

К. Ю. Поляков

Санкт-Петербургский государственный морской технический университет;
школа № 163, Санкт-Петербург, Россия

ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ В ЗАДАЧАХ ОБРАБОТКИ ПОСЛЕДОВАТЕЛЬНОСТЕЙ ЕГЭ ПО ИНФОРМАТИКЕ

Аннотация

В статье рассматривается использование метода динамического программирования для решения заданий на обработку последовательностей в контрольно-измерительных материалах (КИМ) ЕГЭ по информатике. Предлагается серия типовых задач нарастающей сложности, которая может быть использована для подготовки учащихся к выполнению заданий ЕГЭ по информатике и решению олимпиадных задач по программированию.

Ключевые слова: динамическое программирование, информатика, ЕГЭ, КИМ.

DOI: 10.32517/2221-1993-2020-19-5-55-63

1. Введение

Задание 27 в текущей версии КИМ ЕГЭ по информатике — это традиционная «серая лошадка» — неизвестная заранее задача, предусматривающая написание учащимся программы длиной в несколько десятков операторов [4]. Задания 27 в КИМ последних лет, как правило, связаны с обработкой последовательности чисел, причем для полного решения задачи (на 4 балла) эти данные нельзя сохранять в массиве или в другой структуре данных. Требуется найти некоторую характеристику входной последовательности, например, пару значений с максимальной суммой или количество пар с суммой, кратной 13.

Алгоритмы решения задач этого типа чаще всего основаны на использовании формул комбинаторики или применении динамического программирования. В настоящей статье мы подробно рассмотрим второй из этих подходов на примере решения серии типовых задач возрастающей сложности.

2. Динамическое программирование

Термин «динамическое программирование» ввел в науку американский математик Ричард Беллман [2].

Исходное значение этого термина имело весьма отдаленное отношение к программированию на компьютерах. Монография Беллмана [2] — это серьезная математическая работа, посвященная оптимизации непрерывных и дискретных многошаговых процессов.

В компьютерных науках **динамическим программированием** называют решение исходной задачи на основе известных решений задач того же типа, но меньшего размера. Под **размером задачи** понимают число N или набор чисел, которые определяют размер исходных данных.

Для использования метода динамического программирования необходимо, чтобы:

- 1) решение задачи имело вложенную структуру, т. е. задача размера N могла быть легко решена при известных решениях задач меньшего размера (от 1 до $N - 1$);
- 2) решения всех задач меньшего размера, необходимых для построения решения задачи размера N , сохранялись в памяти.

Таким образом, для решения задачи размера N требуется предварительно решить все задачи меньшего размера и сохранить их решения.

Р. Беллман исследовал задачи с большим числом варьируемых переменных, в которых количество «за-

Контактная информация

Поляков Константин Юрьевич, доктор тех. наук, профессор, Санкт-Петербургский государственный морской технический университет; адрес: 190121, Россия, г. Санкт-Петербург, ул. Лоцманская, д. 3; учитель информатики, школа № 163, Санкт-Петербург; адрес: 191015, Россия, г. Санкт-Петербург, ул. Кирочная, д. 54; e-mail: kpolyakov@mail.ru

K. Yu. Polyakov

Saint-Petersburg State Marine Technical University;
School 163, Saint-Petersburg, Russia

DYNAMIC PROGRAMMING IN PROCESSING SEQUENCES IN TASKS IN THE INFORMATICS UNIFIED STATE EXAM

Abstract

The article presents the use of the dynamic programming technique for solving tasks on processing sequences included in test materials for the Unified State Exam on informatics. A series of typical tasks of increasing complexity is presented, which can be used for preparing students for solving tasks of Unified State Exam and programming contests.

Keywords: dynamic programming, informatics, Unified State Exam, test materials.

дач меньшего размера» катастрофически росло при росте N . Это привело к появлению выражения «проклятие размерности»: как правило, при использовании динамического программирования в сложных задачах критическим ресурсом становится память компьютера, необходимая для хранения промежуточных результатов.

К счастью, в задачах, предлагаемых в КИМ ЕГЭ по информатике, не нужно хранить в памяти решения *всех* задач меньшего размера. Чаще всего решение задачи размера N может быть получено непосредственно из известного решения задачи размера $N-1$, что позволяет обойтись для хранения такого решения несколькими переменными или массивом, размер которого не зависит от N .

3. Простые задачи

Начнем с самых простых задач, алгоритмы решения которых, казалось бы, не требуют использования динамического программирования, но могут быть интерпретированы таким образом.

Первая часть условия всех задач, рассмотренных далее, одинакова:

С клавиатуры вводится натуральное число N ($1 \leq N \leq 10\,000$), а затем в следующих N строках — N целых чисел, по одному в каждой строке.

Конкретное содержание задачи уточняется в каждом случае.

Решения будут приводиться на языке Python версии 3.

Шаблон типовой программы на языке Python, использующей метод динамического программирования для обработки последовательностей, выглядит так:

```
N = int(input())
# инициализация
for k in range(N):
    x = int(input())
    # обработка значения x
print( ... ) # вывод результата
```

Задача 1.

Требуется вывести одно целое число — сумму всех элементов последовательности.

Решение.

Попробуем решить эту задачу методом динамического программирования. Обозначим сумму первых k элементов последовательности (*частичную сумму*) через S_k . Тогда рекуррентная формула, выражающая S_k через предыдущие значения S_i ($i < k$), имеет вид:

$$S_k = S_{k-1} + x,$$

где x — очередное число, прочитанное из входного потока.

«Наивное» решение задачи предполагает использование массива S , в котором будут храниться все предыдущие значения частичных сумм. Массив содержит $N+1$ элементов для хранения всех частичных сумм, включая S_N (эту последнюю сумму и нужно вывести на экран).

Инициализируем массив нулями:

```
S = [0] * (N+1)
```

Прочитав очередное значение из входного потока в переменную x , вычисляем новую частичную сумму:

```
S[k+1] = S[k] + x
```

Здесь учтено, что для первого значения x переменная цикла k равна нулю (см. шаблон программы), т. е. порядковый номер значения x в последовательности на единицу больше, чем k .

Приведем полную программу:

```
N = int(input())
S = [0] * (N+1)
for k in range(N):
    x = int(input())
    S[k+1] = S[k] + x
print( S[N] )
```

Здесь и далее ключевые операторы в программах выделены жирным шрифтом.

Это решение, которое не зря названо «наивным», далеко не оптимально, но иллюстрирует основную идею динамического программирования: найти решение задачи размера N через готовые решения задач размеров i ($i < N$), которые хранятся в памяти.

В данном случае нет необходимости использовать массив, так как значение S_k для любого k зависит только от предыдущего значения — S_{k-1} . Поэтому массив можно заменить всего одной переменной S , в которой хранится текущая частичная сумма.

Получается следующее решение, которое обычно и изучают в школьном курсе информатики:

```
N = int(input())
S = 0
for k in range(N):
    x = int(input())
    S += x
print( S )
```

Задача 2.

Требуется вывести одно целое число — количество отрицательных элементов последовательности.

Решение.

Обозначим через Q_k (от *англ.* quantity — количество) количество отрицательных значений среди первых k элементов последовательности. Для того чтобы применить динамическое программирование, нужно получить рекуррентную формулу, выражающую Q_k через решения задач меньшего размера, т. е. Q_i ($i < k$). В данном случае эта формула включает два варианта, выбор зависит от знака x :

$$Q_k = \begin{cases} Q_{k-1}, & x \geq 0; \\ Q_{k-1} + 1, & x < 0. \end{cases}$$

Как и в задаче 1, выделяем массив и заполняем его в цикле в соответствии с полученной формулой:

```
N = int(input())
Q = [0] * (N+1)
for k in range(N):
    x = int(input())
    if x < 0:
        Q[k+1] = Q[k] + 1
    else:
        Q[k+1] = Q[k]
print( Q[N] )
```

Здесь, как и в предыдущей задаче, массив не нужен: следующее значение Q вычисляется на основании всего одного предыдущего. Поэтому можно заменить массив одной переменной, получив «классическое» решение:

```
N = int(input())
Q = 0
for k in range(N):
    x = int(input())
    if x < 0:
        Q += 1
print(Q)
```

Задача 3.

Требуется вывести одно вещественное число — среднее арифметическое всех отрицательных элементов последовательности.

Решение.

Стандартное «школьное» решение сводится к тому, чтобы найти сумму отрицательных элементов последовательности и разделить ее на количество таких элементов. Для этого можно объединить приемы, использованные при решении задач 1 и 2.

Но для нас более интересен другой вариант, когда на каждой итерации цикла будет вычисляться среднее арифметическое A_k (от *англ.* average — среднее) всех отрицательных элементов среди первых k элементов последовательности.

Пусть S_k и Q_k — сумма и количество отрицательных элементов среди первых k элементов последовательности. Тогда по определению:

$$A_k = \frac{S_k}{Q_k}.$$

Теперь нужно вывести формулу, которая связывает среднее арифметическое A_k с предыдущими значениями этой величины, т. е. с A_i ($i < k$). Предположим, что мы прочитали из входного потока значение $x < 0$. Тогда $S_k = S_{k-1} + x = A_{k-1} Q_{k-1} + x$ и $Q_k = Q_{k-1} + 1$. Получаем:

$$A_k = \begin{cases} A_{k-1}, & x \geq 0; \\ \frac{A_{k-1}Q_{k-1} + x}{Q_{k-1} + 1}, & x < 0. \end{cases}$$

Таким образом, для вычисления A_k с помощью этой формулы нам нужно знать не только A_{k-1} , но и Q_{k-1} . В «наивном» решении придется использовать два массива:

```
N = int(input())
A = [0] * (N+1)
Q = [0] * (N+1)
for k in range(N):
    x = int(input())
    if x < 0:
        A[k+1] = (A[k]*Q[k] + x) / (Q[k] + 1)
        Q[k+1] = Q[k] + 1
    else:
        A[k+1] = A[k]
        Q[k+1] = Q[k]
print(A[N])
```

Заметим, что в том случае, когда в последовательности нет отрицательных чисел, программа выводит 0. Ошибки «деление на ноль» не происходит, потому что

деление выполняется при получении отрицательного числа на величину $Q_k + 1$, которая всегда больше нуля.

Поскольку на каждой итерации используются только предыдущие элементы массивов A и Q , можно заменить эти массивы простыми переменными:

```
N = int(input())
A, Q = 0, 0
for k in range(N):
    x = int(input())
    if x < 0:
        A = (A*Q + x) / (Q + 1)
        Q += 1
print(A)
```

Напомним, что при использовании традиционного алгоритма в цикле определяются значения S_N и Q_N , а затем, при $Q_N > 0$, среднее A_N вычисляется как их отношение.

4. Задачи на пары чисел

В демонстрационных КИМ ЕГЭ по информатике последних лет [4] довольно часто встречаются задачи, в которых необходимо работать с парами чисел, обладающими определенными свойствами. Рассмотрим несколько таких задач.

Далее везде предполагается, что $N \geq 2$ и все числа входной последовательности — целые и положительные. Пары, отличающиеся только порядком перечисления элементов, считаются одинаковыми. Это значит, что в последовательности $\{a_i\}$ пары (a_i, a_j) и (a_j, a_i) при $i \neq j$ нужно учитывать как одну пару. Пары (a_i, a_i) , которые каждое число образует само с собой, не учитываются.

Задача 4.

Требуется вывести одно неотрицательное число — количество пар с четной суммой, образованных различными элементами последовательности.

Решение.

Чтобы не учитывать одну и ту же пару дважды, будем рассматривать только упорядоченные пары (a_i, a_j) , для которых $i < j$.

В данной задаче очередное число, прочитанное из входного потока, может образовывать пары с *любым* из предыдущих элементов последовательности. Будем говорить, что все предыдущие элементы принадлежат *множеству выбора пары* для следующего элемента.

Подходящими будем называть пары, которые удовлетворяют условиям задачи. В этой задаче подходящими будут те пары, для которых сумма элементов четна.

Обозначим через Q_k количество подходящих пар для подпоследовательности, состоящей из первых k элементов полной последовательности. Если очередное полученное число x четно, оно образует подходящие пары (с четной суммой) только с предыдущими *четными* элементами последовательности, а если x нечетно, то оно образует подходящие пары только с предыдущими *нечетными* элементами. Поэтому для использования метода динамического программирования нам нужно хранить на каждой итерации количество предыдущих четных чисел E_{k-1} (от *англ.* even — четное) и количество

предыдущих нечетных чисел O_{k-1} (от *англ.* odd — нечетное). Тогда рекуррентная формула для вычисления Q_k может быть записана в виде:

$$Q_k = \begin{cases} Q_{k-1} + E_{k-1}, & x \text{ — четное;} \\ Q_{k-1} + O_{k-1}, & x \text{ — нечетное.} \end{cases}$$

Обновление E_k и O_k выполняется по формулам:

$$E_k = \begin{cases} E_{k-1} + 1, & x \text{ — четное;} \\ E_{k-1}, & x \text{ — нечетное.} \end{cases}$$

$$O_k = \begin{cases} O_{k-1}, & x \text{ — четное;} \\ O_{k-1} + 1, & x \text{ — нечетное.} \end{cases}$$

Применяя эти формулы, мы готовим данные для вычисления значения Q_{k+1} и таким образом добавляем x к множеству выбора пары для следующего элемента.

Поскольку для вычисления Q_k , E_k и O_k используются только предыдущие элементы последовательностей — Q_{k-1} , E_{k-1} и O_{k-1} , массивы здесь не требуются и можно обойтись тремя переменными. В блоке инициализации присваиваем им нулевые значения. После ввода очередного значения x из входного потока нужно обновить Q , а затем обновить E или O — в зависимости от четности x :

```
N = int(input())
Q, E, O = 0, 0, 0
for k in range(N):
    x = int(input())
    if x % 2 == 0:
        Q += E
        E += 1
    else:
        Q += O
        O += 1
print(Q)
```

Так как любое число обязательно либо четное, либо нечетное и на каждой итерации известно общее количество предыдущих элементов последовательности — оно хранится в переменной k , в программе можно заменить переменную O на $k - E$:

```
N = int(input())
Q, E = 0, 0
for k in range(N):
    x = int(input())
    if x % 2 == 0:
        Q += E
        E += 1
    else:
        Q += k - E
print(Q)
```

Задача 5 (Д. Ф. Муфаззалов).

Требуется вывести одно неотрицательное число — количество пар с суммой, равной $S = 20$, образованных различными элементами последовательности.

Решение.

Применим общий подход: получив из входного потока новое значение x , определим, сколько подходящих пар оно может образовать с предыдущими элементами последовательности. Обозначим через Q_k количество подходящих пар в подпоследовательности, состоящей из первых k чисел полной последовательности.

Напомним, что по условию (см. начало этого раздела) все элементы последовательности целые и положительные. Поэтому числа, большие или равные S , не могут образовать ни одной подходящей пары.

Если число x меньше S , оно образует подходящие пары со всеми предыдущими элементами, равными $S - x$. Следовательно, нам нужно хранить массив значений D , где $D[i]$ — это количество предыдущих элементов последовательности, равных i ($1 \leq i < S$). Например, после обработки последовательности 1, 2, 3, 4, 1, 2, 3, 1, 2, 1, 1 массив D имеет вид:

i	0	1	2	3	4	5	6	...	$S-2$	$S-1$
$D[i]$	0	5	3	2	1	0	0	...	0	0

В этой последовательности пять раз встретилось число 1, три раза — число 2, два раза — число 3 и один раз — число 4.

На этапе инициализации значение переменной Q и все элементы массива D обнуляются. Они обновляются на очередной итерации только тогда, когда из входного потока получено число x , которое меньше S :

```
S = 20
N = int(input())
D = [0]*S
Q = 0
for k in range(N):
    x = int(input())
    if x < S:
        Q += D[S-x]
        D[x] += 1
print(Q)
```

Задача 6.

Требуется вывести одно неотрицательное число — количество пар с суммой, меньшей или равной $S = 20$, образованных различными элементами последовательности.

Решение.

Будем использовать обозначения, введенные при решении задачи 5. В отличие от нее, в этой задаче подходящими считаются пары с суммой, не только равной, но и меньшей, чем S . Поэтому число $x < S$, полученное из входного потока на очередной итерации, образует пары со всеми предыдущими числами в диапазоне $[1, S - x]$. Следовательно, в программе потребуется суммирование элементов массива D от $D[1]$ до $D[S - x]$ включительно:

```
S = 20
N = int(input())
D = [0]*S
Q = 0
for k in range(N):
    x = int(input())
    if x < S:
        for i in range(1, S-x+1):
            Q += D[i]
        # или Q += sum(D[1:S-x+1])
        D[x] += 1
print(Q)
```

Несмотря на внутренний цикл, асимптотическая оценка сложности алгоритма по-прежнему равна

$O(N)$, так как число шагов внутреннего цикла от N не зависит.

Задача 7 (Д. Ф. Муфаззалов).

Требуется вывести одно неотрицательное число — количество пар с суммой, большей, чем $S = 20$, образованных различными элементами последовательности.

Решение.

Можно свести эту задачу к предыдущей. Действительно, если мы нашли количество Q всех пар с суммой, меньшей или равной S , то количество пар с суммой, большей S , легко найти как:

$$N \cdot (N - 1) / 2 - Q,$$

где $N \cdot (N - 1) / 2$ — количество всех возможных пар.

Но мы рассмотрим другой способ — будем вычислять желаемое значение непосредственно.

Пусть из входного потока прочитано значение x . Оно может образовать подходящие пары со всеми предыдущими числами, большими, чем $S - x$. Заметим, что числа, большие или равные S , образуют подходящие пары со всеми предыдущими числами (на k -й итерации получается $k - 1$ таких пар), а «маленькие» числа (меньшие S) — не со всеми. Для «маленького» числа x , введенного на k -й итерации цикла, количество подходящих пар вычисляется как:

$$k - 1 - (D[1] + D[2] + \dots + D[S - x]),$$

где $D[i]$ — количество предыдущих элементов последовательности, равных i .

Получаем следующую программу:

```
S = 20
N = int(input())
D = [0]*S
Q = 0
for k in range(N):
    x = int(input())
    Q += k
    if x < S:
        for i in range(1, S-x+1):
            Q -= D[i]
        D[x] += 1
print(Q)
```

Задача 8.

Рассматриваются все пары различных элементов последовательности, в которых хотя бы одно число кратно $F = 17$. Из всех таких пар нужно выбрать пару с максимальной суммой и вывести на экран оба элемента, образующие эту пару. Если таких пар несколько, то нужно вывести любую из них. Если таких пар нет, то нужно вывести два нуля.

Решение.

Поскольку в паре должно быть хотя бы одно число, делящееся на F , возможность образования пар с предыдущими элементами последовательности будет определяться делимостью* введенного значения x на F .

Если число x делится на F , оно может образовать пару с любым из предыдущих элементов. Так как нас интересует подходящая пара с максимальной суммой, нужно хранить максимальное из всех предыдущих чисел; для этого будем использовать переменную *maxAll*.

Если число x не делится на F , оно может образовать подходящую пару только с теми из предыдущих элементов, которые делятся на F . Поэтому будем также сохранять (в переменной *maxF*) максимальное из предыдущих значений, делящихся на F .

В переменных z_1 и z_2 хранятся два числа, которые на каждой итерации представляют собой текущее решение задачи — подходящую пару с максимальной на данный момент суммой. Если введенное значение x позволяет образовать подходящую пару с большей суммой, чем $z_1 + z_2$, значения этих переменных обновляются.

Приведем полную программу:

```
F = 17
N = int(input())
z1, z2 = 0, 0
maxAll, maxF = 0, 0
for k in range(N):
    x = int(input())
    if x % F == 0:
        if maxAll > 0 and maxAll+x > z1+z2:
            z1, z2 = maxAll, x
            maxF = max(maxF, x)
        else:
            if maxF > 0 and maxF+x > z1+z2:
                z1, z2 = maxF, x
            maxAll = max(maxAll, x)
    print(z1, z2)
```

При составлении пар (*maxAll*, x) и (*maxF*, x) нужно проверить, что число, которое выбрано в качестве пары для x , больше нуля. В противном случае мы учтем несуществующую пару (0, x), которая может оказаться парой с наибольшей суммой; при этом программа выведет неправильный ответ.

Если ни одной подходящей пары не найдено, программа выводит два нуля, которые были записаны в переменные z_1 и z_2 при инициализации.

Нужно отметить, что такое решение выглядит непривлекательно из-за вложенных условных операторов в теле цикла. Этот недостаток можно исправить, если объединить переменные *maxAll* и *maxF* в массив *maxPrev*, состоящий из двух элементов. Первый и второй по счету элементы этого массива (в Python они имеют индексы 0 и 1 соответственно) будут выполнять те же функции, что и переменные *maxAll* и *maxF* в приведенной выше программе.

Прочитав очередное значение x из входного потока, определяем индекс делимости *indF*:

```
indF = 0 if x % F == 0 else 1
```

Значение этой переменной определяет элементы последовательности, с которыми x может образовать пару. Если x делится на F (*indF* = 0), он образует пары со всеми предыдущими элементами. Числа, не делящиеся на F , для которых *indF* = 1, образуют пары только с элементами, которые делятся на F .

* Здесь и далее под делимостью везде понимается делимость нацело.

Полная программа на языке Python выглядит так:

```
F = 17
N = int(input())
z1, z2 = 0, 0
maxPrev = [0, 0]
for k in range(N):
    x = int(input())
    indF = 0 if x % F == 0 else 1
    if( maxPrev[indF] > 0 and
        maxPrev[indF]+x > z1+z2 ):
        z1, z2 = maxPrev[indF], x
    maxPrev[0] = max( maxPrev[0], x )
    maxPrev[1] = max( maxPrev[1], x*(1-indF) )
print( z1, z2 )
```

Обновление *maxPrev*[1] в конце тела цикла можно было бы выполнить так:

```
if indF == 0:
    maxPrev[1] = max( maxPrev[1], x )
```

Вместо этого в программе используется прием, позволяющий избавиться от условного оператора:

```
maxPrev[1] = max( maxPrev[1], x*(1-indF) )
```

При *indF* = 0 выражение $1 - indF$ равно 1, поэтому выбирается максимум из *maxPrev*[1] и *x*. При *indF* = 1 второй аргумент функции *max* равен $x \cdot (1 - indF) = 0$, и значение *maxPrev*[1] не изменится, поскольку оно гарантированно не меньше нуля.

Задача 9 [8].

Рассматриваются все пары различных элементов последовательности, разность которых делится на $R = 60$ и при этом хотя бы один элемент из пары больше $B = 80$. Требуется вывести одно неотрицательное число — количество пар, удовлетворяющих указанным условиям.

Решение.

Разность двух чисел будет делиться на R тогда и только тогда, когда эти числа при делении на R дают равные остатки.

Будем сохранять в массиве D количество чисел, дающих при делении на R все возможные остатки от 0 до $R - 1$. Значение элемента $D[i]$ — это количество предыдущих элементов последовательности, которые при делении на R дают в остатке i .

Если полученное из входного потока значение x меньше, чем B , оно может образовать пары не со всеми предыдущими числами с равными остатками, а только с теми, которые больше B . Поэтому нужен второй массив G , в котором элемент $G[i]$ — это количество предыдущих элементов последовательности, **больших** B , которые при делении на R дают в остатке i .

Получив из входного потока очередное значение x , определяем остаток от его деления на R :

```
r = x % R
```

Если $x > B$ (число x «большое»), то увеличиваем счетчик пар Q на величину $D[r]$ (x образует пары со всеми предыдущими элементами с тем же остатком). Если $x \leq B$, то увеличиваем Q на величину $G[r]$ (учитываем только «большие» предыдущие числа с тем же остатком).

Значение $D[r]$ увеличивается на 1 при каждой итерации цикла, а значение $G[r]$ — только тогда, когда $x > B$.

Приведем полную программу:

```
R, B = 60, 80
N = int(input())
Q = 0
D = [0]*R
G = [0]*R
for k in range(N):
    x = int(input())
    r = x % R
    if x > B:
        Q += D[r]
        G[r] += 1
    else:
        Q += G[r]
    D[r] += 1
print( Q )
```

Это решение можно преобразовать, используя прием, который применялся для предыдущей задачи.

Введем двумерный массив (матрицу) $D[i][r]$, в котором строки с индексами 0 и 1 — это массивы D и G из предыдущего решения соответственно. Инициализируем такой массив нулями:

```
D = [[0]*R, [0]*R]
```

Получив очередное значение x из входного потока, вычислим индекс строки *ind* следующим образом: индекс равен 0 при $x > B$ (значение x образует подходящие пары со всеми предыдущими элементами) и равен 1 при $x \leq B$ (значение x образует подходящие пары только с теми предыдущими элементами, которые больше B):

```
ind = 0 if x > B else 1
```

Получается следующая программа:

```
R, B = 60, 80
N = int(input())
Q = 0
D = [[0]*R, [0]*R]
for k in range(N):
    x = int(input())
    r = x % R
    ind = 0 if x > B else 1
    Q += D[ind][r]
    D[0][r] += 1
    D[1][r] += 1-ind
print( Q )
```

При обновлении $D[1][r]$ в конце каждой итерации применяется тот же прием, что и в предыдущей задаче: при *ind* = 1 значение $D[1][r]$ не обновляется, так как к нему прибавляется 0.

Задача 10 [3].

Рассматриваются все пары различных элементов последовательности, разность которых четна, и при этом хотя бы одно число из пары кратно $F = 17$. Из всех таких пар нужно выбрать пару с максимальной суммой и вывести на экран оба элемента, образующие эту пару. Если таких пар несколько, то нужно вывести любую из них. Если таких пар нет, то нужно вывести два нуля.

Решение.

По существу, эта задача является немного усложненным вариантом рассмотренной ранее **задачи 8**, в которую добавлено еще одно условие — четность разности чисел, образующих пару. Решение этой задачи с использованием комбинаторики становится очень громоздким из-за множества вложенных условных операторов, тогда как с помощью динамического программирования удастся получить простое и элегантное решение.

Применим общий подход: предположим, что очередное значение x введено из входного потока, и проверим, какие подходящие пары оно может образовать с предыдущими элементами.

С точки зрения формирования подходящих пар все элементы входной последовательности можно разделить на четыре группы:

- 1) делятся на 17, четные;
- 2) делятся на 17, нечетные;
- 3) не делятся на 17, четные;
- 4) не делятся на 17, нечетные.

Для получения четной разности можно составлять пару из двух четных или из двух нечетных чисел. Кроме того, хотя бы одно из них должно быть кратно 17, т. е. входить в группу 1 или в группу 2. Таким образом, допустимы следующие комбинации групп:

- 1 + (1 или 3);
- 2 + (2 или 4);
- 3 + 1;
- 4 + 2.

Поскольку нас интересует пара с максимальной суммой, нужно хранить максимумы предыдущих элементов последовательности, которые могут образовать пару с элементами каждой из четырех групп. Их удобно объединить в матрицу D размером 2×2 :

$D[0][0]$ — максимум из всех четных чисел (для образования пар с числами из группы 1);

$D[0][1]$ — максимум из всех нечетных чисел (для образования пар с числами из группы 2);

$D[1][0]$ — максимум из четных чисел, которые делятся на F (для образования пар с числами из группы 3);

$D[1][1]$ — максимум из нечетных чисел, которые делятся на F (для образования пар с числами из группы 4).

До выполнения цикла инициализируем массив D и переменные z_1 и z_2 , в которых будет храниться текущее решение (лучшая из предыдущих пар):

```
D = [[0, 0], [0, 0]]
z1, z2 = 0, 0
```

Прочитав из входного потока очередное значение x , определяем его индексы делимости на F и на 2:

```
indF = 0 if x % F == 0 else 1
ind2 = x % 2 # или ind2 = 0 if x % 2 == 0 else 1
```

Если $indF = 0$, число x образует наилучшую пару с максимальным из *всех* предыдущих чисел той же четности, а при $indF = 1$ — с максимальным из предыдущих чисел той же четности, которые делятся на F . Таким образом, наилучшая пара для x находится так:

```
if indF == 0:
    pair = D[0][ind2]
else:
    pair = D[1][ind2]
```

Поскольку значение переменной $indF$ может быть равно только 0 или 1, этот условный оператор преобразуется к виду:

```
pair = D[indF][ind2]
```

Если пара найдена ($pair > 0$) и сумма $pair + x$ больше, чем сумма $z_1 + z_2$ (текущее решение), переменные z_1 и z_2 обновляются:

```
if pair > 0 and pair+x > z1+z2:
    z1, z2 = pair, x
```

Приведем полное решение задачи:

```
F = 17
N = int(input())
D = [[0, 0], [0, 0]]
z1, z2 = 0, 0
for i in range(N):
    x = int(input())
    indF = 0 if x % F == 0 else 1
    ind2 = x % 2
    pair = D[indF][ind2]
    if pair > 0 and pair+x > z1+z2:
        z1, z2 = pair, x
    D[0][ind2] = max(D[0][ind2], x)
    D[1][ind2] = max(D[1][ind2], x*(1-indF))
print( z1, z2 )
```

Задача 11.

Рассматриваются все пары различных элементов последовательности, произведение которых кратно $F = 39$. Из всех таких пар нужно выбрать пару с максимальной суммой и вывести на экран оба элемента, образующие эту пару. Если таких пар несколько, то нужно вывести любую из них. Если таких пар нет, то нужно вывести два нуля.

Решение.

В отличие от предыдущих задач, здесь важно, что число F имеет всего два простых делителя — 3 и 13 (при большем количестве простых делителей решение существенно усложняется). Поэтому все числа с точки зрения выполнения условия задачи можно разделить на четыре группы:

- 1) делятся на 13, делятся на 3;
- 2) делятся на 13, не делятся на 3;
- 3) не делятся на 13, делятся на 3;
- 4) не делятся на 13, не делятся на 3.

Для того чтобы произведение двух чисел было кратно 39, необходимо и достаточно, чтобы оно содержало сомножители 3 и 13. Поэтому возможны следующие комбинации групп:

- 1 + (1 или 2 или 3 или 4);
- 2 + (1 или 3);
- 3 + (1 или 2);
- 4 + 1.

Будем хранить в двумерном массиве D размером 2×2 четыре групповых максимума для предыдущих элементов последовательности:

$D[0][0]$ — максимум из всех чисел (для образования пар с числами из группы 1);

$D[0][1]$ — максимум из чисел, которые делятся на 3 (для образования пар с числами из группы 2);

$D[1][0]$ — максимум из чисел, которые делятся на 13 (для образования пар с числами из группы 3);

$D[1][1]$ — максимум из чисел, которые делятся как на 13, так и на 3 (для образования пар с числами из группы 4).

Прочитав из входного потока очередное значение x , определяем его индексы делимости:

```
ind13 = 0 if x % 13 == 0 else 1
ind3 = 0 if x % 3 == 0 else 1
```

Теперь можно сразу найти наилучшую пару для x , используя данные из массива D :

```
pair = D[ind13][ind3]
```

Остается только проверить, не получилась ли у нас новая пара с максимальной суммой, и обновить массив D . Эти операции были подробно рассмотрены при решении предыдущих задач.

Полная программа выглядит так:

```
N = int(input())
D = [[0, 0], [0, 0]]
z1, z2 = 0, 0
for i in range(N):
    x = int(input())
    ind13 = 0 if x % 13 == 0 else 1
    ind3 = 0 if x % 3 == 0 else 1
    pair = D[ind13][ind3]
    if pair > 0 and pair+x > z1+z2:
        z1, z2 = pair, x
    D[0][0] = max(D[0][0], x)
    D[0][1] = max(D[0][1], x*(1-ind3))
    D[1][0] = max(D[1][0], x*(1-ind13))
    D[1][1] = max(D[1][1], x*(1-ind13)*(1-ind3))
print(z1, z2)
```

При обновлении элементов массива D использован прием с обнулением второго аргумента функции *max*, который был описан в решении **задачи 8**. В последней строке он применяется дважды, так что обновление $D[1][1]$ может произойти только тогда, когда $ind13 = ind3 = 0$, т. е. число x делится как на 13, так и на 3.

5. Использование очереди

Один из возможных путей усложнения задач на пары чисел состоит в том, чтобы задать минимальную разность порядковых номеров чисел, образующих пару, во входной последовательности.

Задача 12.

Рассматриваются все пары различных элементов последовательности, номера которых в последовательности отличаются не менее, чем на $L = 4$, а произведение кратно $F = 39$. Из всех таких пар нужно выбрать пару с максимальной суммой и вывести на экран оба элемента, образующие эту пару. Если таких пар несколько, то нужно вывести любую из них. Если таких пар нет, то нужно вывести два нуля. Предполагается, что $N > L$.

Решение.

При решении предыдущих задач мы предполагали, что множество выбора пары для очередного значения x — это все предыдущие элементы последовательности. В данной задаче это не так: согласно условию, очередной

элемент последовательности не может образовывать пары с ближайшими $L - 1$ предшествующими элементами. Правда, потом все они войдут во множество выбора пары, но до этого момента их нужно где-то сохранить. С этой целью будем использовать структуру данных, которая называется *очередью* [1].

Первые L элементов последовательности не могут образовывать пары с предыдущими элементами, поскольку для таких пар разность номеров элементов меньше, чем L . Поэтому сначала нужно просто заполнить очередь первыми L значениями из входного потока. На рисунке 1 эти числа обозначены как a, b, c и d . Стрелками показано направление движения элементов в очереди: новые значения добавляются в конец очереди (слева), а удаляются элементы из начала очереди (справа). Элементы, выходящие из очереди, добавляются к множеству выбора пары.

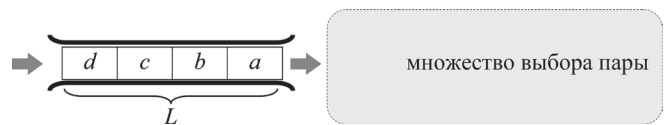


Рис. 1

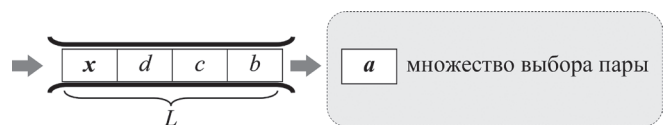


Рис. 2

На рисунке 2 показана ситуация, когда очередной элемент a вышел из очереди и добавлен к множеству выбора пары. Взамен в конец очереди добавлено значение x , полученное из входного потока. Легко проверить, что номера элементов x и a в исходной последовательности отличаются ровно на L , т. е. x может образовать пару с a (а также со всеми элементами, которые поступили раньше, чем a).

В языке Python для моделирования очереди можно использовать специальный класс *Queue* из модуля *queue* [9], но мы для простоты обойдемся простым списком (объектом класса *list*) с именем *queue*.

Шаблон программы, использующей очередь, имеет следующий вид:

```
L = ... # минимальная разность номеров
N = int(input())
queue = []
for i in range(L):
    queue.append( int(input()) )
# инициализация переменных и массивов
for k in range(N-L):
    a = первый элемент очереди
    # добавить a к множеству выбора пар
    x = int(input())
    # добавить x в конец очереди
    # обработка значения x
    print( ... ) # вывод результата
```

В начале программы создается пустой список (массив) *queue*, который сразу заполняется первыми L значениями из входного потока.

Основной цикл теперь выполняется $N - L$ раз, а не N , как было ранее, так как первые L элементов последовательности уже прочитаны и добавлены в очередь до начала этого цикла.

В основном цикле удаляем первый элемент из начала очереди (он имеет индекс 0) и записываем его в переменную a :

```
a = queue.pop(0)
```

Затем нужно добавить a к множеству выбора пар, так как следующий элемент последовательности (пока не прочитанный) уже может образовать с ним пару. Для этого определяем индексы делимости для значения a (см. решение предыдущей задачи) и обновляем массив D :

```
ind13 = 0 if a % 13 == 0 else 1
ind3 = 0 if a % 3 == 0 else 1
D[0][0] = max(D[0][0], a)
D[0][1] = max(D[0][1], a*(1-ind3))
D[1][0] = max(D[1][0], a*(1-ind13))
D[1][1] = max(D[1][1], a*(1-ind13)*(1-ind3))
```

Затем читаем новое значение из входного потока и добавляем его в конец очереди:

```
x = int(input())
queue.append(x)
```

На последнем этапе итерации вычисляем индексы делимости для значения x , находим лучшую пару для этого элемента и обновляем значения переменных z_1 и z_2 , если получилась пара с большей суммой:

```
ind13 = 0 if x % 13 == 0 else 1
ind3 = 0 if x % 3 == 0 else 1
pair = D[ind13][ind3]
if pair > 0 and pair+x > z1+z2:
    z1, z2 = pair, x
```

Заметим, что в решении предыдущей задачи мы добавляли значение x к множеству выбора пары сразу после его обработки, а здесь оно предварительно должно пройти через очередь, т. е. процесс добавления задерживается на L итераций цикла.

Полная программа выглядит следующим образом:

```
L = 4
N = int(input())
queue = []
for i in range(L):
    queue.append(int(input()))
D = [[0, 0], [0, 0]]
z1, z2 = 0, 0
for i in range(N-L):
    a = queue.pop(0)
    ind13 = 0 if a % 13 == 0 else 1
    ind3 = 0 if a % 3 == 0 else 1
    D[0][0] = max(D[0][0], a)
    D[0][1] = max(D[0][1], a*(1-ind3))
    D[1][0] = max(D[1][0], a*(1-ind13))
    D[1][1] = max(D[1][1], a*(1-ind13)*(1-ind3))
    x = int(input())
    queue.append(x)
    ind13 = 0 if x % 13 == 0 else 1
    ind3 = 0 if x % 3 == 0 else 1
    pair = D[ind13][ind3]
    if pair > 0 and pair+x > z1+z2:
        z1, z2 = pair, x
print(z1, z2)
```

Реализация очереди в других языках программирования может потребовать немного больше усилий, чем в Python. Поскольку размер очереди известен и не изменяется в ходе работы программы, в рекомендованных решениях подобных задач часто используется реализация очереди в виде статического массива [5, 6]. Для продвижения элементов в очереди в этом случае можно применить сдвиг элементов в дополнительном цикле [6] или технику, основанную на вычислении остатка от деления номера очередного элемента последовательности на L [5].

Шаблон программы и приемы работы с очередью, описанные в этом разделе, можно применять в любых задачах, где требуется обеспечить заданную минимальную разность между номерами элементов последовательности, образующих пару.

6. Заключение

В настоящей статье подробно рассмотрено использование динамического программирования при решении типовых заданий ЕГЭ по информатике на обработку последовательностей. Предложенная серия задач нарастающей сложности может быть использована при подготовке учащихся к выполнению заданий на программирование в КИМ ЕГЭ (решение задания 27 на максимальный балл), а также к решению олимпиадных задач.

Другие решения рассмотренных задач на различных языках программирования, в том числе основанные на переборе и формулах комбинаторики, можно найти в [7].

Автор благодарит своих коллег А. А. Богданова, Е. И. Поздняка, В. Н. Бабия, А. М. Кабанова, Д. Ф. Муфаззалова и Б. С. Михлина за полезные замечания и предложения, высказанные при обсуждении этой статьи.

Список использованных источников

1. Ахо А. В., Хопкрофт Д., Ульман Д. Д. Структуры данных и алгоритмы: пер. с англ. М.: Вильямс, 2000. 384 с.
2. Беллман Р. Динамическое программирование: пер. с англ. М.: Издательство иностранной литературы, 1960. 400 с.
3. Варианты ЕГЭ досрочного периода 2020 года. Информатика и ИКТ. <http://doc.fipi.ru/o-nas/novosti/varianty-yege-dosrochnogo-perioda-2020-goda/informatika-i-ikt.zip>
4. Демонстрации, спецификации, кодификаторы ЕГЭ. <https://fipi.ru/ege/demoversii-specifikacii-kodifikatory>
5. Демонстрационный вариант ЕГЭ 2015 г. Информатика и ИКТ, 11 класс. http://www.old.fipi.ru/sites/default/files/document/1415806485/inf_11_2015.zip
6. Демонстрационный вариант ЕГЭ 2016 г. Информатика и ИКТ, 11 класс. http://www.old.fipi.ru/sites/default/files/document/1447254044/inf_11_2016.zip
7. Поляков К. Ю. Разбор и решения задач 27 (C4). <http://kpolyakov.spb.ru/download/answ27-C4.doc>
8. Статград. Тренировочная работа № 5 по информатике. 11 класс. 2019–2020 гг. https://statgrad.org/delivery/file/Задание_ИИ11__29042020/
9. queue — A synchronized queue class. <https://docs.python.org/3/library/queue.html>