

Разбор задач

Задача 1 «Призы»

Решение задачи основано на классическом алгоритме поиска второго максимума в массиве. Будем поддерживать две переменные: текущий максимум в массиве и второй по величине элемент. Тогда при добавлении в рассмотрение очередного приза обновим при необходимости первый и второй максимум.

Приведем код решения на языке Си++.

```
scanf("%d", &n);
m1 = m2 = 0;
for (int i = 0; i < n; ++i) {
    int num;
    scanf("%d", &num);
    if (num > m1) {
        m2 = m1;
        m1 = num;
    } else if (num > m2) {
        m2 = num;
    }
    if (i > 0) {
        cout << m2 << " ";
    }
}
```

Альтернативное решение может базироваться на использовании встроенной в язык программирования структуры данных для поддержания упорядоченного множества. Приведем пример решения на языке Си++ с использованием структуры данных `std::multiset`. В множестве поддерживаются два максимальных элемента, каждый раз добавляется один элемент и удаляется минимальный элемент.

```
in >> n;
multiset<int> ma;
int x, y;
in >> x >> y;
ma.insert(x);
ma.insert(y);
for (int i = 1; i < n; i++) {
    out << *ma.begin() << " ";
```

```
in >> x;  
ma.insert(x);  
ma.erase(ma.begin());  
}
```

Сложность такого решения $O(n)$.

Частичные решения для первой подзадачи могут использовать квадратичную сортировку для поддержания массива отсортированным после добавления каждого элемента. Сложность такого решения $O(n^3)$.

Частичные решения для второй подзадачи могут использовать сортировку за $O(n \log n)$ для поддержания массива отсортированным после добавления каждого элемента. В частности, возможно использование встроенной в стандартную библиотеку сортировки. Время работы такого решения есть $O(n^2 \log n)$.

Альтернативным подходом может быть использование сортировки «вставками». Будем поддерживать массив отсортированным и каждый раз при добавлении очередного элемента будем «утапливать» его до необходимого места. Время работы такого решения есть $O(n^2)$.

Задача 2 «Космическое поселение»

Для полного решения этой задачи воспользуемся двоичным поиском. Заметим, что если можно установить защиту толщиной d , то можно установить и меньшую защиту.

Для проверки, что защиту данной толщины можно установить, переберем два варианта ориентации модулей. Пусть, например, модуль ориентирован так, что сторона длиной a ориентирована вдоль стороны поля длиной w . Тогда после установки защиты толщиной d вдоль этой стороны можно установить $w \operatorname{div} (a + 2d)$ модулей (при делении округлять следует вниз). Аналогично, вдоль другой стороны можно установить $h \operatorname{div} (b + 2d)$ модулей. В целом же на поле можно установить $(w \operatorname{div} (a + 2d)) \times (h \operatorname{div} (b + 2d))$ модулей. Если это число больше или равно n , то защиту толщиной d или больше установить можно, иначе нельзя.

Приведем код на Си++.

```
long long L = 0;
long long R = min(w, h) + 1;
while (L + 1 < R) {
    long long M = (L + R) / 2;
    long long ad = a + 2 * M;
    long long bd = b + 2 * M;
    long long ac = w / ad;
    long long bc = h / bd;
    if (ac * bc >= n) { L = M; } else { R = M; }
}
return L;
```

Отметим, следующий аспект. Хотя кажется, что при вычислении может произойти переполнение 64-битного типа, на самом деле это не так. В случае, если M больше ответа на задачу, переполнения не происходит, так как перемножаются числа, которые дают в произведении число, меньшее n . Если же M оказывается меньше ответа, то, благодаря структуре двоичного поиска, M меньше ответа не более чем вдвое, значит вычисляемое значение не превышает $4n$ и помещается в 64-битный тип данных.

Для решения подзадачи 1 достаточно перебрать все возможные значения толщины защиты линейным проходом.

Для решения подзадачи 2 требуется заметить, что при размещении модулей вдоль каждой стороны поля размещается не более n модулей, поэтому можно осуществить перебор, сколько модулей будет расположено вдоль стороны длиной w и проверить для такого количества модулей вдоль этой стороны, какая толщина защиты возможна.

Для решения подзадачи 3 можно развить эту мысль, заметив, что вдоль одной из сторон размещается не более \sqrt{n} модулей, а значит можно ускорить перебор.

Задача 3 «Странные строки»

Докажем сначала вспомогательное утверждение: строка z является странной, если для нее выполняются следующие свойства:

1. z содержит не более двух различных символов;
2. все одинаковые символы в z идут подряд.

Действительно, пусть, например, в строке содержатся два одинаковых символа a , между которыми находится еще один символ, отличный от a . Пусть a встречается в строке k раз. Тогда строка из k символов a является подпоследовательностью z , но не ее подстрокой.

Пусть теперь в z встречаются хотя бы три различных символа. По доказанному, одинаковые символы должны идти подряд. Но тогда, если первый символ строки a , а последний – b , то строка ab является подпоследовательностью z , но не ее подстрокой.

Наоборот, если строка z состоит из k символов a , после которых идет l символов b , то как $W(z)$, так и $Y(z)$ состоит из всех строк, в которых сначала идет не более k символов a , а затем не более l символов b .

Таким образом, для определения странности строки требуется найти количество ее подстрок, которые имеют такую структуру.

Будем решать задачу независимо для каждой упорядоченной пары (a, b) символов.

Пусть у нас есть несколько блоков подряд идущих символов a , после которых идет блок подряд идущих символов b . Сложим все пары (k_i, l_i) длин таких блоков в массив. Теперь нам надо найти число пар (k, l) , таких, что найдется (k_i, l_i) , такая что $1 \leq k \leq k_i$ и $1 \leq l \leq l_i$. Для поиска числа таких пар отсортируем все пары по k_i , а затем по l_i . Для каждой пары (k_i, l_i) заменим l_i на максимум l_j , где $k_j \geq k_i$. Теперь оставим по одной паре с каждым k_i и просуммируем величины $(k_i - k_{i-1}) \times l_i$ для всех i . Получившееся значение и является искомым.

Отметим, что описанный алгоритм является ничем иным, как алгоритмом поиска площади объединения прямоугольников с осями, параллельными осям координат, общим углом $(0, 0)$ и положительными координатами противоположного угла.

В завершение описания полного решения отметим, что построение списков пар (k_i, l_i) можно осуществить одним проходом по массиву.

При решении подзадачи 1 достаточно сформулировать утверждение о том, в каком случае строка является странной. После этого все подстроки могут быть проверены на странность независимо. При этом для устранения дубликатов можно, например, сложить все такие подстроки в структуру данных «множество».

Для решения подзадачи 2 не требуется новых идей по сравнению с подзадачей 1, однако при некоторых вариантах реализации повышается техническая сложность. Именно по этой причине стоимость подзадачи 2 невысока.

Решение подзадачи 3 требует эффективного метода устранения дублирующихся подстрок. Например, все пары (k, l) , таких, что найдется (k_i, l_i) , такая что $1 \leq k \leq k_i$ и $1 \leq l \leq l_i$ можно

сложить в структуру данных «множество», при этом множество должно быть отдельным для каждой пары символов (a, b) .

Задача 4 «Поездка на каникулах»

Это самая сложная задача первого дня. Полное решение этой задачи требует нескольких идей.

Пусть для каждой станции i определена величина $go1[i]$ – максимальный номер станции, до которой можно доехать от станции i , используя ровно 1 билет. Заметим, что нам не важно, на каком месте мы приедем на станцию, на которой мы делаем пересадку, поэтому выгодно от станции i всегда ехать либо до конечной станции маршрута, либо до станции $go1[i]$.

Решим сначала задачу для одного варианта поездки, используя построенный массив $go1$. Начнем со станции f . Если $go1[f] \geq t$, то достаточно одного билета. Иначе переместимся на станцию $go1[f]$, найдем ответ для нее и прибавим 1. Следует также учесть, что если $go1[i] = i$, то от станции i нельзя поехать дальше, и если в процессе перемещения мы встречаем такую станцию, то ответ для рассматриваемого варианта « -1 ». Отметим, что время работы предложенного метода ответа на запрос есть $O(n)$.

Посмотрим теперь, как построить массив $go1$.

Наивное построение, которое для каждой станции и каждого места находит, до какой станции можно доехать с использованием этого билета, и выбирает максимум, в сочетании с описанным методом по массиву $go1$ найти число билетов для одного варианта за время $O(n)$ позволяет решить подзадачу 1.

Для решения подзадачи 2 требуется более совершенный способ построения массива $go1$.

Будем говорить, что место a свободно на протяжении отрезка $[c, d]$, если никакой из проданных билетов не занимает это место на перегонах этого отрезка станций. Отсортировав введенные билеты по месту, а затем по начальной станции, легко построить все максимальные свободные отрезки станций.

Отсортируем свободные отрезки по начальной станции.

Будем рассматривать станции по очереди. Будем поддерживать множество «открытых» свободных отрезков, которые начинаются не позже рассматриваемой станции, в структуре данных, которая позволяет поиск максимума по ключу (двоичная куча, дерево отрезков или `std::set` подойдут). В качестве ключа будем использовать конец отрезка.

Рассматривая очередную станцию i , добавим все свободные отрезки, которые в ней начинаются, в множество открытых отрезков. Теперь $go1[i]$ равно максимальному концу открытого отрезка, либо i , если множество открытых отрезков пусто, или все они заканчиваются до i .

Этот метод позволяет решить подзадачу 2.

Для решения подзадачи 3 требуется быстрее отвечать на запрос. Для этого используем метод «двоичных подъемов». Посчитаем величину $go[j][i]$, которая равна максимальному номеру станции, до которого можно доехать от станции, используя не более 2^i билетов.

Код, вычисляющий $go[j][i]$ по $go1[i]$ приведен ниже.

```
for (int i = 1; i <= sz; i++) {
    for (int j = 0; j < n; j++) {
        int u = go[i - 1][j];
        go[i][j] = go[i - 1][u];
    }
}
```

В качестве sz следует выбрать минимальную величину, такую что $2^{sz} \geq n$.

Теперь для получения ответа на запрос будем делать все уменьшающиеся «прыжки» по степеням двойки. Код, который отвечает на запрос с использованием массива go приведен ниже.

```
// перемещаемся от fr до to
if (go[sz][fr] < to) {
    cout << to << endl;
} else {
    int steps = 0;
    int curK = sz;
    while (go1[fr] < to) {
        while (go[curK][fr] >= to) {
            curK--;
        }
        steps += 1 << curK;
        fr = go[curK][fr];
    }
    cout << steps + 1 << endl;
}
```

Задача 5 «Три сына»

Заметим следующее: чтобы минимизировать сумму квадратов необходимо стараться выбрать числа a , b и c близкими к $n/3$. Формализуем это утверждение.

Докажем, сначала, вспомогательный факт про два числа. Пусть различные положительные целые $a < b$ таковы, что $a + b = m$. Тогда если сумма $a^2 + b^2$ минимальна, то $b - a \leq 2$. Действительно, пусть $b - a > 2$, тогда $a + 1 \neq b - 1$ и $(a + 1)^2 + (b - 1)^2 = a^2 + 2a + 1 + b^2 - 2b + 1 = a^2 + b^2 + 2(a - b) + 2 < a^2 + b^2 - 4 + 2 < a^2 + b^2$, следовательно взяв вместо a и b

числа $a + 1$ и $b - 1$, мы получим два различных числа с такой же суммой, но меньшей суммой квадратов.

Рассмотрим теперь три числа a , b и c , которые требуется найти в задаче. Применяя предыдущее утверждение для a и b при фиксированном c , а также для b и c при фиксированном a , получим, что все числа не более чем на 4 отстоят от значения $n / 3$.

Получаем следующее решение: переберем все тройки целых положительных чисел, где каждое число, не более чем на 4 отличается от $n / 3$ и выберем среди них тройку с минимальной суммой квадратов.

Приведем пример программы на Си++, которая решает поставленную задачу.

```
int n3 = n / 3;
int delta = 4;
int minv = max(n3 - delta, 1);
int maxv = min(n3 + delta, n);
long long best = (long long) n * n;
int ba, bb, bc = 0;
for (long long a = minv; a <= maxv; a++)
    for (long long b = a + 1; b <= maxv; b++)
        for (long long c = b + 1; c <= maxv; c++)
            if (a + b + c == n && a*a + b*b + c*c < best) {
                best = a * a + b * b + c * c;
                ba = a;
                bb = b;
                bc = c;
            }
cout << ba << " " << bb << " " << bc << endl;
```

Отметим, что вместо числа 4 можно использовать и любое большее значение. Главное, чтобы оно не было *слишком* большим, чтобы решение по-прежнему проходило по времени.

Частичные решения для подзадач 1, 2 и 3 позволяют получить частичные баллы в случае неполного развития этой идеи.

Для решения подзадачи 1 можно перебрать все возможные значения a , b и c и выбрать из них оптимальное. Время работы $O(n^3)$.

Для решения подзадачи 2 следует заметить, что можно перебрать лишь два значения, например, a и b , а значение c вычислить из равенства $a + b + c = n$. . Время работы $O(n^2)$.

Наконец, для решения подзадачи 3 можно, например, перебрать большее из трех значений: c , и заметить, что $a + b = n - c = m$, а для минимизации $a^2 + b^2$ следует взять $a = (m - 1) / 2$, $b = (m + 1) / 2$ при нечетном m и $a = m / 2 - 1$, $b = m / 2 + 1$ при четном m . Время работы $O(n)$.

Задача 6 «Счет в гипершашках»

Рассмотрим три случая: все три игрока набрали равное число баллов, все три игрока набрали попарно различное число баллов, либо все два игрока из трех набрали равное число баллов, а у третьего число баллов отлично от тех двух.

Научимся находить количество вариантов счета в каждом из трех случаев, получившиеся значения необходимо сложить. Отметим, что в подзадаче 1 выполнено условие, что $k = 1$, и следовательно имеет место только первый случай, а в подзадаче 3 все числа различны, и следовательно имеет место только второй случай. Этим можно воспользоваться для написания частичных решений для этих подзадач.

Поместим входные данные в массив и отсортируем его. Одним проходом по получившемуся массиву заменим числа на пары (x_i, c_i) , где c_i – количество раз, которое значение x_i встречается на карточках у Андрея, x_i во всех парах различны. Заметим, что массив пар получился отсортированным по x_i .

```
cin >> n >> k;
vector<long long> x(n);
for (int i = 0; i < n; i++) {
    cin >> x[i];
}
sort(x.begin(), x.end());
vector<pair<long long, int>> p;
int pr = -1;
for (int i = 0; i < n; i++) {
    if (i == n - 1 || x[i] != x[i + 1]) {
        p.push_back({x[i], i - pr});
        pr = i;
    }
}

int m = p.size();
```


Андрей может показать счет, в котором все три игрока набрали равное число баллов, в случае, если у него есть хотя бы три карточки с этим числом баллов. Поэтому число таких вариантов, для которых можно показать счет, равно количеству чисел, которые встречаются хотя бы 3 раза. Ниже приведен код на Си++, который считает число таких вариантов.

```
long long ans1 = 0;
for (int i = 0; i < m; i++) {
    if (p[i].second >= 3) {
        ans1++;
    }
}
```

Для определения числа вариантов в остальных двух случаях требуется применить метод двух указателей. Для определения количества вариантов счета, в котором баллы всех игроков различны, сделаем следующее. Для каждого варианта максимального из баллов трех игроков найдем минимальное значение баллов другого игрока, которое можно показать и оно не нарушает условия, что баллы различаются не более чем в k раз. Пусть для максимального значения x_i соответствующее минимальное значение x_j . Тогда два других значения баллов можно выбрать среди значений $x_j, x_{j+1}, \dots, x_{i-1}$, то есть существует $(i-j)(i-j-1)/2$ способов это сделать. Поскольку 3 различных числа можно упорядочить 6 способами, это число необходимо умножить на 6.

Просуммируем эти значения по всем i . Заметим, что при увеличении i значение j также может только увеличиваться, поэтому проход занимает $O(n)$.

Приведем код на Си++, который подсчитывает это число вариантов.

```
long long ans2 = 0;
int j = 0;
for (int i = 0; i < m; i++) {
    while (j < i && p[j].first * k < p[i].first) {
        j++;
    }
    ans2 += 6LL * (i - j) * (i - j - 1) / 2;
}
```

Наконец, аналогичный метод применим для подсчета количества вариантов счета, когда два игрока набрали равное количество баллов, а третий игрок набрал отличное от них количество баллов.

```
long long ans3 = 0;
j = 0;
for (int i = 1; i < m; i++) {
    while (j < i && p[j].first * k < p[i].first) {
        j++;
    }
    if (p[i].second > 1) {
        ans3 += 3 * (i - j);
    }
}
j = m - 1;
for (int i = m - 2; i >= 0; i--) {
    while (j > i && p[i].first * k < p[j].first) {
        j--;
    }
    if (p[i].second > 1) {
        ans3 += 3 * (j - i);
    }
}
```

Для решения подзадачи 1 достаточно рассмотреть вариант, что у всех игроков одинаковое число баллов. Отметим, что дополнительное ограничение $1 \leq x_i \leq 100\,000$ позволяет обойтись без сортировки пар или сложных структур данных и запоминать количество карточек с каждым числом в массиве.

Для решения подзадачи 2 можно перебрать все тройки карточек, для каждого варианта счета в трехмерном массиве отметить, можно ли получить этот счет, и вывести количество вариантов, которое можно получить.

Для решения подзадачи 3 требуется реализовать описанное выше решение, но не требуется разбирать случай, когда два игрока набрали равное число баллов.

Задача 7 «Интересные числа»

Для решения этой задачи необходимо применить динамическое программирование.

Обозначим количество интересных чисел от L до R как $c(L, R)$. Заметим, что $c(L, R) = c(1, R) - c(1, L - 1)$, поэтому достаточно научиться считать количество интересных чисел, не превышающих заданного R .

Рассмотрим сначала решение третьей подзадачи, где $R = 10^k$. Поскольку само число 10^k интересным не является, задача сводится к подсчету количества интересных чисел, состоящих из k цифр, причем ведущие нули разрешаются (будет напрасно посчитано число 0, можно затем вычесть 1 из ответа). Используем динамическое программирование: обозначим как $d[i][j]$ количество интересных чисел из i цифр, последняя цифра которых равна j . Тогда $d[1][j] = 1$ для всех j , а для $i > 1$ выполнено равенство $d[i][j] = \sum(d[i-1][k], k = 0..j)$. Отметим, что все вычисления необходимо производить по модулю $10^9 + 7$, так что реализация работы с длинными числами не требуется.

Обратимся теперь к полному решению. Если R не является степенью 10, то не все интересные числа из k цифр подходят. Тем не менее, если рассмотреть префикс интересного числа, то если в нем есть хотя бы одна цифра строго меньшая соответствующей цифры R , то продолжение может быть любым. Иначе все цифры должны совпадать с соответствующим префиксом R и следующая цифра не должна превышать очередной цифры R .

Переберем общий префикс интересного числа и R . Пусть он равен k . Для каждого значения k запустим отдельное вычисление. Зафиксируем k . Будем использовать такое же динамическое программирование: $d[i][j]$ будет как и раньше содержать количество интересных чисел из i цифр, в которых последняя цифра j , но теперь первые k цифр должны совпадать с соответствующими цифрами R , а следующая цифра должна быть строго меньше. Тогда формула для пересчета не меняется, а вот начальные значения меняются: если первые k цифр числа R идут в неубывающем порядке, то $d[k+1][j] = 1$ для тех j , которые больше или равны k -й цифре числа R и строго меньше его $(k+1)$ -й цифры.

Заметим, что никаких арифметических действий с числом R не производится, поэтому реализация «длинной арифметики» для решения этой задачи не требуется.

Для решения подзадачи 1 достаточно перебрать все числа из диапазона от L до R и для каждого из них проверить, является ли оно интересным.

Для решения подзадачи 2 достаточно перебирать только интересные числа, сохраняя в переборе уже поставленную часть числа и последнюю цифру, будем дописывать только те цифры, которые приводят к сохранению свойства интересности.

Задача 8. «Гармоничная последовательность»

Обозначим первые два числа гармонической последовательности как x и y . Тогда сама последовательность будет иметь вид $x, y, (y-x), -x, -y, (x-y), x, y, \dots$. Заметим, что последовательность имеет цикл длины шесть. Если изначальная последовательность имеет вид b_1, b_2, b_3, \dots , то необходимо подобрать целые числа x и y , которые минимизируют

функцию $|x - b_1| + |y - b_2| + |(y - x) - b_3| + |-x - b_4| + |-y - b_5| + |(x - y) - b_6| + \dots = |x - b_1| + |y - b_2| + |(y - x) - b_3| + |x - (-b_4)| + |y - (-b_5)| + |(y - x) - (-b_6)| + \dots$. Таким образом, необходимо минимизировать сумму модулей величин, которые можно разбить на три группы. Слагаемые первой группы имеют вид $|x - a|$, второй $|y - b|$, третьей $|(y - x) - c|$.

Заметим, что существует пара (x, y) , которая минимизирует сумму модулей, такая, что как минимум в двух из трех групп значения хотя бы одного модуля равно нулю. Это свойство можно легко доказать от противного. Также заметим, что если зафиксировать, какие именно модули равны нулю, то можно однозначно восстановить числа x и y . Таким образом получаем решение за $O(n^3)$: зафиксируем две из трех групп, переберем, какие именно модули равны нулю, восстановим последовательность, обновим ответ.

Пусть все числа в исходной последовательности по модулю не превосходят A . Заметим, что существует правильный ответ в котором одно из чисел x и y не превосходит по модулю A , а другое $2A$. Это справедливо, так как существует модуль вида $|x - a_i|$ или $|y - b_i|$, который равен нулю, и из которого можно получить значение одной из переменных. A в худшем случае значение другой переменной можно будет определить из уравнения вида $|(y - x) - c_j| = 0$.

Таким образом, для решения подзадачи 1 можно просто перебрать значения трех элементов последовательности в пределах от -20 до 20 .

Для решения подзадачи 2 можно перебрать значение x и y в пределах от -200 до 200 , восстановить последовательность и выбрать лучший вариант.

Для решения подзадачи 3 можно перебрать значения x , получить выражение вида $|y - y_1| + |y - y_2| + \dots$, которое необходимо минимизировать. Далее перебрать модуль, который будет равен нулю и, используя префиксные и суффиксные суммы, посчитать значение выражения.

Для решения подзадачи 4 можно перебрать, какие два модуля равны нулю и с помощью префиксных и суффиксных сумм посчитать ответ.

Для решения подзадачи 5 необходимо заметить следующий факт. Если для некоторого x найдено минимальное значение y , которое является оптимальным, то для большего x оптимальное значение y не может быть меньше найденного. Таким образом полное решение задачи выглядит следующим образом. Разобьем все модули на три группы. Отсортируем модули в порядке увеличения значения константы в них. Переберем две группы, в которых будут модули равные нулю (без ограничения общности будем считать, что это группы вида $|x - x_i|$ и $|y - y_i|$). Переберем, какой из модулей первой группы равен нулю. Будем поддерживать указатель на модуль из второй группы, при обнулении которого

получается оптимальный ответ. При изменении выбранного модуля первой группы, попытаемся использовать модуль с большей константой во второй группе (пока следующий модуль дает ответ лучше чем текущий, будем увеличивать указатель). Для того, чтобы посчитать ответ для конкретной пары (x, y) необходимо понять, какие из модулей третьей группы раскроются с плюсом, а какие с минусом. Для этого можно воспользоваться двоичным поиском, а далее с помощью префиксных и суффиксных сумм посчитать значение выражения. Итого общая сложность решения равна $O(n \log n)$.

В заключение заметим, что функция, которую необходимо минимизировать, является выпуклой вниз как по x , так и по y , поэтому для решения задачи можно также воспользоваться различными численными методами, например градиентным спуском или двумерным троичным поиском. В зависимости от эффективности вычисления значения целевой функции такие решения могут набрать от 44 до 100 баллов.