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 ( ). 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 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 и13.

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 $k^2 \le N$.

Чтобы составить программу, нужно определить, что значит «выписать все числа» и «вычеркнуть число». Один из возможных вариантов хранения, данных – массив логических величин с индексами от 2 до N. 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 ≤ √N на равносильное условие k2√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 числа[1] и добавляя 1:

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

Здесь многоточие обозначает операторы, составляющие тело цикла. Теперь преобразуем внутренний цикл: переменная **i** изменяется в диапазоне от k^2 до N с шагом k, поэтому окончательно получаем:

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

Массив для вывода 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 состоит вопрос о том, как хранить такие числа в памяти. Ответ достаточно очевиден: нужно «разбить» длинное число на части так, чтобы использовать несколько ячеек памяти.

Далее мы рассмотрим общие алгоритмы, позволяющие работать с «длинными» числами, при ограниченном размере ячеек памяти. Это позволит вам научиться 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, число12345678 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

$$\text{от } -2^{32} = -4\,294\,967\,296 \quad \text{до} \quad 2^{32} - 1 = 4\,294\,967\,295.$$

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 выполняться арифметические операции, результат которых должен «помещаться» в такую же ячейку памяти. Например, если надо умножать 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! = 1·2·3·…·99·100 и вы-вести его на экран в десятичной системе счисления (*это число состоит более чем из сотни цифр и явно не помещается в одну ячейку памяти*).

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

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

Число 100^{100} содержит 201 цифру, поэтому число 100! содержит не более 200 цифр. Если в каждом элементе массива записано 6 цифр, для хранения всего числа требуется не более 34 ячеек. В решении на 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 ≤ 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 ·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.

	2	1	0
A	1	2	3

Нужно учесть, что в каждой ячейке хранятся 6 цифр, поэтому в массиве 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 число. 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 формулы.

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

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

Метод Герона «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.