Лекция № 3

Т. Ф. Хирьянов

Позиционные системы счисления

Система счисления — это механизм кодирования чисел. Сами числа существуют независимо от формы его записи. Например, тысяча в римской системе счисления записывается, как M, а в арабской – 1000. При этом эти записи обозначают одно и то же число. Самой простой системой счисления является *унарная*. В ней существует только один символ, который в случае числа A ставится подряд A раз.

Числа записываются с помощью цифр. Количество цифр соответствует названию системы. Например, в троичной СС их 3 (0, 1, 2), в двоичной – 2 (0, 1). Рассмотрим запись числа в десятичной системе счисления

$$12345_{10} = 10000 + 2000 + 300 + 40 + 5 = 1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 + 5 * 10^0$$

а также в двоичной

$$1010101_2 = 2^6 + 2^4 + 2^2 + 2^0 = 64 + 16 + 4 + 1 = 85$$

Стоит отметить, что в двоичной системе таблицы умножения и сложения выглядят очень просто.

×	0	1
0	0	0
1	0	1

+	0	1
0	0	1
1	1	10_2

Также она является наиболее удобной для компьютера.

Для того чтобы выполнить обратную операцию воспользуемся представлением числа по схеме Горнера. Например,

$$12345_{10} = (((1 * 10 + 2) * 10 + 3) * 10 + 4) * 10 + 5$$

Из данного представления легко заметить, что последняя цифра числа есть остаток от деления на основание СС. Действительно,

$$12345 \ \% \ 10 == 5$$

Пусть теперь

$$y = 12345 \ div \ 10 == 1234$$

Тогда

$$y \ \% \ 10 == 4$$

Продолжая аналогично получим все цифры числа. Для расчетов на бумаге удобно пользоваться следующим представлением.

Данный алгоритм можно организовать в виде цикла. Для простоты можно написать программу, выводящую на экран цифры числа в обратном порядке, причем само число меняется по ходу выполнения программы.

```python
x = int(input())
while x != 0:
    print(x % 10)
    x //= 10
```

Существуют родственные двоичной системы счисления. Это четверичная, восьмеричная, шестнадцатиричная и т.д.

Рассмотрим таблицу соответствий этих четырех систем.

«2»	«4»	«8»	«16»	«2»	«4»	«8»	«16»
0	0	0	0	1000	20	10	8
1	1	1	1	1001	21	11	9
10	2	2	2	1010	22	12	A
11	3	3	3	1011	23	13	B
100	10	4	4	1100	30	14	C
101	11	5	5	1101	31	15	D
110	12	6	6	1110	32	16	E
111	13	7	7	1111	33	17	F

Внимательно посмотрев на нее, можно заметить, что когда заканчивается разряд в шестнадцатиричной системе счисления, в двоичной тоже оказываются исчерпанными уже четыре разряда. Аналогично разряду в восьмеричной системе счисления соответствует три разряда в двоичной, а в четвертичной — два. Так, например, для перехода из шестнадцатиричной системы необходимо вместо каждой цифры подставить ее четырехразрядное представление в двоичной системе.

$$3DE80C_16 = 0011\,1101\,1110\,1000\,0000\,1100_2$$

Классификация числовых алгоритмов

1. *число → число*
   Такие алгоритмы получаются при определении значения функции от данного числа.

2. *число → последовательность*
   С помощью таких алгоритмов по заданным числам (некоторым параметрам) генерируется последовательность. Обычно она вычисляется либо по выражению для ее члена, либо при помощи рекуррентных соотношений.

3. *последовательность → число*
   В данных алгоритмах обычно реализованны различные подсчеты (например, количества элементов, их суммы или произведения), поиск чисел с заданными свойствами, проверка упорядоченности.

4. *последовательность → последовательность*
   Например, фильтрация.

В третьем пункте обработка последовательности включает цикл, в котором поочередно пробегаются по ее элементам. При этом в зависимости от задачи перед переходом к следующей итерации реализуются соответствующие «микроалгоритмы».

Так, при подсчете количества элементов n, до начала цикла полагается $n = 0$, а затем в теле цикла выполняется всего одна команда $n \mathrel{+}= 1$.

При подсчете суммы s для начальной, пустой последовательности полагается $s = 0$. Это, конечно, не обоснованно, однако уже после первой итерации сумма станет верной. При этом выполнится команда $s \mathrel{+}= x$.

То же самое касается произведения p. Только в этом случае вначале полагается $p = 1$, чтобы на первой итерации, содержащей команду p *= x значение произведения не обратилось в ноль и итоговый результат не был искажен.

При поиске происходит сравнение текущего элемента с некоторым образцом. Результатом этого сравнения является логическое значение (true или false), которое может быть приведено к целому числу (соответственно 1 или 0). В данном алгоритме используется специальная переменная – флаг (f). Вначале полагается $f = 0$. Затем в каждой итерации выполняется $f = int(x == x_0)$. Тем самым по завершении цикла f будет содержать количество элементов, равных x_0.

При реализации данных алгоритмов сама последовательность может быть введена двумя способами. В первом из них вначале программы может быть поданно число элементов последовательности. Тогда их считывание удобно организовать при помощи цикла *for*, считывая вначале каждой итерации текущий элемент. При подсчете суммы это может выглядеть, например, таким образом:

```
N = int(input())
s = 0
for i in range(N):
    x = float(input())
    s += x
print(s)
```

Когда же исходно неизвестно количество элементов, но известно, что признаком конца последовательности является, например, число 0, удобно организовать считывание элементов иначе, используя цикл *while*. При этом число 0 не является последним элементом (в случае подсчета произведения это привело бы к его обнулению). В данном случае ноль — это *терминальный признак*. Переход к телу цикла осуществляется, только если выполнено условие $x! = 0$, т. е. когда значение x нетерминальное. При этом необходимо считать первый элемент до начала цикла, а последующие в конце каждой итерации. Тот же самый подсчет суммы может выглядеть следующим образом:

```
s = 0
x = float(input())
while x != 0:
    s += x
    x = float(input())
print(s)
```

В данной программе ноль будет считан в последней итерации, однако не войдет в сумму, так как при последующей проверке условия цикл завершится.

Такой механизм используется, например, в языке Си, где признаком конца файла является специальный символ *EOF*.

Фильтрация, как правило, не реализуется отдельно, используется при выполнении другой задачи, например, при подсчете среднего арифметического только четных элементов последовательности. Для этого в теле цикла команды-обработчики заключаются внутри условной конструкции, проверяющей, является ли элемент четным числом. Например, так.

```
if x % 2 == 0:
    s += 0
    n += 1
print(s/n)
```

В качестве примера генерации рекурсивной последовательности можно рассмотреть выведение на экран N-ого числа Фибоначчи. В данной последовательности первые два элемента равны 1, а каждый последующий – сумме двух предыдущих.

1 1 2 3 5 8 13 21...

Из этого следует, что для корректной работы программы не требуется хранить в памяти компьютера все члены. Для то чтобы получить следующий элемент, достаточно помнить только последнии два. Вначале алгоритма вводятся две переменные $f1 = 0$, $f2 = 1$ и счетчик $n = 1$. Можно заметить, что при таких значениях элемент с номером n хранится в $f2$. Это утверждение должно быть верным на протяжении всего времени работы программы. Пока номер n меньше N, значение $f1$ меняется на $f2$, а $f2$ – на сумму $f1$ и $f2$. Последнее удобно организовать с помощью присваивания кортежей. В конце итерации, конечно, необходимо увеличить счетчик n.

```python
N = int(input())
f1 = 0
f2 = 1
n = 1
while n < N:
    f1, f2 = f2, f1 + f2
    n += 1
print(f2)
```

Запись чисел в Python

В Python существует способ записи чисел в отличной от десятичной системе счисления. Например, введенное число number может быть записано в семеричной системе счисления, указав в качестве второго аргумента функции int() ее основание.

number = input() x = int(number, 7)

При этом в памяти компьютера числа будут храниться всегда в двоичной системе счисления и только во время вывода они приобретут необходимый вид.

Существует также способ задать переменной значение в виде числа в двоичной, восьмеричной или шестнадцатеричной системе счисления. Для этого в начале числа необходимо записать специальный литерал: $0b$ для двоичной, $0o$ для восьмеричной и $0x$ для шестнадцатеричной.

```python
x = 0b10110
x = 0o756
x = 0xB708
```

Однопроходные алгоритмы

В приведенных ранее алгоритмах количество операций линейно зависело от размера последовательности чисел, а объем требуемой памяти был конечным и не зависел этого размера.

Бывает, что в сложных задачах математика может помочь уменьшить количество вычислений.

Так, например, вычисление среднего квадратичного отклонения от средней величины на первый взгляд осуществляется в два прохода, и в памяти требуется хранить все числа.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2)}{N}}$$

Действительно, кажется, что сначала нужно подсчитать $\bar{x}$, а затем уже само отклонение. Однако если раскрыть квадрат и произвести суммирование, то можно получить известную формулу.

$$\sigma^2 = \frac{\sum_{i=1}^{N}({x_i}^2 - 2x_i\bar{x} + {\bar{x}_i}^2)}{N} =$$

$$= \frac{\sum_{i=1}^{N}{x_i}^2}{N} - 2\bar{x}^2 + \frac{\bar{x}^2}{N}N$$

Откуда

$$\sigma = \sqrt{\overline{x^2} - \bar{x}^2}$$

Из этой формулы следует, что вычислить отклонение можно за один проход, определяя среднее значение x и его средний квадрат.

Логические операции

В питоне существуют логические операции *and*, *or* и *not*. Они возвращают значения *True* или *False* (в качестве обозначений будут использоваться соответственно 1 и 0) и имеют следующие таблицы истинности.

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

A	not A
0	1
1	0