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 div (a + 2d) модулей (при делении округлять следует вниз). Аналогично, вдоль другой стороны можно установить h div (b + 2d) модулей. В целом же на поле можно установить (w div (a + 2d)) × (h 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 4*n* 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 $1 \le k \le k_i$ и $1 \le l \le l_i$. 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 $k_j \ge k_i$. 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 вариантов 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 $1 \le k \le k_i$ и $1 \le l \le l_i$ 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 вариант поездки, используя построенный массив go1. Начнем со станции *f*. Если go1[*f*] ≥ *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 вариант «–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^j 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*