

Глава 6. Алгоритмизация и программирование

Язык Python

Авторы благодарят В.М. Гуровица и
С.С. Михалковича за внимательное чтение этого
материала и полезные критические замечания.

§ 38. Целочисленные алгоритмы

Во многих задачах все исходные данные и необходимые результаты – целые числа. При этом всегда желательно, чтобы все промежуточные вычисления тоже проводились с только 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 переполнение разрядной сетки).

Решето Эратосфена

Во многих прикладных задачах, например, при шифровании с помощью алгоритма RSA, используются простые числа (вспомните материал учебника для 10 класса). 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 математик 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 не вычеркнутое 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 вычеркиваем все чётные числа:

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

Далее вычеркиваем все числа, кратные 3:

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

А 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 вычеркивании чисел, кратных трем, нам не пришлось вычеркивать число 6, так как оно уже было вычеркнуто. Кроме того, все числа, кратные 5 и 7, к последнему шагу тоже оказались вычеркнуты.

Предположим, что мы хотим 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 не вычеркнутое число и присвоить его переменной k ;
- 5) повторять шаги 3 и 4, пока $k^2 \leq N$.

Чтобы составить программу, нужно определить, что значит «выписать все числа» и «вычеркнуть число». Один из возможных вариантов хранения данных – массив логических величин с индексами от 2 до N . Поскольку индексы элементов списков в Python всегда начинаются с нуля, для того, чтобы работать с нужным диапазоном индексов, необходимо выделить логический 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 $k \leq \sqrt{N}$ на равносильное условие $k^2 \leq N$, в котором используются только целые числа.

После завершения этого цикла не вычеркнутыми 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 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 числа¹ 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:

¹ Здесь нужно рассмотреть пограничный случай, когда N – квадрат целого числа, то есть 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.

```
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 сформирует 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+1) 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, то есть размер числа (точнее, отведённый на него объём памяти) автоматически расширяется при необходимости. Однако в других языках (C, C++, Паскаль) для целых чисел отводятся ячейки фиксированных размеров (обычно до 64 бит). Поэтому остро стоит 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 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
A	1	2	3	4	5	6	7	8

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:

	7	6	5	4	3	2	1	0
A	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 ее 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, то есть использовать систему счисления с основанием 1 000 000 000. Однако нужно учитывать, что с такими числами будут вы-

полняться арифметические операции, результат которых должен «помещаться» в такую же ячейку памяти. Например, если надо умножить разряды этого числа число на $k < 100$, и в языке программирования нет 64-битных целочисленных типов данных, можно хранить в ячейке не более 7 разрядов.

Задача 1. Требуется вычислить точно значение факториала $100! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot 99 \cdot 100$ 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

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

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 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 массива 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, остается научиться умножать длинное число на «короткое» ($k \leq 100$). «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, то есть используется система счисления с основанием $d = 1\,000\,000$. Тогда число $\{A\} = 12345678901734567$ хранится в трех 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 перейдет 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 остается 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!, остается 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, с использованием функции **factorial** из модуля **math**:

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

Квадратный корень

Рассмотрим еще 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 число². Поэтому в том случае, когда нужно именно целое значение, приходится использовать специальные методы.

Один из самых известных алгоритмов вычисления квадратного корня, известный ещё в Древней Греции, – метод Герона Александрийского, который сводится к многократному применению формулы³

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

Здесь **a** – число, из которого извлекается корень, а x_{i-1} и x_i – предыдущее и следующее приближения (см. главу 9 из учебника для 10 класса). Фактически здесь вычисляется среднее арифметическое между x_{i-1} и a/x_{i-1} . Пусть одно из этих значений меньше, чем \sqrt{a} , тогда второе обязательно больше, чем \sqrt{a} . Поэтому их среднее арифметическое с каждым шагом приближается к значению корня.

² Заметим, что возведение целого числа в степень 0,5 тоже даёт вещественное число.

³ Фактически эта формула – результат применения метода Ньютона для решения нелинейных уравнений к уравнению $x^2 = a$.

Метод Герона «сходится» (то есть, 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 (то есть *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 оказывается *больше или равно* предыдущему, мы нашли квадратный корень.



Контрольные вопросы

1. Какие преимущества и недостатки имеет алгоритм «решето Эратосфена» в сравнении с проверкой каждого числа на простоту?
2. Что такое «длинные числа» и «длинная арифметика»?
3. В каких случаях необходимо применять «длинную арифметику»?
4. Какое максимальное число можно записать в ячейку размером 64 бита? Рассмотрите варианты хранения чисел со знаком и без знака.
5. Можно ли использовать для хранения длинного числа символьную строку? Оцените достоинства и недостатки такого подхода.
6. Почему неудобно хранить длинное число, записывая первую значащую цифру в начало массива?
7. Почему неэкономично хранить по одной цифре в каждом элементе массива?
8. Сколько разрядов десятичной записи числа можно хранить в одной 16-битной ячейке?
9. Объясните, какие проблемы возникают при выводе длинного числа. Как их можно решать?
10. *Предложите способ вывода «длинного» числа без использования возможностей функции **format**.
11. Объясните алгоритм поиска целого квадратного корня из числа с помощью метода Герона.
12. *Предложите алгоритм поиска целого кубического корня из числа, основанный на аналогичной идее.



Задачи

1. Докажите, что если у числа k нет ни одного делителя в диапазоне от 2 до \sqrt{k} , то оно простое.

2. Напишите две программы, которые находят все простые числа в диапазоне от 2 до N двумя разными способами:
 - а) проверкой каждого числа из этого диапазона на простоту;
 - б) используя решето Эратосфена.
 Сравните число шагов цикла (или время работы) этих программ для разных значений N . Постройте для каждого варианта зависимость количества шагов от N , сделайте выводы о сложности алгоритмов.
3. Без использования программы определите, сколько нулей стоит в конце числа 100!
4. Соберите всю программу и вычислите 100!. Сколько цифр входит в это число?
5. Напишите процедуру для ввода длинных чисел из файла в массив (список).
6. Напишите процедуры для сложения и вычитания длинных чисел (не используя «длинную арифметику» Python).
7. *Напишите процедуры для умножения и деления длинных чисел (не используя «длинную арифметику» Python).
8. Напишите полную программу для извлечения целого квадратного корня из целого числа. Проверьте, как она будет работать, если на вход функции `isqrt` подать нецелое значение.

§ 39. Структуры

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 экземпляров и т.д. Как хранить эти данные?

Поскольку книг много, нужен массив. Но информация о книгах разнородна, она содержит целые числа и символьные строки разной длины. Конечно, можно разбить эти данные на несколько массивов (массив авторов, массив названий и т.д.), так чтобы i -ый элемент каждого массива относился к книге с номером i . Но такой подход оказывается слишком неудобен и ненадежен. 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.

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 только⁴

- 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** (англ. класс) и располагаются выше блока объявления переменных. Имя нового типа – **TBook** – это удобное сокращение от английских слов *Type Book* (*тип книга*), хотя можно было использовать и любое другое имя, составленное по правилам языка программирования.

⁴ Конечно, в реальной ситуации данных больше, но принцип не меняется.

Слово **pass** (англ. 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 сообщения 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 () )
```

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 **Book**, 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 новые значения:

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

использовать при обработке данных:

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

и выводить на экран:

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

Работа с файлами

В программах, которые работают с данными на диске, 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 строчку и разделяя их каким-то символом-разделителем, который не встречается внутри самих полей.

Но есть более грамотный способ, который позволяет хранить данные в файлах во внутреннем формате, то есть 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, то есть 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 – это ошибочная (аварийная) ситуация, в данном случае – неудачное чтение структуры из файла. «Опасные» операторы, которые могут вызвать ошибку, записываются в виде блока (со сдвигом) после слова **try**. После этого в блоке, начинаемся 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 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).

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 отличающиеся 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 учетом 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», то есть 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**, то есть 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." )
```



Контрольные вопросы

1. Что такое структура? В чём её отличие от массива?
2. В каких случаях использование структур дает 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? Расскажите о точечной записи.
5. Что такое двоичный файл? Чем он отличается от текстового?
6. Как можно сортировать структуры?



Задачи

1. Опишите структуру, в которой хранится информация о
 - а) видеозаписи;
 - б) сотруднике фирмы «Рога и Копыта»;
 - в) самолёте;
 - г) породистой собаке.

2. Постройте программу, которая работает с базой данных, хранящейся в виде файла. Ваша СУБД (система управления базой данных) должна иметь следующие возможности:
- а) просмотр записей;
 - б) добавление записей;
 - в) удаление записей;
 - г) сортировка по одному из полей.

§ 40. Словари

Что такое словарь?

Задача 2. 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 списке), а по слову (символьной строке). Например, вывести на экран количество найденных слов «бегемот» можно было бы так:

```
print ( D["бегемот"] )
```

где **D** – имя словаря.

Типы данных для построения словаря есть в некоторых современных языках программирования. В Python для работы со словарями есть встроенный тип данных **dict** (от англ. *dictionary*)⁵.

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

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

Ключом может быть любой неизменяемый тип данных, например, число, символьная строка или *кортеж* (неизменяемый набор значений). В одном словаре можно использовать ключи разных типов.

Алфавитно-частотный словарь

Вернемся 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 }
```

⁵ В других языках программирования используются другие названия, например, «ассоциативный массив» или «хэш».

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, такой оператор 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 создается 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 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 еще и метод **strip** (от 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 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, то есть количество та-

ких 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**.



Контрольные вопросы

1. Что такое словарь? Какие операции он допускает?
2. Можно ли обратиться к элементам словаря по индексам? Как вы думаете, почему сделано именно так?
3. Как создать пустой словарь?
4. Как создать словарь из готовых пар «ключ-значение»? Найдите в литературе или в Интернете разные способы решения этой задачи.
5. Как добавить элемент в словарь?
6. Как обращаться к элементу словаря?
7. Расскажите о методе **get**. Чем его можно заменить?
8. Как получить список всех ключей словаря? всех значений словаря?
9. Как перебрать все пары «ключ-значение»?
10. Зачем при чтении строк из файла используется метод **strip**?



Задачи

1. Постройте полную программу, которая составляет алфавитно-частотный словарь для заданного файла со списком слов.
2. В предыдущей задаче выведите все найденные слова в файл в порядке убывания частоты, то есть в начале списка должны стоять слова, которые встречаются в файле чаще всех.

§ 41. Стек, очередь, дек

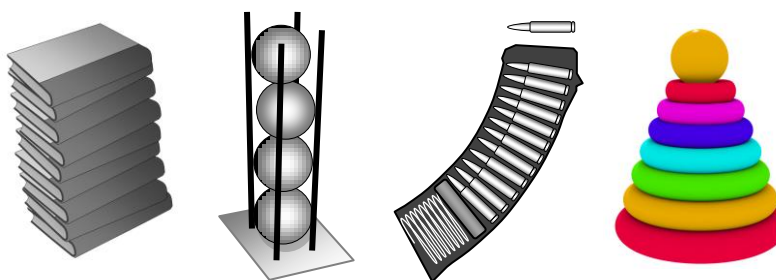
Что такое стек?

Представьте себе стопку книг (подносов, кирпичей и т.п.). С точки зрения информатики её можно воспринимать как список элементов, расположенных в определенном 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 линейный список, в котором элементы добавляются и удаляются только с одного конца («последним пришел – первым ушел»⁶).

На рисунках показаны примеры стеков вокруг нас, в том числе автоматный магазин и детская пирамидка:

⁶ Англ. LIFO = *Last In – First Out*.



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 3. 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* – толкнуть);
- 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 остается «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 снять элемент со стека, используется метод `pop`:

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

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

- 1) удаляет последний элемент списка (если вызывается без параметров⁷);
- 2) возвращает удалённый элемент как результат функции, так что его можно сохранить в какой-либо переменной.

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:

⁷ При вызове метода `pop` в скобках можно указать индекс удаляемого элемента.


```

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.

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 $/ + 5 15 - + 4 7 1$. Скобок здесь не требуется, так как порядок операций строго определен: сначала выполняются два сложения ($+ 5 15$ и $+ 4 7$), затем вычитание, и, наконец, деление. Первой стоит последняя операция.

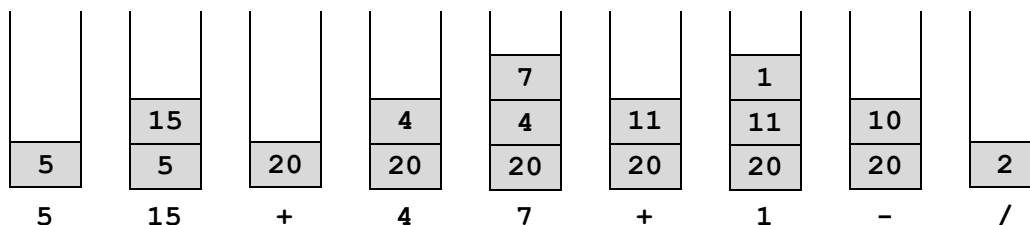
В середине 1950-х годов была предложена *обратная польская нотация* или *постфиксная* форма записи, в которой знак операции стоит *после* операндов:

$5 15 + 4 7 + 1 - /$

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.

Покажем, как работает этот алгоритм (стек «растёт» снизу вверх):



В результате в стеке остается 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:                         # (12)
        stack.append ( x )

```

```
print ( stack[0] ) # (13)
```

В строке программы 1 результат ввода разбивается на части по пробелам с помощью метода **split**, в результате получается список **data**, содержащий отдельные элементы постфиксной записи – числа и знаки арифметических действий. В строке 2 создаём пустой стек, в строке 3 в цикле перебираем все элементы списка. Если очередной элемент, попавший в переменную **x** – это знак арифметической операции (строка 4), снимаем со стека два верхних элемента (строки 5-6), выполняем нужное действие (строки 7-10) и добавляем результат вычисления в стек (строка 11). Если же очередной элемент – это число (не знак операции), просто добавляем его в стек (строка 12). В конце программы в стеке должен остаться единственный элемент (результат), который выводится на экран (строка 13).

Скобочные выражения

Задача 4. 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

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

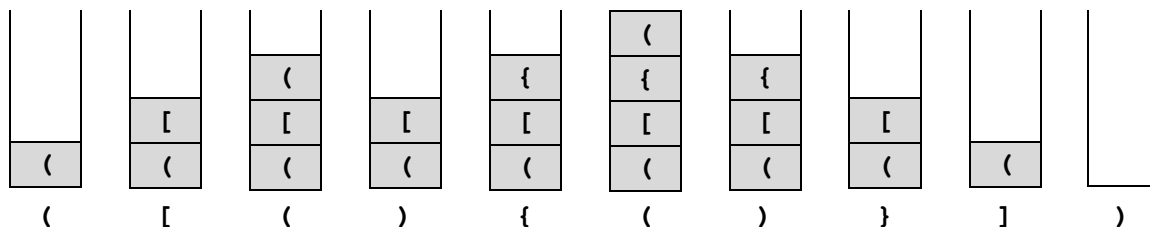
неправильные. В первых трёх есть непарные скобки, а в последних двух не соблюдается вложенность скобок.

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, нужно втолкнуть её на вершину стека. Если это закрывающая скобка, то проверяем, что лежит на вершине стека: если там соответствующая открывающая скобка, то её нужно просто снять со стека. Если стек пуст или на вершине лежит открывающая скобка другого типа, 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 рисунке (для правильного выражения):



Введём строки **L** и **R**, которые содержат все виды открывающих и соответствующих закрывающих скобок:

```
L = " ( [ { "
R = " ) ] } "
```

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 p in L:                               # (2)
        stack.append(c)                     # (3)
    p = R.find(c)                            # (4)
    if p >= 0:                                # (5)
        if not stack: err = True            # (6)
    else:                                     # (7)
        top = stack.pop()                   # (8)
        if p != L.find(top):                # (9)
            err = True                      # (10)
    if err: break                           # (11)
```

Сначала мы ищем очередной символ строки **s** (который попал в переменную **c**) в строке **L**, то есть среди открывающих скобок (строка программы 2). Если это действительно открывающая скобка, вталкиваем ее в стек (строка 3).

Далее ищем символ среди закрывающих скобок (строка 4). Если нашли, то в первую очередь проверяем, не пуст ли стек. Если стек пуст, выражение неверное и переменная **err** принимает истинное значение (строка 6).

Если в стеке что-то есть, снимаем символ с вершины стека в переменную **top** (строка 8). В строке (9) сравнивается тип (номер) закрывающей скобки **p** и номер открывающей скобки, найденной на вершине стека. Если они не совпадают, выражение неправильное, и в переменную **err** записывается значение **True** (строка 10).

Если при обработке текущего символа обнаружено, что выражение неверное (переменная **err** установлена в **True**), нужно закончить цикл досрочно с помощью оператора **break** (строка 11).

После окончания цикла нужно проверить содержимое стека: если он не пуст, то в выражении есть незакрытые скобки, и оно ошибочно:

```
if len(stack) > 0: err = True
```

В конце программы остается 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 линейный список, для которого 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 жестких дисков формируют очереди запросов ввода и вывода данных. В сетевых маршрутизаторах создается очередь из пакетов данных, ожидающих отправки.

Задача 5. Рисунок задан в виде матрицы **A**, в которой элемент **A[y][x]** определяет цвет пикселя на пересечении строки **y** и столбца **x**. Перекрасить в цвет 2 одноцветную область, начиная с пикселя (x_0, y_0) . На рисунке показан результат такой заливки для матрицы из 5 строк и 5 столбцов с начальной точкой (1,0).

	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

(1, 0) →

	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

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 начальной точки
while очередь не пуста:
    взять из очереди точку (x, y)
    if A[y][x] == color:
        A[y][x] = 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 пометчая 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 – отдельная строка матрицы). Тогда можно определить размеры матрицы так:

```

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

```

Значение **YMAX** – это число строк, а **XMAX** – число столбцов.

Определим также цвет заливки:

```
NEW_COLOR = 2
```

Зададим координаты начальной точки, откуда начинается заливка:

```

x0 = 1
y0 = 0

```

и запомним её цвет в переменной **color**:

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

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 «кортеж» 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 изменить.

Остается написать основной цикл⁸:

```

while len(Q) > 0:
    x, y = Q.pop(0)
    if A[y][x] == color:

```

⁸ Использование списка для моделирования очереди – не самый быстродействующий вариант, потому что удаление и добавление в начало массива выполняются долго. В практических задачах лучше использовать класс **deque** из модуля **collections**.

```

A[y][x] = NEW_COLOR
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) )

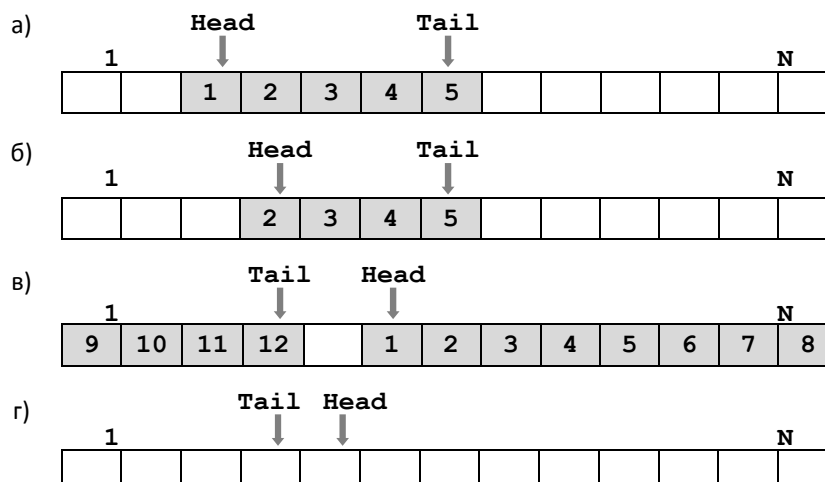
```

Начало очереди всегда совпадает с первым элементом списка (имеющим индекс 0). Цикл в строке 1 работает до тех пор, пока в очереди есть хоть один элемент (её длина больше нуля).

В строке 2 первый элемент удаляется из очереди. Как мы уже говорили, элемент очереди – это кортеж из двух элементов, поэтому мы сразу разбиваем его на отдельные координаты, используя множественное присваивание.

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 а показана очередь из 5 элементов. В этом случае удаление элемента из очереди сводится просто к увеличению переменной **Head** (рисунок б).



При добавлении элемента в конец очереди переменная **Tail** увеличивается на 1. Если она перед этим указывала на последний элемент массива, то следующий элемент записывается в начало массива, а переменной **Tail** присваивается значение 1. Таким образом, массив оказывается замкнутым «в кольцо». На рисунке в показана полностью заполненная очередь, а на рисунке г – пустая очередь. Один элемент массива всегда остается незанятым, иначе невозможно будет различить состояния «очередь пуста» и «очередь заполнена».

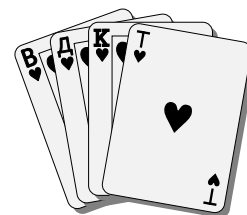
Отметим, что приведенная здесь модель описывает работу кольцевого буфера клавиатуры, который может хранить до 15 двухбайтных слов.

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

Дек (от англ. *deque* = *double ended queue*, двусторонняя очередь) – это линейный список, в котором 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. Для того, чтобы организовать дек в языке Python, также удобно использовать список. При этом 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)` (добав-



ляемый элемент будет иметь индекс 0)

- 3) удаление элемента с конца дека: `d.pop()`
- 4) удаление элемента с начала дека: `d.pop(0)`



Контрольные вопросы

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



Задачи

1. Напишите программу, которая «переворачивает» массив, записанный в файл, с помощью стека. Размер массива неизвестен.
2. Напишите программу, которая вычисляет значение арифметического выражения, записанного в постфиксной форме. Выражение вводится с клавиатуры в виде символьной строки. Предуспомотрите сообщения об ошибках.
3. Напишите программу, которая проверяет правильность скобочного выражения с четырьмя видами скобок: `()`, `[]`, `{}` и `<>`.
4. Дополните предыдущую программу так, чтобы она определяла номер ошибочного символа в строке.
5. Напишите вариант предыдущей программы, в котором в качестве стека используется символьная строка.
6. Найдите в литературе или в Интернете алгоритм перевода арифметического выражения из инфиксной формы в постфиксную, и напишите программу, которая решает эту задачу.
7. Напишите программу, которая выполняет заливку одноцветной области заданным цветом. Матрица, содержащая цвета пикселей, вводится из файла. Затем с клавиатуры вводятся координаты точки заливки и цвет заливки. На экран нужно вывести матрицу, которая получилась после заливки.
8. *Напишите решение задачи о заливке области, в котором точки, добавленные в очередь, как-то помечаются, чтобы не добавлять их повторно. В чём преимущества и недостатки такого алгоритма?

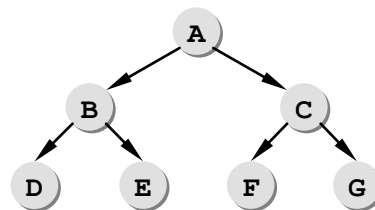
§ 42. Деревья

Что такое дерево?

Как вы знаете из учебника 10 класса, дерево – это структура данных, отражающая иерархию (отношения подчиненности, 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 рисунке справа родитель узла Е – это узел В, а предки узла Е – это узлы А и В, для которых узел Е – потомок. Потомками узла А (корня дерева) являются все остальные узлы.

Высота дерева – это наибольшее расстояние (количество рёбер) от корня до листа. Высота дерева, приведённого на рисунке, 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* (см. главу 8 учебника 10 класса). 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* (или *бинарные*) деревья, то есть такие, в которых каждый узел имеет не более двух сыновей. Их также можно определить рекурсивно.

Двоичное дерево:

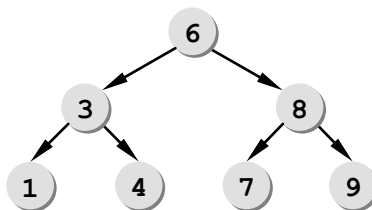
- 1) пустая структура – это двоичное дерево;
- 2) двоичное дерево – это корень и два связанных с ним отдельных двоичных дерева («левое» и «правое» поддеревья).

Деревья широко применяются в следующих задачах:

- поиск в большом массиве данных;
- сортировка данных;
- вычисление арифметических выражений;
- оптимальное кодирование данных (метод сжатия Хаффмана).

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, например:



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 показанного дерева:

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

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

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*, то есть 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 $\log_2 N$,

то есть алгоритм имеет асимптотическую сложность $O(\log N)$. 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):

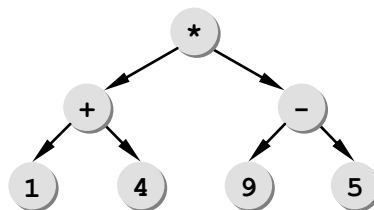
```
обойти левое поддерево
посетить корень
обойти правое поддерево
```

- ЛПК = «левый – правый – корень» (обход в обратном порядке):

```
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 $(1+4) * (9-5)$:



Выражение вычисляется по такому дереву снизу вверх, то есть корень дерева – это последняя выполняемая операция.

Различные типы обхода дают последовательность узлов:

КЛП: * + 1 4 - 9 5

ЛКП: 1 + 4 * 9 - 5

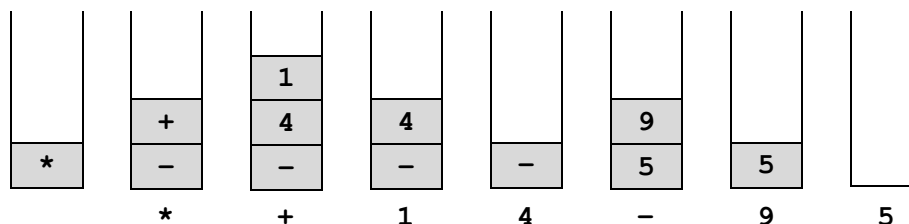
ЛПК: 1 4 + 9 5 - *

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», потому что сначала мы идём вглубь дерева по левым поддеревьям, пока не дойдём до листа. Такой обход можно выполнить с помощью стека следующим образом:

```
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 6.13 показано изменение состояния стека при таком обходе дерева, изображенного на рис. 6.12. Под стеком записана метка узла, который посещается (например, данные из этого узла выводятся на экран).



Существует еще один способ обхода, который называют «обходом в ширину». 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 приведённого выше дерева даст такую последовательность посещения узлов:

обход в ширину: * + - 1 4 9 5

Для того, чтобы выполнить такой обход, применяют очередь. В очередь 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
  
```

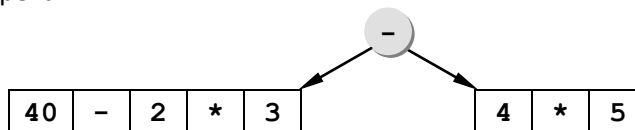
Вычисление арифметических выражений

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

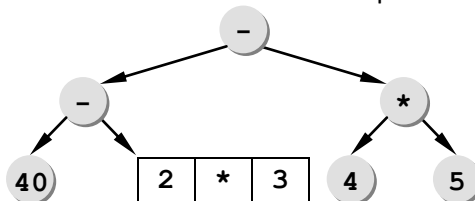
Так как корень дерева – это последняя операция, нужно сначала найти эту последнюю операцию, просматривая выражение слева направо. Здесь последнее действие – это второе вычитание, оно оказывается в корне дерева.



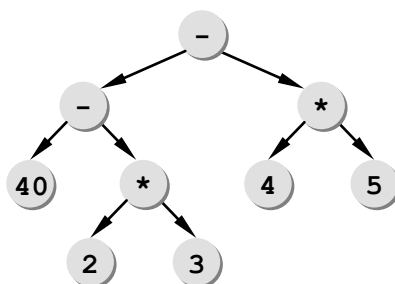
Как выполнить этот поиск в программе? Известно, что операции выполняются в порядке *приоритета* (старшинства): сначала операции с более высоким приоритетом (слева направо), потом – с более низким (также слева направо). Отсюда следует важный вывод:

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

Теперь нужно построить таким же способом левое и правое поддеревья:



Левое поддерево требует еще одного шага:



Эта процедура рекурсивная, её можно записать в виде псевдокода:

```

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

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

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

```

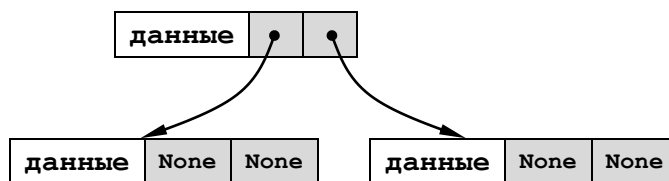
n1 = значение левого поддерева
n2 = значение правого поддерева
результат = операция ( n1, n2 )
  
```

Снова получился рекурсивный алгоритм.

Возможен особый случай (на нём заканчивается рекурсия), когда корень дерева содержит число (то есть это лист). Это число и будет результатом вычисления выражения.

Использование связанных структур

Поскольку двоичное дерево – это нелинейная структура данных, использовать список для размещения элементов 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 (указатель) и ссылка на правое поддерево (второй указатель). У листьев нет «сыновей», в этом случае в указатели будем записывать специальное значение **None** («пусто», «ничто»). Дерево, состоящее из трёх таких узлов, показано на рисунке:



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

Введём новый тип (класс) данных – структуру **TNode** – узел дерева (вспомните работу со структурами из § 39):

```

class TNode:
    pass
  
```

Определим функцию, которая создаёт новый узел, записывает в его поле **data** переданные данные и присваивает нулевые значения указателям на поддеревья (полям **left** и **right**):

```

def newNode( d ):
    node = TNode()
    node.data = d
    node.left = None
    node.right = None
  
```

```
return node
```

Пусть **s** – символьная строка, в которой записано арифметическое выражение (будем предполагать, что это правильное выражение без скобок). Тогда вычисление выражения сводится к двум вызовам функций:

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

Здесь 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 новый узел и искать последнюю выполняемую операцию – это будет делать функция **lastOp**. Она вернет «-1», если ни одной операции не обнаружено, в этом случае создается лист – узел без потомков. Если операция найдена, её обозначение записывается в поле **data**, а в указатели – адреса поддеревьев, которые строятся рекурсивно для левой и правой частей выражения:

```
def makeTree ( s ) :
    k=lastOp(s)
    if k<0:                                # создать лист
        Tree=newNode ( s )
    else:                                  # создать узел-операцию
        Tree=newNode ( 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 Tree.left==None:
        return int(Tree.data)
    else:
        n1=calcTree ( Tree.left )
        n2=calcTree ( Tree.right )
        if Tree.data=="+": res=n1+n2
        elif Tree.data=="-": res=n1-n2
        elif Tree.data=="*": res=n1*n2
        else: res=n1//n2
    return res
```

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.

Осталось написать функцию **lastOp**. Нужно найти в символьной строке последнюю операцию с минимальным приоритетом. Для этого составим функцию, возвращающую приоритет операции (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

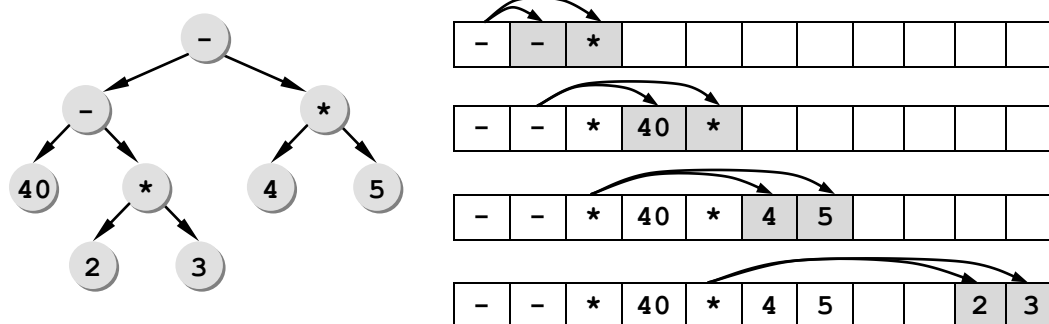
```

Обратите внимание, что в условном операторе указано 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, условный оператор срабатывает, а если в строке нет операций, условие всегда ложно и в переменной **lastOp** остается начальное значение «-1».

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 показан порядок расположения элементов в массиве для дерева, соответствующего выражению

40 - 2 * 3 - 4 * 5



Алгоритм вычисления выражения остается прежним, изменяется только метод хранения данных. Обратите внимание, что некоторые элементы остались пустые, 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, то есть 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 дополни-

тельные 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.

Контрольные вопросы

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

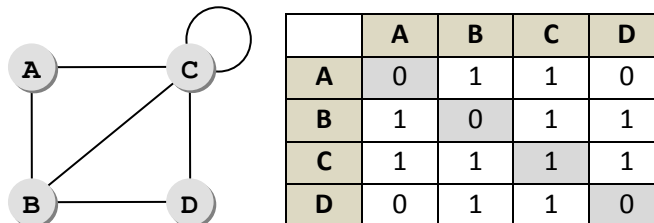
Задачи

1. Соберите программу, которая вводит и вычисляет арифметическое выражение без скобок. Все операции с деревом вынесите в отдельный модуль.
2. Добавьте в предыдущую программу процедуры обхода построенного дерева так, чтобы получить префиксную и постфиксную запись введенного выражения.
3. *Добавьте в предыдущую программу процедуру обхода дерева в ширину.
4. *Усовершенствуйте программу (см. задачу 1), чтобы она могла вычислять выражения со скобками.
5. *Включите в вашу программу обработку некоторых ошибок (например, два знака операций подряд). Поработайте в парах: обменяйтесь программами с соседом и попробуйте найти выражение, при котором его программа завершится аварийно (и не выдаст собственное сообщение об ошибке).
6. *Напишите программу вычисления арифметического выражения, которая хранит дерево в виде массива. Все операции с деревом вынесите в отдельный модуль.

§ 43. Графы

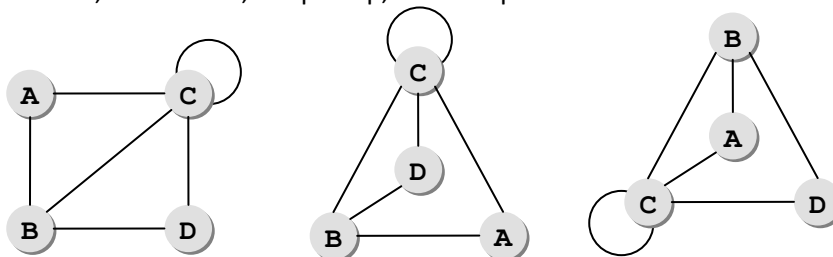
Что такое граф?

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*:



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).

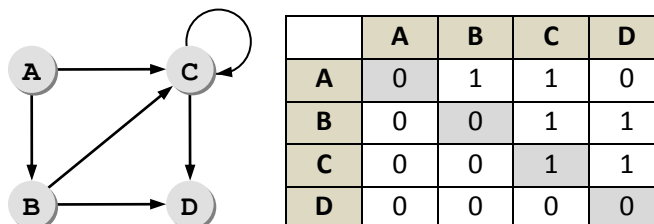
Строго говоря, граф – это математический объект, а не рисунок. Конечно, его можно нарисовать на плоскости (например, как на рис. 1.16, б), но матрица смежности не даёт никакой информации о том, как именно следует располагать вершины друг относительно друга. Для таблицы, приведенной выше, возможны, например, такие варианты:



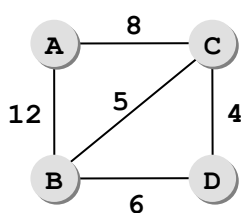
В рассмотренном примере все узлы связаны, то есть, между любой парой вершин существует *путь* – последовательность рёбер, по которым можно перейти из одной вершины в другую. Такой граф называется *связным*.

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:

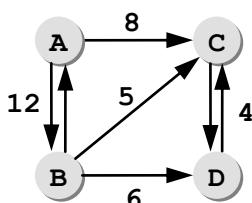


Часто с каждым ребром связывают некоторое число – *вес ребра*. Это может быть, например, расстояние между городами или стоимость проезда. Такой граф называется *взвешенным*. Информация о взвешенном графе хранится в виде *весовой матрицы*, содержащей веса рёбер:



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

У взвешенного орграфа весовая матрица может быть несимметрична относительно главной диагонали:

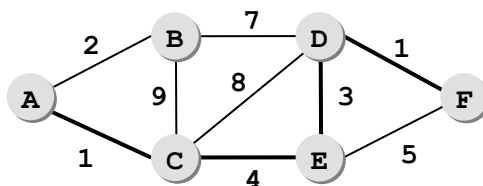


	A	B	C	D
A		12	8	
B	12		5	6
C				4
D			4	

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

«Жадные» алгоритмы

Задача 6. Известна схема дорог между несколькими городами. Числа на схеме (рис. 6.26) обозначают расстояния (дороги не прямые, поэтому неравенство треугольника может нарушаться):

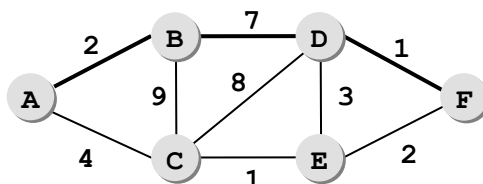


Нужно найти кратчайший маршрут из города А в город F.

Первая мысль, которая приходит в голову – на каждом шаге выбирать кратчайший маршрут до ближайшего города, в котором мы еще не были. Для заданной схемы на первом этапе едем в город С (длина 1), далее – в Е (длина 4), затем в D (длина 3) и наконец в F (длина 1). Общая длина маршрута равна 9.

Алгоритм, который мы применили, называется «жадным». Он состоит в том, чтобы на каждом шаге многоходового процесса выбирать наилучший 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 дает оптимальное решение, но так будет далеко не всегда. Например, для той же задачи с другой схемой:



жадный алгоритм даст маршрут A-B-D-F длиной 10, хотя существует более короткий маршрут A-C-E-F длиной 7.

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:

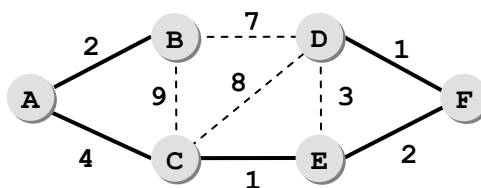
Задача 7. В стране Лимонии есть N городов, которые нужно соединить линиями связи. Между какими городами нужно проложить линии связи, чтобы все города были связаны в одну систему и общая длина линий связи была наименьшей?

В теории графов эта задача называется задачей построения *минимального остовного дерева* (то есть дерева, связывающего все вершины). Остовное дерево для связного графа с N вершинами имеет $N-1$ 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 показано минимальное остовное дерево для одного из рассмотренных выше графов (сплошные жирные линии):



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, то есть 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 выкрашены в один цвет, то есть 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 $col[iMin]$.

Приведем 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]

```

Здесь **W** – весовая матрица размера **N** на **N** (индексы строк и столбцов начинаются с 0); **ostov** – список хранения выбранных рёбер (для каждого ребра хранится кортеж из номеров двух вершин, которые оно соединяет). Если связи между вершинами **i** и **j** нет, в элементе **W[i][j]** матрицы будем хранить «бесконечность» – число, намного большее, чем длина любого ребра. При этом начальное значение переменной **minDist** должно быть ещё больше.

После окончания цикла остается вывести результат – рёбра из массива **ostov**:

```

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

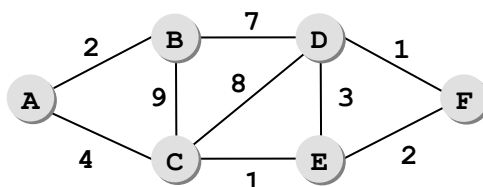
```

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]**.

Кратчайшие маршруты

На примере задачи выбора кратчайшего маршрута (см. задачу 8 выше) мы увидели, что в ней жадный алгоритм не всегда дает правильное решение. В 1960 году Э. 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.

Рассмотрим уже знакомую схему, в которой не сработал жадный алгоритм:



Алгоритм Дейкстры использует дополнительные массивы: в одном (назовем его **R**) хранятся кратчайшие (на данный момент) расстояния от исходной вершины до каждой из вершин графа, а во втором (массив **P**) – вершина, из которой нужно приехать в данную вершину.

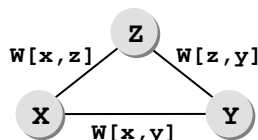
Сначала записываем в массив **R** длины рёбер от исходной вершины **A** до всех вершин, а в соответствующие элементы массива **P** – вершину **A**:

	A	B	C	D	E	F
R	0	2	4	∞	∞	∞
P	x	A	A			

Знак ∞ , обозначает, что прямого пути нет из вершины **A** в данную вершину нет (в программе вместо ∞ можно использовать очень большое число). Таким образом, вершина **A** уже рассмотрена и выделена серым фоном. В первый элемент массива **P** записан символ **x**, обозначающий начальную точку маршрута (в программе можно использовать несуществующий номер вершины, например, «-1» или **None**).

Из оставшихся вершин находим вершину с минимальным значением в массиве **R**: это вершина **B**. Теперь проверяем пути, проходящие через эту вершину: не позволят ли они сократить маршрут к другим, которые мы ещё не посещали. 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 :



Проверяем наш граф: ехать из A в C через B невыгодно (получается путь длиной 11 вместо 4), а вот в вершину D можно проехать (путь длиной 4), поэтому запоминаем это значение вместо ∞ в массиве R , и записываем вершину B на соответствующее место в массив P («в D приезжаем из B »):

	A	B	C	D	E	F
R	0	2	4	9	∞	∞
P	x	A	A	B		

Вершины E и F по-прежнему недоступны.

Следующей рассматриваем вершину C (для нее значение в массиве R минимально). Оказывается, что через неё можно добраться до E (длина пути 5):

	A	B	C	D	E	F
R	0	2	4	9	5	∞
P	x	A	A	B	C	

Затем посещаем вершину E , которая позволяет достигнуть вершины F и улучшить минимальную длину пути до вершины D :

	A	B	C	D	E	F
R	0	2	4	8	5	7
P	x	A	A	E	C	E

После рассмотрения вершин F и D таблица не меняется. Итак, мы получили, что кратчайший маршрут из A в F имеет длину 7, причем он приходит в вершину F из E . Как же получить весь маршрут? Нужно просто посмотреть в массиве P , откуда лучше всего ехать в E – выясняется, что из вершины C , а в вершину C – напрямую из начальной точки A :

	A	B	C	D	E	F
R	0	2	4	8	5	7
P	x	A	A	E	C	E

Поэтому кратчайший маршрут $A-C-E-F$. Обратите внимание, что этот маршрут «раскручивается» в обратную сторону, от конечной вершины к начальной. Заметим, что полученная таблица содержит все кратчайшие маршруты из вершины A во все остальные, а не только из A в F .

Алгоритм Дейкстры можно рассматривать как своеобразный «жадный» алгоритм: действительно, на каждом шаге из всех невыбранных вершин выбирается вершина X , длина пути до которой от вершины A минимальна. Однако можно доказать, что это расстояние – действительно минимальная длина пути от A до X . Предположим, что для всех предыдущих выбранных вершин это свойство справедливо. При этом X – это ближайшая не выбранная вершина, которую можно достичь из начальной точки, проезжая только через выбранные вершины. Все остальные пути в X , проходящие через ещё не выбранные вершины, будут длиннее, поскольку все рёбра имеют положительную длину. Таким образом, найденная длина пути из A в X – минимальная. После завершения алгоритма, когда все вершины выбраны, в массиве R находятся длины кратчайших маршрутов.

Теперь напомним программу на Python. Переменная N будет обозначать количество вершин графа, «список списков» W – это весовая матрица, её удобно вводить из файла. Логический массив **active** хранит состояние вершин (просмотрена или не просмотрена): если значение **active**[i] истинно, то вершина активна (ещё не просматривалась).

В начале программы присваиваем начальные значения (объяснение см. выше):

```
active = [True]*N # все вершины не просмотрены
R = W[0][:]      # скопировать в R строку 0 весовой матрицы
P = [0]*N        # только прямые маршруты из вершины 0
```

Обратите внимание, что нельзя написать

```
R = W[0] # неверное копирование строки W[0] в массив R!
```

потому что таким образом мы записываем в переменную **R** ссылку на строку 0 матрицы **W**, и при любом изменении массива **R** нулевая строка матрицы **W** также будет изменяться. Добавление «[:]» создаёт срез этого массива «от начала до конца», и в переменную **R** записывается ссылка на новый объект, который будет независим от матрицы **W**.

Сразу помечаем, что вершина 1 просмотрена (не активна), с нее начинается маршрут.

```
active[0] = False
P[0] = -1
```

В основном цикле, который выполняется **N-1** раз (так, чтобы все вершины были просмотрены) среди активных вершин ищем вершину с минимальным соответствующим значением в массиве **R** и проверяем, не лучше ли ехать через неё:

```
for i in range(N-1):
    # поиск новой рабочей вершины R[j] -> min
    minDist = 1e10 # очень большое число
    for j in range(N):
        if active[j] and R[j] < minDist:
            minDist = R[j]
            kMin = j
    active[kMin] = False
    # проверка маршрутов через вершину kMin
    for j in range(N):
        if R[kMin] + W[kMin][j] < R[j]:
            R[j] = R[kMin] + W[kMin][j]
            P[j] = kMin
```

В конце программы выводим оптимальный маршрут (здесь – до вершины с номером **N-1**) в обратном порядке следования вершин:

```
i = N-1
while i >= 0: # для начальной вершины P[i] = -1
    print(i, end=" ")
    i = P[i] # переход к следующей вершине
```

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:

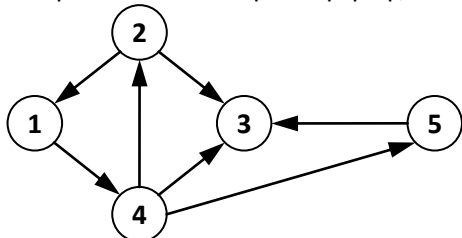
```
for k in range(N):
    for i in range(N):
        for j in range(N):
            if W[i][k] + W[k][j] < W[i][j]:
                W[i][j] = W[i][k] + W[k][j]
```

В результате исходная весовая матрица графа **W** размером **N** на **N** превращается в матрицу, хранящую длины оптимальных маршрутов. Для того, чтобы найти сами маршруты, нужно использовать еще одну дополнительную матрицу, которая выполняет ту же роль, что и массив **P** в алгоритме Дейкстры. Подробное описание и реализацию этого алгоритма вы можете найти в литературе или в Интернете.

Использование списков смежности

Как вы знаете (см. § 4 учебника для 10 класса), граф можно описать с помощью списков смежности. Поскольку в Python есть встроенный тип данных «список», при программировании на этом языке такое описание очень удобно использовать.

Список смежности для какой-то вершины – это *множество* вершин, с которыми связана данная вершина. Рассмотрим орграф, состоящий из 5 вершин:



Для него списки смежности (с учетом 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 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*, то есть 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. Попробуйте улучшить программу самостоятельно так, чтобы исправить эти недостатки.

Некоторые задачи

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

Задача 8. 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 от $N-1$. Это число с увеличением N растёт очень быстро, быстрее, чем любая степень N . Уже для $N=20$ такое решение требует огромного времени вычислений: компьютер, проверяющий 1 млн вариантов в секунду, будет решать задачу «в лоб» около четырёх тысяч лет. Поэтому математики прилагали большие усилия для того, чтобы сократить перебор – не рассматривать те варианты, которые заведомо не дают лучших результатов, чем уже полученные. В реальных ситуациях нередко оказываются полезны приближенные решения, которые не гарантируют точного оптимума, но позволяют получить приемлемый вариант.

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

Задача 9 (о максимальном потоке). Есть система труб, которые имеют соединения в N узлах. Один узел S является источником, еще один – стоком T . Известны пропускные способности каждой трубы. Надо найти наибольший поток (количество жидкости, перетекающее за единицу времени) от источника к стоку.

Задача 10. Имеется N населенных пунктов, в каждом из которых живет p_i школьников ($i=1, \dots, N$). 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 11 (о наибольшем паросочетании). Есть M мужчин и N женщин. Каждый мужчина указывает несколько (от 0 до N) женщин, на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от 0 до M), за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.



Контрольные вопросы

1. Что такое граф?
2. Как обычно задаются связи узлов в графах?
3. Что такое матрица смежности?

4. Что такое петля? Как «увидеть» её в матрице смежности?
5. Что такое путь?
6. Какой граф называется связным?
7. Что такое орграф?
8. Как по матрице смежности отличить орграф от неориентированного графа?
9. Что такое взвешенный граф? Как может храниться в памяти информация о нём?
10. Что такое «жадный алгоритм»? Всегда ли он позволяет найти лучшее решение?
11. Подумайте, как можно было бы ускорить работу алгоритма Крускала с помощью предварительной сортировки рёбер.
12. Объясните, как задать граф в языке Python с помощью списков смежности.
13. Какие достоинства и недостатки, на ваш взгляд, имеют разные способы хранения данных о графе в программе?



Задачи

1. Напишите программу, которая вводит из файла весовую матрицу графа и строит для него минимальное остовное дерево.
2. Оцените асимптотическую сложность алгоритма Крускала.
3. *Программу для поиска минимального остовного дерева можно улучшить, если предварительно составить список ребёр и отсортировать его по возрастанию длин рёбер. Внесите это изменение в программу.
4. Напишите программу, которая вводит из файла весовую матрицу графа, затем вводит с клавиатуры номера начальной и конечной вершин и определяет оптимальный маршрут.
5. Напишите программу, которая вводит из файла весовую матрицу графа и определяет длины всех оптимальных маршрутов с помощью алгоритма Флойда-Уоршелла.
6. Оцените асимптотическую сложность алгоритмов Дейкстры и Флойда-Уоршелла.
7. Напишите программу, которая решает задачу коммивояжера для 5 городов методом полного перебора. Можно ли использовать ее для 50 городов?
8. *Напишите программу, которая решает задачу о размещении школы. Для определения кратчайших путей используйте алгоритм Флойда-Уоршелла. Весовую матрицу графа вводите из файла.
9. Перепишите программу для поиска количества путей в графе так, чтобы она не вычисляла данные повторно. Например, если мы подсчитали количество путей из вершины X в конечную вершину, то когда это значение потребуется снова, нужно сразу взять полученный ранее результат, а не вызывать функцию рекурсивно.
10. Перепишите программу для поиска количества путей в графе так, чтобы она выводила не только количество путей, но и сами маршруты как последовательность номеров вершин.

§ 44. Динамическое программирование

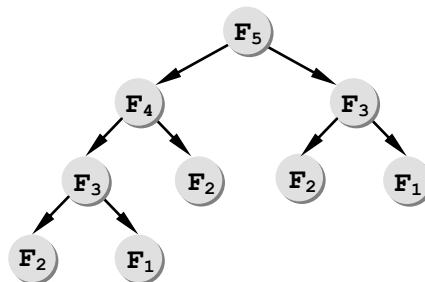
Что такое динамическое программирование?

Мы уже сталкивались с последовательностью чисел Фибоначчи (см. учебник 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 рисунке справа. Таким образом, мы 2 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.

Какой 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:

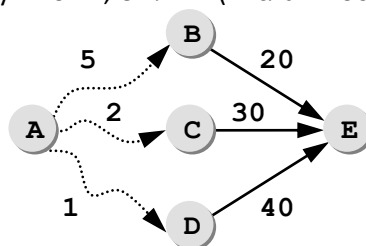
```
F = [1] * (N+1)
```

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 путем сведения их к более простым задачам того же типа

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 А в пункт Е через один из пунктов В, С или D (числами обозначена «стоимость» маршрута):



Пусть уже известны оптимальные маршруты из пунктов В, С и D в пункт Е (они обозначены сплошными линиями) и их «стоимость». Тогда для нахождения оптимального маршрута из А в Е нужно выбрать вариант, который даст минимальную стоимость по сумме двух шагов. В данном случае это маршрут А–В–Е, стоимость которого равна 25. Как видим, такие задачи решаются «с конца», то есть 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 удастся 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
```

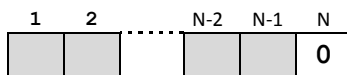
Ответ всегда будет находиться в переменной **f2**.

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

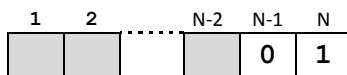
При больших **N** решение задачи методом перебора потребует огромного времени вычисления. Для того, чтобы использовать метод динамического программирования, нужно

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

Самое главное – вывести рекуррентную формулу, выражающую K_N через решения аналогичных задач меньшей размерности. Рассмотрим цепочку из N бит, последний элемент которой – 0.



Поскольку дополнительный 0 не может привести к появлению двух соседних единиц, подходящих последовательностей длиной N с нулем в конце существует столько, сколько подходящих последовательностей длины $N-1$, то есть K_{N-1} . Если же последний символ – 1, то вторым обязательно должен быть 0, а начальная цепочка из $N-2$ битов должна быть «правильной». Поэтому подходящих последовательностей длиной N с единицей в конце существует столько, сколько подходящих последовательностей длины $N-2$, то есть K_{N-2} .



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 последовательности длиной 1 (0 и 1), то есть $K_1 = 2$. Далее, есть 3 подходящих 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 объемом 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 введенного числа N .

Самый простой подход – заполнять сначала бидоны самого большого размера (6 л), затем – меньшие и т.д. Это так называемый «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 определим 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, объем 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 – $K_N = 1 + K_{N-6}$. Так как нам нужно выбрать минимальное значение, то

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

Вариант, выбранный при поиске минимума, определяет последний добавленный бидон, его нужно сохранить в отдельном массиве P . Этот массив будет использован для определения количества выбранных бидонов каждого типа. В качестве начальных значений берем $K_0 = 0$ и $P_0 = 0$.

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

$$K_3 = 1 + K_2 = 3, \quad K_5 = 1 + \min \{K_4, K_0\} = 1.$$

На рисунке показаны массивы для $N = 10$.

№	0	1	2	3	4	5	6	7	8	9	10
К	0	1	2	3	4	1	1	2	3	4	2
Р	0	1	1	1	1	5	6	1	1	1	5

Как по массиву **Р** определить оптимальный состав бидонов? Пусть, для примера $N = 10$. Из массива **Р** находим, что последний добавленный бидон имеет объем 5 л. Остается $10 - 5 = 5$ л, в элементе **Р**[5] тоже записано значение 5, поэтому второй бидон тоже имеет объем 5 л. Остаток 0 л означает, что мы полностью определили набор бидонов.

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 $p_i (i = 1, \dots, N)$ набрать кучу весом ровно W или, если это невозможно, максимально близкую к W (но меньшую, чем W). Все веса камней и значение W – целые числа.

Эта задача относится к трудным задачам целочисленной оптимизации, которые решаются только полным перебором вариантов. Каждый камень может входить в кучу (обозначим это состояние как 1) или не входить (0). Поэтому нужно выбрать цепочку, состоящую из N бит. При этом количество 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 из камней весом 2, 4, 5 и 7 единиц:

	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								

Теперь предположим, что строки с 1-ой по (*i*-1)-ую уже заполнены. Перейдем к *i*-ой строке, то есть добавим в набор *i*-ый 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*], то есть 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), то есть **T**[*i*][*w*] = **T**[*i*-1][*w*- p_i] + p_i .

Как же выбрать, «брать или не брать»? Проверить, в каком случае полученный вес будет больше (ближе к *w*). Таким образом, получается рекуррентная формула для заполнения таблицы:

при $w < p_i$: $\mathbf{T}[i][w] = \mathbf{T}[i-1][w]$

при $w \geq p_i$: $\mathbf{T}[i][w] = \max(\mathbf{T}[i-1][w], \mathbf{T}[i-1][w-p_i] + p_i)$

Используя эту формулу, заполняем таблицу по строкам, сверху вниз; в каждой строке – слева направо:

	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

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 включен в набор, то $T[i][w] = T[i-1][w]$, то есть число в таблице не меняется при переходе на строку вверх. Начинаем с левого нижнего угла таблицы, идем 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 идем 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 , то есть алгоритм полного перебора имеет асимптотическую сложность $O(2^N)$. В данном алгоритме количество операций равно числу элементов таблицы, то есть сложность нашего алгоритма – $O(N \cdot W)$. Однако нельзя сказать, что он имеет линейную сложность, так как есть еще сильная зависимость от заданного веса W . Такие алгоритмы называют *псевдополиномиальными*, то есть «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 20?

Заметим, что при выполнении любой из команд число увеличивается (не может уменьшаться). Начнем с простых случаев, с которых будем начинать вычисления. Понятно, что для числа 1 существует только одна программа – пустая, не содержащая ни одной команды. Для числа 2 есть тоже 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, \dots, K_{N-1} , то есть с решениями таких же задач для меньших N .

Если число N не делится на 3, то оно могло быть получено только последней операцией 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:

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

Остается заполнить таблицу для всех значений от 1 до заданного $N = 20$. Для небольших значений N эту задачу легко решить вручную:

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

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):

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

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 , и заполнить его по приведённым выше формулам:

```

K = [0] * (N+1)
K[1] = 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 $p_i (i = 1, \dots, N)$? 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 заполнить сразу единицами. Для примера мы будем рассматривать задачу для $W = 10$ и набора монет достоинством 1, 2, 5 и 10 рублей:

	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										

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, то есть $T[i][w] = T[i-1][w] + T[i][w - p_i]$. В итоге получается рекуррентная формула

при $w < p_i$: $T[i][w] = T[i-1][w]$

при $w \geq p_i$: $T[i][w] = T[i-1][w] + T[i][w - p_i]$

которая используется для заполнения таблицы:

	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

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.



Контрольные вопросы

1. Что такое динамическое программирование?
2. Какой смысл имеет выражение «динамическое программирование» в теории многошаговой оптимизации?
3. Какие шаги нужно выполнить, чтобы применить динамическое программирование к решению какой-либо задачи?
4. За счёт чего удастся ускорить решение сложных задач методом динамического программирования?
5. Какие ограничения есть у метода динамического программирования?



Задачи

1. Напишите программу, которая определяет оптимальный набор бидонов в задаче с молоком. С клавиатуры или из файла вводится объём цистерны, количество типов бидонов и их размеры.
2. Напишите программу, которая решает задачу о куче камней заданного веса, рассмотренную в тексте параграфа.
3. * Задача о ранце. Есть N предметов, для каждого из которых известен вес $p_i (i = 1, \dots, N)$ и стоимость $c_i (i = 1, \dots, N)$. В ранец можно взять предметы общим весом не более W . Напишите программу, которая определяет самый дорогой набор предметов, который можно унести в ранце.
4. У исполнителя Калькулятор две команды, которым присвоены номера:

1. прибавь 1
3. умножь на 4

Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число 1 в число N , введенное с клавиатуры. Используйте сокращённую таблицу.

5. У исполнителя Калькулятор три команды, которым присвоены номера:

1. прибавь 1
2. умножь на 3
3. умножь на 4

Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число 1 в число N , введенное с клавиатуры.

Самое важное в главе 6:

- Структура – это сложный тип данных, который позволяет объединить данные разных типов. Элементы структуры называют полями. При обращении к полям структуры используется точечная запись:
<имя структуры>. <имя поля>.
- Список – это упорядоченный набор элементов одного типа, для которых введены операции вставки (включения) и удаления (исключения).
- Стек – это линейный список, в котором добавление и удаление элементов разрешается только с одного конца. Системный стек применяется для хранения адресов возврата из подпрограмм и размещения локальных переменных.
- Дерево – это структура данных, которая моделирует иерархию – многоуровневую структуру. Как правило, дерево определяется с помощью рекурсии, поэтому для его обработки удобно использовать рекурсивные алгоритмы. Деревья используются в задачах поиска, сортировки, вычисления арифметических выражений.
- Граф – это набор вершин и связывающих их рёбер. Информация о графе чаще всего хранится в виде матрицы смежности или весовой матрицы. Наиболее известные задачи, которые решаются с помощью теории графов – поиск оптимальных маршрутов.
- Динамическое программирование – это метод, позволяющий ускорить решение задачи за счет хранения решений более простых задач того же типа. Для его использования нужно вывести рекуррентную формулу, связывающее решение задачи с решением подобных задач меньшей размерности, и определить простые базовые случаи (условие окончания рекурсии).