Лекция № 6

Т. Ф. Хирьянов

Поиск корней функции

Данная задача не является тривиальной. Например, для функции

$$f(x) = \sin \frac{1}{x}$$

В сколь угодно малой окрестности нуля найдется бесконечное колличество нулей. Более того, в прикладных задачах поведение функции неизвестно, а значит, между любыми двумя точками она может несколько раз пересечь ось абсцисс. Поэтому для решения данной задачи необходимо использовать выводы из математического анализа, а именно лемму о промежуточных значениях непрерывной функции. Из нее следует, что если функция непрерывна на отрезке $[a, b]$, то на нем она обязательно принимает все значения от $f(a)$ до $f(b)$ (при условии, что $f(b) > f(a)$). Поэтому если найден такой отрезок, на котором функция непрерывна и имеет противоположные по знаку значения, то на этом отрезке функция обязательно имеет корень.

Биссекция

Для описанной выше ситуации существует алгоритм, который позволяет сколь угодно точно определить корень функции. Идея алгоритма биссекции заключается в следующем. Вычисляется значение функции в середине отрезка. Если оно равно нулю, то корень найден, если больше либо меньше нуля, то производится сужение отрезка, содержащего корень. Действительно из постановки задачи следует, что $f(a) * f(b) < 0$. Следовательно либо $f(a) * f(c) < 0$, либо $f(c) * f(b) < 0$. Допустим, что верно первое. Тогда нужно заменить правую границу b отрезка $[a, b]$ на c. Получившийся отрезок $[a, c]$ вдвое меньше и удовлетворяет всем условиям задачи, а значит, с ним можно провести такие же действия. Из этого следует, что искомый корень после n таких итераций будет находится, например, в середине текущего отрезка с точностью до половины последнего. Задание необходимой точности обеспечивает выход из цикла.

```
f_a = f(a)
f_b = f(b)
while abs(b - a) > eps*2:
    c = (a + b)/2
    f_c = f(c)
    if f_c*f_a < 0:
        b = c                # a = a
    elif f_c*f_a > 0:
        a = c                # b = b
    else:
        break
```

Поиск в списке

Если производить поиск конкретного значения в неупорядоченном списке, то будет осуществляться последовательный перебор его элементов и количество операций будет сравнимо с N. Однако если список упорядочен по неубыванию, то можно воспользоваться аналогом алгоритма поиска корня функции методом деления пополам.

Сначала задаются значения индексов, ограничивающих индексы элементов списка. Затем в цикле диапазон значений индексов разделяют пополам аналогично предыдущему алгоритму (деление, конечно, целочисленное). Выход из цикла осуществляется, когда диапазон сокращается до одного элемента. При этом возвращается номер элемента массива наиболее близкого к искомому значению, но превышающего его. Если искомое значение меньше всех элементов, то возвратится ноль, если больше – N.

```
def upper_bond(key, A):
    A = sorted(A)
    l = -1
    r = len(A)
    while r > l + 1:
        m = (l + r)//2
        if A[m] > key:
            r = m
        else:                # A[m] ⩽ key
            l = m
    return r
```

Сортировка списка

Для того чтобы отсортировать список (сделать его упорядоченным) существует множество алгоритмов. Одной из самых долгих является сортировка обезьяны.

Сортировка обезьяны

Для реализации данного алгоритма необходима функция, перемешивающая элементы в массиве. В стандартной библиотеке Python есть такая функция shuffle. Ее можно подключить из модуля random следующим образом.

```
from random import shuffle
```

Описанная ниже функция monkey_sort перемешивает список, пока он не станет отсортированным.

```
def monkey_sort(A):
    while not is_sorted(A):
        shuffle(A)
```

При этом вызывается функция is_sorted, проверяющая, является ли текущий список упорядоченным.

```
def is_sorted(A):
    return A == sorted(A)
```

Так как количество перестановок n элементов равно $n!$, то количество операций пропорционально $n!$.

Сортировка вставками

Существует гораздо более быстрые алгоритмы. Одним из них является алгоритм сортировки вставками. В нем последовательно пробегаются все элементы, правее крайнего левого. Каждый следующий элемент вставляется в уже отсортированную часть списка, расположенную левее, причем так, чтобы упорядоченность сохранилась. В приведенном ниже примере использован циклический сдвиг, и соответствующая сложность алгоритма пропорциональна N^2. Однако алгоритм можно улучшить, если использовать рассмотренную ранее функцию upper_bond для поиска места, в которое необходимо переставить текущий элемент.

```
def insertion_sort(A):
    for i in range(1, len(A)):
        new_elem = A[i]
        j = i - 1
        while j >= 0 and A[j] > new_elem:
            A[j + 1] = A[j] # сдвиг
            j -= 1
        A[j + 1] = new_elem
```

Однако алгоритм можно улучшить, если использовать рассмотренную ранее функцию upper_bond для поиска места, в которое необходимо переставить текущий элемент.