ЛКШ.Август.2015, параллель A'

Преподаватели: Пядеркин Михаил, Мингалиев Наиль, Лопатин Андрей, Курпилянский Евгений, Черникова Ольга

Автор: Ольга Черникова

Собрано: 17 августа 2015 г. 11:58

Оглавление

Глава I

Строки

I.1. Префикс-функция

$\mathfrak{Def}$: Пусть дана строка s длины n.

1. Тогда подстрокой строки s называется строка $s[i \dots j]$, где $0 \leqslant i \leqslant j < n$.
2. Префиксом называются подстроки, где $i = 0$
3. Суффиксом — подстроки, где $j = n - 1$.

$\mathfrak{Def}$: Пусть дана строка s.

Префикс-функция $p[i]$ для строки s, длина наибольшего префикса строки $s[0 \dots i)$(не совпадающий со всей строкой), который одновременно является её суффиксом.

Более формально $p[i] = \max n < i \colon s[0 \dots n) = s[i - n \dots i)$.

```
vector <int> p(n);
p[0] = 0;
for (int i = 1; i < n; ++i) {
    p[i] = p[i - 1];
    while (p[i] != 0 && s[p[i]] != s[i]) {
        p[i] = p[p[i] - 1];
    }
    if (s[p[i]] == s[i]) ++p[i];
}
```

I.1.1. Поиск подстроки в строке. Алгоритм Кнута-Морриса-Пратта.

Задача: Даны две строки s и t. s — образец, а t — текст. Нужно найти все вхождения образца в текст.

▶ Будем решать с помощью префикс-функции.

Создадим новую строку, являющейся конкатенацией трех строк `str = s + symb + t`, где `symb` — строка состоящая из одного символа, не входящего ни в s, ни в t (стоп-символ). На полученной строке посчитаем префикс-функцию. В местах где префикс функция равна длине строки s, там заканчивается очередное вхождение образца в текст. ◀

I.2. z-функция

$\mathfrak{Def}$: z-функция от строки s — это массив z, такое что $z[i]$ равно наидлиннейшему префиксу подстроки, начинающейся с позиции i в строке s, который одновременно является и префиксом всей строки s.

$$z[i] = LCP[0, i]$$

где LCP — это наибольший общий префикс (longest common prefix).

```
int z[maxn];
int l = 0, r = 0; //[l, r)

for (int i = 1; i < n; ++i) {
    if (z[i - l] < r - i) {
        z[i] = z[i - l];
    } else {
        z[i] = max(r - i, 0);
        l = i;
        r = i + z[i];
        while (s[z[i]] == s[i + z[i]]) {
            ++z[i];
            ++r;
        }
    }
}

z[0] = n;
```

I.3. Хеши

Хеши строчки$(s_0 \cdot p^{n-1} + \cdots + s_{n-1} \cdot p^0)$:

```
h[0] = 0;
for (int i = 1; i <= n; ++i) {
    h[i] = s[i - 1] + h[i - 1] * pr;
}
```

Хеш подстроки:

```
long long int Hash(int l, int r) {
    return h[r] - p[r - l + 1] * h[l - 1];
}
```

1. $\forall < p, M > \exists Hash(s_1) = Hash(s_2)$ — анти-хеш тест.
2. $< p = rand, M = rand >$ — неизвестно как построить анти-хеш тест.
3. $< p = rand, M >$ — строка Туе-Морса является анти-хеш тестом.

Правильно брать два рандомных p и фиксированное простое M.
Сравнение строк = бинпоиск по ответу и сравнение префиксов.
Парадокс дней рождений (если $k \geqslant \sqrt{2M}$, то вероятность коллизии больше $\frac{1}{2}$)

▶

$$1 \cdot \frac{M-1}{M} \cdot \ldots \cdot \frac{M-k}{M} = \left(1 - \frac{1}{M}\right) \cdot \left(1 - \frac{2}{M}\right) \cdots \left(1 - \frac{k}{M}\right) =$$

$$e^{\ln(1-\frac{1}{M}) \cdots (1-\frac{k}{M})} = e^{\sum_1^k ln(1-\frac{i}{M})} \approx e^{\sum_1^k (-\frac{i}{M})} = e^{-\frac{k(k+1)}{2M}} \approx \frac{1}{e}$$

◀

I.4. Бор

𝔇𝔢𝔣: Бор - структура для хранения множества строк. Дерево, на каждом ребре которого написана буква. Каждой вершине дерева соответствует строчка - последовательность символов на пути до нее от корня. Вершины, соответствующие строкам из множества помечаются терминальными.

Добавление строки:

```
struct Node {
    static const int ALPHABET = 26;
    int go[ALPHABET];
    bool is_terminal;

    Node() : is_terminal(false) {
        memset(go, -1, sizeof(go));
    }
};

Node nodes[MAX_SIZE];
int size = 1;

void add(const string &s) {
    int v = 0;
    for (int i = 0; i < s.size(); ++i) {
        int &ref = nodes[v].go[s[i] - 'a'];
        if (ref == -1)
            ref = size++;
        v = ref;
    }
    nodes[v].is_terminal = true;
}
```

Аналогично выглядит проверка строки на принадлежность множеству - смотрим пришили бы мы в терминальную вершину, если бы добавляли эту строку.

Хранение бора:

Есть несколько вариантов хранения ребер (V - число вершин, Σ - размер алфавита, LEN - длина обрабатываемой строки):

	Mem	Time	Примечания
`Array`	$\Sigma \cdot V$	LEN	
`map<int, int>`	V	$LEN \log \Sigma$	
`unordered_map<int, int>`	$V \cdot O(1)$	$LEN \cdot O(1)$	константы могут быть заметными

Используем бор для хранения множества строк.

I.4.1. Динамика на боре:

Задача: есть номера городов, одни номера городов являются префиксом других.

Номер относится к номеру с длиннейшим номером города. Посчитать для каждого города количество номеров.

▶ Добавляем все вершины в бор, считаем количество номеров у вершины начиная снизу и у каждой вершины храним количество занятых номеров в поддереве. (Если вершина терминальная, то она занимает все данной длины). Количество номеров у города = max - все занятые номера детей. ◀

I.4.2. Сортировка строк

Строки можно сортировать стандартной сортировкой и строки будут отсортированы за время работы $O(L \log L)$, где L суммарная длина строк.

I.5. Ахо-Корасик

$\mathfrak{Def}$: Суффиксная ссылка — максимальный суффикс этой строки, который встречается в боре от корня и не совпадает с данной строкой.

Если в бор добавлена одна строка, то это префикс-функция.

Хотим научиться находить слово из словаря в тексте (образец это подстрока, а не слово целиком, иначе умеем решать хешами).

Решение:
$O(n^2)$

1. добавляем в бор слова
2. p — предок

   c — символ на ребре к предку.

   go[v][c] — переход по символу c из вершины v, (-1, если нет перехода).

```
k = suflink(p)

while (go[k][c] == -1)
    k = suflink(k);
```

   Работает корректно, аналогично префикс-функции.

   Нужно найти как можно больший суффикс без одного символа и перейти по символу a.

Оптимизация: (Ахо-Корасик)

1. Храним `next[v][c]` = куда мы попадем, если будем прыгать несколько раз (может быть ноль) по суффиксным ссылкам, а потом один раз по данному символу c.

   Чтобы узнать `next[v][c]`:

   (a) если мы можем перейти по символу c, то узнали.

   (b) если не можем, то идем по суффиксной ссылке и узнаем next от него.

2. Идем по суффиксной ссылке предка и узнаем, куда мы попадем по символу c.

Если по суффиксным ссылкам можем добраться до терминальной, то мы тоже терминальные.

```
next[M][26];
suffix[M];
terminal[M];

start = 0;

for (c = 0; c < 26; ++c) {
    if (next[start][c] == -1) {
        next[start][c] = start;
```

```
    } else {
        suffix[next[start][c]] = start;
        que.push_back(next[start][c]);
    }
}

while (que.size() > 0) {
    x = que.front(), que.pop_front();
    for (c = 0; c < 26; ++c) {
        if (next[x][c] == -1) {
            next[x][c] = next[suffix[x]][c];
        } else {
            suffix[next[x][c]] = next[suffix[x]][c];
            que.push_back(next[x][c]);
        }
    }
}
```

I.6. Суффиксный массив

I.6.1. Поразрядная сортировка

1. Сортируем подсчетом по последнему символу.
2. Рассмотрим предпоследний символ и отсортируем подсчетом по нему, но устойчивой сортировкой.
   (a) считаем размер корзин.
   (b) считаем суммы на префиксах этих корзин.
   (c) в данном массиве записано место, на котором должен стоять данный элемент. После постановки элемента увеличиваем размер ячейки.

I.6.2. Алгоритм Карпа-Миллера-Розенберга

1. Вместо суффиксов будем сортировать циклические сдвиги. Дописываем в конец строки решетку(очень легкий символ).
2. Умеем сортировать строки длины один (подсчетом).
3. Хотим перейти от L к $2L$.
   Нужно отсортировать пары <c[i], c[i + L]>
   c — номер класса эквивалентности
   suf — место в отсортированном массиве строчек данной длины.
   `while (L < N):`
   (a) `suf[i] = (suf[i] - L)%N` (отсортировали по второму символу в цифровой сортировке)
   (b) Считаем сколько какого класса эквивалентности.
   (c) Вычисляем позицию для записи(суммы на преффиксах)
   (d) Записывем новый суффиксный массив:

```
        for (int i = 0; i < n; ++i) {
            n_s[cnt[c[suf[i]]]++] = suf[i];
        }
```

(e) Пересчитываем классы

```
    for (int i = 0; i < n; ++i) {
        if (i == 0 || (c[n_s[i]], c[n_s[i] + L]) !=
                      (c[n_s[i - 1]], c[n_s[i - 1] + L])) {
            n_c[n_s[i]] = n_c[n_s[i - 1]] + 1;
        }
    }
```

(f) Копируем, `c = n_c`, `suf = n_suf`.

I.7. LCP, алгоритм Аримуры-Арикавы-Касаи-Ли-Парка

LCP = largest common prefix
LCP(i, j) = min lcp[i ⋯ j]
За линию посчитаем LCP соседних суффиксов.

1. рассмотрим самый длинный суффикс. Считаем его lcp.

   Пусть lcp = x, тогда у следующего суффикса по длине lcp хотя бы x - 1

   Факт, что без одного отрезанного символа суффиксы будут располагаться в том же порядке, но, может быть, не подряд. (Доказывается как-то просто, надо порисовать).

```
x = 0;
for (int i = 0; i < n; ++i) {
    while (s[i + x] == s[suf[where[i] + 1] + x]) ++x;
    lcp[where[i]] = x;
    x = max(0, x - 1);
}
```

Глава II

Корневая эвристика

II.1. sqrt-декомпозиция

Рассмотрим несколько задач:

1. Дан массив чисел, требуется выполнять запросы:
   (a) добавить x на [L, R)
   (b) найти сумму на [L, R)

   ▶

   Разобьём массив на блоки размера k.

   Предподсчитаем сумму в блоках.

   Теперь, что бы ответить на запросы второго типа, нужно посчитать сумму на всех блоках, а потом отдельно посчитать сумму на кусочках. Время работы $O(k + \frac{N}{k})$

   Что бы добавить на отрезке, заведем вспомогательный массив add, прибавка на блоках. Время работы $O(k + \frac{N}{k})$

   k выберем равным $\sqrt{N}$.

   В задачах обычно уже даны ограничения, и k можно брать константой, с константой можно экспериментировать.

```
1     for(i = L; i < R; ) {
2         if (i % k == 0 && i + k <= R) {
3             res += sum[i/k];
4             i += k;
5         } else {
6             res += a[i] + add[i/k];
7             ++i;
8         }
9     }
```

   ◀

2. Дан массив чисел, требуется выполнять запросы:
   (a) прибавить x на [L, R)
   (b) есть ли элемент, равный x на [L, R)

▶

(a) Разобьём на блоки длины k, в каждом блоке будем хранить мультисет

```
multiset<int> S;
s.erase(x);//удаляет все x
s.erase(s.find(x)); //удалить один x
```

При ответе на запрос делаем find в нужных блоках и в остальных пробегаемся втупую. Когда делаем добавку, нужно искать $x - add_i$

(b) Другое решение — хранить отсортированные блоки массива и там искать бинпоиском. На двух блоках, где что-то испортили втупую отсортируем куски. $O(\frac{N}{k} + k\log(k))$
Можно отсортировать куски за линию. Разделить кусок на две части — префикс и суффикс, к одной части прибавили, а дальше merge. Теперь прибавление работает за $O(\frac{N}{k}+k)$
Ответ на запрос работает за $O(k + \frac{N}{k}\log k)$
$k = \sqrt{N\log N}$

$$\sqrt{N\log N} + \frac{N\log\sqrt{N\log(N)}}{\sqrt{N\log N}} = \sqrt{N}\frac{\frac{1}{2}(\log N + \log\log N)}{\sqrt{\log N}} = \sqrt{N\log N}$$

◀

3. Массив, заполнен неотрицательными числами

(a) вычитаем x на [L, R)

(b) $\forall$ элемента найти, на какой операции он стал не больше 0.

▶

(a) Храним отсортированную версию блоков и бинпоиском ищем первый 0 и храним, где у нас начинались неотрицательные числа.
$O(\sqrt{N\log N})$
Можно бинпоиск заменить на указатели.
$O(\sqrt{N})$

(b) Другое решение. У нас есть Q запросов, разобьём запросы на блоки по k.
После k запросов $A \to A'$, можем найти элементы, которые были положительные и стали отрицательные. Для этих элементов локально в блоке ищем момент, когда этот элемент стал отрицательный
Для построения массива A' у нас будет вспомогательный массив pref, когда отрезок появляется делаем -x, когда кончается +x.
swap(A, A')
Время работы $O(\frac{Q}{k}(N+k) + N\cdot k)$

$$k = \sqrt{Q}, O(N\sqrt{Q} + Q)$$

◀

4. Неориентированный граф (V, E). Каждая вершина имеет цвет от $1 \cdots k$.

Запросы:

(a) Перекрасить вершину v в цвет c.

(b) Найти количества различных цветов среди соседей.

▶

Можно в тупую хранить цвет, и пробегать по всем соседям и считать количество разных вершин. Но это слишком наивное решение.

Заметим, что если окрестность у соседей маленькая, то можно решить задачу в тупую. Если количество соседей у вершины больше k, то вершину назовем тяжелой.

Всего количество тяжелых вершин $\leqslant \frac{2m}{k}$.

Заранее сохраним ответ для тяжелых вершин. Храним массив: количество вершин данного цвета и переменную для вершины: количество разных цветов.

Ответ на запрос за $O(k)$, запрос на изменение за $O(\frac{2m}{k})$. ◀

Глава III

Hash-таблицы

1. Закрытая адресация.

   $s \to h(s) \sim 64$ бита.

   $M = 2^{20}$ создадим массив списков размера M.

   add(S):

   (a) `h(s) % M = x`

   (b) Берем список x и добавляем в него строку s.

   (c) Если хотим найти строку s, то ищем ее в соответствующем списке.

   $M = 2^{20}$ для более быстрого взятия по модулю.

   M должно быть раза в два-три больше N. Можно перестраивать хештаблицу по факту, если какой-то список стал очень большой, то стоит сделать хештаблицу в два раза больше.

2. Открытая адресация.

   (a) Есть просто массив объектов. Посчитали hash по модулю M, пусть он равен x. Пытаемся положить объект в ячейку x, если свободно, то кладем. Если в ячейке занято, займем ближайшую справа ячейку.

   (b) Для поиска идем вправо, пока все ячейки заняты и пока не встретили нашу строку идем в следующую ячейку.

   (c) Просто так удалить нельзя.

   (d) Можно пометить ячейку, как плохую. И если плохих стало очень много, перестроим таблицу.

   (e) Для следующий ячейки смотрим важно ли, что ячейка была занята, переносим элемент и рекурсивно удаляем.

Глава IV

Битовое сжатие

Задача о рюкзаке:
W — максимальная вместимость.
$w_1, \cdots, w_k$ — веса предметов.
Хотим найти максимальный вес меньше W, который можно набрать с помощью этих предметов.
▶
`dp[i][j] = dp[i - 1][j] | dp[i - 1][j - w[i]]`
Время работы $O(kW)$
`dp[i] = dp[i - 1] | (dp[i - 1] >> w[i])`
Можно руками превратить в массив int и так сделать or.
`bitset <N> b;`
`N` — размер битсета
`b[i]` — обратиться к элементу
`b << k` сдвинуть битсет на k.
Рюкзак:

```
bitset<S + 1> f;
f[0] = 1;
forn(i, n)
    f |= f << a[i]; // O(S/word)
```

◀

Глава V

Least Common Ancestor

V.1. Тривиальный алгоритм

Времена входов и выходов. Необходимо проверить, является ли одна вершина предком другой. Посчитаем времена входов и выходов. Заметим, что если вершина v, является предком u, то мы зашли раньше в v и вышли из v позже.

Тривиальный алгоритм обхода: поднимаем вершинку вверх, пока она не станет предком другой. Или поднимаем вершинки до одного уровня, а потом прыгаем, пока не станем одной вершинкой.

V.2. Двоичные подъемы, вычисление функции на путях

Предподсчитаем массив `p[v][k]` = куда мы попадем, если прыгнем вверх на 2^k.
`p[v][0]` = просто предок.

```
for (int i = 1; i < maxk; ++i) {
    for (int v = 0; v < n; ++v) {
        p[v][i] = p[p[v][i - 1]][i - 1];
    }
}
```

Способы подсчета LCA

1.
```
int lca(int u, int v) {
    if (h[u] > h[v]) swap(u, v);
    for (int k = maxk - 1; k >= 0; --k) {
        if (h[p[v][k]] >= u) {
            v = p[v][k];
        }
    }
    for (int k = maxk - 1; k >= 0; --k) {
        if (p[v][k] != p[u][k]) {
            v = p[v][k];
            u = p[u][k];
        }
    }
    if (v != u) {
        return p[v][0];
```

```
        } else {
            return v;
        }
    }

    int lca(int v, int u) {
        for (int k = maxk - 1; k >= 0; --k) {
            if (!is_parent(p[v][k], u)) {
                v = p[v][k];
            }
        }
        if (is_parent(v, u)) return v;
        else return p[v][0];
    }
```

Если функция обратима, то можно посчитать функцию от корня до вершины.
`(sum(u, v) = sum(u) + sum(v) - 2sum(lca(u, v)))`
Если функция ассоциативна, разбить путь на log кусков как в двоичных подъемах и значения предподсчитываем. В первом способе можем считать функцию по ходу, во втором нужно сначала вызвать lca(u, v), а потом lca(v, u).

V.3. CHM

Система непересекающихся множеств.
Нужно уметь объединять множества и узнавать лежат ли вершины в одном множестве.

1. Тупое решение: массив цветов, перекрашиваем, когда сливаем.

   Заметим, что если будем перекрашивать то, что меньше, то сложность в итоге будет $O(N \log N)$, поскольку каждая вершина, когда перекрашивается, оказывается в множестве хотя бы в два раза большем. А отвечаем на запросы совсем за быстро.

2. Можем хранить лес, и подвешивать одно дерево к другому, когда сливаем. Дальше проверяем, что корень один и тот же.

Эвристики у последнего решения

1. Сжатие пути

   Ответ за $O(\log n)$.

   Рассмотрим три вида ребер: корневые, тяжелые (ребра, на которых висит более половины поддерева), легкие.

   Заметим, что на пути у нас встретится только одно корневое ребро. Легких не больше log на пути (каждый раз поддерево увеличивается хотя бы в два раза).

   Теперь разберемся с тяжелыми ребрами. Рассмотрим вершину, когда она стала не корнем. Ее размер больше не увеличится. Если мы пройдем по тяжелому ребру в вершину, то ее поддерево уменьшится хотя бы в два раза. Значит количество переходов по тяжелым ребрам в эту вершину не более log.

2. Ранговая эвристика Глубина не больше $\log n$.

   Докажем по индукции.

   Одна вершина ранг = 0, количество вершин 1.

   Теперь пусть верно для k и размер хотя бы 2^k, тогда если у нас увеличился ранг, то два дерева равны по рангу и значит размер дерева ранга k + 1 хотя бы 2^{k+1}.

V.4. Алгоритм Ахо-Хопкрофта-Ульмана-Тарьяна

У вершин три цвета:

1. белые — еще не трогали
2. серые — сейчас обрабатываем (предки)
3. черные — закончили обработку.

Изначально каждая вершина в своем множестве.
При обработки вершины v:

1. u — пара v и u черная, тогда `lca(u, v)` = вершина с min высотой в множестве u.
2. Заканчиваем обработку вершины, красим в черный, присоединяем к множеству предка.

$O(\log^* n + m)$

V.5. Sparse Table

```
m[i][0] = a[i];
m[i][k] = max(m[i][k - 1], m[i + (1 << (k - 1))][k - 1])
```

Ответ на запрос `max(m[k][l], m[k][r - (1 << k)])`
$O(n \log n)$ — память предподсчета.
$O(1)$ — запрос.

V.6. LCA -> RMQ

Записываем в массив вершину, когда в dfs заходим в вершину и выходим из любого его ребенка.
Для ответа на запрос нужно найти два любых вхождения u и v и найти минимум по высоте на отрезке.

Глава VI

Паросочетания

VI.1. Теорема об удлиняющей цепочке

$\mathfrak{Def}$: Граф называется двудольным, если существует его правильная раскраска в два цвета, то есть вершины можно разбить на две доли так, чтобы ребра шли лишь между вершинами разных долей.

$\mathfrak{Def}$: Паросочетанием называется любое множество ребер, не имеющих общих концов.

$\mathfrak{Def}$: Максимальным паросочетанием называется любое максимальное по размеру (количнству ребер) паросочетание.

$\mathfrak{Def}$: Пусть фиксировано некоторое паросочетание. Вершина называется свободной, если нет ни одного ребра паросочетания, смежного с данной вершиой. В противном случае вершина называется занятой.

$\mathfrak{Def}$: Пусть фиксировано некоторое паросочетание. Удлиняющей или чередующейся цепочкой называется любой путь из свободной вершины в свободную, ребра в котором чередуются: ребро не из паросочетания, ребро из паросочетания и т.д.

Теорема VI.1.1. (об удлиняющей цепочке). Паросочетание максимально тогда и только тогда, когда в графе нет удлиняющей цепочки.

▶ $\Rightarrow$ Если в графе есть удлиняющая цепочка, то выбросив ребра этой цепочки из паросочетания и добавив ребра этой цепочки, которые до этого ему не принадлежали, мы увеличим размер паросочетания на 1.

$\Leftarrow$ Теперь докажем, что если паросочетание не максимально, то существует удлиняющая цепочка. Действительно, пусть M — текущее паросочетание, а M' — максимальное. Тогда рассмотрим симметрическую разность $M \sim M'$ двух паросочетаний как множеств ребер, а именно рассмотрим ребра, лежащие ровно в одном и паросочетаний.

Рассмотрим граф, содержащий только рассматриваемые ребра из симметрической разности. В этом графе степень каждой вершины не превосходит 2, так как каждой вершине может быть инцидентно максимум одно ребро каждого из паросочетаний. Все такие графы имеют очень простой вид: это набор цепей и циклов. При этом, так как $|M'| > |M|$, то найдется компонента связности, в которой ребер из M' больше. Эта компонента является цепью. Концы этой цепи являются свободными вершинами в M (иначе мы бы добавили в симметрическую разность еще одно ребро). Таким образом, данная цепь и есть искомая удлиняющая цепочка. ◀

VI.2. Алгоритм Куна

Алгоритм: найдем удлиняющую цепь, увеличим паросочетание...

Будем искать цепочки только из левой доли, потому что цепочка заканчивается в разных долях.

```
1 int curTime;
```

```
bool dfs(v) {
    if(used[v] == curTime) return false; //очевидно, что бессмысленно ходить в одну вершин
    used[v] = curTime;
    for u: (v, u) \in E {
        if (pair[u] == -1 || dfs(pair[u])) {
            pair[u] = v;
            return true;
        }
    }
    return false;
}

int main() {
    for (int i = 1; i <= n; ++i) {
        if (dfs(i)) {
            cnt++;
            ++curTime;
        }
    }
}
```

Такой алгоритм будет работать.

Если вершина насытилась, то она уже не перестанет быть насыщенной в дальнейшем. Пусть из i мы не нашли паросочетание, значит, мы уже не найдём паросочетание с этой вершиной, то есть, когда мы находим удлиняющую цепочку, то цепочка состоит из вершин, не достижимых из i.

Оптимизация от Сережи.

Можно обнулять used только когда мы нашли новую цепь.

Теперь, почему не нужно проверять, что i ненасыщенное. Потому что до этого мы должны были прийти из меньший вершины, но тогда эта вершина должна была быть насыщенна до этого.

Задача: Пусть вершины левой доли взвешены, нужно найти максимальное по весу паросочетание.

▶ Нужно отсортировать вершины в порядке убывания.

Пусть мы запустили Куна и для первых i паросочетание оптимально.

1. от $i + 1$ вершины Кун не нашел паросочетание. Пусть есть более хорошее паросочетание для $i + 1$, тогда $i + 1$ вершина участвует. Единственный вариант, когда мы могли что-то улучшить, это с цепочкой нечётной длины, иначе мы выкинули какую-то более жирную вершину.

2. Если добавили $i + 1$ вершину, то из старого паросочетания мы ничего не выкинули, значит все хорошо. Иначе бы паросочетание для первых i было бы не оптимально.

◀

VI.2.1. Вершинное покрытие и независимое множество

𝔇𝔢𝔣: Вершинное покрытие — множество вершин, которое покрывает каждое ребро хотя бы одной вершиной.

𝔇𝔢𝔣: Независимое множество — это множество несмежных вершин.

Нам интересно максимальное независимое множество и минимальное вершинное покрытие.

Теорема VI.2.1. Дополнение любого вершинного покрытия — независимое множество.

▶ Пусть дополнение зависимое, тогда есть вершины соединённые ребром, тогда это ребро никто не покрывает ◀

Теорема VI.2.2. П.усть есть какое-то максимальное паросочетание.

Размер вершинного покрытия $\geqslant$ максимального паросочетания.

▶ У всех паросочетаний мы должны взять хотя бы одну вершину, мы одной вершиной не можем покрыть два ребра из паросочетания.

M — максимальное паросочетание

Сориентируем ребра: Ребра из паросочетания справа налево, не из паросочетания слева направо.

Запустим dfs из ненасыщенных вершин левой доли.

То что обошли A^+ и B^+ и то что не посетили A^- и B^-

Заметим, что между A^+ и B^- вообще нет ребер. Иначе бы у нас была удлиняющая цепочка. A^+ и B^- независимое множество, а дополнение вершинное покрытие.

Посчитаем размер $|B^+| \cup A^- =$|МП| все вершины из A^- насыщенные, иначе бы мы с них начали. Любая вершинка из B^+ насыщенная, иначе бы у нас была удлиняющая цепочка. Значит $|B^+ \cup A^-| \leqslant$|МП|, значит это минимальное вершинное покрытие. ◀

Теорема VI.2.3. Теорема Кенинга. Минимальное вершинное покрытие = максимальному паросочетанию.

$\mathfrak{Def}$: Антицепь — Множество вершин в G, если любые 2 вершины не соединены.

G — ациклический транзитивно замкнутый ориентированный граф.

Дан G, хотим найти максимальную антицепь.

Теорема VI.2.4. Теорема Дилворта. Размер максимальной антицепи равен минимальному количеству вершино непересекающихся цепей, покрывающих все вершины графа.

▶

1. Давайте расклеим граф. У каждой вершины будет копия и проведем ребра из u в v', граф получился двудольный.

2. Найдем максимальное паросочетание в этом графе.

3. Найдем минимаьное вершинное покрытие.

4. (u, u') не лежат одновременно в вершинном покрытие

5. $A = V \quad (u|u \in min\text{ВП} \cup u' \in min\text{ВП})$, A — антицепь.

   От противного Пусть есть ребро u v соединённые ребром в антицепи, тогда это ребро никто не покрывает, противоречие.

6. A — максимальная антицепь. |A| $= N - K$ Выделим ребра из паросочетания в начальном графе. Начальный граф можно покрыть $N - K$ цепями. |МП| $= N - K$.

   Пойдем по ребрам паросочетания и получится цепь. Цепей будет ровно столько, сколько ненасыщенных вершин в правой доле.

7. не может быть никакая цепочка не может проходить через две вершинки антицепи. Значит что бы покрыть все вершины антицепи нужно хотя бы цепей столько сколько длина антицепи

8. Наша антицепь максимальна. Пусть есть антицепь |A'| $>$ |A| = n - k, тогда две вершинки попали в одну цепь, тогда это не атницепь.

◀

Глава VII

Структуры данных

Все люди делятся на две части: на тех, кто ходят пешком и тех, кто ездит на машине. Те, кто ездит на машине делятся на две части: те, кто ездят на переднем сиденье и на заднем. Те, кто ездит на переднем сиденье делятся на две части: те, кто пристегивается и те, кто не пристегивается. Те, кто не пристегивается: делится на две части, те, кто попадает в аварии и те, кто не попадает в аварии.

VII.1. Дерево отрезков

Отрезок от 1 до N продлим до 2^k.

Отрезки делятся каждый раз на две части. У i-ого элемента массива будут дети $2 \cdot i$ и $2 \cdot i + 1$.

Количество элементов в дереве отрезков $2 \cdot 2^k - 1$, первый элемент уровня это степень двойки.

__builtin_ctz — количество лидирующих нулей у числа.

В дереве отрезков можно хранить, например, сумму на отрезке.

s[i] = s[2i] + s[2i + 1]

То есть дерево отрезков, вы можете построить за $O(N)$.

На отрезке можно считать любые ассоциативные функции.

Структура работает быстрее, чем отдельные массивы, из-за кеша. Выгоднее использовать unsigned int для деления на 2, но в целом, это мелочи.

Напишем функцию обхода дерева.

```
walk(x, l, r) {
    if (l < r) {
        walk(2x, l, (l + r)/2);
        walk(2x + 1, (l + r)/2 + 1, r);
    }
}
```

Переделаем функцию в сумму.

```
//T[x] -> [l, r]
sum(x, l, r, a, b) { // сумма на пересечение отрезков l, r и a, b.
    if (r < a || b < l)  { // если  отрезки не пересекаются
        return 0;
    }
    if (a <= l && r <= b) { // если отрезок вложен.
        return T[x];
    }
    return sum(2x, l, (l + r)/2, a, b) +
```

```
    sum(2x + 1, (l + r)/2, r, a, b);
}
```

Почему дерево отрезков работает за log? Раньше же функция обходила совсем все дерево.

▶ Легко заметить, что на одном уровне мы посетим максимум 4 вершины. Дальше индукция по уровням. Это чуть дольше, чем дерево отрезков снизу, зато, у нас все функции будут писаться одинаково и в них будет сложнее ошибиться. ◀

Функция изменения в точке.

```
//T[x] -> [l, r]
change(x, l, r, a, b, v) { // изменение на пересечение отрезков l, r и a, b.
    if (r < a || b < l)  { // если  отрезки не пересекаются
        return;
    }
    if (a <= l && r <= b) { // если отрезок вложен.
        T[x] = v;
    }
    change(2x, l, (l + r)/2, a, b);
    change(2x + 1, (l + r)/2, r, a, b);
    relax(x, l, r) // T[x] = T[2x] + T[2x + 1];

}
```

Заметим проблемы в случае, если $a != b$.

1. Нужно присваивать другую сумму. $T[x] = v \cdot (a - b + 1)$
2. Нам нужно изменить сумму и на более мелких отрезках, то есть если изменение на большом отрезке, то мы будем работать долго.

   Лень двигатель прогресса, поэтому будем использовать ленивые вычисления и считать что-то только когда нам это пригодилось.

   Заведем еще одно поле структуры, сколько мы присвоили на этом отрезке.

```
struct Node{
    int sum, assigned;
    bool was_assigned;
};

T[x].sum = ...
T[x].assign = v;
T[x].was_ass = 1;
```

```
push(x, l, r) {
    if (T[x].was_ass) {
        T[x].was_ass = false;
        whole();
        whole();
    }
}

//T[x] -> [l, r]
change(x, l, r, a, b, v) { // изменение на пересечение отрезков l, r и a, b.
```

```
    if (r < a || b < l)  { // если  отрезки не пересекаются
        return;
    }
    if (a <= l && r <= b) { // если отрезок вложен.
        whole(x, l, r) //
    }

    push(x, l, r);
    change(2x, l, (l + r)/2, a, b);
    change(2x + 1, (l + r)/2, r, a, b);
    relax(x, l, r) // T[x] = T[2x] + T[2x + 1];
}
```

Дерево отрезков двоичное. Что будет, если сделать троичное дерево отрезков? Вы уже использовали корневое дерево отрезков, только там всего два уровня и вы их разбирали руками.

Можно вместо суммы на отрезке, считать, например, хеш на отрезке. Это позволяет считать хеш на подотрезке за log и теперь вы можете еще и менять символ.

Даны две позиции i и j и нужно посчитать LCP строк [i, n] и [j, n], и меняем какой-нибудь символ. Бинпоиск и хеши в дереве отрезков.

Поиск по номеру в дереве отрезков. В вершине есть функция

```
search(x, l, r, i) {//ищит i-ый элемент на отрезке.
    ...
    if (i < T[2x].sum) {
        search(2x, ...)
    } else {
        search(2x + 1, ..., i - T[2x].sum);
    }
}
```

Очень похоже на написание сочетаний. C(n, k) = 0, C(n - 1, k) либо 1, C(n - 1, k - 1);

Такая запись позволяет искать i-ый объект по номеру. По большому счету у нас написан бор.

Нужно искать объект самый больший меньше данного. Есть число от одного до 1000000 и найти самый большой элемент $\leqslant k$.

Поставим 0 если число взято и 1 если не взято. Возьмем сумму от 1 до k, а потом вызвали функцию search(x, l, r, sum(k));

Другое решение, храним самую правую единицы. У нас отрезок разбивается на log отрезков, найдем самый правую единицу в самом правом отрезке.

a, b, k самый левый меньше k на отрезке [a, b]. Будем хранить в вершинках еще и min. Разбиваем на левую и правую часть, если min нам подходит, идем влево, иначе вправо.

Можно решать задачу, представляя отрезок разбитый на кусочки.

```
search(x, l, r, a, b, k) {
    if (не попало) return +inf;
    if (лист) return l;
    if (полностью лежит) {
        push
        if (слева все нормально) return search(2x, ...);
        else return search(2x + 1, ...)
    }
    push
```

```
    return min(search(слево), (if не нашли ответ слева) search(справо));
}
```

Самый длинный черный подотрезок, перекрашиваем отрезок. Храним pref$|[l, t_1]|$, mid$|[t_1, t_2]|$, suff$|[t_1 \cdots r]|$

```
relax (x, l, r)
    if (T[2x].pref < T[2x].len) {
       T[x].pref = T[2x].pref
    } else {
       T[x].pref = T[2x].len + T[2x + 1].pref;
    }
    для суффикса аналогично.
    T[x].mid = max(T[2x].suf + T[2x  + 1].pref, T[2x].mid, T[2x + 1].mid);
```

VII.2. Декартово дерево

$\mathfrak{Def}$: Декартово дерево — частный случай дерева поиска. То есть все элементы меньше корня слева, все остальные справа.

Но тут у нас есть произвол с выбором корня, например, если корнем каждый раз выбирать первый, то получится бамбук.

Теперь у нас будут пары (x_i, y_i), и эти пары как-то будут нам говорить, кого поставить в корень.

Наложим на y ограничения двоичной кучи. Теперь каждому x будет сопоставлен y и корнем будем выбирать элемент с максимальным y.

Не сложно доказать по индукции, что декартово дерево единственное. Просто наличие y не решает проблему сбалансированности. Например, можно взять все x равные y и получится плохое дерево. Можно доказать, что какую бы мы не взяли зависимость y от x, можно придумать пример, когда глубин будет корень.

Но можно просто сопоставлять случайный y. Как будет генерироваться рандом?

srand(X) возьмем фиксированное, что бы при отладке не потерять баг, так же можно подставлять разные ядра рандома и тестировать самим с собой. RAND_MAX бывает достаточно не большой. Можно попробовать делать высоты размера побольше.

В среднем высота дерева лежит между 2 и 3 log.

Как построить декартово дерево?

По каким попала парам за O(N) не построить, поскольку иначе мы решим задачу о сортировке.

Но если пары уже отсортированы, то можно построить за O(N).

Пусть мы дошли до x_k и построили дерево для первых. Спускаемся до нужного места вниз, и переподвешиваем. Это долго, но если ходить не вниз, а вверх и решение за квадрат превращается в линию. Пусть, когда мы подвешиваем вершину, мы получаем доллар, а когда поднимаемся вверх, платим доллар, понятно, что мы не заплатим, больше чем получили, а вниз мы прошли за время O(N).

Декартовы деревья очень удобно рисовать в декартовой системе координат. Заметим, что каждая вершина соответствует какому-то отрезку x.

```
struct Node {
     Node *left, *right;
     int x, y;
};

Node* search(Node *T, int k) {
```

```
    if (T == NULL) return NULL;
    if (k == T->x) return T;
    if (k < T->x) return search(T->left, k);
    if (k > T->x) return search(T->right, k);
}
```

Рекурсивная функция выглядит немного жестко, обычно ее пишут немного по-другому.

```
Node* search(Node *T, int k) {
    while(T && T->x != k) {
        T = k < T-> x?T->left: T->right;
    }
    return T;
}
```

Главные операции в декартовом дереве — это split и merge. Это обратные друг к друго операции. Split это разделение дерева на две части, заметно, что должно произойти что-то не тривиальное.
split T, k $\rightarrow$ L, R
merge L, R(L $<$ R, все элементы из L строго меньше всех элементов из R) $\rightarrow$ T

```
typedef struct *Node tree;

void split(tree T, int k, tree &L, tree &R) {
    if (!T) {
        L = R = T;
        return;
    }
    if (k < T->x) {
        split(T->left, k, L, T->left);
        R = T;
    } else {
        split(T->right, k, T->right, R);
        L = T;
    }
}

tree merge(tree L, tree R) {
    if (!L) return R;
    if (!R) return L;
    if (L->y > R->y) {
        L->right = merge(L->right, R);
        return L;
    } else {
        R->left = merge(L, R->left);
        return R;
    }
}
```

Функции очень похожи с точностью до замены x на y.

А теперь зачем все это? Мы хотели добавлять элементы в множества, а теперь что-то режим и что-то склеиваем.

Хотим добавить вершинку. Можем разделить по x, теперь сольем два раза. Это метод чайника, но можно добавить элемент за один спуск по дереву, а не за три.

```
void add(tree& T, Node* N) {
    if (!T) {
        T = N;
        return;
    }
    if (N->y > T->y) {
       spilt(T, N->x, N->left, N->right);
       T = N;
    } else if (N->x < T->x) {
        add(T->left, N);
    } else {
        add(T->right, N);
    }
}
```

Удаление — это почти поиск, только нужно еще и удалить.

```
delete (tree& T, int k) {
    if (!T) {...}
    if (T->x == k) {
        T = merge(T->left, T->right);
    } else {
        if (k < T->x) {
            delete(T->left, k);
        } else {
            delete(T->right, k);
        }
    }
}
```

Теперь из содержательной информации начнем хранить количество элементов в поддереве(int c).

Этот параметр будет релаксироваться достаточно естественным образом.

T->c = T->left->c + T->right->c + 1;

Проблема, если у нас есть NULL. Можно сделать функцию, которая возвращает размер, а можно сделать специальную структуру, которая отвечает за NULL. Будем считать, что у нас есть специальная структура и не будет проверять на NULL.

Сейчас мы уничтожим x, будет декартово дерево по неявному ключу.

Теперь у нас есть массив от $[0 \cdots N - 1]$(как бы x).

У каждого поддерева будет своя нумерация. То есть у корня будет нумерация от [0, N - 1]. Теперь в левом поддереве элементы нумеруются $[0, c_l - 1]$, у корня номер c_l, у правого поддерева с точки зрения корня элементы $[c_l + 1, N - 1]$, а с точки зрение правого поддерева $[0 \cdots c_r - 1]$.

```
split (tree T, int k, tree &L, tree &R) {
    if (!T) {
        L = R = NULL;
        return;
    } else if (k < $c_l$) {  // в зависимости от того, что вы поставите < или <= вы включа
        split(T->left, k, L, T->left);
        R = T;
    } else {
```

```
        split(T->right, k - ($c_l$ + 1), T->right, R);
        L = T;
    }
}
```

Рассмотрим операцию перевернуть отрезок.

```
struct Node {
   Node *left, right
   int y, c, v;
   bool r;
};

reverse(tree T) {
    T->r = !T->r;
}

push(tree T) {
    if (T->r) {
        reverse(T->left);
        reverse(T->right);
        swap(T->left, T->right);
        T->r = false;
    }
}
```

С изменением на отрезке нет никакой разницы, то есть вытаскиваем отрезок и можем с ним что-то сделать, так же как и в дереве отрезков можем завести функцию relax и функцию push.

```
walk(tree T, int l, int r, a, b) {
    if (r < a || b < l) return;
    if (a <= l && r <= b) {
        whole(T, ...);
        return;
    }
    push(T);
    walk(T->left, l, T->x - 1, a, b);
    handle(T);
    walk(T->right, T->x + 1, r, a, b);
    relax(T);
}
```

Глава VIII

Персистентные структуры данных

Можем изменять и обращаться к любой версии объекта.

VIII.1. Персистентный стек

Нельзя изменять значения элементов, для всего нужно создавать новое. **Метод копирования путей(part coping)**

```
struct Node {
    int value;
    Node* next;
};

Node* push(Node* s, int x) {
    Node* res = new Node;
    res->next = s;
    return res;
}

pair <int, Node*> pop(Node * s) {
    return make_pair(s.value, s.next);
}
```

`Node[]` — значение элементов стека. `st[]` — указатель на последний элемент стека во времени.

VIII.2. Дерево отрезков

Каждый раз при изменение создаем новую вершину, а не копируем старую.

```
struct Node {
    Node *L, *R;
    int value;
};

Node* update(Node *v, int lv, int rv, int p, int x) {
    if (l + 1 == r) {
        return new Node(0, 0, x);
    }
```

```
    int m = (l + r)/2;
    if (p < m) {
        Node *nL = new Node;
        nL = update(v, lv, m, p, x);
        return new Node(nL, R, nL.val + R.val);
    }
}
```

Памяти $O(n + m \log n)$

Время: $O(m \log n)$

Задача: Онлайн отвечать на запрос количество точек внктри прямоугольника. Ответ на запрос $O(\log n)$

▶ По y дерево отрезков, по x сканлайн.

Событие: встертили точку, увеличели соответствующее значение в персистентном дереве отрезков.

Запрос прямоугольник: бинпоиском находим позицию, где начинается и заканчивается праямоугольник. sum(в правом дереве на нужном отрезке) - sum(в левом на отрезке). ◀

Задача: k-ая порядковая статистика на отрезке.

Идем в массиве сканлайном, если втретили элемент, изменяем счетчик в персистентном дереве отрезков.

Теперь при ответе на запрос, будем параллельно спускаться в двух деревьях/ $D[i] = D_r[i] - D_l[i]$ — количество элементов на отрезке [l, r].

Ответ на запрос $O(\log n)$.

VIII.3. Персистентное декартово дерево

Хотим сделать персистентный Treap. Казалось бы, можно просто брать и каждый раз не менять вершину, а создавать ее копию и работать с ней. Но тогда у нас в какой-то момент может взять и сломаться балансировка. А именно, у нас могут появляться одинаковые y у вершин. Более того, если мы будем делать запросы вида `r=merge(r, r)` кучу раз, то таких вершин будет реально много. Пусть у нас была такая реализация merge:

```
Node *merge(Node *a, Node *b){
    if (a == NULL) return b;
    if (b == NULL) return a;
    if (a->y >= b->y) {
        a->r = merge(a->r, b);
        return a;
    } else {
        b->l = merge(a, b->l);
        return b;
    }
}
```

То после повторения в цикле такой операции: `r=merge(r, r)`, мы получим бамбук, направленный вправо. Пусть мы повторили эту операцию m раз, тогда размер дерева стал 2^m, и его глубина тоже стала 2^m вместо желаемой m.

Решение этой проблемы достаточно простое. Просто не будем хранить y в вершине. Вспомним, зачем нам нужны были случайные y. Мы говорили, что если все y будут случайными, то полученное дерево тоже будет случайным, то есть каждая вершина будет корнем с одинаковой вероятностью. Ну давайте это возьмем и реализуем.

```
Node* merge(Node* a, Node* b) {
    if (a == 0) return b;
    else if (b == 0) return a;
    else if (random(a.size() + b.size()) < a.size) {
        return (Node(a.x, a.y, a.L, merge(a.R, b)));
    } else {
        return (Node(b.x, b.y, merge(a, b.L), b.R));
    }
}
```

Рассмотрим все вершины, входящие в a и b. Мы хотим, чтобы корень был случайной вершиной. Тогда вероятность того, что корень будет лежать в a это как раз и есть $a->size/(a->size+b->size)$, то есть данный код нам как раз дает RBST.

VIII.3.1. Копирование отрезков

Выделяем фиксированную память, когда память кончилась, выписываем дерево, очищаем память, строим дерево заново.

Глава IX

Геометрия

IX.1. Пересечение прямых

Прямую можно задать уравнением прямой: $a_1x + b_1y + c_1 = 0$
Тогда, что бы пересечь прямую нужно решить систему уравнений.

$$a_1x + b_1y + c_1 = 0$$
$$a_2x + b_2y + c_2 = 0$$

Решим систему:

1. Если $a_1b_2 - a_2b_1 = 0$, то прямые параллельны.
   (a) Если $a_1c_2 - c_1a_2 = 0 \Rightarrow$ прямые совпадают.
   (b) в ином случае не совпадают и пересечений нет.
2. Определим функцию

$$\det A = \det(\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}) == a_{11}a_{22} - a_{12}a_{21}$$

$$det2(a_{11}, a_{12}, a_{21}, a_{22})$$

$$y = -\frac{det2(A_1, A_2, C_1, C_2)}{det2(A_1, A_2, B_1, B_2)}$$

Многие говорят, что это клево и удобно, но Женя так делать не любит.
Зададим прямую с помощью точки(P) и вектора(S).
Тогда любая точка на прямой задается как $P + S \cdot t$
Тогда прямые пересекаются, если

$$p_1 + s_1t_1 = p_2 + s_2t_2$$

$\mathfrak{Def}$: Скалярное произведение: $\overline{a} \cdot \overline{b} = |\overline{a}||\overline{b}| \cos(\angle \overline{a}, \overline{b}) = a_xb_x + a_yb_y$
$\mathfrak{Def}$: Псевдовекторное произведение: $\overline{a} \times \overline{b} = |\overline{a}||\overline{b}| \sin(\angle \overline{a}, \overline{b}) = a_xb_y - a_yb_x = det2(a_x, a_y, b_x, b_y)$
Умеем проверять, являются ли вектора перпендикулярными или коллинеарными.
Важно помнить, что $\overline{a} \times \overline{b} \neq \overline{b} \times \overline{a}$, более того $\overline{a} \times \overline{b} = -\overline{b} \times \overline{a}$
Можно доказать, что $\overline{a} \times (\overline{b} + \overline{c}) = \overline{a} \times \overline{b} + \overline{a} \times \overline{b}$

$$p_1 \times s_2 + (s_1 \times s_2)t_1 = p_2 \times s_2 + s_2 \times s_2 t_2$$

$$t_1 = \frac{(p_2 - p_1) \times s_2}{s_1 \times s_2}$$

Осталось подставить t_1 и мы получаем точку пересечения.

Можно считать, что точка и вектор — это одно и тоже.

```
struct Point {
    int x, y;
    Point operator +(const Point &p) const
};

operator * // векторное
operator % // скалярное
```

Тогда, вы можете писать в коде так же, как на бумажке.

IX.2. Выпуклая оболочка

С помощью знака векторного произведения, можно понять угол больше 180 градусов или меньше.

Тогда попробуем проверить, что многоугольник выпуклый.

Многоугольник выпуклый, если для каждой стороны все вершины лежат по одну сторону от ребра. Перебираем все стороны, проверяем, что все вершины лежат с одной строны. Но это за квадрат.

Можно за линию, проверять, что векторное произведение соседних сторон в определенном порядке обхода имеют один знак.

$\mathfrak{Def}$: Выпуклая оболочка — выпуклый многоугольник, который содержит все заданные точки.

Понятно, что вершины многоугольника — это какие-то из заданных точек, иначе можем сжать.

Давайте найдем какую-нибудь точку, которая точно лежит в выпуклой оболочке. Например, самую нижнюю и из них самую левую.

Отсортируем точки по полярному углу относительно этой точки. Если мы возьмем точку с минимальным углом, эта точка тоже точно будет лежать в выпуклой оболочке...

Каждый раз в порядке угла будем добавлять точки в выпуклой оболочке, и поддерживать выпуклую оболочку для данных точке.

Последняя точка точно всегда будет входить в выпуклую оболочку, потому что она крайняя, и иногда нам нужно выкинуть сколько-то последних точек, что бы убрать невыпуклость.

Теперь, как, собственно, такое реализовывать

1. Перемещаем многоугольник в начало координат(из всех вершин вычитаем самую нижнюю, левую точку).

2. Сортируем по полярному углу. atan2(y, x) — возвращает угол от начала координат в полярной системе координат $(-\pi, \pi]$.

   Если использовать эту функуию, то получается долго и не точно, поэтому если и использовать, то предподсчитать угол надо заранее и сортировать по нему.

   Заметим, что нам достаточно знать кто правее, а точный угол знать не надо. Можно использовать для этого векторное произведение.

   Если углы равны, то нужно сравнивать по длине, сначала ближе к начальной точке, потом дальше.

```
bool cmp(Point A, Point B) {
    if (cross_product(A, B) > 0) return 1;
    else return A.len < B.len // заметим, что нам не нужно извлекать корень, для сравн
}
```

3. Теперь, для построения самой выпуклой оболочки, кладем все в стек.

   При обработки новой точки, если при добавление точки выпуклость не портиться, то все хорошо, если выпуклость портиться с последними точками, то выкидываем точку и проверяем еще раз.

4. Добавляем новую точку в стек.

IX.3. Принадлежит ли точка многоугольнику

.

Пускаем луч и считаем, сколько раз луч пересечет сторону. Если четное количество раз, то точка снаружи, иначе — внутри.

Если пересекли вершину, то непонятно, иногда нужно считать как пересечение, иногда нет.

Можно пускать в случайном направление, но это сложно и хрень какая-та.

Давай-те сделаем все границы полуинтервалами, но не в порядке обхода, а по y(нижняя включительно, верхняя исключительно).

Горизонтальные отрезки игнорируем совсем.

Теперь будем пускать горизонтальный отрезок. Как пересекать с таким лучом в целом, понятно.

рисунок

$x_0 < x_1 + (x_2 - x_1)\frac{y_0 - y_1}{y_2 - y_1}$.

То, что точка лежит на границе — это отдельный случай, надо еще раз пройтись и проверить, лежим ли мы на границе.

Или можно проверить, что начало луча правее, векторным произведением.

Еще один способ: посчитать сумму углов из точки, в зависимости от того получилось 0 или 2π.

Угол между двумя векторами `atan2(cp(a, b), dp(a, b));`

IX.4. Отвечать на запросы принадлежит ли точка выпуклому многоугольнику

Рассмотрим точку внутри.

Из какой-то вершины разобьём многоугольник на треугольники. Заметим, что если мы для точки найдем сектор, то дальше все не сложно.

Нужно посмотреть на векторное произведение со стороной.

Сам сектор можем найти бинпоиском, с векторным произведением внутри.

IX.5. Прямая и выпуклый многоугольник, пересекаются ли они.

нужен рисунок

Можно, понятно как, за линию.

Теперь рассмотрим кучу параллельных прямых данной прямой. Какие-то прямые пересекаются, какие-то нет. Если мы найдем параллельные касательные к многоугольнику, тогда Нужно проверить, что прямая лежит между касательными.

Так как многоугольник выпуклый, то вектора, задающие стороны, из точки, будут идти по-кругу.

Как найти касательную?

Рассмотрим напровляющие вектора сторон и перенесем их в начало координат. Посмотрим на направляющий вектор прямой. Он находится между первым и пятым вектором и касательная находится между первой и пятой стороной.

Теперь, просто так по векторному углу сортировать нельзя, поскольку вектора есть как и с верхней стороны, так и с нижней.

Нужно отдельно смотреть, вектора из одной полуплоскости или из разных.

Дальше, загоняем все в массив и делаем lower_bound.

Что бы найти непосредственно точки пересечения с многоуголником, нужно сделать бинпоиск по границе с помощью орентированного расстояния.

IX.6. Существуют ли пересекающиеся отрезки

Идем сканлайном, порядок непересекающихся отрезков не меняется. При появление нового отрезка, понятно в какое место его вставить.

Теперь давай-те подумаем, где у нас могут появиться кандидаты на пересечение. Логично, что при добавление нового отрезка нужно проверить, что он не пересечется с соседями.

Не соседи могут пересечься только, если между ними никого нет. Точнее пересекаться могут, но не сейчас.

То есть, при удаление отрезка нужно проверить, что новые соседи не пересекаются.

События сканлайна

1. открылся отрезок
2. закрылся отрезок

Сортируем события по x, если x одинаковый, то сначала отрезки будут открываться.

У нас будет `set<segment>`

В этом сете нужно с помощью lower_bound найти соседние отрезки, проверяем и добавляем в нужное место.

Каждый раз проверяем максимум две пары.

При удаление, проверяем соседей и удаляем

Компоратор в set

```
class cmp{
    bool operator() (segmetn s1, segment s2) {
        ...
    }
};

set <segment, cmp>
```

Работает за $O(n \log n)$

Что делать с вертикальными прямыми? Сделаем вертикальную почти вертикальной, отклоним на эпсилон. То есть у прямой будет x и y.

Глава X

Разделяй и властвуй и meet-in-the-middle

Есть большая задача, мы не знаем, что с ней делать.

Разделим на две части и решим для них рекурсивно. После этого нужно перебрать решения, которые пересекают границу разреза. Самое сложное "— найти ответ на границе разреза.

X.1. Найти две ближайшие точки.

▶ Можно в тупую за квадрат. Хотим быстрее.

1. Отсортируем точки по x и разделим их на две половины.

   Точки сортируем по x один раз. В дальнейшем точки уже будут отсортированы по x и мы сможем разделять на две отсортированных части линейным проходи.

   Запоминаем какой номер был в какой половине, в не координаты(проблема с повторяющимися точками).

2. Решим задачу рекурсивно. Получили ответ d_1 и d_2 и нужно учесть точки из разных половин.

   $d \leqslant min(d_1, d_2)$. Поэтому рассматривать пары, которые очень далеко не имеет смысла, они нам не помогут.

   Остались только точки которые не далеко от линии разреза, на расстояние не более $d = min(d_1, d_2)$. При этом может так получиться, что мы вообще не отсекли никакие точки.

   Посмотрим на интересную точку. Хотим для нее найти точку из другой половины на расстояние меньше d. Ограничим точку и по координате y. То есть все интересные пары для точки находятся в квадрате $2d \times 2d$.

   Если мы будем искать точку только в верхнем квадрате, то тоже все нормально, мы только обозначили, что первая точка в паре снизу.

   Теперь давай те думать, сколько точек может быть в одном квадрате $d \times d$ в одной из половин. Не более 4, поскольку между любой парой точек в квадрате расстояние не меньше d.

3. Когда мы находимся на границе, оставляем только точки, которые находятся на расстояние не больше d от границе по x.

4. Сортируем эти избранные точки по y. Добавляет лишний log, это грустно. Давай те передавать массив не только отсортированный по x, но и по y. Тогда рекурсивно создавать эти массивы мы можем за линию. Проходимся и смотрим, в какую половину нужно отправить точку.

5. Для каждой точки кандидаты будут только следующие 8 точек. Переберем их и посчитаем расстояние.

```
        for (int i = 0; i < k; ++i) {
            j = i + 1;
            while (y[j] <= y[i] + d) {
                проверяем i и j
                ++j;
            }
        }
```

Оценим время. При сливе работаем за (n), на следующем слое работали за $2 \cdot \frac{n}{2} \cdots$. Слоев $\log n$, время $O(n \log n)$

◀

X.2. Рюкзак

Есть веса $w_1 \cdots w_n$ и стоимости $c_1 \cdots c_n$, не можем набрать вес больше W, хотим максимизировать стоимость.

$w_i \leqslant 10^{16}$, $n \leqslant 32$.

▶ Умеем решать за:

1. $O(2^n)$
2. $O(Wn)$

Давайте разделим все предметы на две части, теперь в каждой части по 16 предметов. В половинах можем все перебрать, и задача для половин решена.

Теперь надо слить решения.

Если мы зафиксировали подмножества из правой части, то мы знаем вес, который мы уже взяли W', тогда из другой части интересны только подмножества с весом не больше, чем $W - W'$.

Отсортируем по W подмножества, а дальше нужно искать максимум на префиксе. Границу в массиве можем найти бинпоиском ну или с помощью двух указателей.

Считаем сложность:

$$2^{\frac{n}{2}} + 2^{\frac{n}{2}} + 2^{\frac{n}{2}} \frac{n}{2} + 2^{\frac{n}{2}} (\frac{n}{2} + 1) = O(n 2^{\frac{n}{2}})$$

◀

X.3. Dynamic Connectivity Offline

Неориентированный граф $G = (V, E)$, запросы:

1. добавить ребро
2. удалить ребро
3. находятся ли вершины в одной компоненте.

▶

1. У нас есть куча запросов, давайте, мы их разделим на две части. Если бы не было удалений, то это просто СНМ, если только добавление, то СНМ с конца, а так грустно.

   Давай те у каждого запроса будет пара, если добавили ребро, пара “— когда удалим.

   В начале добавим запросы добавить ребра которые и так были и в конце добавим запросы удалить все ребра, которые есть. Считаем, что кратных ребер нет.

   Сопоставим каждому запросу пару, как-нибудь это сделаем.

2. Разделили на две части запросы и что мы видим. Есть ребра, которые всегда есть, которых всегда нет, а есть какие-то плохие, которые иногда есть, иногда их нет.

   Если один запрос, то можем на него ответить. Если запросов много, делим на две части

3. Когда разделили на две части, нужно передать СНМ с правильными ребрами(которые не меняются на этом отрезке).

   Нам нужно в СНМ добавить ребра, у которого запрос добавления начинается до первой половины, и заканчиваются во второй половине. Перебираем все запросы удаления из правой части.

   $dsu \Rightarrow dsuL$.

   Получили для L, можем запускаться рекурсивно для левой половины.

4. Теперь нужно как-то получить $dsuR$. Мы испортили снм, копировать его полностью не можем, поскольку долго.

   Не умеем $dsuL \Rightarrow dsuR$, нам бы как-то вернуться к dsu.

   Если мы запомнили, что и где мы присваивали, то можно присвоить все обратно за столько же операций.

   Тогда мы смогли получить из $dsuL \Rightarrow dsu$ и можем перейти к $dsuR$, победа.

5. Теперь более подробно про откатывание изменений.

   Сохранить u и par[u], rank[v]. par[u] = v;

   Все пихаем в стек и в обратом порядке, откатиываем.

   Сжатие путей нам не особо помогает, мы старались, а потом откатились. Достаточно использовать только ранговую эвристику.

   Ранговая эвристика работает за $O(logn)$, а ранговая эвристика + сжатие путей тоже $O(logn)$.

◀

X.4. Максимальный тандемный повтор

$\mathfrak{Def}$: Строка S. T строка. T тандемный повтор S — если в S встречается подстрока TT.

▶ Если будем много букв a, тогда тандемных повторов будет примерно $\frac{n^2}{4}$.

Разделим строку на две части, запустимся рекурсивно от двух частей.

Осталось рассмотреть тандемные повторы, которые пересекаются с серединкой.

Длина TT = $2k$. Символу i будет равен символ i - k, но k мы пока не знаем. Пусть $k = k_1 + k_2$, с левой стороны находятся $k_1 + k_2 + k_1$ символ, во второй части k_1.

Когда мы зафиксировали k, то

$$k_1 \leqslant lcs(s[0 : i-k-1], s[0 : i-1]) = X_1$$

$$k_2 \leqslant lcp(s[i-k :], s[i :]) = X_2$$

Если мы найдем X_1иX_2, то мы справимся. X_1 и X_2 можем найти с помощью z-функции сток s[i:n - 1]#s[0:i] и s[i - 1: 0].

Время работы $O(n \log n)$ ◀

X.5. Алгоритм Карацубы

$A \cdot B$ хотим перемножить, большие числа.
$base = 10^4$, храним числа в массиве интов.
$c[i+j] += a[i] \cdot b[j]$
Время: $O(n^2)$.
Давай те при перемножении, разделим числа на две части.

$$A = A_1 base^k + A_0$$

$$B = B_1 base^k + B_0$$

$$AB = A_1 B_1 base^{2k} + (A_0 B_1 + B_0 A_1) Base^k + A_0 B_0$$

$$T(n) = 4T(\frac{n}{2}) + O(n)$$
$$T(n) = O(n^2)$$
$$T(k) = O(k^2)$$
$$T(2k) = 4T(k) + O(2k) = 4O(k^2) = O((2k)^2)$$

Попробуем выразить $A_0 B_1 + B_0 A_1$

$$(A_0 + A_1)(B_0 + B_1) = A_0 B_0 + A_1 B_1 + (A_0 B_1 + A_1 B_0)$$

$$T(n) = 3T(\frac{n}{2}) + O(n)$$
$$T(n) = O(n^{\log_2 3})$$

Глава XI

Динамика

XI.1. Семинар по ДП

Оценка за задачу равна (количеству человек, не сдавших эту задачу, плюс 1), умноженному на понижающий коэффициент подпункта, если он указан.

1. Наибольшая общая возрастающая подпоследовательность двух последовательностей
   (a) $O(n^4)$
   (b) $O(n^3)$
   (c) $O(n^2)$
2. Последовательность длины n называется выпуклой, если $\forall i\colon 1 < i < n\colon a_i < \frac{(a_{i-1}+a_{i+1})}{2}$.
   Найдите наибольшую выпуклую подпоследовательность данной последовательности за $O(n^3)$.
3. Разорванным числом называется последовательность из ровно n цифр от 1 до 9 такая, что любые две соседние цифры отличаются не менее чем на 3. Упорядочим все разорванные числа по возрастанию. Найдите по заданному разорванному числу его номер в этом списке. $O(9 \cdot n)$
4. Найдите минимальное k-ичное число
   (a) с заданной суммой S последовательных попарных произведений цифр за $O(S \cdot 10^2)$.
   (b) с заданной суммой S попарных произведений цифр как можно быстрее
5. Дана произвольная скобочная последовательность из круглых `( )`, квадратных `[ ]`, фигурных `{ }` и угловых `< >` скобок. Требуется удалить из неё минимальное количество скобок, чтобы она стала правильной (и скобки соответствовали скобкам того же типа). $O(n^3)$
6. Дано дерево. Выведите за $O(n)$:
   (a) для каждого из рёбер — количество простых путей, проходящих через него
   (b) для каждого из рёбер — сумму длин простых путей, проходящих через него
   (c) для каждого из рёбер — сумму квадратов длин простых путей, проходящих через него
7. В языке «Kava» есть n программных конструкций, длина каждой из которых не превосходит m. Валидная программа на языке «Kava» — это произвольная последовательность конструкций, каждую конструкцию можно использовать неограниченное число раз. Сколько различных программ длины L можно написать на языке «Kava»? $O(min(n, m) \cdot L)$
8. Проверить два корневых дерева на изоморфизм

9. Задача о рюкзаке. Есть n слитков золота, каждый имеет вес w_i и рюкзак, который выдерживает вес W.
   (a) Требуется положить в рюкзак как можно больше золота:
      i. (×0.2) Найти только вес за $O(W \cdot n)$, память $O(W \cdot n)$
      ii. (×0.2) Найти только вес за $O(W \cdot n)$, память $O(W)$
      iii. (×0.2) Найти сам способ за $O(W \cdot n)$, память $O(W \cdot n)$
      iv. (×0.5) Найти сам способ за $O(W \cdot n)$, память $O(W)$
   (b) (×0.2) А теперь всё то же самое, но каждый слиток можно брать бесконечное число раз.
   (c) Теперь каждого типа слитка k_i штук. Сделайте всё то же самое за время:
      i. (×0.2) $O(W \times \sum k_i)$
      ii. (×0.5) $O(W \times \sum log k_i)$
   (d) (×0.2) Теперь каждый из слитков весит не больше m, и запас снова бесконечен. Решите ту же задачу за время $O(n \cdot m \cdot log(m))$.
10. Найти какую-либо строчку минимальной длины, подходящую под два заданных шаблона (с вопросиками и звёздочками), или определить, что такой нет. $O(L_1 \cdot L_2)$ времени и памяти.
11. Дана последовательность из N чисел. Пусть $\sum$ — сумма чисел на подотрезке.
   (a) (×0.2) Подотрезок с максимальной суммой за $O(N)$.
   (b) (×0.2) Подотрезок длины от L до R с максимальной суммой за $O(N)$.
   (c) (×0.5) Три непересекающихся подотрезка такие, что $\sum_1 + \sum_2 + \sum_3$ максимально за $O(N)$.
   (d) (×0.5) Пункт a на окружности.
   (e) (×0.5) Пункт b на окружности.
   (f) (×1) Пункт c на окружности.
12. $O(N^3)$. Дан прямоугольник, состоящий из $N \times N$ целых чисел.
   (a) Выбрать подпрямоугольник $\widehat{\max}$ суммы
   (b) Выбрать два непересекающихся подпрямоугольника $\widehat{\max}$ суммы.
   (c) Выбрать три непересекающихся подпрямоугольника $\widehat{\max}$ суммы.
   (d) Выберите подпрямоугольник с $\widehat{\max}$ суммой в четырёх углах
   (e) Выберите подпрямоугольник с $\widehat{\max}$ суммой по периметру
13. Найдите число различных подпоследовательностей в последовательности длины n, составленной из чисел от 1 до n за $O(n)$.

XI.2. Разбор задач семинара

1. ▶
   (a) $O(n^2)$
   `dp[i][j][k]` = `len` максимальная длина последовательности если разобрали первые i в первой строчке, j во второй и остановились последовательность заканчивается на k.
   Давай-те сделаем `dp[i][j]` = `len`, при этом мы взяли i и j.

При переходе нужно перебрать элементы, которые были до нас.

$$dp[i][j] = dp[p][q], p, q\colon p < i, q < j, a[p] == b[q], a[p] < a[i]$$

Квадрат состояний и квадрат переходов.

(b) Динамика за $O(n^3)$. `dp[i][j]` = максимальная длина подпоследовательности, заканчивается в `a[i]`, и в префиксе `b[j]`.

`last[a[i]][j]` = для `a[i]` последний элемент раньше в b раньше j равный `a[i]`.

Теперь нужно найти максимальную общую возрастающую, которая заканчивается раньше i и раньше `last[a[i]][j]`.

$$dp[i][j] = max_p dp[p][last[a[i]][j] - 1]$$

(c) можем заметить, что сейчас у нас максимум очень простой. Какой-то на префиксе. Предподсчитаем максимум на префиксе.

$$pref_d p[i][j] = max_{0 \leqslant p \leqslant i} dp[p][j] = max(dp[i][j], pref_d p[i-1][j])$$

$$dp[i][j] = pref_d p[i-1][last[a[i]][j] - 1]$$

Динамика за $O(n^2)$

◀

2. ▶ Будем хранить два последних элемента последовательности и для них самую длинную выпуклую подпоследовательность. Можем перебрать элемент k, который был еще раньше $a_j < \frac{a_k + a_i}{2}$.

$$dp[i][j] = 2$$

$$dp[i][j] = max_k dp[j][k] + 1 (k\colon a_j < \frac{a_k + a_i}{2})$$

◀

3. ▶ Смотрим на число и смотрим, сколько чисел начинается на данную цифру.

Если мы знаем количество чисел, которые начинаются на x, то нужно прибавить к ответу Все числа, которые начинаются на цифру меньше x.

Далее мы свели задачу к числу с меньшим количеством чисел.

$dp[k][n]$ — количество чисел, которые начинаются на k и имеют длину n.

$$dp[k][n] = \sum_{1 \leqslant p \leqslant 9\colon |p-k| \geqslant 3} dp[p][n-1]$$

Нужно быстрее $O(ND^2)$, но нам не нужно каждый раз считать сумму, мы можем ее предподсчитать.

◀

4. ▶

(a) Построили число, нужно знать длин, сумму и последнюю цифру. Считаем количество чисел.

$$dp[S + dp][N + 1][D_2] = dp[S][N][D]$$

$O(S \cdot D^2 \cdot N)$

(b) Нам нужно хранить сумму всех цифр, последнее число хранить не надо.

$$(d_1 + d_2 + \cdots + d_n)^2 = \sum d_i^2 + 2 \sum d_i d_j$$

◀

5. ▶ Динамика по подотрезкам.

Рассмотрим первую скобку. Есть два варианта, что с ней делать: удалить или оставить.

Если удалили, то решаем задачу без нее и к ответу $+1$.

Если оставили, то нужно найти ей пару, дальше нужно независимо решить задачу внутри скобок и во внешнем мире.

Решаем задачу в подстроке от L до R

$$dp[L][R] = min(k\colon (s[L] \text{и} s[k] \text{— пара}) dp[L + 1][k - 1], dp[L + 1][R] + 1)$$

Отрезки нужно перебирать в порядке увеличения их длины. А можно считать динамику лениво. ◀

6. ▶ i из верхнего поддерева, j из нижнего, w текущее ребро.

k — размер поддерева.

$$\sum_{i,j} (p_i + w + q_j)^2 = \sum (p_i^2 + w^2 + q_j^2 + 2wp_i + 2wq_j + 2p_iq_j) =$$

$$= (n - k) \sum p_i^2 + k \sum q_j^2 + k(n - k)w^2 + 2w(n - k) \sum p_i + 2wk \sum q_j + 2 \sum p_iq_j$$

$$\sum p_iq_j = \sum p_i \sum q_j$$

◀

7.

8. ▶ Если у корней количество детей разное, то все сразу плохо, иначе, нужно как-то переставить детей.

Давай-те сопоставлять деревьям чиселки и у изоморфных деревьев будет одинаковое число.

Запускаем обход по дереву. `map <vector<int>, int>`

По множеству детей нужно понимать, видели мы такое раньше или нет. Если видели, то присваиваем такой же номер, иначе присваиваем какой-то уникальный.

В целом, можно считать полиномиальный хеш. ◀

9. ▶

(a) dp[x] — можем ли мы набрать вес x, если можем, то какой последний предмет мы положили.

$$\forall s\colon dp[s] \neq NULL\colon dp[s + w_i] = i$$

Слева направо идти не можем, поскольку будем использовать один предмет два раза. Можем идти с другой стороны, в порядке уменьшения весов и все будет ок.

```
    for i = 1 .. n
        for s = w .. 0
            if dp[s - w_i] && !dp[s] {
                dp[s] = 1;
                p[s] = i;
            }
```

`!dp[s]` — важное условие.

Иначе есть пример, когда мы в восстановление ответа будем использовать один слиток два раза.

◀

XI.3. Динамика по подмножествам

1. Задача о паросочетание с произвольном графе.

   ▶ Для построения паросочетания нам важно знать только множество вершин, которое мы используем, какие именно ребра нам не важно.

   `dp[mask]` — можно ли построить паросочетание, насыщающие только эти вершины. Заметим, что размер паросочетания можно легко восстановить из маски $\frac{cnt}{2}$.

   $$dp[mask] = OR_{(u,v)\in E; u,v\in mask} dp[(mask \quad \{u,v\})]$$
   $$u \in mask \Leftrightarrow ((mask >> u)\&1) == 1$$
   $$mask \quad u \Leftrightarrow (mask\&(\ (1 << u))$$
   $$dp[0] = true$$

   Можно считать динамику в порядке возрастания маски (for mask $= 1 \ldots 2^n - 1$).

   Получили массив dp, теперь нужно найти маску с максимальным количеством бит. Так же можно для маски хранить последнее ребро. То есть, массив предков.

   Время работы $O(2^n m)$ ◀

2. Хотим раскрасить вершинки в цвета, так, что бы вершины одинакового цвета не были соединены ребром.

   ▶ Выберем независимое подмножество вершин и красим в нулевой цвет. Еще раз выбираем независимое множество и красим$\cdots$. dp[mask] — минимальное количество цветов для покраски вершин из маски.

   $$dp[mask] = min_{mask2\subset mask} dp[mask \quad mask2] + 1$$
   $$mask2 \subset mask \Leftrightarrow (mask2\&mask) == mask2$$

```
    for (int mask = 1; mask < (1 << n); ++mask) {
        for (int mask2 = 1; mask2 <= mask; ++mask2) {
            if (mask2 подмножество mask && mask2 независимое) {
                обновляем dp[mask]
            }
        }
    }
```

Работает за $O(4^n \cdot n^2)$.

Разберемся с проверкой независимого множество. Можно заранее для каждой маски проверить, является ли множество независимым.

dp2[mask] — является ли маска независимым.

Давай те теперь соптимизируем перебор подмножеств.

Посчитаем количество пар, что одно из множеств является в подмножестве другого.

Есть элементы, которые лежат в маске, которые лежат в маске и подмаске и которые нигде не лежат. То есть для каждого элемента три варианта множества, куда мы можем его положить. Значит всего пар $O(3^n)$.

Теперь нужно научиться перебирать только те маски, которые нам нужны.

```
    for (int mask = 1; mask < (1 << n); ++mask) {
        for (int mask2 = mask; mask2 != 0; mask2 = (mask2 - 1)&mask) {
            ...
        }
    }
```

Есть произвольная mask. Посмотрим на какой-то mask2. При таком переходе mask2 всегда подмножество.

Теперь из mask2 вычитаем 1. Тогда самая правая 1 становится 0, все раньше остается таким же, все после стоят 1. То есть в значимых битах мы переходим к следующему числу, а не значимые всегда остаются 0.

То есть мы перебрали ровно все подмаски.

◀

XI.4. Динамическое программирование по профелю

Количество способов замостить доску доминошками.

▶ dp[i][mask] — количество замощенных первых i столбцов + в i столбце торчат доминошки в позициях mask.

dp[i + 1][mask2] Понятно, где стоят горизонтальные доминошки, проверяем что все дырки для вертикальных доминошек четные. Если есть нечетная, то нельзя перейти и количество способов 0, иначе 1.

```
for (int i = 0; i < m; ++i) {
    for (int mask = 0; mask < (1 << n); ++mask) {
        dp[i][mask] ''--- уже лежит правильный ответ.
        for (int mask2 = 0; mask2 < (1 << n); ++mask2) {
            if (good(mask, mask2)) {// так же плохо, если что-то торчало в mask и там же т
                dp[i + 1][mask2] += dp[i][mask];
            }
        }
    }
}

bool good(mask, mask2) {
    if ((mask & mask2) != 0) return false;
    free_mask = (mask | mask2) ^ ((1 << n) - 1); //то, что нужно заполнить вертикальными д
```

```
    Проходимся циклом, находим единичку, если следующий тоже единичка выкидываем оба, ина
}
```

Ответ лежит в dp[n][0].

Работает за $O(m \cdot 4^n \cdot n)$.

Память $O(2^n m)$.

Можем предподсчитать для каждой маски, можно ли добить ее вертикальными доминошками. Теперь время $O(m \cdot 4^n)$.

REM: $n < m$, если не так, то надо swap.

$n \cdot m \leqslant 100$, это значит, что хотя бы одна меньше 10.

Еще можем перебирать только подмножество дополнений, теперь получилось $O(m \cdot 3^n)$. ◀

XI.5. ДП по изломанному профилю

Это не сложнее, чем обычный профиль.

Задача та же.

▶ Теперь мы будем добавлять по одной доминошке.

dp[i][j][mask] — полностью покрыли первые i строк, j первых тоже покрыли, и есть изломанная интересная маска торчащих доминошек.

Если ($j \in mask$) dp[i][j + 1][$mask \setminus \{j\}$]

Если (j не в маске) dp[i][j + 1][$mask \cup \{j\}$] и если(j + 1 не в маске) dp[i][j + 1][$mask \cup \{j+1\}$]

Если (j == n) dp[i + 1][0][mask]

```
dp[0][0][0] = 1
for (int i = 0; i < m; ++i) {
    for (int j = 0; j < n; ++j) {
        for (int mask = 0; mask < (1 << n); ++mask) {
            if (j $\in$ mask) {
                dp[][][] += dp[i][j][mask];
            } else {
                dp[][][] += dp[][][];
                if () {
                    dp[][][] += dp[][][];
                }
            }
        }
    }
    for (int mask = 0; mask < (1 << n); ++mask) {
        dp[i + 1][0][mask] +=
    }
}
```

Время работы $O(n \cdot m \cdot 2^n)$. Память $O(n \cdot m \cdot 2^n)$ ◀

XI.6. bfs

Есть много прямоугольников, какие-то можем двигать по вертикали, некоторые, по горизонтали. Нужно вытащить прямоугольник через дырку.

Можем считать, что каждое состояние — это вершина. Если можно перейти из одного состояния в другое, то есть ребро. Считаем хеш от позиции. У каждого прямоугольника можем записать координату и посчитать хеш от строки.

Запускаем на таком графе bfs, поскольку нужно найти кратчайший путь.

Непонятно, за сколько это работает, но это работает.