Разбор задачи «Тот самый фильм»

Опишем полное решение задачи.

Определим функцию проверки для ответа участника на некоторой позиции $\#j$. Правила начисления очков можно свести к трём следующим утверждениям:

- если символ, обозначающий ответ участника, совпадает с символом, обозначающим правильный ответ, участнику начисляется 2 очка;
- если символ, обозначающий ответ участника, не совпадает с символом, обозначающим правильный ответ, но при этом один из этих символов — A, участнику начисляется 1 очко;
- во всех остальных случаях участнику начисляется 0 очков.

Эта функция может выглядеть следующим образом (язык с C-подобным синтаксисом, rs — символ, обозначающий верный ответ, ps — символ, обозначающий ответ участника):

```
int check(char rs, char ps) {
    int res = 0;
    if (rs == ps) {
        res = 2;
    }
    else if ((rs == ‘A’) || (ps == ‘A’)) {
        res = 1;
    }
    return res;
}
```

Далее нужно применить эту функцию последовательно к каждой паре символов, находящихся на позиции $\#j$ в строке с правильными ответами и в строке с ответами участника, и посчитать сумму полученных значений.

Подзадачи 2, 3 и 4 гарантировали, что участник всегда выбирал один и тот же ответ, и позволяли ограничиться проверкой символа, обозначающего верный ответ. В этом случае проверочная функция могла бы выглядеть следующим образом (пример для подзадачи 4, в которой участник всегда выбирает ответ C):

```
int check_4(char rs) {
    int res = 0;
    if (rs == ‘C’) {
        res = 2;
    }
    else if (rs == ‘A’){
        res = 1;
    }
    return res;
}
```

Наконец, в первой подзадаче ($n = 3$) было достаточно написать три условных оператора, подобных телу функции $check$ — для каждого символа. Эта подзадача не требовала умения работать с циклическими операторами.

Разбор задачи «Разгон облаков»

В этой задаче фактически требуется отыскать минимум и максимум разности между числами из двух наборов.

Ограничения первой подзадачи позволяли искать эти величины алгоритмом с асимптотикой $O(n^2)$.

Важно было правильно инициализировать переменные для хранения минимального и максимального количества дней ожидания выполнения просьбы. Переменную, хранящую минимальное значение, следовало инициализировать достаточно большим значением (например, $s_n + 2$ — такой разности совершенно точно не будет), а переменную, хранящую максимальное значение, напротив, следовало инициализировать достаточно маленьким значением (например, -1 — такой разности тоже совершенно точно не будет).

Инициализация такими «невозможными» значениями гарантирует, что при правильной работе алгоритма они сразу же будут изменены (а если они вдруг остались без изменений — что-то пошло не так).

Далее можно было организовать просмотр

Фрагмент кода, решающего первую подзадачу, может выглядеть следующим образом (язык с С-подобным синтаксисом, нумерация элементов массивов с нуля):

```
for (int j = 0; j < m; j++) {
 for (int i = 0; i < n; i++) {
        int diff = s[i] - p[j];
        if (diff > 0) {
            if ((diff > maxd) {
                maxd = diff;
            }
            else if (diff < mind)) {
                mind = diff;
            }
            break;
        }
    }
}
```

Обратите внимание на сравнение разности с нулём: это важно сделать, поскольку барон Мюнхгаузен не может выполнять просьбы раньше, чем они поступили. Как только будет обнаружено первый подходящий день для исполнения просьбы, поступившей в день p_j, цикл, просматривающий солнечные дни, будет прерван (поскольку барон исполняет просьбы в ближайший день).

Полное решение задачи основано на том, чтобы избавиться от просмотра всех солнечных дней подряд с целью найти первый подходящий для выполнения просьбы.

Рассмотрим элементы последовательностей p_1 и s_1 — номер первого дня, в который поступила просьба об установлении хорошей погоды, и номер первого солнечного дня. Если $p_1 < s_1$, то барон Мюнхгаузен пообещает выполнить в день s_1 просьбу, поступившую в день p_1. Ожидать выполнения этой просьбы придётся $s_1 - p_1$ дней. Если же $p_1 \geqslant s_1$, то день s_1 нельзя выполнить ни одной просьбы: и солнечные дни, и дни, в которые поступают просьбы, упорядочены по возрастанию. Следовательно, разности вида $(p_j - s_1)$ можно вообще не проверять.

Более того, следует просматривать элементы s_i до тех пор, пока не обнаружится тот, что будет больше p_1. Просмотр подходящих солнечных дней для просьбы, поступившей в день p_2, можно будет начать именно с этого элемента. Таким образом можно выполнить один проход вдоль каждого из массивов и найти необходимые величины.

Фрагмент кода, решающего первую подзадачу, может выглядеть следующим образом (язык с С-подобным синтаксисом, нумерация элементов массивов с нуля):

```
int sidx = 0;
int pidx = 0;
while ((sidx < n) && (pidx < m)) {
    if (p[pidx] < s[sidx]) {
        int cur = s[sidx] - p[pidx];
        if (cur < mind) {
            mind = cur;
        }
        if (cur > maxd) {
            maxd = cur;
        }
        pidx++;
```

```
        }
        else {
            sidx++;
        }
    }
```

Техника асинхронного изменения индексов двух массивов, применённая в этом фрагменте кода, часто называется техникой «двух указателей».

Разбор задачи «Установление хорошей погоды»

Полное решение этой задачи могло быть получено «жадным» образом.

Действительно, для каждого дня a_j, в который поступила просьба об установлении хорошей погоды, можно определить левую границу $l_j = a_j + 1$ и правую границу $r_j = a_j + d$ дней, в которые эта просьба должна быть исполнена. Поскольку дни, в которые поступали просьбы об установлении хорошей погоды, упорядочены по возрастанию, то и получаемые левые и правые границы будут точно так же упорядочены.

Выберем для просьбы, поступившей в день a_1, самый последний из возможных дней исполнения, а именно $r_1 = a_1 + d$. В этом случае мы можем надеяться, что просьбы, поступившие в следующие дни, тоже можно будет исполнить в этот день. Действительно, если окажется, что для дня a_2 хотя бы первый возможный день для выполнения просьбы $l_2 \leqslant r_1$, барон сможет сказать, что в день r_1 он выполнил просьбы, поступившие в дни a_1 и a_2.

Аналогичным образом будем проводить рассмотрение для следующих дней. Когда найдётся день i, для которого $l_i > r_1$, это будет означать, что заявить о выполнении этой просьбы в день r_1 уже нельзя. Выберем днём выполнения этой просьбы день r_i, чтобы попробовать использовать этот день для выполнения других (следующих) просьб.

Описанный процесс может считаться примером техники «двух указателей» (см. также разбор задачи B).

Фрагмент кода, выполняющий описанные выше действия, может выглядеть следующим образом (C-подобный синтаксис языка):

```
m = 1;
int left = a[0] + 1;
int right = a[0] + d;
b = new ArrayList<Integer>();
b.add(right);

for (int i = 1; i < n; i++) {
    int curleft = a[i] + 1;
    int curright = a[i] + d;
    if (curleft > right) {
        left = curleft;
        right = curright;
        m++;
        b.add(right);
    }
}
```

Некоторые пояснения: $ArrayList < Integer >$ (Java) может быть заменён массивом длины n (в котором будут иметь осмысленное значение только m первых элементов), а также $vector < int >$ (в C++) или $list$ (в Python).

Первая подзадача допускала решение с асимптотикой $O(n^2)$. В этом случае можно было, получив левую и правую границы для некоторого дня, выполнить проверку для остальных дней, могут ли быть полученные просьбы выполнены в рамках этих границ. Конечно, написание такого кода требовало определённой аккуратности, чтобы избежать учёта одного и того же дня дважды. Для этих целей можно было, например, использовать массив булевых значений, в котором на позиции $\#j$

устанавливалось бы истинное значение, если для просьбы, поступившей в день a_j, устанавливался бы день её выполнения.

Разбор задачи «Облачный уровень»

Для решения задачи реализуем вспомогательную функцию $can(b)$. Эта функция будет возвращать истинное значение ($true$), если при выборе значения b в качестве «критичного» значения облачности горожане всегда будут оставаться в хорошем настроении (не будет более k подряд идущих слишком облачных дней).

Как $can(b)$ будет выглядеть?

Переберём все значения облачности в порядке следования.

Если значение облачности больше b — увеличим специальный счетчик len подряд идущих слишком облачных дней (изначально он равен 0).

Если же значение облачности не превосходит b — счетчик len обнуляется.

В конце каждой итерации проверим: если len превосходит k — возвращаем ложное значение ($false$) и завершаем выполнение функции.

Очевидно, что итоговая сложность работы этой функции составляет $O(n)$.

Также для решения задачи сделаем важное замечание: итоговое значение b не может быть меньше 0 и не может быть больше максимального элемента в массиве $c[]$.

Теперь рассмотрим, как можно решать различные подзадачи, используя $can(b)$.

Для решения подзадачи 1 ($n \leqslant 500$, $0 \leqslant c_i \leqslant 500$) достаточно циклом перебрать все возможные b от 0 до 500 и выбрать минимальное b, для которого $can(b) = true$.

Итоговая сложность такого решения будет $O(n \cdot \max\limits_{j=1}^{n}(c[j]))$.

Для решения задачи на полный балл ($n \leqslant 3 \cdot 10^5$, $0 \leqslant c_i \leqslant 10^9$) рассмотрим, как изменяется значение функции $can(b)$ при увеличении b.

Если для какого-то значения b функция $can(b) = false$ — слишком много дней оказались слишком облачными. В таком случае можно сделать вывод, что и $can(b-1) = false$: при уменьшении b все слишком облачные дни останутся слишком облачными, могут только добавиться новые.

Если же для значения b функция $can(b) = true$, это значит, что не нашлось ни одного непрерывного отрезка слишком облачных дней длины более k. Из этого можно сделать вывод, что и $can(b+1) = true$: при увеличении b новых слишком облачных дней не появится, только уже существующие могут перестать быть таковыми.

Исходя из вышесказанного можно понять, что функция $can(b)$ изменяется монотонно: существует такой $0 \leqslant x$, что для всех $b \geqslant x$ $can(b) = true$, а для всех $b < x$ $can(b) = false$.

Понятно, что это значение x и будет являться ответом на задачу. Чтобы быстро его найти, можно воспользоваться двоичным поиском по функции $can(b)$ — мы уже доказали, что она монотонна.

Сложность такого решения будет $O(n \cdot log(max(c)))$.

Примечание. Двоичный поиск — это способ поиска аргумента, при котором монотонная на отрезке функция принимает определенное значение. Для этого вычисляется и анализируется значение в середине отрезка:

Если оно лежит с той же стороны (меньше/больше), что и значение на левом конце отрезка — то искомый аргумент лежит в правой половине и левую можно отбросить.

Если оно лежит с той же стороны, что и значение на правом конце отрезка — то искомый аргумент лежит в левой половине и правую можно отбросить (аналогично предыдущему).

На каждой итерации длина отрезка уменьшается в 2 раза, а значит итоговая сложность работы будет $O(log(R{-}L){\cdot}F)$, где L и R — левая и правая границы отрезка, соответственно, а F — сложность вычисления функции в точке.

Разбор задачи «Облачные технологии»

Ограничения первой подзадачи ($n \leqslant 15$) позволяли написать любой вид рекурсивного перебора за $O(2^n)$ или $O(2^n \cdot n)$.

В рекурсивном переборе необходимо передавать номер текущего дня и массив (список) уже набранных дней. Обозначим функцию за $f(day, taken)$.

В таком случае есть три вида переходов:

- ничего не делать: $f(day + 1,\, taken)$
- работать (разгонять облака) в день day: $f(day + 2,\, taken.add(day))$
- работать в дни day и $day + 1$: $f(day + 4,\, taken.add(day,\, day + 1))$

Условием остановки рекурсии будет $day > n$. В таком случае необходимо сравнить длину лучшего найденного ответа и текущего списка дней и выбрать наибольший.

Также рекурсивный перебор можно было заменить перебором по двоичным маскам: бит, равный 1, будет означать рабочий день, а 0 — нерабочий.

В таком случае для каждой маски от 0 до $2^n - 1$ необходимо проверить, удовлетворяет ли она условиям отдыха (после d дней работы надо отдыхать не менее d дней; можно работать не более двух дней подряд), и выбрать из корректных масок ту, в которой наибольшее количество единичных битов.

На полный балл задача решается методом динамического программирования.

Обозначим через $dp[j]$ максимальное количество солнечных дней (обеспеченных стараниями барона Мюнхгаузена) на отрезке от j до n.

Базой динамики будет $dp[n + 1] = dp[n + 2] = dp[n + 3] = dp[n + 4] = 0$. Эти «фиктивные» дни мы добавим после «реальных» для удобства; в эти дни разгонять облака мы уже не можем.

Рассмотрим переходы для дня $\#j$:

- Ничего не делали: ответ будет равен ответу для $j + 1$ без каких-либо изменений;

  Формула: $dp[j] = \max(dp[j],\, dp[j + 1])$
- Работали в день $\#j$, но не в день $\#(j+1)$; в этом случае в день $\#(j+1)$ был отдых, а работать можно опять начать с дня $\#(j + 2)$.

  Проверка возможности такого варианта: $u[j] \geqslant c[j]$.

  Формула: $dp[j] = \max(dp[j],\, dp[j + 2] + 1)$.
- Работали в дни $\#j$ и $\#(j + 1)$; в этом случае отдых приходился на дни $\#(j + 2)$ и $\#(j + 3)$, поэтому работать можно начать с дня $\#(j + 4)$.

  Проверка возможности такого варианта: $u[j] \geqslant c[j]$ **и** $u[j + 1]/2 \geqslant c[j + 1]$.

  Формула: $dp[j] = \max(dp[j],\, dp[j + 4] + 2)$.

По построению динамики ответ будет находиться в $dp[1]$.

Для восстановления ответа будем использовать дополнительный массив $p[]$, элементами которого будут типы перехода, дающие лучший результат в динамике (1, 2 или 3).

Восстановление ответа начинаем с дня 1.

Если для текущего рассматриваемого дня $p[day] = 1$ (мы ничего не делали) — то день не добавляем в ответ и переходим в $day + 1$.

Если $p[day] = 2$ (мы работали 1 день) — добавляем день day в ответ и переходим в $day + 2$.

Если же $p[day] = 3$ (мы работали 2 дня) — добавляем дни day и $day + 1$ в ответ и переходим в $day + 4$.

Этот процесс продолжается, пока не выйдем за n (последний «реальный» день).