количество энергии от дома до института и вывести это количество. Дом находится в левом верхнем углу, институт – в правом нижнем.

Формат входных данных:

N M

H11 H12 ... H1M

H21 H22 ... H2M

... ..

HN1 HN2 ... HNM

$1 \leq N, M \leq 1000$

$0 \leq H_{ij} \leq 1000, H \in \mathbb{Z}$

Формат выходных данных:

MinConsumingEnergy

Задание №3. Кратчайшие пути

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

Взвешенный граф с N вершинами задан своими M ребрами E_i , возможно отрицательного веса. Требуется найти все кратчайшие пути от вершины S до остальных вершин. Если граф содержит отрицательные циклы, вывести IMPOSSIBLE. Если от вершины S до какой-либо из вершин нет маршрута, то в качестве длины маршрута вывести слово UNREACHABLE.

Вершины графа нумеруются, начиная с нуля.

$$3 \leq N \leq 800$$

$$1 \leq M \leq 30000$$

$$-1000 \leq W_i \leq 1000$$

Формат входных данных:

$N \ M \ S$

$S1 \ E1 \ W1$

$S2 \ E2 \ W2$

...

Формат выходных данных:

IMPOSSIBLE или $D1 \ D2 \ D3 \ \dots \ DN$

Где D_i может быть UNREACHABLE

Задание №4. Коврики

Ограничение по времени: 2 секунды. Ограничение по памяти: 64 мегабайта

Имеется прямоугольная область, состоящая из $6 \leq N \leq 1000$ на $6 \leq M \leq 1000$ одинаковых квадратных клеток. Часть клеток свободна, часть закрашена. Ковриком называется максимальное множество закрашенных клеток, имеющих общую границу. Коврики размером 1×1 тоже допустимы.

Требуется по заданной раскраске области определить количество находящихся на ней ковриков.

Формат входных данных:

В первой строке содержатся N и M – размеры области. В каждой из следующих N строк находится ровно M символов, которые могут быть точкой, если поле свободно, или плюсом, если поле закрашено.

Формат выходных данных:

Общее количество ковриков на поле.

Задание №5. Передающие станции

Ограничение по времени: 1 секунда. Ограничение по памяти: 256 мегабайт

Компания сотовой связи получила лицензии на установку вышек сотовой связи в одном из новых районов города. На каждую вышку нужно установить приемо-передатчик, который позволит связаться с другими вышками. Для снижения общей стоимости проекта приемо-передатчика решено закупить оптом с большой скидкой, поэтому на всех вышках они будут однотипные. Выпускается много типов передатчиков, каждый из которых имеет различную мощность, причем, чем больше мощность, тем он дороже и имеет больший радиус охватываемой территории. Наша задача — подобрать модель передатчика с наименьшей возможной мощностью, исходя из расположения вышек для того, чтобы можно было передавать сообщения между любыми вышками, возможно, ретранслируя их.

Формат входных данных:

Первая строка содержит количество вышек $2 \leq N \leq 2000$

В остальных N строках – пары координат вышек в Декартовой системе - $10000 \leq X_i, Y_i \leq 10000$.

Формат выходных данных:

Одно число – наименьший требуемый радиус, достаточный для функционирования всей системы с точностью до 4-х знаков после запятой.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Алгоритмы. Построение и анализ*: [учебник] / Томас Кормен [и др.]; [пер. с англ. И. В. Красикова, Н. А. Ореховой, В. Н. Романова; под ред. И. В. Красикова]. - 2-е изд. - Москва : Вильямс, 2011. - 1290 с.
2. *Белов, В. В.* Алгоритмы и структуры данных: учебник для студентов вузов, обучающихся по направлению подготовки 2.09.03.04 "Программная инженерия" (квалификация - Бакалавр) / В. В. Белов, В. И. Чистякова. - Москва : КУРС: ИНФРА-М, 2019. - 240 с.
3. *Вирт, Н.* Алгоритмы и структуры данных. Новая версия для Оберона: [учебник] / Никлаус Вирт ; пер. с англ. под ред. Ткачева Ф. В. - Москва : ДМК [Пресс], 2014. - 272 с.
4. *Дроздов, С. Н.* Структуры и алгоритмы обработки данных: Учебное пособие / С. Н. Дроздов ; М-во образования и науки Рос. Федерации, Юж. федер. ун-т, Инжен.-технол. акад. - Таганрог : Издательство ЮФУ, 2016. - 228 с.
5. *Колдаев, В. Д.* Структуры и алгоритмы обработки данных: учебное пособие для студентов вузов, обучающихся по специальностям 230105 "Программное обеспечение вычислительной техники и автоматизированных систем", 230101 "Вычислительные машины, комплексы, системы и сети", 080801 "Прикладная информатика в экономике" / В. Д. Колдаев. - Москва : РИОР: ИНФРА-М, 2014. - 296 с.
6. *Седжвик, Р.* Алгоритмы на Java: научное издание / Роберт Седжвик, Кевин Уэйн ; [пер. с англ. А. А. Моргунова; под ред. Ю. Н. Артеменко]. - 4-е изд. - Москва : Вильямс, 2017. - 843 с.