Формула оценки: **0.25 дз + 0.3 контест + 0.15 кр + 0.3 экзамен + Бонус**
Домашние задания: сдавать устно(раз в неделю ассисты устраивают доп пару, запись онлайн) или latex, дедлайн 10-21 день.
Контесты: Длинные(код ревью), короткие(раз в 2 недели), неточные, бонусные(идет к бонусу). Штрафов нет.
Контрольные работы: раз в модуль, тестовые вопросы.
Бонусы: бонусные контесты, ACM, работа на семинаре.
Материалы:

- Кормен
- en.wikipedia
- викиконспекты
- e-maxx
- Корте-Фанен Коммбинаторная оптимизация

Теория вероятности. $(\Omega, 2^\Omega, P)$ - вероятностная пространство.
$A \subset \Omega$, $P(A) = \sum_{w \in A} P(w)$.
Def: A, B - события, $P(B) > 0$. **P(A|B)** - вероятность события A, если наступило событие B. Тогда $P(A|B) = \frac{\sum_{w \in A \cap B} P(w)}{\sum_{w \in B} P(w)} = \frac{P(A \cap B)}{P(B)}$.
Def: A и B независимые, если P(A|B) = P(A).
Тогда, если A и B независимые, то $P(A \cap B) = P(A) \cdot P(B)$.

$\xi : \Omega \to \mathbb{R}$ - случайная величина. $\xi(w)$ - значение, $w \in \Omega$.
Пример: Есть 5 марок автомобиля, их стоимости и их количества. A - 1000 - 100; B - 2000 - 5; C - 3000 - 5; D - 2000 - 20; E - 1500 - 30; Тогда нас интересуют $P(\xi = 1000) = \frac{100}{160}, P(\xi = 1500) = \frac{30}{160}, P(\xi = 2000) = \frac{25}{160}, P(\xi = 3000) = \frac{5}{160}$.

Матожидание $E(\xi) = \sum_{w \in \Omega} \xi(w) \cdot P(w) = \sum_x x \cdot P(\xi = x)$.

Индикаторная случайная величина: $I_A = \begin{cases} 1, w \in A \\ 0, w \notin A \end{cases}$ Тогда $E(I_A) = P(A)$.

Пусть есть 2 случайной величины ξ_1 и ξ_2. Тогда $E(\alpha\xi_1 + \beta\xi_2) = \alpha E(\xi_1) + \beta E(\xi_2)$.
$E(\alpha\xi_1 + \beta\xi_2) = \sum_{w \in \Omega}(\alpha\xi_1(w) \cdot P(w) + \beta\xi_2(w) \cdot P(w)) = \alpha \sum_{w \in \Omega} \xi_1(w) \cdot P(w) + \beta \sum_{w \in \Omega} \xi_2(w) \cdot P(w) = \alpha E(\xi_1) + \beta E(\xi_2)$

Две случайные величины называются независимые, если $\forall x, y : P(\xi_1 = x \text{ и } \xi_2 = y) = P(\xi_1 = x) \cdot P(\xi_2 = y)$.
n случайных величин называются попарно независимыми, если любые 2 величины независимы.
независимы в совокупности - см семинар

ξ_1 и ξ_2 - случайные независимые величины. Тогда $E(\xi_1\xi_2) = E(\xi_1)E(\xi_2)$.
$E(\xi_1\xi_2) = \sum_{w \in \Omega} \xi_1(w)\xi_2(w)P(w) = \sum_x x \cdot P(\xi_1\xi_2 = x) = \sum_{(u,v)} uv \cdot P(\xi_1 = u \text{ и } \xi_2 = v) = [\xi_1 \text{ и } \xi_2 \text{ независимы}] = \sum_{(u,v)} uv \cdot P(\xi_1 = u) \cdot P(\xi_2 = v) = (\sum_u u \cdot P(\xi_1 = u)) \cdot (\sum_v v \cdot P(\xi_2 = v)) = E(\xi_1)E(\xi_2)$.

Задача о назначениях. Есть n работников и n работ. Есть таблица, где a_{ij} - сколько i-ый работник берет за j-ую работу. Нужно распределить работников по работам так, чтобы суммарная плата за все работы была миниальна. Оценим матожидание затрат при случайном решении. A_{ij} - событие, когда i-ый работник делает j-ую работу. $\xi = \sum_{(i,j)} I_{A_{ij}} \cdot a_{ij}$. Тогда $E(\xi) = \sum_{(i,j)} E(I_{A_{ij}} = \sum_{(i,j)} a_{ij} P(Aij) = \sum_{(i,j)} a_{ij} \cdot \frac{1}{n}$.

Найти максимальный разрез в неориентированном невзвешанном графе.

Будем строить случайный разрез(каждую вершину либо в A, либо в $\bar{A}$). Тогда ξ - величина нашего разреза. $\xi = \sum_{e \in E(G)} I_{B_e}$, где B_e - событие, когда e лежит в разрезе. $P(e \in \text{разрез}) = \frac{1}{2}$. Тогда $E(\xi) = E(\sum_{e \in E(G)} I_{B_E}) = \sum E(I) = \frac{1}{2}|E(G)|$.

Есть перестановка $p_1 \dots p_n$. Алгоритм жадно набирает возрастающую подпоследовательность. Какое матожидание длины этой подпоследовательности?
Событие A_i - алгоритм возьмет p_i. $E(\xi) = E(\sum I_{A_i}) = \sum P(A_i)$. $P(A_i) = P(\forall j < i : p_j < p_i) = \frac{1}{i}$. Тогда $E(\xi) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(logn)$.

Дисперсия $D(\xi) = \sum_{w \in \Omega} P(w)(\xi(w) - E(\xi))^2$.
Свойства:

- $D(\xi_1 + \xi_2) = D(\xi_1) + D(\xi_2)$, ξ_1 и ξ_2 независимы
- $D(\lambda\xi_1) = \lambda^2 D(\xi_1)$

Неравенство Маркова. $\xi : \Omega \to \mathbb{R}_+$. $P(\xi(w) \geq E(\xi) \cdot k) \leq \frac{1}{k}$.

Неравенство Чебышева. $\xi : \Omega \to \mathbb{R}$. $P(|\xi - E(\xi)| \leq \alpha) \leq \frac{D(\xi)}{\alpha^2}$.

Модели
RAM-модель (Random Access Machine) Вопросы, возникающие при создании модели

1. адресация
2. какие инструкции
3. рекурсия
4. где лежат инструкции
5. размер данных
6. кол-во памяти
7. случайность

Адресация Есть ячейки, в которых можно хранить целые числа (ограничения на $MAXC$ разумные, и на них введена неявная адресация

Замечание. Явная адресация — при создании элемента получаем адрес и можем пользоваться только этим адресом. Неявно — можем получать адреса каким-то своим образом, к примеру, $ptr + 20$.

Кол-во памяти Неявное соглашение RAM — время работы не меньше памяти. По дефолту считаем, что мы его инициализируем мусором

Где инструкции Хранить инструкции можно в памяти и где-то снаружи. Мы будем хранить снаружи (внутри — RASP-модель). Иначе говоря, инструкции и данные отделены.

Какие инструкции В нашей модели есть инструкции следующих типов:

- работа с памятью
- ветвление
- передача управления ($=goto$),
- арифметика (at least $a + b, a - b, \frac{a}{b}, \cdot, mod, \lfloor \frac{a}{b} \rfloor$)
- сревнения (at least $a < b, a > b, a \leq b, a \geq b, a = b, a \neq b$)
- логические (at least $\wedge, \vee, \oplus, \neg$)
- битовые операции ($>>, <<, \&, |, \sim, \oplus$)
- математические функции (опять-таки, в рамках разумного)
- rand

Все инструкции работают от конечного разумного числа операндов (не умеем в векторные операции)

Размер данных $\exists\, C, k \ : \ C \cdot A^k \cdot n^k$ — верхнее ограничение на величины промежуточных вычислений.

Рекурсия Рекурсия всегда линейна по памяити относительно глубины.

Случайность Мы считаем, что у нас есть абсолютно рандомная функция. Будем полагать, что у нас есть источник энтропии, выдающий случайности в промежутке $[0, 1]$.

Время работы.

- наихудшее — $t = \max\limits_{input, random} t(input, random)$
- наилучшее — $t = \max\limits_{input} \min\limits_{random} t(input, random)$
- ожидаемое — $E\ t = \max\limits_{input} Average_{random} t(input, random)$
- на случайных данных — $t = Average_{input} Average_{random} t(input, random)$

Алгоритмы

Методы доказательства корректности алгоритма.

1. индукция
2. инвариант
3. от противного

Способы оценки времени работы:

- Прямой учет
- Рекурсивная оценка
- Амортизационный анализ

Прямой учет Время работы строчки — произведение верхних оценок по всем строчкам-предкам нашей.

К примеру

```
while (!is_sorted()) { // O(# inversions) = O(n^2)
    for (int i = 0; i + 1 < n; i++) { // O(n) * O(parent) = O(n^3)
        if (a[i] > a[i + 1]) {
            swap(a[i], a[i + 1]); // O(n^3)
        }
    }
}
```

Рекурсивная оценка Пример — сортировка слиянием

Нас интересует две вещи: инвариант и переход. Для оценки времени используем рекурренту вида $T(n) = O(f(n)) + \sum\limits_{n' \in calls} T(n')$

При этом если мы доказываем время работы, то показываем $T(n) \leq c \cdot f(n)$, зная, что для n' $\exists c$: $T(n') \leq c \cdot f(n)$

Важно, что c глобальное и не должно увеличиваться в ходе доказательства

stable sort Делает сортировку, не меняя порядок равных элементов относительно исходной последовательности. Merge-sort стабилен.

inplace-algorithm Не требует дополнительной памяти и делает все прямо на данной памяти (у нас есть $\log n$ памяти на рекурсию). Quick-sort inplace.

Время работы qsort

$$T(n) = \max_{input} average_{rand} t(input, rand) = \max_{|input|=n} Et(input)$$

$$T(n) \leq \Theta(n) + \frac{1}{n}\sum_{k=0}^{n-1}(T(k+1) + T(n-k)) \leq \Theta(n) + \frac{2}{n} \cdot \sum_{k=1}^{n} T(k-1) \leq a \cdot n + \frac{2}{n}\sum_{k=1}^{n} c \cdot (k-1) \cdot \log(k-1) \leq$$

$$\leq a \cdot n + \frac{2}{n}\sum_{k=1}^{\frac{n}{2}} c \cdot (k-1) \cdot \log n - \frac{2}{n}\sum_{k=1}^{\frac{n}{2}} c \cdot (k-1) + \frac{2}{n}\sum_{k=\frac{n}{2}+1}^{n} c \cdot (k-1) \cdot \log n \leq$$

$$\leq a \cdot n + \frac{2}{n} \cdot n^2 \cdot \log n - \frac{2c}{n} \cdot \frac{(n-2)^2}{4} \leq a \cdot n + cn \log n - \frac{c(n-2)}{4} \leq cn \log n$$

$$\frac{c(n-2)}{4} \geq a \cdot n$$

$$c \cdot n - 2c \geq 4 \cdot a \cdot n$$

$$c \geq \frac{4 \cdot a \cdot n}{(n-2)}$$

, что верно для достаточно больших n.

Ограничение на число сравнений в сортировке Бинарные сравнения на меньше.

Рассмотрим дерево переходов. Для перестановки есть хотя бы один лист — листьев котя бы $n!$

$$L(T) \leq 2^x, d(T) \leq x$$

, если x — ответ

$$L(T) \geq n!$$

$$d(T) \geq \log n! \geq \log \frac{n}{2}^{\frac{n}{2}} = \log 2^{(\log \frac{n}{2}) \cdot \frac{n}{2}} = \frac{n}{2} \cdot \log \frac{n}{2} = \Omega(n \log n)$$

1 Сортировки основанные на внутреннем виде данных

Имеем n чисел $[0, U-1]$, $U = 2^w$, числа укладываются в RAM-модель

Сортировка подсчетом Заводим массив $cnt[U]$, $cnt[x] = |\{i : a_i = x\}|$. Дальше переводим $count \to pref$, $pref[x] = pref[x-1] + count[x]$

$O(n+U)$

Поразрядная сортировка b_{ij}—j-й бит i-го числа. (Сортируем бинарные строки длины w)

Поочередно сортируем строки, разбивая их на классы эквивалентности по $iter$ последним символам. После чего мы стабильно сортируем по $(iter+1)$-му символу.

$O(n \log U)$

Bucket sort Разбиваем множество на корзины, каждой корзине соответствует отрезок. В каждой корзине запускаемся рекурсивно.

$O(n \log U)$

В продакшне используют первую пару итераций, чтобы сильно снизить размерность на реальных данных.

Пусть мы хотим отсортить равновероятные числа из $[0, 1]$. В каждом бакете отсортируем за квадрат. Получим $O(n)$.

$$t(n) \le \sum_{i=1}^{n} c \cdot (1 + E(cnt_i)^2) \le c \cdot n + c \cdot \sum_{i=1}^{n} E(cnt_i^2) =$$

$$= c \cdot n + c \cdot \sum_{i=1}^{n} \sum_{j=1}^{n} EI_{A_{ij}} = c \cdot n + c \cdot n^2 \cdot \frac{1}{n} \le 2 \cdot c \cdot n$$

2 Иерархия памяти

Нас интересует задержка (latency), пропускная способность (throughput). Подгрузка x данных занимает $l + \frac{x}{t}$

От долгой к быстрой

1. external machine / internet
2. HDD
3. SSD
4. RAM
5. L3
6. L2
7. L1
8. registers

3 Алгоритмы во внешней памяти

n — размер задачи
M — размер RAM
B — блок данных
$B \ll M \ll n$
$\log n \ll B$
$B < \sqrt{M}$ (но это неявно и не факт)

Mergesort во внешней памяти Обычный mergesort, но три типа событий в $merge$:

1. Кончился первый буфер — подгружаем новый
2. Кончился второй — аналогично
3. Кончился буфер для слияния — выписываем обратно в RAM и сбрасываем

$O(\frac{n}{B} \log n) \to O(\frac{n}{B} \log \frac{n}{B}) \to O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$
+2 идеи:

1. Дошли до размера M — явно посортим в RAM
2. Можем сливать сразу $\frac{M}{B}$ массивов

1 Простые структуры данных

Требования к структуре данных:

- От СД мы хотим обработку каких-то наших запросов.
- online-offline. Бывает, что мы знаем все запросы, бывает, что мы узнаем запрос только тогда, когда отвечаем на предыдущий
- При обсуждении времени работы отделяется время на препроцессинг и на последующие ответы на запросы (query).

1.1 Data structure / interface

Структура данных — это какой-то математический объект, который умеет отвечать на наши запросы конкретным способом. Красно-черное дерево — это структура данных.
Интерфейс — это объект, с которым может взаимодействовать пользователь, который каким-то образом умеет отвечать на наши запросы (пользователю все равно, как, его волнует только то, что интерфейс реализует, и за какое время(память) он это делает). $std :: set$ — это интерфейс.
Итератор — это специальный объект, отвечающий непосредственно за ячейку в структуре данных. Для того, чтобы удалить элемент, мы должны иметь итератор на этот элемент. Т.е. удаление по ключу работает за $O(erase)$, а удаление по значению за $O(find) + O(erase)$

list Списки бывают двусвязными, односвязными, циклическими. У каждого элемента есть ссылка на следующий (и иногда на предыдущий), а также есть отдельный глобальный указатель на начало списка.

stack Стек — структура данных, которая умеет делать добавление в конец, удаление из конца, взятие последнего элемента, за $O(1)$.

queue Очередь — структура данных, которая умеет делать добавление в конец, удаление из начала, взятие первого элемента, за $O(1)$.

deque Двусторонняя очередь — структура данных, которая умеет делать добавление, удаление и взятие элемента с любого конца последовательности, за $O(1)$.

priority_queue Очередь с приоритетами ака куча — структура данных, которая умеет делать добавление, удаление, и быстрые операции с минимумом, представляющая из себя дерево с условием $parent(u) = v \to value(v) \leq value(u)$.

Все эти структуры реализуются как на массиве (храним последовательную память и указатели на начало/конец), так и на списках (что на самом деле является тем же самым, что и на массиве, просто ссылка вперед эквивалентна $a_i \to a_{i+1}$ в терминологии массивов, *прим. автора*).

Структура данных на массиве кратно быстрее аналогичной на ссылках, потому что массив проходится по кэшу и не требует дополнительной памяти.

data structure	add	delete	pop	find	top	build	min	get by index
stack	$O(1)$	-	$O(1)$	-	$O(1)$	$O(n)$	$O(n)$	-
dynamic array	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$
queue	$O(1)$	-	$O(1)$	-	$O(1)$	$O(n)$	$O(n)$	-
deque	$O(1)$	-	$O(1)$	-	$O(1)$	$O(n)$	$O(n)$	$O(1)$
linked list	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
sorted array	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(n \log n)$	$O(1)$	$O(1)$
priority_queue	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(n)$	$O(1)$	-	

1.2 Двоичная куча

Реализация двоичной кучи на массиве — создаем массив размера sz, и создаем ребра $i \to 2 \cdot i,\ i \to 2 \cdot i + 1$.

От такой кучи мы хотим:

- insert(x)
- get_min()
- extract_min()
- erase
- change = {decrease_key, increase_key}

Для такой кучи мы реализуем $sift\_up(x),\ sift\_down(x)$ — просеивание вниз и вверх. Процедура должна устранить конфликты с элементом x. Остальные операции умеют реализовываться через нее.

$$insert = add\_leaf + sift_up$$

$$extract\_min = swap(root, last) + last- = 1 + sift_down(root)$$

$$erase = decrease\_key(-\infty) + extract\_min$$

Отдельно отметим построение кучи за $O(n)$ — sift_down поочередно для всех элементов $n,\ n-1,\ \ldots,\ 1$.

```
void sift_up(int v) { // v >> 1 <=> v / 2
    if (key[v] < key[v >> 1]) {
        swap(key[v], key[v >> 1]);
        sift_up(v >> 1);
    }
}

void sift_down(int v, int size) { // indexes [1, size]
    if (2 * v > size) {
        return;
    }
    int left = 2 * v;
    int right = 2 * v + 1;
    int argmin = v;
```

```
    if (key[v] > key[left]) {
        argmin = left;
    }
    if (right <= size && key[right] < key[argmin]) {
        argmin = right;
    }
    if (argmin == v) {
        return;
    }
    swap(key[v], key[argmin]);
    sift_down(argmin, size);
}
```

Какое-то сегодняшнее дополнение про бинарную кучу есть в прошлом конспекте

К-чная куча Куча на полном К-чном дереве. Эту кучу можно так же хранить в массиве, с 0-индексацией.

- $sift\_up$ такой же, $O(\log_k n)$
- $sift\_down$ ищет минимум среди k элементов на каждом шаге, поэтому работает за $O(k \cdot \log_k n)$.

	$2-heap$	$k-heap$
$insert$	$O(\log n)$	$O(\log_k n)$
$extract\_min$	$O(\log n)$	$O(k \log_k n)$
$decreasekey$	$O(\log n)$	$O(\log_k n)$
$increasekey$	$O(\log n)$	$O(k \log_k n)$

Дейкстра на K-ной куче Алгоритм Дейкстры достает минимум n раз, и улучшает ключ m раз

$$a \cdot \log_k n = b \cdot k \cdot \log_k n, a = m, b = n$$

Отсюда $k = \frac{a}{b} = \frac{m}{n}$ в случае Дейкстры. Еще отметим, что $k \geq 2$.

В случае когда $a = b^q,\ q > 1$, то k-куча структура работает за $O(1)$ (вроде бы этот факт мы докажем в домашке), причем с хорошей константой, поэтому применимо на практике (привет, фибоначчиева куча!).

Амортизационный анализ Идея в том, что мы хотим оценить суммарное число операций, а не на каждом шаге работы. То есть вполне может быть итерация алгоритма за $O(n)$, но нам важно, что суммарное число $O(n \log n)$

Так что есть $t_{real} = t,\ t_{amortized} = \tilde{t}$.

Метод кредитов Элементам структуры сопоставляем сколько-то монет. Этими монетами элемент «расплачивается» за операции. Также мы накидываем сколько-то монет на операцию. Запрещаем отрицательное число монет. Начинаем с нулем везде.

Обозначим состояния структуры за $S_0, S_1, \ldots, S_n$. Каждый переход стоил t_i, $t_i \geq |operations|$, Где t_i — это сколько мы потратили. Также на i-м шаге мы вбрасываем в систему $\tilde{t}_i$ монет. Тогда

$$\sum t_i \leq \sum \tilde{t}_i \leq A \to O(A)$$

Стек с минимумом

- min_stack
- push
- pop
- get_min

$m_i = \min(m_{i-1}, a_i)$ — поддерживаем минимумы. Операции тривиальны

Очередь с минимумом на двух стеках Храним два стека с минимумом, один из которых мысленно наращиваем в одну сторону, а другой в другую, при этом очередь выглядит как бы как склеенные стеки. То есть мы добавляем элемент в первый стек, а извлекать хотим из второго.

$$X \to a_n, a_{n-1}, \ldots, a_1, \ | \ , b_1, b_2, \ldots, b_n \to Y$$

Тогда единственная сложная операция — если мы хотим извлечь минимум, а второй стек пустой. Тогда мы все элементы из первого перекинем во второй по очереди с помощью «извлеки-добавь»

Почему это работает за $O(1)$ на операцию амортизированно? Представим каждому элементу при рождении 2 монеты, одну из которых мы потратим на добавление в первый стек, а вторую на удаление через второй.

set для бедных Хотим не делать $erase$, только $insert$, $find$, get_min. Храним $\log n$ массивов, $|a_i| = 2^i$, каждый из которых по инварианту будет отсортирован. Тогда get_min рабтает за $O(\log n)$ — просто берем минимум по всем массивам. Аналогично $find$ делается бинпоисками за $O(\log^2 n)$

А как добавлять за $\tilde{O}(\log n)$? Каждый элемент при добавлении в структуру получает $\log n$ монет. Когда мы добавляем элемент, мы создаем новый массив ранга 0. Если было два массива ранга 0, сольем их в новый массив ранга 1 за суммарный размер (и заберем монетку у всех элементов во время слияния), и так далее, пока не создадим уникальный массив для текущего ранга.

Метод потенциалов $\Phi(S_i)$ — потенциал, который зависит только от состояния структуры (**не от** последовательности действий, которая к такому состоянию привела).

Опять вводим t_i, $\sum t_i = O(f(n))$. Определим амортизированное время работы:

$$\tilde{t_i} = t_i + (\Phi(S_{i+1}) - \Phi(S_i))$$

$$t_i = \tilde{t_i} + \Phi(S_i) - \Phi(S_{i+1})$$

Пусть мы показали $\tilde{t_i} \le f(n)$. Тогда

$$\sum t_i \le n \cdot f(n) + \Phi(0) - \Phi(n)$$

Нормальный потенциал — такой, что из неравенства выше все еще можно показать O-оценку на $\sum t_i = O(n \cdot f(n))$.

deque для богатых Хотим deque с поддержкой минимума.

Храним два стека как для обычной очереди. Все операции хорошо работают как на очереди, кроме перестройки структуры. В случае с очередью надо было переливать стеки только в одну сторону, а теперь иногда нужно туда-сюда.

Теперь мы будем перекидывать только половину элементов. Тогда нам понадобитсяя 3 стека, один из которых будет вспомогательным для перестройки (там иначе стеки развернуты).

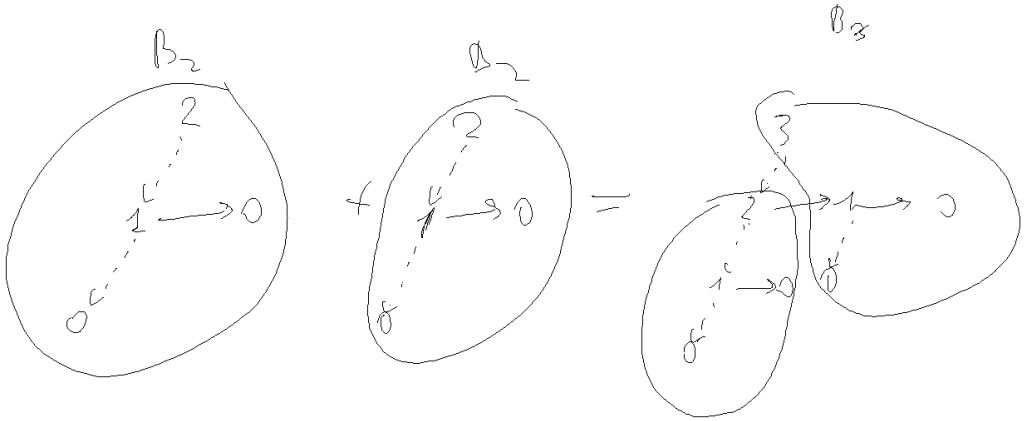
$\Phi(S_i) = |Size_1 - Size_2|$

Куча: insert, extract_min, decrease_min, decrease_key, increase_key, merge. Хотим добавить операцию merge - объединить 2 кучи. Считаем, что можем делать операции амортизировано (не разделяем кучи и кучи не персистентные). Меньшую кучу будем добавлять в большую ("переливать"в большую).

Хотелось бы раздать каждой вершине по $\log n$ монет и говорить, что, когда мы "переливаем"кучу, все элементы этой кучи платят по монете. Тогда монет хватит, так как при каждом переливании размер кучи, где находится вершина увеличивается вдвое, значит каждая вершина перельется не более $\log n$ раз. Но все ломается из-за того, что мы можем удалять вершины. Можно ввести потенциалы (подумать), а можно сказать, что у каждого элемента есть ранг (r(i)) и при каждом переливании все ранги меньшей кучи увеличиваются на 1. Тогда амортизировано merge будет работать за $O(\log^2 n)$.

Биномиальная куча:

Вершин в биномиальной кучи 2^n, на k-ом слое C_n^k вершин. Из каждой вершины храним ребро в старшего сына, в предка и в следующего брата. $B_k = \text{merge}(B_{k-1}, B_{k-1})$. Заметим, что merge деревьев одного ранга работает за O(1).



Если у нас есть n чисел, то как мы их поместим в кучу размера 2^k? $n = 2^{k_1} + 2^{k_2} \cdots + 2^{k_m}$. Тогда можем хранить все элементы как кучи рангов $k_1, \ldots k_m$. Тогда при merge нужно объединять два списка биномиальных куч (будем делать это 2 указателями по 2 массивам), будем объединять кучи одинаковых рангов

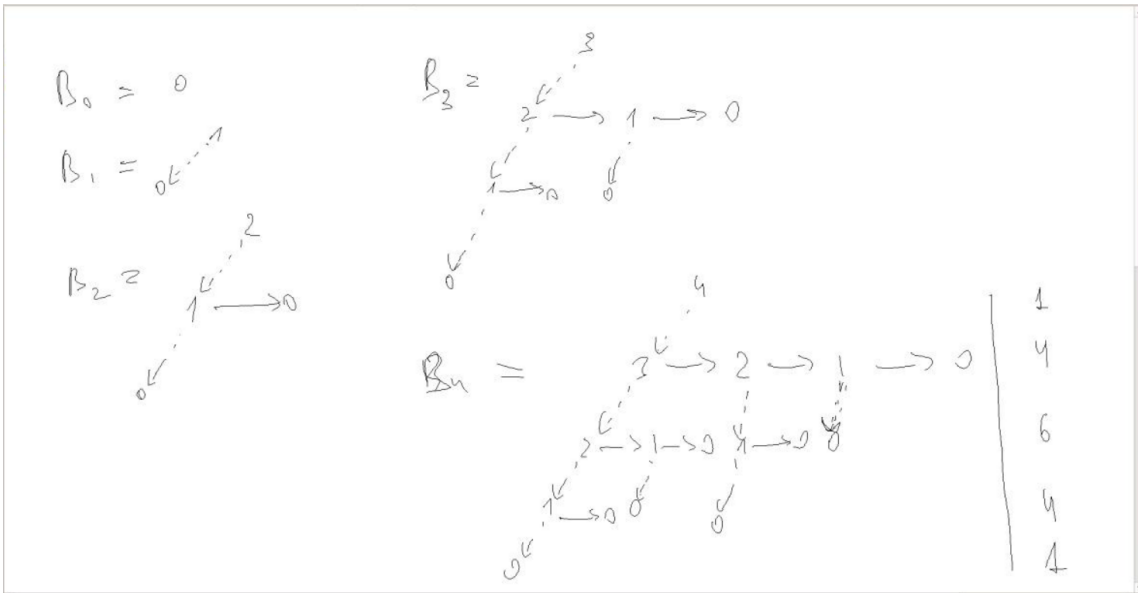
insert - создаем кучу ранга 0 с нашим элементом и делаем merge за $O(\log n)$.

Для поиска минимума будем просто поддерживать глобальный минимум, изменяя его за $O(\log n)$ (пробегаясь по всем кучам) при каждом запросе изменения.

decrease_key

extract_min

increase_key: decrease_key(v, $-\infty$) $\to$ extract_min $\to$ insert(x). Работает за $O(\log n)$.



Фибоначчиева куча: (чет я устал техать, чекайте у Кости)

Операции	binary heap	binomial heap	fibonacci heap
insert	$O(\log n)$	$O(\log n)$	$O(1)$
extract_min	$O(\log n)$	$O(\log n)$	$\tilde{O}(\log n)$
decrease_min	$O(\log n)$	$O(\log n)$	$\tilde{O}(1)$
increase_min	$O(\log n)$	$O(\log n)$	$\tilde{O}(\log n)$
merge	$\tilde{O}(\log^2 n)$	$O(\log n)$	$O(1)$
get_min	$O(1)$	$O(1)$	$O(1)$

Больше куч!

	binaryheap	*binomialheap*	*fibonacciheap*
insert	$O(\log n)$	$O(\log n)$	$O(1)$
extract_min	$O(\log n)$	$O(\log n)$	$\tilde{O}(\log n)$
decrease_key	$O(\log n)$	$O(\log n)$	$\tilde{O}(1)$
increase_key	$O(\log n)$	$O(\log n)$	$\tilde{O}(\log n)$
merge	$\tilde{O}(\log^2 n)$	$O(\log n)$	$O(1)$
get_min	$O(\log n)$ *or* $O(1)$	$O(1)$	

Биномиальная куча

Храним биномиальные деревья. Каждому дереву сопоставим ранг. Ранг дерева полностью определяет его структуру. Дерево ранга 0 — одна вершина. Дерево ранга 1 — одно ребро. В общем случае, дерево ранга n содержит корень и полное двоичное дерево размера 2^{n-1}. Дерево ранга $n+1$ — это два слитых вместе дерева ранга n — корень второго указывает на корень первого, а корень первого теперь будет указывать на оба бинарных дерева.

Биномиальная куча — это набор из логарифма биномиальных куч.

Слияние двух деревьев мы научились делать за $O(1)$ — меньшая вершина по ключу становится новым корнем, а дальше перекидываем указатели.

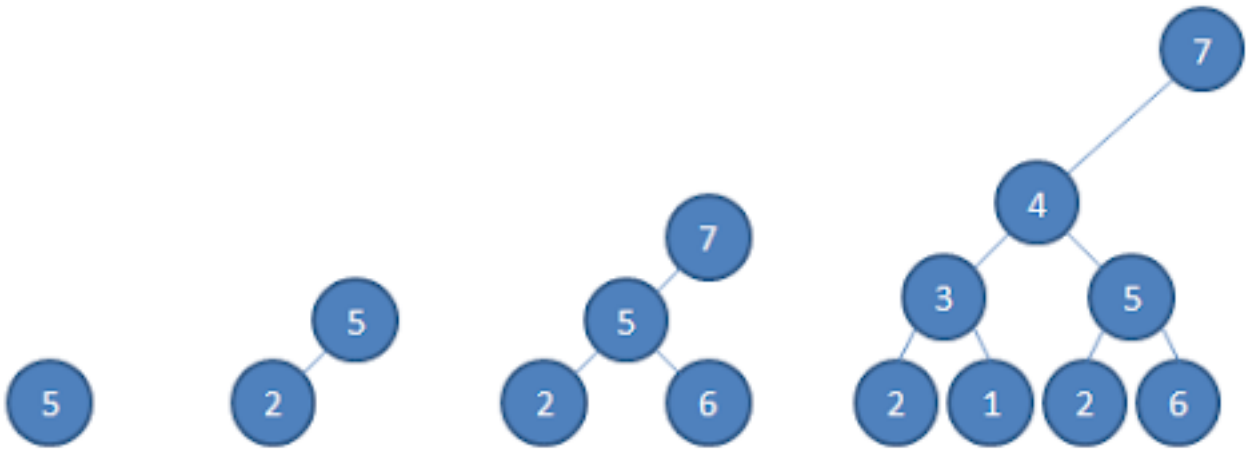
Слияние двух куч — это алгоритм сложения двоичных чисел — при сливании двух деревьев ранга n мы «переносим» прибавление кучи ранга $n+1$

Как добавлять элемент? Создать кучу на 0 элементов и слить их вместе.

Уменьшение ключа — напишем $sift\_up$ на нашей новой куче

Удаление минимума — Заметим, что если в правом дереве пройти по правым детям и обозначим их за корни, а их левых детей за полные бинарные деревья, то мы получим набор деревьев рангов $0, 1, \ldots, n-1$. Обозначим их за новую кучу, и сольем все вместе.

Увеличение ключа — удалим соответствующий элемент и добавим другой.



Фибоначчиева куча

Хотим сделать биномиальную кучу с послаблаблениями — делать операции в самый последний момент, менее четкую структуру, etc

Есть деревья, их корни храним в двусвязном закольцованном списке.

Всех детей для всех вершин храним в двусвязном закольцованном списке.

Новый ранг — это количество вершин в списке детей.

На каждом дереве выполнена куча, а также поддерживаем глобальный минимум.

Улучшение ключа делается так — удаляем вершину из своего списка, вместе с поддеревом. Добавляем в корневой список. Если мы удалили уже вторую вершину в поддереве родителя, то делаем каскадное вырезание — прыгаем по предкам с $mark = 1$, и вырезаем их в корневой список, причем все вырезания делаются по очереди.

Удаление делается так — мы приписываем всех детей к корневому списку, а потом вызываем *compact*, которая должна спасти наше дерево и навести порядок.

Псевдокод тупых операций:

```
struct Node{
    Node *child;
    Node *left;
    Node *right;
    Node *parent;
    int rank;
    bool mark;
    int value;
}

list<Node *> roots;

void insert(int x) {
    Node *node = new Node(x);
    roots.insert(node);
}

void merge(list<Node *> a, list<Node *> b) {
    merge(a, b); // O(1) haha super easy
}

int getmin() {
    return argmin->value;
}
```

compact

- Сбрасываем пометки корневого списка в 0
- Переводим дерево в состояние, где все ранги разлиичны
- Храним ранги, мерджим одинаковые
- Как мерджим? Берем меньший корень, и записываем в его детей второй корень

Обозначим $R = \max rank$, $t(H) = root\ list\ size$, $m(H) = \sum_v mark(v)$
compact работает за $O(R + t(H))$.

Анализ времени работы $extract\_min$ &$increase\_key$ — $\tilde{O}(R)$.
$\Phi(H) = t(H) + 2m(H) < 3n$
Пусть каскадное вырезание сделало t_i действий. $m: \ 1 \to 0, \ t: \ +1$. Тогда $\Phi'(H) = \Phi(H) - 1$ за каждое вырезание. Тогда амортизированно вырезание работает за $O(1)$.
Compact:
$t(H) \le R$, $t'(H) = t(H) - R$
Амортизациованно работает за $2R + 1$
Хотим показать $R = O(\log n)$.

$$\forall\ v \in H\ sz(v) \ge A^{rank(v)},\ A > 1$$

Тогда

$$r(v) \le \log_A s(v) \le c \log n$$

Возьмем

$$s(v) \ge \phi^{rank(v)-2}$$

<Оффтоп про числа Фибоначчи>

1. $F_n = F_{n-1} + F_{n-2},\ F_0 = F_1 = 1$
2. $F_n \ge \phi^{n-2}$
3. $\forall\ i \ge 2: \ F_i = \sum_{j=0}^{i-2} F_j + 1$

</Оффтоп про числа Фибоначчи>

Почему инвариант на размеры сохраняется? Возьмем вершину v с k детьми. Рассмотрим детей в том порядке, в котором их склеивал компакт. Тогда на момент добавления i-й вершины в структуру, ее ранг совпадал с рангом v. Тогда из условия на удаление не более чем одного сына ($mark$) следует, что $rank(i) \ge i - 1$. Тогда мы доказываем по индукции, что у нас все размеры — хотя бы числа фибоначчи, соответствующие рангу. Тогда $s(v) = \sum_{u \in g(v)} s(u) + 1 \ge \sum_{u \in g(v)} F_rank(u) + 1 \ge \sum_{i=0}^{rank(v)-2} F_i + 1 \ge F_{rank(v)-2}$

Хэширование Есть задача сравнения объектов:

$$U = \{objects\},\ u, v \in U :\ u \neq v?$$

Введем функцию $h :\ U \to \mathbb{Z}_m$, такую что $\forall\ u, v \in U :\ u \sim v \to h(v) = h(u)$. Обычно $m \ll |U|$.
Зачем юзать хэши, а не наивное сравнение?

- Бывает много сравнений, и мы не хотим дублировать вычисления
- Безопасность
- Для некоторых хешей верно, что $h(f(v, u)) = g(h(v), h(u))$. То есть можно вычислить хэш от некоего объекта, пользуясь уже посчитанными хэшами для других объектов.
- Иногда мы сравниваем объекты не на равенство, а на изоморфность.
- Сравнивать объекты бывает дорого

Требования к хэшу:

- вычислим за линейное время
- детерменирован
- семейство хэш-функций $\mathbb{H}$ (и можем взять сколько угодно оттуда)
- равномерность $\forall_{v \neq u} v,\ u :\ p_{h \in \mathbb{H}}(h(u) = h(v)) \simeq \frac{1}{m}$
- масштабируемость
- необратимость
- (*optional*) лавинный эффект (при маленьком изменении объекта хэш меняется сильно)

Важно, что мы хотим брать случайную функцию из семейства $\mathbb{H}$ на момент старта программы, потому что псевдослучайная функция на самом деле нам дает детерменированный алгоритм, который неверен.

Идеальная хэш-функция Так как объектов счетно, то отсортируем их, далее для каждого объекта запомним случайную величину от 1 до m (или даже можем запоминать ее лениво!).

Полиномиальный хэш Сводим объекты к строкам и хэшируем строки:

$$s = c_0 c_2 c_3 \dots c_{n-1},\ c_i > 0$$

$$h_b(s) = \sum_{i=0}^{n-1} c_i \cdot b^i\ (mod\ m)$$

$$p_b(h_b(s_1) = h_b(s_2)) \leq \frac{n}{m} \text{ для фиксированного m}$$

Доказательство: Рассмотрим функцию как многочлен, теперь для равных функций смотрим на то, равна ли разность многочленов нулю для какого-то b. У многочлена от b степень равна n, а вероятность попасть в конкретный модуль $\frac{1}{m}$.

$$h(s_1 + s_2) = h(s_1) + h(s_2) \cdot b^{|s_1|}$$

Хэш-таблица

Key-value storage: три типа операций — $set(x, y)$, $get(x)$, $has(x)$. Хэш-таблица умеет выполнять такие запросы за $O(1)$.

Хотим делать индексацию не по ключу, а по хэшу от ключа $x_0 \to h(x_0)$.

К сожалению, бывает так, что в одну и ту же ячейку попало много элементов (матожидание числа коллизий порядка $\frac{n^2}{m}$, где m — размер хэш-таблицы). Мы будем называть это коллизиями.

Коллизии можно решать двумя способами, соответствующие хэш-таблицы имеют **открытую** или **закрытую** адресацию.

Закрытая адресация (или решение коллизии методом цепочек) — в каждой ячейке храним список, в который будем добавлять соответствующие элементы. Матожидание длины списка будет порядка $\frac{n}{m}$.

Хэш-таблица работает линейно от числа элементов, которые в ней когда-либо были, поэтому динамическая хэш-таблица работает за амортизированное время.

«stop-the-world»-концепция — если внутренние параметры системы бьют тревогу, сделаем глобальное изменение. В случае хэш-таблицы, если в таблице сейчас m элементов, создадим новую хэш-таблицу удвоенного размера, в которую перехэшируем оставшиеся элементы.

Замечание. Очень часто разработчик не хочет амортизированное время работы, потому что боится внезапного «stop-the-world», потому что это выключает систему на длительное время. Пример — финал TI8.

Открытая адресация

Делаем вид, будто коллизий не бывает (то есть, в каждой ячейке храним только одно значение). Кроме того, рядом с ячейкой храним ключ элемента (или -1, если там пусто). Тогда если мы хотим найти элемент x, мы смотрим в ячейку $h(x)$, и идем от нее вправо до тех пор, пока не встретим x или -1.

Ожидаемое время — это $\frac{1}{1-\alpha}$, где $\alpha = \frac{n}{m}$. На практике все считают, что время — константа.

Удаление с открытой адресацией — нетривиальная задача, потому что удаление элемента рушит цепочки.

Удаление без «stop-the-world» — удаляем все элементы от нашего элемента до ближайшей -1, но потом вернем их обратно с помощью добавлений

Удаление с «stop-the-world» — кроме пометки -1 делаем пометку «зарезервирована». Тогда при удалении элемента мы ставим в его ячейку пометку «зарезервирована». Тогда при линейном проходе мы делаем вид, что резерв — это настоящий элемент, а при добавлении мы можем записать в резерв. Резерв включается в параметр α, поэтому нам понадобится делать перестройки.

Кроме линейного сканирования можно делать что-то типа «прыжкам по хешу» $(h(x) + jh'(x))$, но на практике линейное сканирование — наш бро.

Совершенное хэширование

Нам изначально дано сколько-то ключей, мы хотим делать $get(x)$, $set(x, y)$ за гарантированные $O(1)$ с ожидаемым $O(n)$ на преподсчет.

Сделаем хэш-таблицу с закрытой адресацией. В каждой ячейке ожидаемое $O(1)$ элементов. Сделаем новую хэш-функцию, которая переносит все элементы из l_i в ячейки размера l_i^2. Вероятность коллизии при таком хэшировании меньше $\frac{1}{2}$, поэтому мы будем просто рандомить хэш-функцию, пока не получим отсутствие коллизий, что займет у нас ожидаемое $O(1)$.

Фильтр Блума

insert, *get*

Запросы *get* работают необычным образом — No → No; Yes → Yes$(1-p)$/No(p). То есть, если структура говорит, что элемент в ней лежит, то он в ней точно не лежит. Дальше минимизируется p.

Фильтр Блума является массивом из битов длины m. Фильтр выбирает k хэш-функций, элементу сопоставляется k значений хэша.

insert — в каждое из k значений ставим 1

get — проверяем, что во всех k значениях стоит 1.

Время работы. $k \sim \frac{m}{n}$. Рассмотрим вероятность ложноположительного срабатывания. Вероятность того, что клетка свободна — $(\frac{m-1}{m})^{kn}$. Тогда вероятность ложноположительного срабатывания это $(1 - (\frac{m-1}{m})^{kn})^k \sim (1 - e^{-\frac{kn}{m}})^k$.

Путем долгих вычислений получаем $k = \ln 2 \cdot \frac{m}{n}$.

Кроме того, ФБ поддерживает операции пересечения и объединения множеств.

Деревья поиска Храним структуру данных, которая должна уметь делать все то же самое, что можно делать с отсортированным массивом, но еще с добавлением-удалением.

Binaary search tree (BST) Корневое дерево. В вершине храним левого сына, правого сына, предка, ключ.

Обозначения:

- $l(v)$ — левый сын
- $r(v)$ — правый сын
- $p(v)$ — предок
- $key(v)$ — ключ
- $T(v)$ — поддерево
- $S(v) = |T(v)|$ — размер
- $A(v)$ — множество предков
- $seg(v) = [\min_{u \in T(v)} key(u); \max_{u \in T(v)} key(u)]$

Условие BST:

$$\forall u \in T(l(v)) : \ key(u) < key(v)$$

$$\forall u \in T(r(v)) : \ key(u) > key(v)$$

То есть, ключи лежат в порядке лево-правого обхода (в порядке «выписать левое поддерево - выписать вершину - выписать правое поддерево»)

$v \in seg(u) \leftrightarrow v \in T(u)$

Поиск в дереве — надо сделать спуск. То есть, если текущая вершина — не та, которая нам нужна, то можно понять, где лежит нужная нам, с помощью условия BST.

Нахождение следующего — либо спуск вправо, либо подъем по предкам до первого большего.

Основная проблема — операции вставки/удаления, которые должны делать дерево сбалансированным (таким, высота которого нас устраивает, то есть примерно $\log n$)

Декартово дерево У каждой вершины будем хранить не только ключ, но и какой-то приоритет y. Построим дерево так, что по y это куча, а по x это BST.

Тогда декартово дерево задается однозначно, если определить все приоритеты. Почему? Расставим точки на плоскости в соответствии с (x, y), после чего найдем корень. У корня будет наименьший приоритет и поэтому он определяется единственным образом (будем считать, что приоритеты различны). Тогда к корню нужно приписать слева и справа по дереву, которые рекурсивно строятся в левой и правых частях.

Утверждение Если взять случайные y, то матожидание глубины дерева $O(\log n)$

Доказательство: Нет.

Утверждение Если взять случайные y, то матожидание глубины каждой вершины $O(\log n)$

Доказательство. Матожидание высоты вершины — это число вершин, которые являются ее предками. Вершина i будет предком вершины j, если $y_i = \max\{y_i, y_{i+1}, \ldots, y_j\}$. Матожидание суммы таких величин для i это $\sum_{j=0}^{i-1} \frac{1}{i-j} + \sum_{j=i+1}^{n-1} \frac{1}{j-i} \leq 2\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$

2-3 Дерево

Представляет собой структуру данных, которая является сбалансированным деревом поиска, удовлетворяющее двум условиям:

- Все листья будут на одной глубине, значения будут храниться в листьях
- У всех вершин число исходящих ребер $deg_v \in \{0, 2, 3\}$

Также мы в каждом поддереве будем хранить максимум, а детей будем хранить упорядоченными по величине максимума.

Операции на дереве: поиск Спуск будем делать рекурсивно. Поскольку мы хранили максимум, то мы среди детей находим первое число, большее x, спускаемся в соответствующего сына.

Операции на дереве: добавление Если степень предка после добавления стала равна 4, то создадим два сына размеров 2 и 2, и рекурсивно рассмотрим отца. Если дошли до корня, то создадим новый корень степени 2.

Операции на дереве: удаление Рассмотрим ситуации, которые могли возникать при удалении. Заметим, что у вершины есть отец, дедушка, дядя, брат (предок, другой сын прапредка, прапредок, другой сын предка).

Если у вершины было 2 брата, то после ее удаления ничего менять не надо.

Если у вершины был 1 брат, то нарушается инвариант на степени. Посмотрим на детей дяди. Если их было 3, то можно перераспределить $3 + 1$ как $2 + 2$, и инвариант не нарушится. Если же там было 2 ребенка, то склеимся в одну вершину степени 3, и уменьшим степень предка на 1. Рекурсивно запустим процесс балансировки от него.

Оптимальное дерево поиска Обозначим число детей за d. Тогда операции работают за $d \cdot \log_d n$. Найдем точку минимума: $(d \cdot \log_d n)' = \ln n \cdot \frac{\ln d - 1}{\ln^2 d}$. Нулевой корень производной при $d = e$. Таким образом, 2-3 дерево достаточно близко к оптимальному.

$B+$ дерево Зафиксируем константу T. Все значения опять храним в листьях. У вершин (кроме корня) степень от T до $2 \cdot T - 1$. $2 \le deg_{root} \le 2 \cdot T - 1$

Обычно T делают достаточно большим, чтобы дерево работало во внешней памяти.

Операции: вставка Если у вершины степень стала $2T$, то мы можем разделить ее на две вершины, подвесить их к предку, и рекурсивно запустить процесс у предка.

Операции: удаление Плохая ситуация при удалении — степень предка стала равна $T-1$. Посмотрим на соседних братьев. Если один из них по степени больше T, то мы можем позаимствовать у него крайнего сына, чем починим свою степень. Если же у обоих братьев степень T, то мы можем смерджить себя с братом, после чего рекурсивно продолжить процесс в предке, потому что у предка степень уменьшилась на 1.

Почему $B+$, а не B? В начале стоит сказать, что B-дерево обладает схожей структурой, но разрешает хранение значений не только в листьях. Его глубина не более чем на 1 меньше, чем у соответствующего ему $B+$-дерева. Но при этом элементы $B+$-дерева можно поддерживать в двусвязном списке, что удобно, а также можно итерироваться во внешней памяти.

Научимся в детермиированный, так сказать, декартач.

2-3 дерево

В этом дереве есть вершины степени 2 и вершины степени 3. Элементы записываются только в листьях.

Ряд условий 2-3 дерева:

1)Все листья на одной и той же глубине. 2)Степень каждой внутренней вершины либо 2, либо 3.

В промежуточных вершинах храним ключи, которыми можно разделять вершины-сыновей.

У вершины с 3 детьми, два ключа - максимум поддеревьев двух первых детей, с 2 детьми - максимум в первом (а ещё храним максимум во всём поддереве).

Смотря на ключ, мы можем понять в какое из двух (или трёх) деревьев идти.

Вставка в 2-3 дерево.

Найдём место, в котором элемент должен быть и просто вставим его. Может случиться, что детей уже было 3, а теперь стало 4. Тогда разобьём нашу вершину (у которой стало 4 сына) и разобьём её на 2 и переподвесим их теперь к её матушке. Ну и продолжаем это процесс пока не дойдём до корня. Если вдруг мы рассплитили корень, то создадим новый корень и подвесим к нему наши две вершины. После того, как мы всё сделали, можно запуститься рекурсивно вверх от только что вставленной и идём вверх. Это работает быстро, потому что уровней логарифм.

Удаление из 2-3 дерева. Найдём элемент и тупо удалим его. Если у его предка было три ребёнка - всё хорошо. Предположим, что у вершины было 2 ребёнка, а остался один (трагично, не правда ли?). Теперь начинаются боль и мучения. В общем тут перебор случаев, который можно разобрать кроме того случая, когда у нас только один брат, у которого ровно два сына. Тогда мы переподвешиваем нашего сына к брату и переходим на уровень выше. Если мы упрёмся в корень, то просто возьмём корень и удалим его за ненадобностью.

Всё 2-3 дерево можно провязать двусвязными списками, связывающие вершины на одном уровне.

Почуму бы вместо 2-3 дерева не сделать бы 5-11 дерево? Оценим это так. Пусть у всех вершин степень d. Тогда стоимость операции:

$$d \cdot \log_d n = d \cdot \frac{\ln n}{\ln d} \Rightarrow$$

$$(d \cdot \log_d n)' = (d \cdot \frac{\ln n}{\ln d})' \Rightarrow$$

$$(d \cdot \log_d n)' = (d \cdot \frac{\ln d - 1}{(\ln d)^2})'$$

0 производной в $d = e$. Значит, 2 и 3 - то хорошие приближения.

$B+$ - дерево.

Степень каждой вершины от t до $2t - 1$. Степень корня - от 2 до $2t - 1$.

Поиск вершины - также.

Такие деревья нужны для алгоритмов во внешней памяти.

Тогда ассимптотика - $O(CPU \cdot (d \log_d n) + HDD \cdot \log_d n)$. Тогда, хочется брать достаточно большой d.

Вставка:

Ищем место для вставки и смотрим, куда вставить. Если в какой-то момент стало $2t$ детей, то сплитим на t и t и переподвешиваем.

Удаление: Пусть в вершине теперь стало $t-1$ детей (иначе всё хорошо). Если у нас соседний брат - "Большой Брат"(хотя бы $t+1$ ребёнок), то всё - мы нашли жертву, которую будем грабить (переподвешиваем одного из сыновей брата к нашей вершине). Если нет, то вдохновившись небезызвестным произведением Кена Кизи, подкидываем брату $t-1$ кукушонка. Переходим на уровень выше и продолжаем процесс рекурсивно. Отдельно оговорим корень. Тут проблема только тогда, когда у него осталась 1 вершина, но тогда корень улетает в небытие, и в этом городе появляется новый корень (время для нового Жожо).

А сейчас будет нерекурсивная вставка в B-дерево. Ослабим ограничения на вершины теперь границы это $t-1$ и $2t-1$. Во время поиска вершины будут разбиваться в момент спуска. Как только вершина может переполниться (её степень $2t-1$), тогда разобьём нашу вершину на $t-1$ и t и пойдём дальше вниз.

Ещё одна отсечка, от которой плкавится мозг - в каждой вершине нам надо хранить массив максимумов детей. Будем хранить эту вещь бинарным деревом поиска (ДД).

Задачи на отрезке. Введем какую-то произвольную операцию $\oplus$, и будем отвечать на запросы $get(l, r)$ на массиве a, ответом на которые будет $a_l \oplus a_{l+1} \oplus \cdots \oplus a_r$. Также введем операцию изменения на отрезке $a_i := change(a_i, x)$.

Потребуем от $\oplus$ ассоциативность — $a \oplus (b \oplus c) = (a \oplus b) \oplus c$.

Префиксные суммы. Если в задаче нет запросов изменения (и элементы образуют группу, то есть есть обратный элемент), то посчитаем $p_i = p_{i-1} \oplus a_i$. Тогда ответ на запрос — это $p_r \oplus p_{l-1}^{-1}$. Если операция некоммутативна, то нужно будет пострадать, но вроде бы можно просто сделать $p_{l-1}^{-1} \oplus p_r$. Построение за $O(n)$, запрос за $O(1)$.

Sparse table. Если элементы не образуют группу, но задача все еще статическая, то можно сохранить значения, соответствующие $\oplus$ по всем отрезкам длины 2^k. Тогда, когда нужно ответить на запрос $get(l, r)$, можно взять перекрывающиеся отрезки $[l, l+2^i)$ и $(r-2^i, r]$. От операции требуется $a \oplus b = b \oplus a$ и $a \oplus a = a$. *Есть модификация, позволяющая обойтись без второго свойства.* Построение за $O(n \log n)$, запрос за $O(1)$.

Segment tree. Хотим к предыдущей задаче добавить обновление в точке. Хотим сохранить какое-то множество отрезков S, чтобы потом по нему восстанавливать ответ на произвольном отрезке $[l, r]$, склеивая не более чем $O(\log n)$ отрезков. Также должно быть не более $O(\log n)$ отрезков, содержащих какой-либо элемент.

Построим двоичное дерево над массивом, где вершина на глубине i будет отвечать за отрезок длины 2^{k-i}, где $n = 2^k$. Разбивать запросы на отрезки будем таким образом: рассмотрим все отрезки внутри запроса, и выкинем вложенные. На каждой глубине мы возьмем не более двух отрезков, поэтому суммарно запросы будут работать за $O(\log n)$.

<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>

Реализация будет такой — сделаем рекурсивную функцию get, которая хочет обойти дерево, зайти во все «ключевые» отрезки запроса, и посчитать итоговый ответ. Для $get(v, l, r)$ бывает три случая:

- v не отвечает ни за что из отрезка $[l, r]$, поэтому не делаем ничего и прекращаем работу.
- $[l, r]$ содержит отрезок вершины v, и тогда можно обработать этот отрезок и прекратить работу.
- Иначе надо спуститься в детей, и повторить процесс

Lazy propagation. Пусть у нас появился запрос $change$, модифицирующий отрезок. Будем хранить в вершине пометки вида «мы хотели сделать со всеми детьми такую-то операцию изменения», которые мы изначально будем проставлять с запросом изменения на все «ключевые отрезки». При этом во время запроса изменения мы по сути не сделаем изменений, только проставим пометки. Но теперь мы при каждом обращении к вершине проталкиваем модификатор (если есть) вниз, и меняем текущее

значение *value*. Таким образом, после проталкивания все величины остаются валидными (кроме тех, которые находятся где-то глубоко в дереве, но к которым мы не спустились).

Иначе говоря, мы считаем, что перед тем, как обратиться к какой-то вершине, мы обязаны обратиться ко всем вершинам-предкам, и тогда можем гарантировать, что в ней будут храниться актуальные значения параметров.

Требования к lazy propagation — дистрибутивность операции *change* относительно $\oplus$, а также модификаторы также должны являться полугруппой.

Запросы на деревьях. Будем обсуждать две задачи: **LCA** и **LA**.

Постановка задачи LCA: даются запросы на вычисления наименьшего общего предка двух вершин v, u. То есть, ответ на запрос — это вершина наибольшей глубины такая, что она является предком и v, и u.

Постановка задачи LA: даются запросы на вычисления предка вершины v на глубине k.

Двоичные подъемы. Для каждой вершины преподсчитаем $p_{v,i}$, которая будет хранить информацию о том, какая вершина является 2^i-ым предком вершины v. Насчитать их можно с помощью обхода дерева за $O(n \log n)$.

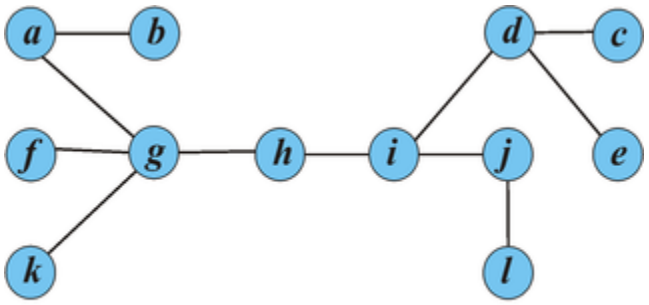
Решение задачи LA с помощью двоичных подъемов тогда будет таким: разложим число k на степени двойки, и сделаем соответствующие «прыжки»: $v \to p_{v,i_1} \to p_{p_{v,i_1},i_2} \to \dots$.

Решение задачи LCA будет таким: сначала мы выровняем вершины по глубине. Иначе говоря, решим задачу LA, чтобы после ее решения вершины запроса имели одинаковую глубину (при этом мы хотим двигать только более глубокую вершину). Теперь, когда вершины на одной глубине (и если не совпали!), мы будем двигаться по степеням двойки от больших к меньшим таким образом: Если LA_{2^i} наших вершин не совпали, то тогда поднимем вершины запроса на 2^i. Такой процедурой мы найдем таких предков вершин u и v, которые не совпадают, находятся на одной глубине, и при этом имеют общего предка. Этот предок и будет ответом на задачу.

Решение с двоичными подъемами работает за $O(n \log n)$ преподсчета и $O(\log n)$ на запрос для обеих задач.

Offline. Решить LA в оффлайн можно обходом в глубину с поддержкой стека за $O(n+q)$. Решение LCA в оффлайн можно сделать с помощью алгоритма Тарьяна с СНМом (просто как факт) за $O((n+q)\alpha)$

Эйлеров обход. Мысленно превратим каждое ребро в два ребра, одно из которых ориентировано вверх, а другое вниз. Тогда в таком графе можно сделать обход по типу Эйлерова — каждое ребро пройдем ровно один раз. Будем выписывать вершину v каждый раз, когда проходим по ребру из v.



Строго говоря, обход можно делать не из корня, а просто потом сделать циклический сдвиг, но обычно все запускают обход из корня и не парятся.

Важное свойство — подотрезок нашего обхода является путем. При этом путь между u и v в эйлеровом обходе содержит $LCA(u, v)$ (потому что путь из u в v точно проходит через $LCA(u, v)$), а еще не содержит предка $LCA(u, v)$ (потому что по ребру в него проход был дважды, и если мы посетим его после lca, то мы уже не могли спуститься обратно). То есть, LCA будет самой высокой вершиной на подотрезке обхода между u и v. Это является сведением к задаче RMQ. Тогда с помощью sparse table можно получить решение за $O(n \log n + q)$.

LA тоже можно решать с помощью эйлерова обхода, делая спуск по дереву отрезков с поиском первой вершины на высоте хотя бы k.

Метод четырех русских. Пушка, которая решит нам LCA за $O(n+q)$. Мы разобьем задачи на большие и маленькие. Нам не обязательно решать маленькие задачи при их появлении, если мы можем заранее решить все возможные маленькие задачи.

$RMQ \pm 1$**.** Заметим, что наше RMQ при поиске LCA обладала тем свойством, что $a_i = a_{i-1} \pm 1,\ a_i \neq a_{i-1}$. Разобьем массив на блоки размера k, в каждом блоке посчитаем максимум и получим массив $b_1,\ \dots\ ,\ b_{\frac{2n}{k}}$. Насчитаем на нем разреженные таблицы. Также в блоке насчитаем префиксные и суффиксные максимумы. Теперь мы умеем за $O(1)$ отвечать на все запросы, кроме тех, которые полностью лежат в одном блоке.

Для решения такой задачи мы посчитаем все возможные последовательности из ± 1 длины k, там возьмем все подотрезки, и для них за линию решим. То есть за $O(2^k \cdot k^3)$. Можно, наверное, и лучше, но нам пофиг.

Теперь положим $k = \lceil \frac{\log n}{2} \rceil$. Тогда маленькая задача решится за $O(\sqrt{n} \cdot \log^3 n)$. А для большой задачи преподсчет будет работать за $O(\frac{n}{k} \log n) = O(n)$. Таким образом, задача решена за $O(n+q)$.

Решение произвольного RMQ делается с помощью $RMQ \to LCA \to RMQ \pm 1$. Первый переход делается с помощью построения ДД на массиве с помощью стека. Тогда RMQ на отрезке это LCA для соответствующих вершин.

Ladder decomposition. Предложим другое решение задачи LA. Разобьем дерево на пути таким образом: возьмем самую высокую вершину, которая еще не покрыта путями, и возьмем из нее самый глубокий путь вниз. Также насчитаем двоичные подъемы. Такое можно решить за $O(n \log n)$. Каждый путь выпишем явно.

После этого мы мысленно удвоим все пути. Возьмем и выпишем еще столько же вершин вверх для каждого пути. Общая память все еще линейна.

Теперь пусть нам надо сделать подъем на k. Найдем наибольшее i такое, что $2^i \leq k$, сделаем такой прыжок. После чего мы оказываемся в вершине, самый глубокий путь из которой вниз был по длине не меньше, чем 2^i. Это значит, что ответ на задачу хранится в выписанном пути для этой вершины и его можно найти за $O(1)$.

Персистентность

Есть структура данных T и операции. Некоторые операции изменяют T, а некоторые не изменяют. Тогда можно говорить о версиях структуры T в разные моменты времени. *Персистенность* - это свойство структуры данных делать запросы к старым версиям.

Уровни персистентности:

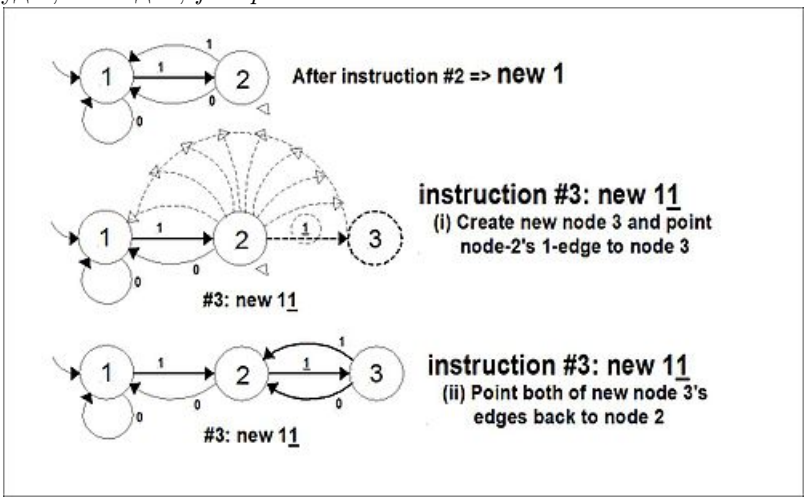
- $parital(weak)$ - версии образуют цепочку, то есть каждая следующая версия - это измененная предыдущая (запросы изменения можно делать только к последней версии).
- $full(strong)$ - версии образуют дерево (запросы изменения можно делать к любой версии).
- $fuctional(confluent)$ - поддерживаются операции сразу для нескольких версий (допустим $merge$ для версий декартова дерева).

Персистенный массив.

Для каждой клеточки храним вектор пар (t_x, v_x) - в момент t_x мы изменили этот элемент массива на v_x (храним в порядке возрастания t_x). Бинпоиском отвечаем на запрос *узнать значение элемента версии t*.

Такая персистентность $partial$, но не $full$ и $functional$.

Pointer machine. Способ организовать структуру данных. Будем хранить все в узловой структуре, поддерживая связи между ними с помощью указателей. Тогда вместо изменения вершины мы просто создаем новую. Таким образом, чтобы изменить вершину x, нужно изменить суммарно $O(h)$ вершин. Такая структура данных будет, очевидно, $full\ persistent$.



Малополезная картинка

Full персистентный массив. Давайте создадим двоичное дерево над массивом размера n с высотой $O(\log n)$, реализовав его как pointer machine. Тогда теперь мы можем сделать изменение произвольного элемента в произвольной версии, получив $O(\log n)$ времени работы (и очевидную реализацию персистентного ДО в придачу).

Персистентное декартово дерево.

Заметим, что наше декартово дерево можно было бы реализовать через pointer machine, что даст нам что-то очень похожее на то, что мы хотим от такой структуры данных. К сожалению, такая структура данных не будет $functional\ persistent$, потому что есть конструктивный способ очень сильно расширить дерево в высоту (достаточно просто мерджить одну и ту же вершину с последней версией дерева, получая бамбук).

Проблема у нас возникла в тот момент, когда мы воспользовались старой идеей приоритетов. Теперь будем вычислять что-то типа приоритетов динамически. А именно, будем считать, что приоритет дерева L больше приоритета дерева R, если $rand() < \frac{S(L)}{S(L)+S(R)}$. Можно показать, что теперь высота ДД все еще $O(\log n)$.

Задачи оптимизации. Во всех предыдущих задачах на структуры данных мы работали, оптимизируя какую-либо очевидную задачу (например, сумму на отрезке). Теперь же мы будем решать задачи, которые непонятно как решать, кроме как полным перебором всех вариантов ответа.

В качестве примера возьмем задачу *subset sum*. В ней нам нужно найти подмножество заданного множества с фиксированной суммой S. Если перебрать все подмножества, и посчитать сумму по каждому подмножеству.

Перебором можно решить любую задачу, если перебрать все возможные ответы (множество вещественных чисел для нас тоже конечно!).

Подмножества хочется как-то пронумеровать. К сожалению, непонятно как закодировать 2^n чисел, если раньше мы разрешали в RAM-модели числа до $C^k \cdot n^k \cdot A$. Поэтому мы просто уточним RAM-модель, и разрешим $C^k \cdot t(n)^k \cdot A$ (Считая $t(n)$ таким временем работы, что внутри нет длинной арифметики).

Динамическое программирование. Иногда бывает полезно запоминать промежуточные величины перебора. Более того, часто перебор можно «ужать», если нам в переборе нужны не все величины (например, в задаче «subset sum» достаточно помнить только общую сумму, если перебирать элементы по очереди).

Сделаем $dp(i, x) \in \{0, 1\}$, которая будет говорить, можно ли набрать сумму x с помощью первых i элементов. Тогда $dp(i, x)$ можно пересчитать через $dp(i-1, x)$ и $dp(i-1, x-w_i)$.

Требования к нашей динамике:

- Граф вычислений ацикличен.
- «Состояния» динамики явно задают нам всю необходимую информацию.

Задача о рюкзаке. Пусть нам заданы n, S, w_i, c_i (то есть элементы с весами и стоимостями). Мы хотим выбрать некоторое подмножество с суммарным весом не более S и максимальной суммой стоимостей.

Мы можем сделать динамику $dp(i, w, c) \in \{0, 1\}$, которая решит нашу задачу. Но заметим, что наше решение монотонно по параметру c (то есть, для равных i и w стоит отдавать предпочтение ответу с максимальным c). Тогда c можно сделать *значением* динамики. То есть, пересчитывать динамику $dp(i, w) \in C$ как максимум из $dp(i-1, w)$ и $dp(i-1, w-w_i) + c_i$. Кстати, заметим, что тут задача монотонна по всем параметрам сразу.

Задача коммивояжера (TSP). Заданы точки на плоскости. Надо найти кратчайший кольцевой маршрут, проходящий по всем точкам хотя бы единожды.

Есть очевидное решение за $O((n-1)!)$. Воспользуемся ДП по подмножествам, основная идея которого — понять, что нам в состоянии важнее всего только то, в каком *множестве* вершин мы уже были, и в каких вершинах мы уже оказались. Закодировать множество мы можем с помощью двоичной маски. Решение с такой идее отработает уже за $O(2^n n^2)$.

ДП по подстрокам. Отдельный трюк, когда подстроки пересчитываются через свои подотрезки. Важное отличие в том, что мы можем пересчитываться через несколько задач сразу ($dp(l, r) = dp(l, k) + dp(k, r)$), а за счет этого порядок пересчета на графе может быть неочевидным.

Meet-in-the-middle. Пусть нам при решении задачи динамического программирования нужно найти кратчайший путь из s в t на графе вариантов. Перебрать весь граф из вершины s может быть тяжело с точки зрения вычислений. Мы попробуем запустить поиск сразу из двух вершин — из s в прямую сторону, и из t в обратную. Тогда любая общая вершина для двух наших переборов задает путь из s в t.

Subset sum. Применим данную технику в задаче *subsetsum*. Поделим множество предметов на две равные группы. Тогда за $2^{\frac{n}{2}}$ мы можем найти все возможные суммы для каждой из двух групп элементов. Теперь за $O(2^{\frac{n}{2}})$ переберем все суммы x для первой группы, и с помощью хэш-таблицы проверить наличие $S - x$ для второй группы.

Рюкзак. Примерно то же самое можно сделать для задачи о рюкзаке, только теперь нам понадобится узнавать максимум на префиксе.

Максимальная клика в графе. Клика — связный подграф. Мы хотим найти максимальную клику в графе. Разобьем граф на две доли (левую и правую, размерами в пополам). Теперь посчитаем $dp_{submask}$ — максимальную подклику для подмножества вершин. Тогда, зная пересечение списков смежности по всем подмножествам вершин, мы можем поступать следующим образом: находить клику в левой доли, брать множество ее соседей в правой доле, и смотреть на соответствующее значение dp.

Компактность динамики. Будем считать динамику *компактной* по какому-то из измерений, если для пересчета значений динамики нужно помнить *не очень много* слоев по этому параметру. Например, динамика для НОП будет компактной по обоим параметрам (и там, и там достаточно помнить всего пару слоев).

Оптимизационные задачи. Часто человечество занимается тем, что берет перебор, и пытается сделать этот перебор оптимально возможным, чтобы работало ну очень-очень быстро. Для примера подобной задачи часто решают задачу коммивояжера (TSP), которая не имеет решения лучше, чем за экспоненту, но при этом современными методами решается для $n \sim 200$. Задача, между тем, актуальная, потому что транспортные компании, вот это все. Что должен сделать хороший перебор:

- Запоминать промежуточные решения, чтобы у нас была возможность экстренно остановить перебор.
- Поставить отсечения таким образом, чтобы не идти в те состояния, где **точно** не будет решений.

Задача о ферзях. Пусть мы хотим расставить n ферзей на доске $n \times n$. Как закодировать состояние? Можно, например, с помощью $2n$ координат. Заметим, что гораздо лучше зафиксировать перестановку, а потом делать ферзей (i, p_i). Тогда такие ферзи гарантированно не бьют друг друга по вертикали или горизонтали. Остается только проверить диагонали. Тогда если перебор будет строить перестановку слева направо, то мы сможем «отрезать» некоторые состояния (например, не ставить ферзя в те клетки, которые еще не бились предыдущими ферзями).

Branch & bound method. Мы будем оптимизировать две вещи:

Границы. Нам хочется понимать, когда мы сможем отрезать ветку перебора, чтобы не упустить оптимальный ответ. Например, выходить из ветки, если текущий ответ превышает нынешний оптимальный. Идеально — ввести соответствующую функцию $f(p)$, и делать так:

```
if (cur + f(p) > best) {
    return;
}
```

Такая функция должна, в случае задачи TSP, возвращать такую длину пути, который нам **точно** понадобится пройти. Например, можно взять вес остовного дерева на оставшихся вершинах. Тогда $span \le TSP \le 2 \cdot span$. Хорошая функция оценки!

Кроме того, очень хорошо иметь нормальное приближение ответа *best*. То есть, изначально как-нибудь (отжигом, жадником, итд) найти неплохой ответ, чтобы перебор не шел в заранее ущербные шаги.

Ветви. Тут мы хотим сделать какую-то магию, чтобы перебирать ветки в правильном порядке. То есть, мы хотим запускать самые хорошие ветви в самом начале, особенно на первых слоях. Например, можно ввести оценочную функцию, и запускать перебор в порядке сортировки по этой оценочной функции. В задаче TSP, из физических соображений, можно запускать перебор сначала из ближайшей вершины к текущей. Также можно попытаться прыгнуть сразу на много уровней вниз, соптимизировав это какой-то простой динамикой ($dp_{mask,i,j}$ — наименьшая длина пути через всю маску, если мы начали в i и закончили в j) на маленьких подмножествах-кластерах. Тогда мы знали самый хороший способ взять какое-то подмножество, а тогда мы вместо 2^k ходов сделаем k ходов.

В ветвях есть много пространства для спекулятивных переборов. Например, можно идти в топ-k веток по оценочной функции.

Можно инициализировать решение для неспекулятивного перебора решением из спекулятивного перебора!

Решаем задачи на графе. Задачи о поиске максимальной клики, максимального независимого множества и минимального контролирующего множества сводятся друг к другу, поэтому если мы решили одну задачу (за какое-то $O(f(n))$), то и другие задачи решили. Будем решать задачу о поиске минимального вершинного покрытия.

Асимптотические оптимизации, которые не казались такими сначала. Задачу о вершинном покрытии можно решать за $O(2^n m)$: для каждой вершины решаем, берем мы ее или нет:

$$t(G, V) = t(G \setminus v, V \cup \{v\}) + t(G \setminus v, V)$$

Заметим, что если мы решили не брать какую-то вершину в ответ, то мы обязаны взять всех ее соседей. Тогда мы можем решать задачу на каком-то меньшем графе, а именно:

$$t(G, V) = t(G \setminus v, V \cup \{v\}) + t(G \setminus \{v \cup N(v)\}, V \cup N(v))$$

Рассмотрим максимальную степень вершины в графе $maxd$. Если, $maxd = 0$, то задача решается за $O(1)$. Тогда пусть $maxd \geq 1$. Таким образом, во втором случае число вершин уменьшается на два:

$$t_n = t_{n-1} + t_{n-2}$$

Получили решение за $O(\phi^n m)$, как оценка на числа Фибоначчи.

Давайте сделаем теперь $maxd \geq 2$ — когда у нас остались ребра, для каждого ребра возьмем одну вершину:

$$t_n = t_{n-1} + t_{n-3}$$

$$x^3 - x^2 - 1 = 0$$

Получили $O(1.47^n m)$.

Если $maxd = 2$, то граф разбивается на пути и циклы, в которых поиск вершинного покрытия тривиален. Аналогично получаем решение за $O(1.38^n m)$.

Перебор в антагонистических играх. Нам дано полное двоичное дерево четной глубины $2n$. Фишка стоит в корне дерева. Каждый игрок на своем ходу перемещает фишку налево или направо. В каждом листе написано «0» или «1», причем мы не знаем значения в листах заранее, а можем только спрашивать у оракула (который не играет против нас, то есть неадаптивный).

Решение за 4^n запросов выглядит так: спросить про все листья, а потом посчитать на дереве динамику на выигрыш-проигрыш:

$$t_v = \begin{cases} 1 & \text{если } t_{v_l} = 0 \vee t_{v_r} = 0 \\ 0 & \text{иначе} \end{cases}$$

Это решение можно соптимизировать. Заметим, что если мы нашли переход из вершины в проигрышного сына, то второго сына можно не рассматривать, поэтому какие-то листья мы можем просто не посещать.

Введем t_n — матожидание количества посещенных листьев для глубины $2n$.

Для начала рассмотрим t_1 и все шесть конфигураций:

Листья	матожидание
$[0,0,0,0]$	2
$[0,0,0,1]$	2.5
$[0,1,0,1]$	3
$[0,0,1,1]$	2.5
$[0,1,1,1]$	2.75
$[1,1,1,1]$	3

Заметим, что за каждый спуск на два уровня вниз, мы рассматриваем в среднем не более трех детей. Тогда $t_n \leq 3 \cdot t_{n-1} \leq 3^n$

$\alpha\beta$-отсечение. Проведем предыдущее рассуждение на минимаксной игре (это такая, где в листьях записаны числа, первый игрок хочет минимизировать итоговое число, а второй максимизировать). Введем параметры α и β — гарантии игроков. β — это минимальное число, которое первый игрок может себе гарантировать (то есть первый игрок знает, что не получит больше, чем β). Аналогично α — это максимальное число, которое гарантирует себе второй игрок). У нас должен соблюдаться инвариант $\alpha \leq \beta$, потому что остальные состояния неиграбельные. Как в них можно попасть по ходу перебора? Если в какой-то момент один из игроков сделает невыгодный для себя ход. Понятно, что при оптимальной игре обоих игроков они не будут делать невыгодные для себя ходы.

Как пересчитываются α и β? В листе $\alpha = \beta = get(v)$. Если второй игрок может пойти в состояние со стоимостью x, то он может сделать $\alpha = max(\alpha, x)$. Аналогично первый игрок будет уменьшать β. При этом эти гарантии будут переходить в сыновей вершины, но не в предков — в предки будут переходить только значение вершины (которое мы либо явно посчитали, либо вообще не сччитали, потому что состояние было неиграбельно).

DFS. Алгоритм обхода графа. Каждой вершине присваивается один из трех цветов — белый, серый или черный. Белые вершины мы еще не рассматривали, серые вершины мы рассматриваем сейчас, черные вершины мы больше не рассматриваем. Алгоритм примерно такой:

1. Покрасить текущую вершину в серый цвет.
2. Рекурсивно запуститься из всех белых вершин, соединенных с нашей.
3. Покрасить текущую вершину в красный цвет.

Что-то из важных утверждений про dfs:

- Серые вершины образуют путь.
- Ребро между двумя серыми несоседними вершинами существует тогда и только тогда, когда в графе есть цикл.
- Из черных вершин нет ребер в белые.

Топологическая сортировка. Пусть у нас есть ориентированный ациклический граф. Давайте выпишем такую перестановку вершин — p_v будет номером покраски вершины v в черный цвет. Тогда все наши ребра будут идти только справа налево.

Поиск компонент сильной связности и конденсация. Компонентой сильной связности называем такой класс эквивалентности, гда v и u сильно связаны, если есть пути $u \to v$ и $v \to u$. Конденсацией мы назовем такой мета-граф, где все компоненты сильной связности «сжаты» в одну вершину, а ребра между новыми компонентами есть, если есть ребра между какой-то парой вершин из этих компонент. Такой граф уже точно будет ацикличным.

Сделаем процедуру, аналогичную топсорту, но теперь у нас ребра уже могут идти слева направо. Про самую последнюю вершину в новом порядке мы знаем, что она точно лежит в компоненте истока. Рассмотрим граф G', построенный на обратных ребрах. Тогда если у нас есть ребро в G' справа налево $u \leftarrow v$, то это значит, что вершины u и v лежат в одной компоненте сильной связности. Тогда мы можем последовательно выделять КСС с помощью обхода по обратным ребрам.

2-sat. Автор опоздал на эту часть лекции, поэтому напишет ее сам позднее.

Мосты и точки сочленения. Назовем мостами такие ребра, при удалении которых граф теряет связность. Аналогичные вершины определим как точки сочленения. Как их найти? Сделаем dfs в неориентированном графе, и построем дерево dfs. Те ребра, по которым мы не переходили, мы назовем обратными. В дереве dfs эти ребра будут идти из вершины в ее какого-то предка.

Что тогда верно про мосты? Из поддерева ребра-моста нет ни одного обратного ребра, которое шло бы выше, чем наше ребро. А найти самое высокое ребро в поддереве можно обычной динамикой. Аналогично с точками сочленения.

Отношения вершинной и реберной двусвязности. Проще всего запомнить так — отношения вершинной двусвязности это отношение на ребрах, а отношение реберной — отношение на вершинах.

Вершины v и u находятся в одной компоненте реберной двусвязности, если существует реберно-простой цикл, содержащий u и v.

Отношение такой двусвязности транзитивно. Можно показать, что если $u \equiv v$ и $v \equiv w$ (при этом v и w соединены ребром), то и $u \equiv w$. Для этого надо склеить два цикла, и найти в них новый — из u в w.

Чтобы найти классы эквивалентности, можно удалить все мосты в графе.

Отношение вершинной двусвязности определяется аналогично, но на ребрах.

Задача о поиске кратчайшего пути. Пусть нам дан граф $G = (V,\ E)$. Каждому ребру задан какой-то вес, а весу пути соответствует суммарный вес всех ребер на пути.

Веса на ребрах могут быть естественными (неотрицательные), или не естественными (разрешаем отрицательный вес ребер). Также отдельный случай для нас — существование циклов отрицательного веса. Кроме того, мы иногда хотим искать любые пути, иногда реберно простые, иногда вершинно простые.

Bfs. Пусть мы хотим найти кратчайший путь, где вес каждого ребра равен 1. Идея такая — мы хотим построить слоистую декомпозицию, где уровню i соответствуют вершины на расстоянии i от стартовой. Алгоритм реализуется на очереди.

Кратчайшие расстояния в графе. В большинстве задач мы считаем, что циклов отрицательного веса нет, оптимальный путь простой, а на кратчайшие расстояния накладывается неравенство треугольника.

Алгоритм Форда-Беллмана. Задача — найти кратчайшие расстояния из s до всех вершин графа. Мы хотим на каждом шаге перебирать все ребра, и поддерживать инвариант — для шага i мы считаем, что алгоритм нашел кратчайшие расстояния среди всех путей длины i. Тогда на следующем шаге мы рассмотрим все ребрра, и продолжим старые пути новыми ребрами, сделаем релаксации.

Циклы отрицательного веса. Заметим, что наш алгоритм не будет делать релаксации после шага n, потому что больше будет нечего релаксировать. Но в случае с циклами отрицательного веса это не так — мы знаем, что на каждом шаге после n-го мы будем делать релаксацию, проходя по улучшающему ответ циклу. Тогда, если мы для каждой релаксации запомнили, какая вершина улучшала наш текущий ответ, мы можем восстановить цикл (или оптимальный путь), просто проходясь по предкам.

Алгоритм Флойда. Задача — найти матрицу кратчайших расстояний. Пусть $f_k(u, v)$ — это кратчайшее расстояние между u и v, если в качестве промежуточных вершин разрешено использовать вершины $1, \ldots, k$. Тогда при переходе $f_k(u, v) \to f_{k+1}(u, v)$ мы хотим сделать релаксацию пути как $u \to k + k \to v$.

Алгоритм Дейкстры. В реальных задачах редко присутствуют ребра отрицательного веса, и в таких ситуациях часто используется алгоритм Дейкстры. Инвариант алгоритма — известны кратчайшие расстояния от s до вершин множества A. Мы знаем, что $\max\ p(v) \le \min\ p(u),\ v \in A,\ u \in V \setminus A$. Давайте для всех вершин поддерживать какое-то текущее $p(v)$. Понятно, что для вершин из A это будет итоговым ответом. Утверждается, что в A можно добавить вершину с минимальным текущим $p(v)$. Доказательство от противного — если текущее расстояние до v было не кратчайшим, то в v был путь, проходящий через вершину $u \in V \setminus A$, но тогда $p(u) < p(v)$, противоречие.

Тогда мы хотим добавлять вершины в A по очереди, по ходу обновляя значения $p(v)$ для всех соседей добавляемой вершиной. Тогда нам нужна структура данных, которая сделает n извлечений минимума и m уменьшений ключа. Оптимальная структура (асимптотически!) — фибоначчиева куча, с ней алгоритм работает за $O(n \log n + m)$.

Потенциалы. Создадим новый граф, где каждой вершине зададим потенциал c_v. Теперь за вес ребра примем $w_c(u, v) = w(u, v) + c_v - c_u$. То есть, вес ребра плюс разность потенциалов. Тогда длина пути $s \to t$ в новом графе определяется как длина в старом графе с дополнительным слагаемым $c_t - c_s = \text{const}$.

Задача о размещении потенциалов эквивалентна поиску отрицательного цикла. То есть, если цикл отрицательного веса есть, то не существует потенциалов, для которых выполняется неравенство треугольника (например, вес такого цикла в новом графе будет меньше нуля). Если же циклов нет, то сделаем $c_v = p(s, v)$. Тогда $w_p(u, v) = w(u, v) - p(v) + p(u)$, но $p(v) \le p(u) + w(u, v)$, а тогда вес новых ребер положителен, а неравенство треугольника выполнено.

A-star. Пусть мы хотим решить задачу на плоскости, при этом хочется не посещать заведомо бесполезные состояния. Введем какую-то метрику $d(u, v)$, в качестве которой введем евклидово расстояние. Теперь мы вытаскиваем новые вершины по минимуму $p(u) + w(u, v) + d(v, t)$ (то есть минимальная сумма расстояния и оценки на метрику). Здесь t - это вершина, кратчайший путь в которую мы пытаемся найти. Поскольку на метрику введено неравенство треугольника, то наша новая Дейкстра все еще работает.

Двусторонняя Дейкстра. Аналогично обычной Дейкстре, запускаем алгоритм в две стороны, и обновляем ответ, если видим ребро $u \leftrightarrow v,\ v \in B,\ u \in A$.

Остовные деревья. Пусть нам дан взвешенный граф. Мы хотим оставить в нем подмножество ребер минимального суммарного веса так, чтобы граф оставался связным.

Лемма о безопасном ребре: Для подмножества вершин A есть ребро наименьшего веса, ведущее в дополнение к A. А именно, ведет из $u \in A$ в $v \notin A$. Существует остов, содержащий это ребро. Доказательство: От противного. Пусть мы не взяли ребро $u \leftrightarrow v$. Посмотрим на путь из u в v в дереве. В какой-то момент там нашлось ребро из A в $V \setminus A$. Тогда можно удалить это ребро и заменить его на $v \leftrightarrow u$ — стоимость не увеличится.

Алгоритм Прима. По аналогии с алгоритмом Дейкстры будем поэтапно строить множество A и добавлять в него минимальное ребро из леммы. Но тут важно, что лемма не гарантирует нам, что итоговый ответ хороший, потому что она говорила нам только о том, что ребро принадлежит какому-то ответу. Но есть усиленная версия леммы — если ребро меньше по весу, чем все остальные, то тогда оно обязательно будет в минимальном остове. Поэтому надо как-то неявно задать веса ребер, чтобы они были различны (это в реальной жизни не нужно, только для доказательства), после чего алгоритм Прима нам уже будет гарантировать минимальный ответ.

Система непересекающихся множеств. Структура данных, которая поддерживает следующие операции:

- $get(v)$ — узнать, в каком множестве находится элемент v
- $union(v, u)$ — объединить множество, содержащее v с множеством, содержащим u.
- $check(v, u)$ — проверить, находятся ли элементы в одном и том же множестве. Выражается через два get-а.

Алгоритм Краскала. В начале упорядочим ребра по весу. Будем поддерживать какое-то множество компонент связности. Если для очередного ребра компоненты вершин различны, то можно сделать $union$ в dsu. Работает за $O(sort(m) + union(n) \cdot n + check(n) \cdot m)$

Алгоритм Борувки. Выделим для каждой вершины минимальное ребро. Теперь добавим все эти ребра в остов, и сожмем граф. Дальше сделаем аналогичную операцию. Повторяем, пока не сойдется до одной компоненты. Каждый раз вершины соединяются в компоненты хотя бы по две вершины. Это значит, что итоговая оценка сложности будет $O(m \log n)$. Заметим, что на плотных графах Борувка работает за $O(m)$ — каждый раз число ребер уменьшается в два раза.

Чтобы объединить две асимптотики, получим:

$$\frac{n^2}{4^k} \leq m$$

$$4^k \geq \frac{n^2}{m}$$

$$k \approx \frac{1}{2} \log \frac{n^2}{m}$$

Значит, через k операций мы получим граф, который можем считать полным (а на нем мы работаем за линию!). Тогда асимптотика это $O(m \log \frac{n^2}{m})$.