Стресс-тестирование

- Метод поиска багов, заключающийся в генерации случайных тестов и сравнивании результатов двух решений
- Очень полезен на школьных олимпиадах, когда есть много времени, или когда уже написанно решение на маленькие подгруппы

Суть такая: * Есть решение `smart` — быстрое, но в котором есть баг, который хотим найти * Пишем решение `stupid` — медленное, но точно корректное * Пишем генератор `gen` — печатает какой-то корректный тест, сгенерированный случайно * Кормим всё в скрипт `checker`, который n раз генерирует тест, даёт его на ввод `stupid`-у и `smart`-у, сравнивает выводы и останавливается, когда они отличаются

Примерный код скрипта:

```
import os, sys

f1, f2, gen, iters = sys.argv

for i in range(int(iters)):
    print('Test', i+1)
    os.popen('python3 %s > test.txt' % gen)
    v1 = os.popen('./%s < test.txt' % f1).read()
    v2 = os.popen('./%s < test.txt' % f2).read()
    if v1 != v2:
        print test
        print("Correct:")
        print v1
        print("Wrong:")
        print v2
        break
```

Автор обычно запускает его командой `python3 checker.py stupid smart gen.py 100`, предварительно скомпилировав `stupid` и `smart` в ту же директорию, что и сам `checker.py`.

Скрипт написан под Linux. Для Windows нужно убрать «`./`» во всех системных вызовах.

`gen.py` автор тоже обычно пишет на питоне, но вообще его тоже можно писать на чём угодно, сделать исполняемым и вызывать через `./gen`. Пример `gen`-а, генерирующего случайную строку из символов “a”, “b” и “c” длины от 1 до 10:

```
from random import randint, choice

n = randint(1, 10)

print(n)

for _ in range(n):
    print(choice('abc'), end='')
```

Разреженная таблица

- Нужна для нахождения минимума на отрезке за $O(1)$ с препроцессингом за $O(n \log n)$ с малой константой.
- Обновления не поддерживает (static RMQ).
- Так как LCA эквивалентна RMQ, может быть применена в разных задачах на деревья.
- Её можно строить на ходу. Это может быть полезно при подсчете всяких динамик (см. последний Всерос).
- Требует $O(n \log n)$ памяти.
- Также можно считать, например, gcd на отрезке за сложность одного gcd, но в основном её используют для RMQ.

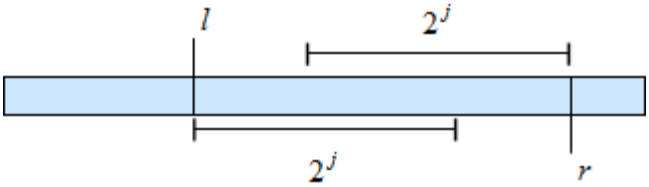
Определим разреженную таблицу как двумерный массив размера $n \times \log n$:

$$t[i][k] = \min\{a_i, a_{i+1}, \dots, a_{i+2^k-1}\}$$

Идея такая: считаем минимум на каждом отрезке длины 2^k.

Такой массив можно посчитать за его размер: $t[i][k] = \min(t[i][k-1], t[i+2^{k-1}][k-1])$. Считать его можно как итерируясь как по i, так и по k, причём какой-то из этих вариантов в несколко раз быстрее из-за кэширования.

Имея таком массив, мы можем в принципе для любого отрезка быстро посчитать минимум на нём. Нужно заметить, что у любого отрезка имеется два отрезка длины степени двойки, которые пересекаются, и, главное, покрывают его и только его целиком. Значит, мы можем просто взять минимум из значений, которые соответствуют этим отрезкам.



Последняя деталь: для того, чтобы константа на запрос стала настоящей, вместо функции log нужно предпосчитать массив округленных вниз логарифмов.

```
int a[maxn], lg[maxn], mx[maxn][logn];

int rmq (int l, int r) {
    int t = lg[r-l+1];
    return min(mx[l][t], mx[r-(1<<t)+1][t]);
}

// Это считается уже где-то в первых строчках main:

for (int l = 1; l < logn; l++)
    for (int i = (1<<l); i < maxn; i++)
        lg[i] = l;
```

```
for (int i = n-1; i >= 0; i--) {
    mx[i][0] = a[i];
    for (int l = 0; l < logn-1; l++)
        mx[i][l+1] = max(mx[i][l], mx[i+(1<<l)][l]);
}
```

2d Static RMQ

Эту структуру тоже можно обобщить на бо́льшие размерности. Пусть мы хотим посчитать RMQ на подквадратах. Тогда вместо массива `t[i][k]` у нас будет массив `t[i][j][k]`, в котором вместо минимума на отрезах будет храниться минимум на *квадратах* тех же степеней двоек. Получение минимума на произвольном квадрате тогда уже распадется на четыре минимума на квадратах длины 2^k.

В общем же случае от нас просят минимум тоже на прямоугольниках. Тогда делаем предподсчет, аналогичный предыдущему случаю, только теперь тут будет $O(n \log^d n)$ памяти и времени на предподсчет.

Алгоритмика

Структуры данных

- Дерево отрезков отложенные операции, динамическое, персистентное
- Декартово дерево treap, дерамида, неявный ключ, персистентное
- Дерево Фенвика многомерное дерево Фенвика, бинпоиск по дереву Фенвика
- Разреженная таблица sparse table, static RMQ
- Битовое сжатие битовые операции, std::bitset, перемножение матриц, метод Гаусса

Общие техники

- Корневая декомпозиция: она где-то есть
- Алгоритм Карацубы: введение в разделяй-и-властвуй, мастер-теорема

Дискретная математика

- Модулярная арифметика теорема Ферма, нахождение обратного по модулю, бинарное возведение в степень, диофантово уравнение, применения в комбинаторике, предподсчёт обратных факториалов за линейное время
- Ро-алгоритм Полларда парадокс дней рождений, факторизация за $O(\sqrt[4]{n})$
- Матроиды алгоритм Радо-Эдмондса, расписания, паросочетания, линейная независимость

Графы

- Остовные деревья алгоритм Прима, алгоритм Крускала
- Паросочетания(http://sereja.me/a/matching алгоритм Куна, покрытие ациклического орграфа, лемма Холла

Потоки

- Поток минимальной стоимости критерий оптимальности, отмена потока, потенциалы Джонсона, «дейкстра с потенциалами»

Деревья

- [Наименьший общий предок](http://sereja.me/a/lca LCA, двоичные подъемы, сведение LCA к RMQ, алгоритм Фараха-Колтона и Бендера
- Центроидная декомпозиция
- Heavy-light декомпозиция

Строки

- Полиномиальное хэширование хэши, парадокс дней рождений, хранение строк в декартовом дереве
- Поиск строки в строке префикс-функция, z-функция
- Бор
- Ахо-Корасик
- Суффиксный массив построение за $O(n \log n)$, LCP

Динамическое программирование

- Пересчёт динамики по слоям оптимизация Кнута, разделяй-и-властвуй, Convex Hull Trick, дискретный метод Лагранжа

Оптимизация

- Метод отжига задача о ферзях

Разное

- Стресс-тестирование

Теория игр

- Теория игр: эту статью определенно надо распилить на несколько

Высшая математика

- Линейная алгебра линейные операторы, матрицы, применения к динамике, метод Гаусса
- Теорвер

Геометрия

- Ликбез по вычислительной геометрии скалярное и векторное произведение, пересечение прямых, классы в C++
- Выпуклые оболочки: они где-то есть

Декартово дерево

Рене Декарт (фр. *René Descartes*) — великий французский математик и философ XVII века.

Рене Декарт не является создателем декартова дерева, но он является создателем декартовой системы координат, которую мы все знаем и любим.

Декартово дерево же определяется и строится так:

- Нанесём на плоскость набор из n точек. Их x зачем-то назовем *ключем*, а y *приоритетом*.
- Выберем самую верхнюю точку (с наибольшим y, а если таких несколько — любую) и назовём её *корнем*.
- От всех вершин, лежащих слева (с меньшим x) от корня, рекурсивно запустим этот же процесс. Если слева была хоть одна вершина, то присоединим корень левой части в качестве левого сына текущего корня.
- Аналогично, запустимся от правой части и добавим корню правого сына.

Заметим, что если все y и x различны, то дерево строится однозначно.

Если нарисовать получившуюся структуру на плоскости, то получится действительно дерево — по традиции, корнем вверх:

Таким образом, декартово дерево — это одновременно *бинарное дерево* по x и *куча* по y. Поэтому ему придумали много альтернативных названий:

- Дерамида (дерево + пирамида)
- ПиВо (пирамида + дерево)
- КуРево (куча + дерево)
- Treap (tree + heap)

Бинарные деревья

С небольшими модификациями, декартово дерево умеет всё то же, что и любое бинарное дерево поиска, например:

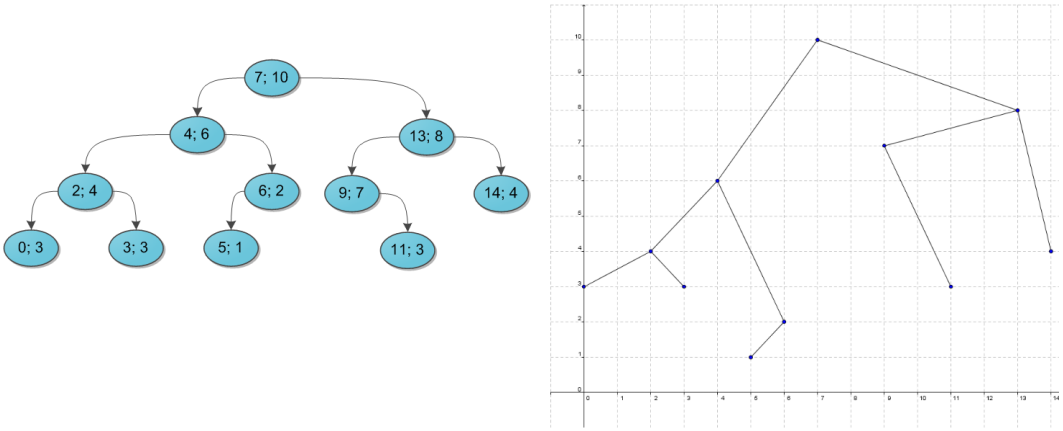


Figure 1: treap

- добавить число x в множество
- определить, есть ли в множестве число x
- найти первое число, не меньшее x (`lower_bound`)
- найти количество чисел в промежутке $[l, r]$

При этом все операции — за $O(\log n)$.

На самом деле, бинарных деревьев очень много. В большинстве из них время выполнения операций пропорционально высоте дерева, поэтому в них придумываются разные инварианты, позволяющие эту высоту минимизировать до $O(\log n)$.

Приоритеты и асимптотика

В декартовом дереве логарифмическая высота дерева гарантируется не инвариантами и эвристиками, а законами теории вероятностей: оказывается, что если все приоритеты (y) выбирать случайно, то средняя глубина вершины будет логарифмической. Поэтому ДД ещё называют рандомизированным деревом поиска.

Теорема. Ожидание глубины вершины в декартовом дереве равно $O(n \log n)$.

Если не знаете, что такое ожидание, то на доказательство забейте: это будет на занятии по теорверу в конце года. Сейчас его можно пропустить.

Доказательство. *Введем функцию $a(x, y)$ равную единице, если x является предком y, и нулем в противном случае. Такие функции называются* индикаторами*.

Глубина вершины равна количеству её предков — прим. К. О. Таким образом, она равна

$$d_i = \sum_{j=1}^{n} a(j, i)$$

Её матожидание равно

$$E[d_i] = E[\sum_{j \neq i} a(j,i)] = \sum_{j \neq i} E[a(j,i)] = \sum_{j \neq i} E[a(j,i)] = \sum_{j \neq i} p(j,i)$$

где $p(x,y)$ это веряотность, что $a(x,y) = 1$. Здесь мы воспользовались важным свойством линейности: матожидание суммы чего угодно равна сумме матожиданий этого чего угодно.

Ок, теперь осталось посчитать эти вероятности и сложить. Но сначала нам понадобится вспомогательное утверждение.

Лемма. Вершина x является предком y, если у неё приоритет больше, чем у всех вершин из отрезка $(x,y]$ (без ограничения общности, будем считать, что $x < y$).

Необходимость. Если это не так, то где-то между x и y есть вершина с большим приоритетом, чем x. Она не может быть потомком x, а значит x и y будут разделены.

Достаточность. Если справа будет какая-то вершина с большим приоритетом, то её левым сыном будет какая-то вершина, которая будет являться предком x. Таким образом, всё, что справа от y, ни на что влиять не будет.

У всех вершин на любом отрезке одинаковая вероятность иметь наибольший приоритет. Объединяя этот факт с результатом леммы, мы можем получить выражение для искомых вероятностей:

$$p(x,y) = \frac{1}{y-x+1}$$

Теперь, чтобы найти матожидание, эти вероятности надо просуммировать:

$$E[d_i] = \sum_{j \neq i} p(j,i) = \sum_{j \neq i} \frac{1}{|i-j|+1} \leq \sum_{i=1}^{n} \frac{1}{n} = O(\log n)$$

Перед последним переходом мы получили сумму гармонического ряда.

Примечательно, что ожидаемая глубина вершин зависит от их позиции: вершина из середины должна быть примерно в два раза глубже, чем крайняя.

Упражнение*. Выведите из этого доказательства асимптотику `quicksort`.

Реализация

Декартово дерево удобно писать на указателях и структурах. Поэтому мы рассказали дерево отрезков на указателях, а не стандартную рекурсию на 5 параметрах. Не знаете, что это такое — посмотрите в предыдущем конспекте.

Создадим структуру `Node`, в которой будем хранить ключ и приоритет, а также указатели на левого и правого сына. Указателя на корень дерева достаточно для идентификации всего дерева. Поэтому, когда мы будем говорить «функция принимает два дерева» на самом деле будут иметься в виду указатели на их корни. К нулевому указателю же мы будем относиться, как к «пустому» дереву.

```
struct Node {
    int key, prior;
    Node *l = 0, *r = 0;
    Node (int _key) { key = _key, prior = rand(); }
};
```

Объявим две вспомогательные функции, изменяющие структуру деревьев: одна будет разделять деревья, а другая объединять. Как мы увидим, через них можно легко выразить почти все функции, которые нам потом понадобятся.

Merge

Принимает два дерева (два корня, L и R), про которые известно, что в левом все вершины имеют меньший ключ, чем все в правом. Их нужно объединить в одно дерево так, чтобы ничего не сломалось: по ключам это всё ещё дерево, а по приоритетами — куча.

Сначала выберем, какая вершина будет корнем. Здесь всего два кандидата — левый корень L или правый R — просто возьмем тот, у кого приоритет больше.

Пусть, для однозначности, это был левый корень. Тогда левый сын корня итогового дерева должен быть левым сыном L. С правым сыном сложнее: возможно, его нужно смерджить с R. Поэтому рекурсивно сделаем `merge(l->r, r)` и запишем результат в качестве правого сына.

```
Node* merge (Node *l, Node *r) {
    if (!l) return r;
    if (!r) return l;
    if (l->prior > r->prior) {
        l->r = merge(l->r, r);
        return l;
    }
    else {
        r->l = merge(l, r->l);
        return r;
    }
}
```

Split

Принимает дерево и ключ x, по которому его нужно разделить на два: L должно иметь все ключи не больше x, а R должно иметь все ключи больше x.

В этой функции мы сначала решим, в каком из деревьев должен быть корень, а потом рекурсивно разделим его правую или левую половину и присоединим, куда надо:

```
typedef pair<Node*, Node*> Pair;

Pair split (Node *p, int x) {
    if (!p) return {0, 0};
    if (p->key <= x) {
```

```
        Pair q = split(p->r, x);
        p->r = q.first;
        return {p, q.second};
    }
    else {
        Pair q = split(p->l, x);
        p->l = q.second;
        return {q.first, p};
    }
}
```

Пример: вставка

`merge` и `split` сами по себе не очень полезные, но помогут написать все остальное.

Вот так, например, будет выглядеть код, добавляющий x в сет.

```
Node *root = 0;

void insert (int x) {
    Pair q = split(root, x);
    Node *t = new Node(x);
    root = merge(q.first, merge(t, q.second));
}
```

Пример: модификация для суммы на отрезке

Иногда нам нужно написать какие-то модификации для более продвинутых операций.

Например, нам может быть интересно иногда считать сумму чисел на отрезке. Для этого в вершине нужно хранить также своё число и сумму на своем «отрезке».

```
struct Node {
    int val, sum;
    // ...
};
```

При `merge` и `split` надо будет поддерживать эту сумму актуальной.

Вместо того, чтобы модифицировать и `merge`, и `split` под наши хотелки, напишем вспомогательные функцию `upd`, которую будем вызывать при обновлении детей вершины.

```
void sum (Node* v) { return v ? v->sum : 0; }
// обращаться по пустому указателю нельзя -- выдаст ошибку

void upd (Node* v) { v->sum = sum(v->l) + sum(v->r) + v->val; }
```

В `merge` и `split` теперь можно просто вызывать `upd` перед тем, как вернуть вершину, и тогда ничего не сломается:

```
Node* merge (Node *l, Node *r) {
    // ...
    if (...) {
        l->r = merge(l->r, r);
        upd(l);
        return l;
    }
    else {
        // ...
    }
}

typedef pair<Node*, Node*> Pair;

Pair split (Node *p, int x) {
    // ...
    if (...) {
        // ...
        upd(p);
        return {p, q.second};
    }
    else {
        // ...
    }
}
```

Тогда при запросе суммы нужно просто вырезать нужный отрезок и запросить эту сумму:

```
int sum (int l, int r) {
    Pair rq = split(root, r);
    Pair lq = split(rq.first, l);
    int res = sum(lr.second);
    root = merge(lq.first, merge(lq.second, rq.second));
    return res;
}
```

Неявный ключ

Обычное декартово дерево — это структура для множеств, каждый элемент которых имеет какой-то ключ. Эти ключи задают на этом множестве какой-то порядок, и все запросы к ДД обычно как-то привязаны к этому порядку.

Но что, если у нас есть запросы, которые этот порядок как-то нетривиально меняют? Например, если у нас есть массив, в котором нужно уметь выводить сумму на произвольном отрезке и «переворачивать» произвольный отрезок. Если бы не было второй операции, мы бы просто использовали индекс элемента в качестве ключа, но с операцией переворота нет способа их быстро поддерживать актуальными.

Решение такое: выкинем ключи, а вместо них будем поддерживать информацию, которая поможет

неявно восстановить ключ, когда он нам будет нужен. А именно, будем хранить вместе с каждой вершиной размер её поддерева:

```
struct Node {
    int key, prior, size = 1;
    //              ^ размер поддерева
    Node *l = 0, *r = 0;
    Node (int _key) { key = _key, prior = rand(); }
};
```

Размеры поддеревьев будем поддерживать по аналогии с суммой — напишем вспомогательную функцию, которую будем вызывать после каждого структурного изменения вершины.

```
int size (Node *v) { return v ? v->size : 0; }

void upd (Node *v) { v->size = 1 + size(v->l) + size(v->r); }
```

`merge` не меняется, а вот в `split` нужно использовать позицию корня вместо его ключа.

Про `split` теперь удобнее думать как “вырежи первые k элементов”.

```
typedef pair<Node*, Node*> Pair;

Pair split (Node *p, int k) {
    if (!p) return {0, 0};
    if (size(p->l) + 1 <= k) {
        Pair q = split(p->r, k - size(p->l) - 1);
        //                   ^ правый сын не знает количество вершин слева от него
        p->r = q.first;
        upd(p);
        return {p, q.second};
    }
    else {
        Pair q = split(p->l, k);
        p->l = q.second;
        upd(p);
        return {q.first, p};
    }
}
```

Всё. Теперь у нас есть клёвая гибкая структура, которую можно резать как угодно.

Пример: ctrl+x, ctrl+v

```
Node* ctrlx (int l, int r) {
    Pair q1 = split(root, r);
    Pair q2 = split(q1.first, l);
    root = merge(q2.first, q1.second);
    return q2.second;
}
```

```
void ctrlv (Node *v, int k) {
    Pair q = split(root, k);
    root = merge(q.first, merge(v, q.second));
}
```

Пример: переворот

Нужно за $O(\log n)$ обрабатывать запросы переворота произвольных подстрок: значение a_l поменять с a_r, a_{l+1} поменять с a_{r-1} и т. д.

Будем хранить в кажой вершине флаг, который будет означать, что её подотрезок перевернут:

```
struct Node {
    bool rev;
    // ...
};
```

Поступим по аналогии с ДО — когда мы когда-либо встретим такую вершину, мы поменяем ссылки на её детей, а им самим передадим эту метку:

```
void push (node *v) {
    if (v->rev) {
        swap(v->l, v->r);
        if (v->l)
            v->rev ^= 1;
        if (v->r)
            v->rev ^= 1;
    }
    v->rev = 0;
}
```

Аналогично, эту функцию будем вызывать в начале `merge` и `split`.

Саму функцию `reverse` реализуем так: вырезать нужный отрезок, поменять флаг.

```
void reverse (int l, int r) {
    Pair q1 = split(root, r);
    Pair q2 = split(q1.first, l)
    q2.second->rev ^= 1;
    root = merge(q2.first, merge(q2.second, q1.second));
}
```

Функциональное программирование*

Реализация большинства операций всегда примерно одинаковая — вырезаем отрезок с l по r, что-то с ним делаем и склеиваем обратно.

Дублирующийся код — это плохо. Давайте используем всю мощь плюсов и определим функцию, которая принимает другую функцию, которая уже делает полезные вещи на нужном отрезке.

```
auto apply (int l, int r, auto f) {
    Pair q1 = split(root, r);
    Pair q2 = split(q1.first, l)
    q2.second = f(q2.second);
    root = merge(q2.first, merge(q2.second, q1.second));
}

void reverse (Node *v) {
    if (v)
        v->rev ^= 1;
}
```

Применять её нужно так:

```
apply(l, r, reverse);
```

Это работает в плюсах, начиная с `g++14`.

Для простых операций можно даже написать лямбду:

```
apply(l, r, [](Node *v){
    if (v)
        v->rev ^= 1;
});
```

Персистентность*

Так же, как и с ДО, персистентной версией ДД можно решать очень интересные задачи.

> Дана строка. Требуется выполнять в ней копирования, удаления и вставки в произвольные позиции.

Построим персистентное ДД. Тогда просто вызвав два `split`-а, мы можем получить копию любой подстроки (указатель вершину), которую потом можно вставлять куда угодно, при этом оригинальную подстроку мы не изменим.

> Дана строка. Требуется выполнять в ней копирования, удаления, вставки в произвольные позиции **и сравнение произвольных подстрок**.

Можно в вершинах хранить **полиномиальный хэш** соответствующей подстроки. Тогда мы можем проверять равенство подстрок сравниванием хэшей вершин, полученных теми же двумя сплитами.

Чтобы полноценно сравнивать стоки лексикографически, можно применить бинарный поиск: перебрать длину совпадающего суффикса, и, когда она найдется, посмотреть на следующий символ.

Реализация почти такая же, как и для всех персистентных структур на ссылках — перед тем, как идти в какую-то вершину, нужно создать её копию и идти в неё. Создадим для этого вспомогательную функцию `copy`:

```
Node* copy (Node *v) { return new Node(*v); }
```

Во всех методах мы будем начинать с копирования всех упоминаемых в ней вершин. Например, персистентный `split` начнётся так:

```
Pair split (Node *p, int x) {
    p = copy(p);
    // ...
}
```

В ДО просто создавать копии вершин было достаточно. Этого обычно достаточно для всех детерминированных структур данных, но в ДД всё сложнее. Оказывается существует тест, который «валит» приоритеты: можно раскопировать много версий одной вершины, а все остальные — удалить. Тогда у всех вершин будет один и тот же приоритет, и дерево превратится в «бамбук», в котором все операции будут работать за линию.

У этой проблемы есть очень элегантное решение — избавиться от приоритетов, и делать теперь следующее переподвешивание: если размер левого дерева равен L, а размер правого R, то будем подвешивать за левое с вероятностью $\frac{L}{L+R}$, иначе за правое.

Теорема. Такое переподвешивание эквивалентно приоритетам.

Доказательство. Покажем, что все вершины всё так же имеют равную вероятность быть корнем. Докажем по индукции:

- Лист имеет вероятность 1 быть корнем себя (база индукции)
- Переход индукции — операция `merge`. Любая вершина левого дерева была корнем с вероятностью $\frac{1}{L}$ (по предположению индукции), а после слияния она будет корнем всего дерева с вероятностью $\frac{1}{L} \cdot \frac{L}{L+R} = \frac{1}{L+R}$. С вершинами правого дерева аналогично.

Получается, что при таком переподвешивании всё так же каждая вершина любого поддерева равновероятно могла быть его корнем, а на этом основывалось наше доказательство асимптотики ДД.

```
Node* merge (Node *l, Node *r) {
    if (!l) return r;
    if (!r) return l;
    l = copy(l), r = copy(r);
    if (rand() % (size(l) + size(r)) < size(l)) {
        // ...
    }
    else {
        // ...
    }
}
```

Философский вопрос: можно ли декартово дерево называть декартовым, если из него удалить и x, и y?

Дерево отрезков

Замечание. Почти везде мы будем использовать полуинтервалы — обозначаемые как $[l, r)$ — вместо отрезков. Несмотря на контринтуитивность, это немного упростит код и вообще является

хорошей практикой в программировании, подобно нумерации с нуля.

Дерево отрезков — очень мощная и гибкая структура данных, позволяющая быстро отвечать на самые разные запросы на отрезках.

Рассмотрим конкретную задачу:

> Дан массив a из n целых чисел, нужно уметь отвечать на запросы двух типов:
>
> 1. Изменить значение в ячейке (т. е. отреагировать на присвоение `a[k] = x`).
> 2. Вывести сумму элементов a_i на отрезке с l по r.
>
> Оба запроса нужно обрабатывать за время $O(\log n)$.

Чтобы решить задачу, сделаем с исходным массивом следующие манипуляции:

Посчитаем сумму всего массива и где-нибудь запишем. Потом разделим его пополам и посчитаем сумму на половинах и тоже где-нибудь запишем. Каждую половину потом разделим пополам ещё раз, и так далее, пока не придём к отрезкам длины 1.

Эту последовательность разбиений можно представить в виде дерева. Корень этого дерева соответствует отрезку $[0, n)$, а каждая вершина (не считая листьев) имеет ровно двух сыновей, которые тоже соответствуют каким-то отрезкам. Отсюда и название — «дерево отрезков».

<table>
<tr><td colspan="16">1: [0, 16)</td></tr>
<tr><td colspan="8">2: [0, 8)</td><td colspan="8">3: [8, 16)</td></tr>
<tr><td colspan="4">4: [0, 4)</td><td colspan="4">5: [4, 8)</td><td colspan="4">6: [8, 12)</td><td colspan="4">7: [12, 16)</td></tr>
<tr><td colspan="2">8:
[0, 2)</td><td colspan="2">9:
[2, 4)</td><td colspan="2">10:
[4, 6)</td><td colspan="2">11:
[6, 8)</td><td colspan="2">12:
[8, 10)</td><td colspan="2">13:
[10, 12)</td><td colspan="2">14:
[12, 14)</td><td colspan="2">15:
[14, 16)</td></tr>
<tr><td>16:
0</td><td>17:
1</td><td>18:
2</td><td>19:
3</td><td>20:
4</td><td>21:
5</td><td>22:
6</td><td>23:
7</td><td>24:
8</td><td>25:
9</td><td>26:
10</td><td>27:
11</td><td>28:
12</td><td>29:
13</td><td>30:
14</td><td>31:
15</td></tr>
</table>

Figure 2: alt text

Строить его можно рекурсивной функцией: * Если вершина является листом, взять в качестве суммы значение соответствующей ячейки. * Если вершина является отрезком, разделить его на два и в качестве суммы взять сумму его детей.

Разные свойства

Высота такого дерева есть величина $\Theta(\log n)$: на каждом новом уровне длина отрезка уменьшается вдвое. Этот факт будет ключевым для оценки асимптотики.

Более того, любой полуинтервал разбивается на $O(\log n)$ неперекрывающихся полуинтервалов, соответствующих в вершинам дерева: с каждого уровня нам достаточно не более двух отрезков.

Дерево также содержит менее $2n$ вершин: первый уровень дерева отрезков содержит одну вершину (корень), второй уровень — в худшем случае две вершины, на третьем уровне в худшем случае будет четыре вершины, и так далее, пока число вершин не достигнет n. Таким образом, число вершин в худшем случае оценивается суммой $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... + 1 < 2n$. Значит, оно линейное по памяти.

При n, отличных от степеней двойки, не все уровни дерева отрезков будут полностью заполнены. Например, при $n = 3$ левый сын корня есть отрезок $[0, 2)$, имеющий двух потомков, в то время как правый сын корня — отрезок $[2, 3)$, являющийся листом.

Ок, как это нам поможет?

Опишем теперь, как с помощью такой структуры решить задачу.

Запрос обновления. Нам нужно обновить значения в вершинах таким образом, чтобы они соответствовали новому значению $a[k] = x$.

Изменим все вершины, в суммах которых участвует k-тый элемент. Их будет $\Theta(\log n)$ — по одной с каждого уровня.

Это можно реализовать как рекурсивную функцию: ей передаётся текущая вершина дерева отрезков, и эта функция выполняет рекурсивный вызов от одного из двух своих сыновей (от того, который содержит k-ый элемент в своём отрезке), а после этого — пересчитывает значение суммы в текущей вершине точно таким же образом, как мы это делали при построении дерева отрезков.

Запрос суммы. Мы знаем, что во всех вершинах лежат корректные значения.

Сделаем тоже рекурсивную функцию, рассмотрев три случая:

- Если отрезок вершины лежит целиком в отрезке запроса, то вернуть записанную в ней сумму.
- Если отрезки вершины и запроса не пересекаются, то вернуть 0.
- Иначе разделиться рекурсивно на 2 и вернуть сумму этой функции от обоих детей.

Чтобы разобраться, почему это работает за $O(\log n)$, нужно оценить количество «интересных» отрезков — тех, которые порождают новые вызовы рекурсии. Это будут только те, которые содержат границу запросов — остальные сразу завершатся. Обе границы отрезка содержатся в $O(\log n)$ отрезках, а значит и итоговая асимптотика будет такая же.

Ликбез по C++

Наша реализация будет на указателях. Никто не говорит, что она самая лучшая (см. раздел «Другие реализации»), но она самая понятная. Вам может поначалу показаться, что она слишком сложная, но позже вы поймёте её преимущества.

Но сначала нам нужно рассказать про объектно-ориентированное программирование и некоторые фишки C++. Если вы их уже знаете, то можете пропускать этот раздел.

Объект — это сущность, которой можно посылать сообщения и которая может на них реагировать, используя свои данные. Инкапсулировать логику в объекты на самом деле очень удобно. Дереву отрезков не важно знать, как устроен окружающий мир, а миру не важно, как внутри устроено дерево отрезков — это просто какая-то структура, которая умеет делать нужные операции за $O(\log n)$.

В C++ есть два способа объявлять классы (объект — это экземпляр класса): через `struct` и через `class`. Их основное отличие в том, что по умолчанию в `class` все поля приватные — к

ним нет прямого доступа снаружи. Это нужно для дополнительной защиты, чтобы в крупных промышленных проектах никто случайно ничего не поломал, но на олимпиадах это не очень актуально.

У классов есть поля (переменные) и методы (функции, привязанные к объектам). Среди них есть особые, например **конструктор** — он вызывается при создании объекта. Чтобы объявить конструктор класса в C++, нужно объявить внутри него метод с тем же названием, что и у самого класса.

```
struct A {
    int param1, param2; // тут можно что-то хранить
    char param3 = 'k';
    A (int var) {
        // эта часть называется конструктором
        // ...
    }
    void do_something () {
        // это какой-то другой метод
        // ...
    }
}; // <- не забудьте точку с запятой
```

Другое важное понятие — указатель. Память можно представлять как просто очень большой массив. На самом деле, когда мы создаем какой-то объект, отдельная программа (*аллокатор*) выделяет место в массиве (*оперативной памяти*) под этот объект и возвращает позицию (*указатель*) на место в этом массиве.

Указатели нам нужны для того, чтобы хранить ссылки на детей. Имея указатель на объект, можно делать всё то же, что и имея сам объект, только синтаксис немного поменяется:

```
A x(179);
x.do_something();
x.param1 = 57;

A *y = new A(42); // new возвращает адрес, по которому можно найти объект
y->do_something();
y.param3 = '!';
```

Кстати, вы не задумывались, почему мы перешли с 32-битных процессоров на 64-битные? Каждый указатель ссылается на байт — более точный адрес менеджер памяти выделять не умеет. Поэтому 32-битный компьютер умеет работать только с не более, чем 2^{32} байтами памяти — ровно 4 гигабайта — что с какого-то момента начало нехватать. Большинство операций — это операции с памятью, и размерность повысили именно из-за этого, а не чтобы операции с `long long` быстрее считались

Реализация

Общий план реализации любых структур данных: 1. Полностью понять все *инварианты* — как должна выглядеть структура, какие значения должны принимать поля, etc. 2. Формально описать,

что должны делать методы и за какую асимптотику. 3. Решить много отдельных задач, реализуя методы, не нарушающие инварианты.

```
struct segtree {
    int lb, rb; // левые и правые границы отрезков
    int sum = 0; // сумма на текущем отрезке
    segtree *l = 0, *r = 0;
    segtree (int _lb, int _rb) {
        lb = _lb, rb = _rb;
        if (lb + 1 < rb) {
            // если не лист, создаем детей
            int t = (lb + rb) / 2;
            l = new segtree(lb, t);
            r = new segtree(t, rb);
        }
    }
    void add (int k, int x) {
        sum += x;
        if (l) {
            if (k < l->rb)
                l->add(k, x);
            else
                r->add(k, x);
        }
    }
    int get_sum (int lq, int rq) {
        if (lb >= lq && rb <= rq)
            // если мы лежим полностью в отрезке запроса, вывести сумму
            return sum;
        if (max(lb, lq) >= min(rb, rq))
            // если мы не пересекаемся с отрезком запроса, вывести ноль
            return 0;
        // иначе всё сложно -- запускаемся от детей и пусть они там сами решают
        return l->get_sum(lq, rq) + r->get_sum(lq, rq);
    }
};
```

> Посчитать число беспорядков в перестановке из n элементов (беспорядок или инверсия — это пара чисел $i < j$, для которых $p_i > p_j$).

Эта задача решается просто, если уметь писать сортировку слиянием вручную. Но мы пойдем по другому пути. Создадим ДО для суммы на n элементов, изначально заполненное нулями. Теперь будем проходить по этому массиву слева направа. Когда обрабатываем очередное число x, будем делать две вещи: * Запросим сумму от k до n в ДО. * Добавим единичку в k-тую позицию в ДО.

Так мы для каждой инверсии учтём её, когда запросим сумму для её правого элемента. Таким образом, мы решили эту задачу за $O(n \log n)$ запросов.

> Даны n точек на плоскости с целыми координатами от 1до n. Требуется ответить на m запросов количества точек на прямоугольнике.

Ответим на все запросы в оффлайн, используя метод сканирующей прямой:

- Разобьем запросы суммы на прямоугольнике на два запроса суммы на префиксах — сумма на прямоугольнике $[x_1, x_2] \times [y_1, y_2]$ равна сумме на прямоугольнике $[0, x_2] \times [y_1, y_2]$ минус сумма на прямоугольнике $[0, x_1] \times [y_1, y_2]$.
- Отсортируем теперь все точки и префиксные запросы по их x. При этом, если у точки и запроса одинаковый x, то точка должна идти раньше.
- Пройдёмся по ним в таком порядке и будем решать задачу для одномерной суммы: у нас есть операция «сделать +1 в y_i» и «вывести сумму с y_1 по y_2».

Отложенные операции

Пусть теперь наш запрос обновления — это присвоение значения x всем элементам некоторого отрезка $[l, r)$, а не только одному.

Мы не хотим спускаться до каждого элемента, где меняется сумма — их может быть очень много. Мы схитрим, и при запросе присваивания будем, по возможности, помечать некоторые вершины, что они и все их дети «покрашены» в какое-то число. Непосредственно спускаться до листьев мы не будем.

Например, если пришел запрос «присвой число x на всем массиве», то мы вообще фактических присвоений делать не будем — только оставим пометку в корне дерева, что оно покрашено.

Когда нам позже понадобятся правильные значения таких вершин и их детей, мы будем делать «проталкивание» информации из текущей вершины в её сыновей: если метка стоит, пересчитаем сумму текущего отрезка и передадим эту метку сыновьям. Когда нам потом понадобятся сыновья, мы будем делать то же самое. Подобная операция будет гарантировать корректность данных в вершине ровно к тому моменту, когда они нам понадобятся.

Понятно, что от использования таких «запаздывающих» обновлений асимптотика никак не уходшается, и мы можем всё так же решить задачу за $O(n \log n)$.

При реализации создадим вспомогательную функцию `push`, которая будет производить проталкивание информации из этой вершины в обоих её сыновей. Вызывать её стоит в самом начале обработки любого запроса — тогда она гарантирует, что в текущей вершине и её сыновьях все значения корректны.

```
struct segtree {
    int lb, rb;
    int sum = 0, assign = -1;
    segtree *l = 0, *r = 0;
    segtree (int _lb, int _rb) {
        lb = _lb, rb = _rb;
        if (lb + 1 < rb) {
            int t = (lb + rb) / 2;
            l = new segtree(lb, t);
            r = new segtree(t, rb);
        }
    }
    void push () {
        if (assign != -1) {
```

```
            sum = (rb-lb) * assign;
            if (l) { // если дети есть
                l->assign = assign;
                r->assign = assign;
            }
        }
        assign = -1;
    }
    void upd (int lq, int rq, int x) {
        push();
        if (lq <= lb && rb <= rq)
            assign = x;
        else if (l && max(lb, lq) < min(rb, rq)) {
            // если есть дети и отрезок запроса хоть как-то пересекается с нашим
            l->upd(lq, rq, x);
            r->upd(lq, rq, x);
            // ...дальше они сами разберутся
        }
    }
    int get_sum (int lq, int rq) {
        push();
        if (lb >= lq && rb <= rq)
            return sum;
        if (max(lb, lq) >= min(rb, rq))
            return 0;
        return l->get_sum(lq, rq) + r->get_sum(lq, rq);
    }
};
```

По-английски эта техника называется *lazy propagation*. **Очень важно научиться её писать** — она часто встречается на олимпиадах.

Идея «давайте будем всё делать в последний момент» применима не только в ДО, но и в других структурах и в реальной жизни.

Динамическое построение

А что, если у нас все индексы лежать не от в пределах 10^5, а, например, 10^9. Все асимптотики нас по прежнему устраивают ($\log_2 10^6 \approx 20$, $\log_2 10^9 \approx 30$), кроме этапа построения.

Можно решить эту проблему так: откажемся от явного создания всех вершин дерева изначально. Изначально создадим только лишь корень, а остальные вершины будем создавать на ходу, когда в них потребуется записать что-то не дефолтное — как в lazy propagation.

Реализовать это можно так же, как и с `push`-ем: в начале всех методов будем проверять, что дети-вершины созданы, и создавать их, если это не так.

```
struct segtree {
    int lb, rb;
    int sum = 0;
```

```
    segtree *l = 0, *r = 0;
    segtree (int _lb, int _rb) {
        lb = _lb, rb = _rb;
        // а тут ничего нет
    }
    void extend () {
        if (!l && lb + 1 < rb) {
            int t = (lb + rb) / 2;
            l = new segtree(lb, t);
            r = new segtree(t, rb);
        }
    }
    void add (int k, int x) {
        extend();
        sum += x;
        if (l) {
            if (k < l->rb)
                l->add(k, x);
            else
                r->add(k, x);
        }
    }
    int get_sum (int lq, int rq) {
        if (lb >= lq && rb <= rq)
            return sum;
        if (max(lb, lq) >= min(rb, rq))
            return 0;
        extend();
        return l->get_sum(lq, rq) + r->get_sum(lq, rq);
    }
};
```

Но вообще, в большинстве случаев, использовать динамическое построение — это как стрелять из пушки по воробьям. Если все запросы известны заранее, то их координаты можно просто сжать перед обработкой запросов. Автор обычно делает это так:

```
vector<int> compress (vector<int> a) {
    vector<int> b = a;
    sort(b.begin(), b.end());
    b.erase(unique(b.begin(), b.end()), b.end());
    for (int &x : a)
        x = int(lower_bound(b.begin(), b.end(), x) - b.begin());
    return a;
}
```

Персистентность

Структуры данных называют **персистентными**, если их можно быстро «откатить» до произвольного предыдущего состояния.

Известны персистентные версии многих структур: стэка, очереди, СНМ, ДО. В случае со структурами данных на ссылках есть следующий общий подход: во всех методах, меняющих значения в вершинах, будем копировать ссылки на детей перед тем, как в них переходить и что-либо менять. Таким образом, мы всегда будем делать копию вершины перед тем, как что-либо менять в ней самой или её потомках. Вершины в момент t никогда не будут ссылаться на вершины, измененные после этого, и поэтому ничего не сломается.

У персистентных структур есть один минус: они обычно требуют больше памяти. В случае ДО мы будем создавать $O(\log n)$ новых вершин на запрос, что означает общее потребление памяти $O(m \log n)$.

```
struct segtree {
    int lb, rb;
    int sum = 0;
    segtree *l = 0, *r = 0;
    segtree (int _lb, int _rb) {
        lb = _lb, rb = _rb;
        if (lb != rb) {
            int t = (lb + rb) / 2;
            l = new segtree(lb, t);
            r = new segtree(t, rb);
        }
    }
    void copy () {
        if (l) {
            l = new segtree(l);
            r = new segtree(r);
        }
    }
    void add (int k, int x) {
        copy();
        sum += x;
        if (l) {
            if (k < l->rb) l->add(k, x);
            else r->add(k, x);
        }
    }
    int get_sum (int lq, int rq) {
        // этот метод ничего не меняет -- он и так хороший
        if (lq <= lb && rb <= rq)
            return sum;
        if (max(lb, lq) >= min(rb, rq))
            return 0;
        return l->get_sum(lq, rq) + r->get_sum(lq, rq);
```

```
    }
};
```

> Даны n точек на плоскости. Нужно *в онлайн* ответить на q запросов суммы на прямоугольнике.

Если бы можно было отвечать в оффлайн, мы бы воспользовались методом сканирующей прямой — но так делать мы не можем. Вместо этого мы будем таким же образом добавлять точки в порядке увеличения x_i и декомпозировать запрос суммы на два, но при ответе на эти запросы мы будем доставать соответствующую версию ДО, которую мы получили, обработав нужное количество точек. Таким образом, можно отвечать на запросы в онлайн, но с $O(n \log n)$ памяти.

> Дан отрезок из n чисел от 1 до n. Требуется ответить на q запросов k-той порядковой статистики на подотрезке.

Сделаем такой стандартный препроцессинг: пройдёмся с персистентным деревом отрезков для суммы по массиву. Когда будем обрабатывать элемент k, добавим единицу к k-ому элементу.

Дальше определим *разность деревьев* как дерево отрезков, которое соответствует разности массивов. Заметим, что он неотрицательный. Его можно получить неявно, спускаясь одновременно в двух ДО и вместо `sum` использовать везде `sum_r - sum_l`.

Что будет находиться в разности r-го и l-го дерева? Там будут количества вхождений чисел на этом отрезке. В таком ДО не составить труда сделать спуск, который находит последнюю позицию, у которой сумма на соответствующем префиксе не превышает k — она и будет ответом.

> Дан массив из n элементов. Требуется ответить на m запросов, есть ли на отрезке $[l, r]$ доминирующий элемент — тот, который встречается на нём хотя бы $\frac{r-l}{2}$ раз.

У этой задачи есть удивительно простое решение — взять около 100 случайных элементов и каждый проверить, является ли он доминирующим (это можно проверить за $O(\log n)$, посчитав для каждого значения отсортированный список позиций, на которых он встречается, и сделав два бинпоиска). Вероятность ошибки в худшем случае равна $\frac{1}{2^{100}}$, и ей на практике можно пренебречь.

Но проверять 100 сэмплов — долго. Можно построить такое же ДО, как в прошлой задаче, и решать задачу «найти число, большее $\frac{n}{2}$ в массиве на n элементов». Это тоже будет спуском по ДО: каждый раз идём в того сына, где сумма больше. Если в листе, куда мы пришли, значение больше нужного, возвращаем `true`, иначе `false`.

Другие реализации

Эта реализация проста и легко расширяема, но весьма медленная и неэффективная по памяти. Есть альтернативы:

На массивах. Можно ввести несложную нумерацию вершин, позволяющую при спуске в ребёнка пересчитывать его номер. Это позволит не хранить границы текущего отрезка. Подробнее у Емакса.

«ДО снизу». Можно делать все операции итеративно — так получится раз в 7 быстрее, но писать что-либо нетривиальное (например, массовые операции) так будет намного труднее. Подробнее смотрите в одном посте с CodeForces.

Задачи

- Первый контест — на базовые операции.
- Второй контест — на отложенные операции.
- Дополнительный контест — на динамическое построение и персистентность.

Heavy-light декомпозиция

HL-декомпозиция — это мощный метод решения задач на запросы на путях, когда также есть запросы обновлений. Если запросов обновления нет, то лучше написать что-нибудь попроще.

*

TODO: найти менее уродливую иллюстрацию

•

Heavy-light декомпозицией корневого дерева называется результат следующего процесса: каждой вершины v посмотрим на всех её непосредственных детей u, выберем среди них ребёнка u_{max} (с самым большим размером поддерева) и назовём ребро (v, u) *тяжелым* (heavy), а все остальные рёбра — *лёгкими* (light). При этом, если самых «тяжелых» детей будет больше одного, то выберем любого.

Таким образом, дерево распалось на тяжелые и лёгкие рёбра. Докажем несколько полезных свойств такого разбиения.

Утверждение. Дерево разбивается на непересекающиеся пути из тяжелых рёбер.

Доказательство. В каждую вершину входит не более одного тяжелого ребра, и из каждой вершины исходит не более одного тяжелого ребра.

Назовём *блоком* либо лёгкое ребро, либо вертикальный путь из тяжелых рёбер.

Утверждение. На любом вертикальном пути будет не более $O(\log n)$ блоков.

Доказательство разбивается на две части:

- Лёгких ребер на вертикальном пути будет не более $O(\log n)$: рассмотрим самую нижнюю вершину и будем идти по вертикальному пути снизу вверх. Каждый раз, когда мы переходим по лёгкому ребру, размер поддерева текущей вершины увеличивается в два раза, потому что если вершина связана с родителем лёгким ребром, то у него есть какой-то другой ребёнок, который не легче текущего.
- Непрерывных путей из тяжелых рёбер будет не более $O(\log n$: если это не конец или начало пути, то каждый такой путь окружают два лёгких ребра, а их всего $O(\log n)$.

Следствие. На любом пути будет не более $O(\log n)$ блоков.

Ради этого мы всё и делали: теперь построим какую-нибудь структуру на каждом тяжелого пути (например, дерево отрезков), а при ответе на запрос (скажем, суммы на пути) разобьём его на $O(\log n)$ запросов либо к подотрезкам тяжелых путей, либо к лёгким рёбрам.

Реализация

Большинство публичных реализаций HLD — это 120-150 строк кода. Мы же воспользуемся следующим трюком, который сильно упростит нам жизнь: перенумеруем вершины дерева так, что для каждого тяжелого пути все его вершины будут иметь последовательные номера.

А именно, на этапе подсчёта размера поддеревьев, изменим список смежности каждой вершины так, чтобы в самом начале шел её «тяжелый» ребёнок. Тогда, если запустить обычный эйлеров обход графа, то `tin`-ы и будут нужной нумерацией, потому что в каждой вершине мы шли в своего тяжелого ребёнка в первую очередь.

Теперь мы можем построить какую-нибудь структуру поверх массива размера n (дерево отрезков) и при запросе к какому-нибудь тяжелому пути делать запрос к отрезку в структуре.

```
vector<int> g[maxn];
int s[maxn], p[maxn], tin[maxn], tout[maxn];
int head[maxn]; // «голова» тяжелого пути, которому принадлежит v
int t = 0;

void sizes (int v = 0) {
    s[v] = 1;
    for (int &u : g[v]) {
        sizes(u);
        s[v] += s[u];
        if (s[u] > s[g[v][0]])
            // &u -- это ссылка, так что её легально использовать при swap-е
            swap(u, g[v][0]);
    }
}

void hld (int v = 0) {
    rin[t] = v;
    tin[v] = t++;
    for (int u : g[v]) {
        // если это тяжелый ребенок -- его next нужно передать
        // в противном случае он сам является головой нового пути
        head[u] = (u == g[v][0] ? head[v] : u);
        hld(u);
    }
    tout[v] = t;
}
```

Как им решать задачи

Простейший пример задачи на HLD: дано дерево, каждой вершине которого приписано какое-то число, и поступают запросы двух типов:

1. Узнать минимальное число на пути между v_i и u_i.
2. Изменить число у v_i-той вершины на x_i.

Подвесим дерево за произвольную вершину и построим на нём HL-декомпозицию с деревом отрезков в качестве внутренней структуры. Его код мы приводить не будем и посчитаем, что оно реализовано примерно так же, как в соответствующей статье и имеет методы `upd(k, x)` и `get_min(l, r)`.

```
int val[maxn];
segtree st(0, n);
```

При операции обновления нам нужно просто обновить нужную ячейку в дереве отрезков:

```
void upd (int v, int x) {
    st.upd(tin[v], x);
}
```

Запрос минимума сложнее: нам нужно разбить исходный запрос на запросы к вертикальным путям.

```
int ancestor (int a, int b) {
    return tin[a] <= tin[b] && tin[b] <= tout[a];
}

void up (int &a, int &b, int &ans) {
    while (!ancestor(head[a], b)) {
        ans = min(ans, st.get_min(tin[head[a]], tin[a]));
        a = p[head[a]];
    }
}

int get_min (int a, int b) {
    int ans = inf;
    up(a, b, ans);
    up(b, a, ans);
    if (!ancestor(a, b))
        swap(a, b);
    ans = min(ans, st.get_min(tin[a], tin[b]));
    return ans;
}
```

Матроиды

Матроиды нужны, чтобы уметь применять некоторые жадники, в которых нужно набрать какое-то множество объектов максимального веса. Например, максимальное вершинно-взвешенное паросочетание или выполнение заказов с ограничениями по времени (см. примеры).

Матроидом называется пара (X, I), где X — множество элементов, называемое **носителем матроида**, а I — некоторое множество подмножеств X, называемое **семейством независимых множеств**. В матроиде должны выполняться следующие свойства:

- Пустое множество является независимым: $\emptyset \in I$

- Любое подмножество независимого множества тоже независимо:

$$A \subset B, B \in I \implies A \in I$$

- Если в независимом множестве A меньше элементов, чем в независимом множестве B, то будет существовать элемент из B, дополняющий A до независимого множества размера $|A| + 1$:

$$A, B \in I, |A| < |B| \implies \exists x \in B \quad A : A \cup \{x\} \in I$$

Матроид называется **взвешенным**, если на нем существует аддитивная весовая функция: $w(A) = \sum w(a_i)$.

Пусть нам нужно найти независимое множество, которое будет включать как можно больше элементов и при этом иметь как можно меньший вес. Утверждается, что можно просто отсортировать элементы по возрастанию весов и пытаться в таком порядке добавлять их в ответ:

```
X.sort()
s = []
for x in X:
    if good(s + [x]):
        s += [x]
```

Здесь под `good` имеется в виду $s \cup x \in I$.

Корректность этого алгоритма для любого матроида доказывает следующая теорема:

Теорема Радо-Эдмондса

Пусть $A \in I$ — множество минимального веса среди всех независимых подмножеств X мощности k. Возьмем $x : A \cup x \in I,\ x \notin A,\ w(x)$ — минимальна. Тогда $A \cup x$ — множество минимального веса среди независимых подмножеств X мощности $k + 1$.

*Доказательство: **

Рассмотрим B — множество минимального веса среди независимых подмножеств X мощности $k + 1$.

Из свойств матроида: $\exists y \in B \quad A : A \cup y \in I$.

Тогда верны два неравенства:

$$\begin{cases} w(A \cup y) = w(A) + w(y) \geq w(B) \implies w(A) \geq w(B) - w(y) \\ w(B \quad y) = w(B) - w(y) \geq w(A) \implies w(A) \leq w(B) - w(y) \end{cases}$$

Величина $w(A)$ с двух сторон ограничивает величину $w(B) - w(y)$. Значит, они равны. Следовательно, $w(A \cup y) = w(A) + w(y) = w(B)$.

Получаем, что если объединить множество A с x — минимальным из таких, что $A \cup x \in I$, — то получим множество минимального веса среди независимых подмножеств X мощности $k + 1$.

Иными словами, если у нас есть оптимальное k-элементарное независимое множество, то мы можем индуктивно построить оптимальное $(k+1)$-элементарное множество, добавив к нему минимальный элемент, оставляющий построенное множество независимым.

Примеры

Для доказательства жадников нужно просто (а иногда не просто) показать, что рассматриваемое множество является матроидом. Первые два критерия доказывать тривиально, третий посложнее.

Минимальный остов

Рассмотрим неориентированный граф $G = (V, E)$. Пусть I — множество лесов графа (ациклических подмножеств E). Тогда $M = (E, I)$ является матроидом:

- Граф без ребер является лесом.
- Если удалить из леса ребра, он останется лесом.
- Пусть есть два леса $|A| \leq |B|$. В A будет $|V|-|A|$ компонент связности, в B будет $|V|-|B|$ компонент связности. Так как в B компонент связности меньше, то будет существовать какое-то ребро x, связывающее две компоненты связности из A. Его и возьмем: $A \cup \{x\}$ тоже будет лесом, так как x только соединило две разные компоненты связности.

Применив к этому матроиду теорему Радо-Эдмондса, мы получаем обоснование алгоритма Крускала для нахождения минимального остова.

Расписания

Пусть у нас есть n заданий, на выполнение каждого требуется 1 час. Награда за выполнение i-го задания не позже d_i-того часа равна w_i. В один час разрешено сделать только одно задание. Цель — максимизировать сумму наград.

Назовём *правильными* те наборы заданий, для которых существует порядок, при котором можно их всех успеть сделать до их дедлайнов. Проверять фиксированый набор можно так: отсортировать по дедлайнам (d_i) и проверить, что $d_i \geq i$ для всех i.

Тогда \$M = \$ (множество всех заданий, множество правильных наборов заданий) является матроидом:

- Пустой набор заданий всегда можно сделать.
- Если у нас стало меньше заданий, то их сделать мы тоже успеем.
- Пусть есть два правильных набора $|A| \leq |B|$. Тогда в B будет существовать задание x с дедлайном позже $|A|$. Все задания A можно сделать не позже $|A|$-го часа, а в $(|A|+1)$-й час будем делать x. Значит, $A \cup x$ — тоже правильный набор.

Значит, можно отсортировать задания по убыванию их стоимости и пытаться в таком порядке добавлять в ответ, проверяя правильность какой-нибудь структурой данных.

Паросочетания

Рассмотрим двудольный граф $G = (L, R, E)$. Пусть I — множество наборов вершин левой доли, которых можно покрыть каким-нибудь паросочетанием. Тогда $M = (L, I)$ является матроидом:

- Любое паросочетание покрывает пустое множество вершин.
- Исходное паросочетание покрывает также и любое подмножество исходных вершин.
- Пусть есть два множества вершин $|A| \leq |B|$. Раскрасим ребра из паросочетания, соответствующего A в красный цвет, B — в синий, а ребра из обоих паросочетаний — в пурпурный. Рассмотрим граф из красных и синих ребер. Любая компонента связности в нём представляет собой либо путь, либо цикл, состоящий из чередующихся красных и синих ребер. В любом цикле будет равное число красных и синих ребер, а так как всего синих ребер больше, то должен существовать путь, начинающийся и оканчивающийся синим ребром. Поменяем в этом пути красный и синий цвета и сделаем пурпурные ребра обратно красными. Теперь в графе из красных ребер на одно ребро больше, а значит к множеству A добавилась какая-то вершина из левой доли, принадлежавшая ранее B.

Пусть у вершин левой доли есть веса, и нам нужно найти максимальное паросочетание минимального веса. Согласно теореме, мы можем отсортировать вершины левой доли по их весам, а затем пытаться добавлять их в паросочетание. Сделать это можно стандартным алгоритмом Куна.

Линейно независимые вектора

(TODO) (Школьники не обязаны знать линал.)

Такие штуки будем называть базисами.

- Ноль есть в любом базисе.
- Подмножество базиса — базис.
- ??? Пусть нельзя выбрать новый вектор. Получается, что все вектора B лежат в A. Значит, размерность B уж точно не больше.