Разбор задач вступительной ЛКЛ-2016, параллели C'–A

Содержание

A. [C'] Сумма максимума и минимума **1**

B. [C', C] Разность сумм **3**

C. [C', C] Дни рождения **3**

D. [C, B'] Круглое упорядочивание **3**

E. [C, B'] Химия **5**

F. [B', B] Разложение в сумму **5**

G. [B', B] Приключения Спайка **7**

H. [B, A] Кроты **7**

I. [B, A] Метро **9**

J. [A] Роборука **10**

K. [A] Прохладительные напитки **11**

A. [C'] Сумма максимума и минимума

Разработчик задачи: Ирина Турова.

Задача решается простым циклом, на C++ можно было реализовать проход следующим образом:

```
int n;
cin >> n;
int mn = 10010, mx = -10010;
for (int i = 1; i <= n; ++i) {
    int a;
    cin >> a;
    if (i % 2 == 1) {
        mn = min(mn, a);
    } else {
        mx = max(mx, a);
    }
}
cout << mn + mx;
```

На других языках реализация была бы похожей. Обратим внимание, что, поскольку гарантировалось $N \geq 2$, и минимум, и максимум хотя бы один раз обновится с инициализируемого значения.

Здесь и далее, для краткости мы будет вставлять только фрагменты кода, непосредственно решающие задачу. Подключение модулей, перенаправление стандартного ввода-вывода в файлы, объявления некоторых структур и тому подобное будет опускаться.

B. [C', C] Разность сумм

Разработчик задачи: Даниил Плющенко.

Эта задача тоже реализационная, самое простое решение делает один проход по табличке двумя циклами:

```
cin >> n;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
        cin >> x;
        ans[i] += x;
        ans[j] -= x;
    }
for (int i = 0; i < n; i++)
    cout << ans[i] << " ";
cout << '\n';
```

Массив *ans* должен иметь размер не меньше n.

C. [C', C] Дни рождения

Разработчик задачи: Кирилл Симонов.

Первой частью решения этой задачи было справиться с длинным условием. Формально задачу можно сформулировать так: необходимо посчитать количество натуральных чисел из отрезка $[N, N+P-1]$, сумма цифр десятичной записи которых делится на P.

Само же решение довольно прямолинейно: поскольку все числа в условии не превосходят 10^5, достаточно было для каждого числа из этого отрезка посчитать сумму цифр и остаток от деления суммы цифр на P.

D. [C, B'] Круглое упорядочивание

Разработчик задачи: Ирина Турова.

Поскольку речь идет об упорядочивании элементов, естественно воспользоваться каким-либо алгоритмом сортировки. Ограничение $n \leq 1000$ позволяло использовать квадратичные алгоритмы, такие как сортировка пузырьком, вставками, обменами. Разумеется, более быстрые алгоритмы

сортировки, в том числе стандартные в C++, Java, Python и других языках, тоже прекрасно работали.

Тонкость же здесь в том, что сортировать числа надо не по их величине, а по введенному в условии задачи отношению, в котором раньше должно идти то число, у которого больше нулей на конце, а при равенстве — просто большее. Для реализации необходимо было заменить в сортировке обычное сравнение чисел на сравнение, описанное в условии, если вы использовали собственноручно написанный алгоритм сортировки. Если же вы пользовались стандартной функцией сортировки, нужно было написать подходящую функцию-компаратор (впрочем, и в случае своей сортировки выделить сравнение двух чисел в функцию не помешало бы). Пример такого компаратора:

```
int zeros(int a) {
    int res = 0;
    while(a % 10 == 0) {
        a /= 10;
        res++;
    }
    return res;
}

bool zless(int a, int b) {
    int za = zeros(a);
    int zb = zeros(b);
    if (za != zb)
        return za < zb;
    return a < b;
}
```

В компараторе важно было не забыть про случай равенства числа нулей на конце и сравнить в этом случае сами числа.

Можно было и заранее для каждого числа посчитать число нулей на конце и запомнить его вместе с самим числом в общую структуру (например, в std::pair на C++), а дальше сортировать уже эти структуры. Так не придется считать число нулей на конце числа каждый раз при сравнении двух чисел — даже в быстрых сортировках сравнений $O(n \log n)$, против $O(n)$ в случае предподсчета, а подсчет числа нулей на конце работает за длину десятичной записи числа, в худшем случае. Но

с учетом того, что n и величины чисел были малы, такая оптимизация не требовалась.

E. [C, B'] Химия

Разработчик задачи: Кирилл Симонов.

В этой задаче формальная версия условия звучит так: дана строка S, на нечетных шагах мы циклически сдвигаем вперед ее начало длины k, на четных циклически сдвигаем назад ее конец длины $n-k$, и необходимо узнать, через какое минимальное число шагов мы получим заданную строку T, либо сказать, что это невозможно.

Ключевая идея решения — нашими операциями нельзя получить много разных строк. А именно, через каждые $2k$ шагов начало строки точно возвращается к своему первоначальному состоянию, и, аналогично, через каждые $2(n-k)$ шагов конец строки. Поэтому вся строка становится такой же, как и в начале, через $2 \cdot \text{НОК}(k, n-k)$ шагов, при этом из-за множителя 2 тип следующего шага также совпадет с типом первого шага. То есть, состояние через $2 \cdot \text{НОК}(k, n-k)$ шагов полностью совпадает с начальным, а значит новых строк после этого точно не получится.

Из ограничений задачи следует $2 \cdot \text{НОК}(k, n-k) \leq 5000$, поэтому достаточно просто явно сделать такое количество шагов и посмотреть, когда в первый раз получилась строка T, или, если она ни разу не возникла, вывести -1. Каждый шаг, включающий в себя сравнение со строкой T, делается за $O(n)$ операций.

Можно было и не считать период явно, а просто применять к строке операции до тех пор, пока на некотором шаге с четным номером не получится исходная строка.

F. [B', B] Разложение в сумму

Разработчик задачи: Ирина Турова.

Типичная задача на рекурсивный перебор комбинаторных объектов. Довольно естественно за шаг рекурсии выбирать очередное число в текущей сумме, осталось придумать, как это организовать. Здесь подсказкой служит условие задачи — порядок, в котором требуется выводить ответы, естественный для правильного решения, и может помочь его приду-

мать.

А именно, если в каждой сумме нужно выводить сначала большие слагаемые, а потом маленькие, то выбор очередного слагаемого прост — оно должно быть не больше, чем предыдущее. И, кроме того, очередное слагаемое не должно быть слишком большим, чтобы вся сумма не стала больше чем n. Поэтому логично передавать в рекурсию два параметра: каким было предыдущее слагаемое (или очень большое число, если слагаемое первое), и сколько еще осталось набрать в текущую сумму, то есть разность n и уже выбранных слагаемых. Когда же мы выбираем очередное слагаемое, мы можем взять любое, не превышающее двух вышеупомянутых ограничений сверху, но поскольку разбиения нужно вывести в порядке, обратным лексикографическому, нет выбора, кроме как перебирать очередное слагаемое в порядке от максимально возможного его значения на текущем шаге до 1.

Нетрудно заметить, что таким образом устроенная рекурсия как раз переберет все разбиения в нужном порядке. Ниже код — один из вариантов реализации.

```
void rec(int n, int mn, string tek) {
    if(n == 0) {
        printf("%s\n", tek.c_str());
        return;
    }
    if(tek != "")
        tek += '+';
    for(int i = mn; i >= 1; --i)
        rec(n - i, min(n - i, min(mn, i)),
            tek + intToStr(i));
}
```

Вызов из основной программы:

```
rec(n, n, "");
```

Здесь в качестве параметров рекурсивной функции передаются: значение, которое нужно набрать оставшейся частью суммы, максимальное значение очередного слагаемого, и текущая строка — начало очередной строки вывода. Функция `intToStr` не является библиотечной, но просто переводит число в строку, поэтому здесь не приведена.

Заметим, что перебираются только корректные разбиения, а их при

данных ограничениях не слишком много — 5604 для n, равного 30, поэтому со скоростью работы проблем нет.

G. [B', B] Приключения Спайка

Разработчик задачи: Антон Чаплыгин.

Итак, по условию задачи нам нужно добраться из левого верхнего угла в правый нижний, набрав как можно меньшую сумму по пути, при этом переходить разрешается вниз, направо, или, из некоторых клеток, одним из двух ходов коня. В любом случае, переходы ведут нас ниже и/или правее нашей текущей клетки, в частности это означает, что вернуться назад мы никогда не сможем.

Пусть мы в каждой клетке хотим посчитать, за какое минимальное время мы можем дойти до нее из левого верхнего угла (для правого нижнего угла нам эту величину считать все равно придется, так как это и есть ответ). Что нужно знать, чтобы посчитать это время в какой-то конкретной клетке? Если мы оказались в этой клетке, то мы пришли либо сверху, либо слева, либо ходом коня, если это было возможно. Таким образом, ответ для клетки — минимум из ответов для клеток, из которых в нее возможны переходы (которых не более 4), плюс время, которое нужно потратить непосредственно в этой клетке.

Теперь вспомним, что все наши переходы ведут вниз и/или направо, а значит если мы будем рассматривать клетки в порядке сверху вниз, слева направо, то при обработке очередной клетки значения во всех клетках, из которых в нее могут быть переходы, будут уже посчитаны! Останется только выбрать из них минимум и добавить значение в самой клетке.

Заметим, что в этой задаче несколько удобнее писать динамику "вперед", то есть, находясь в клетке, обновлять значения в тех клетках, в которые из нее есть переход. Удобство состоит в том, что нам изначально задано, из каких клеток можно делать переход ходом коня.

H. [B, A] Кроты

Разработчик задачи: Антон Чаплыгин.

В этой задаче было необходимо найти последовательность переходов из некоторого положения кротов в некоторое другое. Другими словами,

если мы построим граф, в котором вершина — координаты двух кротов, а ребра — удовлетворяющие условию переходы, то нам нужно найти путь в этом графе, начинающийся и оканчивающийся в вершинах определенного типа. Для этого подойдет любой из алгоритмов обхода графа, поиск в ширину или поиск в глубину. Осталось только понять, как устроен этот граф.

Заметим сначала, что в качестве вершины (состояния) достаточно хранить положение одного из кротов — второй крот по условию всегда находится в симметричной клетке, поэтому восстановить его положение не составит труда. Более того, можно считать, что выбор хода есть только у того крота, координаты которого мы храним — куда бы он не пошел, второй крот должен следовать симметрично. Поэтому, чтобы провести ребра из какой-то вершины, нужно проверить четыре возможных хода крота из соответствующей клетки. Проверить, можно ли сделать переход из данной клетки по данному направлению довольно просто — нужно посмотреть, нет ли препятствия в той клетке, в которую мы попадем, и в клетке, ей симметричной. Если обе эти клетки свободны, то и наш крот, и крот-двойник смогут пройти. Для удобства можно было с самого начала сделать препятствия симметричными — если в одной клетке есть препятствие, а в ей симметричной нет, то в последнюю можно поставить дополнительное препятствие. Оно ничего не изменит, поскольку в эту пару клеток попасть все равно нельзя.

Начальным состоянием может быть любая симметричная пара нор — от нас требуется только чтобы все четыре норы (две стартовые, две конечные), оказались различны, а из-за четности N все клетки делятся на пары симметричных (если бы N было нечетно, центральная клетка была бы симметрична сама себе). Конечным состоянием — тоже любая пара симметричных нор, не совпадающая со стартовой. Заметим, что если две пары симметричных клеток имеют общую клетку, то они совпадают, иначе все четыре клетки различны. Для простоты, можно было перебрать стартовую пару нор, и обходом в глубину или ширину попытаться найти путь до другой пары нор — $O(N^2)$ на перебор стартовой пары, и еще $O(N^2)$ на обход, всего $O(N^4)$, что позволялось ограничениями.

Восстановление ответа здесь работает точно так же, как и всегда — если мы для каждой вершины помним, из какой предыдущей вершины мы в нее попали, то, откатываясь от конечной вершины пути с помощью этой информации, мы восстановим весь путь.

Реализация задачи содержит некоторое количество технических де-

талей, поэтому рекомендуется заглянуть в авторское решение.

I. [B, A] Метро

Разработчик задачи: Кирилл Симонов.

Сразу стоит избавиться от легенды, формальное условие довольно короткое: нам дана последовательность чисел, необходимо найти в ней подотрезок максимальной длины такой, что его сумма не меньше, чем заданное число S.

Так как нам нужно иметь дело с суммами на отрезках неменяющегося массива, естественно рассмотреть префиксные суммы. Пусть $a_1, \ldots, a_N$ — данная нам последовательность, тогда определим $p_k = \sum_{i=1}^{k} a_i$ для k от 1 до N, $p_0 = 0$. Эти значения несложно вычислить: $p_k = p_{k-1} + a_k$. Сумма чисел на любом подотрезке выражается через них, а именно, $a_l + \cdots + a_r = p_r - p_{l-1}$, где $1 \le l \le r \le N$.

Итак, отрезок $[l, r]$ нам подходит, если сумма на нем хотя бы S, то есть, в терминах префиксных сумм, $p_r - p_{l-1} \ge S$. Пусть мы зафиксировали левый конец отрезка l, тогда в качестве правого конца нам подходят такие r, что $p_r \ge S + p_{l-1}$, при этом правая часть неравенства не меняется.

Давайте научимся отвечать на следующий вопрос: правда ли, что существует отрезок длины хотя бы k с заданным левым концом l, имеющий сумму хотя бы S? То есть, нас интересуют такие r, что $r \ge l + k - 1$, и есть ли среди них хотя бы один такой, что $p_r \ge S + p_{l-1}$. Но среди $p_{l+k-1}, \ldots, p_N$ есть такой p_r, что $p_r \ge S + p_{l-1}$ тогда и только тогда, когда $\max(p_{l+k-1}, \ldots, p_N) \ge S + p_{l-1}$! Таким образом, если $m_r = \max(p_r, \ldots, p_N)$, то ответить на наш вопрос очень просто — такой отрезок существует тогда и только тогда, когда $m_{l+k-1} \ge S + p_{l-1}$.

Вычислить значения m_r несложно: $m_N = p_N$, $m_r = \max(p_r, m_{r+1})$ для r от $N - 1$ до 1. Осталось заметить, что при фиксированной левой границе свойство "существует отрезок длины хотя бы k такой, что его сумма хотя бы S" монотонно: если существует такой отрезок длины хотя бы k, то существует такой отрезок длины хотя бы $k - 1$, и наоборот, если не существует такого отрезка длины хотя бы k, то уж тем более не существует и такого отрезка длины хотя бы $k + 1$. Поэтому можно сделать бинпоиск по k.

Итоговая сложность $O(N)$ на подсчет p_k, m_r, $O(N)$ на выбор левой границы, еще $O(\log N)$ на бинпоиск для каждой левой границы, условие

внутри бинпоиска мы научились проверять за $O(1)$, всего наше решение работает за $O(N \log N)$.

J. [A] Роборука

Разработчик задачи: Кирилл Симонов.

Заметим, что каждая из команд делает следующее: первые несколько шарниров остаются на месте, а остальные сдвигаются на один и тот же вектор. Причем этот вектор можно посчитать: команда меняет некоторый сегмент, и вектор сдвига это просто разность вектора-сегмента после изменения с вектором-сегментом до изменения. Теперь, поскольку сдвиг это просто независимое добавление некоторого числа к x-координате и некоторого числа к y-координате, то параллельный перенос всех шарниров, начиная с некоторого места, это прибавление на суффиксе x-координат и еще одно прибавление на суффиксе y-координат шарниров. Осуществлять такие изменения и узнавать в любой момент положение каждой точки могут такие структуры данных, как дерево отрезков с групповыми операциями.

На самом деле решение можно упростить — изменяется положение целого хвоста точек, но для ответа на запрос нужно узнать положение всего лишь двух шарниров. Логичный ход для такого случая — перейти к разностям, и избавиться таким образом от групповых операций.

А именно, давайте хранить не сами координаты шарниров, а разности между каждыми двумя последовательными шарнирами, другими словами — вектора сегментов. Чтобы узнать координаты шарнира, нужно сложить все вектора сегментов, которые ему предшествуют. При выполнении команды, в свою очередь, меняется только один сегмент. Таким образом, если мы храним отдельно x-координаты, отдельно y-координаты, то нужно уметь находить сумму чисел на префиксе, и изменять значение одного числа. С этим справляются многие структуры, эффективнее всего использовать дерево Фенвика.

Если вам понравилась идея задачи, немного другая, более сложная постановка появлялась недавно на Codeforces: `http://codeforces.com/contest/618/problem/E`, можете попробовать решить и ее. В той задаче при повороте сегмента последующая часть конструкции не просто сдвигается, но и поворачивается вместе с ним.

K. [A] Прохладительные напитки

Разработчик задачи: Кирилл Симонов.

В условии на самом деле сказано следующее: дано несколько строк, нужно посчитать количество их различных подстрок. Посчитать количество различных подстрок одной строки — стандартная задача, и те же подходы работают здесь.

Первое решение: хеширование. В качестве наивного решения можно просто выписать все подстроки данных строк, и каким-нибудь образом посчитать число различных, с помощью сортировки, хеш-таблицы, или аналога std::set. Такое решение будет работать за нечто кубическое, и не уложится в ограничение по времени. Но вместо того, чтобы выписывать сами подстроки, можно сделать то же самое с их хешами, тогда кубическое решение превратится в квадратичное! Правда, на этом пути ждало одно препятствие: против обычного модуля 2^{64} легко строится антихештест, и именно это было сделано в тестах к этой задаче. На самом деле, в условии была некоторая подсказка: антихештест основан на строке Туэ–Морса, и эти слова фигурировали в тексте неспроста. Более подробно про антихештест можно узнать из легендарного поста на Codeforces: `http://codeforces.com/blog/entry/4898`.

Таким образом, чтобы заставить работать хеширование, нужно было сделать что-либо большее, чем взятие по стандартному модулю 2^{64}. Хороший рецепт для устойчивого хеширования — взять пару или тройку простых модулей, каждое порядка 10^9, и считать по ним всем одновременно (считать строки равными тогда, когда хеши совпали по *всем* модулям). Для этого удобно завести структуру, которая хранит значения по всем модулям, и умеет производить с ними арифметические операции, тогда основной код выглядит точно так же, как он бы выглядел с обычным хешированием по одному модулю. Пример такой реализации можно найти в одном из авторских решений.

В итоге, решение с хешами не самое быстрое, поскольку приходится считать их по нескольким модулям, и еще, поскольку из них нужно оставить только различные с помощью некоторой структуры, сложность умножается на константу хеш-таблицы, или логарифм для сортировки или структур на основе дерева. Однако, при небольшой аккуратности, даже решение с std::set укладывалось в ограничение по времени.

Второе решение: z-функция. Научимся сначала решать задачу для одной строки. Начнем с пустой строки, будем добавлять ей в конец

по одному символу и понимать, сколько новых подстрок появилось. Поскольку где-то в середине от добавления символа в конец ничего нового появиться не могло, нам нужно проверить только подстроки, являющиеся суффиксами текущей строки. Простое наблюдение: если какой-то суффикс уже встречался в строке, то и любой меньший суффикс тоже встречался, так как является подстрокой большего суффикса. Таким образом, нужно определить, какой максимальный суффикс уже встречался, тогда он и все меньшие уже посчитаны, а все большие точно новые.

Для этого временно развернем накопившуюся сейчас строку, и посчитаем ее z-функцию. Утверждается, что максимум z-функции это и есть длина самого большого суффикса, который уже встречался. Действительно, если в некотором месте строки z-функция достигла своего максимального значения, то в этом же месте начинается совпадение с началом нашей перевернутой строки соответствующей длины. И наоборот, если суффикс исходной строки некоторой длины, то есть префикс перевернутой строки, уже где-то встречался, то и z-функция достигнет в этом месте хотя бы такого значения.

Соответственно, после вычисления z-функции осталось взять ее максимум, и вычесть из текущей длины строки — это и есть количество суффиксов, которых раньше никогда не было.

Чтобы получить решение нашей задачи, нужно совсем немного изменить рассуждение выше. Сначала запишем все данные нам строки подряд, разделив их символом, который в них точно не встречается, например '$'. Если мы теперь просто применим решение для одной строки, мы еще не получим правильного ответа — мы в том числе посчитаем и подстроки, содержащие наш разделяющий символ, которые точно нигде в исходном наборе не встречались.

Однако, это легко преодолеть: когда мы добавляли новый символ в старом решении, мы знали, что все суффиксы длины большей, чем максимальное значение префикс функции, точно встречаются впервые, и добавляли их к ответу. Теперь же, эти суффиксы могли оказаться слишком длинными, и содержать разделяющий символ. Но мы знаем, какие суффиксы его содержат — в точности все суффиксы, которые длиннее, чем расстояние от конца текущей строки до последнего разделяющего символа. Поэтому, нужно просто вычитать максимальное значение z-функции не из длины строки, а из этого расстояния. При этом разность могла получиться и отрицательной, поскольку внутри подсчета z-функции ограничения с разделяющими символами мы не накладывали, но это просто

значит, что все интересные нам суффиксы уже встречались, и к ответу нужно добавить 0.

Заметим, что это решение заметно быстрее решения с хешами: оно чисто квадратичное с довольно небольшой константой.

На самом деле, все вышесказанное можно было повторить, используя префикс-функцию вместо z-функции.

Существуют и более быстрые решения: используя суффиксные структуры, можно добиться линейного времени работы. Но суффиксные структуры мы оставим на лето, для тех, кто поступит в параллель A :)