

## Как сделать большой тест и проверить своё решение

Возможно, некоторые из вас уже привыкли проверять своё решение перед отправкой в тестирующую систему. А как быть, если тестов у вас несколько или если надо проверить достаточно ли быстро работает ваша программа на данных большого объёма?

Очень не хочется вводить тест каждый раз заново. А большой тест вводить “руками” и не получится.

Существует несколько, скажем так, профессиональных подходов к решению этого вопроса. Например, специальные библиотеки для Python, например, `doctest`, `unittest`, `pytest` и другие.

Кроме того, может помочь хорошая среда программирования (например, PyCharm предоставляет возможность создавать тесты и прогонять свой код на них, посмотрите вот [здесь](#) и рядом).

Но кажется, что все эти варианты предполагают знакомство с т.н. ООП (Объектно-Ориентированным Программированием), что пока рановато. Ну или не у всех есть PyCharm.

А ещё есть сайт [polygon.codeforces.com](http://polygon.codeforces.com), предназначенный как раз для разработки задач, проверяемых автоматическими тестирующими системами. Но он “заточен” скорее на спортивное программирование.

Давайте попробуем обойтись уже доступными средствами языка. Ничего, кроме массивов и функций не потребуется. Запускать программу надо будет в консоли (`cmd`).

Возьмём для примера задачу A из этого набора задач.

Напишем решение, пока не задумываясь об эффективности (A.py):

```
def solution_slow(x, k, A):
    i = 0
    while i + k < len(x):
        s = sum(x[i:i + k + 1])
        if s == A:
            return i + 1
        i += 1
    return 0

N = int(input())
k = int(input())
A = int(input())
x = list(map(int, input().split()))
print(solution_slow(x, k, A))
```

Теперь хочется его проверить. И сделать так, чтобы ввод данных (там целых 4 строчки) не вводить каждый раз.

Сделаем для этого в том же каталоге, что и наше решение, такой текстовый файл (test1.dat, да можно дать файлу какое угодно расширение):

```
4
2
9
2 2 3 4
```

Здесь правильный ответ 2.

Теперь запустим нашу программу в консоли хитрым образом, вот так:

```
python A.py < test1.dat
```

Эта штука называется перенаправлением ввода-вывода.

Мы запускаем при помощи интерпретатора языка Python (python) нашу программу (A.py) и даём ей на вход содержимое файла (test1.dat).

Наша программа, вместо того, чтобы ждать в консоли ввод, читает данные из указанного файла. При этом в самой программе у нас никаких операций с файлами нет.

Сделайте ещё один файл, например, test2.dat:

```
6
5
13
1 2 3 4 5 -2
```

и запустите его:

```
python A.py < test2.dat
```

Давайте теперь попробуем проверить, как наша программа будет работать на больших данных. Можно сделать файл с данными для тестирования, но не вводить же туда массив из  $10^5$  элементов?! Давайте вместо этого напишем программу, которая такой файл сделает.

Сначала потренируемся и напишем пример для небольшого теста чтобы убедиться, что всё работает.

Напишем такую функцию `gen_test()` и вызовем её. Можно сделать для этого отдельный файл (`gen_test.py`).

Она печатает какие-то данные (в правильном формате) для нашей программы.

```
def gen_test():
    print(10)
    print(3)
    print(17)
    x = [3, 2, 7, 8, 2, 4, 5, 6, -1, -1]
    print(' '.join(map(str, x)))
```

```
gen_test()
```

Запускаем пока без всякого перенаправления:

```
python gen_test.py
```

Видим вот такой результат:

```
10
3
17
3 2 7 8 2 4 5 6 -1 -1
```

Но перенаправить (в файл) можно не только ввод данных, но и вывод. Вот так:

```
python gen_test.py > test3.dat
```

Такая команда выполнит программу, но всё, что она выведет, будет записано в файл `test3.dat`. Если его в текущем каталоге до этого не было, он будет создан. А если был — перезапишется.

Теперь добавим параметров в функцию `gen_text()`, чтобы она могла сгенерировать какой-нибудь случайный пример.

В примере ниже передаются следующие параметры:

- `size` — размер массива
- `k` — размер окна
- `ans` — правильный ответ, который мы хотим видеть в выводе нашей программы (точнее “ответ — 1”, т.к. в задаче просят вывести номер, а не индекс)

```
from random import randint, seed

def gen_test(size, k, ans):
    print(size)
    print(k)
    ins = [randint(10, 20) for _ in range(k + 1)]
    print(sum(ins))
    x = [randint(-10, -1) for _ in range(size)]
    x[ans:ans + k + 1] = ins
    print(' '.join(map(str, x)))

gen_test(100000, 23456, 61023)
```

Величину числа `A` из условия передавать не будем. Достаточно того, что программа, генерирующая тест, посчитает эту сумму правильно, вставив в отрицательный массив положительный “кусоч”.

Если запустить снова

```
python gen_test.py > test4.dat
```

увидим, например, такой результат.

```
10
3
74
-4 -6 -3 -4 -2 20 17 18 19 -7
```

У вас на компьютере числа наверняка будут другими, но размер и другие параметры будут такие же. Можно запустить программу (командой `python A.py < test4.dat`), убедиться, что ответ равен 6 ( $5 + 1 = 6$ ).

Теперь, когда мы убедились, что всё работает, можно смело в вызове ставить большие числа и запускать вашу программу. Например, в последней строке написать что-нибудь вроде `gen_test(100000, 23456, 61023)`.

Программа всё сделала верно, но работала довольно долго. Вот так можно проверить, сколько именно:

```
from time import perf_counter

def solution_slow(x, k, A):
    i = 0
    while i + k < len(x):
        s = sum(x[i:i + k + 1])
        if s == A:
            return i + 1
        i += 1
    return 0

N = int(input())
k = int(input())
A = int(input())
x = list(map(int, input().split()))
start = perf_counter()
print(solution_slow(x, k, A))
print(perf_counter() - start)
```

На моём ноутбуке она выполнялась 14 секунд.

Можно попробовать сдать программу с этим медленным решением. Она пройдёт первые 10 (маленьких) тестов, но на 11 тесте ей не хватит секунды, чтобы обработать слишком большой массив.

Дело в том, что время работы такой программы зависит и от  $N$  (надо проверить все  $N - k$  положений окна) и от  $k$  (надо посчитать сумму  $k + 1$  элемента на каждой итерации). Получается произведение этих величин:  $(N - k) \cdot (k + 1) = Nk - k^2 + N - k$ . Понятно, что нужный кусок массива может встретиться быстрее, но нас интересует наихудший возможный случай.

Так как  $k \leq N$ , то основной вклад во время работы делает первое слагаемое. Скажем, при  $N = 10^5$ ,  $k = 10^3$  имеем:

$$Nk = 10^8, k^2 = 10^6$$

Остальные слагаемые совсем малы по сравнению с этими величинами.

Кстати, то, что вычисление суммы за нас делает встроенная функция `sum` не меняет картины. Это всё равно цикл, пусть написанный на другом языке и очень эффективный.

Можно заметить, что внутренний цикл, который заново пересчитывает сумму всех элементов в окне, делает много лишней работы. Ведь в окне поменялись только крайние элементы. Надо из текущего значения суммы выкинуть первый элемент (крайний слева) и прибавить новый (справа).

Так что внутренний цикл не нужен вообще и количество операций, которое сделает наша программа, пропорционально  $N - k$ .

Получается такая программа:

```
from time import perf_counter

def solution_fast(x, k, A):
    i = 0
    s = sum(x[i:i + k + 1])
    if s == A:
        return 1
    while i + k + 1 < len(x):
        s = s - x[i] + x[i + k + 1]
        i += 1
        if s == A:
            return i + 1
    return 0

N = int(input())
k = int(input())
A = int(input())
x = list(map(int, input().split()))
start = perf_counter()
print(solution_fast(x, k, A))
print(perf_counter() - start)
```

На том же самом тесте , на котором мы проверяли медленное решение, она работает примерно в 1000 раз быстрее (меньше 0.02 сек).