Глава 6
АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

§ 35
Целочисленные алгоритмы

Ключевые слова:

- решето Эратосфена
- «длинные» числа
- целочисленный квадратный корень

Во многих задачах все исходные данные и необходимые результаты — 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 значительно быстрее, чем с вещественными;
- целые числа всегда точно представляются в памяти компьютера, и вычисления с ними также выполняются без ошибок (если, конечно, не происходит переполнения разрядной сетки).

Решето Эратосфена

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

Пусть задано некоторое натуральное число N и требуется найти все простые числа в диапазоне от 2 до N. Самое простое (но неэффективное) решение этой задачи состоит в том, что в цикле перебираются все 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 очень медленно, он имеет асимптотическую сложность $O(N\sqrt{N})$. Греческий мате-

матик Эратосфен Киренский (275–194 гг. до н. э.) предложил другой алгоритм, который работает намного быстрее (сложность $O(N \log\log N)$):

1) выписать все числа от 2 до N;
2) начать с $k=2$;
3) вычеркнуть все числа, кратные k ($2k, 3k, 4k$ 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 не вычеркнутое число — 2, поэтому вычёркиваем все чётные числа:

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. Таким образом, получены простые числа 2, 3, 5, 7, 11 и 13.

Классический алгоритм можно улучшить, уменьшив количество операций. Заметьте, что при вычёркивании чисел, кратных 3, 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 вычеркнутыми.

Предположим, что мы хотим вычеркнуть 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 $k^2 > N$ вычеркивать 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 $k^2 \le N$.

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.

Как и в 10 классе, для программирования алгоритмов мы будем использовать язык Python.

Поскольку индексы элементов списков в Python всегда начинаются с нуля, для того чтобы работать с нужным диапазоном индексов, необходимо выделить массив из `N + 1` элементов. 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, мы заменили условие $k \le \sqrt{N}$ на равносильное условие $k^2 \le N$, в котором используются только целые числа.

После завершения этого цикла невычеркнутыми остались только простые числа, для них соответствующий элемент массива содержит истинное значение. Эти числа нужно вывести на экран:

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

Теперь попробуем переписать это решение в стиле языка Python. Поскольку при вызове функции `range` нам нужно указывать не последнее значение переменной цикла, а ограничитель,

который на единицу больше, в начале программы увеличим N на 1:

```
N += 1
```

Для того чтобы задать конечное значение k в цикле, используем функцию sqrt из модуля math, округляя её результат до ближайшего меньшего числа[1)] и добавляя 1:

```
from math import sqrt
for k in range(2, int(sqrt(N))+1):
  ...
```

Здесь многоточие обозначает операторы, составляющие тело цикла.

Теперь преобразуем внутренний цикл: переменная *i* изменяется в диапазоне от k^2 до N с шагом k, поэтому окончательно получаем:

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

Массив для вывода сформируем с помощью генератора списка из тех значений i, для которых соответствующие элементы массива A[i] остались истинными (числа не вычеркнуты):

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

«Длинные» числа

Современные алгоритмы шифрования используют достаточно длинные ключи, которые представляют собой числа длиной 256 бит и больше. С ними нужно выполнять разные операции: складывать, умножать, находить остаток от деления.

К счастью, в Python целые числа могут быть произвольной длины, т. е. размер числа (точнее, отведённый на него объём памяти) автоматически расширяется при необходимости. Однако в других языках (C, C++, Паскаль) для целых чисел отводятся

---

1) Здесь нужно рассмотреть пограничный случай, когда N — квадрат целого числа, т. е. число непростое. Вспомним, что мы предварительно увеличили N на единицу. Тогда результат выражения int(sqrt(N)) в точности будет равен $\sqrt{N}$, это значение нужно тоже включить в цикл перебора, поэтому добавляем 1.

ячейки фиксированных размеров (обычно до 64 бит). Поэтому остро состоит вопрос о том, как хранить такие числа в памяти. Ответ достаточно очевиден: нужно «разбить» длинное число на части так, чтобы использовать несколько ячеек памяти.

Далее мы рассмотрим общие алгоритмы, позволяющие работать с «длинными» числами, при ограниченном размере ячеек памяти. Это позволит вам научиться решать такие задачи с помощью любого языка программирования.

«Длинное число» — это число, которое не помещается в переменную одного из стандартных типов данных языка программирования. Алгоритмы работы с длинными числами называют **«длинной арифметикой»**.

Для хранения «длинного» числа будем использовать массив целых чисел. Например, число 12345678 можно записать в массив с индексами от 0 до 7 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:

1) неудобно выполнять арифметические операции, которые начинаются с младшего разряда;
2) память расходуется неэкономно, потому что в одном элементе массива хранится только один разряд — число от 0 до 9.

Чтобы избавиться от первой проблемы, достаточно «развернуть» массив наоборот, так чтобы младший разряд находился в *A*[0]. В этом случае на рисунках удобно применять 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:

	2	1	0
A	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. Во многих современных языках программирования стандартная ячейка для хранения целого числа занимает 4 байта, так что допустимый диапазон её значений:

$$\text{от } -2^{31} = -2\,147\,483\,648 \text{ до } 2^{31} - 1 = 2\,147\,483\,647.$$

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 такими числами будут выполняться арифметические операции, результат которых должен помещаться в такую же ячейку памяти. Например, если надо умножать разряды этого числа на число $k<100$ и в языке программирования нет 64-битных целочисленных типов данных, то в элементе массива можно хранить не более 7 разрядов.

Задача 1. Требуется вычислить точно значение факториала $100! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot 99 \cdot 100$ и вывести его на экран в десятичной системе счисления (это число состоит более чем из сотни цифр и явно не помещается в одну ячейку памяти).

Мы рассмотрим решение этой задачи, которое подходит для большинства распространённых языков программирования. Для хранения «длинного» числа будем использовать целочисленный массив A. Определим необходимую длину массива. Заметим, что

$$1 \cdot 2 \cdot 3 \cdot \ldots \cdot 99 \cdot 100 < 100^{100}.$$

Число 100^{100} содержит 201 цифру, поэтому число 100! содержит не более 200 цифр. Если в каждом элементе массива записано 6 цифр, для хранения всего числа требуется не более 34 ячеек. В решении на языке Python мы не будем заранее строить массив (список) такого размера, а будем наращивать длину списка по мере увеличения длины числа-результата.

Чтобы найти 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 k in range(2, 101):
  {A} *= k
```

Записать в длинное число единицу — это значит создать массив (список) из одного элемента, равного 1. На языке Python это запишется так:

```
A = {1}
```

Таким образом, остаётся научиться умножать длинное число на «короткое» ($k \le 100$). «Короткими» обычно называют числа, которые помещаются в переменную одного из стандартных типов данных.

Попробуем сначала выполнить такое умножение на примере. Предположим, что в каждой ячейке массива хранится 6 цифр «длинного» числа, т. е. используется система счисления с основанием d = 1 000 000. Тогда число {A} = 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 может находиться только 6 цифр, значит, старшая двойка перейдёт в перенос в следующий разряд. В программе для выделения переноса r можно использовать деление на основание системы счисления d с отбрасыванием остатка. Сам остаток — это то, что остаётся в текущем разряде. Поэтому получаем:

```
s = A[0]*k
A[0] = s%d
r = s//d
```

Для следующего разряда будет всё то же самое, только в первой операции к произведению нужно добавить перенос из предыдущего разряда, который был записан в переменную r. Приняв в самом начале r = 0, запишем умножение длинного числа на короткое в виде цикла по всем элементам массива A:

```
r = 0
for i in range(len(A)):
  s = A[i]*k + r
  A[i] = s%d
  r = s//d
if r > 0:
  A.append(r)
```

Обратите внимание на последние две строки: если перенос из последнего разряда не равен нулю, он добавляется к массиву как новый элемент.

Такое умножение нужно выполнять в другом (внешнем) цикле для всех k от 2 до 100:

```
for k in range(2, 101):
  {A} *= 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 в массиве

	2	1	0
A	1	2	3

хранится значение 1000002000003, а не 123. Поэтому при выводе требуется:

1) вывести (старший) ненулевой разряд числа без лидирующих нулей;
2) вывести все следующие разряды, добавляя лидирующие нули до 6 цифр.

Старший разряд выводим обычным образом (без лидирующих нулей):

```
h = len(A)-1
print(A[h], end = "")
```

Здесь h — номер самого старшего разряда числа. Для остальных разрядов будем использовать возможности функции format:

```
for i in range(h-1, -1, -1):
  print("{:06d}".format(A[i]), end = "")
```

Напомним, что фигурные скобки обозначают место для вывода очередного элемента данных. Формат 06d говорит о том, что

нужно вывести целое число в десятичной системе (d, от англ. *decimal* — десятичный), используя 6 позиций, причём пустые позиции нужно заполнить нулями (первая цифра 0).

Отметим, что показанный выше метод применим для работы с «длинными» числами в любом языке программирования. Как мы говорили, в Python по умолчанию используется «длинная» арифметика, поэтому вычисление 100! может быть записано значительно короче:

```
A = 1
for i in range(2, 101):
  A *= i
print(A)
```

или с использованием функции `factorial` из модуля `math`:

```
import math
print(math.factorial(100))
```

Квадратный корень

Рассмотрим ещё одну задачу: вычисление целого квадратного корня из целого числа. На этот раз мы будем использовать встроенную «длинную арифметику» языка Python, так что число может быть любой длины. К сожалению, стандартная функция `sqrt` из модуля `math` не поддерживает работу с целыми числами произвольной длины, и результат её работы — вещественное число[1]. Поэтому в том случае, когда нужно именно целое значение, приходится использовать специальные методы.

Один из самых известных **алгоритмов вычисления квадратного корня**, полученный ещё в Древней Греции, — **метод Герона Александрийского**, который сводится к многократному применению формулы[2]

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

Здесь a — число, из которого извлекается корень, а x_{i-1} и x_i — предыдущее и следующее приближения (см. главу 9 учебника для 10 класса). Фактически здесь вычисляется среднее арифме-

---

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 $x^2 = a$.

тическое чисел x_{i-1} и a/x_{i-1}. Пусть одно из этих значений меньше $\sqrt{a}$, тогда второе обязательно больше $\sqrt{a}$. Поэтому их среднее арифметическое с каждым шагом приближается к значению корня.

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

Приведённая формула служит для вычисления вещественного значения корня. Для того чтобы найти целочисленное значение корня (т. е. *максимальное целое число, квадрат которого не больше*, чем a), 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 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 (в функции) с помощью **return**. Мы начинаем поиск с начального приближения $x_0 = a$, которое (при больших a) заведомо больше правильного ответа. Поэтому каждое следующее приближение будет меньше предыдущего. А как только очередное приближение окажется *большим или равным* предыдущему, квадратный корень будет найден.

Выводы

- Если все данные и необходимые результаты — целые числа, желательно выполнять вычисления, используя только операции с целыми числами.

- «Длинное» число — это число, которое не помещается в переменную одного из стандартных типов данных языка программирования. Алгоритмы работы с длинными числами называют «длинной арифметикой».
- Для хранения «длинного» числа в памяти можно использовать массив.
- В языке Python используется «длинная арифметика»: размер памяти для хранения числа автоматически увеличивается, если это необходимо.

Нарисуйте в тетради интеллект-карту этого параграфа.

Вопросы и задания

1. Какие преимущества и недостатки имеет алгоритм «решето Эратосфена» по сравнению с проверкой каждого числа на простоту?
2. В каких случаях необходимо применять «длинную арифметику»?
3. Какое максимальное число можно записать в ячейку размером 64 бита? Рассмотрите варианты хранения чисел со знаком и без знака.
4. Можно ли использовать для хранения «длинного» числа символьную строку? Оцените достоинства и недостатки такого подхода.
5. Почему неудобно хранить «длинное» число, записывая первую значащую цифру в начало массива?
6. Сколько разрядов десятичной записи числа можно хранить в одной 16-битной ячейке?
7. Объясните, какие проблемы возникают при выводе длинного числа. Как их можно решать?

*8. Предложите способ вывода «длинного» числа без использования возможностей функции `format`.

*9. Предложите алгоритм поиска целого кубического корня из целого числа.

Подготовьте сообщение

а) «Простые числа в криптографии»

б) «Применение факториалов»

в) «Извлечение квадратного корня из больших чисел»

Проекты

а) Программа для генерации ключей алгоритма *RSA*
б) Исследование скорости работы алгоритмов поиска простых чисел

Интересный сайт

ru.numberempire.com/primenumbers.php — онлайн-калькулятор простых чисел

§ 36 Структуры

Ключевые слова:

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

Зачем нужны структуры?

Представим себе базу данных библиотеки, в которой хранится информация о книгах. Для каждой из них нужно запомнить автора, название, год издания, количество страниц, число экземпляром 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 оказывается слишком неудобным и ненадёжным. Например, при сортировке нужно переставлять элементы всех массивов (отдельно!) и можно легко ошибиться и нарушить связь данных.

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

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

Классы

В Python для работы со структурами используют специальные типы данных — **классы**. В программе можно вводить свои классы (новые типы данных). Введём новый класс TBook — структуру, с помощью которой можно описать книгу в базе данных библиотеки. Будем хранить в структуре только[1]:

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

Класс можно объявить так:

```
class TBook:
  pass
```

Объявление начинается с ключевого слова **class**. Имя нового класса (типа данных) — TBook — это удобное сокращение от английских слов *Type Book* (тип «книга»), хотя можно было использовать и любое другое имя, составленное по правилам языка программирования.

Слово **pass** (англ. *pass* — пропустить) стоит во второй строке только потому, что оставить строку совсем пустой нельзя — будет ошибка. В данном случае пока мы не определяем какие-то новые характеристики для объектов этого класса, просто говорим, что есть такой класс.

В отличие от других языков программирования (C, 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 случайных ошибок. Например, при опечатке в названии поля может быть создано новое поле с неверным именем, и сообщение об ошибке не появится.

Теперь уже можно создать объект этого класса:

```
B = TBook()
```

или даже массив (список) из таких объектов:

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

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

Обратите внимание, что при создании массива нельзя было написать

```
Books = [TBook()]*100 # ошибочное создание массива
```

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.

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 им новые значения:

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

использовать их 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, бывает нужно читать массивы структур из файла и записывать в файл. Конечно, можно хранить структуры в текстовых файлах, например, записывая все поля одной структуры в одну строку и разде-

ляя 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()
F = open("books.dat", "wb")
B.author = "Тургенев И.С. "
B.title = "Муму"
B.count = 2
pickle.dump(B, F);
F.close()
```

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

С помощью цикла можно записать в файл массив структур:

```
for B in Books:
  pickle.dump(B, F)
```

а можно даже обойтись без цикла:

```
pickle.dump(Books, F);
```

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 читать.

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

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

Если массив (список) структур записывался в файл за один раз, его легко прочитать, тоже за один вызов функции `load`:

```
Books = pickle.load (F)
```

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

```
for i in range(N):
  Books[i] = pickle.load(F)
```

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

```
Books = []
while True:
  try:
    Books.append (pickle.load(F))
  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.

Исключением может быть, например, деление на ноль, ошибка при вводе или выводе данных, нехватка 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) после слова `try`. После этого в следующем блоке, который начинается с ключевого слова `except`, записывают команды, которые нужно выполнить в случае ошибки (в данном случае — выйти из цикла).

Сортировка

Для сортировки массива структур применяют те же методы, что и для сортировки массива простых переменных. Структуры

обычно сортируют по возрастанию или убыванию одного из полей, которое называют **ключевым полем** или **ключом**, хотя можно, конечно, использовать и сложные условия, зависящие от нескольких полей (составной ключ, как в базах данных).

Отсортируем массив `Books` (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 из курса 10 класса, 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.

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 класса `TBook`. Здесь нужно ему помочь — указать, какое из полей играет роль ключа. Для этого используем именованный параметр `key` метода `sort`. 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 так называемую **«лямбда-функцию»**, т. е. функцию без имени (вспомните материал учебника для 10 класса):

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

Здесь в качестве ключа указана функция, которая из переданного ей параметра `x` выделяет поле `author`, т. е. делает то же самое, что и показанная выше 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 — это сложный тип данных, который позволяет объединить данные разных типов. Элементы структуры называют полями.
- Структуры в языке Python вводятся как классы — новые типы данных.
- При обращении к полям структуры используется точечная запись:

  `<имя структуры>.<имя поля>`

- Для сохранения структур в файле используются подпрограммы модуля `pickle`.
- При сортировке массива структур необходимо задать функцию, возвращающую ключ, по которому выполняется сортировка.

Нарисуйте в тетради интеллект-карту этого параграфа.

Вопросы и задания

1. Что такое структура? В чём её отличие от массива?
2. В каких случаях использование структур даёт преимущества? Какие именно?
3. Как объявляется новый тип данных для хранения структур в Python? Выделяется ли при этом память?
4. Как обращаются к полю структуры?
5. Что такое двоичный файл? Чем он отличается от текстового?
6. Как можно сортировать структуры?

Подготовьте сообщение

а) «Структуры (записи) в языке Паскаль»
б) «Структуры в языке Си»
в) «Структуры в языке JavaScript»

Проект

База данных текстового формата

§ 37 Словари

Ключевые слова:

- словарь
- ключ
- значение

Что такое словарь?

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

Для решения задачи нам нужно составить особую структуру данных — словарь (англ. *dictionary*), в котором хранить пары «слово — количество». Таким образом, мы хотим искать нужные нам данные не по числовому индексу элемента (как в обычном массиве), а по слову (символьной строке). Например, вывести на экран количество найденных слов «бегемот» можно было бы так:

```
print (D["бегемот"])
```

где `D` — имя словаря.

Тип данных «словарь» есть в некоторых современных языках программирования. В языке Python он называется `dict` (от англ. *dictionary*)[1)].

[1)] В других языках программирования используются другие названия, например «ассоциативный массив» или «хэш».

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 в языке. Поэтому сортировку словаря выполнить невозможно, как невозможно, в отличие от списка, указать для какого-то элемента словаря его соседей (предыдущий и следующий элементы).

Ключом могут быть данные любого неизменяемого типа, например число, символьная строка или *кортеж* (неизменяемый набор значений). В одном словаре можно использовать ключи разных типов.

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 программирования.

Словарь определяется с помощью фигурных скобок, например:

```
D = {"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:

```
D = {}
```

Для того чтобы добавить элемент в словарь, используют присваивание:

```
D["самолёт"] = 1
```

Если ключ «самолёт» 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:

```
D["самолёт"] += 1
```

Однако, если ключа «самолёт» нет в словаре, такая команда вызовет ошибку. Для того чтобы определить, есть ли в словаре какой-то ключ, можно использовать оператор **in**:

```
if "самолёт" in D:
  D["самолёт"] += 1
else:
  D["самолёт"] = 1
```

Если есть ключ «самолёт», соответствующее значение увеличивается на 1. Если такого ключа нет, то он создаётся и связывается со значением, равным 1.

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

```
D["самолёт"] = D.get("самолёт", 0)+1
```

В данном случае значение по умолчанию — 0, если ключа «самолёт» нет, создаётся элемент с таким ключом и после выполнения приведённой команды соответствующее значение будет равно 1.

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

```
D = {}
F = open("input.txt")
while True:
  word = F.readline().strip() # (*)
  if not word: break
  D[word] = D.get(word, 0)+1
F.close()
```

Обратим внимание на строку (*) в программе. После чтения очередной строки из файла `F` (это делает метод `readline`, вспомните

материал главы 8 учебника для 10 класса) вызывается ещё и метод `strip` (в переводе с англ. «лишать», «удалять»), который удаляет лишние пробелы и завершающий символ перевода строки `\n`.

Теперь остаётся вывести результат в файл. В отличие от списка к элементам словаря нельзя обращаться по индексам. Тогда возникает вопрос — как же перебрать все возможные ключи? Для этой цели мы запросим у словаря список всех ключей, используя метод `keys`:

```
allKeys = D.keys()
```

Эти ключи нужно отсортировать, тут работает функция `sorted`, которая вернёт отсортированный по алфавиту список:

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

В последних версиях языка Python вместо этого можно записать просто

```
sortKeys = sorted(D)
```

не вызывая явно метод `keys`.

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

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

Ключи из отсортированного списка ключей попадают по очереди в переменную `k`, для каждого ключа выводится сам 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 функция `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 в словаре. Например, вот так можно вывести значения для всех ключей:

```
for i in D.values():
  print(i)
```

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 k, v in D.items():
  print (k, "->", v)
```

В этом цикле две изменяемых переменных: `k` (ключ) и `v` (значение). Поскольку `D.items()` — это список пар «ключ — значение», при переборе первый элемент пары (ключ) попадает в переменную `k`, а второй (значение) — в переменную `v`.

Выводы

- Словарь — это набор пар «ключ — значение».
- Элементы в словаре неупорядочены, т. е. нельзя указать для элемента предыдущий и следующий.
- Ключом может быть любое неизменяемое значение: число, строка, кортеж.

Нарисуйте в тетради интеллект-карту этого параграфа.

Вопросы и задания

1. Какие операции можно выполнять со словарём?
2. Можно ли обратиться к элементам словаря по индексу? Как вы думаете, почему сделано именно так?
3. Как создать словарь из готовых пар «ключ — значение»? Найдите в литературе или в Интернете разные способы решения этой задачи.
4. Как обращаться к элементу словаря?
5. Чем можно заменить метод `get`?
6. Как получить список всех ключей словаря? Всех значений словаря?
7. Как перебрать все пары «ключ — значение»?
8. Зачем при чтении строк из файла используется метод `strip`?

Подготовьте сообщение

а) «Словари в языке C++ (библиотека *STL*)»
б) «Ассоциативные массивы в языках программирования»

Проект

Сравнение текстов разных авторов с помощью алфавитно-частотного словаря

§ 38
Стек, очередь, дек

Ключевые слова:

- стек
- очередь
- 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 тоже нельзя.

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

На рисунке 6.1 показаны примеры стеков вокруг нас, в том числе автоматный магазин и детская пирамидка.

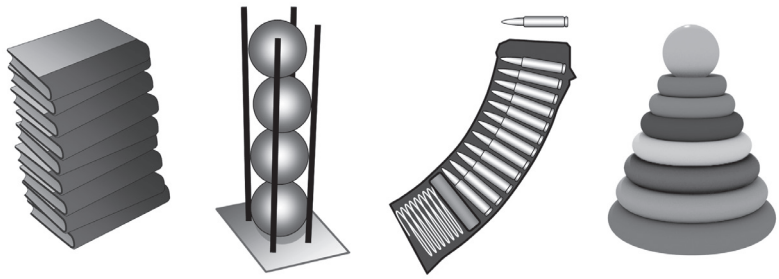


Рис. 6.1

Как вы знаете из главы 8 учебника для 10 класса, стек используется при выполнении программ: в этой области оперативной памяти хранятся адреса возврата из подпрограмм; параметры, передаваемые функциям и процедурам, а также локальные переменные.

Задача 1. В файле записаны целые числа. Нужно вывести их в другой файл в обратном порядке.

В этой задаче очень удобно использовать стек. 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).

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. 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 можно применить уже знакомый нам метод `append`:

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

Чтобы вытолкнуть элемент из стека, используется метод `pop`:

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

Метод `pop` — это функция, которая выполняет две задачи:

1) удаляет последний элемент списка (если вызывается без параметров[1]);
2) возвращает удалённый элемент как результат функции, так что его можно сохранить в какой-либо переменной.

[1] При вызове метода `pop` в скобках можно указать индекс удаляемого элемента.

Теперь несложно написать цикл ввода данных в стек из файла:

```
F = open("input.txt")
stack = []
while True:
  s = F.readline()
  if not s: break
  stack.append(int(s))
F.close()
```

или даже так:

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

Затем выводим элементы массива в файл в обратном порядке:

```
F = open("output.txt", "w")
while len(stack) > 0:
  x = stack.pop()
  F.write(str(x)+"\n")
F.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 ещё сократить, заодно избавившись от переменной x:

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

Вычисление арифметических выражений

Вы не задумывались, как компьютер вычисляет арифметические выражения, записанные в такой форме: `(5+15)/(4+7-1)`? Такая запись называется **инфиксной** — в ней знак операции расположен *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 Лукашевич предложил **префиксную форму**, которую стали называть **польской нотацией**. В ней знак операции расположен *перед* операндами. Например, выражение (5 + 15)/(4 + 7 – 1) может быть записано в виде

```
/ + 5 15 - + 4 7 1
```

Скобки здесь не требуются, так как порядок операций строго определён: сначала выполняются два сложения (`+ 5 15` и `+ 4 7`), затем вычитание, и наконец, деление. Первой стоит последняя операция.

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

```
5 15 + 4 7 + 1 - /
```

В этом случае также не нужны скобки, и выражение может быть вычислено за один просмотр 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, и после этого в стек вталкивается результат выполнения этой операции.

На рисунке 6.2 показано, как работает этот алгоритм (стек «растёт» снизу вверх).

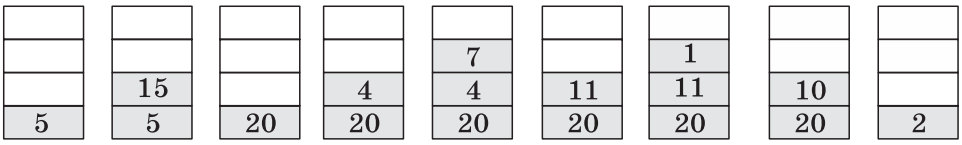


Рис. 6.2

В результате в стеке остаётся значение заданного выражения.

Приведём программу, которая вводит с клавиатуры выражение, записанное в постфиксной форме, и вычисляет его:

```
data = input().split()                          # (1)
stack = []                                      # (2)
for x in data:                                  # (3)
  if x in "+-*/":                               # (4)
    op2 = int(stack.pop())                      # (5)
    op1 = int(stack.pop())                      # (6)
    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 программы 1 (см. нумерацию в комментариях) 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 обратно 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 — результат, — который выводится на экран (строка 13).

Скобочные выражения

Задача 2. Вводится символьная строка, в которой записано некоторое (арифметическое) выражение, использующее скобки трёх типов: `()`, `[]` и `{}`. Проверить, правильно ли расставлены скобки.

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

```
[()     [[[()     [()}     )(     ([)]
```

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

Начнём с аналогичной задачи, в которой используется только один вид скобок. Её можно решить с помощью счётчика скобок.

Сначала счётчик равен нулю. Строка 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 (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 отдельно. Однако это решение неверное. Например, для 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.

Строка просматривается слева направо. Если очередной символ — открывающая скобка, будем вталкивать на вершину стека соответствующую *закрывающую* скобку. Если это закрывающая скобка, то проверяем, что лежит на вершине стека: если там та же самая закрывающая скобка, то её нужно просто снять со стека. Если на вершине лежит другая скобка 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 иллюстрируется на рис. 6.3 (для правильного выражения).

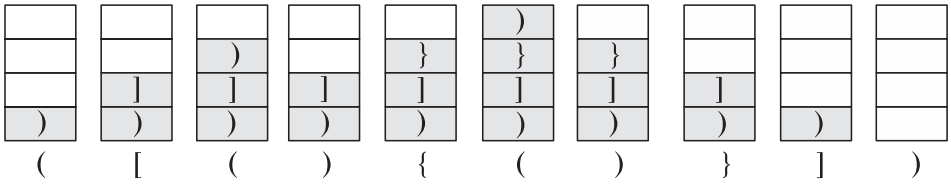


Рис. 6.3

В программе нам будет удобно использовать функцию, которая для каждой открывающей скобки возвращает соответствующую закрывающую:

```
def pair (b):
  if c == "(": return ")"
  elif c == "[": return "]"
  elif c == "{": return "}"
  else: return "?"
```

В основной программе создадим пустой стек:

```
stack = []
```

Логическая переменная `err` будет сигнализировать об ошибке. Сначала ей присваивается значение `False` (ложь):

```
err = False
```

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 ")]}":                                   # (4)
    if len(stack) == 0 or c != stack.pop():          # (5)
      err = True                                     # (6)
      break                                          # (7)
```

Если очередной символ — открывающая скобка, вталкиваем в стек соответствующую ей закрывающую скобку (строка 3).

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

1) стек пуст;
2) текущий символ (в переменной `c`) не совпал с символом, который снят с вершины стека.

Если выполняется хотя бы одно из этих условий, то выражение ошибочно. При этом в переменную `err` в строке 6 записывается значение `True` (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 не пуст, то в выражении есть незакрытые скобки, и оно ошибочно:

```
if stack: err = True
```

В конце программы остаётся вывести результат на экран:

```
if not err:
  print("Выражение правильное.")
else:
  print("Выражение неправильное.")
```

Очереди, деки

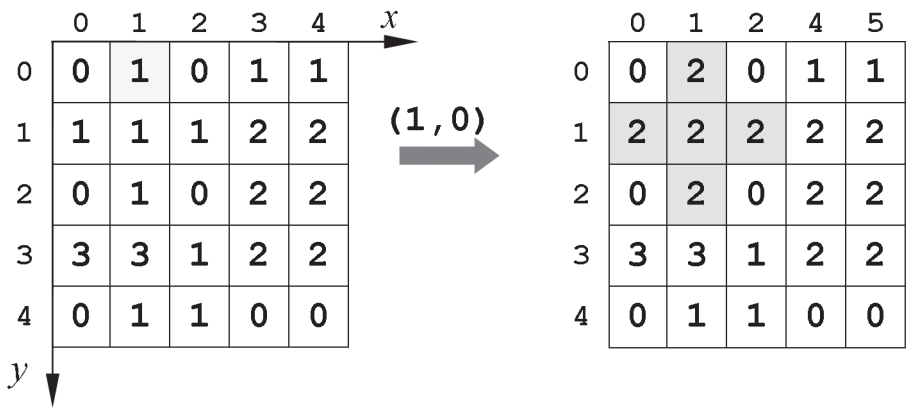
Все мы знакомы с принципом очереди: **первым пришёл — первым обслужен** (англ. **FIFO**: *First In — First Out*). Соответствующая структура данных в информатике тоже называется очередью.

Очередь — это линейная структура данных, для которой введены две операции:
- добавление нового элемента в конец очереди;
- удаление первого элемента из очереди.

Очередь — это не просто теоретическая модель. Операционные системы используют очереди для организации сообщения 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.

Задача 3. Рисунок задан в виде матрицы A, в которой элемент $A[x][y]$ определяет цвет пикселя на пересечении столбца x и строки y. Перекрасить в цвет 2 одноцветную область, начиная с пикселя (x_0, y_0).

На рисунке 6.4 показан результат такой заливки для матрицы из 5 строк и 5 столбцов с начальной точкой (1,0).



y \ x	0	1	2	3	4
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

y \ x	0	1	2	4	5
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. 6.4

Отметим, что для удобства записи мы храним матрицу по столбцам: первый индекс обозначает столбец, а второй — строку.

Эта задача актуальна для графических программ. Один из возможных вариантов решения 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, как-то помечая точки, уже добавленные в очередь, чтобы не добавлять их 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). Тогда можно определить размеры матрицы так:

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

Значение XMAX — число столбцов, а YMAX — это число строк.

Определим также цвет заливки:

```
NEW_COLOR = 2
```

Зададим координаты начальной точки, откуда начинается заливка:

```
(x0, y0) = (1, 0)
```

и запомним её цвет в переменной *color*:

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

Теперь создадим очередь (как список языка Python) и добавим в эту очередь точку с начальными координатами. Две координаты точки связаны между собой, поэтому в программе лучше объединить их в единый блок, 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.

Остаётся написать основной цикл[1]:

```
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 в очереди есть хоть один элемент: запись **while** `Q` для списков равносильна записи **while** `len(Q) > 0`. Начало очереди всегда совпадает с первым 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, поэтому мы сразу разбиваем его на отдельные координаты.

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 текущей и попадающие на поле рисунка.

В некоторых языках программирования размер массива нельзя менять во время работы программы. В этом случае очередь моделируется иначе. Допустим, что мы знаем, что количество элементов в очереди всегда меньше N. Тогда можно выделить статический массив из N элементов и хранить в отдельных переменных номера первого элемента очереди (**«головы»**, англ. *head*) и последнего элемента (**«хвоста»**, англ. *tail*). На рисунке 6.5, *а* показана очередь из 5 элементов. В этом случае удаление элемента из очереди сводится просто к увеличению переменной `Head` (рис. 6.5, *б*).

---

[1] Использование списка для моделирования очереди — не самый быстродействующий вариант, потому что удаление первого элемента массива выполняется долго. В практических задачах лучше использовать класс `deque` из модуля `collections`.

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 1. Таким образом, массив целиком замкнутым «в кольцо». На рисунке 6.5, *в* показана целиком заполненная очередь, а на рис. 6.5, *г* — пустая очередь. Один элемент массива всегда остаётся незанятым, иначе невозможно будет различить состояния «очередь пуста» и «очередь заполнена».

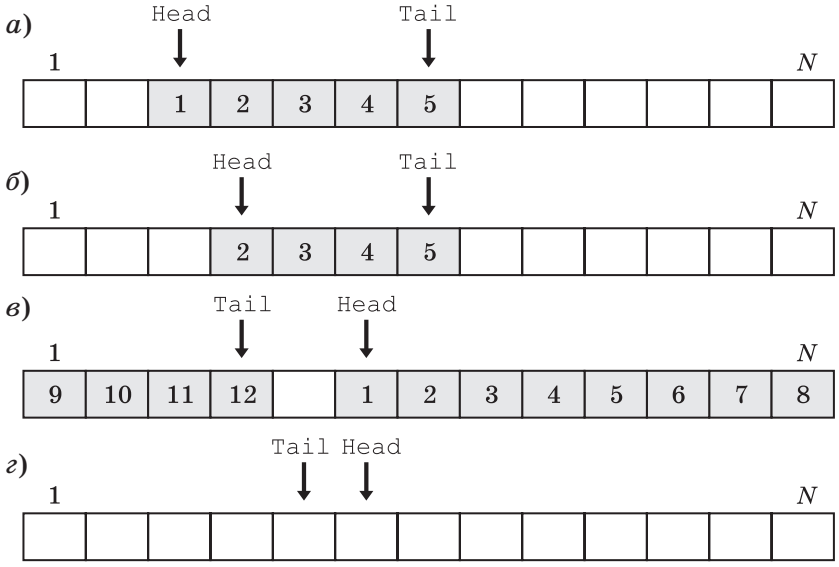


Рис. 6.5

Отметим, что эта модель описывает работу кольцевого буфера клавиатуры, который может хранить до 15 двухбайтных слов.

Существует ещё одна линейная динамическая структура данных, которая называется *дек*.

Дек (от англ. **deque**: *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 (рис. 6.6).

Рис. 6.6

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

1) добавление элемента `x` в конец дека: `d.append(x)`;
2) добавление элемента *x* в начало дека: `d.insert(0, x)` (добавляемый элемент будет иметь индекс 0);
3) удаление элемента с конца дека: `d.pop()`;
4) 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. Системный стек применяется для хранения адресов возврата из подпрограмм и размещения локальных переменных.
- 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.

Нарисуйте в тетради интеллект-карту этого параграфа.

Вопросы и задания

1. Какие операции со стеком разрешены?
2. Какие ошибки могут возникнуть при использовании стека?
3. Какие операции допускает очередь?
4. Как построить очередь на основе массива с неизменяемым размером?
5. Приведите примеры задач, в которых можно использовать очередь.
6. Что такое дек? Чем он отличается от стека и очереди? Какая из этих структур данных наиболее общая (может выполнять функции других)?

Подготовьте сообщение

а) «Стеки и очереди в языке Паскаль»
б) «Стеки и очереди в языке С++»
в) «Очереди с приоритетом»

Проекты

а) Стековый язык программирования
б) Моделирование системы массового обслуживания с помощью очереди
в) Сравнение алгоритмов закраски области
г) Программа для перевода арифметического выражения в постфиксную форму

§ 39 Деревья

Ключевые слова:

- двоичное дерево
- дерево поиска
- ключ
- обход в глубину
- обход в ширину

Что такое дерево?

Как вы знаете из учебника для 10 класса, **дерево** — это структура данных, отражающая иерархию (отношения подчинённости, многоуровневые связи). Напомним некоторые основные понятия, связанные с деревьями.

Дерево состоит из **узлов** и связей между ними (они называются **дугами**). Самый первый узел, расположенный на верхнем уровне (в него не входит ни одна стрелка-дуга), — это **корень дерева**. Конечные узлы, из которых не выходит ни одна дуга, называются **листьями**. Все остальные узлы, кроме корня и листьев, — это промежуточные узлы.

Из двух связанных узлов тот, который находится на более высоком уровне, называется **родителем**, а другой — **сыном**. 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 стрелкам от узла-предка. Соответственно, **предок** какого-то узла — это узел, из которого можно перейти по стрелкам в данный узел.

В дереве на рис. 6.7 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.

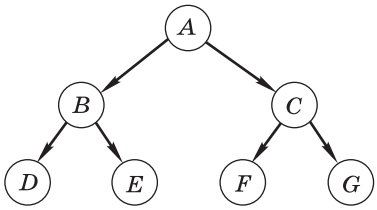


Рис. 6.7

Высота дерева — это наибольшее расстояние (количество рёбер) от корня до листа. Высота дерева, приведённого на рис. 6.7, равна 2.

Формально **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.

Здесь множество объектов (деревьев) определяется через само это множество на основе простого базового случая (пустого дерева). Такой приём называется **рекурсией**. Согласно этому определению, 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.

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).

Деревья широко применяются в следующих задачах:
- поиск в большом массиве данных;
- сортировка данных;
- вычисление арифметических выражений;
- оптимальное кодирование данных (метод сжатия Хаффмана, см. главу 1).

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 специальным образом построенного дерева (рис. 6.8).

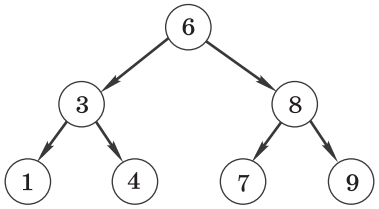


Рис. 6.8

Значения, связанные с каждым из узлов 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, ключи которых меньше или равны ключу данного узла;
- справа от каждого узла находятся узлы, ключи которых больше или равны ключу данного узла.

Дерево, обладающее такими свойствами, называется **двоичным деревом поиска**.

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. Количество операций сравнения в этом случае пропорционально $\log_2 N$, т. е. алгоритм имеет асимптотическую сложность $O(\log_2 N)$. 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 — левый — правый» (обход в прямом порядке):

```
посетить корень
обойти левое поддерево
обойти правое поддерево
```

- ЛКП = «левый — корень — правый» (симметричный обход):

```
обойти левое поддерево
посетить корень
обойти правое поддерево
```

- ЛПК = «левый — правый — 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. 6.9.

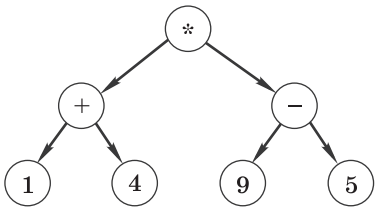


Рис. 6.9

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 мы получили префиксную форму записи арифметического выражения, во втором — привычную нам инфиксную форму (только без скобок), а в третьем — постфиксную форму. Напомним, что в префиксной и в постфиксной формах скобки не нужны.

Обход КЛП называется **обходом 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 несложно записать и без рекурсии, используя вспомогательный *стек*:

```
записать в стек корень дерева
while стек не пуст:
  выбрать 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 6.10 показано изменение состояния стека при таком обходе дерева, изображенного на рис. 6.9. Под стеком записана метка узла, который посещается (например, данные из этого узла выводятся на экран).

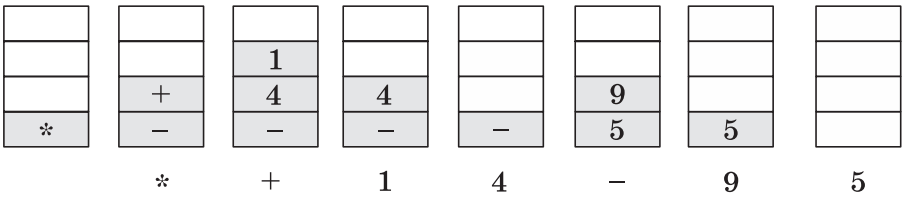


Рис. 6.10

Существует ещё один способ обхода, который называют **обходом в ширину** (англ. **BFS**: *Breadth-first search*). Сначала посещают корень дерева, затем — всех его сыновей, затем — сыновей сыновей («внуков») и т. д., постепенно спускаясь на один уровень вниз. Обход в ширину для дерева на рис. 6.9 даст такую последовательность посещения узлов:

```
* + - 1 4 9 5
```

Для того чтобы выполнить такой обход, применяют *очередь* — в неё записывают узлы, которые необходимо посетить. На псевдокоде обход в ширину можно записать так:

```
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

Теперь попробуем записать все эти операции в программе 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, ссылка на левое поддерево и ссылка на правое поддерево. У листьев нет сыновей, в этом случае в указатели будем записывать специальное значение `None` («пусто», «ничто»). Дерево, состоящее из трёх таких узлов, показано на рис. 6.11.

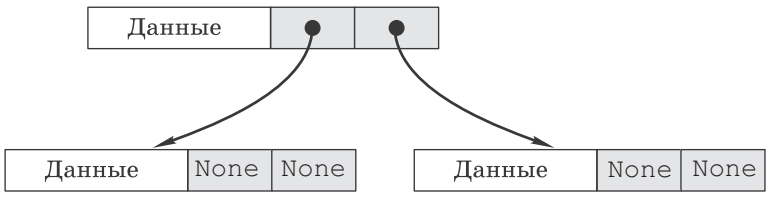


Рис. 6.11

В нашем случае область данных узла будет содержать одно поле — символьную строку, в которую записывается знак операции или число в символьном виде.

Будем хранить дерево в памяти как набор связанных структур-узлов класса `TNode`:

```
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
  newNode.left = L
  newNode.right = R
  return newNode
```

Запись L = None, R = None в заголовке функции означает, что если параметры L и R не заданы, их значения по умолчанию равны None. Функция возвращает новый узел дерева.

Теперь мы можем построить в памяти дерево, показанное на рис. 6.9:

```
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)
```

А вот такой же обход без рекурсии, использующий стек:

```
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) # берём первый элемент из 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 арифметических выражений основан на использовании дерева. Сначала выражение, записанное в линейном виде (в одну строку), нужно «разобрать» и построить соответствующее ему дерево. Затем в результате прохода по этому дереву от листьев к корню вычисляется результат.

Для простоты будем рассматривать только арифметические выражения, содержащие числа и знаки четырёх арифметических действий: + - * /. Построим дерево для 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. 6.12).

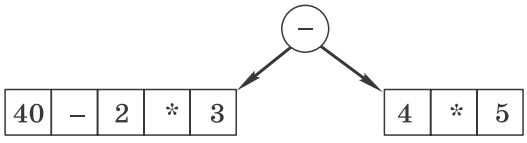


Рис. 6.12

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 нужно построить таким же способом левое и правое поддеревья (рис. 6.13).

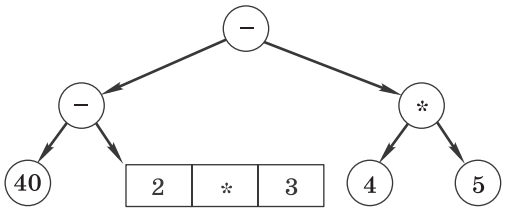


Рис. 6.13

Левое поддерево требует ещё одного шага (рис. 6.14):

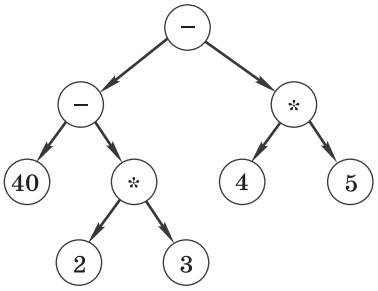


Рис. 6.14

Эта процедура рекурсивная, её можно записать на псевдокоде так:

```
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 s — символьная строка, в которой записано арифметическое выражение (будем предполагать, что это правильное выражение без скобок). Вычисление выражения сводится к двум вызовам функций:

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

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

При построении дерева нужно выделить в памяти место для нового узла, а затем искать последнюю выполняемую операцию — это будет делать функция lastOp. Она возвращает −1, если ни одной операции не обнаружено, в этом случае создаётся лист — узел без потомков. Если операция найдена, в поле data нового узла записывается её обозначение, а в поля left и right — адреса поддеревьев, которые строятся рекурсивно для левой и правой частей выражения:

```
def makeTree(s):
  k = lastOp(s)
  if k < 0:                # создать лист
    Tree = node(s)
  else:                    # создать узел-операцию
    Tree = node(s[k])
    Tree.left = makeTree(s[:k])
    Tree.right = makeTree(s[k+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 oper(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). В противном случае вычисляются значения для левого и правого поддеревьев, и к ним применяется операция, указанная в корне дерева, — эту работу выполняет функция oper:

```
def oper(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 (s):
  minPrt = 50   # любое между 2 и 100
  k = -1
  for i in range(len(s)):
    if priority(s[i]) <= minPrt:
      minPrt = priority(s[i])
      k = i
  return k
```

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

Начальное значение переменной minPrt можно выбрать любое между наибольшим приоритетом операций (2) и условным кодом «не операции» (100). Тогда, если найдена любая операция, услов-

ный оператор срабатывает, а если в строке нет операций, условие всегда ложно и в переменной `k` остаётся начальное значение −1, которое и вернёт функция `lastOp`.

Хранение двоичного дерева в массиве

Двоичные деревья можно хранить в массиве (списке). Вопрос о том, как сохранить структуру (взаимосвязь узлов), решается достаточно просто. Если нумерация элементов массива `A` начинается с 0, то сыновья элемента `A[i]` — это `A[2*i+1]` и `A[2*i+2]`. На рисунке 6.15 показан порядок расположения элементов в массиве для дерева, соответствующего выражению

```
40 – 2 * 3 - 4 * 5
```

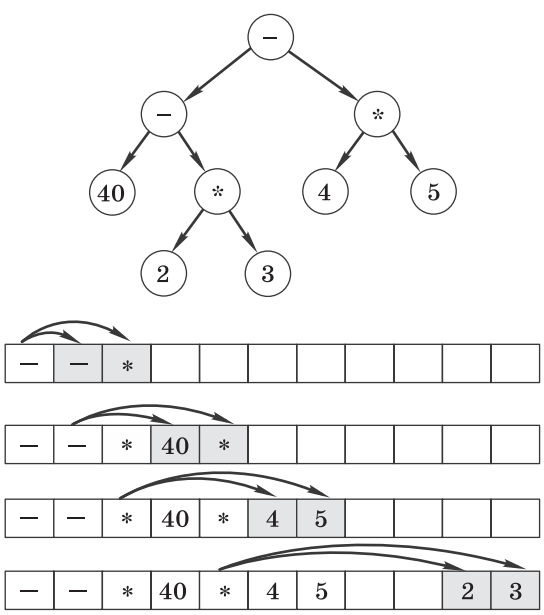


Рис. 6.15

Алгоритм вычисления выражения остаётся прежним, изменяется только метод хранения данных. Обратите внимание, что некоторые элементы остались пустыми, это значит, что их «родитель» — лист дерева. В программе на Python в такие (неиспользуемые) элементы массива можно записывать «пустое» значение `None`. Конечно, лучше всего так хранить сбалансированные деревья, иначе будет много пустых элементов, которые зря расходуют память.

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 связаны друг с другом, но слабо связаны с другими процедурами и функциями.

В нашей программе в отдельный модуль можно вынести все операции с деревьями, т. е. определение класса `TNode` и все подпрограммы. 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 с помощью команды **`import`**:

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

или загружая только нужные нам функции:

```
from bintree import makeTree, calcTree
…
T = makeTree (s)
print("Результат: ", calcTree(T))
```

Обратите внимание, что нам не нужно знать, как именно устроены функции `makeTree` и `calcTree`: какие типы данных они используют, по каким алгоритмам работают и какие дополнительные функции вызывают. Для использования функции модуля достаточно знать *интерфейс* — соглашение о передаче параметров (какие параметры принимают подпрограммы 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 больших программ. Разделение программы на модули облегчает понимание и совершенствование программы, потому что каждый модуль можно разрабатывать, изучать и оптимизировать независимо от других.

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 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 или как список особой структуры.

Нарисуйте в тетради интеллект-карту этого параграфа.

Вопросы и задания

1. Где используются структуры типа «дерево» в информатике и в других областях?
2. Объясните рекурсивное определение дерева.
3. Можно ли считать, что линейный список — это частный случай дерева?
4. Какими свойствами обладает дерево поиска?
5. Подумайте, как можно построить дерево поиска из массива данных.
6. Какие преимущества имеет поиск с помощью дерева?
7. Какие способы обхода дерева вы знаете? Придумайте какие-нибудь другие способы обхода.
8. Как строится дерево для вычисления арифметического выражения?
9. Как можно представить дерево в программе на Python?
10. Как указать, что узел дерева не имеет левого (правого) сына?
11. Как вы думаете, почему рекурсивные алгоритмы работы с деревьями получаются проще, чем нерекурсивные?
12. Как хранить двоичное дерево в массиве? Можно ли использовать такой приём для хранения деревьев, в которых узлы могут иметь больше двух сыновей?

Подготовьте сообщение

а) «Использование деревьев в базах данных»
б) «Префиксные деревья (*trie*)»
в) «Деревья в языке Паскаль»

г) «Деревья в языке Си»
д) «Диаграммы связей (*mind-maps*)»

Проекты

а) Программа для сортировки данных с помощью дерева
б) Калькулятор арифметических выражений

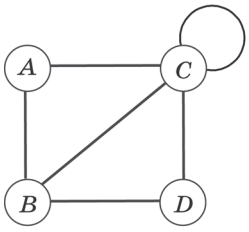
§ 40 Графы

Ключевые слова:

- матрица смежности
- весовая матрица
- орграф
- «жадный» алгоритм
- остовное дерево
- кратчайший маршрут
- алгоритм Дейкстры
- алгоритм Флойда–Уоршелла
- список смежности
- количество путей

Что такое граф?

Как вы знаете из курса 10 класса, граф — это набор вершин (узлов) и связей между ними (рёбер). Информацию о вершинах и рёбрах графа обычно хранят в виде таблицы специального вида — **матрицы смежности** (рис. 6.16).



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

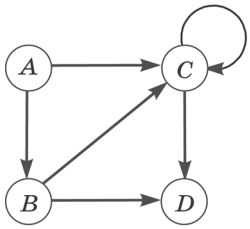
Рис. 6.16

Единица на пересечении строки A и столбца B означает, что между вершинами A и B есть связь. Ноль указывает на то, что связи нет. Матрица смежности симметрична относительно главной диагонали (серые клетки в таблице). Единица на главной диагонали обозначает **петлю** — ребро, которое начинается и заканчивается в одной и той же вершине (в данном случае — в вершине C).

В рассмотренном примере 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 пункта, можно сделать вывод, что дерево — это частный случай связного графа, в котором нет замкнутых путей — **циклов**.

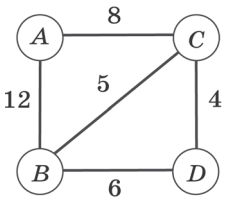
Если для каждого ребра указано направление, граф называют **ориентированным** (или **орграфом**). Рёбра орграфа называют **дугами**. Его матрица смежности не всегда симметричная. Единица, стоящая на пересечении строки A и столбца B, говорит о том, что существует дуга из вершины A в вершину B (рис. 6.17).



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

Рис. 6.17

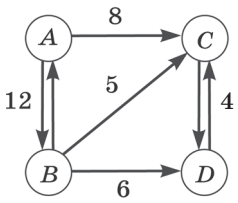
Часто с каждым ребром связывают некоторое число — **вес ребра**. Это может быть, например, расстояние между городами или стоимость проезда. Такой граф называется **взвешенным**. Информация о взвешенном графе хранится в виде **весовой матрицы**, содержащей веса рёбер (рис. 6.18).



	A	*B*	*C*	*D*
A		12	8	
B	12		5	6
C	8	5		4
D		6	4	

Рис. 6.18

У взвешенного орграфа весовая матрица может быть несимметрична относительно главной диагонали (рис. 6.19).



	A	*B*	*C*	*D*
A		12	8	
B	12		5	6
C				4
D			4	

Рис. 6.19

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 задачи.

«Жадные» алгоритмы

Задача 1. Известна схема дорог между несколькими городами. Числа на схеме (рис. 6.20) обозначают расстояния (дороги не прямые, поэтому неравенство треугольника может нарушаться).

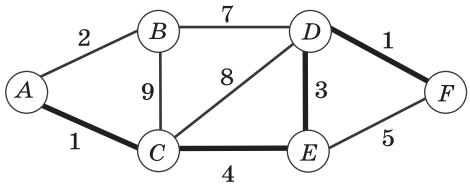


Рис. 6.20

Нужно найти кратчайший маршрут из города A в город F.

Первая мысль, которая приходит в голову: на каждом шаге выбирать кратчайший маршрут до ближайшего города, в котором мы ещё не были. Для заданной схемы на первом этапе едем в город C (длина 1), далее — в E (длина 4), затем в D (длина 3) и наконец в F (длина 1) — жирные линии на рис. 6.20. Общая длина маршрута равна 9.

Алгоритм, который мы применили, называется **«жадным»**. Он состоит в том, чтобы на каждом шаге многоходового процесса выбирать наилучший в данный момент вариант, не думая о том, что впоследствии этот выбор может привести к худшему решению.

Для данной схемы жадный алгоритм на самом деле даёт оптимальное решение, но так будет далеко не всегда. Например, для той же задачи с другой схемой, показанной на рис. 6.21, жадный алгоритм выберет маршрут A–B–D–F длиной 10, хотя существует более короткий маршрут A–C–E–F длиной 7.

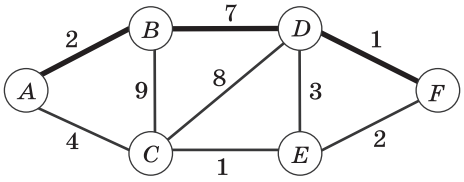


Рис. 6.21

«Жадный» алгоритм не всегда позволяет получить оптимальное решение.

Однако есть задачи, в которых жадный алгоритм всегда приводит к правильному решению. Одна из таких задач (её называют **задачей Прима–Крускала** в честь Р. Прима и Д. Крускала, которые независимо предложили её в середине XX века) формулируется так.

Задача 2. В стране Лимонии есть N городов, которые нужно соединить линиями связи. Между какими городами нужно проложить линии, чтобы все города были связаны в одну систему и общая длина линий связи была наименьшей?

В теории графов эта задача называется задачей построения **минимального остовного дерева** (т. е. дерева, связывающего все вершины). Остовное дерево для связного графа с N вершинами имеем $N - 1$ ребро.

Рассмотрим «жадный» алгоритм решения этой задачи, предложенный Крускалом:

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

На рисунке 6.22 показано минимальное остовное дерево для одного из рассмотренных выше графов (сплошные жирные линии).

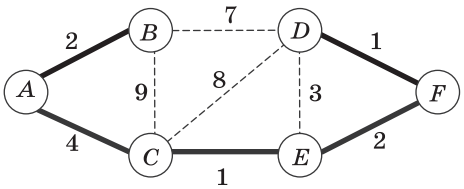


Рис. 6.22

Здесь возможна такая последовательность добавления рёбер: CE, DF, AB, EF, 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, т. е. все рёбра из графа удаляются. Таким образом, мы получаем

множество элементарных деревьев (так называемый **лес**), каждое из которых состоит из одной вершины. Затем последовательно соединяем 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, т. е. все они принадлежат одному остовному дереву. Можно доказать, что это дерево будет иметь минимальный вес, если на каждом шаге выбирать подходящее ребро минимальной длины.

В программе сначала присвоим всем вершинам разные числовые коды:

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

Здесь `N` — количество вершин, а `col` — «цвета» вершин (список из `N` элементов).

Затем в цикле `N – 1` раз (именно столько рёбер нужно включить в дерево) выполняем следующие операции:

1) ищем ребро минимальной длины среди всех рёбер, *концы которых окрашены в разные цвета*;
2) найденное ребро в виде кортежа `(iMin, jMin)` добавляется в список выбранных, и все вершины, имеющие цвет `col[jMin]`, перекрашиваются в цвет `col[iMin]`.

Напишем полную программу на языке Python. Для описания графа используем весовую матрицу `W` размера `N × N` (индексы строк и столбцов начинаются с 0). Если связи между вершинами `i` и `j` нет, в элементе `W[i][j]` матрицы будем хранить «бесконечность» — число, намного большее, чем длина любого ребра. Для графа на рис. 6.22 она может быть задана следующим образом:

```
N = 6
INF = 30000   # "бесконечность", нет связи
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 список ostov будем записывать выбранные рёбра — каждое ребро хранится как кортеж из номеров двух вершин, которые оно соединяет.

Для поиска ребра с минимальной длиной используем переменную minDist, её начальное значение должно быть больше, чем INF. Приведём полностью основной цикл программы:

```
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

На примере задачи выбора кратчайшего маршрута (см. задачу 1) мы увидели, что в ней «жадный» алгоритм не всегда даёт правильное решение. В 1960 году Эдсгер Дейкстра предложил алгоритм, позволяющий найти все кратчайшие расстояния от одной вершины графа до всех остальных и соответствующие им маршруты. Предполагается, что длины всех рёбер (расстояния между вершинами) положительные, это справедливо 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. 6.23).

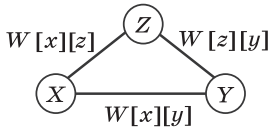


Рис. 6.23

Рассмотрим уже знакомую схему (рис. 6.24), для которой не сработал «жадный» алгоритм.

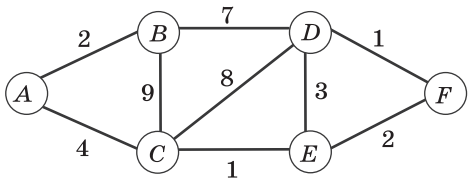


Рис. 6.24

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. Отмечаем стартовую вершину A, расстояние до которой, очевидно, равно 0. Из этой вершины можно попасть напрямую в вершины B и C, т. е. мы уже знаем, что есть маршрут из A и B длиной 2 и маршрут из A в C длиной 4; записываем эти числа около вершин B и C (рис. 6.25). Около остальных вершин (D, E и F) записываем знак ∞ («бесконечность»): они недостижимы напрямую из A.

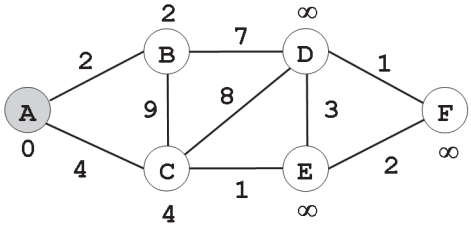


Рис. 6.25

Теперь из всех «непокрашенных» вершин выбираем такую, расстояние до которой от вершины A минимально: в нашем

примере это вершина B (расстояние от A до B равно 2). Поскольку расстояние от A до остальных вершин не меньше 2 и длины всех ребёр 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 вершину B, длина маршрута 2 уже не изменится (рис. 6.26).

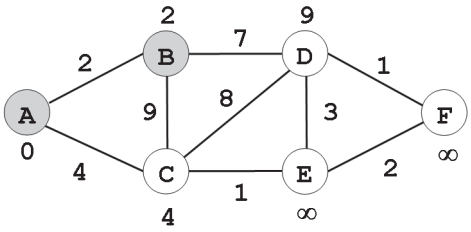


Рис. 6.26

Далее 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 длина равна бесконечности), а теперь найден маршрут длиной $2 + 7 = 9$ (ABD) — см. рис. 6.26.

Следующей по такому же принципу отмечается вершина C. Это позволяет найти маршрут из A в E длиной 5 (ACE) — рис. 6.27.

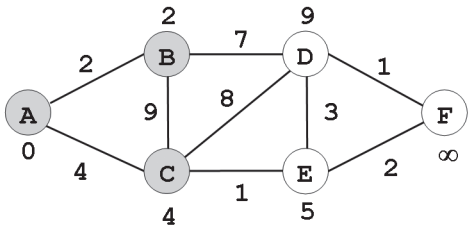


Рис. 6.27

Затем отмечаем 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. 6.28.

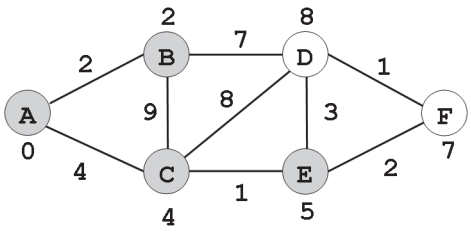


Рис. 6.28

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

В программе нам будут нужны два вспомогательных массива. В одном (назовём его `selected`) для каждой вершины будем хранить логическое значение: `True` («истина»), если она «покрашена» — кратчайший маршрут до неё уже определён, и `False` («ложь»), если не «покрашена». Во второй массив с именем `dist` будем записывать длины кратчайших найденных маршрутов для всех вершин. Сначала все вершины не выбраны и все маршруты неизвестны:

```
INF = 30000
selected = [False]*N
dist = [INF]*N
```

Выбираем стартовую вершину, 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
for j in range(N):
  if not selected[j] and dist[j] < minDist:
    minDist = dist[j]
    V = j
```

Приведём программу полностью:

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

После выполнения программы в массиве `dist` записаны длины оптимальных (кратчайших) маршрутов из вершины A во все вершины.

Остаётся только найти сами оптимальные маршруты (последовательность вершин). Эту задачу можно решить «обратным ходом». Например, найдём оптимальный маршрут из вершины A в вершину F. Как мы видели, его длина равна 7 (см. рис. 6.28).

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 A мы попадаем именно из E. Таким образом, нужно перебрать все рёбра, связывающие вершину F с остальными вершинами, 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)
```

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

Алгорим Флойда–Уоршелла

Теперь рассмотрим более общую задачу: найти все кратчайшие маршруты из любой 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, но существует более красивый метод — **алгоритм Флойда–Уоршелла**, основанный на той же самой идее сокращения маршрута: иногда бывает короче ехать через промежуточные вершины, чем напрямую.

На языке Python этот алгоритм записывается так:

```
for k in range(N):
  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]
```

Фактически на каждом шаге цикла по переменной `k` мы строим матрицу `W`, в которой элемент `W[i][j]` равен длине кратчайшего пути из вершины `i` в вершину `j` при использовании вершин от 1 до `k` в качестве промежуточных. Перед началом цикла (при `k = 0`) эта матрица совпадает с весовой матрицей, учитываются только прямые пути из вершины в вершину.

При подключении новой вершины `k` возможно два варианта: оптимальный маршрут из вершины `i` в вершину `j` либо проходит через вершину `k`, либо не проходит. Фактически нам нужно сравнить длины оптимального маршрута «без вершины `k`» (с использованием только промежуточных вершин от 1 до `k – 1`) и «с вершиной `k`». В первом случае длина маршрута равна `W[i][j]` — это то значение, которое оказалось в матрице `W` после предыдущего шага. Во втором — длина кратчайшего маршрута, проходящего через вершину `k`, вычисляется как сумма кратчайших маршрутов из вершины `i` в вершину `k` и из вершины `k` в вершину `j`. Из двух этих значений алгоритм выбирает минимальное и сохраняет в матрице `W` (рис. 6.29).

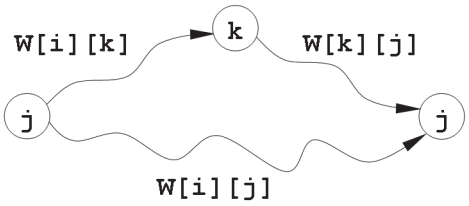


Рис. 6.29

В результате исходная весовая матрица графа W размером $N \times N$ превращается в матрицу, хранящую длины оптимальных маршрутов. Подробное описание и реализацию этого алгоритма вы можете найти в литературе или в Интернете.

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

Информацию о графе можно хранить в памяти не только в виде матрицы смежности и весовой матрицы, но и в виде **списков смежности**. Список смежности для какой-то вершины — это множество вершин, с которыми связана данная вершина. Рассмотрим орграф на рис. 6.30, состоящий из 5 вершин.

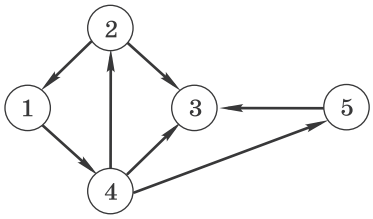


Рис. 6.30

Для него списки смежности (с учётом направлений рёбер) выглядят так:

вершина 1: (4)
вершина 2: (1, 3)
вершина 3: ()
вершина 4: (2, 3, 5)
вершина 5: (3)

Граф, показанный на рисунке выше, описывается в виде вложенного списка так:

```
graph = [ [],       # фиктивный элемент
          [4],      # список смежности для вершины 1
          [1,3],    # ... для вершины 2
          [],       # ... для вершины 3
          [2,3,5],  # ... для вершины 4
          [3] ]     # ... для вершины 5
```

Для того чтобы использовать привычную нумерацию вершин с 1, мы добавили фиктивный элемент — пустой список смежности для несуществующей вершины 0.

Используя такое представление, построим функцию pathCount, которая находит количество путей из одной вершины в другую. Алгоритм её работы основан на следующей идее: общее количество путей из вершины X в вершину Y равно сумме количеств путей из X в Y через все остальные вершины. Чтобы избежать циклов (замкнутых путей), при этом нужно учитывать только те вершины, которые ещё не посещались. Эту информацию необходимо где-то запоминать, для этой цели мы будем использовать список посещённых вершин.

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

- описание графа в виде списков смежности для каждой вершины — graph;
- номера начальной и конечной вершин — vStart и vEnd;
- список посещённых вершин — visited.

Тогда основной цикл функции pathCount приобретает вид

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

Вы, конечно, заметили, что функция pathCount получилась *рекурсивной*, т. е. она вызывает сама себя (в цикле). 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**:

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

Приведём полный текст функции:

```
def pathCount (graph, vStart, vEnd, visited = None):
  if vStart == vEnd: return 1
  if visited is None: visited = []
  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
```

и основную программу, которая находит количество путей из вершины 1 в вершину 3:

```
graph = [[], [4], [1,3], [],  [2,3,5], [3]]
print (pathCount (Graph, 1, 3))
```

У этой программы есть два существенных 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 X в конечную вершину, то когда это значение потребуется снова, желательно сразу использовать полученный ранее результат, а не вызывать функцию рекурсивно ещё раз. Попытайтесь улучшить программу самостоятельно так, чтобы исправить эти недостатки.

Некоторые задачи

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

Задача 3. Бродячий 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 пути (или стоимость) 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. 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 заведомо не дадут лучших результатов, чем уже полученные. В реальных ситуациях нередко оказываются полезны приближённые решения, которые не гарантируют точного оптимума, но позволяют получить приемлемый вариант.

Приведём формулировки ещё некоторых задач, которые решаются с помощью теории графов. Алгоритмы их решения вы можете найти в литературе или в Интернете.

Задача 4 (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.

Задача 5. Имеется N населённых пунктов, в каждом из которых живёт p_i школьников ($i = 1, \ldots, N$). Надо разместить школу в одном из них так, чтобы общее расстояние, проходимое всеми учениками по дороге в школу, было минимальным. В каком пункте нужно разместить школу?

Задача 6 (о наибольшем паросочетании). Есть M мужчин и N женщин. Каждый мужчина указывает несколько женщин (от 0 до N), на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от 0 до M), за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.

Выводы

- Граф — это набор вершин и связывающих их рёбер. Информация о графе чаще всего хранится в виде матрицы смежности или весовой матрицы.

- «Жадный» алгоритм — это алгоритм, в котором на каждом шаге многоходового процесса выбирается наилучший в данный момент вариант, без учёта того, что впоследствии этот выбор может привести к худшему итоговому решению. «Жадный» алгоритм не всегда позволяет найти лучшее решение.
- Алгоритм Дейкстры позволяет определить кратчайшие маршруты из одной вершины во все остальные, его асимптотическая сложность $O(N^2)$.
- Алгоритм Флойда находит длины всех кратчайших маршрутов в графе (из каждой вершины во все остальные), его асимптотическая сложность $O(N^3)$.

Нарисуйте в тетради интеллект-карту этого параграфа.

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 орграфа?
4. Что такое «жадный» алгоритм? Всегда ли он позволяет найти лучшее решение?
5. Как, на ваш взгляд, можно было бы ускорить работу алгоритма Крускала с помощью предварительной сортировки рёбер?

Подготовьте сообщение

а) «Задача о кенигсбергских мостах»
б) «Задача коммивояжёра»
в) «"Жадный" алгоритм в задаче коммивояжёра»
г) «Метод ветвей и границ»
д) «Алгоритм Литтла»
е) «Задача о максимальном потоке»
ж) «Задача о наибольшем паросочетании»
з) «Использование графов для анализа данных в Интернете»
и) «Теория графов в практических задачах»

Проекты

а) Лингвистический анализ текстов с помощью графов
б) Программа для решения выбранной задачи с помощью графов

Интересные сайты

algolist.manual.ru — сайт, посвящённый алгоритмам
uchimatchast.ru — онлайн-сервис для решения задач оптимизации
graphonline.ru — онлайн-сервис для работы с графами

§ 41
Динамическое программирование

Ключевые слова:
- динамическое программирование
- перебор вариантов
- рекуррентная формула

Что такое динамическое программирование?

Мы уже сталкивались с последовательностью чисел Фибоначчи (см. учебник для 10 класса):

$$F_1 = F_2 = 1;\ F_n = F_{n-1} + F_{n-2} \text{ при } n > 2.$$

Для их вычисления можно использовать рекурсивную функцию:

```
def Fib (n):
  if n < 3: return 1
  return Fib(n-1) + Fib(n-2)
```

Каждое из этих чисел связано с предыдущими, вычисление F_5 приводит к рекурсивным вызовам, которые показаны на рис. 6.31.

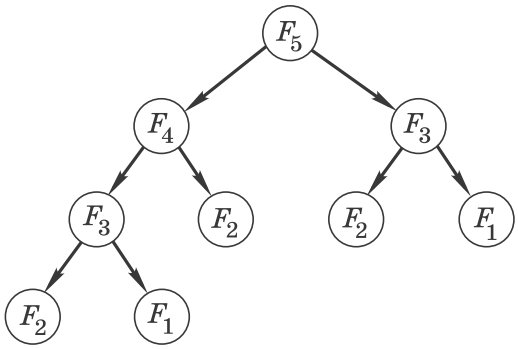


Рис. 6.31

Таким образом, мы два раза вычислили F_3, три раза — F_2 и два раза — F_1. Рекурсивное решение очень простое, но оно неоптимально по быстродействию: компьютер выполняет лишнюю работу, повторно вычисляя уже найденные ранее значения.

В чём же выход? Напрашивается такое решение — для того чтобы быстрее найти F_N, будем хранить все предыдущие числа Фибоначчи в массиве. Пусть этот массив называется F, сначала заполним его единицами:

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

В этой задаче нам удобно применить нумерацию, начиная с 1, так что элемент массива `F[0]` использоваться не будет. Поэтому размер созданного списка на 1 больше, чем N. Для вычисления всех чисел Фибоначчи от F_1 до F_N можно использовать цикл:

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

Динамическое программирование — это способ решения сложных задач путём сведения их к более простым задачам того же типа (задачам меньшей размерности).

Обычно первые задачи-элементы такой последовательности имеют очень простое решение, а последний элемент — это исходная задача. Каждая задача этой последовательности может быть решена с использованием решений подзадач меньшей размерности.

Такой подход впервые систематически применил американский математик Р. Беллман при решении сложных многошаговых задач оптимизации. Его идея состояла в том, что оптимальная последовательность шагов оптимальна на любом участке. Например, пусть нужно перейти из пункта A в пункт E через один из пунктов B, C или D (числами обозначены «стоимости» маршрутов) — рис. 6.32.

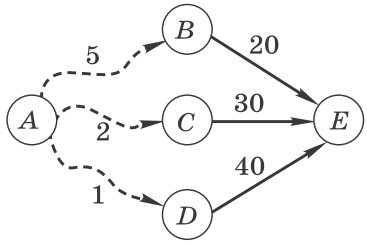


Рис. 6.32

Пусть уже известны оптимальные маршруты из пунктов B, C и D в пункт E (они обозначены сплошными линиями) и их «стоимости». Тогда для нахождения оптимального маршрута из A в E нужно выбрать вариант, который даст минимальную стоимость по сумме двух шагов. В данном случае это маршрут A–B–E, стоимость которого равна 25. Как видим, такие задачи решаются «с конца», т. е. решение начинается от конечного пункта.

В информатике динамическое программирование часто сводится к тому, что мы храним в памяти решения всех задач меньшей размерности. За счёт этого удаётся ускорить выполнение программы. Например, на одном и том же компьютере вычисление F_{35} в программе на Python с помощью рекурсивной функции требует около 58 секунд, а с использованием массива — менее 0,001 с.

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

```
f1 = 1
f2 = 1
for i in range(3, N + 1):
  f2, f1 = f1 + f2, f2
```

Ответ всегда будет находиться в переменной `f2`.

Задача 1. Найти количество K_N цепочек, состоящих из N нулей и единиц, в которых нет двух стоящих подряд единиц.

При больших N решение задачи методом перебора потребует огромного времени вычислений. Для того чтобы использовать метод динамического программирования, нужно:

1) выразить K_N через предыдущие значения последовательности $K_1, K_2, ..., K_{N-1}$;
2) выделить массив для хранения всех предыдущих значений $K_i (i = 1, ... N-1)$.

Самое главное — вывести **рекуррентную формулу**, выражающую K_N через решения аналогичных задач меньшей размерности. Рассмотрим цепочку из N бит, последний элемент которой — 0 (рис. 6.33).

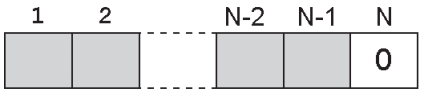


Рис. 6.33

Поскольку дополнительный 0 не может привести к появлению двух соседних единиц, подходящих последовательностей длиной N с нулём в конце существует столько, сколько подходящих последовательностей длины $N-1$, т. е. K_{N-1}. Если же последний символ — это 1, то слева от него обязательно должен быть 0, а начальная цепочка из $N-2$ бит должна быть правильной, т. е. не должна содержать двух стоящих подряд единиц (рис. 6.34). Поэтому подходящих последовательностей длина N с единицей в конце существует столько, сколько подходящих последовательностей длиной $N-2$, т. е. K_{N-2}.

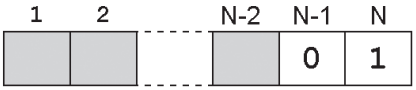


Рис. 6.34

В результате получаем: $K_N = K_{N-1} + K_{N-2}$. Значит, для вычисления очередного числа нам нужно знать два предыдущих.

Теперь рассмотрим простые случаи. Очевидно, что есть две правильные последовательности длиной 1 (0 и 1), т. е. $K_1 = 2$. Далее, есть три подходящие последовательности длины 2 (00, 01 и 10), поэтому $K_2 = 3$. Легко понять, что решение нашей задачи — число Фибоначчи: $K_N = F_{N+2}$.

Поиск оптимального решения

Задача 2. 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 перебором вариантов. Наша задача осложняется тем, что требуется написать программу, которая решает задачу для любого введённого числа N.

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 $(5 + 5)$.

Как и в любом решении, использующем динамическое программирование, главная проблема — составить рекуррентную формулу. Сначала получим оптимальное число бидонов K_N, а потом

подумаем, как определить, какие именно бидоны нужно использовать.

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 выбрать минимальное значение, получаем:

$$K_N = 1 + \min(K_{N-1}, K_{N-5}, K_{N-6}).$$

В качестве начального значения берём $K_0 = 0$.

Полученная формула применима при $N \geq 6$. Для меньших N используются только те данные, которые есть в таблице (рис. 6.35). Например:

$$K_3 = 1 + K_2 = 3, \quad K_5 = 1 + \min(K_4, K_0) = 1.$$

На рисунке 6.35 показан заполненный массив K для $N = 10$.

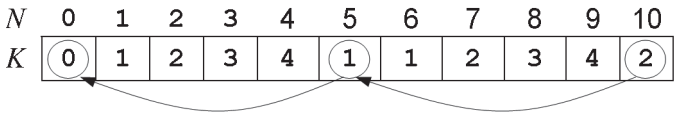


Рис. 6.35

Как теперь по массиву K определить оптимальный состав бидонов? Пусть, например, $N = 10$. По таблице определяем, что $K_{10} = 2$, т. е. требуются два бидона. На последнем шаге мы могли добавить, например, бидон объёмом 1 л, тогда было бы $K_{10-1} = K_9 = 2 - 1 = 1$, но это не так. Проверяем вариант, когда добавляется бидон объёмом 5 л: $K_{10-5} = K_5 = 1$, т. е. этот вариант подходит. Далее таким же способом определяем, что первый бидон тоже имеет объём 5 л (см. рис. 6.35).

Можно заметить, что такая процедура очень похожа на алгоритм Дейкстры, и это не случайно. В алгоритмах Дейкстры и Флойда–Уоршелла, по сути, используется метод динамического программирования.

Задача 3 (задача о куче). Из камней весом p_i (i = 1, …, N) требуется набрать кучу весом ровно W или, если это невозможно, максимально близкую к W (но меньшую, чем W). Все веса камней и значение W — неотрицательные 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 вариантов равно 2^N, и при больших N полный перебор практически невыполним.

Динамическое программирование позволяет найти решение задачи значительно быстрее. Идея состоит в том, чтобы сохранять в массиве решения всех более простых задач этого типа (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 подсчитать вручную. Рассмотрим пример, когда требуется набрать вес 8 единиц из камней весом 2, 4, 5 и 7 единиц (рис. 6.36).

	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								

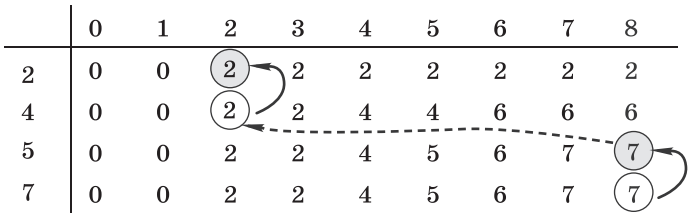
Рис. 6.36

Теперь предположим, что строки с 1-й по $(i-1)$-ю уже заполнены. Перейдем 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 $T[i][w] = T[i-1][w]$, т. е. 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 решить, брать или не брать камень? Надо проверить, в каком случае полученный вес будет больше (ближе к w). Таким образом, получается рекуррентная формула для заполнения таблицы:

$$T[i][w] = \begin{cases} T[i-1][w], & \text{при } w < p_i; \\ \max(T[i-1][w],\ T[i-1][w-p_i] + p_i), & \text{при } w \geq p_i. \end{cases}$$

Используя эту формулу, заполняем таблицу по строкам, сверху вниз; в каждой строке — слева направо (рис. 6.37).



	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. 6.37

Видим, что сумму 8 набрать невозможно, ближайшее значение — 7 (правый нижний угол таблицы).

Эта таблица содержит все необходимые данные для определения выбранной группы камней. Действительно, если камень с весом p_i не включён в набор, то $T[i][w] = T[i - 1][w]$, т. е. число в таблице не меняется при переходе на строку вверх. 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 вверх, 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.

Как мы уже отмечали, количество вариантов в задаче для N камней равно 2^N, т. е. алгоритм полного перебора имеет асимптотическую сложность $O(2^N)$. В данном алгоритме количество операций равно числу элементов таблицы, т. е. сложность нашего алгоритма — $O(N \cdot W)$. Однако нельзя сказать, что он имеет линейную сложность, так как есть ещё сильная зависимость от заданного веса, W. Такие алгоритмы называют *псевдополиномиальными* («как бы полиномиальными»). В них ускорение вычислений достигается за счёт использования дополнительной памяти для хранения промежуточных результатов.

Количество решений

Задача 4. У исполнителя Утроитель две команды, которым присвоены номера:

```
1. прибавь 1
2. умножь на 3
```

Первая из них увеличивает число на экране на 1, вторая — утраивает его. Программа для Утроителя — это последовательность

команд. Сколько есть программ, которые число 1 преобразуют в число $N = 20$?

Заметим, что при выполнении любой из команд число увеличивается (не может уменьшаться). Начнём с простых случаев, с которых будем начинать вычисления. Понятно, что для начального значения $N = 1$ существует только одна программа — пустая, не содержащая ни одной команды. Для числа $N = 2$ тоже есть только одна программа, состоящая из команды сложения. Если через K_N обозначить количество разных программ для получения числа N из 1, то $K_1 = K_2 = 1$.

Теперь рассмотрим общий случай, чтобы построить рекуррентную формулу, связывающую K_N с предыдущими элементами последовательности $K_1, K_2, \ldots, K_{N-1}$, т. е. с решениями таких же задач для меньших N.

Если число N не делится на 3, то последней командой для его получения может быть только операция сложения, поэтому $K_N = K_{N-1}$. Если N делится на 3, то последней командой может быть как сложение, так и умножение. Поэтому нужно сложить K_{N-1} (количество программ с последней командой сложения) и $K_{N/3}$ (количество программ с последней командой умножения). В итоге получаем:

$$K_N = \begin{cases} K_{N-1}, & \text{если } N \text{ не делится на 3;} \\ K_{N-1} + K_{N/3}, & \text{если } N \text{ делится на 3.} \end{cases}$$

Остаётся заполнить таблицу для всех значений от 1 до заданного $N = 20$. Для небольших значений N эту задачу легко решить вручную — рис. 6.38.

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
K_N	1	1	2	2	2	3	3	3	5	5	5	7	7	7	9	9	9	12	12	12

Рис. 6.38

Заметим, что количество вариантов меняется только в тех столбцах, где N делится на 3, поэтому из всей таблицы можно оставить только эти столбцы (и первый), добавляя следующее значение, кратное трём — рис. 6.39.

N	1	3	6	9	12	15	18	21
K_N	1	2	3	5	7	9	12	15

Рис. 6.39

Заданное число 20 попадает в последний интервал (от 18 до 20), поэтому ответ в данной задаче — 12.

При составлении программы с полной таблицей нужно выделить в памяти целочисленный массив K, индексы которого изменяются от 0 до N + 1, и заполнить его по приведённым выше формулам:

```
K = [1]*(N+1)
for i in range(2, N+1):
  K[i] = K[i-1]
  if i % 3 == 0:
    K[i] += K[i//3]
```

Ответом будет значение K[N].

Задача 5 (Размен монет). Сколькими различными способами можно выдать сдачу размером W рублей, если есть монеты достоинством p_i (i = 1, …, N)? Для того чтобы сдачу всегда можно было выдать, будем предполагать, что в наборе есть монета достоинством 1 рубль (p_1 = 1).

Это задача, так же как и задача о куче, решается полным перебором вариантов, число которых при больших N очень велико. Будем использовать динамическое программирование, сохраняя в массиве решения всех задач меньшей размерности (для меньших значений N и W).

В матрице T значение $T[i][w]$ будет обозначать количество вариантов сдачи размером w рублей (w изменяется от 0 до W) при использовании первых i монет из набора. Очевидно, что при нулевой сдаче есть только один вариант (не дать ни одной монеты), так же и при наличии только одного типа монет (напомним, что $p_1 = 1$) есть тоже только один вариант. 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. 6.40).

	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. 6.40

Таким образом, мы определили простые базовые случаи, от которых «отталкивается» рекуррентная формула.

Теперь рассмотрим общий случай. Заполнять таблицу будем по строкам, слева направо. 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 больше p_i, то к этому значению нужно добавить количество вариантов с «участием» новой монеты. Если монета достоинством p_i использована, то нужно учесть все варианты «разложения» остатка $w - p_i$ на все доступные монеты, т. е. $T[i][w] = T[i-1][w] + T[i][w-p_i]$. В итоге получается рекуррентная формула

$$T[i][w] = \begin{cases} T[i-1][w], & \text{при } w < p_i; \\ T[i-1][w] + T[i][w-p_i], & \text{при } w \geq p_i. \end{cases}$$

которая используется для заполнения таблицы (рис. 6.41).

	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. 6.41

Ответ к задаче находится в правом нижнем углу таблицы.

Вы могли заметить, что решение этой задачи очень похоже на решение задачи о куче камней. Это не случайно, две эти задачи относятся к классу сложных задач, для решения которых известны только переборные алгоритмы. Использование методов динамического программирования позволяет ускорить решение за счёт хранения промежуточных результатов, однако требует дополнительного расхода памяти.

Выводы

- Динамическое программирование — это метод, позволяющий ускорить решение задачи за счёт хранения решений более простых задач того же типа.

- Для использования метода динамического программирования нужно вывести рекуррентную формулу, связывающее решение исходной задачи с решением подобных задач меньшей размерности, и определить простые базовые случаи.

Нарисуйте в тетради интеллект-карту этого параграфа.

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 удаётся ускорить решение сложных задач методом динамического программирования?
4. Какие ограничения есть у метода динамического программирования?
5. Сравните метод динамического программирования и рекурсивные решения.

Подготовьте сообщение

а) «Задача о рюкзаке»
б) «Задачи на подпоследовательности»
в) «Поиск оптимального маршрута»
г) «Динамическое программирование в задачах биоинформатики»

Проекты

а) Определение количества решений систем логических уравнений методом отображений
б) Программа для решения задач с исполнителем Калькулятор
в) Программа для решения выбранной задачи с помощью динамического программирования

ЭОР к главе 6 на сайте ФЦИОР (http://fcior.edu.ru)

- Решето Эратосфена
- Основные структуры данных
- Сущность модульного программирования. Программный модуль
- Работа с указателями и структурами (на примере языка Pascal)
- Линейные структуры данных. Список, стек, очередь
- Организация и работа со стеком

- Организация и работа с очередью
- Основы теории графов. Способы представления графов. Обход графа
- Задача о кратчайших путях. Алгоритм Флойда, Дейкстры
- Задачи оптимизации. Динамическое программирование

www

Практические работы к главе 6

Работа № 39 «Решето Эратосфена»
Работа № 40 «Длинные числа»
Работа № 41 «Структуры»
Работа № 42 «Словари»
Работа № 43 «Алфавитно-частотный словарь»
Работа № 44 «Вычисление арифметических выражений»
Работа № 45 «Скобочные выражения»
Работа № 46 «Очереди»
Работа № 47 «Заливка области»
Работа № 48 «Обход дерева»
Работа № 49 «Вычисление арифметических выражений»
Работа № 50 «Хранение двоичного дерева в массиве»
Работа № 51 «Задача Прима–Крускала»
Работа № 52 «Алгоритм Дейкстры»
Работа № 53 «Алгоритм Флойда–Уоршелла»
Работа № 54 «Числа Фибоначчи»
Работа № 55 «Задача о куче»