Contents

1 Проверка деревьев на изоморфность

• **Определение 1.**

Пусть дерево задается матрицей смежности. Два дерева с матрицами смежности c_1 и c_2 называются изоморфными, если существует такая перестановка p, что $\forall i, j : c_1[p_i, p_j] = c_2[i, j]$.

• **Определение 2.**

Изоморфизм корневых деревьев определяется аналогично. Разница в том, что корень переходит в корень ($p_{root_1} = root_2$).

• **Алгоритм 1. Корневое дерево.** $O(n \log n)$**.**

Даны два корневых дерева. Нужно проверить, изоморфны ли они. Определим хеш вершины, как хеш сортированного вектора хешей детей. Деревья изоморфны, если хеши корней совпадают. При этом хеш функция должна обладать следующим свойством: хеши различных векторов различны.

• **Алгоритм 1. Подробности.**

Алгоритм работает за $O(n \log n)$, при этому все, кроме сортировки работает за $O(n)$. Важно обратить внимание на выбор хеш функции. В данном случае полиномиальный хеш не подходит. Пример: пусть есть дерево $(1, 2), (1, 3), (1, 4)$ и дерево $(1, 2), (1, 3), (3, 4), (3, 5)$. Хеш от вектора $a_0, a_1, \ldots, a_k =$

$$\sum_{i=0}^{k} a_i p^i$$

Деревья различны. Тем не менее, хеши от них равны: $1 + 1 \cdot p + 1 \cdot p^2 = 1 + (1 + 1 \cdot p) \cdot p$. Подходит, например, следующая хеш функция:

$$13 + \sum_{i=0}^{k} \log a_i$$

При реализаци на `C++` считать ее следует в типе `long double`. Может показаться, что, поскольку функция не зависит от порядка a_i, можно отказаться от сортировки. Это не так. Появится погрешность, поэтому хеш зависит от порядка. Другой пример хорошей хеш функции:

$$\Big(\sum_{i=0}^{k} a_i^2 + a_i p^i + 3\Big) \mod 2^{64}$$

• **Алгоритм 2. Произвольное дерево.** $O(n \log n)$**.**

Чтобы приспособить предыдущий алгоритм для обычного некорневого дерева, можно подвесить его за центр. Если центром дерева является

ребро, то можно разделить ребро-центр новой вершиной на два ребра, таким образом, центром дерева будет только что добавленная вершина. Деревья изоморфны, если хеши центров совпадают.

• **Алгоритм 3. Избавляемся от сортировки.** $O(n)$.

Давайте для каждой вершины считать ее тип — целое число от 1 до n. Все листья будут иметь тип 1. Чтобы посчитать тип новой вершины, возьмем типы ее детей, отсортируем, посчитаем хеш от получившегося вектора и спросим у хеш-таблицы, какой тип соответствует этому хешу. Если такого хеша еще не было, хеш-таблица вернет первый не использованный тип.

Теперь можно избавиться от сортировки. Будем генерировать «вектор типов детей» автоматически в порядке возрастания:

```
list[1] <-- все листья обоих деревьев
for type = 1..n
  for v in list[type]
    p = parent[v]
    children[p].push_back(type)
    if children[p].size() = degree[p] then
      h = hash(children[p])
      t[p] = get_type(h)
      list[t[p]].push_back(parent[v])
```

Здесь мы перебираем вершины в порядке возрастания типа и добавляем наш тип в вектор отца. Деревья изоморфны, если алгоритм, запущенный от всех листьев обоих деревьев выдал, что их корни (для корневых) или центры (для произвольных) имеют одинаковый тип.

• **Алгоритм 4. Бор вместо хешей и хеш-таблицы.**

Зачем нам хеши и хеш-таблица? Чтобы по вектору из чисел от 1 до n получить число от 1 до n. Данную задачу можно решать бором. Для этого нужно вектор (строку) положить в бор и в вершине, в которой кончается вектор, хранить число от 1 до n — тип, соответствующий этому вектору. Тем не менее, так как алфавит имеет размер n, совсем от хеш-таблицы мы пока не избавились. Чтобы уметь и спускаться по бору за $O(1)$, и быстро добавлять в бор новые вектора, нужно ребра, исходящие из вершины, хранить в хеш-таблице.

• **Алгоритм 5. Полное $O(n)$. Без структур данных.**

Сейчас мы спускаемся по бору каждый раз, когда вектор уже готов. От корня до вершины-конца. Давайте вместо этого спускаться по бору постепенно, пока вектор растет, и хранить для вершины ее текущее

положение в боре. То есть, вместо

```
children[p].push_back(type)
```

нужен следующий код, двигающий позицию отца в боре

```
pos[p] = go_down(pos[p], type)
```

Теперь, поскольку `type` $\uparrow$, для каждой вершины бора нужно хранить только последнее исходящее ребро (ребро с максимальным `type`).

2 Метод двух указателей для offline задач

• **Постановка задачи**

Дан массив a, и q запросов на отрезке, не меняющих массив. Мы хотим быстро в Offline обработать все запросы. Пример запроса: «сколько различных чисел на отрезке?»

• **Случай $l_i \uparrow$ и $r_i \uparrow$**

Если отрезки можно упорядочить так, что левые и правые концы не убывают ($l_i \leqslant l_{i+1}, r_i \leqslant r_{i+1}$), то достаточно научиться обрабатывать две операции — `L++` и `R++`. Решение задачи «сколько различных чисел на отрезке?»:

```
map <int, int> cnt;
int answer = 0, L = 0, R = 0;
for (int i = 0; i < q; i++) {
  while (R <= r[i])
    if (cnt[a[R++]]++ == 0)
      answer++;
  while (L < l[i])
    if (--cnt[a[L++]] == 0)
      answer--;
  result[i] = answer;
}
```

Время работы примера $O(q \log n)$. Время работы в общем случае $2q \cdot t(n)$, где $t(n)$ — максимальное время обработки одной операции `L++` или `R++`.

• **Случай произвольных l_i и r_i**

Давайте разделим все запросы на $\sqrt{n}$ групп. i-й запрос попадает в группу номер $\lfloor \frac{l_i}{\sqrt{n}} \rfloor$. Теперь каждую группу отсортируем по r_i и обработаем отдельно. При этом в процессе обработки одной группы указатель `R`

будет только возрастать, а указатель L при переходе к следующему отрезку может как возрастать, так и убывать, но поменяется не более чем на $\sqrt{n}$.

Оценим время работы: указатель R в сумме сдвинется не более чем на $n\sqrt{n}$ ($\sqrt{n}$ групп, в каждой на n), указатель L в сумме сдвинется не более чем на $q\sqrt{n}$ (для обработки каждого из q запросов произойдет сдвиг не более $\sqrt{n}$). Получаем суммарное время работы $(q+n)\sqrt{n} \cdot t(n)$, где $t(n)$ — максимальное время обработки одной операции `l--`, `l++` или `r++`.

3 Линейная память в квадратных динамиках

• Пример задачи, решение

Есть матрица $n \times m$ из натуральных чисел, нужно дойти из $(1,1)$ в (n,m), сдвигаться можно только на $(0,1)$ или на $(1,0)$. При этом нужно максимизировать сумму чисел на пути. Стандартным решением данной задачи является следующая динамика, работающая за $O(nm)$:

```
int f[n + 1][m + 1];
f[][] <--- 0
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= m; j++)
    f[i][j] += a[i][j]
    f[i + 1][j] = max(f[i + 1][j], f[i][j])
    f[i][j + 1] = max(f[i][j + 1], f[i][j])
```

Ответ (максимальная сумма) содержится в ячейке `f[n][m]`.

• Делаем память линейной

Заметим, что из i-й строки матрицы f мы переходим только в $i+1$-ю (следующий слой). Поэтому код можно изменить следующим образом:

```
int cc = 0, f[2][m + 1];
f[cc] <--- 0
for (int i = 1; i <= n; i++)
  f[cc ^ 1] <-- 0
  for (int j = 1; j <= m; j++)
    f[cc][j] += a[i][j]
    f[cc ^ 1][j] = max(f[cc ^ 1][j], f[cc][j])
    f[cc][j + 1] = max(f[cc][j + 1], f[cc][j])
  cc ^= 1
```

Ответ (максимальная сумма) содержится в ячейке `f[cc^1][m]`.

• Восстановление пути

Чтобы восстановить путь, на котором достигается максимальная сумма, первую версию кода (с квадратной памятью) можно модифицировать, добавив массив `p[n+1][m+1]`, хранящий для каждой точки (i, j) направление, по которому мы в нее пришли. Вторую динамику таким образом не модифицировать. Для восстановления пути нам все еще нужна квадратная память.

• Восстановление пути с линейной памятью

Рассмотрим три строки: `i=1`, `i=n/2`, `i=n`. Давайте по ходу динамики насчитаем такой одномерный массив `p[m+1]`, что оптимальный путь из $(1, 1)$ в (n, x) проходит через точку $(n/2, p[x])$. Теперь мы знаем, что оптимальный путь устроен так: $(1, 1) \to (n/2, p[m]) \to (n, m)$. Чтобы восстановить путь полностью, нужно реализовать нашу динамику, как рекурсивную функцию, которая ищет путь из (y_1, x_1) в (y_2, x_2) за время $O((y_2 - y_1 + 1) \cdot (x_2 - x_1 + 1))$, и вызвать ее от пары точек $(1, 1)$ и $(n/2, p[m])$ (первая половина пути), а затем от $(n/2, p[m])$ и (n, m) (вторая половина пути). Грубо оценим время работы динамики: $T(n, m) = nm + T(n/2, x) + T(n/2, m - x) = nm + \frac{n}{2}m + T(n/4, x_1) + T(n/4, x - x_1) + T(n/4, x_2) + T(n/4, (m - x) - x_2) = \cdots = nm + \frac{n}{2}m + \frac{n}{4}m + ... \leqslant 2nm$. Здесь я пользуюсь тем, что на каждом уровне сумма x-ов будет равна m. Грубость оценки заключается в том, что для упрощения технической сложности оценки здесь намерено опущена «плюс единица».

• Применяем обретенное знание к задаче о рюкзаке

Постановка задачи: дан рюкзак вместимости s и n вещей со стоимостями c_i и размерами w_i. Каждую вещь можно или целиком положить в рюкзак, или целиком не взять. Нужно максимизировать суммарную стоимость вещей в рюкзаке. Стандартным решением данной задачи является следующая динамика, работающая за $O(ns)$:

```
int f[n + 1][s + 1];
f[][] <--- 0
for (int i = 0; i < n; i++)
  for (int j = 0; j <= s; j++)
    f[i + 1][j] = max(f[i + 1][j], f[i][j])
    if (j + w[i] <= s[i])
      f[i + 1][j + w[i]] = max(f[i + 1][j + w[i]], f[i][j] + c[i])
```

Отличие этой динамики от предыдущей в том, что переходы теперь не $(1, 0)$ и $(0, 1)$, а $(1, 0)$ и $(1, w[i])$. Мы посчитали оптимальную стоимость. Чтобы получить не только стоимость вещей в рюкзаке, но и сам набор,

нам нужна квадратная память[1], но можно применить такую же оптимизацию, как и выше. Точка $(n/2, p[x])$ будет соответствовать тому, что первые $n/2$ вещей должны занять ровно $p[x]$ места в рюкзаке, соответственно, динамика распадается на две независимых — разместить первые $n/2$ вещей в рюкзаке размера $p[x]$, и разместить оставшиеся $n - n/2$ вещей в рюкзаке размера $s - p[x]$.

4 Жадность

• Сортируем и радуемся

Для начала рассмотрим несколько простых задач, которые решаются сортировкой + циклом `for`.

1. **Условие.** Есть заказы, у каждого заказа есть время выполнения t_i. Заказы можно выполнять в любом порядке. Мы начинаем в момент времени 0 и к моменту времени d должны выполнить максимально возможное количество заказов.

   Решение. Выполняем заказы в порядке возрастания t_i.

2. **Условие.** Есть заказы, у каждого заказа есть время выполнения t_i и свой deadline d_i. Заказы можно выполнять в любом порядке. Мы начинаем в момент времени 0 и должны успеть выполнить **все** заказы до их deadline-ов.

   Решение. Выполняем заказы в порядке возрастания d_i.

3. **Условие.** Есть коробки, у каждой коробки есть масса m_i и максимальная суммарная масса, которую можно поставить на коробку сверху, чтобы коробка не сломалась w_i. Нужно построить вертикальную башню, используя **все** коробки.

   Решение. Башня снизу вверх = коробки в порядке убывания $m_i + w_i$. Доказательство: самая нижняя коробка обладает свойством $w_i \geqslant \sum\limits_{j \neq i} m_j \Rightarrow w_i + m_i \geqslant \sum\limits_{j} m_j = const \Rightarrow$ вниз можно смело ставить коробку с максимальным $w_i + m_i$.

4. **Условие.** Есть заказы, у каждого заказа есть время выполнения t_i и штраф w_i. Если i-й заказ начать выполнять в момент времени

[1] Для задачи о рюкзаке без стоимостей (например, набрать вещей суммарного размера ровно s) можно восстановить ответ без извращений, используя также линейную память. Тем не менее, в нашем случае без извращений не получится.

C_i, то к суммарному штрафу прибавится величина $w_i C_i$. Заказы можно выполнять в любом порядке. Мы начинаем в момент времени 0. Нужно выполнить все заказы, минимизируя суммарный штраф.

Решение. Выполняем заказы в порядке убывание $\frac{w_i}{t_i}$. Доказательство: рассмотрим два соседних (в порядке выполнения) заказа, выпишем условие, когда их выгодно поменять местами. Условие: $w_2 t_1 > w_1 t_2$.

5. **Условие.** Даны n строк, нужно их сконкатенировать в таком порядке, чтобы конкатенация была лексикографически минимальна.

   Решение. Сортируем строки с помощью следующей функции сравнения
   `bool less(string a, string b) { return a + b < b + a; }`
   Доказательство: см. предыдущую задачу.

Сформулируем общий принцип жадного решения задач сортировкой. Нужно придумать правильный компаратор. Чтобы его придумать, достаточно рассмотреть случай $n = 2$ и понять, какой из порядков $(1, 2)$ или $(2, 1)$ выгодней. Один из способов понять, какой из порядков выгодней — ввести функцию оценки. Для задачи (4) использовалась функция оценки $F(1, 2) = w_2 t_1$ «суммарный штраф, который мы заплатим». Для задачи (5) использовалась функция оценки $F(s, t) = st$ «конкатенация». Для произвольной функции F компаратор выглядеть так:
`bool less(type a, type b) { return F(a, b) < F(b, a); }`
Заметим также, что, если отношение получилось нетранзитивным, решение заведомо некорректно.

• **Еще две задачи на сортировку**

Решим две более сложные задачи на тему «сортировка».

1. **Условие.** Задача «Два конвейера»: есть n деталей и два конвейера, i-ю деталь нужно обрабатывать сперва a_i единиц времени на первом конвейере, затем b_i на втором. Порядки выполнения деталей на первом и втором конвейерах должен быть одинаковы. Нужно выбрать порядок обработки деталей так, чтобы минимизировать момент времени, когда все детали обработаны.

   Решение. Решаем задачу для $n = 2$. $F(1, 2) = a_1 + \max(a_2, b_1) + b_2$. Используем полученную функцию для создания компаратора:
   $i < j \Leftrightarrow a_i + \max(a_j, b_i) + b_j < a_j + \max(a_i, b_j) + b_i$
   Используя то, что $a_i + a_j + b_i + b_j = const$, получаем более короткий вариант: $i < j \Leftrightarrow \min(a_j, b_i) > \min(a_i, b_j)$

2. **Условие.** Даны несколько строк, состоящих из круглых скобок. Нужно сконкатенировать их в некотором порядке так, чтобы конкатенация была правильной скобочной последовательностью.

Решение. *Балансом* назовем количество уже открытых скобок минус количество уже закрытых скобок. Каждая строка характеризуется двумя параметрами — минимальный баланс внутри (или на концах) строки (a_i) и балансом в конце строки (b_i). $a_i \leqslant 0$, $a_i \leqslant b_i$. Сперва возьмем в любом порядке все строки, у которых $a_i \geqslant 0$. Затем будем увеличивать баланс (т.е. возьмем все $b_i \geqslant 0$). Их нужно отсортировать по убыванию a_i. Остались только строки вида $a_i \leqslant b_i < 0$. Отсортируем их универсальным компаратором, максимизируя «минимальный баланс». $i < j \Leftrightarrow \min(b_i, a_i + b_j) > \min(b_j, a_j + b_i)$. Если учесть, что $a_i + b_j < b_j$ и $a_j + b_i < b_i$, получается $i < j \Leftrightarrow a_i + b_j > a_j + b_i \Leftrightarrow a_i - b_i > a_j - b_j$

• Сортируем, затем динамика

Давайте усложним «задачу про коробки», «задачу про скобки» и «задачу про заказы с deadline-ами».

1. **«Задача про коробки».** Теперь не обязательно использовать все коробки. Нужно построить башню, используя максимально возможное количество коробок.

   Решение. Рассмотрим ответ. Те коробки, которые в него входят, можно брать в порядке, заданном уже разобранной жадностью. Давайте все исходные коробки отсортируем в таком порядке. Тогда башня — какая-то подпоследовательность массива коробок. Будем строить башню сверху вниз. Посчитаем динамику M_{ij}, где i — количество уже рассмотренных коробок, j — сколько из них мы взяли в башню, M_{ij} — минимальная возможная суммарная масса коробок в башне или $+\infty$, если такую башню построить невозможно. Очередную коробку мы можем не включить в башню или, если $w_i \geqslant M_{ij}$, включить. Ответ на задачу — такое максимальное j : $M_{nj} \neq +\infty$. Получили решение за $O(sort) + O(n^2)$.

2. **«Задача про скобки».** Теперь не обязательно использовать все строки. Суммарная длина строк не более s. Нужно взять конкатенацию некоторых строк так, чтобы чтобы получился префикс некоторой правильной последовательности (т.е. в середине баланс никогда не был меньше нуля, а в конце он может быть положительным), а длина конкатенации была максимально возможной. Решаем также, как и «коробки». B_{ij} — максимально возможный баланс в конце строки, если мы уже рассмотрели i строк и взяли из них несколько суммарной длины j, или -1, если нельзя так сделать, не опуская баланс ниже нуля. Ответ на задачу — такое максимальное j :

$B_{nj} \neq -1$. Получили решение за $O(sort)+O(ns)$. Задача доступна на тимусе под номером 1745.

3. **«Заказы с deadline-ами».** В прошлой задаче у каждого заказа был свой deadline d_i и время выполнения t_i, нужно было успеть выполнить **все** заказы. Решением была сортировка по возрастанию d_i. Новая версия: нужно успеть выполнить как можно больше заказов. Решение = (сортировка по возрастанию d_i) + (динамика T_{ij}), где i — сколько заказов мы уже рассмотрели, j — сколько из них мы уже выполнили, T_{ij} — минимальное суммарное потраченное время.

• Избавляемся от динамики

На примере задачи «Заказы с deadline-ами» покажем, как можно решить задачу без динамики и получить решение за $O(n \log n)$.

Способ 1.

Наблюдение: если мы уже выбрали множество заказов, которые собираемся выполнить, можно выполнить их в порядке возрастания d_i. Решение: отсортируем по возрастанию d_i и в таком порядке будем пробовать заказы брать. Если очередной заказ взять не получается (т.е. мы не успеем его выполнить к моменту его deadline), пробуем заменить какой-то уже взятый на наш. Заменять нужно, если $t_i < t_{max}$, где t_{max} — максимальное время выполнение по всем уже взятым заказам. Чтобы поддерживать величину t_{max}, используем `set<int>`.

Способ 2.

Наблюдение: если $\forall i : d_i \geqslant t_i$ (а если нет, то можно часть заказов сразу выкинуть), то всегда выгодно выполнить заказ с минимальным t_i (не факт, что первым, но точно выполнить). Доказательство: пусть мы не взяли заказ с минимальным t_i, возьмем первый заказ в порядке выполнения и заменим на заказ с минимальным t_i. Ответ не ухудшился, доказали. Получили решение: сортируем заказы в порядке возрастания t_i. В таком порядке пытаемся добавлять заказы в множество $A = \{$заказы, которые точно нужно выполнить$\}$. Чтобы проверить, что можно добавить заказ j в множество A, отсортируем A по d_i и посмотрим, что нет противоречий. Чтобы проверить за $O(\log n)$, можно ли добавить заказ j, нужно хранить A в декартовом дереве по d_i с функцией в вершине $f_i = d_i - \sum\limits_{j \in \text{левое поддерево}} t_j$ а также функцией в вершине $m_i = \min\limits_{j \in \text{поддерево}} f_j$.

5 Количество деревьев с точностью до изоморфизма

• **Помеченные деревья**

Пусть все вершины имеют номера $1, 2, \dots n$, и поэтому различны. Тогда деревья $(1,2)$, $(2,3)$ и $(1,3)$, $(3,2)$ различаются. Теорема Келли говорит, что количество помеченных деревьев из n вершин $= n^{n-2}$. Для доказательства этого факта, можно, например, закодировать дерево Кодом Прюфера: последовательно выбираем лист с наименьшим номером, удаляем его из дерева и записываем номер его отца. Получили массив из $n-1$ числа, последнее число всегда равно n. Если мы научимся по коду Прюфера однозначно восстанавливать дерево, мы получим, что количество деревьев = количеству различных кодов Прюфера $= n^{n-2}$. Восстановление дерева по коду Прюфера предлагается придумать самостоятельно.

• **Непомеченные деревья**

Классом эквивалентности деревьев назовем максимальное по включению множество попарно изоморфных деревьев. Количество непомеченные деревьев из n-вершин — количество различных классов эквивалентности деревьев из n вершин. Давайте научимся считать это количество. Обозначим за $f_{n,d}$ количество непомеченных корневых деревьев из n вершин глубины d (глубина дерева из одной вершины равна единице).

1. Ответ на задачу выражается через $f_{n,d}$ следующим образом:
$$\sum_{d=1}^{2d \leqslant n+1} \left((x + \sum_{a=1}^{2a<n} f_{a,d} \cdot f_{n-a,d}) + (f_{n,d} - \sum_{a=1}^{a<n} f_{a,d-1} \cdot F_{n-a,d}) \right)$$
здесь $F_{n,d} = \sum_{i=1}^{i<d} f_{n,i}$, а $x = D(f_{n/2,d}, 2)$, если n кратно двум, и нулю в ином случае. $D(n,k)$ — количество способов выбрать из n объектов k упорядоченных объектов с повторениями. $D(n,k) = \binom{n+k-1}{k}$ В этой длинной формуле первые два слагаемых соответствуют случаю «центр дерева — ребро», а вторые два случаю «центр дерева — вершина».

2. У каждого корневого дерева есть две основных характеристики — количество вершин n и глубина d. Назовем пару (d,n) типом дерева. Типы естественным образом упорядочиваются — сперва по d, затем по n. Чтобы посчитать $f_{n,d}$, заведем внутреннюю динамику $g_{n,d,x}$, которая будет набирать в поддерево детей в порядке возрастания типа. $g_{n,d,x}$ — количество способов набрать дерево из n вершин, у корня которого все дети имеют тип строго меньше (d,x). Тогда $f_{n,d} = g_{n,d,1} - g_{n,d-1,1}$ (все деревья из n вершин высоты строго меньше

$d+1$ минус все деревья из n вершин высоты строго меньше d, плюс единица берется от того, что мы учли корень дерева).

3. Осталось выписать формулу пересчета для $g_{n,d,x}$.
$$g_{n,d',x'} = \sum_{a=0}^{ax<n} g_{n-xa,d,x} \cdot D(f_{x,d}, a)$$
Здесь (d', x') — следующий тип после (d, x)
(`d' := d, x' = x + 1; if (x' = maxN) { d'++, x' := 0 }`)

4. База динамики: $f_{1,1} = 1$, $g_{1,1,1} = 1$.

• Непомеченные деревья, оптимизируем решение

1. С точки зрения олимпиад прежде всего заметим, что все значения можно предподсчитать.

2. Заметим, что динамику g можно разбить по слоям по d (т.к. есть только переходы $d \to d, d+1$). Сделаем это, теперь у нас квадратная память $\Rightarrow$ кэшируется лучше $\Rightarrow$ работает быстрее.

3. При $a > 0$ и $f_{x,d} = 0$ мы получаем, что $D(f_{x,d}, a) = 0$, поэтому, если $f_{x,d} = 0$, и a уже больше нуля, делаем `break` из цикла по a.

4. D-шки, как и C-шки (сочетания из n по k) быстро пересчитываются. $D(f, k+1) = \binom{f+k}{k+1} = \binom{f+k-1}{k} \cdot \frac{f+k}{k+1} = D(f, k) \cdot \frac{f+k}{k+1}$ Таким образом каждую следующую $D(f_{x,d}, a)$ мы будем получать за $O(1)$.

На этом можно сказать, что мы используем $O(n^2)$ памяти и $O(n^3 \log n)$ времени, и остановиться (логарифм вылез из условия $ax < n$, количество пар a и x, удовлетворяющих этому ограничению, $O(n \log n)$).

• Непомеченные деревья, учимся брать по модулю

В задаче на контесте предлагалось посчитать количество непомеченных деревьев из n вершин по модулю 2^{32}. Складывать, вычитать и умножать по этому модулю очень удобно — считаем все в типе `unsigned int`, не обращая внимания на переполнения. Делить на нечетное число можно используя стандартный метод: $a^{-1} \mod 2^{32} = a^{\varphi(2^{32})-1} \mod 2^{32} = a^{2^{31}-1} \mod 2^{32} = (a^{2^{16}-1})^2 \cdot a \mod 2^{32} = \ldots$ Таким образом мы можем посчитать обратное число за 10 умножений. Вопрос в том, как делить на четное число? Делить на четное число нам может понадобиться только во время вычисления $\binom{n}{k}$. Причем $\binom{n}{k}$ — целое число, поэтому достаточно

научиться только умножать и делить (сложения и вычитания не нужны). Числа вида $2^k a$, где a нечетное, образуют группу по умножению. Умножение: $(k_1, a_1) \cdot (k_2, a_2) = (k_1 + k_2, a_1 + a_2)$, Деление: $(k_1, a_1)/(k_2, a_2) = (k_1 - k_2, a_1/a_2)$. Поэтому будем считать $\binom{n}{k}$, используя числа именно такого вида. Осталась только одна проблема, по ходу алгоритма нужно считать $\binom{f_{x,d}}{k}$, где $f_{x,d}$ уже взято по модулю 2^{32}. Вопрос: изменится ли остаток «сочетания из n по k» по модулю 2^{32}, если n посчитать по модулю 2^{32}? Ответ: изменится. Решение проблемы: будем считать все по модулю 2^{64}, утверждается, что остаток по модулю 2^{32} будет посчитан верно :-)

6 Два указателя в динамике

- **Задача про профессора и транзисторы**

1. **Условие.** У профессора есть k транзисторов и n-этажный дом. Профессор точно знает, что если бросить транзистор с первого этажа, он не разобьется, если его бросить с n-го этажа, то разобьется, а также что функция $f(i)$ «разобьется ли транзистор, если его бросить с i-го этажа» монотонна. Помогите профессору за минимальное количество бросков транзисторов в худшем случае определить такое i, что $x(i) = 0, x(i+1) = 1$.

2. **Решение.** Пусть $f_{n,k}$ — ответ на задачу с параметрами (n, k). По условию $f_{2,k} = 0$, $f_{n,k} = \min\limits_{i=2..n-1} \max(f_{i,k-1}, f_{n-i+1,k})$. Если посчитать в лоб, получается $O(n^2 k)$.

3. **Мысль 1.** Если $k \geqslant \log_2 n$, то можно делать двоичный поиск, поэтому можно сделать в начале $k = min(k, \lceil \log_2 n \rceil)$. Теперь решение работает за $O(n^2 \log n)$.

4. **Мысль 2.** $f_{i,k} \leqslant f_{i+1,k}$, поэтому $f_{i,k-1}\downarrow$, $f_{n-i+1,k}\uparrow$, и минимум функции $\max(f_{i,k-1}, f_{n-i+1,k})$ можно найти бинарным поиском (ищем корень функции $f_{i,k-1} - f_{n-i+1,k}$). Теперь решение работает за $O(n \log^2 n)$.

5. **Мысль 3.** Обозначим за $p_{n,k}$ оптимальное i, которое используется в формуле пересчета. Тогда $p_{n,k} \leqslant p_{n+1,k}$ и можно использовать метод двух указателей: изначально делаем $p_{n+1,k} = p_{n,k}$ и, пока функция не ухудшается, увеличиваем $p_{n+1,k}$ на 1. Теперь решение работает за $O(n \log n)$.

6. **Эпилог.** На самом деле данная задача имеет решение для $n, k \leqslant 10^{18}$, для этого достаточно заметить, что $f_{n,1} = n - 2$, а $f_{n,2} \approx n^{\frac{1}{2}}$, а $f_{n,k} \approx n^{\frac{1}{k}}$, более-менее точно подобрать формулы (ограничения снизу и сверху) и считать динамику лениво, пропуская все заведомо неоптимальные состояния.

• Разбор задачи про суммы (Day1-G)

1. **Условие.** Есть упорядоченный массив из n чисел, нужно выбрать k из них так, чтобы сумма расстояний от каждого из n чисел до ближайшего из k выбранных была минимально возможной.

2. **Решение.** Заметим, что массив из n чисел в результате разобьется на k отрезков. Ответ для одного отрезка можно посчитать за $O(1)$ — нужно выбрать среднее число (если чисел на отрезке k, берем число с номером $\frac{k}{2}$, округляем в любую сторону), когда число выбрано, сумму мы получаем за $O(1)$, используя предподсчитанные частичные суммы. Пусть $f_{i,j}$ — минимальная сумма расстояний, которая может получиться, если первые i чисел разбить на j отрезков. База этой динамики уже есть, мы знаем ответы для $j = 1$. Сделаем переход:

$$f_{i,j} = \min_{p=1\ldots i}(f_{p,j-1} + s_{p+1,i})$$

где $s_{l,r}$ — значение суммы для отрезка $[l \ldots r]$. Если отрезок пуст, $s_{l,r} = 0$. Данная динамика имеет $O(nk)$ состояний и считается за время $O(n^2k)$.

3. **Оптимизируем решение.** $p_{n+1,k} \geqslant p_{n,k}$. Иначе, откинув последнее число, мы бы получили более хорошее разбиение на отрезки для состояния (n, k). Аналогично $p_{n,k} \geqslant p_{n-1,k}$. Теперь менее тривиальный факт: $p_{n,k-1} \leqslant p_{n,k} \leqslant p_{n,k+1}$. Короткое доказательство я к сожалению привести не могу. Тем не менее, насчитав все $p_{n,k}$ и $f_{n,k}$, этот факт легко проверить на существующих тестах. Из двух вышеприведенных неравенств получаем следующее:

$$p_{n,k-1} \leqslant p_{n,k} \leqslant p_{n+1,k}$$

Давайте будем перебирать $p_{n,k}$ ровно в таком диапазоне. Можно считать $p_{n+1,i} = n$. Суммарное время работы $=$

$$\sum_{i=2}^{k}\sum_{j=1}^{n}(p_{j+1,i} - p_{j,i-1} + 1) = O(n^2)$$

Заметим, что почти все $p_{i,j}$ присутствуют в сумме, как со знаком плюс, так и со знаком минус. Т.е., все успешно сократится. Поэтому $O(n^2)$. Покажем также, что при $k \geqslant 2$ время работы может быть порядка n^2. Пусть $k = 2$, $a_i = i$, тогда выгодно делить равномерно, т.е. $p_{n,k} = n - \frac{n}{k}$. Все значения при $k = 1$ посчитаны за $O(1)$, посчитаем время потраченное для $k = 2$: $\sum\limits_{j=1}^{n}(p_{j+1,2} - p_{j,1} + 1) = \sum\limits_{j=1}^{n}(\frac{j+1}{2} - 0 + 1) \approx \frac{n^2}{4}$

• Разбор задачи command-post

1. **Условие.** Даны n различных точек на окружности, нужно выбрать $3 \leqslant k$ из них так, чтобы площадь получившегося многоугольника была максимально возможной, и центр окружности обязательно лежал внутри. Второе условие в принципе можно опустить, так как, если центр не лежит внутри, то, или невозможно поместить центр внутрь многоугольника, или текущий многоугольник не максимален.

2. **Решение.** Состояние: мы взяли точку l, точку r, и на отрезке $(l \ldots r)$ хотим взять еще k точек. Начальное состояние $[i, i, k-1]$ для всех i. Пара (i, i) означает, что мы должны сделать полный круг. Переход:
$$f_{l,r,k} = \max_{m=l+1..r-1}(f_{l,m,0} + f_{m,r,k-1})$$
Получили решение за $O(n^3 k)$.

3. **Мысль 1.** Давайте разбивать k точек не на 1 и $k-1$, а на $k/2$ и $k - k/2$. Тогда различных k будет $O(\log k)$, а суммарное время работы $O(n^3 \log k)$.

4. **Мысль 2.** Оптимальное m будем запоминать в $p_{l,r}$. Утверждается, что $p_{l,r-1} \leqslant p_{l,r} \leqslant p_{l+1,r}$. Будем перебирать $p_{l,r}$ в указанном промежутке. Получим суммарное время работы для каждого k равное $\sum\limits_{l,r}(p_{l+1,r} - p_{l,r-1})$.
Почти все $p_{i,j}$ присутствуют в сумме, как со знаком плюс, так и со знаком минус, поэтому сумма равна $O(n^2)$. Получили асимптотику $O(n^2 \log k)$.

7 Разбиение числа на слагаемые

Условие задачи: для фиксированного числа n посчитать количество разбиений этого числа на слагаемые. При этом разбиения $1+2$ и $2+1$ считаются одинаковыми.

- **Максимальное слагаемое не более k**

Если можно использовать одинаковые слагаемые, то

$$f_{n,k} = f_{n-k,k} + f_{n,k-1}$$

и решение выглядит так:

```
f[0] = 1
for (x = 1; x <= k; x++)
  for (i = 0; i + x <= n; i++)
    f[i + x] += f[i]
```

Если нельзя использовать одинаковые слагаемые, то

$$f_{n,k} = f_{n-k,k} + f_{n-k,k-1}$$

и решение выглядит так:

```
f[0] = 1
for (x = 1; x <= k; x++)
  for (i = n - x; i >= 0; i--)
    f[i + x] += f[i]
```

- **Количество слагаемых не более k**

Если можно использовать одинаковые слагаемые, то

$$f_{n,k} = f_{n-k,k} + f_{n,k-1}$$

Доказательство: или уменьшили все слагаемые на 1, или сказали, что меньшее из слагаемых равно нулю. Если нельзя использовать одинаковые слагаемые, то

$$f_{n,k} = f_{n-k,k} + f_{n-k,k-1}$$

Почему получились формулы такие же, как для двух предыдущих задач? Если слагаемые нарисовать подряд идущими клетчатыми столбцами, причем высота очередного столбца равна величине соответствующего слагаемого, то получится так называемая диаграмма Юнга. Заметим, что ее можно разворачивать на 90 градусов (смотреть на нее под другим

углом). Количество разбиений числа n на слагаемые равно количеству диаграмм Юнга из n клеток. Количество столбцов в диаграмме равно количеству слагаемых, высота максимального столбца в диаграмме равно максимальному слагаемому. При повороте на 90 градусов понятия «количество столбцов» и «высота максимального столбца» поменяются местами.

- **Не более k и ровно k**

  Если f_k — ровно k, а F_k — не более k, то $f_k = F_k - F_{k-1}$, а $F_k = \sum\limits_i f_i$

- **k различных слагаемых**

Задачу про k различных слагаемых можно свести к задаче про произвольные слагаемые. Для этого нужно из i-го слагаемого вычесть i. Получилось, что теперь число $n - \frac{k(k+1)}{2}$ нужно разбить на k произвольных слагаемых, а заодно мы получили, что $k \leqslant \sqrt{n}$, поэтому задача решается за $O(n\sqrt{n})$

8 Эквивалентность некоторых задач про жадность

Сформулируем задачи и введем обозначения:

1. **«Коробки».** У каждой коробки есть масса m_i и максимальный вес w_i, который коробка может выдержать. Нужно построить максимальную по количеству коробок башню.

2. **«Заказы с deadline-ами».** У каждого заказа есть deadline d_i и время выполнения t_i. Нужно максимальное количество заказов успеть выполнить до наступления deadline-а.

3. **«Школьники в яме».** Школьники пытаются выбраться из ямы глубины H. У каждого школьника есть два параметра — длина рук l_i и высота h_i. Школьники могут вставать друг на друга (выносливость школьника бесконечна). Если все школьники выстроились в башню и $l_i + \sum h_j \geqslant H$, то верхний школьник может вылезти из ямы. Нужно придумать такой порядок вылезания школьников из ямы, что спастись сможет максимальное количество школьников.

4. **«Авторитеты»** У Толика есть изначальный авторитет A. Есть n людей. i-й человек навсегда присоединится к Толику, если авторитет Толика хотя бы a_i и Толик его пригласит. После того, как i-й человек присоединится к Толику, авторитет Толика увеличится на b_i. И a_i, и b_i — целые, возможно отрицательные, числа. Нужно выбрать стратегию для Толика, чтобы в итоге количество его сторонников было максимально возможным.

• **«Коробки» $\leftrightarrow$ «Заказы с deadline-ами»**

$\sum m_j \leqslant w_i$
$m_i + \sum m_j \leqslant w_i + m_i$
$d_i = w_i + m_i$, $t_i = m_i$ (все $t_i > 0$, все $m_i > 0$)

• **«Коробки» $\leftrightarrow$ «Школьники в яме»**

Обозначим $\sum h_i$ за S, рассмотрим людей сверху вниз.
$l_i + (S - \sum h_j) \geqslant H$ (i — текущий человек, j — люди над ним)
$(S - H) + l_i \geqslant \sum h_j$ (перенесли и сгруппировали)
Получили $m_i = h_i$, $w_i = l_i + (S - H)$ (все $m_i > 0$, все $h_i > 0$)

• **«Коробки» $\leftrightarrow$ «Авторитеты»**

Задача про авторитеты решается в два этапа. Сперва пытаемся взять всех людей, повышающих авторитет ($b_i \geqslant 0$). Их мы просто отсортируем по возрастанию a_i и в таком порядке будем пытаться брать. Теперь у нас есть некоторый авторитет A, и каждый очередной человек уменьшает авторитет ($b_i < 0$).

$A + \sum b_j \geqslant a_i$ (i — текущий человек, j — люди с отрицательным b_j, которых мы уже взяли)
$-\sum b_j \leqslant A - a_i$
Получили $w_i = A - a_i$, $m_i = -b_i$ (все $m_i > 0$, все $b_i < 0$)

• **Эпилог**

Поскольку в конце главы про жадность мы уже показали, что задача «Заказы с deadline-ами» решается за $O(n \log n)$ с использованием только сортировки и `set<int>`, можно сделать вывод, что задачи «Коробки», «Школьники в яме», «Авторитеты» решаются такой же жадностью за время $O(n \log n)$.

9 Разбор

• **A. Белоснежка и n гномов.**

Сортировка по $a_i + b_i$.

• **B. Эльфы и Олени.**

Отсортируем массивы a и b по возрастанию. Запускаем бинарный поиск по ответу. Чтобы проверить, что ответ $\geqslant k$, говорим, что i-я пара эльфов $= (b_i, b_{n-k+i})$, и запускаем метод двух указателей по «массиву оленей» и «массиву пар эльфов».

• **C. Приключение.**

См. лекцию. «Часть 8, задача 3», а также «Часть 4, задача про заказы с deadline-ами».

• **D. Авторитеты.**

См. лекцию. «Часть 8, задача 4».

• **E. Коробки.**

См. лекцию. «Часть 8, задача 1».

• **F. Простые пути в дереве.**

Подвесим дерево за превую вершину. Первым `dfs`-ом насчитаем для каждой вершины v $s_1[v]$ — количество вершин в поддереве вершины v, и $s_2[v]$ — суммарную длину всех путей от вершины v вниз. Вторым `dfs`-ом по дереву для каждого ребра насчитаем ответ. Каждое ребро имеет два конца — нижний конец a и верхний конец b. Количество вершин снизу — $s_1[a]$, количество вершин сверху $t_1[b] = n - s_1[a]$, суммарная длина путей вниз — $s_2[a]$, суммарную длину путей вверх $t_2[b]$ можно пересчитывать, при спуске вниз: $t_2[a] = t_2[b]+t_1[b]+s_2[b]-(s_2[a]+s_1[a])$. База: $t_2[root] = 0$. Ответ для ребра (a, b) выражается так: $s_1[a] \cdot t_2[b] + s_2[a] \cdot t_1[b] + s_1[a] \cdot t_1[b]$.

• **G. Редукция дерева.**

Подвесим дерево. Для каждой вершины v насчитаем `f[v,k]` — сколько ребер в поддереве вершины v нужно разрезать, чтобы компонента связности с корнем в вершине v имела размер ровно k. Для удобства пересчета и оценки времени работы предположим, что дерево бинарное. В противном случае нужно будет делать внутреннюю динамику по детям вершины v. Итак, дерево бинарное, переход делается так:

```
for (l = 1; l <= left_size; l++)
  for (r = 1; r <= right_size; r++)
    f[v, l+r] = min(f[v, l+r], f[left, l] + f[right, r])
```

Время работы алгоритма $\sum_{v=1..n}$ (`left_size[v]`·`right_size[v]`). Утверждается, что это $O(n^2)$. Подробнее смотрите здесь:
`http://codeforces.ru/blog/entry/6703#comment-122804`

• **H. Изоморфные деревья.**

См. лекцию. «Часть 5».

• **I. К минимумов на отрезке.**

См. лекцию. «Часть 2».

Операции `L++`, `L--`, `R++`, `R--` делаем с помощью `multiset<int>`
Получаем решение за $O((n+q)\sqrt{n}\log n + qk)$

• J. Инверсии отрезка.

См. лекцию. «Часть 2».

Множество чисел, соответствующее отрезку $[L..R]$ храним в дереве Фенвика. Операции `L++`, `L--`, `R++`, `R--` обрабатываем с помощью дерева Фенвика за $O(\log n)$.

• K. Продуктивный бинпоиск.

Сперва выпишем динамику по подотрезкам за $O(n^3)$.

$$f_{l,r} = \min_{m=l+1\ldots r-1}(\max(f_{l,m-1}, f_{m+1,r}) + m)$$

Заметим, что $f_{l+1,r} \leqslant f_{l,r} \leqslant f_{l,r+1}$. Поэтому $f_{l,m-1}$ возрастает по m, а $f_{m+1,r}$ убывает по m. Найдем минимальное p: $f_{p+1,r} \leqslant f_{l,p-1}$. Позиции, большие, чем p, рассматривать не имеет смысла, так как $(f_{l,m-1} + m)$ возрастает на $[p \ldots r)$. Получаем:

$$f_{l,r} = \min(f_{l,p-1} + p, \min_{m=l+1\ldots p-1}(f_{m+1,r} + m))$$

Зафиксируем l. Будем перебирать r в порядке убывания. Позиция $p_{l,r}$ при этом убывает и пересчитывается следующим образом:
`while (p - 1 > l && f[l][p - 2] > f[p][r]) p--;`
Минимум легко считается за $O(1)$, так как границы отрезка, на котором нужно считать минимум, $l+1$ и $p-1$, обе убывают. Получили решение за $O(n^2)$.

• L. Командный пункт.

См. лекцию. «Часть 6, разбор задачи command-post».

• M. Общая подпоследовательность.

См. лекцию. «Часть 3».

• N. Различные подпоследовательности.

`f[i]` — количество уникальных последовательностей, конец которых — в точности i-й элемент массива. `prev[i]` — ближайшая слева позиция, такая что: `a[prev[i]] = a[i]`. Следующий код решает задачу, и может быть реализован за $O(n)$ с использованием частичных сумм.

```
f[0] = 1
for (i = 1; i <= n; i++)
  f[i] = сумма f[prev[i]..i-1]
result = сумма f[1..n]
```

• **O. Наибольшая общая возрастающая.**

$f_{i,j}$ — максимальная длина общей подпоследовательности, где i — конец подпоследовательности `a`, и j — любая позиция большая конца в `b`. Переходы:

1. $(i, j) \to (i, j + 1)$ Означает, что мы пропускаем элемент номер j последовательности b.

2. `if b[j] > a[i]` : $(i, j) \to (\texttt{next}(i, b_j), j + 1)$ Означает, что пару $(\texttt{next}(i, b_j), j)$ мы взяли в общую подпоследовательность. Здесь $\texttt{next}(i, b_j), j$ — ближайший справа от i элемент, равный b_j, в последовательности `a`. Чтобы реализовать функцию `next`, удобно сжать координаты. После этого $1 \leqslant a_i, b_i \leqslant 2n$ и можно создать для каждого число от 1 до $2n$ упорядоченный список позиций в последовательности `a`.

• **P. k-Рюкзак.**

Предполагаем, что $k_i = 1$, получаем решение за $O(ns)$ времени с $O(s)$ памяти. Пусть текущий слой динамики f_0, следующий слой динамики f_1. Научимся делаеть переход для группы вещей (w_i, c_i, k_i) за $O(s)$. Можно разбить переход на w_i частей по остаткам по модулю w_i: $0 \dots w_i - 1$. При фиксированном остатке r мы хотим посчитать $f_1[r], f_1[r + w_i], f_1[r + 2w_i], \dots$

$$f_1[r + tw_i] = \min_{j=0..k} (f_0[r + (t - k)w_i] + kc_i)$$

Левая и правая граница возрастают, поэтому минимум на отрезке легко считается за $O(1)$.

• **Q. Разбиение на слагаемые.**

См. лекцию. «Часть 7».

• **R. Волшебный лес.**

Решение — жадность.

Очевидный критерий существования ответа: сумма сумм в строках равна сумме сумм в столбцах. Далее предполагаем, что эти суммы равны, тогда ответ существует. Конструктивное доказательство: есть строка с положительной суммой, столбец с положительной суммой, увеличиваем значение в матрице на пересечении данных строк и столбца на 1, уменьшаем сумму в строке на 1 и сумму в столбце на 1, повторяем процесс, пока все суммы не обнулятся.

Делаем бинарный поиск по ответу. Внутри проверяем, что ответ $\leqslant x$, следующей процедурой:

```
Check(x)
  Отсортировали столбцы в порядке убывания суммы
  sum_col = 0
  for (i = 1; i <= n; i++)
    sum_col += сумма в i-м столбце
    sum_row = 0
    for (j = 1; j <= n; j++)
      sum_row += min(row[j], i * x)
    if sum_row < sum_col
      return false
  return true
```

После того, как мы знаем, что ответ $= x$, нужно расставить числа в матрице. Перебираем строки. Расставляем числа в i-й строке. Когда в некоторую ячейки матрицы ставим число, уменьшаем сумму в соответствующем столбце. Нужно пытаться в первую очередь уменьшить столбцы с большой суммой. Определяем бинарным поиском максимальное y, что все столбцы можно уменьшить или на x, или до уровня y. После этого уменьшаем все столбцы до уровня y и еще какие-то столбцы до уровня $y - 1$.

Все в сумме работает за $O(n^2 \log M)$. Вторую часть (восстановление матрицы) при желании можно реализовать без использования бинарного поиска за $O(n^2)$.

К О Н Е Ц