ГОАОУ «Центр поддержки одаренных детей «Стратегия» Липецкой области, к.т.н., доцент Шуйкова И.А. – директор, ООО «Яндекс. Технологии», Полднев А.В. – руководитель службы, МАОУ СамЛИТ г.о. Самара, Панькова М.Г. – учитель информатики, Первеев М.В. – студент Университета ИТМО

Линейные алгоритмы

Понятие линейных алгоритмов. Задача поиска максимума (минимума), второго максимума (минимума). Поиск пропущенного элемента в массиве, состоящего из чисел 1, 2,…,n. Поиск элемента в массиве, который встречается нечетное количество раз (при условии, что все остальные элементы встречаются четное количество раз). Поиск элемента массива, который встречается в массиве более половины раз. Поиск минимума среди элементов массива на окне, длиной k. Реализация очереди с поддержкой минимума. Определение подотрезка массива с заданной суммой. Сумма элементов массива в плавающем окне. Подотрезок массива с минимальной суммой.

Линейные алгоритмы представляют собой класс алгоритмов, позволяющих решать задачи за линейную сложность *O(n)*, где n – размерность задачи. Например, если дан массив, содержащий n элементов, где $1 \leq n \leq 10^5$, то квадратичная сложность алгоритма обработки не позволит выполнить задачу за 1 секунду. Алгоритмы линейной сложности применяются, в частности, для решения однопроходных задач, т.е. задач, которые обрабатывают данные при считывании без их сохранения. К ним, например, относятся алгоритмы поиска минимума (максимума), поиска минимума на отрезке длины k («на окне») в массиве.

Задачи поиска максимумов (минимумов)

Общая постановка задачи: Дан массив элементов. В процессе считывания данных необходимо решить задачу поиска максимума без сохранения данных.

Общее решение задачи:

Будем использовать переменную max_ans для хранения текущего максимума. В начале программы переменная max_ans инициализируется минимально возможным значением исходных данных. При просмотре каждого нового элемента будем сравнивать его с текущим значением переменной *max_ans*, и если элемент больше *max_ans*, то присваиваем переменной *max_ans* значение текущего элемента.

Задача 1

Дан массив из N элементов. В процессе считывания данных необходимо решить задачу поиска максимума и подсчет количества максимумов.

Решение

Введем дополнительную перемененную *cnt* для подсчета количества максимумов. Для решения этой задачи выделим 3 случая:

1. Текущий элемент равен максимуму: увеличиваем счетчик *cnt* на 1.
2. Текущий элемент больше максимума: изменяем значение *max_ans* на значение нового элемента и делаем значение счетчика *cnt* равное 1, т.к. мы встретили новый максимум первый раз.
3. Текущий элемент меньше максимума: значение счетчика *cnt* не изменяется.

Задача 2

Дан массив из N элементов. В процессе считывания данных, необходимо решить задачу поиска первого и второго максимума.

Есть два варианта постановки задачи, которые будут незначительно отличаться реализацией.

Задача 2.1

Дан массив из N ($N \geq 2$) элементов. В процессе считывания данных необходимо решить задачу поиска первого и второго максимума. Иначе говоря, если бы массив был упорядочен по убыванию, то это элементы, которые стоят на первом и втором месте.

ГОАОУ «Центр поддержки одаренных детей «Стратегия» Липецкой области, к.т.н., доцент Шуйкова И.А. – директор, ООО «Яндекс. Технологии», Полднев А.В. – руководитель службы, МАОУ СамЛИТ г.о. Самара, Панькова М.Г. – учитель информатики, Первеев М.В. – студент Университета ИТМО

Для решения данной задачи будем использовать переменную *first_max*, в которой хранится текущий первый максимум массива и переменную *second_max*, в которой будет хранится второй максимум массива. Изначально переменным *first_max* и *second_max* присваивается минимально возможные для данной задачи значения. При считывании текущего элемента массива будем пересчитывать максимумы: если значение считанного элемента больше либо равно первого максимума, то второй максимум получает значение первого максимума, а первый максимум заменяется на считанное значение. Если же считанное значение меньше, чем первый максимум, но больше либо равен второго максимума, то заменяем только второй максимум – он становится равен считанному значению.

C++
```
int first_max = -1e9;
int second_max = -1e9;
int n, val;
cin >> n;
for (int i = 0; i < n; ++i) {
    cin >> val;
    if (val >= first_max) {
        second_max = first_max;
        first_max = val;
    } else if (val > second_max) {
        second_max = val;
    }
}
```

Python
```
first_max = -1e9
second_max = -1e9

n = int(input())
for i in range(n):
    x = int(input())
    if x >= first_max:
        second_max = first_max
        first_max = x
    elif x > second_max:
        second_max = x
```

Задача 2.2

Дан массив из N ($N \geq 2$) элементов. Найти два самых больших различных элемента.

При решении этой задачи возможен случай, когда 2-й максимум не будет найден, если все элементы в массиве одинаковые.

Подумайте, какие изменения нужно внести в алгоритм решения Задачи 2.1, чтобы решить эту задачу.

Задача 3

Дана последовательность N целых положительных чисел. Рассматриваются все пары элементов последовательности, находящихся на расстоянии не меньше 6 (разница в индексах элементов должна быть 6 или более). Необходимо определить максимальную нечётную сумму такой пары. Если пар с нечётной суммой нет, ответ считается равным 0.

Описание входных и выходных данных

В первой строке входных данных задаётся количество чисел N ($6 \leq N \leq 1000$).

В каждой из последующих N строк записано одно натуральное число, не превышающее 10 000.

Пример входных данных	*Пример выходных данных*	*Комментарий*
8 1 3 5 4 6 7 9 8	11	Из восьми чисел (числа нумеруются с единицы) можно составить три пары, удовлетворяющие условию. Это будут элементы с индексами 1 и 7, 1 и 8, 2 и 8. Для заданного набора чисел получаем пары (1, 9), (1, 8), (3, 8). Суммы чисел в этих парах равны 10, 9, 11. Нечётных сумм – две, максимальная из них равна 11

.

Простое решение этой задачи.

```
n = int(input())
a = [int(input()) for i in range(n)]
max_sum = 0
for i in range(n - 6):
    for j in range(i + 6, n):
        if (a[i] + a[j]) % 2 == 1 and (a[i] + a[j]) > max_sum:
            max_sum = a[i] + a[j]
        print(i, j)
print(max_sum)
```

Очевидные минусы этого решения – обязательное хранение всех элементов и сложность - $O(n^2)$. Такое решение не является линейным.

Рассмотрим эффективное решение данной задачи. Обратим внимание, что улучшить ответ с помощью текущего элемента мы сможем, если мы найдем его сумму с каким-то из элементов, встреченным ранее (на 6 или более индексов) и дающим с текущим элементом нечетную максимальную сумму.

Для выполнения первого условия заметим, что нет необходимости хранить ВСЕ элементы, которые встречались раньше нашего, достаточно хранить максимальный, так как никакой другой не даст с нашим элементом бОльшую сумму. Для выполнения второго условия для нечетного элемента нужно рассматривать четные и наоборот.

Делаем вывод: обновить ответ с помощью текущего элемента мы сможем, если возьмем сумму текущего элемента и максимального из встреченных ранее с учетом четности.

Пример однопроходной задачи на подсчет элементов, удовлетворяющих условию

Задача 4

На вход программы поступает последовательность из N целых положительных чисел, все числа в последовательности различны. Рассматриваются все пары различных элементов последовательности (элементы пары не обязаны стоять в последовательности рядом, порядок элементов в паре не важен). Необходимо определить количество пар, для которых произведение элементов не кратно 14.

Описание входных и выходных данных

В первой строке входных данных задаётся количество чисел N ($1 \le N \le 1\ 000\ 000$). В каждой из последующих N строк записано одно натуральное число, не превышающее 10000. В качестве результата программа должна вывести одно число: количество пар, в которых произведение элементов не кратно 14.

Пример входных данных	*Пример выходных данных*	*Комментарий*
4 2 6 5 42	3	Из четырёх заданных чисел можно составить 6 попарных произведений: 2·6, 2·5, 2·42, 6·5, 6·42, 5·42. Из них на 14 не делятся 3 произведения (2·6, 2·5, 6·5).

Эффективное решение этой задачи не предполагает хранение входных данных. Заметим, что удовлетворяющие условиям пары можно составить из комбинаций чисел кратных 7 и кратных 2, кратных 14 и любых чисел. Будем сразу считать количество таких чисел: k7, k2, k14. Для получения ответа нужно посчитать все возможные варианты получения нужных сумм.

ГОАОУ «Центр поддержки одаренных детей «Стратегия» Липецкой области, к.т.н., доцент Шуйкова И.А. – директор, ООО «Яндекс. Технологии», Полднев А.В. – руководитель службы, МАОУ СамЛИТ г.о. Самара, Панькова М.Г. – учитель информатики, Первеев М.В. – студент Университета ИТМО

Поиск пропущенного значения в массиве

Дан массив чисел из *n* элементов, состоящий из чисел от 1 до *n*. На вход программы подается *n-1* число – это элементы исходного массива, причем каждое число массива встречается ровно один раз, но одно из чисел пропущено и его требуется найти. Например, ответом на входную последовательность чисел: 1, 5, 4, 2 является число 3.

Идея решения данной задачи заключается в том, что сумму всех элементов исходного массива можно вычислить с помощью формулы арифметической прогрессии. В процессе считывания элементов мы можем вычислить сумму данных элементов. Далее легко догадаться, как получить ответ к задаче.

Например, n = 10 и пропущен элемент 7. При вычислении $\sum_{i=1}^{n} i$ по формуле суммы членов арифметической прогрессии $\frac{n \times (n+1)}{2}$ получим 55. Если сложим все числа в массиве, то получим s = 1+2 +3 +4 +5 + 6 + 8 + 9 +10 = 48. Значение пропущенного элемента x = 55 – 48 = 7.

Задача о поиске элемента, который встречается в массиве нечетное количество раз, а все остальные - четное количество раз

Дан набор чисел, среди которых все, кроме одного, встречаются чётное число раз. Нужно найти число, встречающееся нечётное число раз в массиве. Например, для массива: 1, 3, 4, 3, 4, 1, 4 ответом будет являться число 4.

Для решения данной задачи потребуется логическая операция побитого исключающего ИЛИ – *xor*. Эта операция является отрицанием операции равносильности, а также сложением по модулю 2. Операция выполняется над каждым битом числа, например 4 xor 5 = 100 xor 101 = 001 = 1.
Выпишем интересные нам свойство *xor*:

(a xor b) xor b = a
b xor b = 0
a xor 0 = a

Используя свойства коммутативности и ассоциативности, можем представить xor всех значений массива как xor пар одинаковых значений: (a_1 xor a_1) xor (a_2 xor a_2) xor … xor (a_k xor a_k) xor b, где за b обозначим элемент, у которого не было пары. По описанным выше свойствам получим в каждой скобке значение 0, а выполнив 0 xor b получим b.

```
int b = 0;
for (int i = 0; i < n; ++i) {
   cin >> val;
   b ^= val;
}
```

В переменной b накапливается результат операции xor над элементами массива в процессе его считывания. По значению переменной b мы сделаем вывод о том, какое число встречается нечетное количество раз в массиве.

Задача о поиске элемента, который встречается в массиве из n элементов больше n/2 раз

Заметим, что если мы вычеркнем из последовательности два различных числа, то ответ задачи останется верным. Поэтому мы можем вычёркивать пары различных чисел до тех пор, пока все элементы не станут равными одному и тому же числу. Это число и будет искомым числом, которое встречается в массиве более половины раз.

Во время работы алгоритма на очередном префиксе информация о доминирующем числе может быть неверной, если в нём вообще нет доминирующего числа. Пример: числа [5, 5, 1, 1, 8] — доминирующим считается 8. Но легко доказать, что для любого префикса, где доминирующее число есть, ответ будет получаться верным.

Чтобы реализовать этот метод, заведём переменную, в которой будет храниться какой-то элемент последовательности, и счётчик — сколько копий этого элемента у нас просмотрено и пока не вычеркнуто. Когда мы считываем очередной элемент, возможны три варианта:

- счётчик равен нулю. Записываем элемент в переменную, увеличиваем счётчик на 1.
- элемент равен значению переменной. Увеличиваем счётчик на 1.
- элемент не равен значению переменной. Уменьшаем счётчик на 1.

После того, как мы просмотрим всю последовательность, в переменной окажется искомое число. Заметим, что данный алгоритм работает корректно только в том случае, если в массиве действительно есть элемент, который встречается больше половины раз

```
int cur, val, bal = 0;
for (int i = 0; i < n; ++i) {
    int val;
    cin >> val;
    if (bal == 0)
        cur = val, bal = 1;
    else if (cur == val)
        bal++;
    else
        bal--;
}
```

Подотрезок с заданной суммой

В задаче требуется найти подмассив массива положительных элементов. Подмассив должен состоять из элементов, сумма которых равна заданному числу. Так как массив состоит из положительных чисел, то можно двигать правую или левую границу в зависимости от того, какая сумма на данном отрезке: если она меньше, чем заданная сумма, то нужно добрать число справа, если она больше, то нужно сдвинуть левую границу, уменьшив тем самым сумму на отрезке. Данный алгоритм работает за O(n), так как каждый элемент мы посмотрим дважды: один раз правой границей, другой раз – левой.

В нашей реализации будем для каждого текущего значения R (правой границы) пытаться набрать сумму ровно k. Для этого будем двигать левую границу до тех пор, пока текущая сумма не станет меньше или равной k. Если для данного значения R не удалось получить сумму k, то будем рассматривать следующее R. Добавление элемента справа увеличит текущую сумму, и мы снова будем её уменьшать за счет левой границы, если в этом есть необходимость.

Метод решения данной задачи также называют методом двух указателей, так как мы оперируем двумя границами – указателями, которые двигаются в зависимости от ситуации определенным образом.

```
int l = 0;
long long sum = 0;
for (int r = 0; r < n; ++r) {
    sum += a[r];
    while (sum > k) {
        sum -= a[l];
        ++l;
    }
    if (sum == k) {
        cout << l + 1 << ' ' << r + 1;
        break;
    }
}
```

ГОАОУ «Центр поддержки одаренных детей «Стратегия» Липецкой области, к.т.н., доцент Шуйкова И.А. – директор, ООО «Яндекс. Технологии», Полднев А.В. – руководитель службы, МАОУ СамЛИТ г.о. Самара, Панькова М.Г. – учитель информатики, Первеев М.В. – студент Университета ИТМО

Сумма в плавающем окне

Дан набор чисел – элементы массива. Левая и правая граница запроса: L, R. В начале решения задачи L = R = 1. Поступают запросы: L++, R++. В задаче требуется найти сумму a[L] + … + a[R]. Идея решения задачи заключатся в том, что можно поддерживать текущую сумму. При увеличении правой границы нужно добавить к текущей сумме правый элемент. При увеличении левой границы нужно вычесть из суммы левый элемент.

```
vector<long long> a(n + 1); // исходный массив
for (int i = 1; i <= n; i++) {
    cin >> a[i];
}
int l = 1, r = 1;
long long sum = 0;
for (int i = 0; i < t; i++) { // обрабатываем запросы
    char c;
    cin >> c;
    if (c == 'R') {
        r++;
        sum += a[r];
        cout << sum << '\n';
    }
    if (c == 'L') {
        sum -= a[l];
        l++;
        cout << sum << '\n';
    }
}
```

Подотрезок с максимальной суммой

Дан массив произвольных чисел (возможно, отрицательных) — найти подотрезок с максимальной суммой. Максимизация a[L] + … + a[R] эквивалентна максимизации s[R] – s[L−1] (полезная идея: префиксные суммы помогли упростить максимизируемое выражение).

Можно решать задачу за квадратичную сложность при помощи перебора R и L. А можно для каждого очередного R находить наиболее оптимальный s[L−1] быстрее: при фиксированном R максимизация s[R] – s[L−1] эквивалентна минимизации s[L−1]. То есть, достаточно просто поддерживать текущий минимальный s[L−1].

Пройдем по всем R и параллельно будем поддерживать текущую минимальную префиксную сумму на отрезке [0, R-1]. Разница между s[R] и s[L-1] - минимальным значением на отрезке [0, R-1] и даст нам максимальную сумму на подотрезке, заканчивающимся в позиции R. Перебрав все позиции и взяв из них максимальное значение, мы и получим ответ.

Можно подсчет префиксных сумм так же добавить в линейное решение, тогда можно реализовать решение этой задачи без хранения всего массива.

s – текущая префиксная сумма от 0-го до i-ого элемента массива

smin – минимальная префиксная сумма на префиксе [0, i - 1].

curr и curl – указатели границ отрезка для ответа

```
for (int i = 0; i < n; ++i) {
     s += a[i];
     if (s - smin > ans) {
         ans = s - smin;      //обновление максимальной суммы
         curr = i;
         curl = cur + 1;
     }
     if (s < smin) {          //обновление левой границы отрезка
         cur = i;
         smin = s;
     }
 }
```

ГОАОУ «Центр поддержки одаренных детей «Стратегия» Липецкой области, к.т.н., доцент Шуйкова И.А. – директор, ООО «Яндекс. Технологии», Полднев А.В. – руководитель службы, МАОУ СамЛИТ г.о. Самара, Панькова М.Г. – учитель информатики, Первеев М.В. – студент Университета ИТМО

Очередь с поддержкой минимума

В данной задаче требуется реализовать очередь с поддержкой минимума. Для решения задачи необходимо уметь реализовывать стек с поддержкой минимума и очередь на двух стеках. Как реализовать стек с поддержкой минимума – необходимо хранить два значения: элемент и минимум, соответствующий его префиксу. Как реализовать очередь на двух стеках - заведем два стека: один для добавления и один для удаления. Если требуется удалить элемент, то посмотрим на стек для удаления. Если он не пуст, то просто удалим верхний элемент, иначе – перекинем все элементы из стека для добавления в стек удаления и уже потом удалим верхний.

Данный алгоритм работает за линейное время. Для того, чтобы это понять, проследим «путь» каждого элемента массива. Каждый элемент будет один раз добавлен в первый стек, один раз удален из него и добавлен во второй стек, и потом один раз удален из второго стека. Таким образом, на каждой элемент приходится константное число операций, а значит суммарно время работы линейное.

```
stack<pair<long long, long long>> add, del; // стек для добавления и удаления
void push(long long element) { // процедура добавления элемента
   if (add.empty())
         add.push({ element, element });
   else
         add.push({ element, min(element, add.top().second) });
}
void push_del(long long element) { //добавление в стек для удаления
   if (del.empty())
         del.push({ element, element });
   else
         del.push({ element, min(element, del.top().second) });
}
void pop() { // процедура удаления элемента из очереди
   if (del.empty())
         while (!add.empty())
               push_del(add.top().first), add.pop();
   del.pop();
}
Long long front() { // запрос на минимум
   if (!del.empty() && !add.empty())
         return min(del.top().second, add.top().second);
   else if (!del.empty())
         return del.top().second;
   else
         return add.top().second;
}
```

Минимум на окне – подотрезки элементов массива длиной k

Рассмотрим следующую задачу. Дан массив a, содержащий n элементов, где n удовлетворяет следующим ограничениям $1 \le n \le 10^5$. Требуется найти минимум на каждом окне длины k, $1 \le k \le n$, то есть среди элементов массива $[a_0, a_2, \dots, a_{k-1}]$, $[a_1, a_3, \dots, a_k]$, …, $[a_{n-k+1}, a_{n-k+1}, \dots, a_n]$.

Есть различные методы решения данной задачи, одно из возможных решений использует дек возрастающей подпоследовательности элементов массива. Дек будет устроен следующим образом: на текущем окне первый элемент в деке будет равен первому минимуму, второй – минимальному элементу, большему первого и т.д. Заведем два указателя L и R – границы рассматриваемого окна. Будем поддерживать такой инвариант: после каждого сдвига L и R минимум на окне от L до R лежит в первом элементе дека. Если при движении окна «уходит» минимум, то он удаляется из начала дека. При добавлении элемента в дек будем удалять из хвоста дека все элементы, строго большие текущего, после чего добавим текущий. Таким образом, в каждый момент времени у нас в деке хранится неубывающая последовательность из возможных минимумов.

Каждый элемент будет добавлен и удален ровно один раз, следовательно, получается линейная сложность алгоритма.

Разберем работу алгоритма на примере
Исходные данные: n = 7, k = 3, a = [4, 2, 7, 8, 1, 4, 0].
При заполнении дека его состояния будут следующими: q = [ ] – вначале дек пустой.
q = [4] – сначала мы положили в дек число 4 из исходного массива.
q = [2] – считав число 2, оно сравнивается со всеми числами, начиная с конца дека и если считанное число меньше текущего в деке, то большее удаляется до тех пор, пока в деке не останется число, большее считанного. Затем считанное число помещается в дек. В данном примере считанное число 2 было меньше числа 4, поэтому число 4 удалено из дека, а число 2 записано в дек.
q = [2, 7] – следующее состояние дека. Процесс продолжается далее, пока не будет считан весь массив.

```
void pb (int x){
    while (!q.empty() && x < q.back()) {
        q.pop_back();
    }
    q.push_back(x);
}

for (int i = 0; i < k; ++i) {
   pb(a[i]);
}
for (int r = k; r < n; ++r) {
    cout << q.front() <<'\n';
    if (a[r - k] == q.front())
        q.pop_front();
    pb(a[r]);
}
cout << q.front();
```

Используемые ресурсы

1. e-maxx. Ресурс с большой коллекцией описанных алгоритмов с примерами кода на C++. Сайт: https://e-maxx.ru/
2. Антти Лааксонен. Олимпиадное программирование. / пер. с англ. А.А. Слинкин – М. ДМК Пресс, 2018
3. Сайт дистанционной подготовки по информатике. http://informatics.msk.ru/