

К. Ю. Поляков

ПРОГРАММИРОВАНИЕ

ПРОФИЛЬНАЯ
ШКОЛА

Python

C++

```
es = timeSec / 60;  
onds = timeSec % 60;  
< timeSec << " sec = "  
< minutes << " min " << s  
< endl;  
0 / 4 << 4;
```



ИЗДАТЕЛЬСТВО
БИНОМ

Часть 3

```
num2 = input("n2 = ")  
num1 + num2  
numma = num1 + num2  
print("Сумма = ", numma)  
print("Первое число = ", num1)  
print("Второе число = ", num2)
```

К. Ю. Поляков. Программирование. Python. C++. Часть 1

К. Ю. Поляков. Программирование. Python. C++. Часть 2

К. Ю. Поляков. Программирование. Python. C++. Часть 3

К. Ю. Поляков. Программирование. Python. C++. Часть 4

12+



ISBN 978-5-9963-4136-8



9 785996 341368



К. Ю. Поляков

ПРОГРАММИРОВАНИЕ
Python
C++

Часть 3

**Учебное пособие
для общеобразовательных
организаций**



Москва
БИНОМ. Лаборатория знаний
2019

УДК 004.9
ББК 32.97
П54

Поляков К. Ю.
П54 Программирование. Python. C++. Часть 3 : учебное пособие / К. Ю. Поляков. — М. : БИНОМ. Лаборатория знаний, 2019. — 208 с. : ил.

ISBN 978-5-9963-4136-8

Книга представляет собой третью часть серии учебных пособий по программированию. В отличие от большинства аналогичных изданий, в ней представлены два языка программирования высокого уровня — Python и C++.

Пособие посвящено способам организации данных и алгоритмам их обработки. Рассмотрены различные методы сортировки массивов, сравнивается их эффективность. Изучается работа с текстовыми и двоичными файлами. Приведены примеры использования структур данных — словарей, стеков, очередей, деревьев, графов — при решении практических задач. Рассматривается метод динамического программирования и его применение в задачах комбинаторики, оптимизации и теории игр.

После каждого параграфа приводится большое число заданий для самостоятельного выполнения разной сложности и вариантов проектных работ.

Пособие предназначено для учащихся средней школы.

УДК 004.9
ББК 32.97

Учебное издание
Поляков Константин Юрьевич

ПРОГРАММИРОВАНИЕ
Python. C++
Часть 3

Учебное пособие

Ведущий редактор *O. A. Полежаева*
Концепция внешнего оформления *B. A. Андрианов*
Художественный редактор *H. A. Новак*
Технический редактор *E. B. Денюкова*
Корректор *E. H. Клитина*
Компьютерная верстка: *B. A. Носенко*

(12+)

В оформлении учебника использована фотография с сайта en.wikipedia.org.

Подписано в печать 06.09.2018. Формат 84x108/16. Усл. печ. л. 21,84
Тираж 3000 экз. Заказ № м7022.

ООО «БИНОМ. Лаборатория знаний»
127473, Москва, ул. Краснопролетарская, д. 16, стр. 3,
тел. (495)181-53-44, e-mail: binom@Lbz.ru
<http://Lbz.ru>, <http://metodist.Lbz.ru>

Отпечатано в филиале «Смоленский полиграфический комбинат»
ОАО «Издательство «Высшая школа». 214020, Смоленск, ул. Смольянинова, 1.
Тел.: +7(4812) 31-11-96. Факс: +7(4812) 31-31-70.
E-mail: spk@smolpk.ru <http://www.smolpk.ru>

ПРЕДИСЛОВИЕ

Это третья часть пособия по программированию на языках Python и C++. Основное внимание в этой книге уделено структурам данных, т. е. тому, как лучше организовать данные, для того чтобы с ними было удобно работать.

Мы продолжаем изучать составной тип данных, который рассматривался во второй части пособия, — массив. Сортировка — одна из самых нужных задач обработки массивов. Простые методы сортировки, к сожалению, медленно работают для массивов большого размера. Мы познакомимся и с быстрыми методами, которые применяют в реальных проектах; программировать их немного сложнее.

Многие программы работают с данными, хранящимися в долговременной памяти в виде файлов. Вы узнаете, как прочитать данные из файла и как записать в файл результаты работы программы.

Новые структуры данных — словари, стеки, очереди — определяют некоторые наборы стандартных операций с данными. С их помощью многие задачи, которые кажутся сложными, получают простое и красивое решение.

Деревья предназначены для хранения многоуровневых (иерархических) структур данных. Представление данных в виде дерева со специальными свойствами помогает ускорить поиск, поэтому деревья повсеместно применяются в базах данных.

Для описания сетей, в которых многочисленные узлы связаны друг с другом, используют графы. С помощью алгоритмов теории графов решаются задачи выбора оптимального маршрута, в том числе для пересылки данных в Интернете.

Использование дополнительной памяти для хранения промежуточных данных часто позволяет ускорить решение сложной задачи. На этой идеи строится метод динамического программирования, который первоначально был разработан Р. Беллманом для решения многошаговых задач оптимизации. В пособии показано, как применять динамическое программирование для обработки данных.

Ещё одна интересная тема — теория игр. Мы займёмся вопросами стратегии: выясним, какие позиции — выигрышные, а какие — проигрышные. Это позволит написать программу, которая безошибочно играет в любую игру с полной информацией (т. е. в такую, в которой

нет неопределённости). Правда, для таких игр, как шахматы или го, программа, перебирающая все возможные варианты, будет работать намного дольше человеческой жизни. Так что в этой области остались нерешённые задачи, которые, возможно, предстоит решить вам.

Дополнительные материалы к пособию, в том числе файлы с программами, можно загрузить с сайта автора:

<http://kpolyakov.spb.ru/school/русср.htm>.

Автор благодарит своих коллег за полезные критические замечания и предложения, которые позволили устраниТЬ неточности и улучшить содержание пособия:

- *Д. В. Богданова*, старшего преподавателя кафедры бизнес-информатики МЭИ;
- *М. Д. Полежаеву*, разработчика веб-сайтов.

Глава 1

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON

§ 1

Простые алгоритмы сортировки

Ключевые слова:

- сортировка
- метод пузырька
- метод выбора
- перестановка элементов
- минимальный элемент
- сложность алгоритмов

Что такое сортировка?

Сортировка — это перестановка элементов массива (списка) в заданном порядке.



Данные, которые сортируются, могут размещаться как в оперативной, так и в долговременной памяти (на жёстких дисках или флэшнакопителях).

Порядок сортировки может быть любым; числа обычно сортируют по возрастанию или убыванию значений, а символьные строки — по алфавиту.

Для массивов, в которых есть одинаковые элементы, используются понятия «сортировка по неубыванию» и «сортировка по невозрастанию».

Возникает естественный вопрос: «Зачем сортировать данные?». На него легко ответить, вспомнив, например, работу со словарями: сортировка слов по алфавиту ускоряет поиск нужного слова.

Программисты изобрели множество способов сортировки. В целом их можно разделить на две группы: 1) простые, но медленно работающие на больших массивах и 2) сложные, но быстрые. В этом параграфе мы изучим два классических метода из первой группы.

Метод пузырька (сортировка обменами)

Название этого метода (по-английски — *bubble sort*) произошло от известного физического явления — пузырьёк воздуха в воде поднимается вверх. Если говорить о сортировке массива, то в первую очередь поднимается «наверх» (к началу массива) самый «лёгкий» (минимальный) элемент, затем следующий и т. д.

Сначала сравниваем последний элемент с предпоследним. Если они расположены неправильно (меньший элемент стоит «ниже»), то меняем их местами. Затем так же рассматриваем следующую пару элементов и т. д. (рис. 1.1).

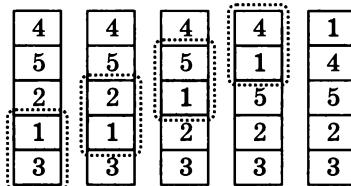


Рис. 1.1

Когда мы, пройдя весь массив снизу вверх, обработаем пару ($A[0]$, $A[1]$), минимальный элемент встанет на место $A[0]$. Это значит, что на следующих этапах перемещать его не нужно.

Пусть массив состоит из N элементов. Первый цикл, устанавливающий на своё место первый (минимальный) элемент $A[0]$, можно на псевдокоде записать так:

```
# для j от N-2 до 0 шаг -1
if A[j] > A[j+1]:
    # поменять местами A[j] и A[j+1]
```

Заголовок цикла тоже написан на псевдокоде. Обратите внимание, что на очередном шаге сравниваются элементы $A[j]$ и $A[j+1]$, поэтому цикл начинается с $j = N - 2$. Если начать с $j = N - 1$, то на первом же шаге произойдёт выход за границы массива — обращение к несуществующему элементу $A[N]$.

Поскольку цикл `for in range(...)` в Python «не доходит» до заданного конечного значения, мы принимаем это значение равным -1 , а не 0 :

```
for j in range(N-2, -1, -1):
    if A[j] > A[j+1]:
        # поменять местами A[j] и A[j+1]
```

Здесь в записи `range(N-2, -1, -1)` первое число -1 — это ограничитель (число, следующее за конечным значением), а второе число -1 — это шаг изменения переменной цикла.

За один проход такой цикл ставит на своё место один элемент ($A[0]$). Чтобы «подтянуть» второй элемент, нужно написать ещё один почти такой же цикл, который будет отличаться только конечным значением переменной j в заголовке цикла. Так как «верхний» элемент уже стоит на месте, его не нужно трогать:

```
for j in range(N-2, 0, -1):
    if A[j] > A[j+1]:
        # поменять местами A[j] и A[j+1]
```

При установке следующего (третьего по счёту) элемента конечное значение при вызове функции `range` будет равно 1 и т. д.

Для полной сортировки массива нужно сделать $N-1$ проход по массиву. Почему не N ? Дело в том, что в результате каждого прохода на своё место устанавливается один элемент. Если $N-1$ элементов поставлены на свои места, то оставшийся автоматически встаёт на своё место — другого места для него нет. Поэтому полный алгоритм сортировки представляет собой такой вложенный цикл:

```
for i in range(N-1):
    for j in range(N-2, i-1, -1):
        if A[j] > A[j+1]:
            A[j], A[j+1] = A[j+1], A[j]
```

Составить полную программу вы можете самостоятельно.

Поскольку в этом алгоритме кроме сравнения используется только обмен значениями двух элементов массива, метод пузырька часто называют сортировкой простыми обменами.

Часто используют ещё один вариант этого метода, который можно назвать «методом камня» — самый «тяжёлый» элемент опускается в конец («на дно») массива.

```
for i in range(N-1):
    for j in range(N-1-i):
        if A[j] > A[j+1]:
            A[j], A[j+1] = A[j+1], A[j]
```

Давайте попробуем оценить сложность этого алгоритма. Подсчитаем, сколько раз выполнится условный оператор в теле цикла. В первый раз, при $i = 0$, внутренний цикл (по переменной j) выполняется $N-1$ раз, при $i = 1$ уже $N-2$ раза, а при $i = N-2$ — только 1 раз. Общее количество проверок:

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

По формуле суммы первых членов арифметической прогрессии получаем:

$$T(N) = \frac{1 + (N - 1)}{2}(N - 1) = \frac{1}{2}N^2 + \frac{1}{2}N.$$

В этой формуле старшая степень — у слагаемого N^2 , поэтому асимптотическая сложность этого алгоритма — $O(N^2)$, это алгоритм квадратичной сложности. Если размер массива увеличится в 10 раз, то количество операций (и время работы алгоритма) увеличится примерно в 100 раз.

Для больших массивов этот алгоритм сортировки будет работать очень долго. Компьютер, выполняющий 1 миллион итераций внутреннего цикла в секунду, отсортирует массив из 1000 элементов примерно за 0,5 секунды, массив из 100 000 элементов — за 1 час 23 минуты, а на обработку массива из 1 000 000 элементов потребуется около 6 суток.

Метод выбора

Ещё один популярный простой метод сортировки — метод выбора, при котором на каждом этапе выбирается минимальный элемент (из оставшихся) и ставится на своё место. Алгоритм в общем виде можно записать так:

```
for i in range(N-1):
    # найти номер nMin минимального
    # элемента среди A[i]..A[N-1]
    if nMin != i:
        # поменять местами A[i] и A[nMin]
```

Здесь перестановка происходит только тогда, когда $nMin$ не равно i , т. е. найденный минимальный элемент стоит не на своём месте.

Поскольку поиск номера минимального элемента выполняется в цикле, этот алгоритм сортировки также представляет собой вложенный цикл:

```
for i in range(N-1):
    nMin = i
    for j in range(i+1, N):
        if A[j] < A[nMin]: nMin = j
    if nMin != i:
        A[i], A[nMin] = A[nMin], A[i]
```

Фоном выделен поиск номера минимального элемента в той части массива, которая начинается с элемента $A[i]$.

Оценив количество операций при выполнении этого алгоритма (сделайте это самостоятельно), можно показать, что его асимптотическая сложность — $O(N^2)$, такая же, как у метода пузырька. Поэтому для больших массивов он также будет работать очень долго.

Выводы

- Сортировка — это расстановка элементов массива (списка) в заданном порядке.
- Цель сортировки — облегчить последующий поиск.
- Метод пузырька состоит в том, что наименьший элемент последовательными обменами продвигается к началу массива.
- Метод выбора состоит в том, что в массиве ищется минимальный элемент и ставится на первое место. Затем этот алгоритм повторяется для оставшейся части массива.
- Метод пузырька и метод выбора минимального элемента имеют асимптотическую сложность $O(N^2)$, они работают медленно для больших массивов данных.

Вопросы и задания



- Объясните, почему в описанных методах сортировки используется вложенный цикл, а не одинарный.
- Сравните метод пузырька и метод выбора. Какой из них требует меньшего числа сравнений и перестановок элементов?
- Как нужно изменить приведённые алгоритмы, чтобы элементы массива были отсортированы по убыванию?
- Константин предложил использовать для сортировки массива такой вложенный цикл:

```
for i in range(N-1):
    for j in range(N-1):
        if A[j] < A[j+1]:
            A[j], A[j+1] = A[j+1], A[j]
```

Как именно будут отсортированы элементы? Будет ли правильно выполнена сортировка?

- Семён сортирует массив из 10 элементов методом пузырька. По ошибке он записал внешний цикл так:

```
for i in range(3):
    ...
```

Что получится в результате такой обработки массива?

- В массив А записаны 10 натуральных чисел:

```
A = [6, 2, 8, 4, 9, 7, 3, 1, 10, 5]
```

Определите, какие значения окажутся в переменных с и d после выполнения фрагмента программы.

a) N = 10

c = d = 0

```
for i in range(N-1):
    if A[i] > A[i+1]:
        c += 1
        A[i], A[i+1] = A[i+1], A[i]
    else:
        d += 1
```

б) N = 10

c = d = 0

```
for i in range(N-1, 0, -1):
    if A[i-1] > A[i]:
        c += 1
        A[i], A[i-1] = A[i-1], A[i]
    else:
        d += 1
```

```
b) N = 10
c = d = 0
for i in range(N-1):
    if A[i] > A[N-1]:
        c += 1
        A[i], A[N-1] = A[N-1], A[i]
    else:
        d += 1
```

- *7. Используя информацию из дополнительных источников и модуль time, выясните зависимость времени выполнения сортировки от размера массива для рассмотренных методов. Постройте эти зависимости с помощью электронных таблиц.
- 8. Напишите программу, которая сортирует массив в порядке невозрастания и находит количество различных чисел в нём.
- 9. Напишите программу, которая выполняет неполную сортировку массива: ставит в начало массива четыре самых больших по величине элемента в порядке невозрастания. Положение остальных элементов не важно.
- 10. Напишите вариант метода пузырька, который заканчивает работу, если на очередной итерации внешнего цикла не было перестановок (такой алгоритм называется «сортировкой с флагжком»).
- *11. Напишите рекурсивную функцию, которая сортирует массив одним из рассмотренных методов.
- *12. Напишите программу, которая сортирует массив по возрастанию первой цифры числа.
- 13. Напишите программу, которая сортирует массив по убыванию суммы цифр числа.
- 14. Напишите программу, которая сортирует первую половину массива по возрастанию, а вторую — по убыванию (элементы из первой половины не должны попадать во вторую и наоборот). Считайте, что в массиве чётное число элементов.
- 15. Напишите программу, которая сортирует массив, а затем находит максимальное из чисел, встречающихся в массиве несколько раз (если такие есть).
- *16. Напишите программу, которая сравнивает количество перестановок при сортировке одного и того же массива разными методами. Проведите эксперименты для возрастающей последовательности (уже отсортированной), убывающей (отсортированной в обратном порядке) и случайной.

Интересные сайты

informatics.mccme.ru — сайт для подготовки к олимпиадам по информатике с автоматической проверкой решений

algolist.manual.ru — сайт «Алгоритмы, методы, исходники»

§ 2**Быстрые алгоритмы сортировки****Ключевые слова:**

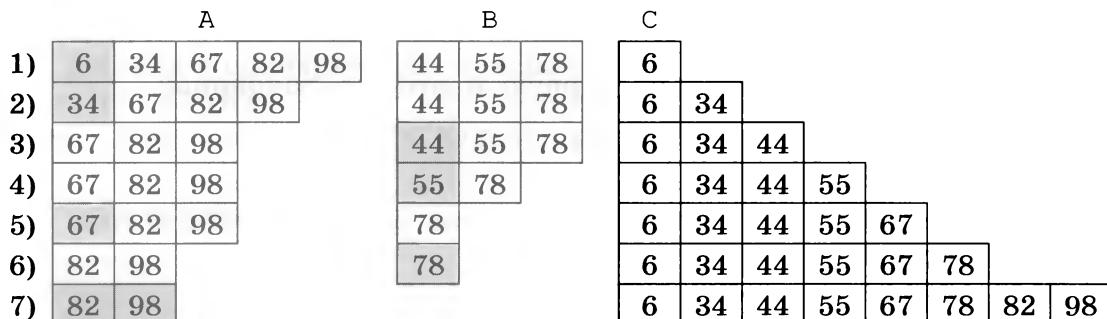
- сортировка слиянием
- быстрая сортировка
- лямбда-функция

Сортировка слиянием

Простые методы сортировки, с которыми вы уже знакомы, работают медленно для больших массивов данных, которые часто встречаются в практических задачах. Поэтому в середине XX века математики и программисты приложили много усилий для разработки более эффективных алгоритмов сортировки.

В 1945 году Джон фон Нейман предложил алгоритм сортировки, основанный на процедуре слияния двух заранее отсортированных массивов.

Предположим, что есть два отсортированных массива А и В, размеры которых, соответственно, N_a и N_b , и мы хотим объединить их элементы в новый отсортированный массив С размером $N_a + N_b$. Для этого будем на каждом шаге сравнивать два первых (ещё не использованных) элемента массивов А и В, добавляя в конец нового массива С наименьший из них. Так продолжается до тех пор, пока один из массивов не закончится, тогда оставшийся «хвост» второго массива просто добавляется в конец массива С (он ведь уже был отсортирован!). Пример такого слияния показан на рис. 1.2.

**Рис. 1.2**

Фоном выделены элементы, которые включаются в массив С на очередном шаге. Сначала в массив С добавляются два первых элемента массива А (шаги 1–2), потому что они меньше, чем первый элемент массива В (44). На следующих двух шагах в массив С добавляются два первых элемента массива В, потому что они меньше, чем первый из оставшихся элементов массива А (67), и т. д. На шаге 7 в конец массива С дописывается «хвост» массива А (весь массив В уже добавлен в массив С).

Первая часть процедуры слияния на языке Python может быть записана так:

```
C = []
while A and B:
    if A[0] <= B[0]:
        C.append( A.pop(0) )
    else:
        C.append( B.pop(0) )
```

Сначала массива С пуст. Основной цикл выполняется, пока оба массива, А и В, непустые: условие **A and B** в языке Python равносильно такому условию:

```
len(A) > 0 and len(B) > 0
```

На каждом шаге цикла с помощью метода append в конец массива С добавляется очередной элемент массива А или очередной элемент массива В (в зависимости от того, который из них меньше). Запись **A.pop(0)** — это вызов метода pop, который выполняет две операции: 1) возвращает элемент массива А[0]; 2) удаляет из массива А элемент с индексом 0.

Теперь остаётся добавить в массив С оставшиеся части массивов А и В (одна из них — пустая):

```
C = C + A + B
```

На основе этой идеи был разработан быстрый алгоритм *сортировки слиянием* (англ. *merge sort*). Пусть у нас есть массив А и требуется отсортировать его по возрастанию. Разделим массив на две равные части. Для этого вычислим индекс элемента, стоящего в середине массива:

```
mid = len( A ) // 2
```

Теперь отсортируем отдельно первую и вторую половины:

```
L = mergeSort( A[:mid] )
R = mergeSort( A[mid:] )
```

и выполним слияние этих половин, вызвав функцию **merge** (мы скоро напишем её полностью):

```
C = merge( L, R )
```

Эта функция объединит уже отсортированные массивы L и R и построит новый массив С той же длины, что и исходный массив А. Получается такая функция сортировки:

```
def mergeSort( A ):
    if len( A ) <= 1: return A
    mid = len( A ) // 2
    L = mergeSort( A[:mid] )
    R = mergeSort( A[mid:] )
    return merge( L, R )
```

Процедура `mergeSort` дважды вызывает сама себя, т. е. является рекурсивной. Стока

```
if len(A) <= 1: return A
```

останавливает рекурсию. Если массив пустой или в нём всего один элемент, то сортировать нечего, нужно выйти из функции и вернуть исходный массив.

Осталось написать функцию `merge`, которая сливает две отсортированные части. Она фактически повторяет алгоритм слияния, приведённый выше:

```
def merge(A, B):
    C = []
    while A and B:
        if A[0] <= B[0]:
            C.append(A.pop(0))
        else:
            C.append(B.pop(0))
    return C + A + B
```

Сортировка слиянием — это пример применения подхода «разделяй и властвуй» (англ. *divide and conquer*) при решении сложных задач:

- 1) задача разбивается на несколько подзадач;
- 2) эти подзадачи решаются с помощью рекурсивных вызовов того же (или другого) алгоритма;
- 3) решения подзадач объединяются, и получается решение исходной задачи.

Сортировка слиянием применима для данных с последовательным доступом, например записанных в файлы. Этот алгоритм можно использовать в параллельных вычислительных системах, где несколько процессоров работают одновременно и используют общую память. Его главный недостаток состоит в том, что нужен дополнительный массив того же размера, что и исходный (в нашей программе это массив `C`).

Быстрая сортировка



Ч. Э. Хоар
(р. 1934)

Один из самых популярных «быстрых» алгоритмов, разработанный в 1960 году английским учёным Ч. Хоаром, так и называется — быстрая сортировка (англ. *quick sort*).

Пусть дан массив `A` из N элементов. Выберем сначала наугад любой элемент массива (назовем его `X`). Затем разделим массив на три части: в первую включим все элементы, меньшие `X`, во вторую — все элементы, равные `X` (их может быть несколько), а в третью — все элементы, большие `X`.

Понятно, что в отсортированном массиве все элементы первой части будут стоять слева от элементов второй части, а элементы третьей части — правее всех. Таким образом, остаётся отсортировать (рекурсивно!) первую и третью части, а потом соединить все три массива в один. Это тоже подход «разделяй и властвуй», с которым мы познакомились на примере сортировки слиянием.

Проще всего оформить алгоритм быстрой сортировки в виде функции, которая возвращает новый, отсортированный массив:

```
import random
def qSort( A ):
    if len( A ) <= 1: return A # (1)
    X = random.choice( A ) # (2)
    BL = [b for b in A if b < X] # (3)
    BX = [b for b in A if b == X] # (4)
    BR = [b for b in A if b > X] # (5)
    return qSort( BL ) + BX + qSort( BR ) # (6)
```

Если массив пустой или в нём всего один элемент, то результат сортировки — это сам массив, сортировать тут нечего (строка 1). Иначе выбираем в качестве разделителя X («стержневого элемента», англ. *pivot*) случайный элемент массива (строка 2); для этого используется функция `choice` (по-английски — выбор) из модуля `random`.

В строках 3–5 с помощью генераторов списков создаём три вспомогательных массива: в массив `BL` войдут все элементы, меньшие X , в массив `BX` — все элементы, равные X , а в массив `BR` — все элементы, большие X . Теперь остаётся отсортировать списки `BL` и `BR` (вызвав функцию `qSort` рекурсивно) и построить окончательный список-результат, который «складывается» из отсортированного списка `BL`, списка `BX` и отсортированного списка `BR` (строка 6).

Применим эту функцию для сортировки массива A :

```
Asorted = qSort( A )
```

Обратите внимание, что функция возвращает новый массив, а исходный массив A при этом не изменяется.

Недостаток этого метода в том, что он расходует много памяти — при каждом рекурсивном вызове строятся вспомогательные массивы `BL`, `BX` и `BR`. При изучении главы по языку C++ мы рассмотрим ещё один вариант быстрой сортировки, при котором дополнительная память не нужна.

Скорость работы быстрой сортировки зависит от того, насколько удачно выбирается вспомогательный элемент X . Лучше всего выбирать X так, чтобы в обеих частях было равное количество элементов. Такое значение X называется *медианой* массива. Однако найти медиану тоже очень непросто. Легче всего в качестве X выбрать любой элемент массива, например первый или средний или, как в нашей функции, случайный.

Худший случай — когда в одной части оказывается только один элемент, а в другой — все остальные. При этом глубина рекурсии достигает N , что может привести к переполнению *стека* — специальной области памяти, где хранятся временные данные при вызовах подпрограмм (подробнее мы изучим эту тему в § 8). Для того чтобы уменьшить вероятность худшего случая, в алгоритм вводят случайность: в качестве X на каждом шаге выбирают случайный элемент массива (именно это мы сделали в программе).

В следующей таблице сравниваются время сортировки (в секундах) массивов разного размера, заполненных случайными значениями, с использованием изученных алгоритмов.

N	Метод пузырька	Метод выбора	Сортировка слиянием	Быстрая сортировка
1000	0,08	0,05	0,006	0,002
5000	1,80	1,30	0,033	0,006
15000	17,30	11,20	0,108	0,019

Как показывают эти данные, преимущество быстрой сортировки становится значительным при увеличении N .

Сортировка в языке Python

В языке Python есть встроенная функция для быстрой сортировки массивов (списков), которая называется `sorted`. Она использует алгоритм Timsort¹⁾. Вот так можно построить новый массив B , который совпадает с отсортированным в порядке возрастания массивом A :

```
B = sorted( A )
```

По умолчанию выполняется сортировка по возрастанию (неубыванию). Для того чтобы отсортировать массив по убыванию (невозрастанию), нужно передать функции дополнительный аргумент с именем `reverse` (по-английски — обратный), равный `True`:

```
B = sorted( A, reverse = True )
```

Иногда требуется особый, нестандартный порядок сортировки, который отличается от сортировок по возрастанию и по убыванию. В этом случае необходимо определить ещё один именованный аргумент функции `sorted`, который называется `key` (по-английски — ключ). Этому аргументу присваивается ссылка на объект-функцию, которая возвращает число (или символ), используемое при сравнении двух элементов массива.

Предположим, что нам нужно отсортировать числа по возрастанию последней цифры (поставить сначала все числа, оканчивающиеся на

¹⁾ <http://ru.wikipedia.org/wiki/Timsort>

0, затем — все, оканчивающиеся на 1, и т. д.). В этом случае ключ сортировки — это последняя цифра числа. Напишем функцию, которая возвращает эту последнюю цифру:

```
def lastDigit( n ):
    return n % 10
```

Теперь сортировку массива A по возрастанию последней цифры можно выполнить так:

```
B = sorted( A, key = lastDigit )
```

Функция `lastDigit` получилась очень короткая, и часто не хочется создавать лишнюю функцию с помощью оператора `def`, особенно тогда, когда она нужна всего один раз. В таких случаях удобно использовать функции без имени — так называемые *лямбда-функции*. Они записываются прямо при вызове функции в параметре `key`:

```
B = sorted( A, key = lambda x: x % 10 )
```

Здесь фоном выделена лямбда-функция, которая для переданного ей значения `x` возвращает результат `x % 10`, т. е. последнюю цифру десятичной записи числа.

Функция `sorted` не изменяет исходный массив и возвращает его отсортированную копию. Если нужно отсортировать массив «на месте» (не выделяя дополнительную память), лучше использовать метод `sort` для списка:

```
A.sort( key = lambda x: x % 10, reverse = True )
```

Он имеет те же именованные аргументы, что и функция `sorted`, но изменяет исходный список. В приведённом примере элементы массива A будут отсортированы по убыванию последней цифры.

Выводы

- Метод сортировки слиянием основан на алгоритме слияния двух отсортированных половин массива.
- В методе быстрой сортировки выбирается произвольный элемент массива X, и массив делится на три части: элементы, меньшие X, равные X и большие X. Первая и последняя части далее сортируются с помощью того же алгоритма.
- Методы сортировки слиянием и быстрой сортировки используют рекурсию.
- В языке Python встроенный метод быстрой сортировки применяется в функции `sorted` и методе `sort` для списков.
- Лямбда-функция — это безымянный объект-функция, который передаётся в функцию или процедуру.

Вопросы и задания



1. Подумайте, как можно использовать метод сортировки слиянием для сортировки данных в файле, если этих данных настолько много, что они не помещаются в оперативную память.
2. Как вы думаете, можно ли использовать метод быстрой сортировки для нечисловых данных, например для символьных строк?
3. От чего зависит скорость быстрой сортировки? Какой случай самый лучший и какой — самый худший?
4. Как вы думаете, может ли метод быстрой сортировки работать дольше, чем метод выбора (или другой «простой» метод)? Если да, то при каких условиях?
5. Как нужно изменить приведённые в параграфе алгоритмы, чтобы элементы массива были отсортированы по убыванию?
6. Чем различаются встроенные средства сортировки массивов в Python: функция `sorted` и метод `sort` для списков?
7. В массиве S находятся символьные строки. Как отсортировать их по убыванию длины, используя функцию `sorted`?
8. Что такое лямбда-функции? Когда их удобно использовать?
- *9. Используя информацию из дополнительных источников и модуль `time`, попробуйте выяснить зависимость времени выполнения сортировки от размера массива для разных методов. Постройте эти зависимости с помощью электронных таблиц.

§ 3 Двоичный поиск

Ключевые слова:

- линейный поиск
- двоичный поиск

Поиск в массиве

Во второй части пособия мы уже рассматривали задачу поиска элемента в массиве. Напомним некоторые результаты.

Пусть массив A состоит из N элементов и требуется найти номер элемента, равного X . Алгоритм поиска сводится к просмотру всех элементов массива до тех пор, пока не будет найден нужный элемент или не закончится массив:

```
i = 0
while i < N and A[i] != X:
    i += 1
if i < N:
    print( "A[{}]={}".format(i, X) )
else:
    print( "Не нашли!" )
```

Такой поиск называют *линейным*. Для массива из 1000 элементов в худшем случае нужно выполнить 1000 итераций цикла, чтобы убедиться, что заданного элемента в массиве нет. Если число элементов (например, записей в базе данных) очень велико, время поиска может оказаться недопустимо большим, и пользователь просто не дождётся ответа.

Что такое двоичный поиск?

Рассмотрим игру «Угадай число»: программа «задумывает» (с помощью датчика случайных чисел) число от 1 до 100, а человек пытается его отгадать, используя подсказки компьютера. После ввода своей догадки человек получает одно из следующих сообщений: «Слишком много!», «Слишком мало!», «Угадал!». Когда игрок угадывает число, игра заканчивается.

Мы надеемся, что такую игру вам уже легко написать самостоятельно — она содержит один цикл и условные операторы.

Лучшая стратегия в этой игре — деление оставшегося интервала пополам на каждом ходу, так чтобы область поиска каждый раз уменьшалась в два раза.

Если задумано число на отрезке [1; 100], в первый раз нужно ввести число-догадку 50. Если программа ответила «Слишком много!», то задуманное число находится на отрезке [1; 49], и нужно выбрать середину этого отрезка — 25. Если же программа отвечает «Слишком мало!», это значит, что число находится на отрезке [51; 100], и в следующий раз нужно выбрать середину этого отрезка — 75 или 76.

Такая стратегия называется *двоичным* или *бинарным* (от англ. *binary* — двоичный) поиском, потому что на каждом шаге область поиска уменьшается в два раза. В этой задаче бинарный поиск возможен потому, что мы ищем нужное число в ряду чисел, расположенных по возрастанию: 1, 2, 3, ..., 100, и компьютер, используя операцию сравнения, сообщает нам, в какой половине находится нужное число.

Двоичный поиск мы часто используем в жизни. Пусть, например, нужно найти в словаре слово «гравицапа». Сначала открываем словарь примерно в середине и смотрим, какие там слова. Если они начинаются на букву «Л», то слово «гравицапа» явно находится на предыдущих страницах, и вторую часть словаря можно не смотреть.

Затем проверяем страницу в середине первой половины и т. д. Понятно, что двоичный поиск можно применить только тогда, когда данные заранее отсортированы.

Двоичный поиск в массиве данных

Как вы помните, основная задача сортировки — облегчить последующий поиск данных. Пусть данные в массиве A уже отсортированы, и нужно найти в нём заданное значение X или установить, что его там нет. В этом случае можно успешно применить двоичный поиск.

На рисунке 1.3 показан двоичный поиск числа $X = 44$ в отсортированном массиве.

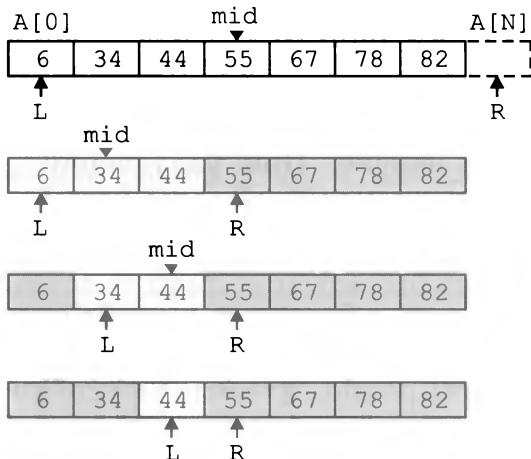


Рис. 1.3

Серым фоном выделены ячейки, которые уже не рассматриваются, потому что в них не может быть заданного числа. Переменные L и R ограничивают «рабочую зону» массива: L содержит номер первого элемента, а R — номер элемента, *следующего* за последним. Таким образом, нужный элемент (если он есть) находится в части массива, которая начинается с элемента A[L] и заканчивается элементом A[R-1]. Обратите внимание, что элемент A[R] не входит в область, где может находиться нужный элемент, он выступает как «барьер».

На каждом шаге вычисляем номер среднего элемента «рабочей зоны», он записывается в переменную mid. Если $X < A[mid]$, то значение X может находиться в массиве только левее A[mid], и правая граница R перемещается в mid. В противном случае нужный элемент находится правее середины или совпадает с A[mid]; при этом левая граница L перемещается в mid.

Поиск заканчивается при выполнении условия $L = R-1$, когда в рабочей зоне останется один элемент. Если при этом $A[L]=X$, то в результате найден элемент, равный X, иначе такого элемента нет.

На языке Python этот алгоритм запишется так:

```
L, R = 0, N          # начальный отрезок
while L < R-1:
    mid = (L+R) // 2      # нашли середину
    if X < A[mid]:        # сжатие отрезка
        R = mid
    else: L = mid

    if A[L] == X:
        print( "A[{}]={}" .format(L, X) )
    else:
        print( "Не нашли!" )
```

Какой алгоритм поиска лучше?

Двоичный поиск работает значительно быстрее, чем линейный. Для массива из 8 элементов удаётся решить задачу, выполнив 4 сравнения вместо 8 при линейном поиске. При увеличении размера массива эта разница становится впечатляющей. В следующей таблице приводится наибольшее количество сравнений для двух методов при разных значениях N .

<i>N</i>	Линейный поиск	Двоичный поиск
2	2	2
16	16	5
1024	1024	11
1 048 576	1 048 576	21

Однако при этом нельзя сказать, что двоичный поиск лучше линейного. Ведь для использования двоичного поиска данные необходимо предварительно отсортировать, а сортировка занимает значительно больше времени, чем сам поиск. Поэтому этот подход эффективен, если данные меняются (и сортируются) редко, а поиск выполняется часто. Такая ситуация характерна, например, для баз данных.

Двоичный поиск по ответу

Двоичный поиск можно использовать даже тогда, когда массива нет, но мы можем вычислить каждое значение, которое могло бы быть записано в такой массив. Поясним идею на примере.

Предположим, что Фёдор — работник издательства, и ему нужно вписать заданный текст в поле шириной W и высотой H , используя как можно более крупный шрифт.

Фёдор может сначала выбрать очень мелкий шрифт, чтобы набранный им текст гарантированно поместился в отведённое ему поле. Затем,

постепенно увеличивая размер шрифта на 1 пункт (размеры шрифтов измеряются в пунктах), пока текст помещается в поле, выбрать наибольший допустимый размер. Это будет линейный поиск (на практике это вполне работающий вариант). Но можно ускорить решение задачи.

Сначала выберем два размера шрифта, очень маленький размер L и очень большой R . Если использовать шрифт размера L , то текст помещается в поле, а если набрать его шрифтом размера R , то не помещается. Дальше будем использовать двоичный поиск. Вычислим среднее арифметическое значений L и R , обозначим его через mid . Если текст, набранный шрифтом размера mid , не помещается в поле, то нужный нам размер шрифта находится на отрезке $[L+1, mid-1]$, а если помещается, будем искать дальше на отрезке $[mid, R-1]$.

Это тоже алгоритм двоичного поиска. Дело в том, что ситуацию, когда текст помещается в поле, можно обозначить как `True` (или 1), а противоположный случай — как `False` (или 0). Тогда представим себе, что у нас есть массив таких логических значений для всех размеров шрифта от L до R . Причём он отсортирован в том смысле, что первые элементы массива равны `True`, а все следующие — `False`. Эти значения не могут чередоваться, т. е. после значения `False` не может стоять `True` — это означало бы, что текст, набранный более мелким шрифтом, *не* помещается в поле, а более крупный — помещается. Как раз для такого отсортированного массива можно использовать двоичный поиск.

Покажем, как запрограммировать такое решение. Напишем логическую функцию `fit`, которая определяет, вписывается ли текст в область заданного размера:

```
def fit( fontSize ):
    ... # определяем высоту текста height
        # в полосе шириной W
    return height <= H
```

Параметр этой функции `fontSize` — размер шрифта. Функция `fit` вычисляет и записывает в переменную `height` высоту текста, набранного таким шрифтом в полосе шириной W . Если рассчитанная высота не больше допустимой высоты H , функция возвращает значение `True`, иначе — значение `False`.

Пусть размер шрифта 10 позволяет вписать текст в поле, а размер 100 — не позволяет. Тогда двоичный поиск может быть записан так:

```
L, R = 10, 100          # начальный отрезок
while L < R-1:
    mid = (L+R) // 2      # нашли середину
    if not fit( mid ):
        R = mid           # сжатие отрезка
    else: L = mid
print( L )
```

После завершения работы цикла в переменной L находится ответ: максимальный размер шрифта, соответствующий условию.

Выводы

- Для отсортированного массива можно применить двоичный поиск, который работает значительно быстрее, чем линейный.
- При двоичном поиске на каждом шаге зона поиска уменьшается в два раза.
- Двоичный поиск выгодно использовать, когда данные меняются (и сортируются) редко, а поиск выполняется часто.



Вопросы и задания

1. Приведите примеры использования двоичного поиска в обычной жизни.
2. Как можно примерно подсчитать количество шагов при двоичном поиске?
3. Сравните достоинства и недостатки линейного и двоичного поиска.
- *4. Предложите, как можно усовершенствовать двоичный поиск, если известно, что распределение данных на отрезке близко к равномерному. Проведите эксперимент, который позволяет сравнить скорость классического двоичного поиска и вашего улучшенного варианта для разных наборов исходных данных.
5. Напишите программу, которая играет с человеком в игру «Угадай число». Программа загадывает число от 1 до 100, а человек пытается его отгадать.
6. Напишите программу, которая пытается за наименьшее число попыток отгадать число, задуманное человеком. Программа должна выводить сообщение об ошибке, если человек вводит неверный ответ или пытается жульничать.
7. Напишите программу, которая сортирует массив по убыванию и определяет, сколько в нём есть значений, равных заданному числу X . Программа должна работать быстрее, чем линейный поиск.
- *8. Напишите вариант алгоритма двоичного поиска, в котором элемент $A[R]$ входит в область поиска. *Подсказка:* при проверке условий в цикле нужно ещё рассмотреть вариант, когда элемент $A[mid]$ равен X .
9. В массиве, который отсортирован по убыванию, записаны площади ($\text{в } m^2$) всех свободных участков, пригодных для строительства. Фирма-застройщик хочет выкупить участок, площадь которого примерно равна $X \text{ } m^2$. Найдите участки, имеющие площадь, ближайшую к X : наибольший участок, площадь которого меньше X , и наименьший участок, площадь которого больше X . Программа должна работать быстрее, чем линейный поиск.

10. Составляя карту местности, геологи построили профиль горы и записали его по точкам в массив. Рельеф получился такой, что данные в массиве сначала возрастают, а потом убывают. Напишите программу, которая определяет высоту вершины горы быстрее, чем линейный поиск.
11. Массив A отсортирован по возрастанию. Напишите программу, которая ищёт в массиве «фиксированную точку»: такой элемент, для которого $A[i] = i$, или сообщает, что фиксированной точки нет. Программа должна работать быстрее, чем линейный поиск.
12. Напишите программу, которая по двум отсортированным массивам строит новый отсортированный массив, содержащий только значения, которые входят в оба исходных массива. Используйте двоичный поиск.
13. В массиве хранятся баллы, набранные участниками олимпиады. Они отсортированы в порядке убывания. С помощью двоичного поиска определите количество призёров — тех, кто набрал более 80% от количества баллов победителя.
- *14. *Проект.* Проведите исследование: определите, сколько операций поиска должно приходиться на одну сортировку для того, чтобы программа с двоичным поиском работала в среднем быстрее, чем с линейным. Зависят ли результаты от размера массива?

§ 4 Обработка файлов

Ключевые слова:

- файл
- текстовый файл
- двоичный файл
- файловая переменная
- открытие файла
- закрытие файла

Какие бывают файлы?

Файл — это набор данных во внешней памяти (на диске, флэш-накопителе и т. п.), имеющий имя. С точки зрения программиста, бывают файлы двух типов:

- *текстовые*, которые содержат текст, разбитый на строки; таким образом, из всех специальных символов в текстовых файлах могут быть только символы перехода на новую строку (обозначаются в программе как "`\n`") и возврата в начало строки ("`\r`");
- *двоичные*, в которых могут содержаться любые двоичные данные без ограничений; в двоичных файлах хранятся, например, рисунки, звуки, видеофильмы и т. д.

Пока мы будем работать только с текстовыми файлами.

Можно считать, что текстовый файл — это поток байтов. При чтении они поступают на вход программы один за другим (последовательно), поэтому файл обеспечивает последовательный доступ к данным. Это значит, что для того, чтобы прочитать 100-е по счёту значение из файла, нужно сначала прочитать предыдущие 99.

Принцип сэндвича

Работа с файлом из программы включает три этапа. Сначала надо *открыть файл*, т. е. сделать его активным для программы. Если файл не открыт, то программа не может к нему обращаться.

При открытии файла указывают *режим работы*: чтение, запись или добавление данных в конец файла. Открытый файл блокируется так, что другие программы не могут одновременно использовать его.

Когда файл открыт (активен), программа выполняет все необходимые операции с ним. После этого нужно *закрыть файл*, т. е. освободить его, разорвать связь с программой. Именно при закрытии файла все последние изменения, сделанные программой в этом файле, записываются на диск.

Такой принцип работы иногда называют «принципом сэндвича», в котором три слоя: хлеб, затем начинка, и потом снова хлеб (рис. 1.4).

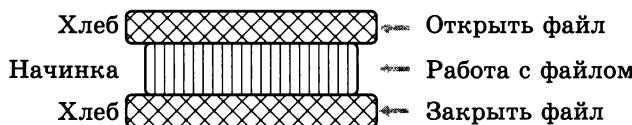


Рис. 1.4

В большинстве языков программирования с файлами работают через вспомогательные переменные (их называют *файловыми переменными*, идентификаторами файлов и т. п.).

В Python есть функция `open`, которая открывает файл и возвращает файловую переменную, через которую потом можно обращаться к файлу. Функция `open` принимает два аргумента. Первый — это имя файла или путь к файлу, если файл находится не в том каталоге, откуда запускается программа. Второй — режим открытия файла:

- "r" — чтение данных из существующего файла;
- "w" — запись данных в новый файл (если файл с таким именем уже есть, его старое содержание удаляется);
- "a" — добавление данных в конец файла (если файл не существует, создается пустой файл с заданным именем).

Чтобы закрыть файл, нужно вызвать метод `close` для файловой переменной.

```
Fin = open( "input.txt" )
Fout = open( "output.txt", "w" )
# здесь работаем с файлами
Fout.close()
Fin.close()
```

При первом вызове функции open режим работы с файлом не указан, в этом случае по умолчанию предполагается режим "r" (чтение данных).

Если файл, который открывается на чтение, не найден, возникает ошибка, и программа завершается аварийно. Если существующий файл открывается на запись, его содержимое уничтожается.

После окончания работы программы все открытые файлы закрываются автоматически.

Чтение данных

Текстовый файл разбит на отдельные строки, поэтому его можно читать и обрабатывать построчно. Чтение одной строки из открытого текстового файла выполняет метод readline, его нужно вызвать для файловой переменной:

```
Fin = open( "input.txt" )
s = Fin.readline()
```

Чтобы разделить данные, записанные в одной строке через пробел, используют метод split (как при вводе с клавиатуры). Этот метод разбивает строку по пробелам (и символам перевода строки "\n") и строит список из соответствующих «слов»:

```
s = Fin.readline().split()
```

Теперь, если в строке, прочитанной из файла, были записаны через пробел символы 1 и 2, список s будет выглядеть так:

```
["1", "2"]
```

Элементы этого списка — символьные строки. Поэтому для того, чтобы выполнять с этими данными вычисления, их нужно перевести в числовой формат с помощью функции int, применив её для каждого элемента списка:

```
a, b = int( s[0] ), int( s[1] )
```

То же самое можно записать в виде генератора:

```
a, b = [int(x) for x in s]
```

или вызвав функцию map:

```
a, b = map( int, s )
```

Здесь map(int, s) означает «применить функцию int ко всем элементам списка s». Вспомните, что подобные операции мы использовали при вводе данных с клавиатуры, когда нужно было ввести несколько чисел в одной строке.

В своей внутренней памяти система хранит положение указателя (*файлового курсора*), который определяет текущее место в файле. При открытии файла указатель устанавливается в самое начало файла, при

чтении смещается на позицию, следующую за прочитанными данными, а при записи переводится на следующую свободную позицию.

Если нужно повторить чтение с начала файла, проще всего закрыть файл, а потом снова открыть его в режиме чтения. При этом указатель снова будет установлен на начало файла.

Запись данных

Для вывода строки в файл применяют метод `write`. Если нужно вывести в файл числовые данные, их сначала преобразуют в строку, например с помощью метода `format`:

```
Fout.write( "{}+{}={}\\n".format(a, b, a+b) )
```

Так мы записали в файл результат сложения двух чисел. Обратите внимание, что, в отличие от функции `print`, при таком способе записи символ перехода на новую строку "`\n`" автоматически не добавляется, и мы добавили его в конце строки вручную.

Вывод файла на экран

Начнём с простой задачи — вывести на экран содержимое текстового файла. Для того чтобы определить, когда данные закончатся, будем использовать особенность метода `readline`: когда файловый курсор указывает на конец файла, метод `readline` возвращает пустую строку, которая воспринимается как ложное логическое значение:

```
while True:  
    s = Fin.readline()  
    if not s: break  
    print( s, end="" )
```

В этом примере при получении пустой строки цикл чтения заканчивается с помощью оператора `break`. Обратите внимание, что переход на новую строку в функции `print` отключён (`end=""`), потому что при чтении из файла символы перевода строки "`\n`" в конце строк сохраняются.

Возможны и другие варианты. Например, метод `readlines` позволяет прочитать все строки сразу в список:

```
Fin = open( "input.txt" )  
allStrings = Fin.readlines()  
Fin.close()  
for s in allStrings:  
    print( s, end="" )
```

Строки из файла загружаются в список `allStrings` и уже из списка выводятся на экран.

В Python есть ещё один способ работы с файлами, при котором закрывать файл не нужно, он закроётся автоматически. Это конструкция `with-as`:

```
with open( "input.txt" ) as Fin:
    for s in Fin:
        print( s, end="" )
```

В первой строке файл `input.txt` открывается в режиме чтения и связывается с файловой переменной `Fin`. Затем в цикле перебираются все строки в этом файле, каждая из них по очереди попадает в переменную `s` и выводится на экран. Закрывать файл с помощью метода `close` не нужно, он закроется автоматически после окончания цикла.

Наконец, приведём ещё один способ перебора строк в текстовом файле в стиле языка Python:

```
for s in open( "input.txt" ):
    print( s, end="" )
```

Этот вариант обычно рекомендуют к использованию. В этом случае файл также не нужно закрывать вручную.

Суммирование данных из файла

Предположим, что в текстовом файле записано в столбик неизвестное количество чисел, и требуется найти их сумму. В этой задаче не нужно одновременно хранить все числа в памяти (и не нужно выделять массив!), достаточно читать по одному числу и сразу его обрабатывать:

```
while не конец файла:
    # прочитать число из файла
    # добавить его к сумме
```

Будем считать, что файловая переменная `Fin` связана с файлом, открытым на чтение. Основная часть программы (без команд открытия и закрытия файлов) может выглядеть, например, так:

```
sum = 0
while True:
    s = Fin.readline()
    if not s: break
    sum += int(s)
```

или так:

```
sum = 0
for s in open( "input.txt" ):
    sum += int( s )
```

Обработка массивов

Пусть в текстовом файле записаны целые числа, по одному в строке. Требуется вывести в другой текстовый файл те же числа, отсортированные в порядке возрастания.

Особенность этой задачи в том, что для сортировки нам нужно удерживать в памяти все числа, т. е. для их хранения необходимо использовать массив (список).

Поскольку список в языке Python легко расширяется, мы можем загрузить в него достаточно много значений из файла (пока хватит оперативной памяти). Цикл, в котором данные читаются из файла и сохраняются в списке, может выглядеть так:

```
A = []
for s in open( "input.txt" ):
    A.append( int(s) )
```

Отметим, что эта программа работает правильно только тогда, когда в каждой строке записано одно число.

Есть и другой вариант, в стиле Python. Метод `read` читает (в отличие от `readline`) не одну строку, а весь файл. Затем мы разобьём получившуюся символьную строку на «слова» (символьные цепочки без пробелов и символов перевода строки "`\n`") с помощью метода `split`:

```
allStrings = Fin.read().split()
```

и преобразуем эти слова в числа, составив из них список:

```
A = list( map( int, allStrings ) )
```

В этом случае можно записывать несколько чисел в одной строке.

Теперь нужно отсортировать массив `A` (этот код вы уже можете написать самостоятельно) и вывести его во второй файл, открытый на запись:

```
Fout = open( "output.txt", "w" )
Fout.write( str(A) )
Fout.close()
```

В этом варианте мы использовали функцию `str`, которая представляет какой-то объект как символьную строку в стандартном формате. Если нам нужно форматировать данные по-своему, например вывести их в столбик, можно обработать каждый элемент вручную в цикле:

```
for x in A:
    Fout.write( str(x)+"\n" )
```

К символьной записи очередного элемента массива мы добавляем символ перехода на новую строку "`\n`". В этом случае числа выводятся в файл в столбик.

Обработка строк

Предположим, что текстовый файл содержит данные о собаках, привезённых на выставку. В каждой строке записаны кличка собаки, её возраст (целое число) и порода, разделённые точками с запятой, например:

Мухтар;4;немецкая овчарка

Нужно вывести в другой файл сведения о собаках, которым меньше 5 лет.

В этой задаче данные можно обрабатывать по одной строке (не нужно загружать все строки сразу в оперативную память):

```
while не конец файла ( Fin ):
    # прочитать строку из файла Fin
    # разобрать строку - выделить возраст
    if возраст < 5:
        # записать строку в файл Fout
```

Здесь, как и раньше, Fin и Fout — файловые переменные, связанные с файлами, открытыми на чтение и запись соответственно.

Будем считать, что все данные корректны, т. е. первая точка с запятой отделяет кличку от возраста, а вторая — возраст от породы. Для разбора очередной прочитанной строки s применим метод split, передав ему символ-разделитель — точку с запятой. Он вернёт список, где второй по счёту элемент (он имеет индекс 1) — это возраст собаки. Его и нужно преобразовать в целое число:

```
s = Fin.readline()
dogData = s.split(";")
sAge = dogData[1]
age = int( sAge )
```

Эти операции можно записать в краткой форме:

```
s = Fin.readline()
age = int( s.split(";") [1] )
```

Полная программа (без учёта команд открытия и закрытия файлов) выглядит так:

```
while True:
    s = Fin.readline()
    if not s: break
    age = int( s.split(";") [1] )
    if age < 5:
        Fout.write( s )
```

Её можно записать несколько иначе, используя метод readlines, который читает сразу все строки в список:

```
allStrings = Fin.readlines()
for s in allStrings:
    age = int ( s.split(";") [1] )
    if age < 5:
        Fout.write ( s )
```

или конструкцию **with-as**:

```
with open( "input.txt" ) as Fin:
    for s in Fin:
        age = int( s.split( ";" )[1] )
        if age < 5:
            Fout.write( s )
```

Вот ещё один вариант в стиле Python, возможно, наилучший:

```
for s in open( "input.txt" ):
    age = int( s.split( ";" )[1] )
    if age < 5:
        Fout.write( s )
```

Выводы

- Файл — это набор данных на диске, имеющий имя.
- С точки зрения программиста, бывают файлы двух типов: текстовые и двоичные. В текстовых файлах не может быть никаких специальных символов, кроме символов перехода на новую строку "\n" и символов возврата в начало строки "\r".
- До выполнения операций с файлом нужно открыть файл (сделать его активным), а после завершения всех действий — закрыть (освободить). Когда программа заканчивает работу, все файлы закрываются автоматически.
- После открытия файла для обращения к нему используют файловую переменную, которую вернула функция open.
- Файл обеспечивает последовательный доступ к данным. Текущее место в файле определяется положением файлового курсора.



Вопросы и задания

1. Чем различаются текстовые и двоичные файлы по внутреннему содержанию? Можно ли сказать, что текстовый файл — это частный случай двоичного файла?
2. Объясните, зачем нужно открывать и закрывать файлы.
3. Как вы думаете, почему для работы с файлом используют не имя файла, а файловую переменную?
4. Как вы думаете, почему открытый программой файл блокируется, и другие программы не могут получить к нему доступ?
5. В каких ситуациях может понадобиться закрыть файл с помощью метода close?
6. Что такое последовательный доступ к данным?
7. Как повторно прочитать данные с самого начала файла?
8. Как определить, что данные в текстовом файле закончились?
9. В каких случаях необходимо открывать одновременно несколько файлов?

10. Напишите программу, которая находит среднее арифметическое всех чисел, записанных в файле в столбик, и выводит результат в другой файл.
11. В файле в столбик записаны натуральные числа. Напишите программу, которая выводит в другой файл те же числа в двоичной (восьмеричной, шестнадцатеричной) системе счисления.
12. Напишите программу, которая находит минимальное и максимальное среди чётных положительных чисел, записанных в файле, и выводит результат в другой файл. Учтите, что таких чисел может вообще не быть.
13. В файле в столбик записаны целые числа. Напишите программу, которая определяет длину самой длинной цепочки идущих подряд одинаковых чисел и выводит результат в другой файл. Учтите, что таких цепочек может вообще не быть.
14. В файле записаны в столбик целые числа. Напишите программу, которая записывает в другой файл те же числа, отсортированные по возрастанию последней цифры.
15. В файле записаны в столбик целые числа. Напишите программу, которая записывает в другой файл те же числа, отсортированные по возрастанию суммы цифр.
16. В двух файлах записаны в столбик отсортированные по возрастанию последовательности целых чисел неизвестной длины. Напишите программу, которая объединит их и запишет все эти числа в третий файл в порядке возрастания. Не разрешается использовать списки.
17. Дополните решение задачи о собаках (из параграфа) так, чтобы программа обрабатывала ошибки в исходных данных. При любых ошибках программа должна сообщать о них и не завершаться аварийно.
18. В исходном файле записана речь подростка, в которой часто встречается слово-паразит «короче», например: «Мама, короче, мыла, короче, раму». Напишите программу, которая убирает из текста все слова-паразиты (должно остаться «Мама мыла раму»).
19. Напишите программу, которая читает текст из файла и определяет количество слов в нём.
20. Напишите программу, которая читает текст из файла и выводит в другой файл только те строки, в которых есть слова, начинающиеся с буквы «А».
21. Напишите программу, которая читает текст из файла и выводит в другой файл в столбик все слова, которые начинаются с буквы «А».
22. Напишите программу, которая читает текст из файла, заменяет везде слово «паровоз» на слово «поезд» и записывает изменённый текст в другой файл.

23. В файле записаны данные о результатах сдачи экзамена. Каждая строка содержит фамилию, имя и количество баллов, разделённые пробелами:

<Фамилия> <Имя> <Количество баллов>

Напишите программу, которая выводит на экран фамилии и имена учеников, получивших больше 80 баллов.

24. В предыдущей задаче добавьте к списку нумерацию, например:

- 1) Иванов Вася
- 2) Петров Петя

25. В предыдущей задаче сократите имя до одной буквы и поставьте её перед фамилией, например:

- 1) В. Иванов
- 2) П. Петров

26. В предыдущей задаче отсортируйте список по алфавиту (по фамилии).

*27. В предыдущей задаче отсортируйте список по убыванию полученных баллов (баллы тоже нужно вывести в выходной файл).

§ 5

Целочисленные алгоритмы

Ключевые слова:

- решето Эратосфена
- итерационный метод
- квадратный корень

Во многих задачах все исходные данные и результаты, которые требуется получить, — целые числа. При этом желательно, чтобы все промежуточные вычисления тоже проводились только с целыми числами. На это есть, по крайней мере, две причины:

- процессор, как правило, выполняет операции с целыми числами немного быстрее, чем с вещественными;
- целые числа всегда точно представляются в памяти компьютера, и вычисления с ними всегда выполняются без ошибок (если, конечно, не происходит переполнение разрядной сетки).

Решето Эратосфена

Во многих прикладных задачах, например при шифровании данных, используются простые числа. Основные задачи при работе с простыми числами — это проверка числа на простоту и нахождение всех простых чисел в некотором диапазоне.

Пусть задано некоторое натуральное число N и требуется найти все простые числа на отрезке $[2; N]$. Самое простое (но неэффективное) решение этой задачи состоит в том, что в цикле перебираются все натуральные числа от 2 до N , и каждое из них отдельно проверяется на простоту. Например, можно проверить, есть ли у числа k делители в диапазоне от 2 до \sqrt{k} . Если ни одного такого делителя нет, то число k простое.

Описанный метод при больших N работает довольно медленно, он имеет асимптотическую сложность $O(N\sqrt{N})$. Греческий математик Эратосфен Киренский (276–194 гг. до н. э.) предложил другой алгоритм, который работает намного быстрее:

- 1) выписать в строчку все натуральные числа от 2 до N ;
- 2) начать с $k = 2$;
- 3) вычеркнуть все числа, кратные k ($2k, 3k, 4k, \dots$ и т. д.);
- 4) найти следующее невычеркнутое число и присвоить его переменной k ;
- 5) повторять шаги 3 и 4, пока $k < N$.

Покажем работу алгоритма при $N = 16$:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Первое невычеркнутое число — 2, поэтому вычёркиваем все чётные числа, начиная с 4:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 10 11 ~~12~~ 13 14 15 16

Далее вычёркиваем все числа, кратные 3:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ 10 11 ~~12~~ 13 14 15 16

Все числа, кратные 5 и 7, уже вычеркнуты, а чисел, кратных 11 и 13 (кроме самих этих чисел), в последовательности нет. Таким образом, остались не вычеркнутыми только простые числа 2, 3, 5, 7, 11 и 13.

Согласно легенде, Эратосфен писал числа на дощечке, покрытой воском. «Вычёркивая» составные числа, он прокалывал дырочки на их месте. Можно представить, что составные числа «проваливаются в решето» и в решете остаются только простые числа.

Классический алгоритм можно ещё улучшить, уменьшив количество операций. Заметьте, что при вычёркивании чисел, кратных трём, нам не пришлось вычёркивать число 6, так как оно уже было вычеркнуто. Кроме того, все числа, кратные 5 и 7, к последнему шагу тоже оказались вычеркнуты.

Предположим, что мы хотим вычёркнуть все числа, кратные некоторому k , например $k = 5$. При этом числа $2k, 3k$ и $4k$ уже были вычеркнуты на предыдущих шагах, поэтому нужно начать не с $2k$, а с k^2 . Тогда получается, что при $k^2 > N$ вычёркивать уже будет нечего, что мы и увидели в примере. Поэтому можно использовать улучшенный алгоритм:

- 1) выписать все числа от 2 до N ;
- 2) начать с $k = 2$;
- 3) вычеркнуть все числа, кратные k , начиная с k^2 ;
- 4) найти следующее невычеркнутое число и присвоить его переменной k ;
- 5) повторять шаги 3 и 4, пока $k^2 \leq N$.

Чтобы составить программу, нужно определить, что значит «выписать все числа» и «вычеркнуть число». Один из возможных вариантов хранения данных — массив логических величин с индексами от 2 до N .

Поскольку индексы элементов списков в Python всегда начинаются с нуля, для того чтобы работать с нужным диапазоном индексов (от 2 до N), необходимо выделить логический массив из $N+1$ элементов. Элементы с индексами 0 и 1 мы не будем использовать.

Если число i не вычеркнуто, будем хранить в элементе массива $A[i]$ истинное значение (True), если вычеркнуто — ложное (False). В самом начале нужно заполнить массив истинными значениями:

```
N = 100
A = [True] * (N+1)
```

В основном цикле выполняется описанный выше алгоритм:

```
k = 2
while k*k <= N:
    if A[k]:
        i = k*k
        while i <= N:
            A[i] = False
            i += k
    k += 1
```

Обратите внимание, что для того, чтобы вообще не применять вещественную арифметику (и не потерять точность вычислений!), мы заменили условие $k \leq \sqrt{N}$ на равносильное условие $k^2 \leq N$, в котором используются только целые числа.

Внутренний цикл можно переписать как цикл по переменной: переменная i изменяется, начиная с k^2 , с шагом k , пока не достигнет значения N :

```
k = 2
while k*k <= N:
    if A[k]:
        for i in range(k*k, N+1, k):
            A[i] = False
    k += 1
```

После завершения этого цикла невычеркнутыми останутся только простые числа, для них соответствующие элементы массива содержат истинное значение. Эти простые числа и нужно вывести на экран:

```
for i in range(2, N+1):
    if A[i]:
        print(i)
```

Можно поступить по-другому: отобрать в новый массив только невычеркнутые числа, для которых соответствующие элементы массива A остались истинными. Для этого используется генератор списка:

```
primes = [i for i in range(2, N+1) if A[i]]
print(primes)
```

Квадратный корень

Рассмотрим ещё одну задачу: вычисление целочисленного квадратного корня из натурального числа. Требуется найти максимальное целое число, квадрат которого не больше, чем заданное натуральное число.

К сожалению, стандартная функция `sqrt` из модуля `math` не поддерживает работу с целыми числами произвольной длины, и результат её работы — вещественное число¹⁾. Поэтому в том случае, когда нужно именно целое значение, приходится использовать специальные методы.

Один из самых известных алгоритмов вычисления квадратного корня, придуманный ещё в Древней Греции, — метод Герона Александрийского, который сводится к многократному применению формулы²⁾

$$x_i = \frac{1}{2} \left(x_{i-1} + \frac{a}{x_{i-1}} \right).$$

Здесь a — это число, из которого извлекается корень, а x_{i-1} и x_i — предыдущее и следующее *приближения* (неточные значения корня). Фактически по формуле Герона вычисляется среднее арифметическое между x_{i-1} и (a/x_{i-1}) . Пусть одно из этих значений меньше, чем \sqrt{a} , тогда второе обязательно больше, чем \sqrt{a} . Поэтому их среднее арифметическое с каждым шагом приближается к значению корня.

Метод Герона — это метод, в котором используются циклические вычисления, или *итерации* (от латинского слова *iteratio* — повторяю). Такой метод называется *итерационным*.

Метод Герона сходится (т. е. приводит к точному решению) при любом начальном приближении x_0 , не равном нулю. Например, можно выбрать начальное приближение $x_0 = a$.

¹⁾ Заметим, что возведение целого числа в степень 0,5 тоже даёт вещественное число.

²⁾ Фактически эта формула — результат применения метода Ньютона для решения нелинейных уравнений к уравнению $x^2 = a$.

Приведённая формула служит для вычисления вещественного значения корня. Для того чтобы найти целочисленное значение корня, можно заменить оба деления на целочисленные, на языке Python это запишется так:

```
x = (x + a // x) // 2
```

или привести выражение в скобках к общему знаменателю для того, чтобы использовать всего одно целочисленное деление:

```
x = (x * x + a) // (2 * x)
```

Функция для вычисления целочисленного квадратного корня может выглядеть так:

```
def isqrt(a):
    x = a
    while True:
        x1 = (x * x + a) // (2 * x)
        if x1 >= x: return x
        x = x1
```

Здесь наиболее интересный момент — условие выхода из цикла. Как вы знаете, цикл с заголовком `while True` — это бесконечный цикл, из которого можно выйти только с помощью оператора `break` (или `return` в теле функции). Мы начинаем поиск с начального приближения $x_0 = a$, которое (при больших a) заведомо больше правильного ответа. Поэтому каждое следующее приближение должно быть меньше предыдущего. А как только очередное приближение окажется *большим или равным* предыдущему, целочисленный квадратный корень будет найден.

Выводы

- Если все данные и результаты, которые нужно получить, — целые числа, желательно выполнять вычисления, используя только операции с целыми числами.
- Решето Эратосфена — это алгоритм быстрого поиска простых чисел на отрезке $[2; N]$. В нём используется дополнительный массив из N элементов.
- Для поиска целочисленного квадратного корня можно использовать итерационный метод Герона.



Вопросы и задания

1. Докажите, что если у числа k нет ни одного делителя на отрезке $[2; \sqrt{k}]$, то оно простое.
2. Какие преимущества и недостатки имеет алгоритм «решето Эратосфена» по сравнению с проверкой каждого числа на простоту?

3. Напишите две программы, которые находят все простые числа в диапазоне от 2 до N двумя разными способами:

- 1) проверкой каждого числа из этого диапазона на простоту;
- 2) используя решето Эратосфена.

Сравните число итераций внутреннего цикла (время работы) этих программ для разных значений N . Постройте для каждого варианта зависимость количества итераций от N , сделайте выводы о сложности алгоритмов.

*4. Предложите алгоритм и напишите программу для поиска целого кубического корня из целого числа.

§ 6 Словари

Ключевые слова:

- словарь
- ключ
- значение
- перебор элементов
- сортировка по ключу

Что такое словарь?

Рассмотрим такую задачу. В файле находится список слов, среди которых есть повторяющиеся. Каждое слово записано в отдельной строке. Требуется построить алфавитно-частотный словарь: все различные слова должны быть записаны в другой файл в алфавитном порядке, справа от каждого слова должно быть указано, сколько раз оно встречается в исходном файле.

Каждое слово, найденное в файле, нужно хранить вместе с числом — количеством таких слов. Для этой цели удобно применить специальный тип данных — *ассоциативный массив*, или *словарь*. Он хранит пары «ключ — значение» и поддерживает три основные операции:

- добавление пары «ключ — значение»;
- поиск значения по ключу;
- удаление пары по ключу.

В нашей задаче ключ — это слово (символьная строка), а значение — количество таких слов (целое число).

В языке Python тип данных «словарь» называется **dict** (от англ. *dictionary*)¹⁾. Обращение к элементу словаря похоже на обращение к элементу массива, только вместо числового индекса указывается ключ (в нашей задаче — символьная строка).

¹⁾ В других языках программирования такие структуры данных могут называться иначе, например, «ассоциативный массив», «отображение» или «хэш».

Например, вывести на экран количество найденных слов «бегемот» можно так:

```
print( wordList["бегемот"] )  
где wordList — имя словаря.
```



Словарь — это неупорядоченный набор элементов, в котором доступ к элементу выполняется по ключу.

Слово «неупорядоченный» в этом определении говорит о том, что порядок элементов в словаре никак не задан, он определяется внутренними механизмами хранения данных языка Python. Поэтому сортировку словаря выполнить невозможно, как невозможно, в отличие от списка, указать для какого-то элемента словаря его соседей — предыдущий и следующий элементы.

Ключом может быть любое неизменяемое значение, например число или символьная строка. В одном словаре можно использовать ключи разных типов.

Неизменяемость чисел и символьных строк — это особенность языка Python. Например, пусть имя `greet` связано с символьной строкой:

```
greet = "Hi!"
```

Изменять эту строку нельзя, поэтому следующий оператор вызовет сообщение об ошибке:

```
greet[1] = "а" # Ошибка! Стока неизменяема!
```

Если присвоить строке новое значение:

```
greet = "Привет!"
```

то будет выделен новый участок памяти, с которым связывается имя `greet` (рис. 1.5). А память, которую занимает строка "Hi!", будет освобождена сборщиком «мусора».

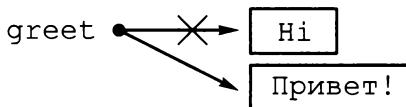


Рис. 1.5

Точно так же транслятор работает с числами.

Массив (список) — это изменяемый объект, поэтому он не может быть ключом. Вместо массива можно использовать *кортеж* — неизменяемый список, который заключается в круглые скобки вместо квадратных:

```
d = {} # Пустой словарь  
d[[1,2]] = "no" # Ошибка! Так нельзя!  
d[(1,2)] = "yes" # Так можно!
```

Алфавитно-частотный словарь

Вернёмся к нашей задаче построения алфавитно-частотного словаря. Алгоритм, записанный в виде псевдокода, может выглядеть так:

```
# создать пустой словарь
while есть слова в файле:
    # прочитать очередное слово
    if слово есть в словаре:
        # увеличить на 1 счётчик для этого слова
    else:
        # добавить слово в словарь
        # записать 1 в счётчик слова
```

Теперь нужно записать все шаги этого алгоритма с помощью операторов языка Python.

Словарь определяется с помощью фигурных скобок, например:

```
wordList = {"бегемот": 0, "пароход": 2}
```

В этом словаре два элемента: ключ «бегемот» связан со значением 0, а ключ «пароход» — со значением 2. Пустые фигурные скобки задают пустой словарь, в котором нет ни одного элемента:

```
wordList = {}
```

Для того чтобы добавить элемент в словарь, используют присваивание:

```
wordList["полёт"] = 1
```

Если ключ "полёт" уже есть в словаре, соответствующее значение будет изменено, а если такого ключа нет, то он будет добавлен и связан со значением 1.

Нам нужно увеличивать значение счётчика слов на 1, это можно сделать так:

```
wordList["полёт"] += 1
```

Однако, если ключа "полёт" нет в словаре, такая команда приведёт к ошибке. Для того чтобы определить, есть ли в словаре какой-то ключ, можно использовать оператор **in**:

```
if "полёт" in wordList:
    wordList["полёт"] += 1
else:
    wordList["полёт"] = 1
```

Если ключ "полёт" есть в словаре, соответствующее ему значение увеличивается на 1, если такого ключа нет, то он создаётся и связывается со значением 1.

Можно обойтись вообще без условного оператора, если использовать метод `get` для словаря. Этот метод возвращает значение, связанное с существующим ключом. Если ключа нет в словаре, метод возвращает значение по умолчанию, которое задаётся как второй аргумент:

```
wordList["полёт"] = wordList.get("полёт", 0) + 1
```

В данном случае значение по умолчанию равно 0. Поэтому, если ключа "полёт" нет в словаре, создаётся новый элемент с таким ключом, и после выполнения команды связанное с ним значение будет равно 1. Если слово уже есть в словаре, его счётчик увеличивается на 1.

Теперь у нас есть всё для того, чтобы написать полный цикл для ввода данных из файла и составления списка:

```
wordList = {}
for s in open("input.txt"):
    word = s.strip() # (1)
    if word:
        wordList[word] = wordList.get(word, 0) + 1
```

Обратим внимание на строку 1 в программе. После чтения очередной строки из файла вызывается метод `strip` (в переводе с английского — лишать, удалять), который удаляет лишние пробелы в начале и в конце строки `s`, а также завершающий символ перевода строки "\n".

Перебор элементов словаря

В отличие от списка к элементам словаря нельзя обращаться по индексам. Тогда возникает вопрос — как же перебрать все существующие ключи, если мы не знаем их заранее? Для этой цели мы запрошим у словаря список всех ключей, вызвав метод `keys`:

```
allKeys = wordList.keys()
```

Ключи можно отсортировать по алфавиту с помощью функции `sorted`:

```
sortKeys = sorted(wordList.keys())
```

В современных версиях языка Python метод `keys` можно не вызывать явно:

```
sortKeys = sorted(wordList)
```

Остаётся только перебрать в цикле все элементы этого списка, например так:

```
F = open("output.txt", "w")
for key in sorted(wordList):
    F.write("{}: {}\n".format(key, wordList[key]))
F.close()
```

Ключи из отсортированного списка ключей попадают по очереди в переменную `key`, в каждой строке выводится очередной ключ и через двоеточие — связанное с ним значение, т. е. количество таких слов в исходном файле.

Для того чтобы преобразовать данные перед выводом в символьную строку, используется метод `format`. Две пары фигурных скобок в строке форматирования обозначают места для вывода первого и второго аргументов функции.

Отметим, что у словарей есть метод `values`, который возвращает список значений. Например, вот так можно вывести значения для всех ключей:

```
for value in wordList.values():
    print( value )
```

Если же нас интересуют пары «ключ — значение», удобно использовать метод `items`, который возвращает список таких пар. Перебрать все пары и вывести их на экран можно с помощью следующего цикла:

```
for key, value in wordList.items():
    print( key, "->", value )
```

В этом цикле используются две переменные: `key` (ключ) и `value` (значение). Поскольку `wordList.items()` — это список пар «ключ — значение», при очередной итерации первый элемент каждой пары (ключ) попадает в переменную `key`, а второй (значение) — в переменную `value`.

Выводы

- Словарь — это набор пар «ключ — значение», поддерживающий операции добавления пары, поиска значения по ключу и удаления пары по ключу.
- Элементы в словаре неупорядочены, т. е. нельзя указать для элемента предыдущий и следующий.
- Ключом может быть любое значение неизменяемого типа, например число или символьная строка.
- Для перебора ключей в словаре используется метод `keys`, для перебора значений — метод `values`, а для перебора всех пар «ключ — значение» — метод `items`.

Вопросы и задания

1. Какие операции можно выполнять со словарём?
2. Можно ли обратиться к элементу словаря по индексу (номеру)? Как вы думаете, почему сделано именно так?
3. Как создать словарь из готовых пар «ключ — значение»? Найдите в дополнительных источниках разные способы решения этой задачи.
4. Чем можно заменить метод `get`?



5. Используя дополнительные источники, выясните, как можно удалить ключ из словаря.
 6. Зачем при чтении строк из файла используется метод `strip()`?
 7. Постройте полную программу, которая составляет алфавитно-частотный словарь для заданного файла.
 8. В предыдущей задаче выведите все найденные слова в файл в порядке убывания частоты, т. е. в начале списка должны стоять слова, которые встречаются в файле чаще других.
- *9. *Проект.* Доработайте программу для построения алфавитно-частотного словаря так, чтобы она выполняла обработку файла, разбивая текст на отдельные слова и удаляя знаки препинания.
- *10. *Проект.* Используя программу для построения алфавитно-частотного словаря, сравните тексты разных авторов (используйте материалы из Интернета). Например, для текстов на русском языке можно сравнить частоты предлогов «в», «на», «с».

§ 7

Структуры

Ключевые слова:

- структура
- поле
- точечная запись
- класс
- исключение
- сортировка
- ключ

Зачем нужны структуры?

В базе данных библиотеки хранится информация о книгах. Для каждой из них нужно запомнить автора, название, год издания, количество страниц, число экземпляров и т. д. Как хранить эти данные?

Поскольку книг много, нужен массив. Но информация о книгах разнородна, она содержит целые числа и символьные строки разной длины.

Конечно, можно разбить эти данные на несколько массивов (массив авторов, массив названий и т. д.), так чтобы i -й элемент каждого массива относился к книге с номером i . Но такой подход слишком неудобен и ненадёжен. Например, при сортировке нужно синхронно переставлять элементы нескольких массивов (отдельно!), при этом можно легко ошибиться и нарушить связь данных.

Возникает естественная идея — объединить все данные, относящиеся к каждой книге, в единый блок памяти, который программисты называют структурой или записью.

Структура (запись) — это тип данных, который может включать в себя несколько *полей* — элементов разных типов (в том числе и другие структуры).



При обработке событий в играх мы уже использовали блоки данных (т. е. *структуры*), к полям которых обращались с помощью точечной записи. Вместо большого количества отдельных данных обработчик события получает от системы один объект — структуру, в которой «упакована» вся нужная информация о событии (*x*-координата и *y*-координата курсора мыши, состояние её кнопок и т. п.).

Классы

Структуры — это составные типы данных, которые определяются программистом. В языке Python такие типы данных называются *классами*.

Введём новый класс `TBook` — структуру, с помощью которой можно описать книгу в базе данных библиотеки. Будем хранить в структуре только¹⁾:

- фамилию автора (символьная строка);
- название книги (символьная строка);
- имеющееся в библиотеке количество экземпляров (целое число).

Класс можно объявить так:

```
class TBook:  
    pass
```

Объявление начинается служебным словом `class`. Имя нового *класса* (типа данных) — `TBook` (от *Type Book*, по-английски — «тип книга»), хотя можно было использовать и любое другое имя. Названия классов в языке Python принято начинать с прописной буквы.

Слово `pass` (в переводе с английского — пропустить) — это пустой оператор («заглушка»), не выполняющий никаких действий. Он стоит во второй строке только потому, что оставить строку совсем пустой нельзя — будет ошибка. В данном случае мы пока не определяем свойства объектов этого класса, просто говорим, что есть такой класс.

При объявлении класса можно сразу определить поля (данные) и их значения по умолчанию:

```
class TBook:  
    author = "?"  
    title = ""  
    count = 0
```

Это объявление говорит о том, что все структуры класса `TBook` имеют три поля: `author` (в переводе с английского — автор), `title` (название) и `count` (количество). В поле `author` по умолчанию

¹⁾ Конечно, в реальной ситуации данных больше, но принцип не меняется.

записывается знак вопроса, в поле `title` — пустая строка, а в поле `count` — число 0.

В отличие от других языков программирования (C++, Паскаль) при объявлении класса не обязательно сразу перечислять все *поля* (данные) структуры, они могут добавляться по ходу выполнения программы. Такой подход имеет свои преимущества и недостатки. С одной стороны, программисту удобнее работать, больше свободы. С другой стороны, повышается вероятность ошибок. Например, при опечатке в названии поля может быть создано новое поле с неверным именем, и найти эту ошибку можно только вручную, просматривая код программы.

В Python 3.7 в стандартную библиотеку был добавлен модуль `dataclasses`, предназначенный специально для работы со структурами. Информацию о нём вы можете найти в справочной системе.

Создание структур

Теперь уже можно построить объект этого класса:

```
b = TBook()
```

Массив из таких объектов можно создать с помощью цикла:

```
N = 100
books = []
for i in range(N):
    books.append( TBook() )
```

или генератора списка:

```
books = [ TBook() for i in range(N) ]
```

Обратите внимание, что такой способ использовать нельзя:

```
books = [ TBook() ]*N # ошибка!
```

Дело в том, что список `books` содержит указатели (адреса) объектов типа `TBook`, и в последнем варианте фактически создаётся один объект, адрес которого записывается во все 100 элементов массива. Поэтому изменение одного элемента массива «синхронно» изменит и все остальные элементы.

Что можно делать с полями структуры?

Для того чтобы работать не со всей структурой, а с отдельными полями, используют так называемую *точечную запись*, разделяя точкой имя структуры и имя поля. Например, `b.author` обозначает «поле `author` структуры `b`», а `books[5].count` — «поле `count` элемента массива `books[5]`».

С полями структуры можно обращаться так же, как и с обычными переменными соответствующего типа. Можно вводить их с клавиатуры (или из файла):

```
b.author = input()
b.title = input()
b.count = int( input() )
```

присваивать новые значения прямо в программе:

```
b.author = "Пушкин А.С."
b.title = "Полтава"
b.count = 1
```

использовать при обработке данных:

```
fam = b.author.split()[0]      # только фамилия
print( fam )
b.count -= 1                  # одну книгу взяли
if b.count == 0:
    print( "Этих книг больше нет!" )
```

и выводить на экран:

```
print( b.author, b.title + ".", b.count, "шт." )
```

Работа с файлами

В программах, которые работают с данными на диске, нужно читать массивы структур из файла и записывать в файл. Конечно, можно хранить структуры в текстовых файлах, например, записывая все поля каждой структуры в одну строку и разделяя их каким-то символом-разделителем, который не встречается внутри самих полей.

Но есть более грамотный способ, который позволяет хранить данные в файлах во внутреннем формате, т. е. так, как они представлены в памяти компьютера во время работы программы. Для этого в Python используется стандартный модуль `pickle`, который подключается с помощью команды `import`:

```
import pickle
```

Запись структуры в файл выполняется с помощью процедуры `dump`:

```
b = TBook()
Fout = open( "books.dat", "wb" )
b.author = "Тургенев И.С."
b.title = "Муму"
b.count = 2
pickle.dump( b, Fout );
Fout.close()
```

Обратите внимание, что файл открывается в режиме `wb`; первая буква `w`, от англ. *write* — писать, нам уже знакома, а вторая — `b` — это сокращение от англ. *binary* — двоичный, она говорит о том, что данные записываются в файл в двоичном (внутреннем) формате, а не как текстовые строки.

С помощью цикла можно записать в файл массив структур:

```
for b in books:  
    pickle.dump( b, Fout )
```

а можно даже обойтись без цикла:

```
pickle.dump( books, Fout );
```

При чтении этих данных нужно использовать тот же способ, что и при записи: если структуры записывались по одной, читать их тоже нужно по одной, а если записывался весь массив за один раз, так же его нужно и загружать в память.

Прочитать из файла одну структуру и вывести её поля на экран можно следующим образом:

```
Fin = open( "books.dat", "rb" )  
b = pickle.load( Fin )  
print( b.author, b.title, b.count, sep = ", " )  
Fin.close()
```

Если массив (список) структур записывался в файл за один раз, читать его нужно тоже одним вызовом функции `load`:

```
books = pickle.load( Fin )
```

Если структуры записывались в файл по одной и их количество N известно, при чтении этих данных в массив можно применить цикл с переменной:

```
books = [0]*N  
for i in range(N):  
    books[i] = pickle.load( Fin )
```

В этом случае массив `books` нужного размера N должен быть заранее создан, чтобы обращение `books[i]` в теле цикла не привело к выходу за границы массива.

Если же число структур неизвестно, нужно выполнять чтение до тех пор, пока файл не закончится, т. е. пока при очередном чтении не произойдёт ошибка:

```
books = []  
while True:  
    try:  
        books.append( pickle.load( Fin ) )  
    except:  
        break
```

Мы сначала создаём пустой список `books`, а затем в цикле читаем из файла очередную структуру и добавляем её в конец списка с помощью метода `append`.

Обработка ошибки выполнена с помощью исключений.

Исключение (англ. *exception* — исключительная ситуация) — это аварийная ситуация, которая делает дальнейшие вычисления по основному алгоритму невозможными или бессмысленными.



Исключение может произойти, например, при делении на ноль, ошибке при вводе или выводе данных, нехватке памяти и т. п. В нашем случае исключение — это неудачное чтение структуры из файла.

«Опасные» операторы, которые могут вызвать ошибку, записываются в виде блока со сдвигом вправо после слова **try**. После этого в следующем блоке, который начинается со служебного слова **except**, записывают команды, которые нужно выполнить в случае ошибки. В нашем случае тело цикла содержит примерно такую команду компьютеру: «попробуй прочитать из файла ещё одну структуру, если получилось — добавь её в массив, если не получилось — прерви выполнение цикла».

Сортировка

Для сортировки массива структур применяют те же методы, что и для сортировки массива простых переменных. Структуры можно сортировать по возрастанию или убыванию одного из полей, которое называют **ключевым полем** или **ключом**. Иногда используют и сложные (**составные**) ключи, зависящие от нескольких полей.

Отсортируем массив **books**, состоящий из структур типа **TBook**, по фамилиям авторов в алфавитном порядке. В данном случае ключом будет поле **author**. Предположим, что фамилия состоит из одного слова, а за ней через пробел следуют инициалы. Тогда сортировка методом пузырька выглядит так:

```
N = len(books)
for i in range(0, N-1):
    for j in range(N-2, i-1, -1):
        if books[j].author > books[j+1].author:
            books[j], books[j+1] = books[j+1], books[j]
```

Как вы знаете, при сравнении двух символьных строк они рассматриваются посимвольно до тех пор, пока не будут найдены первые различающиеся символы. Далее сравниваются коды этих символов по кодовой таблице. Так как код пробела меньше, чем код любой русской (и латинской) буквы, строка с фамилией «Волк» окажется выше в отсортированном списке, чем строка с более длинной фамилией «Волков», даже с учётом того, что после фамилии есть инициалы. Если фамилии одинаковы, сортировка происходит по первой букве инициалов, затем — по второй букве и т. д.

Отметим, что при такой сортировке данные, входящие в структуры, не перемещаются. Дело в том, что в массиве books хранятся указатели (адреса) отдельных элементов, поэтому при сортировке переставляются именно эти указатели. Это очень важно, если структуры имеют большой размер или их по каким-то причинам нельзя перемещать в памяти.

Теперь покажем сортировку в стиле Python. Попытка просто вызвать метод sort для списка books приводит к ошибке, потому что транслятор не знает, как сравнить два объекта типа TBook. Нужно ему помочь — указать, какое из полей играет роль ключа. Для этого при вызове метода sort мы используем именованный аргумент key. Например, можно указать в качестве ключа функцию, которая выделяет поле author из структуры:

```
def getAuthor( b ):
    return b.author
books.sort( key = getAuthor )
```

Более красивый способ — использовать лямбда-функцию, т. е. функцию без имени (вспомните материал § 2):

```
books.sort( key = lambda x: x.author )
```

Здесь в качестве ключа указана лямбда-функция, которая из переданного ей параметра x выделяет поле author, т. е. делает то же самое, что и функция getAuthor.

Если не нужно изменять сам список books, можно использовать не метод sort, а функцию sorted, например так:

```
for b in sorted( books,
                 key = lambda x: x.author ):
    print( b.author, b.title + ".", b.count, "шт." )
```

Выводы

- Структура (запись) — это составной тип данных, который позволяет объединить данные разных типов в один блок памяти. Элементы структуры называют полями.
- Структуры в языке Python вводятся как классы — типы данных, определяемые программистом.
- К полям структуры обращаются с помощью точечной записи: <имя структуры>.<имя поля>.
- Для сохранения данных в файле в двоичном виде используются подпрограммы модуля pickle.
- Сортировка массива структур выполняется по ключу. Ключом может быть поле структуры или условие, зависящее от значения нескольких полей.

Вопросы и задания

1. В чём отличие структуры от массива, содержащего однотипные элементы?
2. В чём отличие структуры от списка, который может содержать элементы разных типов?
3. В каких случаях использование структур даёт преимущества? Какие именно?
4. Как объявляется новый тип данных для хранения структур в Python? Выделяется ли при этом память?
5. Зачем используется точечная запись?
6. Что такое двоичный файл? Чем он отличается от текстового?
7. Как можно сортировать структуры? Опишите разные способы и сравните их.
8. Опишите структуру, в которой хранится информация о:
 - 1) видеозаписи;
 - 2) сотруднике фирмы;
 - 3) самолёте;
 - 4) породистой собаке.
9. *Проект.* Постройте программу, которая работает с базой данных собственного формата, хранящейся в виде файла. Ваша система управления базой данных должна иметь следующие возможности:
 - 1) просмотр записей;
 - 2) добавление записей;
 - 3) удаление записей;
 - 4) сортировка по одному из полей.
- *10. *Проект.* Постройте программу, которая работает со списком деловых встреч бизнесмена. Запись о каждой встрече содержит дату, время, место и данные человека, с которым назначена встреча. Программа должна иметь следующие возможности:
 - 1) просмотр отсортированного списка встреч (начиная с ближайших);
 - 2) добавление новой встречи;
 - 3) удаление встречи;
 - 4) автоматическое удаление встреч, которые уже состоялись (дата и время которых меньше текущих).

§ 8

Стек, очередь, дек

Ключевые слова:

- стек
- вызов подпрограмм
- префиксная форма
- кадр стека
- постфиксная форма
- очередь
- системный стек
- дек

Что такое стек?

Представьте себе стопку книг (подносов, кирпичей и т. п.). С точки зрения информатики её можно воспринимать как список элементов, расположенных в определённом порядке. Этот список имеет одну особенность — удалять и добавлять элементы можно только с одной («верхней») стороны. Действительно, для того чтобы вытащить какую-то книгу из стопки, нужно сначала снять все те книги, которые находятся на ней. Положить книгу сразу в середину стопки тоже нельзя.



Стек (англ. *stack* — стопка) — это линейная структура данных, в которой элементы добавляются и удаляются только с одного конца (англ. *LIFO*: *Last In* — *First Out*, «последним пришёл — первым ушёл»).

На рисунке 1.6 показаны примеры стеков: стопка книг, держатель для шаров, автоматный магазин и детская пирамидка.

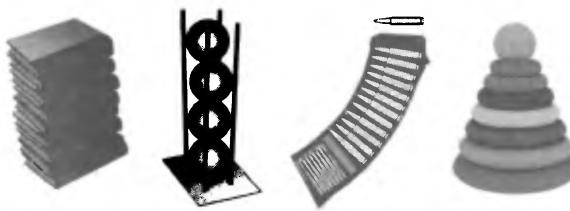


Рис. 1.6

При выполнении программ используется *системный стек* — область оперативной памяти, где хранятся адреса возврата из подпрограмм, параметры, передаваемые функциям и процедурам, а также локальные переменные.

Для стека определены две операции:

- добавить элемент на вершину стека (англ. *push* — втолкнуть);
- получить элемент с вершины стека и удалить его из стека (англ. *pop* — вытолкнуть).

Рассмотрим такую задачу. В файле в столбик записаны целые числа — результаты измерений прибора, записанные в хронологическом порядке (самые новые — в конце файла). Нужно переписать их в дру-

гой файл в обратном порядке (самые новые — впереди). В этой задаче очень удобно использовать стек.

Запишем алгоритм решения на псевдокоде. Сначала читаем данные и добавляем их в стек:

```
while файл не пуст:
    # прочитать x
    # добавить x в стек
```

Теперь верхний элемент стека — это последнее число, прочитанное из файла (самый новый результат измерения). Поэтому остаётся «вытолкнуть» все записанные в стек числа, они будут выходить в обратном порядке:

```
while стек не пуст:
    # вытолкнуть число из стека в x
    # записать x в файл
```

Использование списка

Поскольку стек — это линейная структура данных с переменным количеством элементов, для работы со стеком в программе на языке Python удобно использовать список. Вершина стека будет находиться в конце списка. Тогда для добавления элемента на вершину стека (т. е. в конец списка) можно применить уже знакомый нам метод append:

```
stack.append( x )
```

Снять элемент со стека (удалить последний элемент списка) можно с помощью метода pop:

```
x = stack.pop()
```

Метод pop — это функция, которая выполняет две задачи:

- удаляет последний элемент списка (если вызывается без параметров);
- возвращает удалённый элемент как результат функции, так что его можно сохранить в какой-либо переменной.

Теперь напишем цикл ввода целых чисел в стек из файла:

```
stack = []
for s in open( "input.dat" ):
    stack.append( int(s) )
```

Затем выводим элементы массива в файл в обратном порядке:

```
Fout = open( "output.txt", "w" )
while len( stack ) > 0: # пока стек не пуст
    x = stack.pop()
    Fout.write( str(x) + "\n" )
Fout.close()
```

Заметим, что перед записью в файл с помощью метода `write` все данные нужно преобразовать в формат символьной строки, это делает функция `str`. Символ перехода на новую строку "`\n`" добавляется в конец строки вручную.

Поскольку пустой список воспринимается интерпретатором Python как ложное значение, программу можно сократить, заодно избавившись от переменной `x`:

```
Fout = open( "output.txt", "w" )
while stack:
    Fout.write( str(stack.pop()) + "\n" )
Fout.close()
```

Вычисление арифметических выражений

Вы не задумывались, как компьютер вычисляет арифметические выражения, например $(5+15)/(4+7-1)$? Такая привычная для нас запись называется *инфиксной* — в ней знаки операций расположены между *операндами* (данными, участвующими в операции). Инфиксная форма неудобна для автоматических вычислений из-за того, что выражение содержит скобки и его нельзя вычислить за один проход слева направо.

В 1920 году польский математик Ян Лукасевич (*Jan Łukasiewicz*) предложил *префиксную* форму, которую стали называть польской нотацией. В ней знак операции расположен *перед* operandами. Например, выражение $(5+15)/(4+7-1)$ может быть записано в виде

$$/ + 5 \ 15 - + 4 \ 7 \ 1$$

Скобок здесь не требуется, так как порядок операций строго определён: сначала выполняются два сложения ($+ 5 \ 15$ и $+ 4 \ 7$), затем вычитание, и, наконец, деление. Первой стоит *последняя* выполняемая операция, это значит, что выражение нужно вычислять справа налево.

В середине 1950-х годов была предложена обратная польская нотация, или *постфиксная* форма записи, в которой знак операции стоит после operandов:

$$5 \ 15 + 4 \ 7 + 1 - /$$

В этом случае также не нужны скобки, и выражение может быть вычислено за один просмотр слева направо с помощью стека следующим образом:

- если очередной элемент — число (или переменная), он записывается в стек;
- если очередной элемент — операция, то она выполняется с верхними элементами стека (они снимаются со стека), после этого в стек добавляется результат выполнения этой операции.

На рисунке 1.7 показано, как работает этот алгоритм (стек «растёт» снизу вверх).

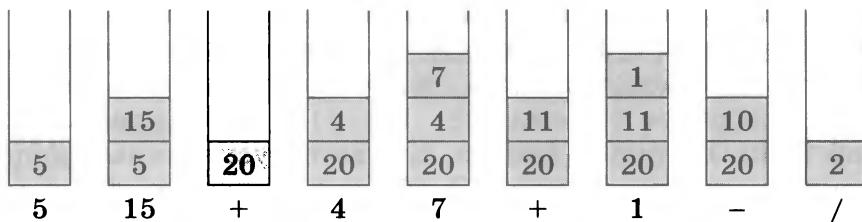


Рис. 1.7

В результате в стеке осталось значение заданного выражения.

Следующая программа вводит с клавиатуры символьную строку, в которой записано выражение в постфиксной форме, и вычисляет его:

```

data = input().split()                      # (1)
stack = []                                  # (2)
for x in data:                             # (3)
    if x in "+-*/":
        op2 = int( stack.pop() )            # (4)
        op1 = int( stack.pop() )            # (5)
        if x == "+": res = op1 + op2      # (7)
        elif x == "-": res = op1 - op2    # (8)
        elif x == "*": res = op1 * op2    # (9)
        else:             res = op1 // op2 # (10)
        stack.append( res )                # (11)
    else:
        stack.append( x )                  # (12)
print( stack[0] )                          # (13)

```

В строке 1 результат ввода разбивается на части (по пробелам) с помощью метода `split`, в результате получается список `data`, содержащий отдельные элементы постфиксной записи — числа и знаки арифметических действий.

В строке 2 создаётся пустой стек, в строке 3 в цикле перебираются все элементы списка. Если очередной элемент, попавший в переменную `x`, — это знак арифметической операции (строка 4), снимаем со стека два верхних элемента (строки 5–6), выполняем нужное действие (строки 7–10) и добавляем результат вычисления в стек (строка 11).

Если же очередной элемент — это число (не знак операции), просто добавляем его в стек (строка 12). В конце программы в стеке должен остаться единственный элемент — значение выражения, которое выводится на экран в строке 13.

Скобочные выражения

Пусть задана символьная строка, в которой записано некоторое арифметическое выражение, использующее скобки трёх типов: `()`, `[]` и `{}`. Нужно проверить, правильно ли расставлены скобки. Скобочным выражением будем называть последовательность скобок в исходном выражении.

Например, скобочное выражение `()[{{()}}[]]` — правильное, потому что каждой открывающей скобке соответствует закрывающая и вложенность скобок не нарушается. Скобочные выражения

$$[() \quad [[[() \quad [() } \quad) (\quad ([]]$$

неправильные. В первых трёх есть непарные скобки, а в последних двух не соблюдается вложенность скобок.

Начнём с более простой задачи, в которой используется только один вид скобок. Её можно решить с помощью счётчика скобок. Сначала счётчик равен нулю. Стока просматривается слева направо, если очередной символ — открывающая скобка, то счётчик увеличивается на 1, если закрывающая — уменьшается на 1. В конце просмотра счётчик должен быть равен нулю, если все скобки парные. Кроме того, во время просмотра он не должен становиться отрицательным (должна соблюдаться вложенность скобок).

В исходной задаче (с тремя типами скобок) хочется завести три счётчика и работать с каждым из них отдельно. Однако это решение неверное. Например, для скобочного выражения `({{[]}})` условия «правильности» выполняются отдельно для каждого вида скобок, но не для выражения в целом.

Задачи, в которых важна вложенность объектов, удобно решать с помощью стека. Нас интересуют только открывающие и закрывающие скобки, на остальные символы можно не обращать внимания.

Стока просматривается слева направо. Если очередной символ — открывающая скобка, будем добавлять на вершину стека соответствующую закрывающую скобку. Если это закрывающая скобка, то проверяем, что лежит на вершине стека: если там закрывающая скобка того же типа, то её нужно просто снять со стека. Если на вершине лежит скобка другого типа или стек пуст, выражение неверное, и нужно закончить просмотр. В конце обработки правильной строки стек должен быть пуст. Кроме того, во время просмотра не должно быть ошибок. Работа такого алгоритма для правильного скобочного выражения `([(){}])` иллюстрируется на рис. 1.8.

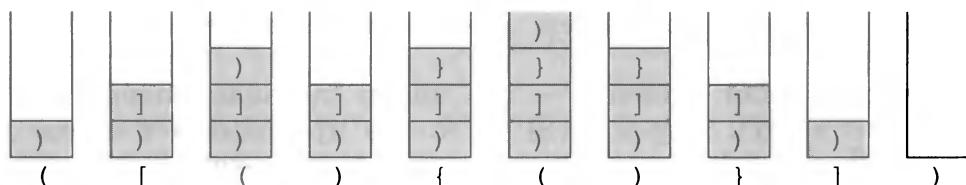


Рис. 1.8

В начале программе создадим пустой стек:

```
stack = []
```

Введём логическую переменную `valid` (в переводе с английского — правильный). Её значение будет равно `True` (истина), если выражение правильное, и `False` (ложь), если обнаружена ошибка. Вначале присваиваем ей значение `True`.

```
valid = True
```

Для работы со скобками удобно использовать словарь, который каждой открывающей скобке ставит в соответствие закрывающую:

```
pair = { "(": ")",
         "[": "]",
         "{": "}" }
```

В основном цикле перебираем все символы строки `s`, в которой записано скобочное выражение:

```
for c in s:                                # (1)
    if c in "([{":                           # (2)
        stack.append( pair[c] )                # (3)
    elif c in ")]}":
        if len(stack) == 0 or \
            c != stack.pop():                  # (5)
            valid = False                     # (6)
        break                                 # (7)
```

В ходе выполнения цикла `for c in s` все символы строки `s` по очереди, начиная с самого первого, оказываются в переменной `c`. Если очередной символ `c` — открывающая скобка, добавляем в стек соответствующую ей закрывающую скобку (строка 3).

Далее ищем символ `c` среди закрывающих скобок (строка 4). Если нашли, то в строке 5 проверяем два условия:

- стек пуст (длина массива `stack` равна нулю);
- текущий символ (в переменной `c`) не совпал с символом, который снят с вершины стека.

Если выполняется хотя бы одно из этих условий, выражение ошибочно. При этом в переменную `valid` в строке 6 записывается значение `False` (произошла ошибка) и происходит досрочный выход из цикла с помощью оператора `break` (строка 7).

Заметим, что условие `len(stack)==0` в строке 5 можно заменить на равносильное условие `not stack` (пустой стек воспринимается как ложное значение).

После окончания цикла нужно проверить содержимое стека. Если он пуст, то в выражении остались незакрытые скобки, и оно ошибочно:

```
if stack:
    valid = False
```

В конце программы остаётся вывести результат на экран:

```
if valid:
    print( "Выражение правильное." )
else:
    print( "Выражение неправильное." )
```

Как вызываются подпрограммы?

Стек играет очень важную роль при вызовах процедур и функций. Рассмотрим простую программу, в которой вызывается процедура `printLine` с одним параметром:

```
def printLine( n ): # определение процедуры
    print( "-"*n )
printLine( 10 )      # вызов процедуры
print( "Done!" )
```

Выполняемая программа хранится в памяти компьютера, которую можно рассматривать как последовательность байтов. Эти байты пронумерованы, самый первый байт памяти имеет номер 0. Каждая команда имеет свой адрес — номер байта в памяти, с которого она начинается. Например, на рис. 1.9 показано, что процедура `printLine` размещается по адресу 1172, вызов процедуры — по адресу 1212, а следующая команда — по адресу 1236 (все адреса условные).

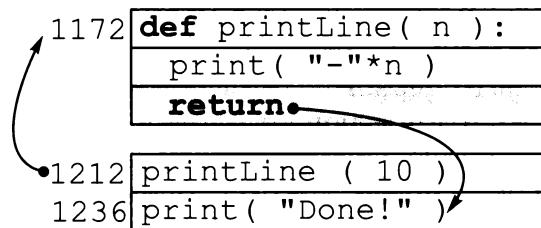


Рис. 1.9

Если в конце процедуры нет команды `return` (возврат из процедуры), транслятор добавит её автоматически.

При выполнении команды, расположенной по адресу 1212, происходит следующее:

- 1) управление передаётся процедуре, расположенной по адресу 1172;
- 2) выполняются все команды процедуры, пока не встретится оператор `return`;
- 3) при выполнении команды `return` управление передаётся на строку с адресом 1236.

Чтобы организовать такой вызов, необходимо:

- 1) передать процедуре аргумент — число 10;
- 2) запомнить адрес возврата 1236, чтобы затем использовать его при выполнении оператора `return`.

Очевидно, что для решения этих задач нужно использовать память. Причём это должна быть не та память, в которой размещается программа, а какая-то другая, способная временно хранить данные. Такая область памяти в компьютере называется системным стеком.

Системный стек — особая область памяти, в которой хранятся аргументы, локальные переменные и адреса возврата из подпрограмм.



Мы рассмотрим общие принципы вызова процедур и функций с помощью стека в любом языке программирования. Некоторые детали в каждом конкретном случае могут немного отличаться от этого описания.

Системный стек состоит из двух частей: первая уже заполнена данными, вторая свободна. Специальная переменная, которую мы будем обозначать символами *SP* (от англ. *stack pointer* — указатель стека), содержит адрес последней непустой ячейки стека.

Часть стека, заполненная на момент вызова процедуры, на рис. 1.10, *а* выделена серым фоном. Будем считать, что стек «растёт» вверх, т. е. новые данные добавляются сразу над заполненной частью (рис. 1.10, *б*).

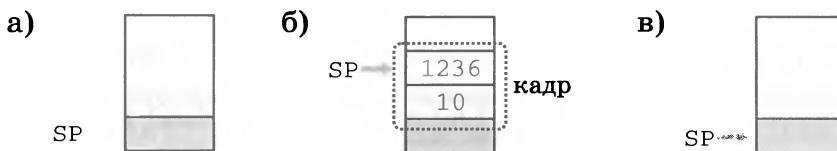


Рис. 1.10

При вызове процедуры *printLine* в стек добавляется сначала значение аргумента (число 10), а затем — адрес возврата 1236, по которому нужно передать управление после завершения работы процедуры (рис. 1.10, *б*). Таким образом, все данные, нужные процедуре, сохранены в стеке. Блок данных, используемый для вызова процедуры, называют *кадром стека* (англ. *stack frame*) или *записью активации* (англ. *activation record*).

Процедура во время работы может обращаться к ячейкам стека, в том числе и к значению параметра *n*, равному 10. Если в процедуре есть локальные переменные, они тоже размещаются в стеке, после адреса возврата.

Когда в процедуре выполняется оператор *return*, управление передаётся по адресу возврата 1236, который записан в стеке. Одновременно указатель стека возвращается в то состояние, которое было перед вызовом процедуры (рис. 1.10, *в*). Это фактически означает, что все вспомогательные данные, которые использовались для вызова процедуры, из стека удалены.

Обычно размер стека, который использует программа, ограничен. При каждом вложенном вызове подпрограммы в стеке размещается новый «кадр» с данными об этом вызове, и поэтому размер стека растёт. При многократных вложенных вызовах подпрограмм свободное место в стеке может закончиться, тогда произойдёт переполнение стека (англ. *stack overflow*). Особенно опасны с этой точки зрения рекурсивные процедуры и функции. Поэтому максимальная глубина рекурсии, как правило, ограничивается.

Очередь

Все мы знакомы с принципом очереди: «первым пришёл — первым обслужен» (англ. *FIFO: First In — First Out*). Соответствующая структура данных в информатике тоже называется очередью.



Очередь — это линейная структура данных, для которой введены две операции:

- добавление нового элемента в конец очереди (*append*);
- удаление первого элемента из очереди (*pop*).

Очередь можно представить себе в виде трубы, в которой элементы добавляются с одного конца, а удаляются с другого (рис. 1.11).

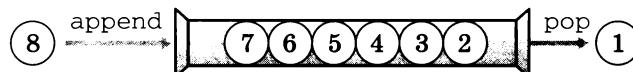


Рис. 1.11

Очередь — это не просто теоретическая модель. Операционные системы используют очереди для обмена данными между программами: каждая программа имеет свою очередь сообщений. Контроллеры жёстких дисков формируют очереди запросов ввода и вывода данных. В сетевых маршрутизаторах создаётся очередь из пакетов данных, ожидающих отправки.

Рассмотрим задачу заливки области рисунка, для решения которой мы применим очередь. Рисунок задан в виде матрицы A , в которой элемент $A[x][y]$ определяет цвет пикселя на пересечении столбца x и строки y . Для удобства записи мы храним матрицу *по столбцам*: первый индекс обозначает столбец, а второй — строку.

Требуется перекрасить в заданный цвет *одноцветную* область, начиная с пикселя с координатами (x_0, y_0) . Такую задачу решают программы, работающие с растровыми рисунками.

На рисунке 1.12 показан результат заливки области для матрицы размером 5×5 . Заливка выполняется цветом 2, начиная с точки $(1, 0)$.

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

(1,0)

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

Рис. 1.12

Один из возможных вариантов решения этой задачи использует очередь, элементы которой — координаты пикселей (точек):

```
# добавить в очередь точку (x0, y0)
# color = цвет точки (x0, y0)
while очередь не пуста:
    # взять из очереди точку (x, y)
    if A[x][y] == color:
        # A[x][y] = новый цвет
        # добавить в очередь точку (x-1, y)
        # добавить в очередь точку (x+1, y)
        # добавить в очередь точку (x, y-1)
        # добавить в очередь точку (x, y+1)
```

Конечно, в очередь добавляются только те точки, которые находятся в пределах рисунка (матрицы A). Заметим, что в этом алгоритме некоторые точки могут быть добавлены в очередь несколько раз (подумайте, когда это может случиться). Поэтому решение можно несколько улучшить, например, не добавлять точки в очередь повторно (попробуйте сделать это самостоятельно).

Пусть изображение записано в виде матрицы A, которая на языке Python представлена как список списков (каждый внутренний список — отдельный столбец матрицы). Определим размеры матрицы:

```
XMAX = len(A)
YMAX = len(A[0])
```

Значение XMAX — число столбцов, а YMAX — это число строк (длина одного столбца).

Зададим цвет заливки:

```
NEW_COLOR = 2
```

и координаты начальной точки, откуда начинается заливка:

```
x0, y0 = 1, 0
```

Запомним цвет области в переменной color:

```
color = A[x0][y0]
```

Нам нужно закрашивать новым цветом все точки цвета `color`, начиная с начальной точки.

Создадим очередь как список и добавим в эту очередь точку с начальными координатами. Две координаты точки связаны между собой, поэтому в программе лучше объединить их в единый объект, который в Python называется *кортежем* и заключается в круглые скобки. Таким образом, каждый элемент очереди — это кортеж из двух элементов:

```
Q = [ (x0, y0) ]
```

Кортеж очень похож на список (обращение к элементам также выполняется по индексу в квадратных скобках), но его, в отличие от списка, нельзя изменять.

Остается написать основной цикл:

```
while Q:                                # (1)
    x, y = Q.pop(0)                      # (2)
    if A[x][y] == color:                 # (3)
        A[x][y] = NEW_COLOR             # (4)
        if x > 0:          Q.append( (x-1, y) )
        if x < XMAX-1:     Q.append( (x+1, y) )
        if y > 0:          Q.append( (x, y-1) )
        if y < YMAX-1:     Q.append( (x, y+1) )
```

Цикл в строке 1 работает до тех пор, пока очередь не пуста.

Начало очереди всегда совпадает с первым по счёту элементом списка (имеющим индекс 0), поэтому в строке 2 мы как раз получаем первый элемент очереди (и удаляем его из очереди). Элемент очереди — это кортеж из двух значений, и мы сразу разбиваем его на отдельные координаты, которые записываются в переменные `x` и `y`.

Если цвет текущей точки совпадает с цветом начальной точки, который хранится в переменной `color` (строка 3), эта точка закрашивается новым цветом (строка 4), и в очередь добавляются все точки, граничащие с точкой (x, y) и попадающие на поле рисунка.

Дек

Существует ещё одна линейная динамическая структура данных, которая называется деком.

Дек (от англ. *dequeue*: *double ended queue*, двусторонняя очередь) — это линейная структура данных, в которой можно добавлять и удалять элементы как с одного, так и с другого конца.

Из этого определения следует, что дек может работать и как стек, и как очередь. С помощью дека можно, например, моделировать колоду игральных карт.

В языке Python для моделирования дека также удобно использовать список. Основные операции с деком d выполняются так:

- добавление элемента x в конец дека: `d.append(x)`;
- добавление элемента x в начало дека: `d.insert(0, x)` (добавляемый элемент будет иметь индекс 0);
- удаление элемента с конца дека: `d.pop()`;
- удаление элемента с начала дека: `d.pop(0)`.

Выводы

- Стек — это линейная структура данных, в которой добавление и удаление элементов разрешаются только с одного конца.
- С помощью стека удобно решать задачи, в которых важен порядок вложенности элементов (например, скобок). Одна из таких задач — вычисление арифметических выражений.
- При вызовах подпрограмм для временного хранения данных используется специальная область памяти — системный стек. В нём размещаются аргументы, локальные переменные и адреса возврата из подпрограмм.
- Кадр стека (запись активации) — это блок данных в стеке, используемый для вызова подпрограммы.
- Очередь — это линейная структура данных, для которой введены две операции: 1) добавление нового элемента в конец очереди; 2) удаление первого элемента из очереди.
- Дек — это линейная структура данных, в которой можно добавлять и удалять элементы как с одного, так и с другого конца. Дек можно использовать и как стек, и как очередь.

Вопросы и задания



1. Сравните стек, очередь и дек. Какая из этих структур данных наиболее общая (может выполнять функции других)?
2. Какие ошибки могут возникнуть при использовании стека, очереди, дека?
3. Закончите программу из параграфа, которая вычисляет значение арифметического выражения, записанного в постфиксной форме. Выражение вводится с клавиатуры в виде символьной строки. Предусмотрите сообщения об ошибках.
4. Напишите программу, которая вычисляет значение арифметического выражения, записанного в префиксной форме.
5. Напишите программу, которая проверяет правильность скобочного выражения с четырьмя видами скобок: (), [], {} и <>.
6. Дополните программу из предыдущего задания так, чтобы она определяла номер ошибочного символа в строке.
- *7. Как, на ваш взгляд, изменяется системный стек при выполнении такой программы?

```
def printStar( n ):
    print( "*" * n )
def printStarLine( n ):
    printStar( n )
    print( "—" * n )
    printStarLine( 10 )
    print("Done!")
```

Адреса процедур и операторов программы выберите сами, используя рис. 1.9 как образец.

*8. Как, на ваш взгляд, изменяется системный стек при выполнении программы с рекурсивной процедурой?

```
def printStar( n ):
    if n <= 0: return
    print( "*" * n )
    printStar( n-1 )
    printStar( 3 )
    print("Done!")
```

Адреса процедур и операторов программы выберите сами, используя рис. 1.9 как образец.

9. Закончите программу из параграфа, которая выполняет заливку одноцветной области заданным цветом. Матрица, содержащая цвета пикселей, вводится из файла. Затем с клавиатуры вводятся координаты точки заливки и цвет заливки. На экран нужно вывести матрицу, которая получилась после заливки.

*10. Напишите решение задачи о заливке области (из параграфа), в котором одни и те же точки не добавляются в очередь повторно. В чём преимущества и недостатки такого алгоритма?

*11. *Проект.* Напишите программу, которая применяет алгоритм заливки области (из параграфа) для поиска пути в лабиринте.

§ 9 Деревья

Ключевые слова:

- узел
- ребро
- вершина
- двоичное дерево
- дерево поиска
- ключ
- обход дерева
- обход в глубину
- обход в ширину

Что такое дерево?

Дерево — это структура данных, отражающая иерархию (отношения подчинённости, многоуровневые связи).

Дерево состоит из узлов (вершин) и связей между ними (рёбер или дуг¹⁾). Самый первый узел, расположенный на верхнем уровне (в него не входит ни одна дуга), — это корень дерева. Корень дерева на рис. 1.13 — это узел *A*.

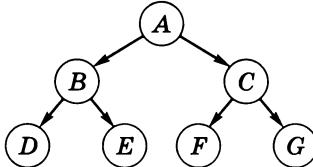


Рис. 1.13

Конечные узлы, из которых не выходит ни одна дуга, называются листьями. Листья дерева на рис. 1.13 — это узлы *D*, *E*, *F* и *G*.

Все остальные узлы, кроме корня и листьев, — это промежуточные узлы (*B* и *C* на рис. 1.13).

Из двух связанных узлов тот, который находится на более высоком уровне иерархии, называют «родителем», а другой (на более низком уровне) — «сыном» или дочерним узлом (англ. *child* — ребёнок). Корень — это единственный узел, у которого нет родителя; у листьев нет сыновей.

Используются также понятия «предок» и «потомок». Потомок какого-то узла — это узел, в который можно перейти по дугам от узла-предка, спускаясь вниз по иерархии. Соответственно, предок какого-то узла — это узел, из которого можно перейти по дугам в данный узел. Корень дерева — предок всех остальных узлов.

В дереве на рис. 1.13 родитель узла *E* — это узел *B*, а предки узла *E* — это узлы *A* и *B*, для которых узел *E* — потомок. Потомками узла *A* (корня дерева) являются все остальные узлы.

Высота дерева — это наибольшее расстояние (количество рёбер) от корня до листа. Высота дерева на рис. 1.13 равна 2.

Поддерево — эта часть дерева, которая тоже является деревом.

Дерево можно определить следующим образом:

- 1) пустая структура — это дерево;
- 2) дерево — это корень и несколько связанных с ним отдельных (не связанных между собой) деревьев (поддеревьев).



Здесь множество объектов (деревьев) определяется через само это множество на основе простого базового случая (пустого дерева). Таким образом, это рекурсивное определение, и получается, что дерево — это *рекурсивная структура данных*. Поэтому можно ожидать, что при работе с деревьями будут полезны рекурсивные алгоритмы.

Чаще всего в информатике используются *двоичные* (бинарные) деревья, т. е. такие, в которых каждый узел имеет не более двух сыновей. Их также можно определить рекурсивно.

¹⁾ Дугой называют ребро, имеющее направление.

Двоичное дерево:

- 1) пустая структура — это двоичное дерево;
- 2) двоичное дерево — это корень и два связанных с ним отдельных двоичных дерева («левое» и «правое» поддеревья).

В информатике деревья применяются во многих практических задачах:

- для поиска в большом массиве данных;
- при сортировке данных;
- для вычисления арифметических выражений;
- при оптимальном кодировании данных (метод сжатия Хаффмана).

Деревья поиска

Известно, что для того, чтобы найти заданный элемент в неупорядоченном массиве из N элементов, может понадобиться N сравнений. Теперь предположим, что элементы массива организованы в виде дерева, построенного специальным образом (рис. 1.14):

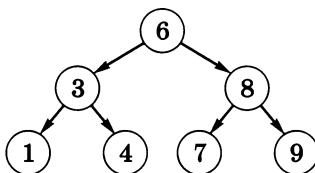


Рис. 1.14

Значения, связанные с узлами дерева, по которым выполняется поиск, называются *ключами* этих узлов (кроме ключа узел может содержать множество других данных). Перечислим важные свойства этого дерева:

- слева от каждого узла находятся узлы с меньшим ключом;
- справа от каждого узла находятся узлы, ключ которых больше или равен ключу данного узла.

Дерево, обладающее такими свойствами, называется *двоичным деревом поиска* (англ. *BST*: *binary search tree*).

Например, нужно найти узел, ключ которого равен 4. Начинаем поиск по дереву с корня. Ключ корня — 6 (больше заданного), поэтому дальше нужно искать только в левом поддереве, и т. д.

Скорость поиска наибольшая в том случае, если дерево *сбалансировано*, т. е. для каждой его вершины высота левого и правого поддеревьев различается не более чем на единицу. Если при линейном поиске в массиве за одно сравнение отсекается 1 элемент, здесь — сразу примерно половина оставшихся. Конечно, нужно учитывать, что предварительно дерево должно быть построено. Поэтому такой алгоритм выгодно применять в тех случаях, когда данные меняются редко, а поиск выполняется часто (например, в базах данных).

Обход дерева

Обойти дерево — это значит «посетить» все узлы по одному разу. Если перечислить узлы в порядке их посещения, мы представим данные в виде списка.

Существуют несколько способов обхода двоичного дерева:

- **КЛП** = «корень — левое — правое» (обход в прямом порядке):
 - посетить корень
 - обойти левое поддерево
 - обойти правое поддерево
- **ЛКП** = «левое — корень — правое» (симметричный обход):
 - обойти левое поддерево
 - посетить корень
 - обойти правое поддерево
- **ЛПК** = «левое — правое — корень»:
 - обойти левое поддерево
 - обойти правое поддерево
 - посетить корень

Как видим, это рекурсивные алгоритмы. Они должны заканчиваться без повторного вызова, когда текущий корень — пустое дерево.

Рассмотрим дерево, которое может быть составлено для вычисления арифметического выражения $(1 + 4) * (9 - 5)$ (рис. 1.15).

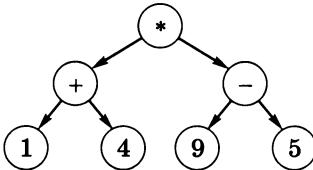


Рис. 1.15

Выражение вычисляется по такому дереву снизу вверх, т. е. корень дерева — это последняя выполняемая операция.

Различные типы обхода дают последовательность узлов:

КЛП: $* + 1 4 - 9 5$

ЛКП: $1 + 4 * 9 - 5$

ЛПК: $1 4 + 9 5 - *$

В первом случае (обход КЛП) мы получили *префиксную* форму записи арифметического выражения, во втором — привычную нам *инфиксную* форму (только без скобок), а в третьем — *постфиксную* форму. Напомним, что префиксная и постфиксная формы однозначно определяют порядок вычисления выражения, так что скобки здесь не нужны.

Обход КЛП называется *поиском в глубину* (англ. *DFS: depth-first search*), потому что сначала мы идём вглубь дерева по левым

поддеревьям, пока не дойдём до листа. Алгоритм обхода в глубину можно записать и без рекурсии, используя вспомогательный стек:

```
# записать в стек корень дерева
while стек не пуст:
    # снять узел V с вершины стека
    # посетить узел V
    if у узла V есть правый сын:
        # добавить в стек правого сына V
    if у узла V есть левый сын:
        # добавить в стек левого сына V
```

На рисунке 1.16 показано изменение состояния стека при таком обходе дерева, изображенного на рис. 1.15. Под стеком записана метка узла, который посещается (например, данные из этого узла выводятся на экран).

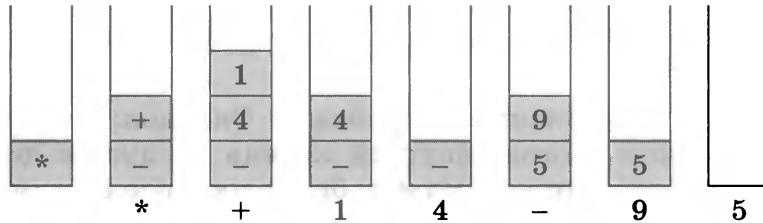


Рис. 1.16

Иногда применяется *поиск в ширину* (англ. *BFS: breadth-first search*): сначала посещаем корень, потом его сыновей, потом — сыновей сыновей («внуков») и т. д.

Для обхода в ширину используется очередь. Сначала помещаем в очередь корень дерева. На каждом шаге цикла выбираем узел из начала очереди, посещаем его, и добавляем в очередь всех его сыновей:

```
# записать в очередь корень дерева
while очередь не пуста:
    # выбрать первый узел V из очереди
    # посетить узел V
    if у узла V есть левый сын:
        # добавить в очередь левого сына V
    if у узла V есть правый сын:
        # добавить в очередь правого сына V
```

Использование связанных структур

Пока при изучении операций с деревьями мы записывали все операции на псевдокоде. Теперь попробуем составить программу на языке Python.

Поскольку двоичное дерево — это нелинейная структура данных, хранить данные в виде списка не очень удобно (хотя возможно). Вместо этого будем использовать связанные узлы. Каждый такой узел — это структура, содержащая три области: область данных, ссылку на левое поддерево и ссылку на правое поддерево. У листьев нет сыновей, в этом случае в указатели будем записывать специальное значение `None` (по-английски — «пусто», «ничто»). Дерево, состоящее из трёх таких узлов, показано на рис. 1.17.

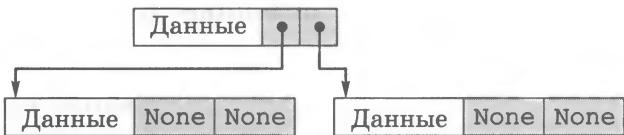


Рис. 1.17

Если мы строим дерево для вычисления арифметического выражения, область данных узла будет содержать одно поле — символьную строку, в которую записывается знак операции или число.

Будем хранить дерево в памяти как набор связанных структур-узлов класса `TNode` (по-английски *node* — узел):

```
class TNode:
    pass
```

Каждый узел содержит три поля: данные `d`, ссылку на левое поддерево `L` и ссылку на правое поддерево `R`.

Построим функцию, которая будет возвращать новую структуру этого класса:

```
def node(d, L = None, R = None):
    newNode = TNode()          # создаём новый узел
    newNode.data = d            # данные узла
    newNode.left = L            # левое поддерево
    newNode.right = R           # правое поддерево
    return newNode
```

Запись `L = None, R = None` в заголовке функции означает, что параметры `L` и `R` при вызове этой функции можно не задавать. Если эти параметры не заданы, им присваиваются значения по умолчанию, равные `None`. Функция `node` возвращает новый узел дерева.

Теперь мы можем построить в памяти дерево, показанное на рис. 1.14:

```
T = node("*",
         node("+", node("1"), node("4")),
         node("-", node("9"), node("5"))
     )
```

Обход дерева в глубину (обход КЛП) очень просто записать в виде рекурсивной процедуры:

```
def DFS( T ):
    if not T: return
    print( T.data, end=" " )
    DFS( T.left )
    DFS( T.right )
```

Такой же обход без рекурсии, с помощью стека:

```
def DFS_stack( T ):
    stack = [T]
    while stack:
        V = stack.pop()
        print( V.data, end=" " )
        if V.right:
            stack.append( V.right )
        if V.left:
            stack.append( V.left )
```

Процедура для **обхода в ширину** использует очередь вместо стека:

```
def BFS( T ):
    queue = [T]
    while queue:
        V = queue.pop(0) # первый элемент очереди
        print( V.data, end=" " )
        if V.left:
            queue.append( V.left )
        if V.right:
            queue.append( V.right )
```

Вычисление арифметических выражений

Один из способов вычисления арифметического выражения основан на его представлении в виде дерева.

Для простоты будем рассматривать только арифметические выражения, содержащие числа и знаки четырёх арифметических действий: $+-*/$.

Построим дерево для вычисления выражения $40 - 2 * 3 - 4 * 5$.

Так как корень дерева — это последняя операция, нужно сначала найти эту последнюю операцию, просматривая выражение слева направо. Здесь последнее действие — это второе вычитание, оно оказывается в корне дерева (рис. 1.18).

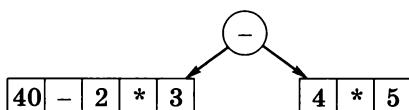


Рис. 1.18

Как выполнить этот поиск в программе? Известно, что операции выполняются в порядке *приоритета* (старшинства): сначала операции с более высоким приоритетом (слева направо), потом — с более низким (также слева направо). Отсюда следует важный вывод.

В корень дерева нужно поместить *последнюю* из операций с наименьшим приоритетом.



Теперь строим таким же способом левое и правое поддеревья (рис. 1.19).

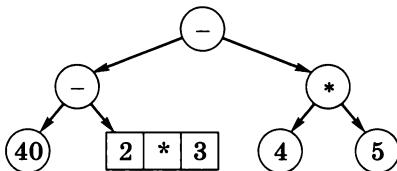


Рис. 1.19

Левое поддерево требует ещё одного шага (рис. 1.20).

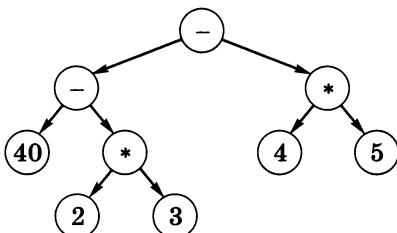


Рис. 1.20

Эта процедура рекурсивная, её можно записать в виде псевдокода:

```

# найти последнюю выполняемую операцию
if операций нет:
    # создать узел-лист
    return
# поместить найденную операцию в корень дерева
# построить левое поддерево
# построить правое поддерево
  
```

Рекурсия заканчивается, когда в оставшейся части строки нет ни одной операции, значит, там находится число (это лист дерева).

Теперь вычислим выражение по дереву. Если в корне находится знак операции, её нужно применить к результатам вычисления поддеревьев:

```
n1 = значение левого поддерева
n2 = значение правого поддерева
результат = операция( n1, n2 )
```

Снова получился рекурсивный алгоритм.

Возможен особый случай (на нём заканчивается рекурсия), когда корень дерева содержит число (т. е. это лист). Тогда это число и будет результатом вычисления выражения.

Напишем программу на языке Python, используя введённые выше класс TNode и функцию node. Пусть expr (от англ. *expression* — выражение) — символьная строка, в которой записано арифметическое выражение. Будем предполагать, что это правильное выражение без скобок.

Вычисление выражения сводится к двум вызовам функций:

```
T = makeTree( expr )
print( "Результат: ", calcTree(T) )
```

Функция makeTree строит в памяти дерево по строке expr, а функция calcTree вычисляет значение выражения по готовому дереву.

При построении дерева нужно искать последнюю выполняемую операцию. Для решения этой задачи мы напишем функцию lastOp. Она вернёт -1, если ни одной операции не будет обнаружено, в этом случае создаётся лист — узел без потомков.

Если операция найдена, в поле data нового узла записывается символ этой операции, а в поля left и right — адреса поддеревьев, которые строятся рекурсивно для левой и правой частей выражения:

```
def makeTree( expr ):
    pos = lastOp( expr )
    if pos < 0: # создать лист
        Tree = node( expr )
    else: # создать узел-операцию
        Tree = node( expr[pos] )
        Tree.left = makeTree( expr[:pos] )
        Tree.right = makeTree( expr[pos+1:] )
    return Tree
```

Функция calcTree (вычисление арифметического выражения по дереву) тоже будет рекурсивной:

```
def calcTree( Tree ):
    if not Tree.left:
        return int( Tree.data )
    else:
        n1 = calcTree( Tree.left )
        n2 = calcTree( Tree.right )
        return doOperation( Tree.data, n1, n2 )
```

Если ссылка на узел, переданная функции, указывает на лист (нет левого поддерева), то значение выражения — это результат преобразования числа из символьной формы в числовую (с помощью функции int).

Если функция получает ссылку на узел-операцию (у него есть левое и правое поддеревья), то вычисляются значения для обоих поддеревьев, и к ним применяется операция, указанная в корне дерева — эту работу выполняет функция doOperation:

```
def doOperation( op, n1, n2 ):
    if op == "+": return n1 + n2
    elif op == "-": return n1 - n2
    elif op == "*": return n1 * n2
    else:           return n1 // n2
```

Осталось написать функцию lastOp. Нужно найти в символьной строке последнюю операцию с минимальным приоритетом. Сначала составим функцию, возвращающую приоритет операции (для переданного ей символа):

```
def priority( op ):
    if op in "+-": return 1
    if op in "*/": return 2
    return 100
```

Сложение и вычитание имеют приоритет 1, умножение и деление — более высокий приоритет 2, а все остальные символы (не операции) — приоритет 100 (условное значение).

Функция lastOp может выглядеть так:

```
def lastOp( expr ):
    minPrt = 50      # любое число между 2 и 100
    pos = -1
    for i in range( len(expr) ):
        prt = priority( expr[i] )
        if prt <= minPrt:
            minPrt = prt
            pos = i
    return pos
```

Обратите внимание, что в условном операторе используется нестрогое неравенство (\leq), чтобы найти именно *последнюю* операцию с наименьшим приоритетом.

Начальное значение переменной minPrt можно выбрать любым между наибольшим приоритетом операций (2) и условным кодом неоперации (100). Если найдена любая операция, условие if выполнится, а если в строке нет операций, условие всегда ложно и в переменной pos остаётся начальное значение -1, которое и вернёт функция lastOp.

Цикл, в котором перебираются все элементы символьной строки, можно записать несколько иначе, «в стиле Python»:

```
for i, sym in enumerate( expr ):
    prt = priority( sym )
    if prt <= minPrt:
        minPrt = prt
        pos = i
```

Здесь функция `enumerate` перебирает все пары «индекс — символ». На каждой итерации цикла очередной индекс попадает в переменную `i`, а соответствующий ему символ `expr[i]` — в переменную `sym`.

Модульность

При разработке больших программ нужно разделить работу между программистами так, чтобы каждый делал свой независимый блок (*модуль*). Все подпрограммы, входящие в модуль, должны быть тесно связаны друг с другом, но как можно меньше связаны с подпрограммами других модулей.

В нашей программе в отдельный модуль можно вынести все операции с деревьями, т. е. определение класса `TNode` и все наши процедуры и функции. Назовём этот модуль `bintree` (от англ. *binary tree* — двоичное дерево) и сохраним в файле с именем `bintree.py`. Теперь в основной программе можно использовать этот модуль стандартным образом, загружая его целиком с помощью команды `import`:

```
import bintree
...
T = bintree.makeTree( expr )
print( "Результат: ", bintree.calcTree(T) )
```

При этом придётся вызывать все функции через точечную запись, указывая название модуля. Можно применить и другой способ импорта, выбрав только нужные нам функции:

```
from bintree import makeTree, calcTree
...
T = makeTree( expr )
print( "Результат: ", calcTree(T) )
```

В этом случае при обращении к функции название модуля указывать не нужно.

Обратите внимание, что нам не обязательно знать, как именно устроены функции `makeTree` и `calcTree`, какие типы данных они используют, по каким алгоритмам работают и какие дополнительные функции вызывают. Для применения функций из модуля достаточно

знать *интерфейс* — соглашение о передаче параметров (какие параметры принимают функции и какие результаты они возвращают).

Модуль в чём-то подобен айсбергу: видна только надводная часть (интерфейс), а значительно более весомая подводная часть скрыта. За счёт этого все, кто используют модуль, могут не думать о том, как именно он выполняет свою работу. Это один из приёмов, которые позволяют справляться со сложностью разработки больших программ. Разделение программы на модули облегчает понимание и совершенствование программы, потому что каждый модуль можно разрабатывать, изучать и оптимизировать независимо от других модулей.

Выводы

- Дерево — это структура данных, которая моделирует иерархию (многоуровневую структуру). Как правило, дерево определяется рекурсивно, поэтому для его обработки удобно использовать рекурсивные алгоритмы.
- В двоичном (бинарном) дереве каждый узел имеет не более двух сыновей.
- Деревья используются в задачах поиска, сортировки, для вычисления арифметических выражений и оптимального кодирования.
- Существует несколько способов обхода всех узлов дерева, наиболее известные из них — обход в глубину и обход в ширину.
- Дерево может храниться в памяти как набор связанных структур. Каждая такая структура содержит данные узла и ссылки на другие связанные с ней структуры (узлы-сыновья).

Вопросы и задания



1. Используя дополнительные источники, выясните, где используются структуры типа «дерево» в информатике и в других областях.
2. Объясните рекурсивное определение дерева. Почему его можно назвать рекурсивным?
3. Как вы думаете, почему рекурсивные алгоритмы работы с деревьями получаются проще, чем нерекурсивные?
4. Можно ли считать, что линейный список — это частный случай дерева? Ответ обоснуйте.
5. Какими свойствами должно обладать дерево поиска?
6. Как указать, что узел двоичного дерева не имеет левого (правого) сына?
7. Как можно было бы закодировать в дереве для арифметического выражения применение функций (например `abs`, `sin`, `cos`, `sqrt`)?

- *8. Найдите в дополнительных источниках (или предложите самостоятельно) алгоритм, с помощью которого можно построить дерево поиска из массива данных, которые поступают на вход программы.
9. Сравните поиск с помощью дерева:
- с линейным поиском в массиве;
 - с двоичным поиском в массиве.
10. Какие способы обхода дерева вы знаете? Придумайте какие-нибудь другие способы обхода.
11. Напишите модуль, который содержит процедуры для обхода дерева всеми рассмотренными способами.
12. При построении дерева для арифметического выражения Илларион решил искать *первую* операцию с наименьшим приоритетом (а не последнюю). Приведите пример, когда его программа неверно вычислит значение выражения.
13. Соберите программу, которая вводит и вычисляет арифметическое выражение без скобок. Все операции с деревом вынесите в отдельный модуль.
- *14. *Проект.* Усовершенствуйте программу из предыдущего задания так, чтобы она могла вычислять выражения со скобками.
- *15. *Проект.* Включите в вашу программу обработку некоторых ошибок (например, два знака операций подряд). Поработайте в парах: обменяйтесь программами с соседом и попробуйте найти выражение, при котором программа соседа завершится аварийно (и не выдаст собственное сообщение об ошибке).

§ 10 Графы

Ключевые слова:

- вершина
- ребро
- матрица смежности
- весовая матрица
- орграф
- жадный алгоритм
- оствовное дерево
- кратчайший маршрут
- алгоритм Дейкстры
- алгоритм Флойда–Уоршелла
- список смежности
- количество путей

Что такое граф?

Давным-давно жители немецкого города Кёнигсберга придумали задачу: пройти по всем семи мостам города, не проходя дважды ни по одному из них (рис. 1.21, а).

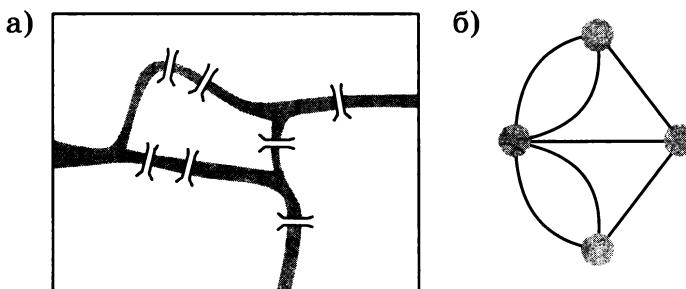


Рис. 1.21

В 1736 году задачу удалось решить математику Леонарду Эйлеру: он доказал, что это невозможно. И попутно открыл новое направление в математике — *теорию графов*.

Эйлер представил задачу в виде схемы (рис. 1.21, б), на которой участки суши обозначались как точки (вершины), а мосты — как линии связей между ними (ребра).

Граф — это набор вершин (узлов) и связей между ними — рёбер.



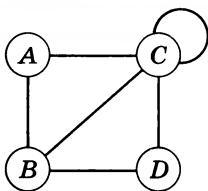
Для каждой вершины можно подсчитать количество ребёр, с которыми она связана. Это число называется *степенью вершины*.

Эйлер доказал, что задача с мостами разрешима, если все вершины имеют чётные степени (в этом случае прогулка завершается в той же вершине, откуда началась), или же ровно две вершины имеют нечётную степень (тогда нужно начать путь в одной из таких вершин и закончить в другой). Как видно из рис. 1.21, все четыре вершины в нашем граfe имеют нечётные степени, поэтому задача неразрешима.

Так как каждое ребро графа имеет два конца, то сумма степеней всех вершин графа — всегда чётное число: оно равно удвоенному количеству рёбер. Отсюда следует, что число вершин графа, имеющих нечётную степень, должно быть чётно (иначе сумма степеней вершин окажется нечётной). Эти результаты известны как *лемма о рукопожатиях*. Такое название связано с математической задачей: доказать, что в любой компании число людей, пожавших руку нечётному числу других людей, чётно.

Как описать граф?

Данные о связях между вершинами графа можно записать в виде *матрицы смежности*: таблицы, в которой единица на пересечении строки *A* и столбца *B* означает, что существует дуга из вершины *A* в вершину *B*, а ноль указывает на то, что такой дуги нет (рис. 1.22).



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	1	1
D	0	1	1	0

Рис. 1.22

Матрица смежности симметрична относительно своей главной диагонали (эти клетки в таблице выделены фоном). Единица на главной диагонали обозначает *петлю* — ребро, которое начинается и заканчивается в одной и той же вершине. В графе на рис. 1.22 есть петля в вершине C.

Граф можно описать иначе: для каждого узла перечислить все узлы, с которыми связан данный узел. В этом случае мы получим *список смежности*. Для рассмотренного графа список смежности выглядит так:

```
[ A: [B, C],  
  B: [A, C, D],  
  C: [A, B, C, D],  
  D: [B, C] ]
```

Граф — это не рисунок, а математический объект. Конечно, его можно нарисовать на плоскости, но матрица смежности (и список смежности) не дают никакой информации о том, как именно располагать узлы друг относительно друга. Для графа на рис. 1.22 возможны, например, ещё такие варианты (рис. 1.23).

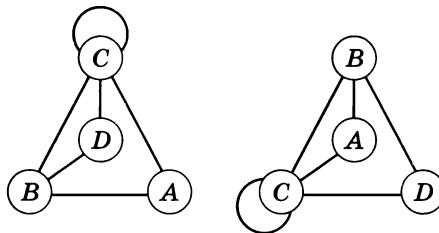


Рис. 1.23

В примере на рис. 1.22 все узлы связаны, т. е. между любой парой вершин существует *путь* — последовательность рёбер, по которым можно перейти из одной вершины в другую. Такой граф называется *связным*.

Вспоминая материал предыдущего параграфа, можно сделать вывод, что дерево — это связный граф, в котором нет замкнутых путей — *циклов*.

Если для каждого ребра указано направление, граф называют *ориентированным* (или *орграфом*). Рёбра орграфа называют *дугами*. Его матрица смежности не всегда симметрична. Единица на пересечении

строки A и столбца B , говорит о том, что существует дуга из вершины A в вершину B , а ноль означает, что такой дуги нет (рис. 1.24).

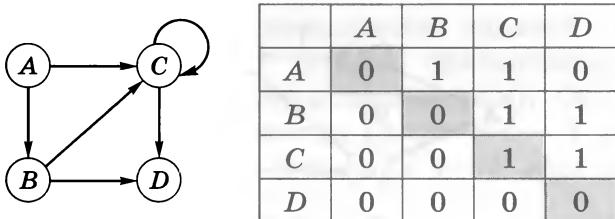


Рис. 1.24

Часто с каждым ребром связывают некоторое число — *вес ребра*. Это может быть, например, расстояние или стоимость проезда между городами. Такой граф называется *взвешенным*. Информация о взвешенном графе хранится в виде *весовой матрицы*, содержащей веса рёбер (рис. 1.25).

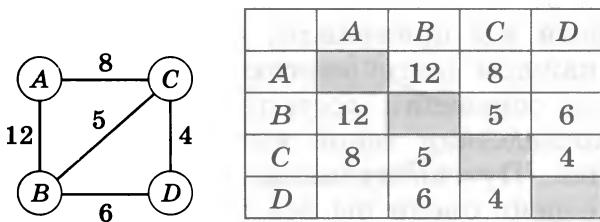


Рис. 1.25

У взвешенного орграфа весовая матрица может быть несимметрична относительно главной диагонали (рис. 1.26).

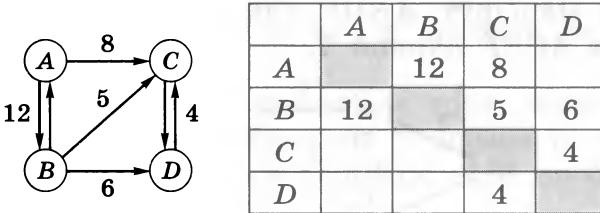


Рис. 1.26

Если связи между двумя вершинами нет, на бумаге можно оставить ячейку таблицы пустой, а при хранении в памяти компьютера записывать в неё условный код, например 0, -1 или очень большое число (∞), в зависимости от решаемой задачи.

«Жадные» алгоритмы

Одна из самых важных практических задач, которые решаются с помощью теории графов, — это задача определения кратчайшего расстояния (и кратчайшего маршрута) между двумя вершинами. Эту задачу, например, приходится решать в программах-навигаторах, которые строят оптимальный маршрут.

Пусть известна схема дорог между некоторыми городами. Числа на схеме (рис. 1.27) обозначают расстояния (дороги не прямые, поэтому неравенство треугольника может нарушаться).

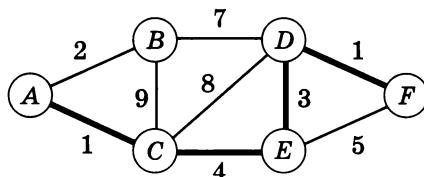


Рис. 1.27

Нужно найти кратчайший маршрут из города A в город F .

Первая мысль, которая приходит в голову, — на каждом шаге ехать в ближайший соседний город, в котором мы ещё не были. Для схемы на рис. 1.27 сначала из A едем в город C (длина 1), далее — в E (длина 4), затем в D (длина 3) и, наконец, в F (длина 1). Общая длина маршрута равна 9.

Алгоритм, который мы применили, называется жадным. Он состоит в том, чтобы на каждом шаге многоходового процесса выбирать наилучший вариант на основании доступной в данный момент информации. При этом впоследствии такой выбор может привести к худшему итоговому решению. Путешественники, применяющие жадный алгоритм, в первый же день съели бы всё самое вкусное, а в конце похода питались бы одними сухарями.

Для схемы на рис. 1.27 жадный алгоритм на самом деле даёт *оптимальное* (самое лучшее) решение, но так будет далеко не всегда. Например, для той же схемы с другими весами (рис. 1.28) жадный алгоритм выберет маршрут $ABDF$ длиной 10, хотя существует более короткий маршрут $ACEF$ длиной 7.

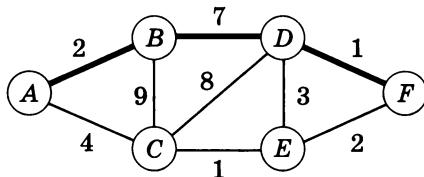


Рис. 1.28



Жадный алгоритм не всегда позволяет получить оптимальное решение.

Минимальное остовное дерево

Однако есть задачи, в которых жадный алгоритм всегда приводит к правильному решению. Одна из таких задач формулируется так.

В некоторой стране есть N городов, которые нужно соединить линиями связи. Между какими городами нужно проложить линии связи, чтобы все города были связаны в одну систему и общая длина линий связи была наименьшей?

Эту задачу называют *задачей Прима–Краскала* в честь математиков Р. Прима и Д. Краскала, которые независимо предложили и решили её в середине XX века.

В теории графов эта задача называется задачей построения *минимального остовного дерева* (т. е. дерева, связывающего все вершины, в котором сумма весов всех рёбер минимальна). Остовное дерево для связного графа с N вершинами всегда имеет $N - 1$ ребро.

Рассмотрим жадный алгоритм решения этой задачи, предложенный Краскалом:

- 1) начальное дерево — пустое;
- 2) на каждом шаге к дереву добавляется ребро минимального веса, которое ещё не было выбрано и не приводит к появлению цикла (замкнутого маршрута).

На рисунке 1.29 показано минимальное остовное дерево для одного из рассмотренных выше графов (сплошные жирные линии).

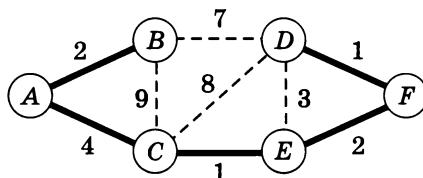


Рис. 1.29

Здесь возможна такая последовательность добавления рёбер: CE , DF , AB , AC . Обратите внимание, что после добавления ребра EF следующее «свободное» ребро минимального веса — это DE (длина 3), но оно образует цикл с рёбрами DF и EF , и поэтому не было включено в дерево.

При программировании этого алгоритма сразу возникает вопрос: как определить, что ребро ещё не включено в дерево и не образует цикла в нём? Существует очень красивое решение этой проблемы, основанное на раскраске вершин.

Сначала все вершины раскрашиваются в разные цвета, это значит, что все рёбра из графа удаляются. Таким образом, мы получаем множество (так называемый *лес*) элементарных деревьев, каждое из которых состоит из одной вершины.

Затем последовательно объединяем отдельные деревья, каждый раз выбирая ребро минимальной длины, соединяющее разные деревья (выкрашенные в разные цвета). Объединённое дерево перекрашивается в один цвет, совпадающий с цветом одного из вошедших в него поддеревьев. В конце концов все вершины оказываются выкрашенными в один цвет. Это значит, что все они принадлежат одному дереву. Полученное таким образом дерево будет иметь минимальный вес, если на каждом шаге выбирать подходящее ребро минимальной длины.

Пусть информация о графе хранится в виде весовой матрицы W размера $N \times N$ (индексы строк и столбцов начинаются с 0). Если связи между вершинами i и j нет, в элементе $W[i][j]$ этой матрицы будем хранить «условную бесконечность» — число, намного большее, чем длина любого ребра.

Для графа на рис. 1.29 весовая матрица может быть задана следующим образом:

```
N = 6
INF = 30000 # "бесконечность", нет связей
W = [
    [0,      2,      4,  INF,  INF,  INF],
    [2,      0,      9,      7,  INF,  INF],
    [4,      9,      0,      8,      1,  INF],
    [INF,     7,      8,      0,      3,      1],
    [INF,  INF,     1,      3,      0,      2],
    [INF,  INF,  INF,     1,      2,      0]
]
```

В программе сначала присвоим всем вершинам разные числовые коды (разные «цвета»):

```
col = [ i for i in range(N) ]
```

Здесь N — количество вершин графа, а массив col (список из N элементов) хранит «цвета» вершин — значения от 0 до $N - 1$.

На каждом шаге цикла нужно найти самое короткое ребро, которое соединяет вершины разных «цветов» (это значит, что оно ещё не выбрано). Для этого используем вложенный цикл, который перебирает все возможные пары вершин¹⁾:

```
minDist = 1e10          # очень большое число
for i in range(N):
    for j in range(N):
        if col[i] != col[j] and W[i][j] < minDist:
            iMin = i
            jMin = j
            minDist = W[i][j]
```

Каждое ребро определяется парой вершин, которые оно соединяет. После выполнения этого цикла номера вершин, которые связывает найденное ребро минимальной длины, будут записаны в переменные $iMin$ и $jMin$.

Выбранные рёбра будем добавлять в список $ostov$. Из номеров вершин для каждого ребра можно составить *кортеж* (неизменяемый

¹⁾ В дополнительных источниках можно найти более эффективный алгоритм выбора рёбер, использующий очередь с приоритетами.

массив). Такие кортежи и будем записывать в список `ostov`. Сначала он пуст:

```
ostov = []
```

Когда будет найдено очередное ребро, составим кортеж из значений переменных `iMin` и `jMin` и запишем его в список `ostov`:

```
ostov.append( (iMin, jMin) )
```

После этого нужно «перекрасить» все вершины, имеющие тот же цвет, что и вершина `jMin`, в цвет вершины `iMin` — таким образом мы соединяем два дерева в одно только что выбранным ребром:

```
c = col[jMin]
for i in range(N):
    if col[i] == c:
        col[i] = col[iMin]
```

Эти операции (выбор подходящего ребра и включение его в дерево) нужно выполнить ровно $N-1$ раз: именно столько рёбер входит в оставшееся дерево). Приведём полностью основной цикл:

```
ostov = []
for k in range(N-1):
    # поиск ребра с минимальным весом
    minDist = 1e10          # очень большое число
    for i in range(N):
        for j in range(N):
            if col[i] != col[j] and W[i][j] < minDist:
                iMin = i
                jMin = j
                minDist = W[i][j]
    # добавление ребра в список выбранных
    ostov.append( (iMin, jMin) )
    # перекрашивание вершин
    c = col[jMin]
    for i in range(N):
        if col[i] == c:
            col[i] = col[iMin]
```

После окончания цикла остаётся вывести результат — рёбра из массива `ostov`:

```
for edge in ostov:
    print( "(", edge[0], ", ", edge[1], ")" )
```

В цикле перебираются все элементы списка `ostov`; каждый из них — кортеж из двух элементов — попадает в переменную `edge`, так что номера вершин определяются как `edge[0]` и `edge[1]`.

Алгоритм Дейкстры

В начале параграфа мы увидели, что в задаче выбора кратчайшего маршрута жадный алгоритм не всегда даёт лучшее решение. В 1960 году нидерландский учёный Эдсгер Дейкстра предложил алгоритм, позволяющий найти все кратчайшие маршруты из одной вершины графа во все остальные. Его метод работает при условии, что длины всех рёбер (расстояния между вершинами) положительные. К счастью, это условие верно для большинства реальных задач.

Основная идея состоит в следующем: если сумма весов $W[x][z] + W[z][y]$ меньше, чем вес $W[x][y]$, то из вершины X лучше «ехать» в вершину Y не напрямую, а через вершину Z (рис. 1.30).

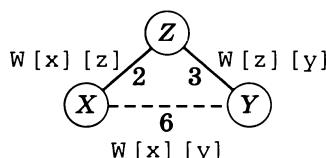


Рис. 1.30

Рассмотрим уже знакомую схему, для которой не сработал жадный алгоритм (рис. 1.31).

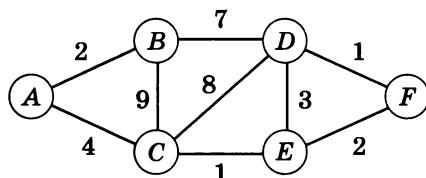


Рис. 1.31

Вершины, минимальные расстояния до которых мы уже определили, будем как-то отмечать («красить»). Сначала ни одна вершина не отмечена, длины кратчайших маршрутов до всех вершин неизвестны (можно считать, что они равны условной «бесконечности»).

Понятно, что кратчайший маршрут из вершины A в саму эту вершину имеет длину 0. Отмечаем стартовую вершину A . Из этой вершины можно попасть напрямую в вершины B и C , т. е. мы уже знаем, что есть маршрут из A в B длиной 2 и маршрут из A в C длиной 4; записываем эти числа около вершин B и C (рис. 1.32).

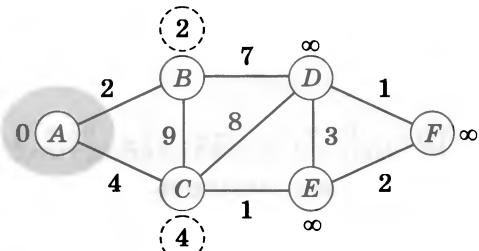


Рис. 1.32

Около остальных вершин (D , E и F) записываем знак ∞ («бесконечность»): они недостижимы напрямую из A .

Теперь из всех «непокрашенных» вершин выбираем такую, расстояние до которой от покрашенной вершины A минимально: в нашем примере это вершина B (длина ребра AB равна 2).

Поскольку расстояние от A до остальных вершин не меньше 2 и длины всех ребер неотрицательны, длина любого маршрута из A в B через другие вершины будет не меньше 2. Следовательно, мы нашли длину кратчайшего маршрута из A в B , поэтому отмечаем («красим») вершину B , длина кратчайшего маршрута из A в B равна 2, она уже не изменится (рис. 1.33).

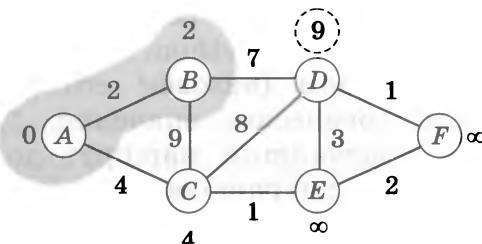


Рис. 1.33

Здесь серым фоном выделена уже обработанная часть графа: кратчайшие расстояния от вершины A до всех остальных вершин внутри этой области уже известны.

Проверяем, не удастся ли сократить путь в другие вершины, если «ехать» через B . Для вершины C это не получается — нам уже известен маршрут длиной 4, а маршрут через B даёт $2 + 9 = 11$. А вот для вершины D можно улучшить результат: до этого ни одного маршрута не было известно (условно расстояние равно «бесконечности»), а теперь найден маршрут ABD длиной $2 + 7 = 9$ (см. рис. 1.33).

Проверяем, какие вершины доступны из уже отмеченных вершин A и B — это вершины C и D . Ближайшая из них — вершина C , расстояние до неё от вершины A равно 4. Отмечаем вершину C , включаем её в обработанную часть графа и находим новый маршрут из A в E длиной 5 (ACE) (рис. 1.34).

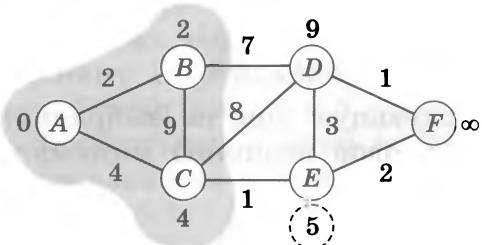


Рис. 1.34

Затем по такому же принципу выбираем вершину E (длина маршрута из A в E равна 5), это позволяет сократить длину маршрута из

A в D с 9 до 8 (ACED) и найти маршрут из A в F длиной 7 (ACEF) (рис. 1.35).

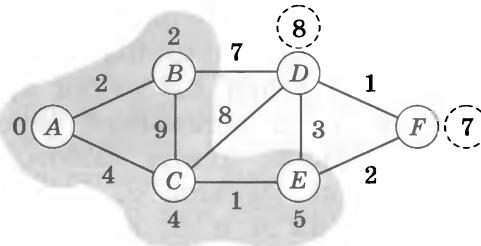


Рис. 1.35

После подключения вершин *D* и *F* результаты не изменяются.

Напишем программу на языке Python. Нам будут нужны два вспомогательных массива. В одном (назовём его *selected*) для каждой вершине будем хранить логическое значение: *True* («истина»), если она «покрашена» (т. е. кратчайший маршрут до неё уже определён), и *False* («ложь»), если не «покрашена». Во второй массив с именем *dist* будем записывать длины найденных на данный момент кратчайших маршрутов для всех вершин. Сначала все вершины не выбраны и все маршруты неизвестны:

```
INF = 30000
selected = [False]*N    # все вершины не выбраны
dist = [INF]*N          # все маршруты неизвестны
```

Выбираем стартовую вершину — вершину с номером 0, записываем её номер в переменную *V*:

```
start = 0
dist[start] = 0
V = start
```

На каждом шаге алгоритма нужно выполнить два действия. Во-первых, улучшить (если возможно) длины маршрутов, используя пути, проходящие через вершину *V*:

```
selected[V] = True
for j in range(N):
    if dist[V] + W[V][j] < dist[j]:
        dist[j] = dist[V] + W[V][j]
```

Во-вторых, найти следующую ещё не выбранную вершину *V*, для которой расстояние от стартовой вершины минимально:

```
minDist = 1e10 # большое число
for j in range(N):
    if not selected[j] and dist[j] < minDist:
        minDist = dist[j]
        V = j
```

Будем считать, что количество вершин N и весовая матрица W размером $N \times N$ определены ранее. Приведём полностью программу поиска оптимальных маршрутов из вершины A с индексом 0 во все остальные вершины:

```

INF = 30000
selected = [False]*N
dist = [INF]*N
start = 0
dist[start] = 0

V = start
minDist = 0
while minDist < INF:
    selected[V] = True
    # проверка маршрутов через вершину V
    for j in range(N):
        if dist[V] + W[V][j] < dist[j]:
            dist[j] = dist[V] + W[V][j]
    # поиск новой рабочей вершины: min dist[j]
    minDist = 1e10
    for j in range(N):
        if not selected[j] and dist[j] < minDist:
            minDist = dist[j]
            V = j

```

После завершения работы цикла в массиве $dist$ будут записаны длины оптимальных (кратчайших) маршрутов из вершины A во все вершины.

Остаётся только найти сами оптимальные маршруты (последовательности вершин). Эту задачу можно решить «обратным ходом» благодаря тому, что массив $dist$ и весовая матрица W содержат всю необходимую для этого информацию.

Например, найдём оптимальный маршрут из вершины A в вершину F . Как мы видели, его длина равна 7 (см. рис. 1.34).

Вершина F связана с двумя вершинами — D и E . Если бы маршрут проходил через вершину D , его длина была бы равна $8 + 1 = 9$, а маршрут через вершину E даёт как раз $5 + 2 = 7$, поэтому в F мы попадаем именно из E . В программе нужно перебрать все рёбра, связывающие вершину F с остальными, и выбрать такое ребро с номером i , для которого

```
dist[i]+W[i][V] == dist[V],
```

где V — номер вершины F . Эта операция повторяется, пока мы не дойдём до стартовой вершины:

```

V = N-1
print(V)
while V != start:
    for i in range(N):
        if i != V and dist[i]+W[i][V] == dist[V]:
            V = i
            break
    print(V)

```

Эта программа выводит последовательность вершин оптимального маршрута в обратном порядке.

Оценим сложность алгоритма Дейкстры. Основной цикл выполняется $N - 1$ раз, причём при каждой итерации этого цикла мы дважды просматриваем все N вершин. Поэтому алгоритм имеет квадратичную асимптотическую сложность $O(N^2)$.

Алгоритм Флойда–Уоршелла

Теперь рассмотрим более общую задачу: найти все кратчайшие маршруты из каждой вершины во все остальные. Как мы видели, алгоритм Дейкстры находит все кратчайшие пути только из одной заданной вершины.

Конечно, можно было бы применить этот алгоритм N раз, для каждой вершины, но существует более красивый метод — *алгоритм Флойда–Уоршелла*, основанный на той же самой идее сокращения маршрута (англ. *relaxation* — ослабление): иногда бывает короче ехать через промежуточные вершины, чем напрямую.

Весовая матрица графа W показывает только прямые связи между вершинами. Возьмём первую вершину (с индексом 0) и проверим для каждого ребра (i, j) , не будет ли короче маршрут из вершины i в вершину j , проходящий через промежуточную вершину 0 (рис. 1.36).

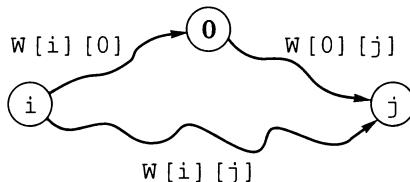


Рис. 1.36

Если сумма длин кратчайших маршрутов из вершины i в вершину 0 и из вершины 0 в вершину j окажется меньше, чем длина ребра $W[i][j]$, использование объезда через вершину 0 позволит сократить путь. Новую длину оптимального маршрута запоминаем в той же матрице W .

Если мы таким же образом последовательно «подключим» все N вершин, проверяя идущие через них обходные маршруты, в матрице W

окажутся длины кратчайших путей между всеми вершинами графа. На языке Python этот алгоритм записывается так:

```
for k in range(N):      # подключаем вершину k
    for i in range(N):
        for j in range(N):
            if W[i][k] + W[k][j] < W[i][j]:
                W[i][j] = W[i][k] + W[k][j]
```

Алгоритм Флойда–Уоршелла включает три вложенных цикла, каждый из которых выполняется N раз, поэтому его асимптотическая сложность — $O(N^3)$. Но, по сравнению с алгоритмом Дейкстры, он позволяет найти кратчайшие маршруты сразу для всех вершин. Если бы мы решали ту же задачу, используя N раз алгоритм Дейкстры, то сложность такого алгоритма тоже оценивалась бы как $O(N^3)$.

Заметим, что алгоритм Флойда–Уоршелла, в отличие от алгоритма Дейкстры, работает даже для графов, в которых веса рёбер могут быть отрицательными. Запрещены только циклы отрицательного веса (обход которых даёт отрицательный суммарный вес рёбер) — в этом случае кратчайших маршрутов может вообще не существовать.

Использование списков смежности

Поскольку в Python есть встроенный тип данных «список», при программировании на этом языке очень удобно использовать описание графа с помощью *списков смежности*.

Список смежности для какой-то вершины — это множество соседних вершин, в которые можно попасть из данной вершины по рёбрам (учитывая направления рёбер в орграфе). Рассмотрим орграф на рис. 1.37, состоящий из 5 вершин.

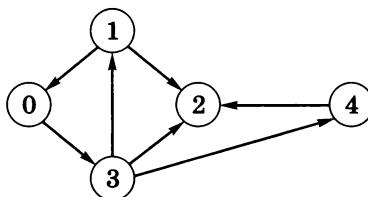


Рис. 1.37

Для удобства вершины нумеруются с нуля, как и элементы списка в Python. Списки смежности для вершин выглядят так:

- вершина 0: (3)
- вершина 1: (0, 2)
- вершина 2: ()
- вершина 3: (1, 2, 4)
- вершина 4: (2)

Граф, показанный на рис. 1.37, описывается в виде вложенного списка так:

```
graph = [ [3],           # список для вершины 0
          [0,2],         # список для вершины 1
          [],            # список для вершины 2
          [1,2,4],       # список для вершины 3
          [2] ]          # список для вершины 4
```

Используя такое представление, построим функцию `pathCount`, которая находит количество путей N_{XY} из одной вершины (X) в другую (Y), не содержащих циклов.

Алгоритм решения задачи основан на следующей идее (рис. 1.38).

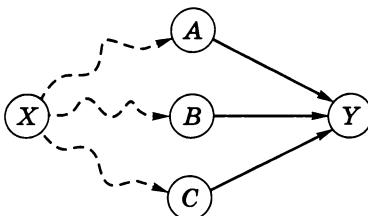


Рис. 1.38

Пусть в конечную вершину Y можно перейти только из вершин A , B или C , и мы уже определили количество маршрутов из X в эти вершины: N_{XA} , N_{XB} и N_{XC} . Тогда общее количество маршрутов из вершины X в вершину Y можно найти по формуле:

$$N_{XY} = N_{XA} + N_{XB} + N_{XC}.$$

Чтобы избежать циклов, в этой сумме нужно учитывать количество маршрутов только из тех вершин, которые ещё не посещались. Список посещённых вершин будем хранить в переменной `visited`.

Таким образом, в функцию `pathCount` нужно передать следующие данные (аргументы):

- описание графа в виде списков смежности всех вершин `graph`;
- номера начальной и конечной вершин `vStart` и `vEnd`;
- список посещённых вершин `visited`.

Её заголовок должен выглядеть так:

```
def pathCount( graph, vStart, vEnd, visited ):
    ...
```

Сначала стартовую вершину нужно добавить в список посещённых вершин:

```
visited.append( vStart )
```

Затем в цикле перебираем все вершины, в которые можно перейти из стартовой вершины. Для каждой ещё не посещённой вершины счи-

таем количество путей из стартовой вершины и суммируем эти значения в переменной `count`:

```
count = 0
for v in graph[vStart]:
    if not v in visited:
        count += pathCount( graph, v, vEnd, visited )
```

Затем убираем стартовую вершину из списка посещённых:

```
visited.pop()
```

Вы, конечно, заметили, что функция `pathCount` получилась *рекурсивной*: она вызывает сама себя в теле цикла. Поэтому нужно определить условие окончания рекурсии: если начальная и конечная вершины совпадают, то существует только один путь, и можно сразу выйти из функции с помощью оператора `return`, вернув результат 1:

```
if vStart == vEnd:
    return 1
```

Приведём полный текст функции:

```
def pathCount( graph, vStart, vEnd, visited ):
    if vStart == vEnd: return 1
    visited.append( vStart )
    count = 0
    for v in graph[vStart]:
        if not v in visited:
            count += pathCount( graph, v, vEnd, visited )
    visited.pop()
    return count
```

Основная программа заполняет список смежности и вызывает функцию `pathCount`:

```
graph = [[3], [0,2], [], [1,2,4], [2]]
print( pathCount( graph, 0, 2, [] ) )
```

Здесь мы находим количество путей из вершины 0 в вершину 2.

Отметим, что при первом вызове функции передаётся пустой список посещённых вершин `visited` — ни одну вершину мы пока не посетили.

У приведённой программы есть два недостатка. Во-первых, она не выводит сами маршруты, а только определяет их количество. Во-вторых, некоторые данные могут вычисляться повторно: если мы уже нашли количество путей из какой-то вершины в конечную вершину, то, когда это значение потребуется снова, желательно сразу использовать полученный ранее готовый результат, а не вызывать функцию рекурсивно ещё раз. Попытайтесь улучшить программу самостоятельно, исправив эти недостатки.

Для поиска количества путей мы использовали обход графа в глубину (англ. *depth-first search*). В некоторых задачах полезен обход графа в ширину (англ. *breadth-first search*). Например, таким способом по графу компьютерной сети определяется, на какой из маршрутизаторов направить пакет данных, чтобы он быстрее пришёл в место назначения. Подобный метод применяется в системах спутниковой навигации для определения кратчайшего маршрута.

С помощью графа, который описывает знакомства (например, в социальной сети), можно установить длину кратчайшей цепочки, которая связывает двух любых людей в этом графе¹⁾.

Обход в ширину используют роботы-пауки, выкачивающие из Интернета содержимое сайтов и переходящие по ссылкам на другие сайты.

Некоторые задачи

С графиками связаны некоторые классические задачи. Самая известная из них — задача коммивояжёра (бродячего торговца).

Задача коммивояжёра. Торговец должен посетить N городов по одному разу и вернуться в город, откуда он начал путешествие. Известны расстояния между городами (или стоимость переезда из одного города в другой). В каком порядке нужно посещать города, чтобы суммарная длина (или стоимость) пути оказалась наименьшей?

Эта задача оказалась одной из самых сложных задач оптимизации. По сей день известно только одно надёжное решение — полный перебор вариантов, число которых равно факториалу числа $N - 1$. Это число с увеличением N растёт очень быстро, быстрее, чем любая степень N . Уже для $N = 20$ такое решение требует огромного времени вычислений: компьютер, проверяющий 1 млн вариантов в секунду, будет решать задачу «в лоб» около четырёх тысяч лет. Поэтому математики прилагали большие усилия для того, чтобы сократить перебор — не рассматривать те варианты, которые заранее не дадут лучших результатов, чем уже полученные. В реальных ситуациях нередко оказываются полезны приближённые методы. Они не гарантируют оптимального решения, но часто дают хороший вариант, пригодный для практического использования. Мы рассмотрим эту задачу подробно в главе 2.

Задача о максимальном потоке. Есть система труб, которые имеют соединения в N узлах. Один узел S является источником, ещё один — стоком T . Известны пропускные способности каждой трубы. Надо найти наибольший поток (количество жидкости, перетекающее за единицу времени) от источника к стоку.

¹⁾ Оказалось, что для социальной сети Facebook средняя длина такой цепочки равна 4,74.

Задача о размещении магазина. Имеется N домов, в каждом из которых живёт r_i жителей ($i = 1, \dots, N$). В одном из них надо разместить магазин так, чтобы общее расстояние, проходимое всеми жителями по дороге в магазин, было минимальным. В каком доме нужно разместить магазин?

Задача о наибольшем паросочетании. Есть M мужчин и N женщин. Каждый мужчина указывает несколько (от 0 до N) женщин, на которых он согласен жениться. Каждая женщина указывает от 0 до M мужчин, за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.

Выводы

- Граф — это набор вершин и связывающих их рёбер. Часто с каждым ребром связывают некоторое число — вес ребра.
- Информацию о графе можно хранить в виде матрицы смежности, весовой матрицы, или списка смежности.
- Жадный алгоритм — это алгоритм, в котором на каждом шаге многоходового процесса выбирается наилучший (по имеющейся в данный момент информации) вариант, без учёта того что впоследствии этот выбор может привести к худшему итоговому решению. Жадный алгоритм не всегда позволяет найти лучшее решение задачи.
- Задача Прима–Краскала (построение минимального остовного дерева) решается с помощью жадного алгоритма: на каждом шаге выбирается ребро с минимальным весом, не образующее цикла с уже выбранными рёбрами.
- Алгоритм Дейкстры позволяет определить кратчайшие маршруты из одной вершины во все остальные, его асимптотическая сложность $O(N^2)$.
- Алгоритм Флойда–Уоршелла находит длины всех кратчайших маршрутов в графе (из каждой вершины во все остальные), его асимптотическая сложность $O(N^3)$.

Вопросы и задания

1. Сравните известные вам способы хранения информации о графах в памяти компьютера. Какие достоинства и недостатки имеет каждый из них?
2. Сравните понятия «матрица смежности» и «весовая матрица».
3. Какие особенности может иметь весовая матрица ориентированного графа в отличие от весовой матрицы неориентированного графа?
4. Что такое жадный алгоритм? Всегда ли он позволяет найти лучшее решение?
5. Напишите программу, которая вводит из файла весовую матрицу графа и строит для него минимальное остовное дерево.



6. Оцените асимптотическую сложность алгоритма Краскала.
7. Напишите программу, которая вводит из файла весовую матрицу графа, затем вводит с клавиатуры номера начальной и конечной вершин и определяет оптимальный маршрут между ними.
8. Напишите программу, которая вводит из файла весовую матрицу графа и определяет длины всех оптимальных маршрутов с помощью алгоритма Флойда–Уоршелла.
- *9. Напишите программу, которая находит все кратчайшие маршруты в графе (а не только их длины!) с помощью алгоритма Флойда–Уоршелла. Можно применить тот же подход, который мы использовали для алгоритма Дейкстры.
- *10. Напишите программу, которая решает задачу коммивояжёра для 5 городов методом полного перебора. Можно ли с её помощью за приемлемое время получить ответ для задачи с 50 городами?
- *11. Напишите программу, которая решает задачу о размещении магазина. Для определения кратчайших путей используйте алгоритм Флойда–Уоршелла. Весовую матрицу графа вводите из файла.
- *12. Перепишите программу для поиска количества путей в графе так, чтобы она не вычисляла данные повторно.
13. Напишите программу, которая для графа, заданного списком смежности, определяет все возможные маршруты из одной вершины в другую.
14. Напишите программу, которая находит и выводит на экран все циклы в графе, заданном списком смежности.
15. Напишите программу, которая определяет кратчайший маршрут между двумя узлами компьютерной сети. Связи между узлами задаются в виде матрицы смежности или списков смежности. Для поиска используйте обход графа в ширину.

§ 11

Динамическое программирование

Ключевые слова:

- рекуррентная формула
- одномерная задача
- динамическое программирование
- двумерная задача
- принцип оптимальности

Числа Фибоначчи

Мы уже сталкивались с последовательностью чисел Фибоначчи:

$$\begin{aligned} F_1 &= F_2 = 1, \\ F_N &= F_{N-1} + F_{N-2} \text{ при } N > 2. \end{aligned}$$

Такую формулу, выражающую очередной член последовательности через предыдущие, называют *рекуррентной*. Используя эту формулу, легко написать рекурсивную функцию:

```
def fibRec( N ):
    if N < 3: return 1
    return fibRec( N-1 ) + fibRec( N-2 )
```

Попытка вызвать эту функцию для больших значений N (например, больших, чем 40) приводит к «зависанию» компьютера — программа не заканчивает работу за разумное время.

Дело в том, что каждое число Фибоначчи связано с предыдущими. Например, вычисление F_5 приводит к рекурсивным вызовам, которые показаны на рис. 1.39.

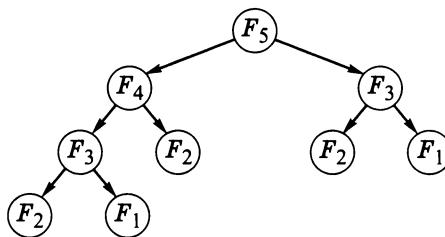


Рис. 1.39

Мы дважды вычисляем F_3 , три раза F_2 и два раза F_1 . Рекурсивное решение очень простое, но оно не оптимально по быстродействию: компьютер выполняет лишнюю работу, повторно вычисляя уже найденные ранее значения.

Какой же выход? Напрашивается такое решение: для того чтобы быстрее найти F_N , будем сохранять все предыдущие числа Фибоначчи (например, в словаре или в списке). Первые два известных значения можно сразу записать в словарь:

```
Fib = {1: 1, 2: 1} # F[1]=1, F[2]=1
```

Для вычисления остальных чисел используем рекурсивную функцию:

```
def fibMemo( N ):
    if N in Fib:                                     # (1)
        return Fib[N]                                 # (2)
    Fib[N] = fibMemo( N-1 ) + fibMemo( N-2 )      # (3)
    return Fib[N]                                     # (4)
```

В строке 1 проверяем, нет ли уже в словаре готового значения, которое нам нужно. Если есть, сразу возвращаем его как результат функции (строка 2). Иначе вычисляем значение F_N , используя рекурсию (строка 3), и возвращаем результат (строка 4).

Такой приём — сохранение промежуточных результатов в памяти — называется *memoизацией* (от англ. *memoization*). Он позволяет значительно ускорить расчёты. Если вычисление F_{35} с помощью функции fibRec занимает около 2,5 секунды, то использование мемоизации в функции fibMemo на том же компьютере уменьшает время расчёта до 16 мс, т. е. более чем в 150 раз.

Вычисленные значения можно хранить не в словаре, а в массиве (списке), назовём его F. Сначала заполним массив единицами:

```
F = [1] * (N+1)
```

В этой задаче нам удобно работать с элементами с номерами от 1 до N, не используя элемент F[0]. Поэтому размер созданного списка на 1 больше, чем N.

Для вычисления всех чисел Фибоначчи от F_1 до F_N можно использовать цикл:

```
def fibArray( N ):
    F = [1] * (N+1)
    for i in range(3, N+1):
        F[i] = F[i-1] + F[i-2]
    return F[N]
```

Такая функция вычисляет F_{35} за 6,7 мс, т. е. ещё быстрее, чем функция fibMemo со словарём.

Заметим, что в данной простейшей задаче можно обойтись вообще без массива:

```
F1, F2 = 1, 1
for i in range(3, N+1):
    F2, F1 = F1 + F2, F2
```

Ответ всегда будет находиться в переменной F2. Такой алгоритм выполняется около 2,5 мс; он работает в тысячу раз быстрее, чем первая рекурсивная функция.

Что такое динамическое программирование?



Динамическое программирование — это метод, позволяющий ускорить решение задачи за счёт использования сохранённых в памяти решений подобных задач меньшего «размера».

Обычно есть некоторое число N или несколько чисел, которые определяют «размер» задачи (и сложность её решения). При использовании метода динамического программирования задачи решаются последовательно, в порядке увеличения «размера». Первые задачи-элементы такой последовательности имеют очень простое решение, а последний

элемент — это исходная задача. В решении каждой следующей задачи используются сохранённые результаты решения таких же задач меньшего «размера».

Эту идею впервые применил американский математик Р. Беллман при исследовании сложных многошаговых процессов. Он сформулировал *принцип оптимальности*: оптимальная последовательность шагов оптимальна на любом участке. Например, чтобы найти оптимальный маршрут из вершины X в вершину Y , проходящий через вершину Z , нужно соединить два оптимальных маршрута: из X в Z и из Z в Y (рис. 1.40).

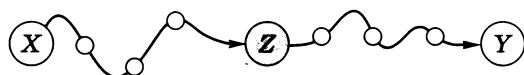


Рис. 1.40

Из принципа оптимальности следует, что можно строить оптимальное решение по шагам, учитывая уже построенные оптимальные решения для всех предыдущих шагов. Например, найдём оптимальный маршрут из пункта A в пункт E (рис. 1.41).

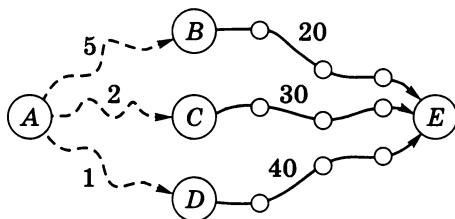


Рис. 1.41

Из пункта A есть дороги в B , C и D . Пусть уже известны оптимальные маршруты из пунктов B , C и D в пункт E (они обозначены сплошными линиями) и их «стоимости» — 20, 30 и 40.

Согласно принципу оптимальности, оптимальный маршрут из A в E через B будет состоять из ребра AB и оптимального маршрута из B в E , он имеет «стоимость» $5 + 20 = 25$. Стоимости оптимальных маршрутов, проходящих через C и D , равны 32 и 41, т. е. эти варианты хуже, чем первый. Поэтому в данной задаче оптимальный маршрут — ABE , его стоимость равна 25.

Кошки и собаки

Динамическое программирование позволяет достаточно быстро решать задачи, в которых нужно определить количество объектов, удовлетворяющих заданным условиям.

Предположим, что в зоомагазине нужно расставить в один ряд N клеток с кошками и собаками, причём нельзя ставить рядом две клетки с собаками (иначе собаки подерутся). Директор магазина хочет узнать, сколькими способами можно расставить клетки.

Давайте *формализуем задачу*: обозначим клетки с кошками нулями, а клетки с собаками — единицами. Требуется найти количество K_N цепочек длины N , состоящих из нулей и единиц, в которых нет двух стоящих подряд единиц. Такие цепочки далее будем называть «правильными».

Заметим, что количество всех битовых цепочек длины N равно 2^N . При больших N решение задачи методом полного перебора потребует огромного времени вычисления, так как необходимо проверить каждую из 2^N цепочек и определить, нет ли там соседних единиц.

Для того чтобы применить метод динамического программирования, нужно:

- определить начальные значения последовательности;
- выразить K_N через предыдущие значения последовательности: K_1 , K_2 , ..., K_{N-1} .

«Размер» нашей задачи определяется одной величиной — заданной длиной цепочки N . Такая задача называется *одномерной*. Для хранения решений всех более простых задач такого типа нужен обычный массив. В него будем записывать все найденные значения K_i ($i = 1, \dots, N$).

Очевидно, что при $N = 1$ есть две правильные цепочки, не содержащие двух соседних единиц (0 и 1), а при $N = 2$ — три (00, 01 и 10). Поэтому начальные значения последовательности: $K_1 = 2$, $K_2 = 3$.

Теперь нужно вывести рекуррентную формулу, выражающую K_N через решения таких же задач меньшего «размера». Рассмотрим цепочку из N бит, последний элемент которой — 0 (рис. 1.42).

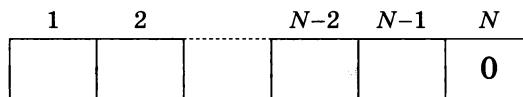


Рис. 1.42

Поскольку дополнительный ноль не может привести к появлению двух соседних единиц, правильных цепочек длины N с нулём в конце существует столько же, сколько правильных цепочек длины $N - 1$, т. е. K_{N-1} .

Если же последний символ — единица, то слева от него обязательно должен быть ноль, а начальная цепочка из $N - 2$ бит должна быть правильной (не содержащей двух соседних единиц). Поэтому правильных цепочек длины N с единицей в конце существует столько, сколько правильных цепочек длины $N - 2$, т. е. K_{N-2} (рис. 1.43).

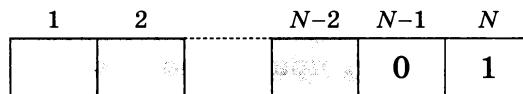


Рис. 1.43

В результате получаем: $K_N = K_{N-1} + K_{N-2}$. Значит, для вычисления очередного числа нам нужно знать два предыдущих. Легко понять, что решения нашей задачи — это числа Фибоначчи. Как их вычислять, мы уже обсудили выше.

Количество программ для исполнителя

Теперь применим динамическое программирование к задаче, в которой нужно определить количество возможных программ для исполнителя.

У исполнителя Калькулятор две команды, которым присвоены номера:

- 1) прибавь a ;
- 2) умножь на b .

Первая из них увеличивает число на экране на a , вторая умножает его на b . Числа $a > 0$ и $b > 1$ — целые. Программа для Калькулятора — это последовательность команд. Нужно определить, сколько есть программ, которые число M преобразуют в число N ($N \geq M$).

Обозначим количество разных программ для получения числа N из начального числа M через K_N .

Так как $a > 0$ и $b > 1$, при выполнении любой из команд число увеличивается. Поэтому с помощью такого исполнителя невозможно получить из M число меньшее, чем M . Следовательно, $K_i = 0$ для всех $i < M$.

Понятно, что для получения числа M из начального значения M существует только одна программа — пустая, не содержащая ни одной команды. Это значит, что $K_M = 1$.

Теперь нужно построить рекуррентную формулу, связывающую K_N с предыдущими элементами последовательности K_1, K_2, \dots, K_{N-1} .

Если число N не делится на b , то на последнем шаге оно может быть получено только операцией сложения, в этом случае $K_N = K_{N-a}$. Если же N делится на b , то последней командой может быть как сложение, так и умножение. Поэтому нужно сложить K_{N-a} (количество программ с последней командой сложения) и $K_{N/b}$ (количество программ с последней командой умножения). В итоге получаем:

$$K_N = \begin{cases} K_{N-a}, & \text{если } N \text{ не делится на } b, \\ K_{N-a} + K_{N/b}, & \text{если } N \text{ делится на } b. \end{cases}$$

Решим вручную задачу такого типа для $a = 1$, $b = 3$, $M = 1$ и $N = 17$ (рис. 1.44).

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
K_N	1	1	2	2	2	3	3	3	5	5	5	7	7	7	9	9	9

Рис. 1.44

Для начального значения $N = 1$ количество $K_N = 1$ мы уже определили. При $N = 2$ (не делится на 3) получаем $K_2 = K_1 = 1$, а при $N = 3$ (делится на 3) получаем $K_3 = K_2 + K_1 = 2$, и т. д.

Заметим, что количество вариантов меняется только в тех столбцах, где N делится на 3, поэтому из всей таблицы можно оставить только эти столбцы (и самый первый) — рис. 1.45.

N	1	3	6	9	12	15	18
K_N	1	2	3	5	7	9	12

Рис. 1.45

Заданное число 17 попадает в последний интервал (от 15 до 18), поэтому ответ в данной задаче — 9.

При составлении программы с полной таблицей нужно выделить в памяти целочисленный массив K , индексы которого изменяются от 0 до N , и заполнить его по приведённым выше формулам:

```
# здесь нужно ввести a, b, M, N
K = [0] * (N+1) # заполняем массив нулями
K[M] = 1 # начальное значение
for i in range(M+1, N+1):
    K[i] = K[i-a]
    if i % b == 0: # если может быть команда 2
        K[i] += K[i//b]
```

Ответом будет значение $K[N]$.

Двумерная задача

Во всех предыдущих задачах для хранения промежуточных решений мы использовали одномерный массив (список). Теперь рассмотрим задачи, в которых решение с помощью динамического программирования строится на основе таблицы (двумерного массива).

Задача о размене монет. Сколькоими различными способами можно выдать сдачу W рублей, если у кассира есть монеты достоинством p_i ($i = 1, \dots, N$)? Для того чтобы сдачу всегда можно было выдавать, будем предполагать, что в наборе есть монета достоинством 1 рубль ($p_1 = 1$).

Это задача решается полным перебором вариантов, число которых при больших N очень велико. Для ускорения расчётов используем идею динамического программирования: будем сохранять в памяти решения всех задач меньшего «размера» — для меньших значений N и W .

«Размер» задачи определяется двумя числами: N и W . Поэтому для хранения решений нужно использовать не линейный массив, а матрицу. Такие задачи называются *двумерными*.

Пусть в матрице T значение элемента $T[i][w]$ — это количество вариантов выдачи сдачи размером w рублей (w изменяется от 0 до W) при использовании первых i типов монет.

Очевидно, что при нулевой сдаче есть только один вариант (не дать ни одной монеты), так же и при наличии только одного типа монет есть тоже только один вариант (напомним, что $p_1 = 1$). Поэтому нулевой столбец и первую строку таблицы можно сразу заполнить единицами. Для примера мы будем рассматривать задачу для $W = 10$ и набора монет достоинством 1, 2, 5 и 10 рублей (рис. 1.46).

	0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1
2		1									
5		1									
10		1									

Рис. 1.46

Таким образом, мы выделили простые базовые случаи, от которых «отталкивается» рекуррентная формула.

Теперь рассмотрим общий случай. Заполнять таблицу будем по строкам: слева направо, сверху вниз.

Для вычисления $T[i][w]$ предположим, что мы добавляем в набор монету достоинством p_i . Если сумма w меньше, чем p_i , то количество вариантов не увеличивается, и $T[i][w] = T[i-1][w]$. Если сумма w больше, чем p_i , то к этому значению нужно добавить количество вариантов с «участием» новой монеты. Если монета достоинством p_i использована, то нужно учесть ещё все варианты «разложения» остатка $w-p_i$ на все доступные монеты (включая монету достоинством p_i):

$$T[i][w] = T[i-1][w] + T[i][w-p_i].$$

В итоге получается рекуррентная формула:

$$T[i][w] = \begin{cases} T[i-1][w], & \text{если } w < p_i, \\ T[i-1][w] + T[i][w-p_i], & \text{если } w \geq p_i. \end{cases}$$

По этой формуле заполняем таблицу (рис. 1.47).

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

Рис. 1.47

Ответ к задаче находится в правом нижнем углу таблицы (это число 11).

Использование динамического программирования позволяет ускорить решение, однако требует дополнительной памяти — нужно использовать вспомогательный массив размером $N \times (W + 1)$ для хранения промежуточных результатов.

Поиск оптимального решения

Динамическое программирование помогает и при решении сложных (переборных) задач оптимизации, в которых нужно найти самое лучшее из возможных решений.

Задача о разливе молока. В цистерне N литров молока. Есть бидоны объёмом 1, 5 и 6 литров. Нужно разлить молоко в бидоны так, чтобы все используемые бидоны были заполнены и их количество было минимальным.

Человек, скорее всего, будет решать задачу простым перебором вариантов. Но нам нужно написать программу, которая решает задачу автоматически для любого введённого числа N (оно определяет «размер» задачи).

Самый простой подход — заполнять сначала бидоны самого большого размера (6 л), затем — меньшие и т. д. Это жадный алгоритм. Как вы знаете, он не всегда приводит к оптимальному решению. Например, для $N = 10$ жадный алгоритм даёт решение $6 + 1 + 1 + 1 + 1$ — всего 5 бидонов, в то время как можно обойтись двумя ($5 + 5$).

Сначала мы найдём оптимальное число бидонов K_N , а потом подумаем, как определить, какие именно бидоны нужно использовать.

Начнём с составления рекуррентной формулы. Представим себе, что мы выбираем бидоны постепенно. Тогда последний выбранный бидон может иметь, например, объём 1 л, в этом случае $K_N = 1 + K_{N-1}$. Если последний бидон имеет объём 5 л, то $K_N = 1 + K_{N-5}$, а если 6 л, то $K_N = 1 + K_{N-6}$. Так как нам нужно использовать минимальное количество бидонов, то

$$K_N = 1 + \min(K_{N-1}, K_{N-5}, K_{N-6}).$$

В качестве начального значения берём $K_0 = 0$.

Полученная формула применима при $N > 6$. Для меньших N в той же формуле используются только значения K_i для $i \geq 0$.

Например:

$$K_3 = 1 + K_2 = 3, \quad K_5 = 1 + \min(K_4, K_0).$$

На рисунке 1.48 показан заполненный массив K для $N = 10$.

N	0	1	2	3	4	5	6	7	8	9	10
K_N	0	1	2	3	4	1	1	2	3	4	2

Рис. 1.48

Как теперь по массиву K определить оптимальный набор бидонов? Пусть, для примера, $N = 10$. По таблице определяем, что $K_{10} = 2$, т. е. требуются два бидона. На последнем шаге мы могли добавить, например, бидон объёмом 1 л, тогда бы было $K_{10-1} = K_9 = 2 - 1 = 1$, но это не так (в таблице $K_9 = 4$). Если последним добавили бидон объёмом 5 л, то $K_{10-5} = K_5 = 1$ — этот вариант подходит. Далее таким же способом определяем, что первый бидон тоже имеет объём 5 л (см. рис. 1.47).

Можно заметить, что эта процедура очень похожа на алгоритм Дейкстры, и это не случайно. В алгоритмах Дейкстры и Флойда–Уоршелла фактически используется метод динамического программирования.

Задача о куче. Из набора камней весом p_i ($i = 1, \dots, N$) нужно набрать кучу весом ровно W или, если это невозможно, максимально близкую к W (но меньшую, чем W). Все веса камней и значение W — целые числа.

Эта задача относится к трудным задачам целочисленной оптимизации, которые решаются только полным перебором вариантов. Каждый камень может входить в кучу (обозначим это состояние как 1) или не входить (0). Поэтому нужно выбрать цепочку, состоящую из N бит. При этом количество возможных вариантов равно 2^N , и при больших N полный перебор практически невыполним.

Динамическое программирование позволяет найти решение задачи значительно быстрее за счёт использования дополнительной памяти. Идея состоит в том, чтобы сохранять в массиве решения всех задач меньшего «размера»: при меньшем количестве камней и меньшем весе W .

Построим матрицу T , где элемент $T[i][w]$ — это оптимальный вес, полученный при попытке собрать кучу весом w из i первых по счёту камней. Очевидно, что первый столбец заполнен нулями (при заданном нулевом весе никаких камней не берём).

Рассмотрим первую строку (есть только один камень). В начале этой строки будут стоять нули, а дальше, начиная со столбца p_1 , — значения p_1 (взяли единственный камень). Это простые варианты задачи, решения для которых легко найти вручную. Они будут основой для заполнения таблицы.

Рассмотрим пример, когда требуется набрать вес 8 из камней весом 2, 4, 5 и 7 единиц (рис. 1.49).

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

Рис. 1.49

Предположим, что строки с 1-й по $(i-1)$ -ю уже заполнены. Перейдём к i -й строке, т. е. добавим в набор i -й камень. Он может быть взят или не взят в кучу. Если мы *не* добавляем его в кучу, то $T[i][w] = T[i-1][w]$: решение не меняется от добавления в набор нового камня. Если камень с весом p_i добавлен в кучу, то остается «добрать» остаток $w - p_i$ оптимальным образом (используя только предыдущие камни), т. е. $T[i][w] = T[i-1][w - p_i] + p_i$.

Как же выбрать, «брать или не брать»? Проверить, в каком случае полученный вес будет больше (ближе к w). Таким образом, получается рекуррентная формула для заполнения таблицы:

$$T[i][w] = \begin{cases} T[i-1][w], & \text{если } w < p_i, \\ \max(T[i-1][w], T[i-1][w - p_i] + p_i), & \text{если } w \geq p_i. \end{cases}$$

Используя эту формулу, заполняем таблицу по строкам, сверху вниз; в каждой строке — слева направо (рис. 1.50).

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

Рис. 1.50

Видим, что сумму 8 набрать невозможно, ближайшее значение — 7 (в правом нижнем углу таблицы).

Эта таблица содержит все необходимые данные для определения выбранной группы камней. Действительно, если камень с весом p_i не включён в набор, то $T[i][w] = T[i-1][w]$. Это значит, что число в таблице не меняется при переходе на одну строку выше. Начинаем с правого нижнего угла таблицы, идём вверх, пока значения в столбце равны 7. Последнее такое значение — для камня с весом 5, поэтому он и выбран. Вычитая его вес из суммы, получаем $7 - 5 = 2$, переходим во второй столбец на одну строку выше, и снова идём вверх по столбцу, пока значение не меняется (равно 2). Так как мы успешно дошли до самого верха таблицы, взят первый камень с весом 2.

Вы могли заметить, что есть ещё и второй вариант — взять один камень весом 7. Чтобы найти этот вариант по таблице, нужно проверить выполнение равенства

$$T[i][w] = T[i-1][w - p_i] + p_i.$$

Если равенство верно, то камень с весом p_i есть в одном из вариантов набора. В нашем случае $T[4][8] = 7 = T[3][1] + 7$. Поэтому во втором наборе присутствует камень с весом 7 (и он один, потому что $T[3][1] = 0$).

Как мы уже отмечали, количество возможных вариантов в задаче для N камней равно 2^N , т. е. алгоритм полного перебора имеет асимптотическую сложность $O(2^N)$. В алгоритме, использующем динамическое программирование, количество операций равно числу элементов таблицы, т. е. сложность нашего алгоритма — $O(N \times W)$. Однако нельзя сказать, что он имеет линейную сложность, так как есть ещё сильная зависимость от заданного веса W . Такие алгоритмы называют *псевдополиномиальными*, т. е. «как бы полиномиальными¹⁾». В них ускорение вычислений достигается за счёт использования дополнительной памяти, в которой хранятся промежуточные результаты.

Решение этой задачи очень похоже на решение задачи о размене монет. Это не случайно, две эти задачи относятся к классу сложных задач, для решения которых известны только переборные алгоритмы.

Выводы

- Динамическое программирование — это метод, позволяющий ускорить решение задачи за счёт использования сохранённых в памяти решений подобных задач меньшего «размера».
- Для применения метода динамического программирования нужно вывести рекуррентную формулу, связывающую решение исходной задачи с решением подобных задач меньшего «размера», и определить простые базовые случаи.
- Задачи, «размер» которых зависит от одного параметра, называются одномерными. Для их решения методом динамического программирования используется линейный массив.
- Задачи, размер которых зависит от двух параметров, называются двумерными. При решении двумерных задач нужно использовать матрицу (двумерный массив).
- С помощью динамического программирования удается избавиться от полного перебора вариантов, но при этом нужно использовать дополнительную память для хранения промежуточных результатов.

Вопросы и задания



1. Какой смысл имеет выражение «динамическое программирование» в теории многошаговой оптимизации?
2. Какие шаги нужно выполнить, чтобы применить динамическое программирование к решению какой-либо задачи?
3. За счёт чего удается ускорить решение сложных задач методом динамического программирования?
4. Какие ограничения есть у метода динамического программирования?

¹⁾ Для алгоритмов полиномиальной сложности количество операций ограничено полиномом (многочленом) от размера задачи N . Это значит, что их асимптотическая сложность равна $O(N^k)$, где k — некоторое число. Алгоритмы с линейной, квадратичной и кубической сложностью относятся именно к этому типу.

5. Сравните метод динамического программирования и рекурсивные решения.
6. Исследуйте приведённую в параграфе программу, определяющую количество программ для исполнителя. В каком случае возможна ошибка, связанная с выходом за границы массива? Как её исправить?
7. У исполнителя Калькулятор три команды, которым присвоены номера:
 - 1) прибавь a ;
 - 2) умножь на b ;
 - 3) умножь на c .Здесь a , b и c — целые числа, такие что $a > 0$, $c > b > 1$. Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число M в число N с помощью этого исполнителя.
8. В зоомагазине нужно расставить в один ряд N клеток с кошками, собаками и енотами, причём нельзя ставить рядом две клетки с собаками (иначе они подерутся). Напишите программу, которая определяет, сколькими способами можно расставить клетки.
9. Лягушка прыгает по кочкам, расположенным на одной прямой на равных расстояниях друг от друга. Кочки имеют порядковые номера от 1 до N . В начале лягушка сидит на кочке с номером 1. Она может прыгнуть вперёд на расстояние от 1 до K кочек, считая от текущей. Напишите программу, которая вычисляет количество способов, которыми Лягушка может добраться до кочки с номером N . Значения N и K вводятся с клавиатуры.
- *10. Лягушка прыгает по кочкам, расположенным на одной прямой на равных расстояниях друг от друга. Кочки имеют порядковые номера от 1 до N . В начале лягушка сидит на кочке с номером 1. Она может прыгнуть вперёд на расстояние от 1 до K кочек, считая от текущей. На каждой кочке, кроме первой и последней, Лягушка может получить или потерять несколько золотых монет (для каждой кочки это число известно). Напишите программу, которая определяет, как нужно прыгать лягушке, чтобы собрать наибольшее количество золотых монет. Значения N и K , а также число монет, которые получает или теряет лягушка на каждой кочке, вводятся с клавиатуры или из файла.
11. Напишите программу, которая решает задачу о размене монет (из параграфа).
12. Напишите программу, которая определяет оптимальный набор бидонов в задаче с молоком (из параграфа). С клавиатуры или из файла вводится объём цистерны, количество типов бидонов и их размеры.
13. Напишите программу, которая решает задачу о куче камней заданного веса (из параграфа).

- *14. *Проект.* Сравните время решения одной из рассмотренных в параграфе задач двумя способами: методом динамического программирования и полным перебором вариантов. Исследуйте зависимость времени работы программы от «размера» задачи.
- *15. Задача о ранце. Есть N предметов, для каждого из которых известен вес w_i и стоимость c_i . В ранец можно взять предметы общим весом не более W . Напишите программу, которая определяет самый дорогой набор предметов, который можно унести в ранце.
- *16. Прямоугольный остров разделён на квадраты так, что его размеры — $N \times M$ квадратов. В каждом квадрате зарыто некоторое число золотых монет, эти данные хранятся в матрице (двумерном массиве) Z , где $Z[i][j]$ — число монет в квадрате с координатами (i, j) . Пират хочет пройти из юго-западного угла острова в северо-восточный, причём он может двигаться только на север или на восток. Как пирату собрать наибольшее количество монет? Напишите программу, которая находит оптимальный путь пирата и число монет, которое ему удастся собрать.

§ 12

Игровые модели

Ключевые слова:

- игровая модель
- выигрышная позиция
- игра с нулевой суммой
- проигрышная позиция
- стратегия
- рекурсия
- игра с полной информацией
- динамическое программирование

Выигрышные и проигрышные позиции

Игровые модели — это модели, которые описывают соперничество двух (или более) сторон, каждая из которых стремится к выигрышу. Часто цели участников противоречивы — выигрыш одного означает проигрыш других. Такую игру называют *игрой с нулевой суммой*.

Построением и изучением игровых моделей занимается *теория игр* — раздел прикладной математики. Задача состоит в том, чтобы найти *стратегию* (алгоритм игры), которая позволит тому или другому участнику получить наибольший выигрыш (или, по крайней мере, наименьший проигрыш) в предположении, что соперники играют безошибочно.

Стратегия — это алгоритм игры, который позволяет добиться цели в игре в предположении, что соперники играют безошибочно.



Мы будем изучать *игры с полной информацией*, в которых результат не зависит от случая (говорят, что в таких играх нет неопределённости). Игроки делают ходы по очереди, в любой момент им известна позиция и все возможные дальнейшие ходы. Играми с полной информацией можно считать «крестики-нолики», шахматы, шашки. Пример игр с неполной информацией — карты.

Теоретически в играх с полной информацией можно построить последовательность ходов из заданной начальной позиции, которая приведёт одного из игроков к выигрышу или, по крайней мере, к ничьей. Нас будут интересовать простые игры, где не так много вариантов развития событий и такой перебор возможен за приемлемое время. В большинстве игр, например в шахматах или го, количество вариантов настолько велико, что их полный перебор требует огромного времени и поэтому нереален на практике.

Все *позиции* (игровые ситуации) делятся на выигрышные и проигрышные.



Выигрышная позиция — это такая позиция, в которой игрок, делающий очередной ход, может гарантированно выиграть при любой игре соперника, если не сделает ошибку.

При этом говорят, что у него есть *выигрышная стратегия* — алгоритм выбора очередного хода, позволяющий ему выиграть.



Проигрышная позиция — это такая позиция, в которой игрок, делающий очередной ход, обязательно проиграет, если его соперник не сделает ошибку.

В этом случае говорят, что у него нет выигрышной стратегии, а выигрышная стратегия есть у его соперника. Таким образом, общая стратегия игры состоит в том, чтобы очередным своим ходом создать проигрышную позицию для соперника.

Кто выигрывает?

Одна из простых игр, где можно перебрать все варианты, — это игра с камнями. Перед двумя игроками лежит куча из N камней или других одинаковых предметов (монет, бусин, пуговиц, фантиков и т. п.). За один ход игрок может взять один или два камня. Тот, кто возьмёт последний («заколдованный») камень, проигрывает.

Напишем программу, которая определяет, кто выиграет в заданной позиции. Главным элементом этой программы будет функция `isWinPos`, которая для заданной игровой позиции определяет, выигрышная она или проигрышная. Функция должна вернуть `True` («да»), если переданная ей позиция выигрышная, и `False` («нет»), если она проигрышная.

В нашей игре с камнями позиция задаётся одним целым числом — количеством оставшихся камней, его и нужно передать функции как параметр:

```
def isWinPos( N ):
    ...
```

По условию игра заканчивается, когда камней не остается, поэтому можно написать логическую функцию, определяющую, завершена ли игра:

```
def gameOver( N ):
    return (N <= 0)
```

Согласно условию, игрок выиграл, если соперник закончил игру (взял последний камень и проиграл), поэтому в этом случае функция isWinPos сразу вернёт значение True:

```
def isWinPos( N ):
    if gameOver( N ):
        return True
    ...
```

Если же игра ещё не завершена (камни остались), нужно проверить все возможные ходы: в этой игре мы можем уменьшать количество камней на один или на два. Если хотя бы один из этих ходов ведёт в проигрышную позицию, то данная позиция выигрышная. Попав в неё, соперник обязательно проиграет. Если же оба хода ведут в выигрышные (для соперника) позиции, то позиция проигрышная и функция должна вернуть значение False:

```
def isWinPos( N ):
    if gameOver( N ):
        return True
    for take in range(1, 3):
        if not isWinPos( N-take ):
            return True
    return False
```

Теперь можно вызывать эту функцию для произвольного значения N :

```
for startCount in range(15):
    if isWinPos( startCount ):
        print( startCount, "Можно выиграть!" )
    else:
        print( startCount, ":-(" )
```

Функция, которую мы только что построили, позволяет определить выигравшего игрока в игре с полной информацией. Она основана на полном переборе вариантов, поэтому для сложных игр (типа шахмат) будет работать очень долго. Кроме того, при больших значениях аргумента мы можем столкнуться с ошибкой «*maximum recursion depth exceeded*» — превышена максимальная глубина вложенных рекурсивных вызовов.

Динамическое программирование

Попробуем для той же игры использовать динамическое программирование. Таблицу выигрышных и проигрышных позиций будем хранить в виде массива W , в котором элемент $W[i]$ равен -1 для проигрышной позиции и 1 для выигрышной. Введём эти значения как константы и выделим память для массива:

```
N = 15
LOSE = -1
WIN = 1
W = [LOSE] * (N+1)
```

Пока считаем все позиции проигрышными — во все элементы массива мы записали константу $LOSE$.

Мы точно знаем, что ситуация, когда остался один камень, — это проигрышная позиция. А все позиции, из которых с помощью разрешённых ходов можно оставить один камень, — это выигрышные позиции. Отмечаем их значениями WIN и ищем следующую проигрышную позицию, и т. д., пока не дойдём до конца массива:

```
for i in range(1, N+1):
    if W[i] == LOSE:           # проигрышная позиция
        for take in range(1, 3): # все возможные ходы
            if i + take <= N:   # не выходим за границы
                W[i+take] = WIN
```

Поскольку мы идём в обратную сторону (от последних ходов к первым), вместо удаления камней используем обратные действия — добавление камней. Перед записью значения в массив программа проверяет, не произойдёт ли выход за границы массива.

Игра с фишками

Рассмотрим игру двух игроков с фишками, на которых написаны числа. За один ход игрок добавляет в конец цепочки одну фишку так, чтобы первая цифра на этой фишке совпадала с последней цифрой уже выставленной цепочки. Например, в конец цепочки «23 — 65» можно добавить любую фишку, на которой написано число, начинающееся с цифры 5: 51, 58, 516 и т. п. Фишки нельзя вращать, т. е. вместо фишки 51 нельзя поставить фишку 15. Тот, кто не может сделать очередной ход, проигрывает.

Напишем программу, которая по заданной начальной цепочке и набору оставшихся фишек определяет, выигрышная это позиция или проигрышная.

Будем хранить выставленную цепочку фишек как символьную строку $chain$, а оставшиеся фишki — как массив (список) символьных строк $fishSet$. Например:

```
chain = "44"
fishSet = ["12", "14", "21", "22", "24", "41", "42"]
```

В переменную `chain` записано только значение последней выставленной фишкой, так как предыдущая цепочка не влияет на результат игры.

Мы хотим написать логическую функцию `isWinPos`, которая возвращает `True` («да»), если переданная ей позиция выигрышная, и `False` («нет»), если проигрышная:

```
if isWinPos( chain, fishSet ):
    print( "Выигрыш!" )
else:
    print( "Проигрыш..." )
```

Сначала построим функцию `gameOver`, которая определяет окончание игры, т. е. такую ситуацию, когда цепочка `chain` закончилась на цифру X , и в наборе больше не осталось фишек, числа на которых начинаются на эту цифру:

```
def gameOver( chain, fishSet ):
    if not chain:                      # (1)
        return not fishSet             # (2)
    final = chain[-1]                  # (3)
    for fishka in fishSet:            # (4)
        if final == fishka[0]:         # (5)
            return False               # (6)
    return True                        # (7)
```

В первую очередь, проверяем, не пустая ли цепочка `chain` (строка 1). Если она пустая (условие `not chain` истинно), то результат функции зависит от набора фишек. Если ни одной фишки нет, то игра закончилась (нужно вернуть `True`), а если есть, то можно ставить любую фишку, и функция возвращает `False` (строка 2).

В строке 3 определяем последнюю цифру цепочки и записываем её в переменную `final`. Затем в цикле перебираем все элементы оставшегося набора фишек (строка 4). Если нашли фишку, число на которой начинается с цифры, завершающей цепочку (строка 5), то можно сразу вернуть `False` (строка 6), потому что мы нашли фишку, которую можно добавить в конец цепочки. Если ни одной подходящей фишки не найдено, то функция придёт на строку 7 и вернёт `True` — игра окончена.

Основная функция `isWinPos` принимает текущую цепочку и набор оставшихся фишек. В самом начале проверяем, не закончена ли игра: если закончена, функция сразу возвращает `False` — это проигрышная позиция, очередной ход сделать невозможно.

```
def isWinPos( chain, fishSet ):
    if gameOver( chain, fishSet ):
        return False
    ...
    ...
```

Теперь нужно выделить последнюю цифру цепочки. Учтём, что цепочка может быть и пустой — в этом случае запишем в переменную `final` пустую строку:

```
if chain:
    final = chain[-1]
else: final = ""
```

Далее в цикле перебираем все фишки, проверяя, какую из них можно поставить в конец цепочки:

```
for fishka in fishSet:
    if final == "" or final == fishka[0]:      # (1)
        newSet = fishSet[:]                      # (2)
        newSet.remove( fishka )                  # (3)
        if not isWinPos( chain+"-"+fishka,       # (4)
                          newSet ):
            return True                         # (5)
        return False                         # (6)
```

Фишку можно выставить, если цепочка пустая или оканчивается на ту же цифру, с которой начинается число на этой фишке (строка 1). В этом случае мы создаём копию списка фишек (строка 2), удаляем из этого набора выбранную фишку (строка 3), и вызываем функцию `isWinPos` рекурсивно (строка 4).

Функции передаются цепочка с добавленной новой фишкой и набор оставшихся фишек `newSet`. Если при вложенном вызове функция `isWinPos` вернула `False`, это значит, мы нашли ход в проигрышную позицию. Поэтому наша позиция — выигрышная, и сразу возвращаем ответ `True`. Заметим, что остальные ходы можно уже не проверять, потому что одного хода в проигрышную позицию достаточно для того, чтобы выиграть.

Если работа цикла закончилась и ни одного хода в проигрышную позицию не найдено (оператор `return` в строке 5 не выполнился), функция возвращает `False` (строка 6). Это значит, что все возможные ходы ведут в выигрышные позиции, и заданная позиция — проигрышная.

Выводы

- Стратегия — это алгоритм, который позволяет добиться цели в игре в предположении, что соперники играют безошибочно.
- Выигрышная позиция — это такая позиция, в которой игрок, делающий очередной ход, может гарантированно выиграть при любой игре соперника, если не сделает ошибку.
- Проигрышная позиция — это такая позиция, в которой игрок, делающий очередной ход, обязательно проиграет, если его соперник не сделает ошибку.
- Позиция, из которой все возможные ходы ведут в выигрышные позиции, — проигрышная.

- Позиция, из которой хотя бы один из возможных ходов ведёт в проигрышную позицию, — выигрышная, при этом выигрышная стратегия игрока состоит в том, чтобы перевести игру в эту проигрышную (для соперника) позицию.
- Для того чтобы избежать глубокой рекурсии, можно использовать динамическое программирование — сохранять в массиве промежуточные результаты, а потом их использовать.

Вопросы и задания



1. Что такое выигрышная стратегия в игре?
2. Как доказать, что заданная позиция в игре является выигрышной (или проигрышной)? В каких случаях это сделать не удаётся?
3. Напишите программу, определяющую выигрышные и проигрышные позиции для игры *Баше*. Перед двумя игроками лежит куча из N камней. По очереди за один ход каждый может убрать из кучи от 1 до 3 камней. Кто взял последний камень, выигрывает. Опишите словами оптимальную стратегию выигрывающего игрока для любого начального N .
- *4. Доработайте последнюю программу из параграфа (игра с фишками), так, чтобы она выводила на экран все возможные партии.
5. Напишите программу, определяющую выигрышные и проигрышные позиции в следующей игре. Перед двумя игроками лежат M фишек. Игроки ходят по очереди, за один ход игрок может добавить две фишк или увеличить количество фишек на столе в два раза. У каждого есть неограниченное количество фишек. Победителем считается игрок, первым собравший на столе N или больше фишек ($N > M$).
- *6. Напишите программу, которая определяет выигрышные и проигрышные позиции в следующей игре. Перед двумя игроками лежат две кучи орехов, в первой из которых a орехов, а во второй — b орехов. У каждого игрока неограниченно много орехов. Ход состоит в том, что игрок или увеличивает в 2 раза число орехов в какой-то куче, или увеличивает на 3 число орехов в одной из куч. Игра завершается, когда количество орехов в одной из куч становится не менее N . Игрок, первый получивший такую позицию, выигрывает.
- *7. Напишите программу, которая определяет выигрышные и проигрышные позиции в следующей игре. На координатной плоскости в точке с координатами (a, b) стоит фишка. Два игрока ходят по очереди. За один ход игрок может переместить фишку из точки с координатами (x, y) в одну из трёх точек: $(x+3, y)$, $(x, y+3)$ или $(x, y+4)$. Выигрывает игрок, после хода которого расстояние по прямой от фишки до начала координат становится не меньше N единиц.
- *8. *Проект.* Напишите программу, которая анализирует позицию в игре «крестики-нолики»: определяет, выигрышная она или нет.
- **9. *Проект.* Напишите программу, которая вводит из файла расположение фигур на шахматной доске и выясняет, могут ли белые выиграть за 3 хода.

Глава 2

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++

В этой главе мы продолжаем изучение языка C++, с которым знакомились в предыдущие два года. Основной теоретический материал уже изложен в главе, посвящённой языку Python, поэтому здесь мы рассмотрим особенности программирования тех же алгоритмов на языке C++.

§ 13

Простые алгоритмы сортировки

Ключевые слова:

- сортировка
- метод пузырька
- сортировка вставками
- перестановка элементов
- сложность алгоритма

Метод пузырька (сортировка простыми обменами)

Здесь и далее будем считать, что в программе объявлен массив A, состоящий из N элементов — целых чисел:

```
const int N = 10;  
int A[N];
```

Также будем предполагать, что он заполнен некоторыми значениями, которые нам нужно отсортировать в порядке возрастания (неубывания).

При сортировке методом пузырька мы последовательно «поднимаем» (т. е. продвигаем к началу массива) самые «лёгкие» элементы — имеющие наименьшие значения:

```
for( int i = 0; i < N-1; i++ )  
    for( int j = N-2; j >= i; j-- )  
        if ( A[j] > A[j+1] ) {  
            int temp = A[j];  
            A[j] = A[j+1];  
            A[j+1] = temp;  
        }
```

Для того чтобы поменять местами значения двух элементов массива (или любых двух переменных), в C++ нужно использовать временную переменную, которую часто называют `temp` (от англ. *temporary* — временный).

Нужно стараться объявлять переменные как можно ближе к тому месту в программе, где они используются, и максимально ограничивать их «видимость». Тогда текст программы будет значительно легче читать. Так, в приведённом фрагменте программы, который выполняет сортировку методом пузырька, переменные i и j объявляются прямо в заголовках циклов, а переменная $temp$ — в теле условного оператора, где она используется для перестановки элементов массива.

Интересно сравнить быстродействие одинаковых программ сортировки, написанных на разных языках программирования. Программа сортировки методом пузырька на Python сортирует массив из 10 000 элементов за 18,4 секунды, а такая же программа на C++ — за 0,235 секунды (на том же компьютере), т. е. почти в 80 раз быстрее.

Сортировка вставками

Рассмотрим ещё один простой метод сортировки, который называется *сортировкой вставками*.

Предположим, что все первые элементы массива A до i -го уже отсортированы, и нужно включить в отсортированную часть элемент $A[i]$ (рис. 2.1, а).

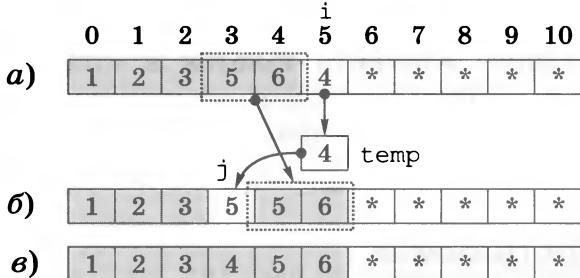


Рис. 2.1

Сохраним значение $A[i]$ во временной переменной $temp$:

```
int temp = A[i];
```

Затем сдвигаем все элементы, большие, чем $temp$, на одну позицию к концу массива, чтобы освободить место для нового элемента (рис. 2.1, б):

```
int j = i;
while( j > 0 and A[j-1] > temp ) {
    A[j] = A[j-1];
    j--;
}
```

Этот цикл останавливается в двух случаях: когда $j = 0$, т. е. новый элемент первый в отсортированной части, или когда $A[j-1] \leq temp$, т. е. мы нашли элемент, стоящий перед тем элементом, который вставляется. В обоих случаях остаётся скопировать в $A[j]$ значение, сохранённое в переменной $temp$:

```
A[j] = temp;
```

В результате все элементы до i -го включительно теперь отсортированы (рис. 2.1, в).

Поскольку массив из одного элемента всегда отсортирован, эти операции нужно выполнить для всех i от 1 до $N-1$. После каждого повторения цикла длина отсортированной части будет увеличиваться на один элемент, и после $N-1$ итераций цикла будет отсортирован весь массив.

Приведём полный цикл сортировки вставками:

```
for( int i = 1; i < N; i++ ) {
    int temp = A[i];
    int j = i;
    while( j > 0 and A[j-1] > temp ) {
        A[j] = A[j-1];
        j--;
    }
    A[j] = temp;
}
```

Существует ещё одна разновидность сортировки вставками — *метод бинарных вставок*. Он отличается от рассмотренного метода тем, что место вставки очередного элемента ищется в отсортированной начальной части массива с помощью двоичного поиска (см. § 3). Этот метод не так часто используется на практике, детали вы можете найти в литературе или в Интернете.

Массивы в подпрограммах

Теперь разберёмся, как оформить алгоритм сортировки (например, сортировку методом пузырька) в виде процедуры.

Очевидно, что процедуре нужно передать сам массив. Но массив в C++ — это просто адрес некоторой области памяти, он «не знает» своего размера. Поэтому размер массива нужно передать отдельно как второй параметр.

Тело процедуры — это двойной цикл, который мы уже использовали выше:

```
void bubbleSort( int A[], int size )
{
    for( int i = 0; i < size-1; i++)
        for( int j = size-2; j >= i; j--)
            if ( A[j+1] < A[j] ) {
                int temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
}
```

Чтобы вызвать такую процедуру, нужно передать ей в скобках сам массив и количество его элементов:

```
bubbleSort( A, N );
```

Процедура работает не с копией массива вызывающей программы (как в Python), а сортирует его «на месте».

В процедуру на самом деле передаётся не массив, а адрес начала массива, т. е. адрес элемента $A[0]$. Получение адреса в C++ обозначается знаком $\&$, поэтому процедуру можно было вызвать и так:

```
bubbleSort( &A[0], N );
```

но такая запись менее понятна.

Особенности языка C++ позволяют применить такую процедуру для сортировки любой части массива (не затрагивая остальные элементы). Например, чтобы отсортировать только первые пять элементов массива, мы просто передадим процедуре второй аргумент, равный 5:

```
bubbleSort( A, 5 );
```

Процедуре «сообщили», что в массиве только пять элементов, и остальные она не будет трогать.

Кроме того, можно начать не с первого элемента массива, а, например, с $A[3]$:

```
bubbleSort( &A[3], 5 );
```

То же самое записывается проще

```
bubbleSort( A+3, 5 );
```

«Сумма» $A+3$ в языке C++ как раз и означает адрес элемента массива $A[3]$. Учитывая, что массив A состоит из элементов типа `int`, каждый из которых обычно занимает 4 байта памяти, к адресу массива добавляется $3 \cdot 4 = 12$ байт.

Выводы

- Сортировка вставками заключается в том, что на каждом шаге новый элемент вставляется на своё место в отсортированную часть массива.
- Массивы можно передавать как аргументы в подпрограммы. При этом фактически передаётся адрес массива в памяти, поэтому все операции выполняются с исходным массивом, а не с его копией.
- Размер массива нужно передавать в подпрограмму как отдельный аргумент.

Вопросы и задания

- Как нужно изменить приведённые в параграфе алгоритмы, чтобы элементы массива были отсортированы по убыванию?



2. В массив А записаны 10 натуральных чисел:

```
int A[] = {8, 2, 6, 4, 10, 1, 5, 7, 9, 3}
```

Определите, какие значения окажутся в переменных с и d после выполнения фрагмента программы.

a)

```
const int N = 10;
int c = 0, d = 0;
for( int i = 1; i < N; i++ )
    if( A[i] < A[0] ) {
        c++;
        int temp = A[i];
        A[i] = A[0];
        A[0] = temp;
    }
else
    d++;
```

б)

```
const int N = 10;
int c = 0, d = 0;
for( int i = 0; i < N-1; i++ )
    if( A[i] > A[9] ) {
        c++;
        int temp = A[i];
        A[i] = A[i+1];
        A[i+1] = temp;
    }
else
    d++;
```

в)

```
const int N = 10;
int c = 0, d = 0;
for( int i = 1; i < N; i++ )
    if( A[i] < A[0] ) {
        c++;
        int temp = A[i];
        A[i] = A[i-1];
        A[i-1] = temp;
    }
else
    d++;
```

3. Как нужно вызвать процедуру сортировки массива, чтобы отсортировать последние 10 из N элементов?
4. Напишите процедуру, в которой сортировка выполняется «методом камня» — самый «тяжёлый» элемент «опускается» в конец массива.

5. Напишите функцию, которая сортирует массив методом пузырька с «флажком»: сортировка заканчивается, если при очередном проходе не было ни одной перестановки. Функция должна вернуть количество перестановок.
- *6. Напишите программу, которая сравнивает количество перестановок элементов при сортировке одного и того же массива разными методами. Проведите эксперименты для возрастающей последовательности (уже отсортированной), убывающей (отсортированной в обратном порядке) и случайной.
- *7. Используя информацию из дополнительных источников и библиотеку `time.h`, попробуйте построить зависимость времени выполнения сортировки от размера массива для рассмотренных методов. Какие зависимости у вас получились? Сравните их с результатами таких же исследований программы на языке Python.
- *8. Используя программу, написанную при выполнении предыдущего задания, сравните время сортировки массивов разного размера обычным методом вставок и бинарными вставками. Сделайте выводы.

Интересные сайты

cppstudio.com — программирование на C++ для начинающих
ideone.com — онлайн-сервис для программирования на разных языках
repl.it — онлайн-сервис для программирования на разных языках.
onlinegdb.com — онлайн-компилятор и отладчик для разных языков программирования

§ 14

Быстрые алгоритмы сортировки и поиска

Ключевые слова:

- сортировка слиянием
- быстрая сортировка
- двоичный поиск
- лямбда-функция

Сортировка слиянием

Как вы уже знаете, сортировка слиянием включает предварительную сортировку половин массива, а затем их слияние в новый отсортированный массив. Процедура сортировки будет вызывать сама себя рекурсивно. Для того чтобы сортировать массив на месте, нам нужно определить границы сортируемой части, назовём их L (от англ. *left* — левый) и R (англ. *right* — правый).

Итак, в массиве A мы хотим отсортировать по возрастанию все элементы, начиная с A[L] до A[R] включительно. Очевидно, что, если L ≥ R, ничего сортировать не нужно, можно просто выйти из процедуры:

```
void mergeSort( int A[], int L, int R )
{
    if( L >= R ) return;
    ...
}
```

Если в рабочей части массива больше одного элемента, применяем принцип «разделяй и властвуй»:

- 1) определяем середину рабочей части, записываем индекс среднего элемента в переменную mid;
- 2) сортируем отдельно первую и вторую половины массива;
- 3) сливаем две отсортированные половины массива.

В результате получается такая процедура:

```
void mergeSort( int A[], int L, int R )
{
    if( L >= R ) return;
    int mid = (L + R)/2;
    mergeSort( A, L, mid );
    mergeSort( A, mid+1, R );
    merge( A, L, mid, R );
}
```

Осталось только написать процедуру merge, которая сливает две (уже отсортированные!) соседние части массива — первая состоит из элементов с индексами от L до mid (включительно), а вторая — из элементов с индексами от mid+1 до R. После слияния эти части будут занимать то же самое место в массиве A — это будут элементы с индексами от L до R включительно. Процедура должна получить массив и значения L, mid и R:

```
void merge( int A[], int L, int mid, int R )
{
    ...
}
```

Будем использовать тот же алгоритм слияния, который описан в § 2. Нам нужен будет вспомогательный массив temp, в котором мы построим результат слияния. Его размер должен быть такой, чтобы вместить все нужные элементы исходного массива, с A[L] до A[R] включительно. Это значит, что нужно выделить в памяти массив для хранения R-L+1 целых чисел:

```
int temp[R-L+1];
```

Размещение этого массива в памяти будет происходить заново при каждом входе в процедуру `merge`, после её завершения память освобождается¹⁾.

В переменные `i1` и `i2` запишем индексы начальных элементов каждой части:

```
int i1 = L, i2 = mid + 1;
```

а в переменной `k` будем хранить индекс первого ещё не заполненного элемента массива `temp`:

```
int k = 0;
```

Теперь сливаем две части массива, выбирая на каждом шаге минимальный из оставшихся элементов:

```
while( i1 <= mid and i2 <= R )
    if( A[i1] < A[i2] )
        temp[k++] = A[i1++];
    else
        temp[k++] = A[i2++];
```

Здесь нужно пояснить странную запись вида

```
temp[k++] = A[i1++];
```

Она говорит о том, что нужно сначала скопировать значение `A[i1]` в `temp[k]`, а после этого увеличить оба счётчика. То есть так в краткой форме записаны три оператора:

```
temp[k] = A[i1];
k++;
i1++;
```

После того как один из сливаемых массивов закончится, нужно будет добавить оставшиеся элементы второго массива в конец массива `temp`. Чтобы не выяснять, какой именно массив закончился, добавим оставшиеся части обоих массивов:

```
while( i1 <= mid )
    temp[k++] = A[i1++];
while( i2 <= R )
    temp[k++] = A[i2++];
```

Для одного из этих циклов условие будет сразу ложным, и он не выполнится ни разу.

¹⁾ Обратите внимание, что размер массива `temp` — не константа, а выражение, которое зависит от значений переменных `R` и `L`. Такие массивы называются массивами переменной длины (англ. *variable-length array*), потому что их размер неизвестен во время компиляции. Их можно использовать в современных версиях языка C++, начиная с C++14. К сожалению, на момент написания пособия (2018 год) компилятор Microsoft Visual Studio эту возможность не поддерживает.

На последнем этапе нужно переписать элементы массива `temp` на свои места в массив `A`:

```
for( int i = L; i <= R; i++ )
    A[i] = temp[i-L];
```

Приведём процедуру `merge` полностью:

```
void merge( int A[], int L, int mid, int R )
{
    int temp[R-L+1];
    int i1 = L, i2 = mid + 1;
    int k = 0;

    while( i1 <= mid and i2 <= R )
        if( A[i1] < A[i2] )
            temp[k++] = A[i1++];
        else
            temp[k++] = A[i2++];

    while( i1 <= mid )
        temp[k++] = A[i1++];
    while( i2 <= R )
        temp[k++] = A[i2++];

    for( int i = L; i <= R; i++ )
        A[i] = temp[i-L];
}
```

Быстрая сортировка

Недостаток алгоритма быстрой сортировки, который мы применяли в языке Python, состоит в том, что он использует много памяти на создание вспомогательных массивов. Рассмотрим теперь быструю сортировку «на месте», которая не требует дополнительной памяти.

Пусть дан массив `A` из N элементов. Выберем сначала наугад любой элемент массива (назовем его X). На первом этапе (он называется *этапом разделения*) мы расставим элементы так, что слева от некоторой границы (в первой группе) будут находиться все числа, меньшие или равные X , а справа (во второй группе) — большие или равные X (рис. 2.2). Заметим, что элементы, равные X , могут находиться в обеих частях массива.

$A[i] \leq X$	$A[i] \geq X$
---------------	---------------

Рис. 2.2

Теперь элементы расположены так, что ни один элемент из первой группы при сортировке не окажется во второй группе, и наоборот. Поэтому далее достаточно отсортировать *отдельно* каждую часть массива — это снова уже знакомый подход «разделяй и властвуй».

Как же разделить массив? Предположим, что мы выбрали некоторое значение X , равное одному из элементов массива. Сначала будем просматривать массив слева до тех пор, пока не обнаружим элемент, который больше или равен X . Мы всегда найдём такой элемент; в крайнем случае (если X — наибольший элемент массива) это будет элемент, равный X . Затем просматриваем массив справа до тех пор, пока не обнаружим элемент, который меньше или равен X . Теперь поменяем местами эти два элемента и продолжим просмотр с двух сторон до тех пор, пока два «просмотра» не «встретятся» где-то в середине массива. В результате массив окажется разбитым на две части: левую со значениями меньшими или равными X , и правую со значениями большими или равными X . На этом первый этап («разделение») закончен.

Чтобы понять, как работает этот алгоритм, рассмотрим пример. Пусть задан массив

78	6	82	67	55	44	34
↑ X						

Выберем в качестве X средний элемент массива, равный 67. Найдём первый слева элемент массива, который больше или равен X и должен стоять во второй части. Это число 78. Обозначим индекс этого элемента через L .

Теперь найдём самый правый элемент, который меньше X и должен стоять в первой части. Это число 34. Обозначим его индекс через R .

78	6	82	67	55	44	34
↑ L						
R						

Теперь поменяем местами эти два элемента. Сдвигая переменную L вправо, а R — влево, находим следующую пару, которую надо переставить. Это числа 82 и 44:

34	6	82	67	55	44	78
↑ L						
R						

Следующая пара элементов для перестановки — числа 67 и 55:

34	6	44	67	55	82	78
↑ L						
R						

После этой перестановки дальнейший поиск приводит к тому, что переменная L становится больше R , т. е. массив разбит на две части. В результате все элементы массива, расположенные левее $A[L]$, меньше или равны X , а все элементы правее $A[R]$ — больше или равны X :

34	6	44	55	67	82	78
↑ R						
L						

Теперь нужно применить тот же алгоритм к двум полученным частям массива: первая часть — с первого до R-го элемента, вторая часть — с L-го до последнего элемента. Таким образом, сортировка исходного массива свелась к сортировкам двух частей массива, т. е. к двум задачам того же типа, но меньшего «размера». Как вы знаете, такой прием называется *рекурсией*.

Процедура сортировки должна принимать три параметра:

- 1) массив A, который нужно отсортировать;
- 2) индекс start первого элемента сортируемой области;
- 3) индекс end последнего элемента сортируемой области.

Сейчас уже можно написать заголовок процедуры и условие выхода:

```
void quickSort( int A[], int start, int end )
{
    if( start >= end ) return;
    ...
}
```

Понятно, что если $start \geq end$, то в рабочей части массива осталось меньше двух элементов, так что сортировать нечего, нужно просто выйти из процедуры.

Сначала выбираем разделитель X — случайный элемент массива:

```
int X = A[randInt( start, end )];
```

Здесь вызывается функция `randInt(a, b)`, которая выдаёт случайное целое число на отрезке $[a; b]$:

```
int randInt( int a, int b )
{
    return a + rand() % (b-a+1);
}
```

Запишем в переменные L и R индексы крайних элементов рабочей области:

```
int L = start, R = end;
```

Основной этап сортировки — это разделение массива на две части и перестановки элементов из одной половины в другую:

```
while( L <= R ) {
    // поиск элементов для перестановки
    while( A[L] < X ) L++;
    while( A[R] > X ) R--;
    // перестановка, если нужно
    if ( L <= R ) {
        int temp = A[L];
        A[L] = A[R];
        A[R] = temp;
        L++; R--;
    }
}
```

Теперь осталось отсортировать отдельно первую и вторую части, вызвав процедуру рекурсивно:

```
quickSort( A, start, R );
quickSort( A, L, end );
```

Приведём получившуюся процедуру целиком:

```
void quickSort( int A[], int start, int end )
{
    if( start >= end ) return;
    int L = start, R = end;
    int X = A[randInt( start, end )];
    // разделение
    while( L <= R ) {
        while( A[L] < X ) L++;
        while( A[R] > X ) R--;
        if ( L <= R ) {
            int temp = A[L];
            A[L] = A[R];
            A[R] = temp;
            L++; R--;
        }
    }
    // сортировка двух частей
    quickSort( A, start, R );
    quickSort( A, L, end );
}
```

Обратите внимание, что наша процедура может сортировать любую часть массива. Для того чтобы отсортировать весь массив размера N , нужно вызвать её так:

```
quickSort( A, 0, N-1 );
```

А вот такой вызов

```
quickSort( A+3, 0, 5 );
```

отсортирует 6 элементов, начиная с $A[3]$.

В следующей таблице сравниваются времена работы изученных алгоритмов сортировки (в секундах) для массивов разного размера, заполненных случайными значениями.

N	Метод пузырька	Метод вставок	Сортировка слиянием	Быстрая сортировка
5000	0,039	0,016	0,00047	0,00031
15000	0,359	0,141	0,00172	0,00078
50000	3,985	1,469	0,00610	0,00297

Как показывают эти данные, преимущество быстрой сортировки становится всё более убедительным при увеличении N .

Стандартная сортировка в языке C++

В стандартной библиотеке языка C++ есть встроенная функция для сортировки массивов, которая называется `sort`. Она объявлена в пространстве имён `std` в библиотеке `algorithm`, которую нужно подключить в начале программы:

```
#include <algorithm>
```

Функция `sort` вызывается так:

```
sort( A, A+N );
```

Первый аргумент, переданный этой функции, — это адрес первого элемента массива, а второй — адрес элемента, следующего за рабочей частью массива. Напомним, что запись `A+N` означает то же самое, что и `&A[N]`. Если массив имеет размер N , то элемент `A[N]` уже не принадлежит массиву, этот адрес выступает как ограничитель.

По умолчанию функция `sort` сортирует числа в порядке возрастания. Если нужен другой порядок сортировки, можно указать третий аргумент — название логической функции с двумя параметрами. Эта функция должна возвращать истинное значение, если значение её первого параметра нужно поставить в отсортированном массиве раньше, чем значение второго параметра.

Например, пусть нужно отсортировать массив по убыванию последней цифры числа — так, чтобы сначала шли все числа, которые заканчиваются на цифру 9, потом — все числа, которые заканчиваются на 8, и т. д. Составим функцию, которая сравнивает последние цифры переданных ей чисел (остатки от деления на 10):

```
bool compare(int n1, int n2)
{
    return (n1 % 10) > (n2 % 10);
}
```

Теперь можно вызывать функцию `sort`:

```
sort( A, A+N, compare );
```

Для того чтобы не оформлять короткую функцию сравнения в виде отдельной подпрограммы, можно применить *лямбда-функцию*. Так называют функцию без имени, которая используется только один раз (см. § 2). Предыдущий вызов функции `sort` можно было оформить иначе:

```
sort( A, A+N,
      [] ( int n1, int n2 )
      { return (n1 % 10) < (n2 % 10); } );
```

Третий аргумент — это лямбда-функция, равносильная написанной выше функции compare. Вместо типа возвращаемого значения и имени функции записаны квадратные скобки, а список параметров и тело функции — такие же, как у функции compare. Тип возвращаемого значения компилятор определяет автоматически (в данном случае это тип `bool` — логическое значение).

Лямбда-функция может «захватывать» переменные, находящиеся в области видимости. В последнем примере можно было записать основание системы счисления в переменную `base`:

```
int base = 10;
sort( A, A+N,
      [base] ( int n1, int n2 )
              { return (n1 % base) < (n2 % base); } );
```

Имена всех захваченных переменных перечисляются через запятую в квадратных скобках. Эти переменные можно использовать внутри лямбда-функции так же, как и параметры.

Двоичный поиск

В отсортированном массиве легче искать, потому что мы можем использовать не линейный, а двоичный поиск (см. § 3).

Применим алгоритм двоичного поиска в массиве, подробно рассмотренному в § 3, «завернув» его в функцию:

```
int binSearch( int X, int A[], int size )
{
    int L = 0, R = size;           // начальный отрезок
    while( L < R-1 ) {
        int mid = (L+R) / 2;       // нашли середину
        if( X < A[mid] )          // сжатие отрезка
            R = mid;
        else L = mid;
    }
    if( A[L] == X )
        return L;
    else return -1;
}
```

Эта функция принимает три параметра:

- 1) значение `X`, которые мы ищем;
- 2) массив `A`, в котором мы ищем значение `X`;
- 3) размер этого массива `size`.

Функция возвращает индекс первого найденного элемента, равного `X`, а если такого элемента нет, то возвращает `-1`. Вызывать эту функцию, можно, например, так:

```

int X = 12;
int nX = binSearch( X, A, N );
if( nX < 0 )
    cout << "Не нашли " << X << "...";
else
    cout << "A[ " << nX << " ] = " << X;

```

Здесь предполагается, что целочисленный массив A размера N уже был предварительно отсортирован по возрастанию.

Двоичный поиск можно применять не только для поиска элемента, но и при сортировке вставками для определения места нового элемента в отсортированной части массива. Такой вариант называется методом бинарных вставок.

Выводы

- Сортировка слиянием и быстрая сортировка в C++ обычно выполняются «на месте», т. е. переставляются элементы исходного массива.
- При сортировке слиянием используется временный массив на стадии слияния. Он удаляется из памяти после окончания работы процедуры.
- Метод быстрой сортировки предусматривает разделение массива на две части. В первую входят элементы, меньшие или равные выбранному значению X , а во вторую — элементы, большие или равные X . Обе части далее сортируются с помощью того же алгоритма.
- Методы сортировки слиянием и быстрой сортировки используют рекурсию.
- В языке C++ существует стандартная функция `sort`, выполняющая быструю сортировку массивов.
- Для поиска значения в отсортированном массиве можно применять алгоритм двоичного поиска, который работает значительно быстрее, чем линейный поиск.



Вопросы и задания

- Почему при быстрой сортировке массива на этапе разделения нельзя использовать нестрогие неравенства в циклах?

```

while( A[L] <= X ) L++;
while( A[R] >= X ) R--;

```

 Покажите это на примере массива {3, 2, 2}.
- Попробуйте применить стандартную функцию `sort` для сортировки нечисловых данных, например символьных строк.
- Как нужно изменить приведённые в параграфе алгоритмы, чтобы элементы массива были отсортированы по убыванию?

- *4. Используя информацию из дополнительных источников и библиотеку `time.h`, постройте зависимость времени выполнения сортировки от размера массива для разных методов.
5. Напишите программу, которая считает среднее число шагов при двоичном поиске для массива из 32 элементов на отрезке [0; 100]. Для поиска используйте 1000 случайных чисел в этом же отрезке.
6. Используя дополнительные источники, изучите стандартный алгоритм двоичного поиска `binary_search` из библиотеки STL и попробуйте его применить.
7. Используя дополнительные источники, выясните, что означают знаки `&` и `=` в квадратных скобках при определении лямбда-функции. Например:

```
int base = 8;
sort( A, A+N,
      [=] ( int n1, int n2 )
      { return (n1 % base) < (n2 % base); } );
```

Интересные сайты

stepik.org/course/363/syllabus — онлайн-курс по программированию на C++ для начинающих

cplusplus.com — сайт, посвящённый языку C++

ru.cppreference.com — онлайн-справка по C++

§ 15

Обработка файлов

Ключевые слова:

- файл
- файловый поток
- открытие файла
- закрытие файла
- чтение из файла
- запись в файл
- конец файла
- аргументы командной строки

Принцип сэндвича

Так же, как и в языке Python, работа с файлом из программы на C++ включает три этапа:

- 1) открытие файла;
- 2) операции чтения или записи данных;
- 3) закрытие (освобождение) файла.

Такую последовательность иногда называют «принципом сэндвича».

Открыть файл — значит сделать его доступным для чтения и/или записи данных из программы. Если файл не открыт, то программа не

может к нему обращаться. Как правило, при открытии файл блокируется, и другие программы не смогут с ним работать, пока мы его не закроем.

При открытии файла указывают *режим работы*: чтение, запись или добавление данных в конец файла.

Когда файл открыт, программа выполняет все необходимые операции с ним. После этого нужно *закрыть файл*, т. е. освободить его, разорвать связь с программой. Именно при закрытии файла все последние изменения, сделанные программой в этом файле, записываются на диск.

Файловые потоки

В языке C++ можно работать с текстовыми файлами, используя файловые потоки ввода/вывода. Для работы с ними нужно подключить библиотеку `fstream` (от англ. *file stream* — файловый поток):

```
#include <fstream>
```

Файловые потоки работают так же, как и стандартные потоки ввода-вывода `cin` и `cout`, но для чтения и записи данных используют файлы, а не консольное окно.

Входной поток (файл для чтения) связан с переменной типа `ifstream` (от англ. *input file stream* — входной файловый поток), а выходной поток — с переменной типа `ofstream` (от англ. *output file stream* — выходной файловый поток). Вот так можно объявить входной поток `Fin` и выходной поток `Fout`:

```
ifstream Fin;
ofstream Fout;
```

Для открытия файла (в виде потока) используется метод `open`, а для закрытия — метод `close`:

```
Fin.open( "input.txt" );
Fout.open( "output.txt" );
// здесь работаем с файлами
Fin.close();
Fout.close();
```

Если файл `output.txt` существует, его содержимое удаляется, если не существует, то создаётся новый файл.

Этот пример демонстрирует общие принципы работы с файлами. Однако язык C++ позволяет значительно сократить текст программы, сохранив её работоспособность. Во-первых, можно открывать потоки сразу при объявлении переменных:

```
ifstream Fin( "input.txt" );
ofstream Fout( "output.txt" );
```

Во-вторых, файловые потоки автоматически закрываются, когда поток выходит из области видимости, ограниченной фигурными скобками. Например, все потоки, созданные в функции, закрываются, когда функция заканчивает работу. Поэтому чаще всего метод `close` можно не вызывать явно.

Открытие потока может завершиться неудачно. Это происходит, например, если:

- 1) попытаться открыть файл, заблокированный другой программой;
- 2) попытаться связать выходной поток с файлом, для которого установлен режим «только для чтения»;
- 3) закончилось место на диске, и т. п.

Если открыть поток не удалось, преобразование переменной потока в логическое значение даёт `false` («ложь»). Эта особенность полезна для обработки ошибки. Вместо этого можно также использовать метод потока `is_open`. Если поток открыт удачно, этот метод возвращает значение `true`:

```
ifstream Fin( "input.txt" );
if( Fin ) { // или if( Fin.is_open() )
    // здесь работаем с файлом
}
else
    cout << "Открыть файл не удалось.";
```

Если `Fin` и `Fout` — это потоки, открытые, соответственно, на ввод и на вывод, то можно написать так:

```
int a, b
Fin >> a >> b;
Fout << a << "+" << b << "=" << a + b;
```

В этом фрагменте программы из одного файла в переменные `a` и `b` читаются два целых числа, а затем их сумма записывается в другой файл. Числа в файле с исходными данными могут быть записаны по-разному: оба в одной строке или каждое число в отдельной строке.

Когда все данные из файла уже прочитаны (файловый курсор указывает на конец файла), логическая функция-метод `eof` (от англ. *end of file* — конец файла) возвращает истинное значение:

```
if( Fin.eof() )
printf( "Данные закончились" );
```

Выходной поток можно открыть в режиме добавления, указав второй аргумент — режим доступа `ios_base::app` (от англ. *append* — дополнить):

```
ofstream Fout( "output.txt", ios_base::app );
```

В этом случае файловый указатель сразу после открытия файла устанавливается не на начало, а на конец файла. Поэтому существую-

щий файл не удаляется, и новые данные будут записываться в конец файла. Такой приём используется для добавления записей в журнал событий какой-нибудь программы, операционной системы или веб-сайта.

Неизвестное количество данных

Пусть в текстовом файле записано неизвестное количество чисел, и требуется найти их сумму. В этой задаче не нужно одновременно хранить все числа в памяти (и не нужно выделять массив!), достаточно читать по одному числу и сразу его обрабатывать:

```
while( /* не конец файла */ ) {  
    // прочитать число из файла  
    // добавить его к сумме  
}
```

Предположим, что `Fin` — файловый поток, успешно открытый на чтение. Для того чтобы определить, когда данные закончились, будем использовать метод `eof`:

```
int sum = 0;  
while( not Fin.eof() ) {  
    Fin >> x;  
    sum += x;  
}  
cout << sum;
```

Этот фрагмент можно записать в другой форме:

```
int sum = 0;  
while( Fin >> x )  
    sum += x;  
cout << sum;
```

Заголовок цикла `while(Fin >> x)` означает «пока чтение из потока заканчивается успешно». Если прочитать очередное значение не удалось, цикл завершает работу.

Обработка массивов

Теперь рассмотрим задачу, в которой числа, записанные в файл, нужно отсортировать. Самый простой способ — загрузить их в массив и затем сортировать этот массив любым методом.

В языке C++ память под массив нужно выделить заранее, поэтому необходимо знать наибольшее возможное число элементов массива. Пусть массив способен хранить `NMAX` целых чисел:

```
const int NMAX = 100;  
int A[NMAX];
```

Основная «интрига» состоит в том, что точное количество чисел неизвестно. Следовательно, нам нужно считать, сколько чисел мы вводим из файла, и записывать их последовательно в первые ячейки массива. Если данные закончились, цикл чтения останавливается.

Кроме того, необходимо сделать защиту от слишком большого количества данных: если NMAX чисел уже записаны в массив, цикл должен остановиться, потому что следующие числа записывать некуда:

```
int N = 0;
while( N < NMAX and Fin >> A[N] )
    N ++;
```

Здесь целая переменная N служит счётчиком прочитанных из файла чисел.

Цикл завершает работу, если закончились данные в файле или счётчик N стал равен NMAX (т. е. максимально возможное количество чисел прочитано и записано в массив).

Теперь нужно отсортировать первые N (а не NMAX!) значений массива A (этот код вы уже можете написать самостоятельно) и вывести их во второй файл, открытый на запись (Fout — выходной файловый поток):

```
ofstream Fout( "output.txt" );
for( int i = 0; i < N; i++ )
    Fout << A[i] << endl;
```

Чтение файлов по словам

Пусть текстовый файл состоит из слов, которые отделены друг от друга пробельными символами (пробелами, символами табуляции, символами перехода на новую строку). Требуется определить количество слов в этом файле.

Читая символьную строку из потока, оператор >> останавливает чтение, когда встретится пробельный символ. Поэтому вывести все слова из файла на экран можно с помощью простого цикла:

```
string s;
while( Fin >> s )
    cout << s << endl;
```

Отметим, что для работы с переменными типа **string** нужно в начале программы подключить библиотеку **string**.

Для подсчёта слов вводим переменную-счётчик, которая увеличивается на единицу при чтении очередного слова:

```
string s;
int k = 0;
while( Fin >> s )
    k++;
cout << k;
```

Построчная обработка файлов

В предыдущих задачах этого параграфа все данные в файле были однотипными: числа или символьные строки. Рассмотрим задачу обработки файла со смешанными данными, которую мы решали в § 4 на языке Python.

Текстовый файл содержит данные о собаках, привезённых на выставку. В каждой строке в символьном виде записаны: кличка собаки, её возраст (целое число) и порода, разделённые точками с запятой. Например:

```
Мухтар;4;немецкая овчарка
```

Нужно вывести в другой файл сведения о собаках, которым меньше 5 лет.

В этой задаче разделитель данных — точка с запятой, а не пробел. Кроме того, внутри строки могут встречаться пробелы. Поэтому будем обрабатывать данные в файле построчно. Получается такая программа на псевдокоде:

```
while( не конец файла Fin ) {
    // прочитать строку из файла Fin
    // разобрать строку - выделить возраст
    if( возраст < 5 )
        // записать строку в файл Fout
}
```

Здесь, как и раньше, Fin и Fout — файловые потоки, открытые на чтение и на запись соответственно.

Будем считать, что все данные корректны, т. е. первая точка с запятой отделяет кличку от возраста, а вторая — возраст от породы.

Сначала разберёмся с чтением символьных строк из файла. Для этой цели в языке C++ используется функция `getline`:

```
getline( Fin, s );
```

Она читает из файлового потока Fin символьную строку и загружает её в переменную `s` типа `string`. В отличие от оператора `>>` эта функция может прочитать строку, содержащую пробелы. Чтение останавливается на символе перехода на новую строку `"\n"`.

Функция `getline` возвращает результат, показывающий, успешно ли выполнено чтение. Если функция вернула ненулевое значение, то чтение завершилось удачно, если нулевое — операция не выполнена. Поэтому очень удобно использовать результат функции `getline` для обнаружения конца файла. Цикл «пока не конец файла» можно организовать так:

```
string s;
while( getline( Fin, s ) ) {
    ...
}
```

Теперь подумаем, как обработать прочитанную строку. Нам нужно выделить возраст собаки — часть, которая ограничена первым и вторым знаками «точка с запятой». Разбор строки можно выполнить следующим образом:

```
// найти в строке знак ;
// удалить из строки кличку вместе с первым знаком ;
// найти в строке знак ;
// выделить возраст перед знаком ;
// преобразовать возраст в числовой вид
```

Запишем эти операции на языке C++:

```
int pos = s.find( ';' );           // (1)
string s1 = s.substr( pos+1 );     // (2)
int age = stoi( s1 );             // (3)
```

Сначала определяем индекс первой точки с запятой и записываем его в переменную pos (строка 1). Затем в новую строку s1 записываем всю исходную строку s без клички собаки (строка 2). Например, при обработке строки

Мухтар;4;немецкая овчарка

в переменную s1 попадёт значение

4;немецкая овчарка

Далее (строка 3) с помощью функции stoi из стандартной библиотеки преобразуем символьную строку в число. Эта функция останавливается на первом символе, для которого такое преобразование не удалось. Поэтому из всей строки s1 только первая часть (до точки с запятой) будет преобразована в число. В данном примере мы получим число 4 в переменной age.

Осталось вывести результат в файловый поток Fout, если возраст собаки меньше пяти лет:

```
if( age < 5 )
    Fout << s << endl;
```

Приведём полностью основной цикл:

```
while ( getline( Fin, s ) ) {
    int pos = s.find( ';' );
    string s1 = s.substr( pos+1 );
    int age = stoi( s1 );
    if( age < 5 )
        Fout << s << endl;
}
```

Как передать имя файла программе?

В предыдущих программах, работающих с файлами, мы указывали имя файла прямо в программе. Чтобы было возможно работать с другим файлом, нужно было менять код программы и заново компилировать её (переводить в машинный код).

Программа будет значительно полезнее, если имя файла можно будет изменить, не меняя программу. При этом имя файла нужно как-то передать программе. Это можно сделать с помощью командной строки, т. е. строки, с помощью которой программа запускается в командном процессоре.

Пусть исполняемый файл нашей программы называется `myproga` (в операционной системе Windows у него будет расширение `.exe`). Тогда можно запустить его из командной строки, записав название нужного файла после имени программы (через пробел):

```
myproga input.txt
```

Чтобы «поймать» и использовать имя файла в программе, мы укажем два параметра основной программы `main`:

```
int main( int argc, char* argv[] )  
{  
    ...  
}
```

По традиции они называются `argc` и `argv`, от английских выражений *argument count* (количество аргументов) и *argument vector* (вектор аргументов, значения аргументов).

С первым параметром `argc` всё ясно — это целое число (тип `int`), а вот тип второго задан необычно: `char* argv[]`. Мы уже знаем, что `argv[]` обозначает параметр-массив с именем `arg`. Его элементы относятся к типу `char*`. Если `char` — это символ, то `char*` — это адрес символа (указатель на символ). Таким образом, каждый элемент массива `argv` — это адрес какого-то символа в памяти. На самом деле `argv[i]` — это адрес в памяти *i*-го аргумента, переданного основной программе. Вот эта программа выводит на экран все аргументы командной строки:

```
int main( int argc, char* argv[] )  
{  
    for( int i = 0; i < argc; i++ )  
        cout << "argv[" << i << "] = "  
            << argv[i] << endl;  
    cin.get();  
}
```

При запуске этой программы из командной строки
`myproga input.txt output.dat`

мы увидим (в системе Windows) примерно такой результат:

```
argv[0] = C:\C++\projects\myproga.exe
argv[1] = input.txt
argv[2] = output.dat
```

Итак, мы передали в программу два аргумента в командной строке, они записаны в элементы массива argv[1] и argv[2]. А самый первый элемент массива, argv[0], всегда указывает на полный адрес запускаемой программы.

Вот пример программы, которая принимает в командной строке имени входного и выходного файлов, читает из первого файла два числа и записывает во второй файл их сумму:

```
int main( int argc, char* argv[] )
{
    ifstream Fin( argv[1] );
    int a, b;
    Fin >> a >> b;

    ofstream Fout( argv[2] );
    Fout << a << "+" << b << "=" << a + b;
}
```

Конечно, в профессиональной программе нужно ещё обработать возможные ошибки (не указаны одно или оба имени файла, файла для чтения не существует, файл для записи имеет атрибут «только для чтения» и т. п.).

Выводы

- Работа с файлами в C++ выполняется с помощью потоков. Для использования файловых потоков необходимо подключить библиотеку `fstream`.
- Перед началом работы с файлом поток нужно открыть методом `open`, а после окончания работы — закрыть методом `close`.
- Метод входного потока `eof` — это логическая функция, которая возвращает значение `True`, если данные в файле закончились.
- Функция `getline` читает из входного потока символьную строку в переменную типа `string` и возвращает `0`, если чтение прошло неудачно.
- Программа может получить аргументы командной строки через параметры функции `main`. Первый параметр — это количество аргументов командной строки, а второй — массив этих аргументов (символьных строк).
- Аргумент командной строки с индексом `0` — это полный адрес запускаемой программы.



Вопросы и задания

- Сравните методы работы с файлами в языках Python и C++.
- Используя дополнительные источники, изучите другие методы файловых потоков в C++.
- Сравните два способа обнаружения конца файла в программе на C++.
- Найдите ошибку в программе, которая читает и выводит на экран данные из файла:

```
ifstream Fin;  
string s;  
while( Fin >> s )  
    cout << s;
```

- В файле записаны целые числа. Напишите программу, которая выводит в другой файл все числа-палиндромы (которые читаются одинаково в обоих направлениях, как, например, 373 и 2442).
- В файле записаны целые числа. Напишите программу, которая выводит в другой файл все простые числа, содержащиеся в исходном файле.
- В файле записан текст, содержащий числа. Напишите программу, которая переписывает в другой файл все строки текста, где есть числа.
- В файле записан текст. Напишите программу, которая очищает его от знаков препинания, оставляя только слова.
- Проект.* В файле записан текст. Напишите программу, которая считает среднюю длину слов в этом тексте. Используйте эту программу для анализа текстов разных авторов и на разных языках (их можно найти в сети Интернет).
- Проект.* В файле записан текст, выровненный по левому краю. Напишите программу, которая форматирует текст с выравниванием по ширине в пределах 80 позиций.
- Проект.* В файле записан текст. Напишите программу, которая считает количество слов разной длины и выводит отдельно количество слов из одной буквы, из двух, и т. д. до 30.
- *12. Доработайте программу из предыдущего задания так, чтобы она строила в графическом окне диаграмму распределения слов по длинам. Сравните такие диаграммы для текстов разных авторов и текстов, написанных на разных языках. Сделайте выводы: можно ли использовать такой метод для автоматического определения языка и авторства текста?
- **13. *Проект.* В файле в столбик записаны слова. Напишите программу, которая пытается составить из них кроссворд: расположить одно слово вертикально, а остальные — горизонтально. Постарайтесь использовать как можно больше слов из файла.

14. В файле в два столбика записаны данные, разделённые пробелами: сначала дата измерения, а потом — результат измерения уровня радиации (вещественное число). Напишите программу, которая записывает в другой файл день (или дни, если их несколько), когда уровень радиации был наибольшим.
15. В текстовом файле записаны данные о занятости ячеек в камере хранения в течение суток. В каждой строке файла записан номер ячейки (целое число от 1 до 100), затем — время в формате чч:мм (часы, потом минуты как двузначные числа), а в конце — знак «+», если положили вещи в ячейку или «-», если освободили ячейку. Все данные в строке разделены пробелами. Напишите программу, которая определяет среднее время, в которое одна ячейка была занята в течение этих суток.
- *16. *Проект.* Напишите две программы, одна из которых шифрует файл с помощью шифра Виженера (с заданным ключевым словом), а вторая — расшифровывает его. Информацию о шифре Виженера найдите в дополнительных источниках.
17. В текстовом файле записаны данные об участниках соревнований (их не более 100): фамилия, имя, отчество, год рождения и пять оценок, выставленных членами жюри. Все данные разделены точками с запятой, например так:

Иванов;Иван;Иванович;1991;92;87;89;93;90

Напишите программу, которая обрабатывает эти данные и записывает результаты в другой файл. Итоговая оценка каждого участника равна среднему арифметическому трёх оценок судей, полученных после отбрасывания наименьшей и наибольшей из пяти оценок. Список должен быть отсортирован по убыванию итоговой оценки участника, слева нужно вывести место, которое он занял. Например:

1. Иванов Иван Иванович, 1999 г.р. - 90.33
2. ...

- *18. В предыдущей задаче измените порядок вывода: нужно подвесить итоги отдельно по каждой возрастной группе: младше 18 лет, 18–25 лет, 25–40 лет и старше 40 лет.

- *19. В файле находятся сведения об участниках лыжных гонок. Для каждого из них в одной строке записаны фамилия, имя, отчество и год рождения, разделённые точкой с запятой, например:

Иванов;Иван;Иванович;1991

Напишите программу, которая выводит в графическое окно диаграмму распределения участников по возрастам (годам рождения).

20. В файле записаны данные о сотрудниках компании «Соки — воды», в каждой строке — фамилия, имя, отчество и дата рождения, разделённые точками с запятой, например:

Иванов; Иван; Иванович; 23.01.1991

Напишите программу, которая выводит в другой файл список, отсортированный по убыванию возраста (самый пожилой человек должен быть на первой позиции списка).

*21. В файле записаны результаты участников олимпиады по информатике (их не более 500). Данные каждого из них записаны в одной строке: фамилия, имя, отчество, класс, сумма баллов. Например:

Иванов; Иван; Иванович; 8; 362

В олимпиаде приняли участие ученики 5–11 классов. Запишите в отдельные файлы результаты олимпиады по каждой параллели в формате языка HTML (изучите самостоятельно команды языка HTML для работы с таблицами). Списки должны быть отсортированы по убыванию набранной суммы баллов, слева нужно добавить нумерацию, например:

```
<table>
<tr>
<td>1</td><td>Иванов Иван Иванович</td>
<td>8</td><td>362</td>
</tr>
<tr>
<td>2</td><td>Семёнов Иван Петрович</td>
<td>8</td><td>360</td>
</tr>
...
</table>
```

*22. В веб-странице, которая представляет собой текстовый файл на языке HTML, записаны данные об участниках олимпиады (см. предыдущее задание). Напишите программу, которая записывает в другой файл те же данные в текстовом формате:

1; Иванов Иван Иванович; 8; 362

2; Семёнов Иван Петрович; 8; 360

...

23. В конкурсе участвуют 50 исполнителей, телезрители голосуют за понравившегося участника, отправляя SMS-сообщение с его фамилией. Все эти сообщения сохранены в файле, по одному сообщению (по одной фамилии) в строке, таких строк может быть очень много. Напишите программу, которая выводит список исполнителей в порядке уменьшения популярности.

§ 16

Целочисленные алгоритмы

Ключевые слова:

- решето Эратосфена
- разряды
- длинные числа
- перенос

Решето Эратосфена

Напишем программу, которая находит все простые числа в заданном диапазоне (от 2 до N), используя алгоритм «решето Эратосфена» (см. § 5):

- 1) выписать все числа от 2 до N ;
- 2) начать с $k = 2$;
- 3) вычеркнуть все числа, кратные k , начиная с k^2 ;
- 4) найти следующее невычеркнутое число и присвоить его переменной k ;
- 5) повторять шаги 3 и 4, пока $k^2 \leq N$.

Данные о том, какие числа «вычеркнуты», а какие — нет, будем хранить в логическом массиве:

```
const int N = 100;
bool A[N+1];
```

Мы будем использовать элементы массива с индексами от 2 до N , поэтому выделили на один элемент больше (нумерация элементов массива в C++ начинается с нуля, а $A[0]$ и $A[1]$ мы не используем).

Сначала заполняем весь массив истинными значениями (ни одно число не вычеркнуто):

```
for( int i = 2; i <= N; i++ )
    A[i] = true;
```

Затем в основном цикле выполняется алгоритм, описанный в § 5:

```
int k = 2;
while( k*k <= N ) {
    if( A[k] ) {
        int i = k*k;
        while( i <= N ) {
            A[i] = false;
            i += k;
        }
    }
    k++;
}
```

После завершения этого цикла невычеркнутыми остались только простые числа, для них соответствующий элемент массива содержит истинное значение. Эти числа нужно вывести на экран:

```
for( int i = 2; i <= N; i++ )
    if( A[i] )
        cout << i << " ";
```

«Длинные» числа

В отличие от языка Python в C++ нет встроенных средств для выполнения расчётов с очень большими целыми числами. А работать с такими числами нужно: современные алгоритмы шифрования используют достаточно длинные ключи — числа длиной 256 бит и больше. С ними необходимо выполнять разные операции: складывать, умножать, находить остаток от деления.

Вопрос состоит в том, как хранить такие числа в памяти, где для целых чисел отводятся ячейки значительно меньших размеров (обычно до 64 бит). Ответ достаточно очевиден: нужно «разбить» длинное число на части так, чтобы можно было хранить его в нескольких ячейках памяти.

0 «Длинное» число — это число, которое не помещается в переменную ни одного из стандартных типов данных языка программирования. Алгоритмы работы с длинными числами называют «длинной арифметикой».

Для хранения длинного числа удобно использовать массив целых чисел. Например, число 12345678 можно записать в массив с индексами от 0 до 9 таким образом:

	0	1	2	3	4	5	6	7	8	9
A	1	2	3	4	5	6	7	8	0	0

Такой способ имеет ряд недостатков:

- нужно где-то хранить длину числа, иначе числа 12345678, 123456780 и 1234567800 будет невозможно различить;
- неудобно выполнять арифметические операции, которые начинаются с младшего разряда;
- память расходуется неэкономно, потому что в одном элементе массива хранится только один разряд — число от 0 до 9.

Чтобы избавиться от первых двух проблем, достаточно «развернуть» массив наоборот: так, чтобы младший разряд находился в $A[0]$. В этом случае на рисунках удобно использовать обратный порядок элементов:

	9	8	7	6	5	4	3	2	1	0
A	0	0	1	2	3	4	5	6	7	8

Теперь нужно найти более экономичный способ хранения длинного числа. Например, разместим в одной ячейке массива три разряда числа, начиная справа:

	9	8	7	6	5	4	3	2	1	0
A	0	0	0	0	0	0	0	12	345	678

Здесь использовано равенство

$$12345678 = 12 \cdot 1000^2 + 345 \cdot 1000^1 + 678 \cdot 1000^0.$$

Фактически мы представили исходное число в системе счисления с основанием 1000.

Сколько разрядов можно хранить в одной ячейке массива? Это зависит от её размера. Например, если ячейка занимает 4 байта и число хранится со знаком, то допустимый диапазон её значений: от $-2^{31} = -2\ 147\ 483\ 648$ до $2^{31} - 1 = 2\ 147\ 483\ 647$. В такой ячейке можно хранить до 9 разрядов десятичного числа, т. е. использовать систему счисления с основанием 1 000 000 000.

Однако нужно учитывать, что с числами будут выполняться арифметические операции, результат которых тоже должен «помещаться» в ячейку памяти. Например, если надо умножать разряды этого числа на число $k < 100$, то в 32-битной ячейке можно хранить не более 7 разрядов.

Напишем программу, которая точно вычисляет значение факториала $100! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot 99 \cdot 100$ и выводит его на экран в десятичной системе счисления. Это число состоит более чем из сотни цифр и явно не помещается в одну ячейку памяти, даже типа `int64_t`.

«Длинное» число будем хранить в целочисленном массиве A. Чтобы определить необходимую длину массива, заметим, что

$$1 \cdot 2 \cdot 3 \cdots \cdot 99 \cdot 100 < 100^{100}.$$

Число 100^{100} содержит 201 цифру, поэтому число $100!$ содержит не более 200 цифр. Если в каждый элемент массива записывать 6 цифр, для хранения всего числа потребуется не более 34 ячеек:

```
const int N = 33;
int A[N+1];
```

Чтобы вычислить $100!$, нужно сначала присвоить «длинному» числу значение 1, а затем последовательно умножать его на все числа от 2 до 100. Запишем эту идею на псевдокоде, обозначив через {A} длинное число, находящееся в массиве A:

```
{A} = 1;
for( int k = 2; k <= 100; k++ )
    {A} *= k;
```

Записать в «длинное» число единицу — это значит присвоить элементу A[0] значение 1, а в остальные ячейки записать нули:

```
A[0] = 1;
for( int i = 1; i <= N; i++ )
    A[i] = 0;
```

Но проще всего заполнить массив прямо при объявлении:

```
int A[N+1] = {1};
```

Мы явно указали, что в $A[0]$ нужно записать единицу, а остальные элементы массива (для которых не указаны начальные значения) будут автоматически заполнены нулями.

Таким образом, остаётся научиться умножать длинное число на «короткое» ($k \leq 100$). «Короткими» обычно называют числа, которые помещаются в переменную одного из стандартных типов.

Попробуем сначала выполнить такое умножение на примере. Предположим, что в каждой ячейке массива хранятся 6 цифр длинного числа, т. е. используется система счисления с основанием $base = 1\,000\,000$. Тогда число 12345678901734567 хранится в трёх ячейках:

	2	1	0
A	12345	678901	734567

Пусть $k = 3$. Начинаем умножать с младшего разряда: $734567 \cdot 3 = 2203701$. В нулевом разряде могут находиться только 6 цифр, значит, старшая двойка перейдёт в перенос в следующий разряд.

В программе для выделения переноса p можно использовать деление нацело на основание системы счисления $base$ (с отбрасыванием остатка). Сам остаток — это то, что остается в текущем разряде. Поэтому получаем:

```
const int base = 1000000;
int newValue = A[0]*k;
A[0] = newValue % base;
int p = newValue / base;
```

Для следующего разряда будет всё то же самое, только к произведению нужно добавить перенос из предыдущего разряда, который был записан в переменную p . Приняв в самом начале $p = 0$, запишем умножение длинного числа на короткое в виде цикла по всем элементам массива, от $A[0]$ до $A[N]$:

```
int p = 0;
for( int i = 0; i <= N; i++ ) {
    int newValue = A[i]*k + p;
    A[i] = newValue % base;
    p = newValue / base;
}
```

В свою очередь, эти действия нужно выполнить в другом (внешнем) цикле для всех k от 2 до 100:

```
for( int k = 2; k <= 100; k++ ) {
    ...
}
```

После этого в массиве A будет находиться искомое значение 100!, остаётся вывести его на экран. Нужно учесть, что в каждой ячейке хранятся 6 цифр, поэтому, например, в массиве

A	2	1	0
	1	2	3

на самом деле хранится значение 1000002000003, а не 123. Кроме того, старшие нулевые разряды выводить на экран не надо. Поэтому при выводе требуется:

- найти первый (старший) ненулевой разряд числа;
- вывести это значение без лидирующих нулей;
- вывести все следующие разряды, добавляя лидирующие нули до 6 цифр.

Поскольку мы знаем, что всё число не равно нулю, старший ненулевой разряд можно найти в таком цикле¹⁾:

```
int i = N;
while( not A[i] )
    i--;
```

Старший разряд выводим обычным образом (без лидирующих нулей):

```
cout << A[i];
```

Остальные разряды будем выводить в цикле. Каждый элемент массива выводится в шести позициях, пустые позиции заполняются нулями. Такой формат вывода можно установить с помощью манипуляторов потока (нужно подключить библиотеку iomanip):

```
#include <iomanip>
...
cout << setfill('0');           (1)
while( i >= 0 ) {
    cout << setw(6) << A[i];   (2)
    i--;
}
```

В строке 1 устанавливаем новый символ-заполнитель — цифру ноль. Манипулятор setw задаёт ширину поля вывода — 6 символов (строка 2). Этот манипулятор нужно применять перед выводом каждого числа.

¹⁾ Подумайте, что изменится, если выводимое число может быть нулевым.

Выводы

- Решето Эратосфена позволяет эффективно найти все простые числа в диапазоне от 2 до N .
- «Длинное» число — это число, которое не помещается в переменную ни одного из стандартных типов данных языка программирования. Алгоритмы работы с длинными числами называют «длинной арифметикой».
- Длинные числа в C++ удобно разбивать на части и хранить эти части в массиве. Один из вариантов — представить число в системе счисления с большим основанием, например 1 000 000.
- Операции с длинными числами выполняются отдельно с каждым разрядом, учитывая перенос из одного разряда в другой.



Вопросы и задания

1. Напишите программу, которая получает чётное число, большее двух, и выводит все способы представления его в виде суммы двух простых чисел¹⁾. При вводе нечётного числа нужно сообщить об ошибке и повторить ввод.
2. Напишите программу, которая получает нечётное число, большее 5, и выводит все способы представления его в виде суммы трёх простых чисел²⁾. При вводе чётного числа нужно сообщить об ошибке и повторить ввод.
3. В каких случаях необходимо применять «длинную арифметику»?
4. Какое максимальное число можно записать в ячейку размером 64 бита? Рассмотрите варианты хранения чисел со знаком и без знака.
5. Можно ли использовать для хранения длинного числа символьную строку? Какие проблемы при этом могут возникнуть?
6. Почему неудобно хранить длинное число, записывая старшую значащую цифру в начало массива?
7. Почему неэкономично хранить по одной цифре в каждом элементе массива?
8. Сколько разрядов числа можно хранить в одной 16-битной ячейке?
9. Докажите, что в приведённой в параграфе программе вычисления $100!$ не будет переполнения при использовании 32-битных целых переменных.
10. Можно ли в приведённой в параграфе программе в одной ячейке массива хранить 9 цифр «длинного» числа? Ответ обоснуйте.
11. Без использования программы попробуйте определить, сколько нулей стоит в конце числа $100!$.

¹⁾ Гипотеза К. Гольдбаха утверждает, что такое представление всегда возможно. На момент написания пособия (2018 г.) она не доказана, но и не опровергнута.

²⁾ В 2013 году математик Х. Гельфготт доказал это утверждение, которое называют тернарной гипотезой Гольдбаха.

12. Соберите всю программу и вычислите $100!$. Сколько цифр входит в это число?
13. Объясните, какие проблемы возникают при выводе «длинного» числа. Как их можно решать?
- *14. Попробуйте предложить другой способ вывода «длинного» числа, не использующий манипуляторы потока. Какой способ вам больше нравится? Почему?
15. Оформите вывод всего «длинного» числа на экран в виде отдельной процедуры. Учтите, что число может быть нулевым.
16. Сравните быстродействие алгоритма «решето Эратосфена» в языках Python и C++. Исследуйте, зависит ли разница в скорости от размера массива.
17. Напишите подпрограммы, которые вводят «длинное» число из файла и записывают «длинное» число в файл.
18. Напишите программу, которая вводит из файла два «длинных» целых числа и вычисляет:
- 1) сумму этих чисел;
 - 2) разность этих чисел;
 - 3) произведение этих чисел;
 - 4) частное этих чисел;
 - 5) остаток от деления первого числа на второе.

Результат работы программы нужно записать в новый файл.

- *19. Напишите программу, которая вводит «длинное» число из файла и вычисляет его целочисленный квадратный корень.
- **20. Напишите программу, которая вводит «длинное» число из файла и раскладывает его на два простых сомножителя. Используйте метод Ферма (найдите его описание в дополнительных источниках).

§ 17

Динамические массивы и словари

Ключевые слова:

- динамический массив
- ключ
- указатель
- значение
- контейнер
- перебор элементов
- вектор
- сортировка по ключу
- словарь
- итератор

Зачем это нужно?

Когда мы объявляем массив, место для него выделяется во время компиляции, т. е. до выполнения программы. Иногда такой способ не подходит, потому что размер массива данных заранее неизвестен, например вводится с клавиатуры или передаётся по компьютерной сети.

В этом случае есть два варианта: 1) выделить заранее максимально большой блок памяти и 2) выделять память уже во время выполнения программы (т. е. *динамически*), когда станет известен необходимый размер массива.

Таким образом, мы пришли к понятию динамических структур данных, которые позволяют во время выполнения программы:

- создавать новые объекты в памяти;
- изменять их размер;
- удалять объекты из памяти, когда они не нужны.

Динамические массивы

В языке C++ можно выделять память под массив во время работы программы. Для этого нужно ввести специальную переменную, в которую можно записывать адрес нового массива, она называется *указателем*:

```
int *A;
```

Это ещё не массив, а просто ячейка, в которую можно записать адрес нового массива (или адрес любой переменной типа `int`). Обращение `A[0]` или `A[55]` не запрещается, но бессмысленно и опасно, ведь неизвестно, какое значение оказалось в ячейке `A`!

Память для массива выделяется с помощью специального оператора `new` (в переводе с английского — *новый*):

```
int N = 10;
A = new int[N];
```

По этой команде в динамической памяти (её также называют *кучей*, англ. *heap*) размещается массив из `N` ячеек для хранения целых чисел и его адрес записывается в переменную-указатель `A`. Заметим, что в квадратных скобках можно записывать любое арифметическое выражение, которое может быть вычислено в момент выполнения команды. Например, значение `N` мы могли ввести с клавиатуры или принять через компьютерную сеть.

Все ячейки массива `A`, который мы только что разместили в памяти, будут содержать «мусорные» значения. При желании можно перечислить начальные значения всех (или нескольких первых) элементов в фигурных скобках:

```
int* A = new int[10]{1, 2, 3};
```

Здесь из 10 элементов заданы значения первых трёх, а остальные будут заполнены нулями.

Теперь, когда выделена память для массива и её адрес сохранён в переменной `A`, обращаться к этому массиву можно точно так же, как и к «обычному» массиву `A`:

```
for( int i = 0; i < N; i++ ) {
    A[i] = i*i;
    cout << A[i] << "";
}
```

По этому фрагменту программы никак нельзя догадаться, что на самом деле А — это динамический массив.

Когда работа с массивом закончена, и он больше не нужен, можно удалить его из памяти с помощью оператора **delete** (от англ. *delete* — удалить):

```
delete [] A;
```

Квадратные скобки указывают на то, что из памяти удаляется именно массив, а не отдельная переменная.

После освобождения памяти работать с массивом А нельзя, потому что адрес в переменной А недействителен.

Если случайно удалить массив второй раз, произойдёт серьёзная ошибка работы с памятью. Такие ошибки не обнаруживает компилятор, поэтому иногда их очень сложно найти.

Указатель можно использовать повторно. Для этого нужно выделить новый блок памяти с помощью оператора **new** и записать его адрес в тот же указатель А. Но это будет уже другой массив, расположенный в другой области памяти.

Тип **vector** из библиотеки STL

В языке C++ есть стандартная библиотека шаблонов STL (англ. *STL: Standard Template Library*). Она вводит множество различных типов данных и функций для их обработки. В эту библиотеку входит тип данных **vector**, который фактически представляет собой динамический массив с возможностью расширения.

Объект типа **vector** — это *контейнер*, т. е. объект, который может содержать *коллекцию* (набор) данных другого типа. В библиотеке STL есть готовые методы для работы с контейнерами, в том числе для добавления, перебора и удаления элементов.

Чтобы использовать массивы типа **vector**, нужно подключить библиотеку **vector**:

```
#include <vector>
using namespace std;
```

Все объекты и методы библиотеки STL введены в стандартном пространстве имён **std**, поэтому мы сразу подключаем его для удобства работы с векторами.

В отличие от списков языка Python, все данные в контейнере **vector** должны быть одного типа — это ускоряет работу с ними, потому что размеры всех элементов одинаковы и легко вычислить, где расположен элемент с заданным индексом.

Динамический массив (вектор) можно составить из данных любого типа. Вектор для целых чисел (элементов типа `int`) можно объявить так:

```
vector <int> A;
```

Тип значений, которые будет содержать контейнер, записывают в угловых скобках. Данные других типов в этом массиве хранить нельзя.

Количество элементов в массиве можно определить в любой момент с помощью метода `size`:

```
cout << A.size();
```

Таким образом, вектор «знает свой размер» (как и список в языке Python).

При создании вектора в нём нет ни одного элемента, его размер равен нулю. Можно сразу определить размер вектора и задать для его элементов одинаковые начальные значения:

```
vector <int> A(10, 1);
```

Этот вектор содержит 10 элементов, равных единице. Если начальное значение не указано, все элементы обнуляются.

Метод `push_back` добавляет новый элемент в конец вектора:

```
for( int i = 0; i < N; i++ )  
    A.push_back( i+1 );
```

а метод `pop_back` удаляет последний элемент. Вот так можно удалить все элементы вектора:

```
while( A.size() )  
    A.pop_back();
```

Работают с вектором так же, как и с обычным массивом. Например, этот цикл выводит вектор на экран:

```
for( int i = 0; i < A.size(); i++ )  
    cout << A[i] << " ";
```

В современных версиях языка C++, начиная с C++11, можно использовать и такой цикл:

```
for( int x: A )  
    cout << x << endl;
```

При этом все элементы вектора по очереди оказываются в переменной `x` и выводятся на экран.

Попытка обращения к несуществующему элементу (с неверным индексом) не считается синтаксической ошибкой, но может вызвать нестабильную работу программы и аварийное завершение. В случае

записи значения по неверному адресу испортится «чужая» область памяти, не принадлежащая вектору.

Для изменения размера вектора применяют метод `resize`:

```
A.resize( 15 );
```

Если массив расширяется (новый размер больше, чем существующий), все новые элементы по умолчанию заполняются нулями.

Метод `clear` очищает вектор (удаляет все его элементы). После этого метод `empty` вернёт значение `true` — вектор пустой:

```
A.clear();
if( A.empty() )
    cout << "Тут совсем ничего нет!" ;
```

Более подробную информацию об этих и других методах работы с векторами вы можете найти в дополнительной литературе или в Интернете.

Итераторы

Итераторы (курсоры, указатели) — это специальные объекты, которые позволяют перебрать все элементы контейнера. Итератор указывает на текущий объект в контейнере, с которым мы можем что-то сделать, например вывести его данные в файл или изменить их.

Итератор для вектора целых значений нужно объявить так:

```
vector <int>::iterator it;
```

Здесь введён итератор с именем `it`. Он предназначен для контейнеров типа `vector <int>`.

Итератор — это специальный *указатель* на элемент вектора. Если итератор `it` установлен на какой-то элемент вектора, обратиться к этому элементу можно как `*it`, эта операция называется разыменованием (как для указателей):

```
it = A.begin();
*it = 100;           // то же, что A[0] = 100;
cout << *it;        // то же, что cout << A[0];
```

Метод `begin` (от англ. *begin* — начало) возвращает итератор на первый по счёту элемент контейнера. Поэтому здесь `*it` — это то же самое, что `A[0]`, и последние две строки приведённого фрагмента равносильны таким операторам:

```
A[0] = 100;
cout << A[0];
```

Операция `++` для итератора означает переход к следующему элементу контейнера, а метод `end` возвращает указатель на элемент, который следует за последним и уже не принадлежит вектору (рис. 2.3).

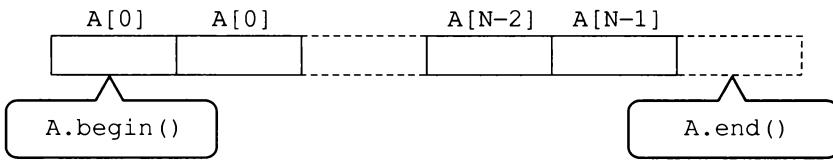


Рис. 2.3

Если вектор *A* не содержит ни одного элемента, значения *A.begin()* и *A.end()* равны.

Предположим, что итератор *it* указывает на начало вектора (равен *A.begin()*). При увеличении с помощью оператора *++* этот итератор в какой-то момент обязательно станет равен *A.end()*. Этот факт удобно использовать как условие окончания перебора в цикле. Вот так можно вывести все элементы вектора на экран:

```
for( it = A.begin(); it != A.end(); it++ )
    cout << *it << " ";
```

а вот так — заполнить все элементы значением 2018:

```
for( it = A.begin(); it != A.end(); it++ )
    *it = 2018;
```

При переборе элементов контейнера с помощью итераторов в условии цикла принято везде использовать отношение *!=*, а не *<*, как для массивов. Дело в том, что операции *==* и *!=* определены для всех типов итераторов, а другие операции сравнения (*<*, *<=*, *>*, *>=*) — только для итераторов с произвольным доступом (в том числе для типа **vector**).

В версии C++11 (и более поздних) можно объявлять итератор в заголовке цикла с помощью описателя **auto**. При этом нужный тип переменной определяется автоматически:

```
for( auto it = A.begin(); it != A.end(); it++ )
    *it = 2019;
```

Транслятор «сообразит», что переменная *it* должна быть итератором для вектора *A*. Заметим, что в этом случае переменная *it* определена только внутри цикла.

Существуют *константные* итераторы (от англ. *constant* — постоянный), которые позволяют только читать, но не изменять данные контейнера. Для этого при объявлении вместо слова **iterator** нужно написать **const_iterator**:

```
vector<int>::const_iterator it;
```

Операция изменения значения через константный итератор запрещена:

```
*it = 2011; // ошибка!
```

Такие ограниченные итераторы нужны для того, чтобы всем, кто будет разбираться в программе, было понятно, что эти данные не будут изменяться.

Константные итераторы, указывающие на начало и конец вектора, можно получить с помощью методов `cbegin` и `cend` (вместо `begin` и `end`). Например, вывод элементов вектора на экран можно оформить так:

```
for( auto it = A.cbegin(); it != A.cend(); it++ )  
    cout << *it << " ";
```

С помощью итераторов выполняются все основные операции с векторами. Для вставки элементов в вектор используется метод `insert`. Чтобы добавить элемент в самое начало вектора, этому методу нужно передать указатель на начало массива и значение нового элемента:

```
A.insert( A.begin(), 101 );
```

Добавив к указателю на первый элемент натуральное число, мы получим итератор, указывающий на элемент с заданным индексом. Вот так можно вставить новый элемент перед элементом с индексом 3:

```
A.insert( A.begin() + 3, 2019 );
```

Для удаления элемента из вектора нужно передать методу `erase` итератор — указатель на удаляемый элемент:

```
A.erase( A.begin() + 2 );
```

Эта команда удаляет третий по счёту элемент, имеющий индекс 2. Чтобы удалить несколько элементов, необходимо указать два итератора — указатели на первый удаляемый элемент и на элемент, который стоит за удаляемой частью:

```
A.erase( A.begin() + 2, A.begin() + 5 );
```

Здесь из вектора `A` удаляются элементы `A[2]`, `A[3]` и `A[4]`. А элемент `A[5]`, на который указывает второй итератор, остаётся в массиве!

Зачем нужны итераторы

Возможно, при первом знакомстве с итераторами не совсем понятно, зачем они вообще нужны. Действительно, элементы вектора доступны по индексам, поэтому можно было бы запрограммировать все описанные выше операции, используя индексы вместо итераторов. И в этом случае код, наверное, получился бы проще и понятнее. Какие преимущества дают итераторы?

Дело в том, что итераторы — это очень важная идея, определяющая устройство всей библиотеки STL. Эта библиотека содержит много разных типов данных (контейнеров). На первый взгляд любой алгоритм (например, поиск элемента в коллекции) для каждого типа данных нужно программировать заново, учитывая внутреннее устройство контейнера. Но в библиотеке STL используется другой подход. Его основ-

ная идея состоит в том, что для программирования многих стандартных алгоритмов совершенно не нужно знать, как устроен контейнер. Например, для поиска в любом контейнере достаточно уметь перебирать все его элементы и определять значение каждого из них. Как раз эти возможности и обеспечивают итераторы. Поэтому можно написать универсальный алгоритм поиска, принимающий три аргумента: два итератора (указатели на начало и конец области поиска) и искомое значение.

Таким образом, удаётся отделить алгоритмы от способа хранения данных. Итераторы играют роль связующего звена между этими двумя частями STL: они обеспечивают доступ к данным контейнеров, необходимый для работы алгоритмов.

Пусть A — это любой контейнер для хранения целых чисел, итераторы которого пригодны для поиска. Найти значение 2018 в этом контейнере можно так:

```
#include <algorithm>
...
auto it = find( A.begin(), A.end(), 2018 );
if( it == A.end() )
    cout << "Не нашли 2018..." << endl;
else
    cout << "Ура! Нашли 2018!" << endl;
```

Для использования функции `find` нужно подключить библиотеку `algorithm`. Мы передаём этой функции указатели на начало и конец коллекции, т. е. поиск выполняется среди всех её элементов.

Функция `find` возвращает итератор, указывающий на найденный элемент. Если такого элемента нет, функция вернёт итератор, равный `A.end()`. Он указывает за пределы контейнера и говорит о том, что поиск закончился неудачно.

Отметим, что приведённый выше код будет работать для любых коллекций целых чисел, даже для тех, в которых невозможен доступ к элементу по индексу.

Другие функции библиотеки STL также используют итераторы-аргументы. Например, следующий вызов функции `fill` заполняет первые 10 элементов контейнера A значением -1:

```
fill( A.begin(), A.begin() + 10, -1 );
```

Алгоритм сортировки требует, чтобы итераторы контейнера обеспечивали доступ к элементам по индексу. Например, вот этот вызов функции `sort` сортирует всю коллекцию по возрастанию:

```
sort( A.begin(), A.end() );
```

При этом совершенно безразлично, как именно хранятся данные в контейнере.

Словари

Как вы знаете из § 6, словарь, или *ассоциативный массив*, — это набор элементов, доступ к которым выполняется по ключу, причём ключом может быть не только целое число (индекс), но и, например, символьная строка. В библиотеке STL есть тип данных **map** (от англ. *map* — отображение), который работает как словарь.

Словарь — это контейнер, который, в отличие от вектора, содержит пары данных: каждый элемент состоит из ключа и соответствующего значения.

Для работы со словарями нужно подключить библиотеку **map**:

```
#include <map>
```

Вспомним задачу построения алфавитно-частотного словаря из § 6. В файле находится список слов, среди которых есть повторяющиеся. Каждое слово записано в отдельной строке. Требуется построить алфавитно-частотный словарь: все различные слова должны быть записаны в другой файл в алфавитном порядке, справа от каждого слова должно быть указано, сколько раз оно встречается в исходном файле.

В нашей задаче *значение* — это целое число (тип **int**), а искать его нам нужно по *ключу* — символьной строке типа **string** (слову из файла). В этом случае словарь объявляется так:

```
map <string, int> wordList;
```

В угловых скобках сначала записывается тип ключа, а затем через запятую — тип значения.

Что можно делать с таким словарём? Во-первых, можно определить, есть ли слово в словаре:

```
string word = "vulpes"
int count = wordList.count( word );
```

Значение переменной *count* будет равно 0, если слова, записанного в символьной строке *word*, ещё нет в словаре и 1, если оно уже было добавлено.

Для того чтобы увеличить счётчик для данного слова, можно обратиться к элементу по символьному индексу:

```
wordList[word]++;
```

Вставку элемента в массив выполняет метод **insert** (от англ. *insert* — вставить):

```
wordList.insert( {word, 1} );
```

В качестве аргумента этому методу передается пара «строка — целое число» (*{word, 1}*), составленная из слова *word* и числа 1 (в первый раз встретили это слово, поэтому счётчик равен 1).

Таким образом, основной цикл в нашей программе получается очень простым:

```
while( Fin >> word ) {
    if( wordList.count(word) == 1 )
        wordList[word]++;
    else
        wordList.insert( {word, 1} );
}
```

Здесь `Fin` — это входной файловый поток, а `word` — переменная типа `string`. Запись `while(Fin >> word)` обозначает «пока чтение очередного слова из файла завершается удачно».

Это цикл можно ещё упростить так:

```
while( Fin >> word )
    wordList[word]++;
```

Дело в том, что при обращении к неизвестному элементу в словаре типа `map` новый элемент будет создан автоматически! Это значит, что если слова `word` ещё не было в словаре, в словарь добавляется новый элемент. Его ключом становится слово `word`, а связанный с ним счётчик обнуляется. Затем мы сразу же увеличиваем счётчик до 1 с помощью оператора `++`.

Перебор элементов словаря

Осталась одна нерешённая задача: вывести результаты в файл. Контейнер типа `map` позволяет легко найти элемент по символьному индексу, но как перебрать все элементы контейнера, если мы заранее не знаем их ключей?

Для этого как раз и служат *итераторы*, с которыми мы уже знакомы. Итератор для нашего контейнера объявляется так:

```
map <string, int>::iterator it;
```

Здесь введён итератор с именем `it`. Он предназначен для контейнеров, состоящих из пар «строка — целое», к которым относится и наш словарь `wordList`.

Для того чтобы установить итератор на первый по счёту элемент контейнера, вызовем его метод `begin` (от англ. *begin* — начало):

```
it = wordList.begin();
```

Итератор `it` — это специальный указатель на элемент словаря. В каждом таком элементе нашего словаря есть пара: ключ и значение. Ключ — это первое поле элемента (с именем `first`), а целое значение — второе (с именем `second`). Обращаться к ним нужно с помощью оператора «стрелка», который составлен из знаков «минус» и «`>>`». Например, вывести в выходной поток `Fout` данные из очередного элемента можно так¹⁾:

```
Fout << it->first << ":" << it->second;
```

¹⁾ Вместо `it->first` можно записать `(*it).first`, но эта запись хуже читается.

Оператор `++` сдвигает итератор к следующему элементу. Если значение итератора после очередного продвижения совпало с `wordList.end()`, это значит, что все элементы коллекции уже просмотрены. Таким образом, цикл, который выводит в поток `Fout` ключи и значения всех элементов словаря, выглядит так:

```
for( auto it = wordList.begin();
      it != wordList.end(); it++)
    Fout << it->first << ":" 
    << it->second << endl;
```

Теперь можно записать всю программу целиком:

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    ifstream Fin( "input.txt" );
    string word;
    map <string, int> wordList;
    while( Fin >> word )
        wordList [word]++;
    ofstream Fout( "output.txt" );
    for( auto it = wordList.begin();
          it != wordList.end(); it++)
        Fout << it->first << ":" 
        << it->second << endl;
}
```

Обратите внимание, что здесь нужный тип итератора `it` определяется автоматически — он описан с помощью служебного слова `auto`. Это возможно, начиная с версии C++11.

Если посмотреть на файл с результатами работы этой программы, можно заметить, что слова в списке отсортированы по алфавиту, хотя мы их не сортировали. Дело в том, что контейнер `map` всегда хранит данные в отсортированном виде. Новое слово добавляется в словарь так, чтобы при переборе с помощью итераторов слова выводились в алфавитном порядке.

Программа на C++ получилась достаточно короткой, потому что мы использовали готовый тип данных `map` из библиотеки STL. В нём уже запрограммированы все необходимые функции.

Выводы

- Для выделения памяти в C++ используют оператор **new**, для удаления — оператор **delete**.
- Адрес блока памяти, выделенного под массив, записывают в специальную переменную-указатель. После этого с указателем можно работать так же, как и с обычным массивом.
- Стандартная библиотека шаблонов (STL) содержит описание различных типов данных и готовые алгоритмы для работы с ними.
- Динамический массив можно построить на основе типа данных **vector** из библиотеки STL.
- Словарь (ассоциативный массив) — это контейнер, который содержит пары «ключ — значение». Словарь в C++ строится на основе типа данных **map**.
- Для перебора элементов контейнера (вектора, словаря и др.) используют специальные объекты — *итераторы*. Итератор **begin()** указывает на первый элемент контейнера, а итератор **end()** — на элемент, следующий за последним.



Вопросы и задания

1. Приведите примеры задач, в которых использование динамических массивов даёт преимущества (какие именно?).
2. Какими возможностями должны обладать динамические структуры данных?
3. Программист забыл удалить динамический массив, память для которого выделялась внутри функции. Какие проблемы могут возникнуть из-за этой ошибки?
4. Как вы думаете, какие значения будут иметь элементы массива A?

```
int* A = new double[5]{};
```

5. Нужно ли хранить размер вектора в отдельной переменной?
- *6. Используя дополнительные источники, выясните, как удалить из вектора элемент с заданным значением (например, равный 125).
7. Как расширить вектор (контейнер **vector** из библиотеки STL) в ходе работы программы? Проверьте, не потеряются ли при этом уже записанные в нём данные.
8. Используя дополнительные источники, выясните, как работают реверсивные итераторы. Ответьте на вопросы.
 - а) Как вывести все элементы вектора на экран в обратном порядке?
 - б) Как будет отсортирован вектор A в результате выполнения этой команды?
`sort(A.rbegin(), A.rend());`
9. Используя дополнительные источники, выясните, как добавить несколько элементов в контейнер с помощью метода **insert**.
10. Используя дополнительные источники, изучите алгоритмы библиотеки STL **find_if**, **count**, **count_if**, **copy**, использующие итераторы.

11. Предложите, как можно сохранить динамический массив в файле и как потом прочитать его из файла.
12. Сравните словари в языках Python и C++.
13. Как создать словарь из готовых пар «ключ — значение», записанных в файл?
14. Выполните задания 6–9 из § 6 на C++.
- *15. Используя дополнительные источники, изучите контейнер **set** из библиотеки STL.

§ 18

Структуры

Ключевые слова:

- структура
- сортировка
- поле
- ключ
- точечная запись

Структуры в C++

В языке C++ структуры¹⁾ — это специальные типы данных, которые вводятся с помощью служебного слова **struct** (от англ. *structure* — структура). При этом нужно перечислить все элементы структуры — **поля** — и указать тип каждого из них. В отличие от языка Python добавить новые поля к структуре во время работы программы нельзя.

Структура, описывающая книгу в библиотеке, может быть объявлена так:

```
struct TBook {
    string author; // автор: строка
    string title; // название: строка
    int count; // количество: целое число
};
```

В результате такого объявления никаких структур в памяти не создаётся, это просто «заявление о намерениях»: мы описали новый тип данных, чтобы компилятор знал, что делать, если мы (вдруг) захотим его применить.

Теперь можно использовать тип **TBook** так же, как и простые типы данных (**int**, **float**, **double**, **char**, **bool**), для объявления переменных и массивов:

```
TBook b;
const int N = 100;
TBook books[N];
```

¹⁾ В некоторых языках, например в языке Паскаль, структуры называются записями (англ. *record*).

Здесь введена переменная `b` типа `TBook` и массив `books`, состоящий из 100 элементов того же типа.

При объявлении новой структуры желательно сразу задать начальные значения её полей, например так:

```
TBook b { "D. Knuth",
            "The Art of Computer Programming",
            4 } ;
```

Значения полей в фигурных скобках записываются в порядке их перечисления при описании структуры: в нашем примере "D. Knuth" — это значение поля `author`, "The Art of Computer Programming" — значение поля `title`, и 4 — значение поля `count`. Все поля, значения которых не заданы, обнуляются.

Можно явно указать названия полей, используя точечную запись:

```
TBook b { .author = "D. Knuth",
            .title = "The Art of Computer Programming"
            } ;
```

В этой структуре поле `count` будет равно нулю.

Обращение к полям структуры

Для обращения к отдельным полям структуры используют так называемую *точечную запись*, разделяя точкой имя структуры и имя поля. Например, `b.author` обозначает «поле `author` структуры `b`», а `books[5].count` — «поле `count` элемента массива `books[5]`».

С полями структуры можно обращаться так же, как и с обычными переменными соответствующего типа. Можно вводить значения полей с клавиатуры (или из файла):

```
getline( cin, b.author );
getline( cin, b.title );
cin >> b.count;
```

присваивать новые значения прямо в программе:

```
b.author = "Пушкин А.С.";
b.title = "Полтава";
b.count = 1;
```

использовать при обработке данных:

```
b.count--; // одну книгу взяли
if( b.count == 0 )
    cout << "Этих книг больше нет!" ;
```

и выводить на экран:

```
cout << b.author << " " << b.title << ". "
<< b.count << " шт." ;
```

Если нужно присвоить новые значения сразу всем полям структуры, можно записать их в фигурных скобках:

```
b = { "Тургенев И.С.", "Муму", 1 };
```

Порядок перечисления значений полей — тот же, что и при объявлении структуры, все не заданные поля обнуляются.

Работа с файлами

Данные, хранящиеся в структуре, можно записывать в текстовые файлы и читать из текстовых файлов так же, как и значения обычных переменных. Стандартных функций для выполнения операций со структурами в языке C++ нет, поэтому каждое поле придётся читать и записывать отдельно.

Значительно удобнее хранить структуры в файлах во внутреннем формате, т. е. так, как они представлены в памяти компьютера во время работы программы.

К сожалению метод, который описывается далее, нельзя использовать, если в состав структуры входят сложные типы данных, такие как **string**. Поэтому в дальнейших примерах будем использовать простую структуру, которая описывает свойства кирпича:

```
struct TBrick {
    int materialCode; // код материала
    double size[3]; // размеры кирпича
};
```

Здесь все данные относятся к простым типам, они расположены в памяти рядом, и транслятору легко определить, сколько места выделяется для каждого элемента.

Интересно, что общий размер структуры не обязательно равен сумме размеров полей. В этом легко убедиться с помощью оператора `sizeof`, который определяет размер переменной переданного ему типа:

```
cout << sizeof(int) << " "
    << sizeof(double) << " "
    << sizeof(TBrick);
```

Результат работы этого оператора вывода:

```
4 8 32
```

говорит о том, что целая переменная типа **int** занимает в памяти 4 байта, вещественная переменная с двойной точностью типа **double** занимает 8 байт, а вся структура **TBrick** занимает 32 байта.

Действительно, размер структуры (32 байта) *не равен* сумме размеров полей: $4 + 3 \cdot 8 = 28$ байт. Дело в том, что компилятор для ускорения обработки данных использует выравнивание, так что после поля `materialCode` остаётся свободный 4-байтный блок.

Объявим в программе одну такую структуру `b` и массив из 15 структур `bricks`:

```
TBrick b = { 1, {25, 12, 8.8} };
TBrick bricks[15] = {};
```

Здесь полям структуры `b` присвоены начальные значения: поле `materialCode` равно 1, а элементы массива `size` — 25, 12 и 8,8. Массив структур `bricks` обнуляется — все поля всех 15 элементов заполняются нулями.

Предположим, что нужно сохранить структуру `b` и массив `bricks` в двоичном файле.

В языке C++ с двоичными файлами можно работать через потоки, только при открытии такого потока нужно установить режим `ios::binary` (от англ. *binary* — двоичный). Тогда поток открывается для записи двоичных (а не текстовых!) данных:

```
#include <fstream>
...
ofstream Fout( "bricks.dat", ios::binary );
```

Теперь в него можно записать структуру `b`:

```
Fout.write( (char*)&b, sizeof(b) );
```

Методу `write` передаются два аргумента: адрес блока памяти, откуда взять данные для записи, и количество байт, которые нужно записать. Первый аргумент должен относиться к типу `char*` (указатель на символ), поэтому мы сразу преобразуем к этому типу адрес структуры `&b`.

Теперь запишем в файл 6 первых структур из массива `bricks`:

```
Fout.write( (char*)bricks, 6*sizeof(TBrick) );
```

Напомним, что вместо адреса начала массива (`&bricks[0]`) можно указать просто имя массива (`bricks`).

Прочитать из двоичного файла одну структуру и вывести её поля на экран можно следующим образом:

```
ifstream Fin( "bricks.dat", ios::binary );
Fin.read( (char*)&b, sizeof(b) );
cout << b.materialCode << " "
    << b.size[0] << "x"
    << b.size[1] << "x" << b.size[2];
```

Метод `read` принимает те же аргументы, что и `write`. Можно прочитать из файла и записать в массив сразу несколько структур (в данном случае три):

```
Fin.read( (char*)bricks, 3*sizeof(TBrick) );
```

Чтение останавливается, когда прочитано столько структур, сколько требовалось, или закончился файл.

Если нужно прочитать неизвестное количество структур (но не более 100), мы пытаемся читать все 100, а фактическое количество прочитанных структур определяем с помощью метода `gcount`, который возвращает число прочитанных байт. Если разделить это число на размер одной структуры, получаем количество прочитанных структур:

```
const int NMAX = 100;
TBrick bricks[NMAX];
ifstream Fin( "bricks.dat", ios::binary );
Fin.read( (char*)bricks, NMAX*sizeof(TBrick) );
int count = Fin.gcount() / sizeof(TBrick);
cout << "Прочитано " << count << " структур.";
```

Теперь разберёмся, что не так с символьными строками типа `string`. Дело в том, что переменная типа `string` содержит указатель на цепочку символов (её адрес). Сами символы, входящие в строку, находятся совершенно в другой области памяти, которая не относится к структуре. Поэтому если сохранить таким же образом структуру, содержащую строки типа `string`, вместо строк в файл будут записаны одни указатели (их размер — 4 байта, проверьте!). При загрузке структур из файла эти указатели прочитаются, но они уже будут недействительны — по этим адресам будут расположены совершенно другие данные, и, скорее всего, произойдёт ошибка обращения к памяти.

Сортировка

Структуры обычно сортируют по возрастанию или убыванию одного из полей, которое называют *ключевым полем* или *ключом*. Можно использовать и составные ключи, зависящие от значений нескольких полей.

Отсортируем массив `books` (типа `TBook`) по фамилиям авторов в алфавитном порядке. В данном случае ключом будет поле `author`. Пусть фамилия состоит из одного слова, а за ней через пробел записаны инициалы. Тогда сортировка методом пузырька выглядит так:

```
for( int i = 0; i < N-1; i++ )
    for( int j = N-2; j >= i; j-- )
        if( books[j].author > books[j+1].author ) {
            TBook temp = books[j];
            books[j] = books[j+1];
            books[j+1] = temp;
        }
```

Здесь `N` — размер массива структур, а `temp` — вспомогательная структура типа `TBook`.

Выводы

- Структура в C++ — это новый тип данных, который вводится с помощью служебного слова **struct**. При этом сразу определяются поля структуры и их типы, потом их нельзя изменять.
- К полям структуры обращаются с помощью точечной записи: <имя структуры>.<имя поля>.
- Для сохранения в файле структур, содержащих простые типы данных (**int**, **float**, **double**, **char**, **bool**), можно использовать потоки, работающие с двоичными файлами.
- Записать стандартными способами в поток структуры со сложными объектами (например, с символьными строками типа **string**) не удаётся, потому что некоторые из их данных размещаются в другой области памяти, не принадлежащей структуре.



Вопросы и задания

1. Вспомните, какие преимущества даёт использование структур.
2. Сравните структуры в языках Python и C++.
3. Как объявляется новый тип данных для хранения структур в C++?
Выделяется ли при этом память?
4. Что такое двоичный файл? Чем он отличается от текстового?
5. *Проект.* Постройте программу, которая работает с базой данных, хранящейся в виде файла (см. задание 9 к § 7).

§ 19

Стек, очередь, дек

Ключевые слова:

- стек
- очередь
- дек

Стек

Для работы со стеками в библиотеке STL введён специальный тип данных — **stack**, который может хранить элементы любого типа. Чтобы использовать этот контейнер в своей программе, необходимо подключить библиотеку **stack**:

```
#include <stack>
```

Стек для целых чисел (типа **int**) объявляется так:

```
stack <int> S;
```

У типа данных **stack** есть готовые методы, которые мы будем использовать:

- push** — добавление элемента на вершину стека;
- pop** — снятие элемента с вершины стека;
- top** — функция, возвращающая верхний элемент стека, не удаляя его;
- empty** — логическая функция, которая возвращает истинное значение, если стек пуст, и ложное, если в нём есть элементы.

В качестве примера использования стека рассмотрим следующую задачу. В файле записаны целые числа. Требуется переписать те же данные в другой файл в обратном порядке. Полная программа выглядит так:

```
#include <fstream>
#include <stack>
using namespace std;
int main ()
{
    ifstream Fin( "input.txt" );
    ofstream Fout( "output.txt" );
    stack <int> data;
    // чтение данных из файла
    int x;
    while( Fin >> x )
        data.push ( x );
    // запись результата в файл
    while( not data.empty() ) {
        Fout << data.top() << endl;
        data.pop();
    }
}
```

Эта программа получилась такой короткой только потому, что мы использовали готовые функции для работы со стеком из библиотеки STL.

Очередь

Библиотека STL содержит специальный тип для моделирования очереди. Он так и называется — **queue** (по-английски — очередь). Все нужные операции уже готовы:

- push** — добавить элемент в конец очереди;
- pop** — удалить первый элемент в очереди;
- front** — получить первый элемент из очереди, не удаляя его;
- empty** — логическая функция, которая возвращает истинное значение (**true**), если очередь пуста.

Для того чтобы использовать очереди, нужно подключить библиотеку `queue`:

```
#include <queue>
```

Рассмотрим задачу заливки области растрового рисунка, которую мы решали в § 8 на языке Python. Алгоритм заливки использует очередь, куда добавляются координаты точек, которые нужно перекрасить. Две координаты точки, *x* и *y*, мы объединим в структуру, которую назовём `TPoint` (по-английски *point* — точка):

```
struct TPoint {
    int x;
    int y;
};
```

Обе координаты — целые числа.

Для удобства напишем функцию, которая строит структуру типа `TPoint` по двум координатам и возвращает её как результат:

```
TPoint Point( int x, int y )
{
    TPoint pt = { .x = x, .y = y };
    return pt;
}
```

Очередь состоит из структур типа `TPoint`, поэтому объявить её нужно так:

```
queue <TPoint> Q;
```

В нашей программе очередь имеет имя `Q`.

Добавим в начало программы объявление матрицы, в которой будут храниться цвета пикселей:

```
const int XMAX = 5, YMAX = 5;
int A[XMAX][YMAX];
```

Как и в программе на языке Python (см. § 8), мы будем считать, что первый индекс элемента массива — это *x*-координата точки, а второй — её *y*-координата.

Определим точку, откуда начинается заливка, и запомним её цвет в переменной `color`:

```
int x0 = 1, y0 = 0; // начать заливку отсюда
int color = A[x0][y0];
```

Цвет заливки запишем в константу `NEW_COLOR`:

```
const int NEW_COLOR = 2;
```

Приведём основную часть программы заливки:

```

Q.push( Point(x0, y0) );           // (1)
while( not Q.empty() ) {
    TPoint pt = Q.front();        // (2)
    Q.pop();                      // (3)
    if( A[pt.x][pt.y] == color ) {
        A[pt.x][pt.y] = NEW_COLOR; // (4)
        if( pt.x > 0 )
            Q.push( Point(pt.x-1,pt.y) );
        if( pt.y > 0 )
            Q.push( Point(pt.x,pt.y-1) );
        if ( pt.x < XMAX-1 )
            Q.push( Point(pt.x+1,pt.y) );
        if ( pt.y < YMAX-1 )
            Q.push( Point(pt.x,pt.y+1) );
    }
}
}

```

В строке 1 в очередь (до этого момента — пустую!) добавляется первая точка, с которой начинается заливка.

Цикл выполняется, пока очередь не пуста, т. е. пока метод `empty` не вернёт значение `true`.

Метод `front` в строке 2 копирует первый элемент очереди в новую переменную `pt` типа `TPoint`. При этом эта точка не удаляется из очереди, и мы вынуждены сделать это вручную, вызвав метод `pop` в строке 3.

Если точка имеет тот же цвет, что и начальная точка, она перекрашивается в новый цвет (строка 4), и в очередь добавляются все её соседи, которые оказались в пределах области рисунка.

Хранение очереди в массиве

Если максимальная длина очереди известна, можно организовать очередь с помощью обычного массива. Такой приём работает в любых языках программирования, даже в тех, в которых нет динамических массивов.

Пусть количество элементов в очереди всегда меньше N . Создадим обычный массив из N элементов. В отдельных переменных будем хранить индексы первого элемента очереди («головы», англ. *head*) и последнего элемента («хвоста», англ. *tail*).

На рисунке 2.4, а показана очередь из 5 элементов.

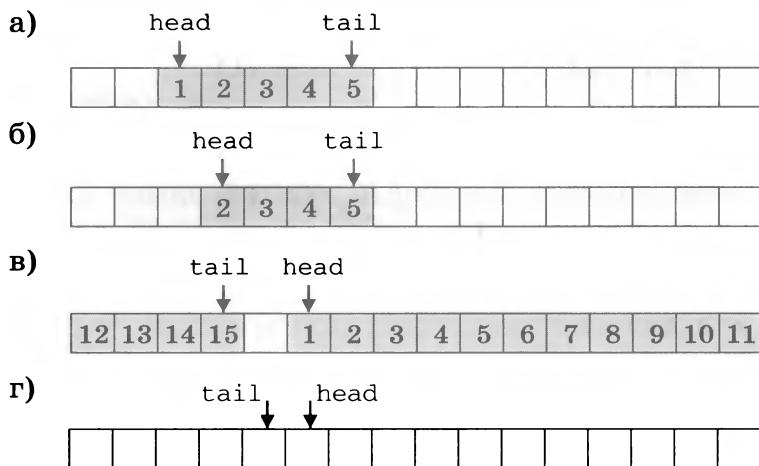


Рис. 2.4

Удаление элемента из очереди сводится просто к увеличению переменной `head` (рис. 2.4, б).

При добавлении элемента в конец очереди переменная `tail` увеличивается на 1. Если она перед этим указывала на последний элемент массива, то следующий элемент записывается в начало массива, а переменной `tail` присваивается значение 0. Таким образом, массив оказывается замкнутым «в кольцо».

На рисунке 2.4, в показана полностью заполненная очередь, а на рис. 2.4, г — пустая очередь. Один элемент массива всегда остаётся незанятым, иначе невозможно будет различить состояния «очередь пуста» и «очередь заполнена».

Массив можно также использовать для моделирования любых линейных структур, например стека (если его максимальный размер известен).

Дек

Дек — это двусторонняя очередь, он может работать и как стек, и как очередь. Основные операции с деком — это вставка и удаление элементов на обоих концах; они должны выполняться как можно быстрее. Лучше всего, если время выполнения этих операций не зависит от количества элементов в деке. Говорят, что в этом случае сложность операций — $O(1)$.

В состав библиотеки STL входит тип для моделирования дека, который называется `deque`. Для работы с такими контейнерами нужно подключить библиотеку `deque`:

```
#include <deque>
```

Дек для работы с целыми числами (`int`) объявляется так:

```
deque <int> D;
```

Здесь D — имя переменной-дека.

Методы дека позволяют работать с двумя его концами:

<code>push_front</code>	— добавить элемент в начало дека;
<code>push_back</code>	— добавить элемент в конец дека;
<code>pop_front</code>	— удалить первый элемент дека;
<code>pop_back</code>	— удалить последний элемент дека;
<code>front</code>	— получить первый элемент дека, не удаляя его;
<code>back</code>	— получить последний элемент дека, не удаляя его;
<code>empty</code>	— логическая функция, которая возвращает истинное значение (<code>true</code>), если дек пуст.

Дек в библиотеке STL позволяет обращаться к любому своему элементу по индексу. Например, вывести на экран все элементы дека D можно с помощью такого цикла:

```
for( auto i = 0; i < D.size(); i++ )
    cout << D[i] << " ";
```

Здесь мы работаем с деком так же, как и с массивом (вектором).

Стек и очередь в библиотеке STL строятся на основе дека (типа `deque`). Такие типы данных называются *адаптерами* (от англ. *adapt* — приспособливать). Они обеспечивают нужный набор операций для нового типа (стека или очереди), используя возможности готового типа `deque`.

Выводы

- Для моделирования стека, очереди и дека можно применить типы `stack`, `queue` и `deque` из библиотеки STL.
- Линейные структуры — стек, очередь, дек — можно моделировать с помощью массива, если известно максимально допустимое число элементов в контейнере.

Вопросы и задания



1. Напишите программу, которая с помощью стека вычисляет значение арифметического выражения, записанного в постфиксной форме. В выражении используются только целые числа и знаки арифметических операций. Предусмотрите сообщения об ошибках.
- *2. *Проект.* Доработайте программу из предыдущего задания так, чтобы в выражении можно было использовать функции `abs`, `sin`, `cos`, `sqrt`.
- *3. Доработайте программу из предыдущего задания так, чтобы в выражении можно было использовать имена переменных. Все необходимые значения переменных программа должна запрашивать у пользователя или вводить из файла.
- *4. *Проект.* Найдите в дополнительных источниках алгоритм для перевода арифметического выражения в постфиксную форму (обрат-

ную польскую запись) с помощью стека и напишите программу, которая выполняет такой перевод. В выражении используются только целые числа, знаки арифметических операций и круглые скобки.

*5. Доработайте программу из предыдущего задания так, чтобы в выражении можно было использовать функции `abs`, `sin`, `cos`, `sqrt` и имена переменных.

*6. *Проект.* В языке HTML, на котором создаются веб-страницы, большинство команд (*тэгов*) — парные. Они состоят из двух частей — открывающей и закрывающей, которые ограничивают область действия тэга. Например, при выводе на экран фрагмента

`Хаски` — это порода собак.

браузер выделит жирным шрифтом все символы между открывающим тэгом `` и закрывающим тэгом ``, который отличается от открывающего только символом `/`. Все тэги записываются в угловых скобках. Тэги должны закрываться в правильном порядке: если первым открыт тэг ``, его можно закрыть только тогда, когда закрыты все тэги, открытые после него. Например, эта строка ошибочна:

`Хаски — <i>это порода</i> собак.`

Напишите программу, которая обрабатывает веб-страницу (текстовый файл) и проверяет правильность закрытия тэгов.

*7. *Проект.* Доработайте программу из предыдущего задания так, чтобы она учитывала атрибуты (дополнительные свойства) тэгов. Они записываются только в открывающих тэгах, например, так

```
<p align="right">
<b>Хаски</b> — это порода собак.
</p>
```

8. В файле записаны адреса веб-страниц (URL), которые последовательно посещал пользователь, по одному адресу в строке. Известно, что всегда, когда это возможно, он переходил на нужную страницу с помощью кнопок «вперёд» и «назад» в браузере. Администратор хочет определить по этим данным, сколько раз пользователь:

- нажимал на кнопку «вперёд»;
- нажимал на кнопку «назад»;
- вводил адрес сайта вручную.

Напишите программу, которая обрабатывает файл и выводит на экран (или в файл) необходимые сведения.

*9. *Проект.* В файле записана последовательность действий пользователя одностroчного текстового редактора. Каждая команда записывается в отдельной строке. Разрешены следующие команды:

- | | |
|-------------|--|
| + "символы" | — добавить текст; |
| - число | — удалить указанное количество символов справа от курсора; |
| * число | — удалить указанное количество символов слева от курсора; |

- > число — перевести курсор на заданное количество позиций вправо;
- < число — перевести курсор на заданное количество позиций влево;
- UNDO — отменить последнюю операцию;
- REDO — выполнить снова последнюю отменённую операцию.

В начале работы курсор стоит в первой позиции строки, строка пустая. Напишите программу, которая выполняет последовательность команд и выводит на экран полученную строку. Если во время выполнения программы произошла ошибка, нужно вывести сообщение ERROR.

***10. Проект.** Напишите программу, которая моделирует работу стека целых чисел, управляемого текстовыми командами, аналогичными командам языка Форт. В начале работы стек пуст. Затем последовательно выполняются команды, записанные в файле. Для управления стеком используются следующие команды:

- <число> — добавить целое число на вершину стека;
- DROP — удалить число с вершины стека;
- SWAP — поменять местами два верхних числа в стеке (A B -> B A);
- DUP — дублировать верхнее число в стеке (A -> A A);
- OVER — скопировать второе число в стеке на вершину стека (A B-> A B A)
- + — сложить два верхних числа в стеке (A B -> A+B)
- — вычесть из второго числа в стеке первое (A B -> A-B);
- *
- перемножить два верхних числа в стеке (A B -> A*B);
- / — разделить нацело второе число в стеке на первое (A B -> A /B).

Программа должна вывести все числа, оказавшиеся в стеке после выполнения всех команд в файле. Слева должно быть дно стека, справа — вершина. Если стек пуст, нужно вывести слово EMPTY. Если во время выполнения команд произошла ошибка, нужно вывести слово ERROR.

11. Напишите программу, которая выполняет заливку одноцветной области заданным цветом. Матрица, содержащая цвета пикселей, вводится из файла. Затем с клавиатуры вводятся координаты начальной точки заливки и цвет заливки. На экран нужно вывести матрицу, которая получилась после заливки.

12. Проект. Робот ходит по клетчатому полю. Некоторые клетки представляют собой стены, непроходимые для Робота, они образуют лабиринт. Напишите программу, которая ищет оптимальный путь между двумя клетками в лабиринте. Формат хранения лабиринта и способ ввода исходных данных выберите самостоятельно.

13. Напишите набор функций, которые позволяют моделировать очередь длиной не более N элементов с помощью массива. Опишите очередь как структуру, содержащую сам массив, его размер, а также индексы первого и последнего элементов в очереди (head и tail на рис. 2.4).
14. Напишите набор функций, которые позволяют моделировать очередь с помощью массива, причём каждый раз при удалении элемента из начала очереди все остальные элементы сдвигаются к началу массива (первый элемент очереди всегда располагается в элемента массива с индексом 0). Сравните время, которое требуется для удаления первого элемента очереди для такого варианта и для модели очереди на рис. 2.4.
15. Напишите набор функций, которые позволяют моделировать стек размером не более N элементов с помощью массива.
16. В файле записаны результаты измерений лазерного излучения, выполненных прибором «Солярис» с интервалом в 1 секунду. Все данные – целые числа, не превышающие 10 000. Их так много, что загрузить их в память одновременно невозможно. Напишите программу, которая находит наибольшую (по модулю) разницу между результатами измерений, выполненных с интервалом в 15 секунд. Для хранения данных используйте очередь.
17. По условию предыдущей задачи напишите программу, которая находит наибольшую энергию излучения — сумму квадратов всех результатов измерений в течение 15 секунд.
18. По условию предыдущей задачи напишите программу, которая находит наименьшую сумму квадратов двух результатов измерений, выполненных с интервалом не менее, чем в 15 секунд.
19. Прибором МУХ-2БК управляет оператор, на пульте у которого 4 кнопки: «Загрузить», «Обработать», «Сохранить» и «Отмена». Команда «Загрузить» выполняется 5 минут, команда «Обработать» — 20 минут, команда «Сохранить» — 10 минут, а команда «Отмена» отменяет (удаляет из очереди заданий) последнюю команду. Каждая следующая команда начинает выполняться только тогда, когда закончено выполнение предыдущей команды. В файле записаны все команды оператора, по одной команде в строке. Слева от каждой команды записано время нажатия на соответствующую кнопку (в минутах и секундах), например:

10:10 Загрузить
10:11 Обработать
10:20 Сохранить

Напишите программу, которая выводит протокол работы прибора, т. е. определяет, какие команды и в какой последовательности выполнялись, когда началось и когда закончилось выполнение каждой из них.

§ 20

Деревья

Ключевые слова:

- дерево
- ключ
- двоичное дерево
- хранение в массиве
- обход в глубину
- модуль
- обход в ширину
- проект
- дерево поиска
- заголовочный файл

Деревья в C++

Как вы знаете из § 9, дерево — это структура данных, моделирующая иерархию (отношения подчинённости, многоуровневые связи).

Стандартного класса для работы с деревьями в языке C++ нет, поэтому нужно, как и в других языках, использовать связанные структуры.

Мы будем моделировать двоичное (бинарное) дерево, в котором у каждого узла может быть не более двух подчинённых узлов (сыновей): левого и правого. Данные в каждом узле будем хранить в виде символьной строки.

```
using PNode = struct TNode*;
struct TNode {
    string data; // данные узла
    PNode left; // ссылка на левое поддерево
    PNode right; // ссылка на правое поддерево
};
```

В этом фрагменте объявляются два новых типа данных. В первой строке вводится тип **PNode** — указатель на структуру **TNode**, описывающую узел дерева. Указатель — это адрес структуры, на это указывает звёздочка после названия типа.

Далее определяется сама структура **TNode**. Она содержит три поля: поле данных **data** и два указателя **left** и **right** — ссылки на левое и правое поддеревья.

Построим функцию, которая будет создавать новый узел в памяти и заполнять его поля заданными значениями:

```
PNode node( string data, PNode L = nullptr,
            PNode R = nullptr )
{
    PNode newNode = new TNode; // (1)
    newNode->data = data; // (2)
    newNode->left = L; // (3)
    newNode->right = R; // (4)
    return newNode; // (5)
}
```

Прежде всего, обратим внимание на заголовок функции. Она возвращает результат типа **PNode** — адрес нового узла. Параметры функции — это символьная строка **data** (содержимое узла, данные) и два указателя на узлы-сыновья: **L** (левый) и **R** (правый). По умолчанию (т. е. если мы их не задали явно) последние два параметра будут равны **nullptr** — это нулевой указатель¹⁾, который обозначает «пусто» или «не задано» (как **None** в языке Python).

В строке 1 в теле функции мы создаём в памяти новый узел²⁾ — структуру типа **TNode**. Память выделяется с помощью оператора **new**, адрес выделенного блока записывается в переменную-указатель **newNode**.

В строках 2–4 в поля новой структуры заносятся переданные данные. Поскольку **newNode** — это адрес новой структуры, для обращения к полям структуры *по адресу* используется не точка, а оператор «стрелка», состоящий из двух знаков, «минус» и «>», записанных рядом.

В строке 5 функция возвращает результат — адрес вновь созданного узла.

С помощью этой функции дерево, соответствующее арифметическому выражению $(1+4)*(9-5)$ (см. рис. 1.15), может быть построено так:

```
PNode T = node( "*",
                  node( "+", node("1"), node("4") ),
                  node( "-", node("9"), node("5") )
                );
```

Здесь **T** — это переменная типа **PNode**, т. е. указатель на корневой узел (его адрес в памяти). Через этот адрес мы можем обращаться к дереву и выполнять все необходимые операции с ним.

Итак, создавать деревья с помощью функции **node** мы уже научились. Теперь нужно понять, как освобождать эту память, когда дерево станет ненужным.

Написать просто

```
delete T;
```

нельзя, потому что так мы удалим только корневой узел дерева. Все остальные узлы останутся в памяти, только они станут «непривязанными» — к ним будет невозможно обратиться и их будет невозможно удалить. Выделенная им память будет помечена как занятая до окончания работы программы. Поэтому прежде, чем удалить узел, необходимо удалить всех его сыновей, а ещё раньше — сыновей этих сыновей и т. д. Получается такая рекурсивная процедура:

```
void deleteTree( PNode& Tree )
{
  if( not Tree ) return;      // (1)
  deleteTree( Tree->left );   // (2)
```

¹⁾ До принятия стандарта C++11 вместо **nullptr** использовалось число 0 или обозначение **NULL**.

²⁾ Заметка на будущее: поодумайте, где и когда нужно освобождать эту память.

```

deleteTree( Tree->right ); // (3)
delete Tree;                // (4)
Tree = nullptr;              // (5)
}

```

В строке 1 мы проверяем, не пустое ли дерево; если оно пустое, то удалять нечего, и происходит возврат из процедуры. В строках 2 и 3 с помощью рекурсивных вызовов этой же процедуры удаляются левое и правое поддеревья, и только после этого можно удалить из памяти корень дерева (строка 4).

В строке 5 в корень записывается нулевой указатель `nullptr` — признак того, что дерево пустое. Обратите внимание, что параметр `Tree` передаётся в процедуру по ссылке (со знаком &), поэтому в вызывающей программе после удаления дерева корень тоже станет равен `nullptr`.

Обходы дерева

Поскольку дерево — это структура данных, которую можно определить рекурсивно, для обхода дерева удобно использовать рекурсию. При этом программа получается очень короткой и понятной.

Например, обход дерева в порядке ЛПК («левое — правое — корень») для дерева, описывающего арифметическое выражение, с которым мы работали выше, даёт его постфиксную форму: 1 4 + 9 5 - *. Эту процедуру легко написать на языке C++:

```

void LPK( PNode Tree )
{
    if( not Tree ) return;
    LPK( Tree->left );
    LPK( Tree->right );
    cout << Tree->data << " ";
}

```

Процедура принимает один параметр — указатель на корневой узел дерева. Если он равен `nullptr` (пустое дерево), условие `not Tree` будет истинно, и произойдёт выход из процедуры (это условие окончания рекурсии). Далее мы вызываем ту же процедуру для левого поддерева, а затем — для правого поддерева, и в последней строке выводим содержимое корня.

Аналогичный рекурсивный обход в порядке «левое — корень — правое» — это обход дерева в глубину (англ. *DFS: depth-first search*):

```

void DFS( PNode Tree )
{
    if( not Tree ) return;
    DFS( Tree->left );
    cout << Tree->data << " ";
    DFS( Tree->right );
}

```

Чтобы выполнить обход в ширину (англ. *BFS: breadth-first search*), нам придётся использовать вспомогательную очередь. А рекурсии здесь не будет:

```
void BFS( PNode Tree )
{
    queue <PNode> Q;
    Q.push( Tree ); // (1)
    while( not Q.empty() ) { // (2)
        PNode temp = Q.front(); // (3)
        Q.pop(); // (4)
        cout << temp->data << " "; // (5)
        if( temp->left ) Q.push( temp->left );
        if( temp->right ) Q.push( temp->right );
    }
}
```

Сначала создаём в памяти локальную очередь *Q* из ссылок типа **PNode**, и в строке 1 записываем в неё адрес корня дерева, переданного как параметр.

Затем строим цикл с заголовком «пока очередь не пуста» (строка 2). В цикле создаём временную переменную *temp*, в которую копируем первый элемент-ссылку из очереди (строка 3). Сразу же удаляем этот первый элемент методом *pop* (строка 4).

В строке 5 выводим на экран данные выбранного узла *temp* — «посещаем» узел. Поскольку *temp* — это адрес узла, обращение к полю структуры через адрес выполняется с помощью оператора «*->*».

Наконец, добавляем в очередь левого и правого сыновей (если они есть) и снова берём первый элемент из очереди, если она не пустая.

Деревья поиска

Как вы знаете из § 9, двоичное дерево поиска обладает следующими свойствами:

- слева от каждого узла находятся узлы с меньшим ключом;
- справа от каждого узла находятся узлы, ключ которых больше или равен ключу данного узла.

Напишем программу, которая читает из файла целые числа и строит дерево поиска.

Структура, которая описывает узел дерева **TNode**, будет точно такой, как и в начале этого параграфа, но в поле данных *data* будем хранить не символьную строку, а целое число. Также будем считать, что тип **PNode** — это указатель на такой узел.

Сначала создадим файловый поток и откроем файл на чтение:

```
ifstream Fin( "input.txt" );
```

Строим пустое дерево — объявляем указатель и записываем в него значение `nullptr` — это означает, что никакого дерева ещё нет:

```
PNode T = nullptr;
```

Будем в цикле читать очередное целое число в переменную `x` и сразу добавлять её в дерево с помощью процедуры `addValue` (скоро мы её напишем):

```
int x;
while( Fin >> x )
    addValue( T, x );
```

Этот цикл будет работать до тех пор, пока очередное чтение целого числа не закончится неудачно.

Процедура `addValue` принимает два параметра: ссылку на корень дерева и значение, которое нужно добавить в это дерево:

```
void addValue( PNode& Tree, int data )
{
    ...
}
```

Заметим, что в некоторых случаях адрес корня может измениться. Например, когда мы добавляем значение в пустое дерево, нужно создать новый узел и сохранить его адрес. Поэтому мы передаём параметр `T` по ссылке, на это указывает знак `&` после типа `PNode`. Начнём как раз с такого случая — добавления в пустое дерево:

```
void addValue( PNode& Tree, int data )
{
    if( not Tree ) {
        Tree = node( data );
        return;
    }
    ...
}
```

Если дерево пустое (адрес корня равен `nullptr`), строится новый узел, и его адрес сохраняется в переменной `Tree`. Для создания узла используем функцию `node`, написанную выше, тип её первого параметра надо изменить на `int`. Параметр `Tree` передаётся по ссылке, поэтому процедура напрямую изменяет значение переменной в вызывающей программе.

Если дерево непустое, нужно сравнить переданное значение с полем `data` корня. Как следует из свойств дерева поиска, если значение `data` меньше, чем число в корне дерева, его нужно добавлять в левое поддерево, иначе — в правое. В итоге получается такая процедура:

```
void addValue( PNode& Tree, int data )
{
```

```

if( not Tree ) {
    Tree = node( data );
    return;
}
if( data < Tree->data )
   .addValue( Tree->left, data );
else addValue( Tree->right, data );
}

```

С помощью такого дерева легко отсортировать данные: достаточно обойти дерево в порядке «левое — корень — правое», т. е. выполнить обход в глубину.

В самом лучшем случае, с точки зрения скорости сортировки, построенное дерево поиска будет *полным*. В таком дереве все вершины, кроме листьев, имеют двух сыновей, а все листья расположены на одинаковой глубине (рис. 2.5, а).

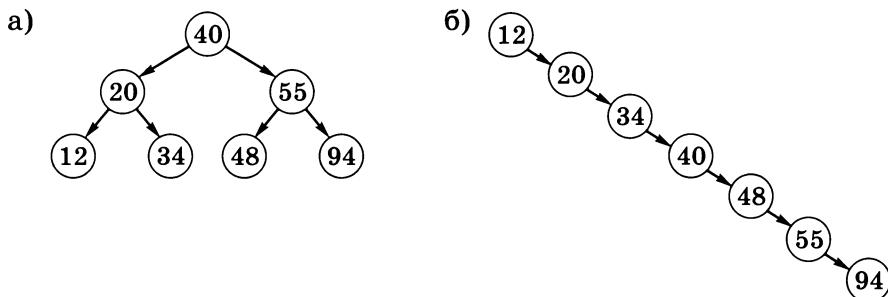


Рис. 2.5

Полное двоичное дерево имеет наименьшую возможную высоту для этого количества данных, поэтому поиск в нём выполняется за минимальное время.

В худшем случае дерево превращается в список (рис. 2.5, б), и поиск занимает столько же времени, сколько и линейный поиск в массиве.

Вычисление арифметических выражений

Используя структуру **TNode**, введённую в начале параграфа, напишем программу для вычисления арифметических выражений, записанных в символьной строке. Решение этой задачи включает две операции: построение дерева и вычисление выражения по дереву.

Процедура **makeTree** строит дерево и возвращает указатель на его корень — значение типа **PNode**:

```

PNode makeTree( string expr )
{
    PNode Tree;
    int pos = lastOp( expr );           // (1)
    if( pos == -1 )                  // (2)
        Tree = node( expr );          // (3)
}

```

```

else {
    Tree = new TNode;           // (4)
    Tree->data = expr[pos];     // (5)
    Tree->left = makeTree( expr.substr(0, pos) );
    Tree->right = makeTree( expr.substr(pos+1) );
}
return Tree;
}

```

Она полностью повторяет такую же функцию, написанную в § 9 на языке Python.

Сначала ищём последнюю выполняемую операцию, эту подзадачу в строке 1 решает функция lastOp (скоро мы её напишем). Дальнейшие действия зависят от того, нашли мы такую операцию или нет (строка 2). Если не нашли, это значит, что вся строка — это число, и нужно просто создать новый узел, который содержит полученную строку (строка 3).

Если операция найдена, создаём новый узел (строка 4), записываем в поле данных знак операции (строка 5) и в следующих двух строках вызываем функцию рекурсивно: строим левое и правое поддеревья из левой и правой частей строки.

Функция lastOp находит номер символа, в котором записана последняя операция:

```

int lastOp ( string expr )
{
    int minPrt = 50, pos = -1;
    for( int i = 0; i < expr.size(); i++ ) {
        int prt = priority( expr[i] );
        if( prt <= minPrt ) {
            minPrt = prt;
            pos = i;
        }
    }
    return pos;
}

```

Здесь вызывается функция priority, определяющая приоритет операции:

```

int priority( char op )
{
    if( op == '+' or op == '-' ) return 1;
    if( op == '*' or op == '/' ) return 2;
    return 100;
}

```

Итак, дерево мы строить научились, теперь нужно вычислить выражение по готовому дереву. Это делает функция calcTree:

```

int calcTree( PNode Tree )
{
    if ( not Tree->left )                                // (1)
        return stoi( Tree->data );                         // (2)
    int n1 = calcTree( Tree->left ),                      // (3)
        n2 = calcTree( Tree->right );                     // (4)
    return doOperation( Tree->data, n1, n2 );
}

```

Функция получает один параметр — указатель на корень дерева. Предполагается, что это дерево непустое.

В строке 1 проверяем левое поддерево. Если его нет, значит, этот узел содержит число, и чтобы получить результат, нам нужно просто перевести его из символьной формы в числовую (строка 2).

Если левое поддерево есть, то и правое тоже должно быть. Такой узел задаёт операцию. Сначала вычислим (рекурсивно!) левое и правое поддеревья (строки 3 и 4), и затем выполним заданную операцию между этими значениями с помощью функции `doOperation`:

```

int doOperation( string op, int n1, int n2 )
{
    if( op == "+" ) return n1 + n2;
    if( op == "-" ) return n1 - n2;
    if( op == "*" ) return n1 * n2;
    return n1 / n2;
}

```

Приведём пример основной программы, которая строит дерево по символьной строке и выводит на экран результат вычисления:

```

int main()
{
    string expr = "40-2*3-4*5";
    PNode T = makeTree( expr );
    cout << calcTree( T );
    deleteTree( T );
    cin.get();
}

```

Обратите внимание, что после вывода результата на экран дерево удаляется из памяти с помощью процедуры `deleteTree`. В данной программе это можно и не делать, потому что при завершении работы программы вся выделенная ей память будет освобождена автоматически. Но если такое вычисление арифметических выражений выполняется многократно, освобождение ненужной памяти обязательно. Если этого не сделать, программа будет захватывать все новые и новые блоки памяти, произойдёт «утечка памяти» (англ. *memory leak*), которая может привести к серьёзному сбою в работе программы.

Дерево в массиве

Двоичные деревья можно хранить в массиве. Вопрос о том, как сохранить структуру (взаимосвязь узлов), решается достаточно просто.

Пусть нумерация элементов массива T , где хранится дерево, начинается с 0. Тогда можно разместить сыновей элемента $T[i]$ в ячейках $T[2i+1]$ и $T[2i+2]$. На рис. 2.6, б показан порядок расположения элементов в массиве для дерева, соответствующего арифметическому выражению $40 - 2 * 3 - 4 * 5$ (рис. 2.6, а).

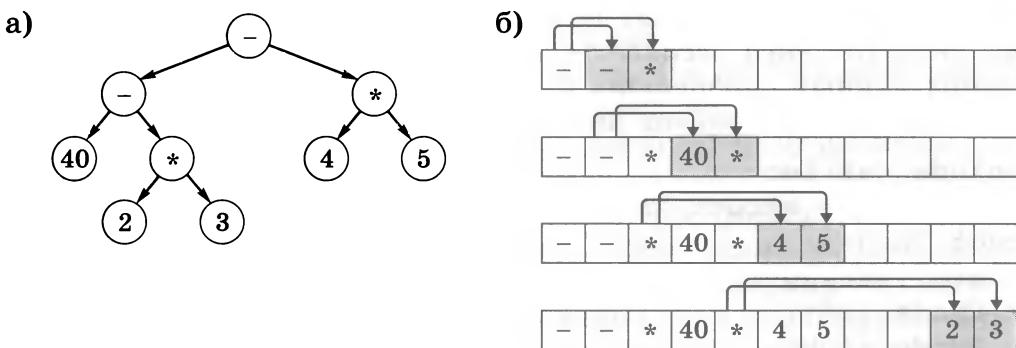


Рис. 2.6

Алгоритм вычисления выражения остаётся прежним, изменяется только метод хранения данных. Обратите внимание, что некоторые элементы массива остались пустыми, это значит, что их родитель — лист дерева.

Такой способ хорош для хранения деревьев, в которых все ветви имеют одинаковую высоту. Если ветви заканчиваются на разных уровнях, то в массиве будут пустые элементы, которые зря расходуют память.

Модульность

В C++, как и в других языках программирования, программу обычно разбивают на модули, каждый из которых можно разрабатывать и отлаживать независимо от других.

Модуль — это отдельный файл, в котором есть только функции, но нет основной программы (функции с именем `main`). Функции в модуле должны быть тесно связаны между собой, но как можно меньше связаны с другими модулями.

Например, мы можем построить модуль `bintree.cpp`, включив в него все подпрограммы для вычисления арифметических выражений с помощью двоичных деревьев. Тогда из файла, где находится основная программа, эти подпрограммы можно убрать.

Итак, мы фактически разбили программу на две части, поместив их в разные файлы. При этом возникают две проблемы:

- 1) как собрать эти две части в один исполняемый файл?
- 2) как сообщить основной программе, что функции makeTree, calcTree и др. действительно существуют, только расположены в другом файле?

Чтобы построить исполняемый файл из нескольких исходных файлов, применяют *проекты*. В среде программирования, с которой вы работаете, нужно создать проект и включить в него файл с основной программой, а также файл bintree.cpp.

Для решения второй проблемы в языке C++ используются так называемые *заголовочные* файлы, которые по традиции имеют расширение .h или .hpp (от англ. *header* — заголовок). В таких файлах объявляются типы данных, процедуры и функции. В нашем случае заголовочный файл bintree.h может выглядеть так:

```
#include <string>
using PNode = struct TNode*;
struct TNode {
    std::string data;
    PNode left;
    PNode right;
} TNode;
PNode makeTree( std::string s );
int calcTree( PNode Tree );
void deleteTree( PNode& Tree );
```

В первой строке мы подключаем стандартную библиотеку **string** для работы с символьными строками.

В заголовочных файлах нежелательно подключать пространства имён полностью командой **using**. Это может нарушить планы программиста, использующего ваши файлы. Например, пусть в пространстве имён myspace вашей библиотеки есть функция doBest. Команда

```
using namespace myspace;
```

добавляет имя doBest в область видимости программы. Программист не планирует использовать вашу функцию doBest, а пишет свою функцию с таким же именем. В результате получается конфликт имён. Поэтому в нашем заголовочном файле для типа **string** мы везде явно указываем пространство имён: пишем **std::string**.

Заголовочный файл содержит только объявление новых типов данных и *прототипы* функций — заголовки, заканчивающиеся точкой с запятой. Они составляют *интерфейс* — способ обмена данными между функциями и вызывающей программой. Другими словами, интерфейс — это всё, что нужно знать программе, чтобы правильно вызывать функции.

А команд, которые выполняются в теле этих функций (это называется *реализацией*), в заголовочном файле нет — они находятся в файле bintree.cpp. Кроме того, здесь нет прототипов функций node,

`priority`, `lastOp` и `doOperation`. Пользователю модуля совершенно не нужны эти функции, они вызываются только внутри самого модуля.

Если программа создаётся командой программистов, разработку (или улучшение) файла `bintree.cpp` можно поручить кому-то одному, а остальным достаточно знать только интерфейс — способ вызова функций.

Заголовочный файл подключается к основной программе и к модулю `bintree.cpp` с помощью команды `include`:

```
#include "bintree.h"
```

Мы уже много раз использовали этот приём, подключая заголовочные файлы стандартных библиотек языка C++. Однако на этот раз вы наверняка заметили, что имя файла заключено не в угловые скобки, как обычно, а в кавычки. Это означает, что транслятор будет искать файл в первую очередь не в своём стандартном каталоге с заголовочными файлами, а в текущем каталоге, где находятся все файлы проекта.

Структура модуля подобна айсбергу: всем видна только надводная часть (*интерфейс*, содержимое заголовочного файла), а значительно более весомая подводная часть (*реализация*, код процедур и функций) скрыта. Поэтому все могут использовать модуль, думая только о том, какие операции он выполняет, а не о том, как именно он это делает. Это один из приёмов, которые позволяют справляться со сложностью больших программ.

Разделение программы на модули облегчает понимание и совершенствование программы, потому что каждый модуль можно разрабатывать, изучать и оптимизировать независимо от других. При этом ускоряется компиляция больших программ, так как каждый модуль компилируется отдельно, причём только в том случае, если он был изменён.

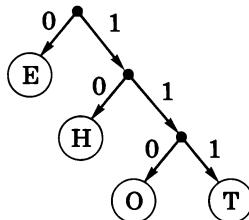
Выводы

- Дерево может храниться в памяти как набор связанных структур.
- Для обхода дерева в глубину можно использовать рекурсию или вспомогательный стек, а для обхода в ширину — очередь.
- Обход дерева поиска в глубину даёт отсортированную последовательность.
- Двоичное дерево можно хранить в массиве.
- Модули в программе на C++ — это отдельные файлы, не содержащие основную программу (функцию `main`). Функции в модуле должны быть тесно связаны между собой, но как можно меньше связаны с другими модулями.
- Для объединения нескольких исходных файлов в один исполняемый файл используют проекты.
- Заголовочный файл (обычно он имеет расширение `.h` или `.hpp`) содержит описание новых типов данных и объявления функций.



Вопросы и задания

1. Сколько элементов нужно выделить для массива, в котором хранится двоичное дерево высотой n ?
2. Приведите последовательность чисел, для которой высота двоичного дерева поиска, построенного по приведённому в параграфе алгоритму, будет:
 - а) наименьшей;
 - б) наибольшей.
3. Как нужно обойти двоичное дерево поиска, чтобы построить убывающую последовательность значений? Напишите процедуру, которая решает эту задачу.
4. Напишите логическую функцию, которая возвращает значение `true`, если переданное ей двоичное дерево – дерево поиска (обладает всеми нужными свойствами).
5. *Проект.* Напишите программу, которая вводит и вычисляет арифметическое выражение со скобками. Все операции с деревом вынесите в отдельный модуль.
- *6. *Проект.* Усовершенствуйте программу из предыдущего задания так, чтобы она могла вычислять выражения, использующие функции `abs`, `sin`, `cos`, `sqrt`.
7. *Проект.* Напишите модуль, включающий функции для различных способов обхода дерева, хранящегося в массиве.
8. *Проект.* Напишите программу, которая вычисляет арифметические выражения, записанные в файл (по одному выражению в строке) и выводит результаты в другой файл.
- *9. *Проект.* Напишите программу для вычисления арифметических выражений, которая хранит дерево в массиве. Сравните скорость работы этой программы с программой, которая использует динамическую структуру.
- *10. *Проект.* Для кодирования сообщений используются неравномерные коды, в которых выполняется *условие Фано*: ни одно кодовое слово не совпадает с началом другого кодового слова. Напишите программу, которая читает из файла кодовую таблицу, строит в памяти соответствующее дерево и декодирует введённое сообщение с помощью этого дерева. Например, набору кодов $E = 0$, $H = 10$, $O = 110$, $T = 111$ соответствует такое дерево:



Сообщение **111110111110100111110** декодируется (однозначно!) как **ТОТОНЕТО**. Предусмотрите сообщения об ошибках.

§ 21

Графы

Ключевые слова:

- жадный алгоритм
- оствовное дерево
- задача коммивояжёра
- гамильтонов цикл
- полный перебор
- рекурсия
- хвостовая рекурсия
- случайный поиск
- глобальные переменные
- структуры

Графы в языке C++

Как вы знаете из § 10, граф — это набор вершин и соединяющих их рёбер. Граф можно задавать матрицей смежности или списками смежности, в которых для каждой вершины перечисляются все смежные с ней вершины. Взвешенный граф, в котором с каждым ребром связано некоторое число — «вес», задаётся весовой матрицей.

Особенности работы с графиками в языке C++ связаны с тем, что вместо списков часто используются массивы, память под которые выделяется заранее.

Для примера рассмотрим решение задачи Прима–Краскала (см. § 10). Требуется построить оствовное дерево, которое соединяет все вершины графа и имеет минимальную сумму весов рёбер.

Алгоритм решения этой задачи, предложенный Краскалом, — это пример жадного алгоритма, который здесь, тем не менее, даёт наилучшее решение:

- 1) начальное дерево — пустое;
- 2) на каждом шаге к дереву добавляется ребро минимального веса, которое ещё не было выбрано и не приводит к появлению цикла (замкнутого маршрута).

Пусть информация о графике записана в виде весовой матрицы W размера N на N (индексы строк и столбцов начинаются с 0). Если связи между вершинами i и j нет, в элементе $W[i][j]$ этой матрицы будем хранить «условную бесконечность» — число, намного большее, чем длина любого ребра.

Для графа на рис. 2.7

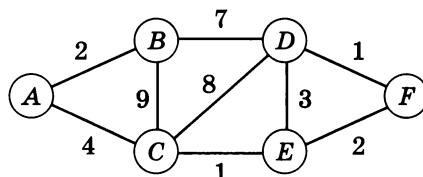


Рис. 2.7

весовая матрица может быть задана следующим образом:

```
const int N = 6;
const int INF = 30000; // "бесконечность"
int W[N][N] =
{ {0,      2,      4,  INF,  INF,  INF},
  {2,      0,      9,    7,  INF,  INF},
  {4,      9,      0,    8,    1,  INF},
  {INF,    7,      8,    0,    3,    1},
  {INF,    INF,    1,    3,    0,    2},
  {INF,    INF,    INF,   1,    2,    0} };
```

Будем строить оствовное дерево, используя раскраску вершин. Выделим массив для хранения цветов всех вершин и сначала «покрасим» все вершины разными цветами:

```
int col[N];
for( int i = 0; i < N; i++ )
  col[i] = i;
```

Выбранные рёбра будем хранить в массиве `ostov`. В программе на языке Python мы использовали кортежи, которые объединяют два значения — номера вершин, которые связывают ребро. Чтобы не вводить новый тип данных (структурную), в C++ можно использовать матрицу, у которой второй индекс принимает только два значения, 0 и 1:

```
int ostov[N-1][2];
```

Тогда для некоторого i значения $ostov[i][0]$ и $ostov[i][1]$ — это номера вершин, которые соединяет выбранное ребро с номером i . Для графа с N вершинами оствовное дерево состоит из $N-1$ рёбер, поэтому в массиве выделено место под $N-1$ пар чисел.

В остальном программа повторяет аналогичную программу на языке Python:

```
for( int k = 0; k < N-1; k++ ) {
  // поиск ребра с минимальным весом
  int iMin, jMin, minDist = 1000000000;
  for( int i = 0; i < N; i++ )
    for( int j = 0; j < N; j++ )
      if( col[i] != col[j] and
          W[i][j] < minDist ) {
        iMin = i;
        jMin = j;
        minDist = W[i][j];
      }
  // добавление ребра в список выбранных
  ostov[k][0] = iMin;
  ostov[k][1] = jMin;
```

```
// перекрашивание вершин
int c = col[jMin];
for( int i = 0; i < N; i++ )
    if( col[i] == c )
        col[i] = col[iMin];
}
```

Отметим, что начальное значение переменной `minDist` должно быть больше, чем значение константы `INF`, но должно помещаться в переменную типа `int` (если она занимает в памяти 4 байта, то её максимально допустимое значение равно `2 147 483 647`).

После окончания цикла остаётся вывести результат – рёбра из массива `ostov`, т. е. сохранённые номера двух вершин для каждого ребра:

```
for( int i = 0; i < N-1; i++ )
    cout << "(" << ostov[i][0] << ", "
           << ostov[i][1] << ")" << endl;
```

Задача коммивояжёра

Рассмотрим одну из самых сложных задач целочисленной оптимизации — *задачу коммивояжёра* (бродячего торговца). Ему нужно с наименьшими затратами объехать N городов и вернуться в тот город, откуда он начал путешествие.

Переведём задачу на язык графов: нужно найти цикл, который проходит через каждую вершину по одному разу. Такой цикл называется *гамильтоновым*.

К сожалению, эффективный алгоритм решения этой задачи неизвестен. Это значит, что в общем случае она решается только полным перебором вариантов.

Сначала формализуем задачу: запишем её в виде математических соотношений. Пронумеруем города — вершины графа — целыми числами, начиная с 0 (так удобнее, потому что именно с нуля начинается нумерация элементов массива в C++).

Порядок посещения городов можно закодировать как *перестановку* номеров городов, т. е. последовательность, в которую все числа от 0 до $N-1$ входят по одному разу. Нам нужно найти такую перестановку, при которой длина циклического маршрута наименьшая.

Так как маршрут циклический, можно считать, что он начинается в городе 0 и заканчивается там же. А следующие $N-1$ чисел мы можем переставлять, как угодно.

Напишем программу для полного перебора всех возможных перестановок, использующую рекурсию. Мы договорились, что рассматриваем маршруты, начинающиеся с 0. Для перебора всех возможных вариантов нам нужно последовательно ставить на следующее место все вершины, с 1-й по $(N-1)$ -ю, и для каждой такой пары начальных вершин снова перебирать все возможные варианты перестановок оставшихся чисел.

Пусть в ходе перебора мы выбрали и зафиксировали все элементы перестановки с номерами меньшими, чем `from` (рис. 2.8), и нужно перебрать все перестановки оставшихся элементов, обозначенных знаком вопроса.

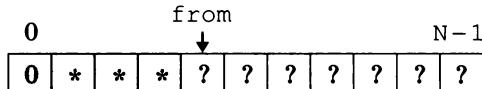


Рис. 2.8

Тогда задача перебора «хвоста», т. е. элементов с номерами от `from` до `N-1` включительно, сводится к тому, чтобы ставить на место с номером `from` каждый элемент этого «хвоста» и решать ту же самую задачу перебора для оставшихся элементов, с номерами от `from+1` до `N-1`. Таким образом, получился рекурсивный алгоритм.

Для того чтобы двигаться дальше, нужно определить, как хранить данные. Пока для простоты объявим данные, описывающие граф и оптимальный маршрут, глобальными¹⁾ (они будут доступны всем процедурам и функциям). Пусть график задан весовой матрицей размера $N \times N$:

```
const int N = 6;
double W[N][N] = {};
```

Далее везде мы будем считать, что после объявления матрица `W` заполняется значениями (с клавиатуры, из файла, случайными числами и т. п.).

Также введём глобальный массив `bestWay` (лучший маршрут на данный момент), и переменную `minLength` для хранения длины этого лучшего маршрута, которая сначала равна большому числу:

```
int bestWay[N] = {};
double minLength = 1e10; // Это 10000000000
```

Процедура `TSP` (от англ. *Travelling Salesman Problem* — задача коммивояжёра) будет выглядеть примерно так:

```
void TSP( int way[], int N, int from )
{
    ...
    for( int i = from; i < N; i++ ) {
        swap( way[from], way[i] );
        TSP( way, N, from+1 );
        swap( way[from], way[i] );
    }
}
```

Процедура принимает три параметра: массив `way`, в котором записана текущая перестановка; его размер `N` и количество уже зафиксированных элементов перестановки `from` (остальные еще нужно перебирать).

¹⁾ Потом мы увидим, как этого можно избежать.

В цикле ставим на место с номером `from` по очереди все элементы из «хвоста» перестановки:

```
swap( way[from], way[i] );
```

Здесь `swap` — это встроенная процедура библиотеки C++, которая меняет местами значения двух ячеек памяти (или элементов массива).

После этого рекурсивно вызывается та же процедура с увеличением последнего параметра на 1 («хвост» укоротился на один элемент). Как только она закончит работу, возвращаем выбранный ранее элемент «хвоста» обратно на своё место.

Единственное, что мы упустили, — условие окончания рекурсии (базовый случай). Когда значение `from` (третий параметр) станет равным $N-1$, это будет означать, что в «хвосте» остался только один элемент, т. е. перестановка полностью построена. Тут нам нужно будет определить общую длину полученного маршрута (он записан в массиве `way`). Если она меньше, чем `minLength`, запомним новый оптимальный маршрут в массиве `bestWay` и его длину в переменной `minLength`. Вместо многоточия в процедуру TSP нужно добавить такой фрагмент:

```
if( from == N-1 ) {
    double length = lengthOfWay( way, N );
    if( length < minLength ) {
        minLength = length;
        for( int i = 0; i < N; i++ )
            bestWay[i] = way[i];
    }
    return;
}
```

Функцию `lengthOfWay`, которая определяет длину циклического маршрута, мы сейчас напишем.

Длина цикла — это сумма длин образующих его рёбер. Номера вершин, через которые проходит цикл, записаны в массиве `way`. Рассмотрим пример цикла в графе с шестью вершинами (рис. 2.9).

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

Рис. 2.9

Для этого примера первое ребро, входящее в цикл, соединяет вершины 0 и 4, а в общем случае — вершины с номерами `way[0]` и `way[1]`. Учитывая, что длины рёбер хранятся в весовой матрице `W`, длину первого ребра можно найти как `W[way[0]][way[1]]`. Таким образом, функция `lengthOfWay` должна содержать цикл по всем рёбрам, входящим в маршрут:

```
double lengthOfWay( int way[], int N )
{
    double length = 0;
```

```

for( int i = 1; i < N; i++ )
    length += W[way[i-1]][way[i]];
length += W[way[N-1]][way[0]];      // (1)
return length;
}

```

К сумме, полученной в цикле, нужно добавить длину ребра из последней посещённой вершины в вершину с индексом 0 (строка 1).

В основной программе выделяем массив way и записываем в него любую перестановку чисел от 0 до N-1 (например, можно поставить эти числа по возрастанию):

```

int way[N] = {};
for( int i = 0; i < N; i++ )
    way[i] = i;

```

Теперь вызываем процедуру TSP, считая, что один (первый) элемент перестановки уже зафиксирован — это вершина 0:

```
TSP( way, N, 1 );
```

Процедура строит оптимальный маршрут (номера посещаемых вершин) и находит его длину. Эти данные хранятся в глобальных переменных bestWay и minLength. Выведем их на экран:

```

for( int i = 0; i < N; i++ )
    cout << bestWay[i] << " ";
cout << endl << minLength << endl;

```

Построенная программа находит точное решение задачи коммивояжёра полным перебором вариантов. Экспериментируя с ней, вы можете обнаружить, что при увеличении N она работает очень долго. Например, вряд ли вам удастся дождаться результата для $N = 20$. Это связано с тем, что алгоритм имеет факториальную сложность: количество перестановок $N-1$ чисел равно факториалу от $N-1$, т. е. $(N-1)! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (N-1)$. Для вычисления длины каждого цикла требуется ещё N шагов, таким образом, сложность алгоритма — $O(N!)$. Эта функция очень быстро возрастает при увеличении N . Например, для $N = 10$ она будет работать в 10 раз дольше, чем для $N = 9$.

Задача коммивояжёра: жадный алгоритм

Что же делать, если все-таки нужно решать задачу коммивояжёра для больших N ? Во-первых, разработаны методы ветвей и границ¹⁾, которые позволяют сокращать перебор. Они сразу отсекают варианты, заранее худшие, чем уже найденный.

Кроме того, часто хороший (хотя и не оптимальный) результат можно получить с помощью приближённых алгоритмов, которые работают значительно быстрее, чем полный перебор и даже методы ветвей и границ.

Мы рассмотрим два приближённых метода: жадный алгоритм и метод случайных перестановок.

¹⁾ Самый известный из них — алгоритм Литтла (*J.D.C. Little*).

Пусть выбрана начальная вершина с номером 0. Теперь ищем из оставшихся вершин ближайшую к ней и ставим её на следующее место. Этот алгоритм тоже можно записать в рекурсивной форме:

```
void TSP_Greedy( int way[], int N, int from )
{
    if ( from == N-1 ) return;
    double minDist = 1e10;
    int iNext = from;
    for( int i = from; i < N; i++ )
        if( W[way[from-1]][way[i]] < minDist ) {
            minDist = W[way[from-1]][way[i]];
            iNext = i;
        }
    swap( way[from], way[iNext] );
    TSP_Greedy( way, N, from+1 );
}
```

Если параметр `from` равен `N-1`, все элементы последовательности уже выбраны, и нужно просто выйти из процедуры. Иначе в «хвосте» массива `way` ищем вершину (из оставшихся), расстояние до которой от последней выбранной вершины (её номер `way[from-1]`) минимально. Переставляем номер этой вершины `way[iNext]` на место элемента `way[from]` и вызываем процедуру рекурсивно для оставшегося «хвоста».

Обратите внимание, что в процедуре всего один рекурсивный вызов, причём он стоит в самом конце. Это так называемая хвостовая рекурсия. От неё легко избавиться, заменив рекурсию на цикл.

Действительно, следующий рекурсивный вызов делает то же самое, но с другим значением `from`, большим на единицу. Поэтому можно записать вариант без рекурсии так:

```
void TSP_Greedy( int way[], int N )
{
    for( int from = 0; from < N; from++ ) {
        double minDist = 1e10;
        int iNext = from;
        for( int i = from; i < N; i++ )
            if( W[way[from-1]][way[i]] < minDist ) {
                minDist = W[way[from-1]][way[i]];
                iNext = i;
            }
        swap( way[from], way[iNext] );
    }
}
```

Третий параметр процедуры, `from`, здесь уже не нужен. Добавленный цикл по переменной `from` выделен в программе фоном.

Запуская программу при разных N , вы обнаружите, что даже при $N > 100$ программа мгновенно выдаёт ответ. В некоторых случаях он даже будет оптимальным, например когда все вершины расположены равномерно на одной окружности.

Задача коммивояжёра: случайные перестановки

Существует ещё один подход к приближённому решению сложных задач оптимизации — *метод случайного поиска*. Для задачи коммивояжёра можно использовать его следующим образом.

Пусть в массиве `way` записана перестановка, соответствующая лучшему на данный момент маршруту. Выберем случайным образом два элемента массива с номерами j_1 и j_2 ($1 \leq j_1, j_2 \leq N-1$) и переставим их в массиве `way`:

```
int j1 = randInt( 1, N-1 );
int j2 = randInt( 1, N-1 );
swap( way[j1], way[j2] );
```

Здесь использована функция `randInt`, которая возвращает случайное целое число на отрезке $[a; b]$:

```
int randInt ( int a, int b )
{
    return a + rand() % (b-a+1);
}
```

Теперь вычислим длину нового маршрута и, если она окажется меньше, чем предыдущая минимальная длина (в переменной `minLength`), запомним новый оптимальный маршрут.

Процедура случайного поиска принимает три параметра. Начальная перестановка записана в массиве `way`, второй и третий параметры — это количество вершин графа N и количество циклов случайного поиска `cycles`:

```
void TSP_Rand( int way[], int N, int cycles )
{
    for( int i = 0; i < cycles; i++ ) {
        int j1 = randInt( 1, N-1 );
        int j2 = randInt( 1, N-1 );
        swap( way[j1], way[j2] );

        double length = lengthOfWay( way, N );
        if( length < minLength ) {
            minLength = length;
            for( int k = 0; k < N; k++ )
                bestWay[k] = way[k];
        }
    }
}
```

Конечно, такой алгоритм не гарантирует, что решение будет самым лучшим из возможных, но вполне может найти очень хороший вариант за допустимое время даже для больших значений N . Чем больше случайных перестановок мы сделаем, тем лучше получится решение (и больше вероятность, что оно будет действительно оптимальным).

Заметим, что можно использовать и другие варианты случайных перестановок в массиве `way`, например:

- перестановку участка массива `way` с индексами от j_1 до j_2 в обратном порядке;
- вставку элемента `way[j_1]` сразу после элемента `way[j_2]`, и т. д.

Как избавиться от глобальных переменных?

Глобальные переменные в программе лучше не использовать. Для этого есть много причин:

- подпрограммы могут работать только с одним экземпляром глобальной переменной; например, написанная выше процедура `TSP_Rand` всегда использует глобальную весовую матрицу графа W , никакую другую она использовать не может (это называется *плохой масштабируемостью*);
- глобальные переменные доступны отовсюду, поэтому любая функция может свободно изменить их (в том числе и по ошибке!);
- глобальные переменные затрудняют поиск ошибок: обнаружить, когда именно была изменена глобальная переменная, очень сложно;
- функцию, которая работает с глобальными переменными, сложно использовать в другом проекте;
- ухудшается читаемость кода (непонятно, нужна ли в данном месте конкретная глобальная переменная);
- глобальные переменные увеличивают количество связей в программе, делая её менее понятной;
- глобальные переменные очень осложняют тестирование отдельного модуля.

Попробуем избавиться от глобальных переменных в решении задачи коммивояжёра методом случайных перестановок.

Объединим все данные, связанные с графом (весовую матрицу W , массивы `way` и `bestWay`, переменную `minLength`), в структуру `GraphData`. Для того чтобы не хранить отдельно значение N , вместо массивов будем использовать *векторы* (см. § 17), которые «знают свой размер»:

```
#include <vector>
struct GraphData {
    vector<vector<double>> W;
    vector<int> way, bestWay;
    double minLength;
};
```

Напомним, что для работы с векторами необходимо подключить библиотеку `vector`.

Здесь мы в первый раз встретились с типом данных

```
vector<vector<double>>
```

Это вектор, составленный из векторов типа `vector<double>`, т. е. двумерный массив (матрица) из значений `double`. В нашей программе в таком векторе будет храниться весовая матрица графа.

В основной программе нужно создать структуру типа `GraphData`:

```
const int N = 6;
GraphData graph;
```

и выделить в памяти место под матрицу размером $N \times N$ и два вектора размером N :

```
graph.W = vector<vector<double>> (N, vector<double>(N));
graph.way = vector<int>(N);
graph.bestWay = vector<int>(N);
```

После этого заполняем весовую матрицу `graph.W` и записываем в массив `graph.way` начальную перестановку.

Обратите внимание, что теперь константа `N` и структура `graph` — локальные данные основной программы. Таким образом, никаких глобальных данных в нашей программе нет. Все данные о графе передаются в процедуру через структуру `graph`:

```
TSP_Rand( graph, cycles );
```

Процедуру `TSP_Rand` тоже нужно поменять: у неё остаётся только два параметра — структура с данными о графе и количество циклов случайного поиска. Кроме того, все обращения к полям структуры — матрице `W`, массивам `way` и `bestWay` и переменной `minLength` — выполняются с помощью точечной записи (так мы «вытаскиваем» их из структуры):

```
void TSP_Rand( GraphData& graph, int cycles )
{
    int N = graph.W.size();
    graph.minLength = 1e10;
    for( int i = 0; i < cycles; i++ ) {
        int j1 = randInt(1, N-1);
        int j2 = randInt(1, N-1);
        swap( graph.way[j1], graph.way[j2] );
        double length = lengthOfWay( graph );
        if( length < graph.minLength ) {
            graph.minLength = length;
            for( int k = 0; k < N; k++ )
```

```
    graph.bestWay[k] = graph.way[k];
}
}
}
```

Поскольку поля структуры `graph` в процедуре изменяются, эта структура передаётся по ссылке. Для удобства мы вводим локальную переменную `N` — размер квадратной матрицы `graph.W`.

В функцию `lengthOfWay` теперь достаточно передать только структуру `graph`. Она содержит все необходимые данные, в том числе и массив `way`. Эти изменения сделайте самостоятельно.

Передача данных по ссылке

В последней версии программы мы передали структуру типа **GraphData** в процедуру по ссылке. На это указывает символ & («амперсанд») перед именем параметра в заголовке процедуры. Такой приём используют в двух случаях:

- 1) если нужно в подпрограмме изменить переменную (или структуру) вызывающей программы;
 - 2) если в подпрограмму передаётся большой объект.

Дело в том, что при «обычной» передаче данных (передаче по значению) создаётся копия аргумента, с которой работает подпрограмма. Копирование большого объекта требует множества лишних операций, копия занимает много места в памяти. В таких случаях лучше использовать передачу по ссылке, т. е. передать только адрес блока данных в памяти. Например, так обычно передаются объекты типа **vector**. Следующая процедура выводит на экран все элементы вектора из целых значений:

```
void printVector( const vector<int>& A )
{
    for( auto x: A )
        cout << x << " ";
    cout << endl;
}
```

Служебное слово **const** в заголовке процедуры говорит о том, что это *константная ссылка*. Таким образом, при чтении заголовка процедуры мы сразу получаем дополнительную информацию: параметр А в процедуре не изменяется.

Выводы

- Задача коммивояжёра состоит в том, чтобы с наименьшими затратами обехать N городов и вернуться в город, откуда он начал путешествие. Такой замкнутый маршрут в графе называется гамильтоновым циклом.

- Оптимальное решение задачи коммивояжёра можно найти только полным перебором вариантов. Асимптотическая сложность решения этой задачи — $O(N!)$.
- Приближённые методы (например, жадные алгоритмы) не гарантируют, что будет получено оптимальное решение, но позволяют найти достаточно хорошее решение за приемлемое время.
- В качестве приближённого метода можно использовать случайный поиск (например, случайные перестановки вершин графа в циклическом маршруте).



Вопросы и задания

1. Напишите программу, которая решает задачу коммивояжёра для N городов, расставленных равномерно на одной окружности. Для какого значения N вам удастся решить задачу на своём компьютере, затратив не более 5 минут?
2. *Проект.* Напишите программу, которая решает задачу коммивояжёра с помощью жадного алгоритма и не использует глобальные переменные. Сравните скорость её работы со скоростью полного перебора.
3. *Проект.* Напишите программу, которая решает задачу коммивояжёра с помощью случайных перестановок. Попробуйте разные варианты перестановок (см. текст параграфа), выберите лучший по результатам испытания на нескольких тестовых примерах.
4. Исследуйте работу программы, написанной в предыдущем задании: постройте зависимость длины найденного кратчайшего пути от количества циклов случайного поиска.

§ 22

Динамическое программирование

Ключевые слова:

- динамическое программирование
- перебор вариантов
- мемоизация
- оптимальная программа
- редактирование строк
- расстояние Левенштейна

Одномерные задачи

Как вы знаете, динамическое программирование удобно использовать для задач, в которых решение имеет вложенную структуру: решение задачи «размера» N можно представить через решения задач меньшего «размера».

Такие алгоритмы часто описываются с помощью рекурсии, но использование рекурсии «в лоб» приводит к тому, что мы много раз решаем уже решённые задачи, и на это расходуется очень много времени.

Идея динамического программирования состоит в том, чтобы хранить в памяти решения всех предыдущих задач меньшего «размера». Такой приём называется *мемоизацией* (дословно — запоминанием).

Рассмотрим задачу для исполнителя Калькулятор, с которым мы познакомились в § 11. У Калькулятора есть две команды, которым присвоены номера:

- 1) прибавь a ;
- 2) умножь на b .

Первая из них увеличивает число на экране на a , вторая — умножает его на b . Числа $a > 0$ и $b > 1$ — целые. Программа для Калькулятора — это последовательность команд. Нужно определить самую короткую программу, которая преобразует число M ($M > 0$) в число N ($N \geq M$).

Полный перебор всех возможных программ для больших значений N нереален, потому что количество вариантов огромно, и ни один современный компьютер не сможет выполнить такой объём работы за время жизни человека.

Обозначим через K_N длину самой короткой программы для получения числа N из начального числа M .

Попробуем использовать динамические программы, сохраняя в памяти для всех значений i от M до N длину самой короткой программы, которая преобразует число M в число i .

Так как $M > 0$, $a > 0$ и $b > 1$, при выполнении любой из команд число увеличивается (не может уменьшаться). Поэтому получить с помощью такого исполнителя из $M > 0$ число, меньшее, чем M , невозможно. Следовательно, можно принять, что $K_i = \infty$ для всех $i < M$ (длина «невозможной» программы равна бесконечности).

Понятно, что для получения числа M из начального значения M существует только одна программа — пустая, не содержащая ни одной команды. Это значит, что $K_M = 0$.

Теперь рассмотрим общий случай, чтобы построить рекуррентную формулу, связывающую K_N с предыдущими элементами последовательности K_1, K_2, \dots, K_{N-1} .

Если число N не делится на b , то на последнем шаге оно может быть получено только операцией сложения, поэтому

$$K_N = K_{N-a} + 1.$$

Если же N делится на b , то последней командой может быть как сложение, так и умножение. Нам нужно выбрать минимальное из двух значений: $K_{N-a} + 1$ и $K_{N/b} + 1$. В итоге получаем:

$$K_N = \begin{cases} K_{N-a} + 1, & \text{если } N \text{ не делится на } b, \\ \min(K_{N-a}, K_{N/b}) + 1, & \text{если } N \text{ делится на } b. \end{cases}$$

Это одномерная задача, её размер определяется одним параметром N . Поэтому для хранения промежуточных решений нам нужен линейный массив.

Чтобы понять работу алгоритма, решим вручную задачу такого типа для $a = 3$, $b = 2$, $M = 4$ и $N = 17$ (рис. 2.10).

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
K_N	*	*	*	0	*	*	1	1	*	2	2	*	3	2	*	2	3

Рис. 2.10

Звёздочкой обозначены значения, равные «бесконечности» (соответствующие числа нельзя получить из числа 4 с помощью заданных операций).

Теперь подумаем, как получить саму последовательность команд. Начнём с конечного числа. Если оно было получено последней командой 1, то должно выполняться равенство

$$K_{17} = K_{17 - 3} + 1.$$

Это равенство верно, поэтому последняя команда — 1, мы получили 17 из 14.

Далее, если число 14 получено командой 1, то должно выполняться равенство

$$K_{14} = K_{14 - 3} + 1.$$

Это равенство неверно, зато выполняется равенство

$$K_{14} = K_{14/2} + 1.$$

Поэтому предпоследняя команда — 2, мы получили 14 из числа 7. Таким же способом определяем, что 7 было получено из числа 4 командой 1. Оптимальная программа — 121.

Напишем функцию, которая принимает начальное и конечное числа — M и N — и возвращает оптимальную программу (символьную строку):

```
string minProgram( int M, int N )
{
    ...
}
```

В теле функции введём константы

```
int a = 3, b = 2;
```

и выделим в памяти массив для хранения промежуточных результатов:

```
int K[N+1];
```

Размер этого массива на единицу больше, чем N , так как нумерация элементов массива в C++ начинается с нуля, а нам будет нужен элемент с индексом N (это и будет длина самой короткой программы).

Заполняем массив: сначала все значения считаем равными «бесконечности» (большому числу), а в элемент $K[M]$ записываем 0:

```
const int INF = 1000000000;
for( int i = 0; i <= N; i++ )
    K[i] = INF;
K[M] = 0;
```

В основном цикле используется полученная выше рекуррентная формула:

```
for( int i = M+1; i <= N; i++ ) {
    if( i >= M+a )
        K[i] = K[i-a] + 1;
    if( i % b == 0 )
        K[i] = min( K[i], K[i/b]+1 );
}
```

Обратите внимание, что в первом условном операторе мы проверяем, что значение $i-a$ не меньше, чем M , иначе может быть выход за границы массива и аварийное завершение программы.

Остаётся найти саму оптимальную программу, этот алгоритм мы уже обсуждали выше:

```
string optProg = "";
int i = N;
while( i > M and K[i] < INF ) {
    if( K[i] == K[i-a]+1 ) {
        optProg = "1" + optProg;
        i -= a;
    }
    else {
        optProg = "2" + optProg;
        i /= b;
    }
}
```

В последней строке функции возвращаем результат — оптимальную программу:

```
return optProg;
```

Бот полный текст функции:

```
string minProgram( int M, int N )
{
```

```

int a = 3, b = 2;
int K[N+1];

const int INF = 1000000000;
for( int i = 0; i <= N; i++ )
    K[i] = INF;
K[M] = 0;

for( int i = M+1; i <= N; i++ ) {
    if( i >= M+a )
        K[i] = K[i-a] + 1;
    if( i % b == 0 )
        K[i] = min( K[i], K[i/b]+1 );
}

string optProg = "";
int i = N;
while( i > M and K[i] < INF ) {
    if( K[i] == K[i-a]+1 ) {
        optProg = "1" + optProg;
        i -= a;
    }
    else {
        optProg = "2" + optProg;
        i /= b;
    }
}
return optProg;
}

```

Редактирование строк

Существует наука *биоинформатика*, которая применяет математические методы в биологии. Биоинформатики занимаются, например, анализом молекул ДНК, в которых хранится вся наследственная информация человека. Эти молекулы состоят из четырёх типов блоков (нуклеотидов): аденина (A), тимина (T), цитозина (C) и гуанина (G).

В одной из задач, которую приходится решать биоинформатикам, нужно найти самый простой способ преобразования одной цепочки нуклеотидов в другую, например из цепочки ATG нужно получить CGAT.

Компьютерная модель этой задачи — преобразование символьных строк, при котором допустимы три операции:

- 1) замена одного символа цепочки другим;
- 2) удаление символа;
- 3) вставка символа.

Минимальное количество операций, необходимое для преобразования одной заданной строки в другую, называется *расстоянием редактирования* (англ. *edit distance*) или *расстоянием Левенштейна*.

Обозначим через $T[i][j]$ минимальное число операций для преобразования первых i символов исходной строки в первые j символов целевой строки (той, которую нужно получить). Это двумерная задача, потому что её размер зависит от двух параметров — длин обеих строк.

Пусть $sFrom$ и sTo — исходная и целевая строки, а $nFrom$ и nTo — их длины. Очевидно, что для решения полной задачи нужно найти $T[nFrom][nTo]$.

Для того чтобы использовать динамическое программирование, нужно выразить $T[i][j]$ через решения более простых задач, которые мы будем хранить в двумерном массиве.

Если $i = 0$ (исходная строка пустая), то для составления целевой строки длиной j нужно просто добавить j символов, т. е. выполнить j операций вставки. Поэтому $T[0][j] = j$.

Если $j = 0$ (целевая строка пустая), то для составления такой целевой строки из исходной строки длиной i нужно удалить i символов. Поэтому $T[i][0] = i$ (i операций удаления). Следовательно, нужно начать заполнение матрицы T так:

```
for( int i = 0; i <= nFrom; i++ )
    T[i][0] = i;
for( int j = 0; j <= nTo; j++ )
    T[0][j] = j;
```

Это базовые случаи, от которых мы будем «отталкиваться» при заполнении таблицы.

Теперь необходимо вывести формулу для заполнения остальных ячеек таблицы. Для примера рассмотрим случай, когда из цепочки ATG нужно получить CGAT. Здесь длины слов $nFrom = 3$ и $nTo = 4$, в таблице должны быть 4 строки и 5 столбцов (рис. 2.11).

		C	CG	CGA	CGAT		
		0	1	2	3	4	
		0	0	1	2	3	4
A	1	1	1	2	3	4	
AT	2	2	2	2	3	4	
ATG	3	3	3	3	3	4	

Вставка Замена Удаление

Рис. 2.11

Пусть мы знаем, как из цепочки А получить цепочку CG. Чтобы теперь из цепочки AT получить ту же CG, нужно в начале программы выполнить одно удаление (удалить символ T). Поэтому переход в таблице на одну ячейку вниз соответствует операции удаления.

Если мы знаем, как из AT получить C, то программа получения CG из той же исходной строки AT будет содержать дополнительную

команду вставки (в конец строки). Следовательно, переход в таблице на одну ячейку вправо — это операция вставки.

Если известно, как из А получить С, то получить СГ из АТ можно, добавив в конец программы команду замены символа. Если последние символы исходной и целевой строк одинаковы, никакой операции вообще не требуется («стоимость» не увеличивается).

При переходе к следующей ячейке мы добавляем к программе одну из трёх команд (см. рис. 2.11). Подсчитаем длины получившихся программ в ячейке $T[i][j]$ для всех возможных операций:

удаление: $T[i-1][j] + 1$

вставка: $T[i][j-1] + 1$

замена: $T[i-1][j-1]$, если буквы $sFrom[i-1]$ и $sTo[j-1]$ одинаковые (замена не нужна), или $T[i-1][j-1] + 1$, если эти буквы разные.

Можно считать, что «стоимость» замены равна k , где $k = 0$ или $k = 1$, в зависимости от того, совпадают ли последние буквы подстрок. Тогда длина программы при использовании замены на последнем шаге равна $T[i-1][j-1] + k$.

Нам нужно построить программу минимальной длины, поэтому будем всегда выбирать минимальное из всех возможных значений:

$$T[i][j] = \min(T[i-1][j] + 1, T[i][j-1] + 1, T[i-1][j-1] + k).$$

По этой формуле заполним все ячейки таблицы, сверху вниз и слева направо, начиная с $T[1][1]$:

```
for( int i = 1; i <= nFrom; i++ )
    for( int j = 1; j <= nTo; j++ ) {
        int k = 1;
        if( sFrom[i-1] == sTo[j-1] ) k = 0;
        T[i][j] = min( T[i-1][j-1]+k,
                       min( T[i-1][j]+1, T[i][j-1]+1 ) );
    }
```

Чтобы решить задачу, нужно заполнить всю таблицу. Расстояние Левенштейна между строками — это угловой элемент матрицы $T[nFrom][nTo]$, т. е. «стоимость» преобразования полной исходной строки в целевую. Приведём полный текст функции:

```
int minEditDistance( string sFrom, string sTo )
{
    int nFrom = sFrom.size(),
        nTo = sTo.size();
    int T[nFrom+1][nTo+1];

    for( int i = 0; i <= nFrom; i++ )
        T[i][0] = i;
    for( int j = 0; j <= nTo; j++ )
        T[0][j] = j;
```

```

for( int i = 1; i <= nFrom; i++ )
    for( int j = 1; j <= nTo; j++ ) {
        int k = 1;
        if( sFrom[i-1] == sTo[j-1] ) k = 0;
        T[i][j] = min(T[i-1][j-1]+k,
                        min(T[i-1][j]+1, T[i][j-1]+1));
    }

return T[nFrom][nTo];
}

```

Эта функция возвращает только длину самой короткой программы, преобразующей одну строку в другую. Если нужна сама программа, т. е. операции, которые применяются на каждом шаге, можно использовать ещё одну вспомогательную матрицу. В ней запоминаются коды команд, применявшихся при заполнении каждой ячейки матрицы Т.

Оптимальная стратегия

Динамическое программирование оказывается полезным при определении оптимальной стратегии в играх.

Рассмотрим простую игру. На столе в ряд лежат N монет разного достоинства, их стоимости записаны в массив coins. Игроки ходят по очереди, за один ход игрок может взять первую или последнюю монету в ряду (средние монеты брать не разрешается). Выигрыш игрока — это общая стоимость взятых им монет. Нужно определить величину гарантированного выигрыша того игрока, который ходит первым.

Применим метод динамического программирования. Через $T[i][j]$ обозначим гарантированный выигрыш первого игрока при условии, что выложены монеты с i -й по j -ю включительно.

Очевидно, что при $i > j$ ни одной монеты на столе нет, и выигрыш равен нулю. Если лежит одна монета, то игрок берёт её и это и есть его выигрыш:

$$T[i][i] = \text{coins}[i] \text{ при всех } i.$$

Если монет две (с i -й по $(i+1)$ -ю), игрок берёт большую из двух монет:

$$T[i][i+1] = \max(\text{coins}[i], \text{coins}[i+1]) \text{ при всех } i.$$

Рассмотрим пример. Дано пять монет такого достоинства:

```

const int N = 5;
int coins[N] = {1, 6, 2, 5, 7}

```

При этом матрица Т имеет структуру, показанную на рис. 2.12.

		j				
		0	1	2	3	4
		0	1	6	*	*
		1	0	6	*	*
		2	0	0	2	5
		3	0	0	0	5
		4	0	0	0	7

Рис. 2.12

Звёздочкой отмечены элементы, которые пока не определены.

Заполнять таблицу далее будем по диагоналям (см. рис. 2.12). Первая слева незаполненная диагональ — это элементы $T[j][j+2]$, вторая — $T[j][j+3]$ и т. д.

Рассмотрим элемент матрицы $T[j][j+i]$. Это значит, что на игровое поле выставлены все монеты от j -й до $(j+i)$ -й включительно.

У первого игрока есть два варианта хода: взять монету с номером j или монету с номером $j+i$. В первом случае его соперник получает цепочку монет от $(j+1)$ -й до $(j+i)$ -й, и он своим ходом может оставить одну из двух позиций: $T[j+2][j+i]$ (если возьмёт монету слева) или $T[j+1][j+i-1]$ (если возьмёт монету справа).

Мы считаем, что противник не ошибается, поэтому должны выбрать худшую (для нас) ситуацию из возможных — ту, у которой оценка нашего выигрыша минимальная. Таким образом, оценка варианта с взятием монеты с номером j вычисляется как

$$cost_1 = coins[j] + \min(T[j+2][j+i], T[j+1][j+i-1]).$$

Рассуждая таким же образом, найдём оценку варианта со взятием последней монеты (с номером $j+i$):

$$cost_2 = coins[j+i] + \min(T[j+1][j+i-1], T[j][j+i-2]).$$

Здесь мы тоже при выборе хода берём худший случай для первого игрока (и лучший для второго!).

Итак, у первого игрока есть два возможных хода, при первом его гарантированный выигрыш составит $cost_1$, при втором — $cost_2$. Какой выбрать? Очевидно, что первый игрок должен выбрать тот ход, который приносит ему наибольший выигрыш:

$$T[j][j+i] = \max(cost_1, cost_2)$$

Такая стратегия называется стратегией *максимины* — мы выбираем ход, дающий нам максимальный выигрыш, но при оценке каждого хода рассчитываем на худший для себя случай (соперник уступает нам минимально возможную сумму).

Теперь можно написать функцию, которая возвращает гарантированный (минимально возможный) выигрыш первого игрока для заданного набора монет:

```
int winGain( int N, int coins[] )  
{  
    int T[N][N];  
    // Заполняем матрицу нулями  
    for( int i = 0; i < N; i++ )  
        for( int j = 0; j < N; j++ )  
            T[i][j] = 0;  
    // Если осталась одна монета, берём её  
    for( int i = 0; i < N; i++ )  
        T[i][i] = coins[i];  
    // Если остались две монеты, берём большую  
    for( int i = 0; i < N-1; i++ )  
        T[i][i+1] = max( coins[i], coins[i+1] );  
  
    for( int i = 2; i < N; i++ )  
        for( int j = 0; j+i < N; j++ ) {  
            int cost1 = coins[j] +  
                min( T[j+2][j+i], T[j+1][j+i-1] );  
            int cost2 = coins[j+i] +  
                min( T[j+1][j+i-1], T[j][j+i-2] );  
            T[j][j+i] = max(cost1, cost2);  
        }  
    return T[0][N-1];  
}
```

Функция `winGain` не использует никаких глобальных переменных, её работа зависит только от переданных ей параметров — количества монет `N` и массива стоимостей монет `coins`. Поэтому такую функцию легко перенести в другие проекты, например выделив в отдельный модуль.

Выводы

- В одномерных задачах динамического программирования решение зависит от одного параметра, в двумерных — от двух и т. д.
- Для хранения промежуточных решений в одномерных задачах используется линейный массив, а в двумерных — матрица.
- Минимальное количество операций, необходимое для преобразования одной заданной строки в другую, называется расстоянием редактирования или расстоянием Левенштейна.
- При решении игровых задач применяется принцип максимина: при оценке возможных ходов используются худшие для игрока варианты (с минимальным выигрышем), и из всех ходов выбирается ход с максимальной оценкой.



Вопросы и задания

1. У исполнителя Калькулятор три команды, которым присвоены номера:

- 1) прибавь a ;
- 2) умножь на b ;
- 3) умножь на c .

Здесь a , b и c — целые числа, такие что $a > 0$, $c > b > 1$. Напишите программу, которая находит самую короткую программу для этого исполнителя, преобразующую число M ($M > 0$) в число N ($N \geq M$).

2. Измените программу из предыдущего задания так, чтобы она учтывала разное время выполнения операций: умножение выполняется в 2 раза медленнее, чем сложение. Сравните результаты работы изменённой и исходной программы. Для каких исходных данных решение не изменится?
3. Лягушка прыгает по кочкам, расположенным на одной прямой на равных расстояниях друг от друга. Кочки имеют порядковые номера от 1 до N . Вначале лягушка сидит на кочке с номером 1. На каждой кочке есть трамплин: с трамплина на i -й кочке лягушка может прыгнуть вперёд на расстояние от 1 до K_i кочек, считая от текущей. Напишите программу, которая вычисляет наименьшее число прыжков, за которое лягушка может добраться до кочки с номером N .
4. Бревно длиной L метров нужно распилить на части так, чтобы эти части можно было продать как можно дороже. Длина каждой части в метрах должна быть натуральным числом, цены за бревна разной длины хранятся в массиве `price`: $price[i]$ — это цена бревна длиной i метров ($i = 0, \dots, L$). Напишите программу, которая определяет, на какие части нужно разрезать бревно и какую сумму можно выручить от продажи бревна по частям.
5. Напишите программу для определения расстояния Левенштейна между двумя строками, которые вводятся с клавиатуры.
6. Дополните программу из предыдущего задания так, чтобы она выводила оптимальную последовательность операций преобразования, а не только их количество.
- *7. Измените программу из предыдущего задания так, чтобы она учтывала разную «стоимость» операций: операция замены символа «стоит» 1 балл, а операции удаления и вставки — 2 балла. Сравните результаты работы этого и предыдущего вариантов программ. В каких случаях результаты работы двух программ будут одинаковыми?
- *8. Напишите программу, которая для двух заданных строк определяет их самую длинную общую подстроку. Если таких строк несколько, найдите все общие подстроки максимальной длины.

- **9. *Проект.* Напишите программу, которая определяет, соответствует ли имя файла заданной маске. В масках можно использовать символы * (любое количество любых символов) и ? (один любой символ). Например, имя файла runo.doc соответствует маскам *no.d?? и ?u*.*c, но маска ??na?.r* для него не подходит. Обозначьте через $T[i][j]$ соответствие первых i символов имени файла и первых j символов маски.
- *10. Фермер хочет построить дом прямоугольной формы, стены которого смотрят на север, восток, юг и запад. У него есть карта участка, разбитая на квадраты размером 1×1 м. Все квадраты, непригодные для строительства, перечёркнуты. Выберите способ хранения данных и напишите программу, которая определяет наибольшую площадь дома, который может построить фермер.
- *11. Измените решение задачи определения оптимальной стратегии из параграфа так, чтобы функция определяла гарантированный выигрыш второго игрока.
- **12. Дополните решение задачи определения оптимальной стратегии из параграфа так, чтобы можно было увидеть все ходы в игре, где оба игрока следуют оптимальным для себя стратегиям.

ОГЛАВЛЕНИЕ

Предисловие	3
Глава 1. Программирование на языке Python	5
§ 1. Простые алгоритмы сортировки	5
Что такое сортировка?	5
Метод пузырька (сортировка обменами)	5
Метод выбора	8
Выводы	8
§ 2. Быстрые алгоритмы сортировки	11
Сортировка слиянием	11
Быстрая сортировка	13
Сортировка в языке Python	15
Выводы	16
§ 3. Двоичный поиск	17
Поиск в массиве	17
Что такое двоичный поиск?	18
Двоичный поиск в массиве данных	19
Какой алгоритм поиска лучше?	20
Двоичный поиск по ответу	20
Выводы	22
§ 4. Обработка файлов	23
Какие бывают файлы?	23
Принцип сэндвича	24
Чтение данных	25
Запись данных	26
Вывод файла на экран	26
Суммирование данных из файла	27
Обработка массивов	27
Обработка строк	28
Выводы	30
§ 5. Целочисленные алгоритмы	32
Решето Эратосфена	32
Квадратный корень	35
Выводы	36
§ 6. Словари	37
Что такое словарь?	37
Алфавитно-частотный словарь	39
Перебор элементов словаря	40
Выводы	41
§ 7. Структуры	42
Зачем нужны структуры?	42
Классы	43
Создание структур	44
Что можно делать с полями структуры?	44
Работа с файлами	45
Сортировка	47
Выводы	48
§ 8. Стек, очередь, дек	50
Что такое стек?	50

Использование списка	51
Вычисление арифметических выражений	52
Скобочные выражения	54
Как вызываются подпрограммы?	56
Очередь	58
Дек	60
Выводы	61
§ 9. Деревья	62
Что такое дерево?	62
Деревья поиска	64
Обход дерева	65
Использование связанных структур	66
Вычисление арифметических выражений	68
Модульность	72
Выводы	73
§ 10. Графы	74
Что такое граф?	74
Как описать граф?	75
«Жадные» алгоритмы	77
Минимальное остовное дерево	78
Алгоритм Дейкстры	82
Алгоритм Флойда–Уоршелла	86
Использование списков смежности	87
Некоторые задачи	90
Выводы	91
§ 11. Динамическое программирование	92
Числа Фибоначчи	92
Что такое динамическое программирование?	94
Кошки и собаки	95
Количество программ для исполнителя	97
Двумерная задача	98
Поиск оптимального решения	100
Выводы	103
§ 12. Игровые модели	105
Выигрышные и проигрышные позиции	105
Кто выиграет?	106
Динамическое программирование	108
Игра с фишками	108
Выводы	110
Глава 2. Программирование на языке C++	112
§ 13. Простые алгоритмы сортировки	112
Метод пузырька (сортировка простыми обменами)	112
Сортировка вставками	113
Массивы в подпрограммах	114
Выводы	115
§ 14. Быстрые алгоритмы сортировки и поиска	117
Сортировка слиянием	117
Быстрая сортировка	120
Стандартная сортировка в языке C++	124
Двоичный поиск	125
Выводы	126
§ 15. Обработка файлов	127
Принцип сэндвича	127

Файловые потоки.....	128
Неизвестное количество данных	130
Обработка массивов.....	130
Чтение файлов по словам.....	131
Построчная обработка файлов	132
Как передать имя файла программе?.....	134
Выводы	135
§ 16. Целочисленные алгоритмы	139
Решето Эратосфена	139
«Длинные» числа.....	140
Выводы	144
§ 17. Динамические массивы и словари.....	145
Зачем это нужно?	145
Динамические массивы.....	146
Тип <code>vector</code> из библиотеки STL.....	147
Итераторы.....	149
Зачем нужны итераторы.....	151
Словари	153
Перебор элементов словаря	154
Выводы	156
§ 18. Структуры.....	157
Структуры в C++.....	157
Обращение к полям структуры	158
Работа с файлами	159
Сортировка	161
Выводы	162
§ 19. Стек, очередь, дек	162
Стек	162
Очередь	163
Хранение очереди в массиве	165
Дек.....	166
Выводы	167
§ 20. Деревья	171
Деревья в C++	171
Обходы дерева	173
Деревья поиска	174
Вычисление арифметических выражений	176
Дерево в массиве	179
Модульность	179
Выводы	181
§ 21. Графы.....	183
Графы в языке C++	183
Задача коммивояжёра.....	185
Задача коммивояжёра: жадный алгоритм	188
Задача коммивояжёра: случайные перестановки	190
Как избавиться от глобальных переменных?	191
Передача данных по ссылке	193
Выводы	193
§ 22. Динамическое программирование	194
Одномерные задачи	194
Редактирование строк	198
Оптимальная стратегия.....	201
Выводы	203