Задание 27

https://yandex.ru/tutor/subject/problem/?problem_id=T4838

Дана последовательность **N** целых положительных чисел. Рассматриваются все пары элементов последовательности, находящихся на расстоянии не меньше **6** друг от друга (разница в индексах элементов должна быть **6** или более). Необходимо определить **максимальную сумму** такой **пары**.

Описание входных и выходных данных

В первой строке входных данных задаётся количество чисел **N** (**7<=N<=1000**). В каждой из последующих **N** строк записано одно натуральное число, не превышающее **10000**.

Пояснение для первого примера: из представленных восьми чисел можно составить три пары (1,9), (1,8) и (3,8) – максимальная сумма из этих пар равна 11.

```
8
1
3
5
4
6
7
9
8
```

Правильный ответ: `11`.

Решение

Чтобы решать на 2 балла такого рода задачи вы должны понимать, как использовать одномерный массив (список), логику операторов ветвления и работу циклов.

Чтобы успешно – на 4 балла – решить данную задачу нужно дополнительно понимать такое абстрактное понятие (структуру данных) в программировании как очередь и выработать для себя систему размышлений, потренировавшись с несколькими аналогичными задачами.

Приступаем к разбору.

Итак, в задаче нужно «*найти **максимальную** сумму пары чисел*».

Сразу определимся с названиями – у нас будет «левое» число в паре и «правое» число в паре.

Давайте для простоты размышлений пока не будем учитывать дополнительное условие «***находящихся на расстоянии не меньше 6 друг от друга***». Нас наталкивают на то, что нужно перебрать все сочетания чисел попарно. Шаги такого алгоритма:

– берём первое число в последовательности (это будет левое число в паре) и складываем его по очереди со всеми остальными (ищем правое число в паре) – так, последовательным перебором мы ищем максимальную пару для левого числа – правое число будем выбирать из последовательности начиная со второго в списке – первое не берём, так как само с собой не считается парой чисел;

Значит – из этой последовательности:

1	2	3	4	5

берём такие пары (1,2), (1,3), (1,4), (1,5)
для них максимум: 1+5=6

– потом берём второе число в последовательности и складываем по очереди со всеми остальными, начиная с третьего – перебираем пары для второго числа (второй со вторым не пара чисел, второй с первым мы уже проверили на прошлой итерации – там было первый со вторым – от перемены мест слагаемых сумма не меняется – аналогично и на последующих шагах: мы будем брать в качестве второго члена пары только те, которые стоят правее);

Из этой последовательности на втором шаге

1	2	3	4	5

берём только пары (2,3), (2,4), (2,5)
для них максимальная сумма: 2+5=7

– продолжаем дальше перебирать левые до предпоследнего элемента – для него найдётся только одна пара – с последним числом;

Из этой последовательности:

1	2	3	4	5

на последнем шаге алгоритма берём пару (4,5)
максимальная сумма для всего списка 4+5=9
нашлась только на последней итерации алгоритма

– чтобы алгоритм работал следует хранить временный максимум в специальной переменной, каждую новую пару сравниваем с этой переменной – после перебора всех пар в ней окажется максимум из сумм всех пар.

Давайте реализуем «наивный алгоритм»:

```
n = int(input())  # кол-во элементов

arr = [0]*n  # инициируем список длиной n
for i in range(n):  #  перебираем последовательность
    arr[i] = int(input())  # сохраняем элементы в список

mx = 0  #  переменная для хранения максимума
for i in range(n-1):  # перебираем левые члены пар чисел
    for j in range(i + 1, n):  # для каждого i перебираем правые члены
        if arr[i] + arr[j] > mx:  # если сумма пары больше
            mx = arr[i] + arr[j]  # то её сохраняем

print(mx)  # выводим ответ
```

Если на вход этой программы подать следующую последовательность:

```
5
1
2
3
4
5
```

то получим в ответе **9**.

В программировании такого рода алгоритмы называют «**наивными**» – они работают по принципу как вижу, так и делаю. Наивные алгоритмы не учитывают возможностей языков программирования и структур для хранения данных, которые могут оптимизировать затраты времени и памяти. В частности, в приведённом алгоритме, так называемая, сложность (имеются ввиду затраты времени на поиск ответа) считается равной N^2, где N – это количество элементов в последовательности. Так получается, потому что алгоритм в процессе поиска должен перебрать в качестве первого члена пары все элементы последовательности (до предпоследнего) и для каждого из них перебрать все остальные в качестве второго члена пары. Единственное наше усовершенствование, что мы не берем в качестве второго члена те, которые были рассмотрены ранее. По этой причине, на самом деле, число итераций (повторений основных действий) алгоритма меньше, чем N^2, а равно (N–1)*N/2 – тем не менее, в этом случае, говорят, что сложность решения растёт квадратично при росте N.

За это, достаточно простое для понимания, решение вы получаете два балла. Если вам нужно больше, то следует отказаться от:

1) хранения всех элементов последовательности – у нас это список:

```
arr = [0]*n  # инициируем список длиной n
```

2) и уменьшить квадратичную сложность поиска ответа.

Обсудим пример (это *пример #2*) как можно уменьшить затраты памяти. Пусть на вход подаётся последовательность чисел:

```
5
1
2
3
4
5
```

и нужно найти в ней максимум. Вовсе не обязательно сначала их все сохранять в массив (список) и только потом его перебирать для поиска максимума – можно считывать поэлементно и сразу сравнивать, сохраняя только текущий максимум:

```
n = int(input())  # кол-во элементов

mx = 0
for i in range(n):  # перебираем последовательность
    elm = int(input())  # берём текущий элемент
    if elm > mx:  # если он больше
        mx = elm  # он новый максимум

print(mx)  # выводим ответ
```

Аналогичный подход будем использовать при решении задачи 27, но немного чисел последовательности всё-таки хранить будем – это связано как раз с дополнительным условием в задаче – расстояние между элементами должно быть «**не меньше 6 друг от друга**». При последовательном считывании элементов мы будем двигаться вдоль последовательности слева направо и, каждый раз, считывая очередной элемент, будем считать его правым и сравнивать суммы только для тех элементов слева, которые удовлетворяют условию «расстояние 6 и более».

Пусть будут вводиться последовательно числа от 1 до 9, тогда при последовательном (по одному элементу за раз) вводе шести подряд первых элементов ни один из них мы ещё не имеем права складывать с другим – напомню, что каждое новое введённое число мы будем рассматривать как правое в паре – числа от 1 до 6 включительно не могут иметь пару с левым от них числом, находящимся на расстоянии 6 и более – таких чисел попросту нет:

1	2	3	4	5	6	7	8	9

И только при вводе числа 7 мы, наконец-то, получаем возможность взять число слева от него на удалении 6 – это число 1 – его и сохраняем в нашу переменную **mx**, которая предназначена для хранения текущего максимума:

1	2	3	4	5	6	7	8	9

В этой позиции текущая максимальная сумма в паре чисел равна 8=1+7.

Сдвинемся ещё на один шаг вправо, получим следующее правое число – число 8, в левой части расширятся возможность доступных парных левых к нему – это или 1 или 2 – оба этих элемента на расстоянии 6 и более:

1	2	3	4	5	6	7	8	9

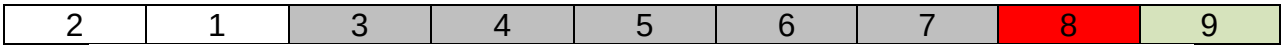
Серым фоном обозначены элементы, не участвующие в рассмотрении на текущем шаге алгоритма, тут парный максимум это 2+8=10.

На этом этапе разработки алгоритма важно осознать два момента:

1) элементы, которые на текущем шаге не участвуют в рассмотрении (обозначены серым фоном), нужно где-то хранить (пусть это будет массив **arr**) до тех пор, пока они не войдут в допустимую зону слева – нам нужен массив (список) длиной **6** (само это расстояние будем хранить в переменной **p** – от англ. pass – пройти, миновать – идентификатор pass не использую в программе, так как это зарезервированное Питоном слово – есть такая команда pass);

2) нам нужно сравнивать новый справа элемент (на текущем шаге это **8**) не просто с последним слева (на текущем шаге это **2**), а со всеми элементами белой зоны из левой части.

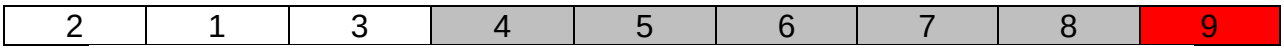
По поводу второго пункта – вот вариант, который показывает, что, если сравнивать только с последним, то ответ может быть и неправильным:



Если брать только последний левый, то сумма в паре будет 1+8=9, но максимальная сумма в паре это 2+8=10.

Конечно, мы не будем на каждом шаге (при получении нового правого) перебирать всю левую (белую) часть полностью – мы поступим по аналогии с примером #2 – просто будем в некоторой переменной **max_left** накапливать «левый» максимум по ходу (по мере накопления белой зоны).

Итак, у нас всегда будет известен текущий максимум в левой части списка, значит для текущего нового правого мы всегда (за одну операцию без дополнительного цикла) можем дать ответ какая именно максимальная сумма в паре чисел в условиях ограничения по расстоянию, даже когда дойдём до последнего элемента:



В итоге, для последнего элемента максимальная парная сумма будет 3+9=12.

Надо только как-то организовать доставание слева элемента из серой части – из нашего короткого массива **arr**, предназначенного для временного хранения элементов в запрещённой зоне.

Например, можно сделать так:

1) при поступлении нового справа элемента **num** берём крайний левый из серой зоны и отдаём в левую (белую) часть – там не надо хранить массив, там нужно только сравнить с текущим белым максимумом и выбрать новый текущий белый максимум;
2) после доставания слева из серой зоны одного элемента все члены массива **arr** нужно сдвинуть влево по массиву на 1 шаг и в крайнюю справа позицию серой зоны положить полученный новый **num**.

Рассмотрите программу (правильную и эффективную, как по памяти, так и по времени), реализующую описанный алгоритм (на вход подавайте последовательность длиной не менее 7):

```
p = 6  # по условию задачи
n = int(input())
arr = [0] * p  # список для хранения серой зоны
for i in range(p):  # вводим серую зону в начале списка
    arr[i] = int(input())

max_left = 0  # максимум левой части
max_pair = 0  # максимальная пара
pos = 0
for i in range(p, n):  # перебираем оставшиеся
    num = int(input())  # продолжаем вводить элементы справа
    if arr[0] > max_left:  # узнаём новый текущий максимум слева
        max_left = arr[0]
```

```
    if max_left + num > max_pair:  # узнаём новую максимальную пару
        max_pair = max_left + num
    for j in range(p-1):  # сдвигаем элементы списка arr влево
        arr[j] = arr[j+1]
    arr[p-1] = num  # правый элемент добавляем в серую зону справа

print(max_pair)  # выводим сумму максимальной пары
```

У этого подхода есть небольшой недостаток: сдвиг содержимого массива серой зоны влево на один шаг нужно будет сделать **(N–p)** раз, но это не снижает итоговую оценку при сдаче ЕГЭ.

Возможное улучшение алгоритма состоит в том, чтобы не сдвигать влево каждый раз элементы массива **arr**, а просто менять циклически номер позиции, откуда забираем элемент в левую (белую) часть, заменяя его новым справа элементом. Циклический сдвиг позиции в массиве (вместо сдвига всех элементов массива **arr**) позволяет ещё в три раза сократить работу алгоритма. Этот момент можно рассмотреть в индивидуальном порядке или реализуйте его самостоятельно.