Первый курс, осенний семестр 2015/16
Конспект лекций по алгоритмам

Собрано 19 июня 2025 г. в 17:32

Содержание

Лекция по алгоритмам #1
Асимптотика

12 сентября

1.1. $\mathcal{O}$-обозначния

Def 1.1.1. $f = \mathcal{O}(g) \quad \exists N > 0, C > 0 \colon \forall n \geq N, f(n) \leq C \cdot g(n)$

Def 1.1.2. $f = \Omega(g) \quad \exists N > 0, C > 0 \colon \forall n \geq N, f(n) \geq C \cdot g(n)$

Def 1.1.3. $f = o(g) \quad \forall C > 0 \exists N > 0 \colon \forall n \geq N, f(n) \leq C \cdot g(n)$

Def 1.1.4. $f = \omega(g) \quad \forall C > 0 \exists N > 0 \colon \forall n \geq N, f(n) \geq C \cdot g(n)$

Def 1.1.5. $f = \Theta(g) \quad \exists N > 0, C_1 > 0, C_2 > 0 \colon \forall n \geq N, C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$

Следствие 1.1.6. $f = \Theta(g) \Leftrightarrow g = \Theta(f)$

Следствие 1.1.7. $f = \mathcal{O}(g), g = \mathcal{O}(f) \Leftrightarrow f = \Theta(g)$

Следствие 1.1.8. $f = \Omega(g) \Leftrightarrow g = \mathcal{O}(f)$

Следствие 1.1.9. $f = \omega(g) \Leftrightarrow g = o(f)$

Следствие 1.1.10. $f = \mathcal{O}(g), g = \mathcal{O}(h) \Rightarrow f = \mathcal{O}(h)$

Следствие 1.1.11. Обобщение: $\forall \beta \in \{\mathcal{O}, o, \Theta, \Omega, \omega\} \colon f = \beta(g), g = \beta(h) \Rightarrow \boxed{f = \beta(h)}$

1.2. $\mathcal{O}$-обозначния через пределы

Def 1.2.1. $f = o(g)$ *Определение через предел:* $\lim\limits_{n \to +\infty} \frac{f(n)}{g(n)} = 0$

Def 1.2.2. $f = \mathcal{O}(g)$ *Определение через предел:* $\overline{\lim\limits_{n \to +\infty}} \frac{f(n)}{g(n)} < +\infty$

Здесь необходимо пояснение: $\overline{\lim\limits_{n \to +\infty}} f(n) = \lim\limits_{n \to +\infty} (\sup\limits_{x \in [n..+\infty]} f(x))$, где sup – верхняя грань.

1.3. Суммы и интегралы

Lm 1.3.1. $\forall f(x) \nearrow [a..a+1] \Rightarrow f(a) \leq \int\limits_a^{a+1} f(x)dx \leq f(a+1)$

Lm 1.3.2. $\forall f(x) \nearrow [a..b+1] \Rightarrow \sum\limits_{i=a}^{b} f(i) \leq \int\limits_a^{b+1} f(x)dx$

Доказательство. Сложили неравенства из 1.3.1 ■

Lm 1.3.3. $\forall f(x) \nearrow [a..b], f > 0 \Rightarrow \int\limits_a^b f(x)dx \leq \sum\limits_{i=a}^{b} f(i)$

Доказательство. Сложили неравенства из 1.3.1, выкинули $[a-1, a]$ из интеграла. ■

Lm 1.3.4. $\forall f(x) \nearrow [a..b{+}1] \int\limits_a^{b+1} f(x)dx - \sum\limits_{i=a}^{b} f(i) \le f(b{+}1) - f(a)$

Теорема 1.3.5. *Замена суммы на интеграл*

$$\forall f(x) \nearrow [1..\infty), f > 0, S(n) = \sum_{i=1}^{n} f(i), I_1(n) = \int\limits_1^n, I_2(n) = \int\limits_1^{n+1}, I_1(n) = \Theta(I_2(n)) \Rightarrow \boxed{S(n) = \Theta(I_1(n))}$$

Доказательство. Из лемм 1.3.2 и 1.3.3 имеем $I_1(n) \le S(n) \le I_2(n)$.
$C_1 I_1(n) \le I_2(n) \le C_2 I_1(n) \Rightarrow I_1(n) \le S(n) \le I_2(n) \le C_2 I_1(n)$ ■

1.4. Примеры

- **Вложенные циклы for**

```
#define forn(i, n) for (int i = 0; i < n; i++)
int counter = 0, n = 100;
forn(i, n)
  forn(j, i)
    forn(k, j)
      forn(l, k)
        forn(m, l)
          counter++;
cout << counter << endl;
```

Чему равен `counter`? Во-первых, есть точный ответ: $\binom{n}{5} \approx \frac{n^5}{5!}$. Во-вторых, мы можем сходу посчитать число циклов и оценить ответ как $\mathcal{O}(n^5)$, правда константа $\frac{1}{120}$ важна, оценка через $\mathcal{O}$ не даёт полное представление о времени работы.

- **Число делителей числа**

```
vector<int> divisors[n + 1]; // все делители числа
for (int a = 1; a <= n; a++)
  for (int b = a; b <= n; b += a)
    divisors[b].push_back(a);
```

За сколько работает программа?
$\sum\limits_{a=1}^{n} \lceil \frac{n}{a} \rceil = \mathcal{O}(n) + \sum\limits_{a=1}^{n} \frac{n}{a} = \mathcal{O}(n) + n \sum\limits_{a=1}^{n} \frac{1}{a} \overset{1.3.5}{=} \mathcal{O}(n) + n \cdot \Theta(\int\limits_1^n \frac{1}{x} dx) = \Theta(n \log n)$

1.5. Сравнение асимптотик

Def 1.5.1. *Линейная сложность* $\mathcal{O}(n)$

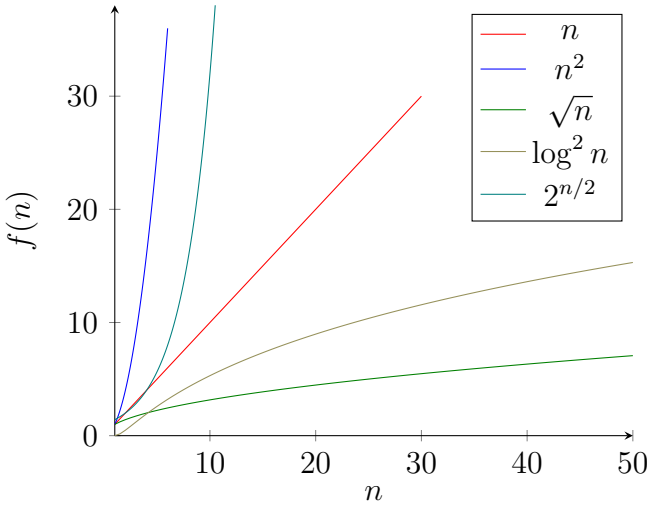
Def 1.5.2. *Квадратичная сложность* $\mathcal{O}(n^2)$

Def 1.5.3. *Полиномиальная сложность* $\exists k > 0 \colon \mathcal{O}(n^k)$

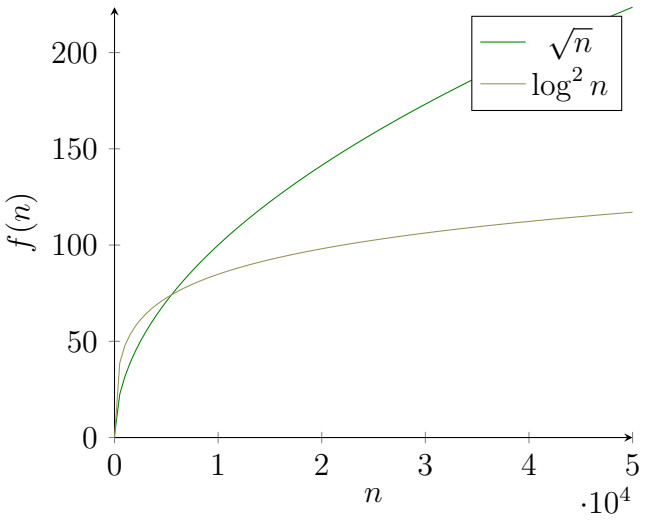
Def 1.5.4. *Полилогарифм* $\exists k > 0 \colon \mathcal{O}(\log^k n)$

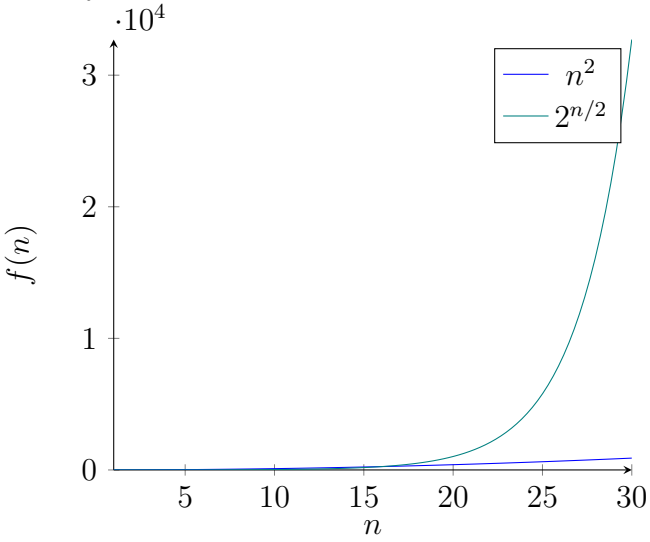
Def 1.5.5. *Экспоненциальная сложность* $\exists c > 0 \colon \mathcal{O}(2^{cn})$

Попробуем сравнить функции:



Заметим, что $2^{n/2}, n^2$ и $\log^2 n, \sqrt{n}$ на бесконечности ведут себя иначе:





Теорема 1.5.6. *Сравнение асимптотик* $\forall a > 0, b > 0, c > 1, \log^b = \mathcal{O}(n^a), n^a = \mathcal{O}(c^n)$

Доказательство. Смотри в следующей лекции... ■

1.6. Рекуррентности

• Пример: алгоритм Карацубы

Чтобы перемножить два десятичных числа A и B длины n, разделим их на части по $k = \frac{n}{2}$ цифр – A_1, A_2, B_1, B_2. Заметим, что $A \cdot B = (A_1 + 10^k A_2)(B_1 + 10^k B_2) = A_1 B_1 + 10^k (A_1 B_2 + A_2 B_1) + 10^{2k} A_2 B_2$. Если написать рекурсивную функцию умножения (числа длины 1 будем умножать за $\mathcal{O}(1)$), то получим время работы:

$$T_1(n) = 4T_1(\tfrac{n}{2}) + \Theta(n)$$

Из последующей теоремы мы сделаем вывод, что $T_1(n) = \Theta(n^2)$. Алгоритм можно улучшить, заметив, что $A_1 B_2 + A_2 B_1 = (A_1 + A_2)(B_1 + B_2) - A_1 B_1 - A_2 B_2$, где вычитаемые величины уже посчитаны. Итого три умножения вместо четырёх:

$$T_2(n) = 3T_2(\tfrac{n}{2}) + \Theta(n)$$

Из последующей теоремы мы сделаем вывод, что $T_2(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.585\ldots})$.

Lm 1.6.1. *Доказательство по индукции*
Есть простой метод решения рекуррентных соотношений: угадать ответ, доказать его по индукции. Рассмотрим на примере $T(n) = \max\limits_{x=1..n-1}\big(T(x) + T(n-x) + x(n-x)\big)$.
Докажем, что $T(n) = \mathcal{O}(n^2)$, для этого достаточно доказать $T(n) \le n^2$:
База: $T(1) = 1 \le 1^2$.
Переход: $T(n) \le \max\limits_{x=1..n-1}\big(x^2 + (n-x)^2 + x(n-x)\big) \le \max\limits_{x=1..n-1}\big(x^2 + (n-x)^2 + 2x(n-x)\big) = n^2$

Теорема 1.6.2. *О простом рекуррентном соотношении*
Пусть $T(n) = aT(\frac{n}{b}) + n^c \log^d n$. При этом $a > 0, b > 1, c \ge 0, d \in \mathbb{R}$.
Определим глубину рекурсии $k = \log_b n$, обозначим $f(n) = n^c \log^d n$. Тогда верно одно из трёх:

$$\begin{cases} T(n) = a^k = n^{\log_b a} & a > b^c \\ T(n) = f(n) & a < b^c \\ T(n) = k \cdot f(n) & a = b^c \end{cases}$$

Доказательство. Раскроем рекуррентность:
$T(n) = f(n) + aT(\frac{n}{b}) = f(n) + af(\frac{n}{b}) + a^2 f\big((\frac{n}{b})^2\big) + \cdots = n^c \log^d n + a\frac{n}{b}^c \log^d \frac{n}{b} + a^2 \frac{n}{b^2}^c \log^d \frac{n}{b^2} + \ldots$
Все полилогарифмы примерно равны, поговорим об этом позже.
Тогда $T(n) = f(n)(1 + \frac{a}{b^c} + \big(\frac{a}{b^c}\big)^2 + \cdots + \big(\frac{a}{b^c}\big)^k)$. При этом в сумме $k+1$ слагаемых.
Обозначим $q = \frac{a}{b^c}$ и оценим сумму $S(q) = 1 + q + \cdots + q^k$.
Если $q = 1$, то $S(q) = k + 1 = \log_b n + 1 = \Theta(\log_b n)$
Если $q < 1$, то $S(q) = \frac{1-q^{k+1}}{1-q} = \Theta(1)$
Если $q > 1$, то $S(q) = q^k + \frac{q^k - 1}{q-1} = \Theta(q^k)$ ■

Теорема 1.6.3. *О экспоненциальном рекуррентном соотношении*
Пусть $T(n) = \sum b_i T(n - a_i)$. При этом $a_i > 0, b_i > 0, \sum b_i > 1$.
Тогда $T(n) = \Theta(\alpha^n)$, при этом α можно найти бинарным поиском.

Доказательство. Предположим, что $T(n) = \alpha^n$, тогда
$\alpha^n = \sum b_i \alpha^{n-a_i} \Leftrightarrow 1 = \sum b_i \alpha^{-a_i} = f(\alpha)$.
Если $\alpha = 1$, то $f(\alpha) = \sum b_i > 1$, если $\alpha = +\infty$, то $f(\alpha) = 0 < 1$. Кроме того $f(\alpha) \nearrow [1, +\infty)$.
Получаем, что на $[1, +\infty)$ есть единственный корень уравнения $1 = f(\alpha)$ и его множно найти бинарным поиском. Теперь можно по индукции доказать,
что $T(n) = \mathcal{O}(\alpha^n)$ (оценку сверху) и $T(n) = \Omega(\alpha^n)$ (оценку снизу). ■

Лекция по алгоритмам #2
Время работы программ

11 сентября

2.1. Время работы некоторых операций

При написании программы, если хочется, чтобы она работала быстро, стоит обращать внимание не только на асимптотику, но и избегать использования некоторых операций, которые работают дольше, чем кажется.

1. Операции библиотеки `math.h`: `sqrt, cos, sin, tg, arctg` и т.д. Эти операции раскладывают переданный аргумент в ряд, что происходит не за $\mathcal{O}(1)$.
2. Взятие числа по модулю, деление с остатком (`a / b, a % b`);
3. `Random access` к памяти при больших массивах. Существует 2 способа прохода по массиву
   `Random access:` `for (i = 0; i < n; i++)sum += a[p[i]];`, где p – случайная перестановка
   `Sequential access:` `for (i = 0; i < n; i++)sum += a[i];`
   Первый обращается к случайным элементам, а второй пробегает весь массив строго по порядку. Если массив не влезает в кеш ($\sim 4M$), то он хранится в оперативной памяти. Когда мы обращаемся к одному из элементов, часть массива (около 2^{12} элементов) переносится в кеш, где происходит дальшейшая работа с ним. Если мы захотим обратится к элементу, которого в кеше нет, но он находится в оперативной памяти, то скопированный в кеш кусок удаляется и записывается следующая часть, где содержится нужный нам элемент. Поэтому, когда мы обращаемся к элементам массива в случайном порядке, в худшем случае мы каждый раз будем переносить большие куски массива в кеш из оперативной памяти. Это долго. Когда же мы идем по массиву по порядку, то переносить части массива нам придется намного реже. Если у нас есть два алгоритма: $T_1 = T_2 = \mathcal{O}(n^2)$; $M_1 = \mathcal{O}(n^2)$; $M_2 = \mathcal{O}(n)$, то второй алгоритм будет работать быстрее для больших значений n.
4. Функции работы с памятью: `new`, `delete`.
5. Вызов функций. Для оптимизации можно использовать `inline` – указание оптимизатору, что функцию следует попытаться вставить в код.
6. Ввод и вывод данных. `cin/cout`, `scanf/printf`... Используйте буфферизированный ввод/вывод через `fread/fwrite`.

• Теперь о хорошем

Быстрые операции: `memcpy(a, b, n)`, `strcmp(s, t)`. Работают в 8 раз быстрее цикла `for`. Почему они быстрые? SSE и AVX регистры!

2.2. Оценка количества операций.

Теорема 2.2.1. $\forall x, y > 0, z > 1 : \exists N : \forall n > N : \log^x n < n^y < z^n$

Доказательство. Пусть $\log n = k$, тогда $\log^x n < n^y \Leftrightarrow k^x < 2^{ky} = (2^y)^k = z^k \Leftarrow n^y < z^n$ ■
Докажем вторую часть исходного неравенства $n^y < z^n$: $n^y < z^n \Leftrightarrow n < 2^{\frac{1}{y} n \log z}$
Пусть $n' = \frac{1}{y} n \log z$, обозначим $C = \frac{1}{y} \log z$, пусть $C \leq n'$ (возьмём достаточно большое n), тогда $n^y < z^n$: $n^y < z^n \Leftrightarrow n < 2^{\frac{1}{y} n \log z} \Leftrightarrow C \cdot n' < 2^{n'} \Leftrightarrow (n')^2 < 2^{n'}$

Докажем по индукции: если $n \to 2n : n^2 \to 4n^2 : 2^n \to 2^n \cdot 2^n$ (т.е. одно увеличилось в 4 раза, второе в 2^n раз)
База: для любого значения из интервала [10..20) верно т.к. $20^2 = 400 < 1024 = 2^{10}$.
$[10..20) \to [20..40) \to [40..80) \to \ldots$

■

Лекция по алгоритмам #3
Структуры данных

11 сентября

3.1. Массив

Создать массив целых чисел на 10 элементов: $inta[10]$; индексация начинается с 0, массивы имеют фиксированный размер.
Функции:

1. $a[i]$ – обратиться к элементу массива с номером i
2. $a[i] = x$ – присвоить элементу под номером i значение x
3. $find(x)$ – найти элемент со значением x
4. $add, insert, put, append$ – добавить элемент
5. $delete, erase, remove$ – удалить элемент

Последние две команды работают очень долго – надо найти новый кусок памяти нужного размера, скопировать весь массив туда, удалить старый.

3.2. Двусвязный список

Двусвязный список состоит из элементов данных, каждый из которых содержит ссылки на следующий и на предыдущий элементы. Первый элемент списка ссылается на $NULL$ в качестве своего левого соседа, последний – на $NULL$, как на правого.
Функции:

1. $add\_front(x), add\_after(node * v, intx), add\_back(x) : \mathcal{O}(1)$ – добавить элемент (в начало, в середину, в конец)
2. $del\_front(), del\_after(node*v), del\_back() : \mathcal{O}(1)$ – удалить элемент (из начала, из середины, из конца)
3. $find(x) : \mathcal{O}(length)$ – найти элемент со значением x
4. $get(i) : \mathcal{O}(min(i, length - i))$ – получить значения элмента с номером i
5. $add(i, x) = get + add\_after$

```
1 struct node {
2   node *l, *r;
3   int data;
4 }
5 struct List {
6   node *head, *tail;
7   int length;
8 }
```

3.3. Односвязный список

Это список, где указатели ссылаются только на одного соседа (либо левого, либо правого, для каждого узла одинаково). Всем остальным оно не отличается от двусвязного списка.

```
1 struct node {
2   node *next;
3   int data;
4 }
```

3.4. Стек

Функции:

1. *pop_back* – удалить с конца
2. *push_back* – добавить в конец
3. *back* – посмотреть последний элемент

3.5. Очередь

Функции:

1. *pop_front* – удалить из начала
2. *push_back* – добавить в конец
3. *front* – посмотреть первый элемент

3.6. Дек

Функции:

1. *pop_back* – удалить с конца
2. *pop_front* – удалить из начала
3. *push_back* – добавить в конец
4. *push_front* – добавить в начало
5. *back* – посмотреть последний элемент
6. *front* – посмотреть первый элемент

При использовании стека, дека и очереди могут быть проблемы с временем работы, так что лучаше реализовывать их самостоятельно, например, на списках.

3.7. Векторы

Вектор – динамический массив (array + stack).
Функции:

1. *pop_back* – удалить с конца
2. *push_back* – добавить в конец
3. $vect[i]$ – обращение к элементу с номером i
4. $set(i, x)$ – добавить элемент со значением x после элемента под номером i

Вместо того, чтобы каждый раз выделять новую память под один элемент, можно один раз увеличить размер вектора в 2 раза, когда он весь заполнится элементами, и понадобится новое место, снова увеличим его в 2 раза и т.д. В среднем, время работы такого алгоритма будет среднее арифметическое подряд идущих команд *push*.

Лекция по алгоритмам #4
Амортизированное время работы

11 сентября

4.1. Амортизация

Реальное время работы операции – t_i.
Среднее время работы операции – $\frac{t_i}{n}$.
Рассмотрим произвольную функцию φ (потенциал).

Def 4.1.1. *Амортизированное время работы программы*
$a_i = t_i + \Delta\varphi$, *где* $\Delta\varphi = \varphi_{i+1} - \varphi_i$ *(реальное время плюс изменение потенциала).*

Изначально потенциал равен φ_0, после выполнения i-ой операци – φ_i, в конце φ_{last}. Амортизированное время можно определять с использованием любой потенциальной функции.

Заметим, что $\sum a_i = \sum t_i + (\varphi_{last} - \varphi_0) \Rightarrow \frac{\sum t_i}{n} \leq \max a_i + \frac{\varphi_0 - \varphi_{last}}{n}$. В амортизационном анализе наша цель – подобрать φ так, чтобы и $\max a_i$, и $\frac{\varphi_0 - \varphi_{last}}{n}$ были не велики.

Лекция по алгоритмам #5
Очередь с минимумом

14 февраля

5.1. Стек с поддержанием минимума

Структура данных, реализующая возможности стека, но модифицированная для нахождения минимума среди всех элементов в стеке за $\mathcal{O}(1)$.

Идея состоит в том, чтобы поддерживать сразу два стека. Первый стек содержит в себе элементы как они есть. Второй же содержит в себе «частичные минимумы», то есть минимумы на отрезке от вершины стека до позиции элемента. Назовём эти два стека a и min.

Рассмотрим пример. Пусть мы добавили поочередно элементы 1, 9, 3, 8, 9, 4, 5, 7. Тогда стеки будут выглядеть следующим образом:

a : 7 5 4 9 8 3 9 1
min : 7 5 4 4 4 3 3 1

- **Реализация методов**

```
1  void push(T x) {
2    a.push(x);
3    min.push(min(min.top(), x));
4  }
5
6  T min() {
7    return min.top();
8  }
9
10 T pop() {
11   min.pop();
12   return a.pop();
13 }
14
15 T top() {
16   return a.top();
17 }
```

5.2. Очередь с поддержанием минимума

Структура данных, реализующая возможности очереди, но модифицированная для нахождения минимума среди всех элементов в очереди за амортизированное $\mathcal{O}(1)$.

Идея состоит в том, чтобы научится реализовывать очередь с помощью двух стеков. Тогда если вместо обычных стеков взять стеки с поддержанием минимума, то очередь также сможет поддерживать минимум.

- **Реализация очереди с помощью двух стеков**

Заведём два стека: a и b. Добавлять новые элементы будем всегда в стек b, а извлекать элементы – только из стека a. При этом, если при попытке извлечения элемента из стека a он оказался пустым, просто перенесём все элементы из стека b в стек a (при этом элементы в стеке a получатся уже в обратном порядке, что нам и нужно для извлечения элементов; стек b же станет пустым).

- **Реализация методов**

```
1  void push(T x) {
2    b.push(x);
3  }
4
5  T pop() {
6    if (a.size() == 0)
7      while(b.size() > 0)
8        a.push(b.pop());
9    return a.pop();
10 }
```

- **Время работы**

Все методы работают за амортизированное $\mathcal{O}(1)$. Объясняется это тем, что каждый элемент за время работы очереди один раз попадёт в стек b, один раз перейдёт в стек a и один раз покинет его. Таким образом, если всего элементов n, то стек затратит на выполнение всех операций $\mathcal{O}(n)$ времени, то есть $\mathcal{O}(1)$ на каждую операцию в среднем.

Лекция по алгоритмам #6
Очередь с минимумом

20 сентября

Нам нужна структура данных, которая умеет делать следующие операции:

1. $push\_back(a)$ - добавить элемент a в конец
2. $front()$ - узнать, какой элемент стоит в начале
3. $pop\_front()$ - удалить элемент, который стоит в начале, из структуры
4. $get\_min()$ - узнать минимум среди всех элементов в структуре

Причём амортизированная сложность (время работы) всех операций должна быть $\mathcal{O}(1)$.

6.1. Алгоритм

Для того, чтобы решить эту задачу, пристроим к обычной очереди дек. В очереди мы честно будем хранить все элементы, а в деке мы будем хранить все элементы, которые когда-нибудь могут стать минимумами. Причём элементы в деке хранятся в порядке добавления в очередь и возрастают.

В чём идея? Пусть в конец очереди добавился элемент, который меньше последнего. То есть произошло следующее:

1. $push\_back(a)$

   ...
2. $push\_back(b)$
3. $a \geq b$

Тогда элемент a больше не влияет на минимум. Действительно, так как a ближе к началу очереди, чем b, то a удалится из очереди раньше b, а пока b находится в очереди, $min \leq b \leq a$.

Теперь, кажется, понятно, как добавлять элементы в дек. Мы знаем, что элементы там отсортированы от начала к концу (база - один элемент). Поэтому можно просто удалить весь суффикс[1] дека, элементы которого больше нового (они минимумами уже никогда не станут) и добавить новый в конец. Ясно, что после этой операции дек сохранит все нужные нам свойства.

Итак, мы имеем очередь Q и дек M. Как выглядят наши операции:

- $push\_back(a)$ - добавим a в конец Q, удалим из M суффикс, элементы которого больше a и добавим a в конец M.
- $front()$ - $Q.front()$.
- $pop\_front()$ - сделать $Q.pop\_front()$ и проверить, надо ли удалить первый элемент M. Это можно сделать так: присвоим каждому элементу уникальный номер (например время добавления) и будем делать $M.pop\_front()$, если $M.front().Time == Q.front().Time$.

[1] отрезок $i \dots n-1$

- $get\_min()$ - $M.front()$

6.2. Псевдокод

```
\\ Q - очередь, M - дек, Time - глобальная переменная "времени"
push_back(a){
     while (!M.empty() && M.back().value > a) M.pop_back();
     Q.push_back(a, Time);
     M.push_back(a, Time);
     Time++;
}

front(){
     return Q.front().value;
}

get_min(){
     return M.front().value;
}

pop_front(){
     if (Q.front().Time == M.front().Time) M.pop_front();
     Q.pop_front();
}
```

6.3. Асимптотика

Из кода видно, что всё, кроме $push\_back()$ работает за $\mathcal{O}(1)$. В цикле $while$ в $push\_back()$ мы удаляем элементы из дека. Ясно, что мы не сможем удалить более N элементов, так как каждый элемент можно удалить не более одного раза. Поэтому $while$ в сумме отработает за $\mathcal{O}(N)$. На языке амортизации:

$$\varphi(n) = M.size()$$

С каждой итерацией $while$ $t(n)$ увеличивается на 1, а $\Delta\varphi$ - уменьшается на 1.

$$\alpha(n) = \Delta\varphi + t(n) = \mathcal{O}(1)$$

$$\sum_{n=1}^{N} t(n) = \sum_{n=1}^{N} \alpha(n) + \varphi_0 - \varphi_N \leq N \cdot \mathcal{O}(1) + N - 0 = \Theta(N)$$

Лекция по алгоритмам #7
Структуры данных

18 сентября

7.1. Частичные суммы

Partial sums, Префиксные суммы. В C++ лежит в <numeric>.

```
Build psum:
psum[0] = 0;
for i = 0..n
  psum[i+1] = psum[i] + a[i]; //O(n)

Get [L..R]:
return psum[R+1] - psum[L];
```

7.2. Бинарный поиск по отсортированному массиву

- **Проверка на наличие**

Хотим узнать, есть ли элемент в массиве. Возвращаем его индекс, если есть, -1 — если нет.

```
l = 0; r = n - 1;
while (l <= r){
  m = (l + r) / 2;
  if (a[m] == x) return m;
  if (a[m] > x) r = m - 1;
  else l = m + 1;
}
return -1;
```

- **Поиск ближайшего**

Хотим узнать, на какой позиции находится первый элемент, больший или равный данному.

```
l = -1; r = n;
while (l <= r){
  m = (l + r) / 2;
  if (a[m] > x) r = m;
  else l = m;
}
return l;
```

7.3. Два указателя

- **Интересные слова**

1. *Offline*. Запросы поступают «пачками», ответ дается на все сразу. Время работы = время на обработку всех запросов.
2. *Online*. Запросы поступают по одному, ответ дается сразу. Время работы = время на обработку одного запроса.

• **Метод двух указателей**

Даны два массива A и B. Про каждый эл-т массива A сказать, есть ли он в B.

```
1  sort (A); sort(B); A[n] = INF; i = 0;
2  for j = 0..B.size()-1{
3    while (A[i] < B[j]) i++;
4    if (A[i] == B[j]) cout << "yes";
5  }
```

Это Offline решение. У задачи есть и Online решение.

```
1  sort(A);
2  for j = 0..B.size()-1{
3    BS(A, B[j]);
4  }
```

Что изменилось: сортируем теперь только один массив, вместо метода двух указателей используем бинпоиск (BS).

7.4. Хеш-таблица

Хотим такую структуру данных, в которой за $\mathcal{O}(1)$ будем уметь добавлять и удалять элементы, искать их. В C++ есть структуры HashSet и HashMap, которые удовлетворяют этим условиям. Различия: у HashMap ключ не обязательно дожен быть int, он может быть любого типа.

unordered_set<int>
unordered_map<string, int>

Чтобы они быстрее работали, стоит заранее выделить им какой-то объем памяти (лучше максимальный, какой может потребоваться). А если еще написать a.rehash(N); то это ускорит процесс на 10%.

• **Хеш-таблица на списках** Будем хранить наши m элементов в n списках. При хорошей хеш-функции длины списков будут примерно $\frac{n}{m}$, то есть все операции будут выполняться за $\frac{n}{m}$. А так как мы хотим $\mathcal{O}(1)$, то m должен быть примерно равен n. Пример хеш-функции: $i = x\%m$, где i - номер списка, x - элемент,к оторый хотим добавить, а m - простое число, не сильно большее n. Если нам n заранее неизвестно, то может случиться ситуация, когда в один список запишется слишком эл-тов, время работы сильно ухудшится. Чтобы этого избежать, нужно время от времени удваивать кол-во списков и перезаписывать уже умеющиеся эл-ты в новую хештаблицу.

Функция поиска в хеш-таблице на списках

```
struct Node{
  Node* Next;
  int x;
}

Node* h[m];

Node* Find(int x){
  Node* v = &h[x % m];
  while (v&&v -> x != x)
  v = v -> next;
  return v;
}
```

Добавление

```
if (finx(x) == 0)
h[x % m] = {h[x % m], x}
```

Удаление

```
find(x) -> x = -1;
```

• **Хеш-таблица с открытой адресацией** Все элементы хранятся непосредственно в хеш-таблице, без использования связных списков. Когда мы попытаемся добавить элемент в уже занятую ячейку, то будем последовательно рассматривать следующие после нее ячейки, пока не найдем пустую. Туда и запишем наш эл-т. Опять-таки может возникнуть ситуация, когда хеш-таблица окажется полностью заполненной. И это тоже решается увеличением размера таблицы и перезаписью эл-тов.

```
int n[m];
fill(h, h+m, -1); // заполняем хеш-таблицу размера m - 1

h[Find(x)] = x; // добавляем эл-т x, если его еще нет
h[Find(x)] = -2; // удаляем x

int Find(int x){
  int i = x % m;
  while (h[i] != -1 && h[i] != x)
    i = (i + 1) % m;
  return i;
}
```

Лекция по алгоритмам #8
Способы собственной аллокации памяти

21 сентября

8.1. Бинарная куча кусков памяти

Задачи этого способа:

1. $new(x)$, возвращает указатель на кусок памяти размера x.
2. $delete(a)$, удаляет кусок памяти a.

Сложности в этом способе начинают возникать, когда после удаления мы хотим выделить кусок. Ведь сумма размеров кусков возможно нам подходит, но нам нужен последовательный отрезок памяти размера x.
В современных операционных системах написаны хитрые аллокаторы, которые довольно оптимально выделяют память. Мы же обсудим упрощенную версию аллокатора основанного на бинарной кучке кусков. Кусок – это пара $\langle$`size, address`$\rangle$. Будем на вершине кучи хранить максимум из размеров. Тогда операции имеют следующий вид:

1. $new(x)$:

   Достаем максимальный элемент из кучи $\langle$`maxsize, addr`$\rangle$. Если $maxsize < x$, то возвращаем кусок в кучу и говорим, что не смогли найти. Если же он нас устраивает $maxsize \geq x$, то скажем, что мы нашли такой адрес $= addr$ и, если $maxsize \neq x$, добавим в кучу $\langle$`maxsize-x, addr+x`$\rangle$.

   Сложность операции: $\mathcal{O}$(извлечение и добавление в кучу) $= \mathcal{O}(log_2 n)$, где n - число кусков в куче.

2. $delete(a)$:

   Чтобы корректно поддерживать куски в куче, нам нужно знать соседей текущего куска. Правый сосед, очевидно, начинается в $addr+size$. Для быстрого нахождения левого соседа заведем HashTable, которая для каждой правой границы куска будет хранить его начало. Тогда, в зависимости от состояния соседей куска мы удалим из кучи тех соседей, которые свободны, и добавим результирующий кусок.

   Сложность операции: $\mathcal{O}$(извлечения и добавление в кучу) + $\mathcal{O}$(запрос к HashTable) $= \mathcal{O}(log_2 n) + \mathcal{O}(1) = \mathcal{O}(log_2 n)$, где n - число кусков в куче.

8.2. Стек

Нужно уметь делать:

1. $new(x) \approx$ POP, возвращает указатель на кусок памяти размера x.
2. $deletelast() \approx$ PUSH, удаляет кусок памяти, выделенный последним.

Преимуществом стека является то, что если его размер не превышает в любой момент k байт памяти, то есть мы будем добавлять какую-то пачку кусков и потом ее освобождать, то они могут закэшироваться и наш стек будет работать быстрее.
Практический пример использованию стека для собственного аллокатора с целью ускорения STL-стуктур (например set или map). Правда, мы не будем освобождать память из-за временных затрат, а воспользуемся только идеей $new(x)$ стека.

```
char mem[1 << 28];
int pos = 0;

void* operator new(size_t x) {
  void* res = mem + pos;
  pos += x;
  return res;
}

void operator delete(void* a) {
}
```

Также в функциях, которые локально используют STL-структуры, можно хорошо "очищать" память, возвращая переменную *pos* в ее состояние до вызова.

```
void f() {
  int old = pos;
  set<int> s;
  s.insert(5);
  s.insert(20);
  ...
  pos = old;
}
```

Но бывают сложные функции с несколькими точками выхода. Следовательно, нам придется везде возвращать значение переменной, а это легко забыть.

```
void f() {
  int old = pos;
  if () {
    pos = old;
    return;
  }
  if () {
    pos = old;
    return
  }
  ...
  pos = old;
}
```

Решение состоит в создании собственной структуры с деструктором. Тогда нужно всего лишь создать экземпляр этой структуры при входе в функцию.

```
struct T {
  int old;
  T() : old(pos) {}
  ~T() {
    pos = old;
```

```
  }
};

void f() {
  T t {};
  if () {
    return;
  }
  if () {
    return
  }
  ...
}
```

8.3. Односвязный список

Функциональность:

1. *new*(), возвращает указатель на кусок памяти константного размера C.
2. *delete*(a), удаляет кусок памяти a.

Будем использовать список свободных кусков. Для этого реализуем структуру Node и переменную указывающую на начало списка.

```
struct Node {
  void* data;
  Node* next;
};

Node* head;
```

Информация занимает C байт, а указатель, например, 4 байта. Тогда размер, на который нам нужно разбить всю память, это $C + 4$ байта.
Операции выглядят следующим образом:

```
void* operator new() {
  void* res = head->data;
  head = head->next;
  return res;
}

void operator delete(void* a) {
  b = (Node*)a;
  b->next = head;
  head = b;
}
```

Нужно заметить, что здесь тоже уместно кэширование, т. к. после удаления куска памяти он становится первым в списке, и будет выдан быстрее, чем все следующие куски в списке.

Лекция по алгоритмам #9
Структуры данных

21 сентября

9.1. Избавление от амортизации. Ленивое копирование

1. **Вектор #1**

   Время работы практически всех операций вектора чистое $O(1)$. Исключение - операция push_back, которая работает за амортизированную константу, так как если приходится удваивать размер вектора, то push_back отработает за линию.

   Удвоение вектора состоит из трех этапов:

   1)Выделение памяти для нового вектора

   2)Копирование элементов из старого вектора в новый

   3)Освобождение памяти из-под старого вектора

   Выделение и освобождение памяти технически зависимые действия, и с точки зрения алгоритмизации мы не можем на них повлиять. Заметим лишь, что если теоретически они работают за $\Theta(n)$, то на практике они намного быстрее и здесь мы можем считать их время работы равным $O(1)$.

   Копирование же в стандартной реализации вектора работает за $\Theta(n)$ и его оптимизацией мы и займёмся. Давайте вместо единоразового копирования всех элементов разобьём его на много маленьких операций, которые мы будем выполнять параллельно с работой основных операций вектора. Выделим для нового вектора память в 2 раза больше размера старого, где первая половина будет соответствовать уже существующим элементам, а во второй будет элемент, добавление которого и потребовало выделение памяти. Но мы не будем копировать все элементы в новый вектор, а будем некоторое время хранить оба и при обращении к элементу, проверять в каком векторе он находится. Копировать же мы будем по одному элементу при каждом вызове push_back и pop_back. Тогда, так как мы копируем по одному элементу при вызове push_back, то как только память нового вектора заполнится, все старые элементы гарантированно будут скопированы, и значит мы сможем его удалить, и нам никогда не придётся хранить больше двух векторов. В pop_back же копирование необходимо, чтобы избежать затирания элементов в новом массиве при поочерёдном вызове push_back, pop_back. Таким образом, если обозначить старый вектор a, новый b, а номер последнего нескопированного элемента k, то код функций теперь выглядит так:

```
1  at*(i)
2      if i <= k
3          return a.at(i)
4      else
5          return b.at(i)
6
7  copy()
8      if k >= 0
9          b[k] = a[k]
10         k--
11
12 push_back*()
13     push_back()
```

```
14      copy()
15
16  pop_back*()
17      pop_back()
18      copy()
```

2. **Вектор #2**
   В первом варианте мы избавились от линейного копирования, разбив его на поэлементный перенос элементов параллельно с работой функций вектора. Но можно копировать элементы не после того, как он заполнился, а заранее. Как только у вектора заполнилась половина ячеек, заведём новый удвоенный вектор и будем переносить в него элементы аналогично прошлой реализации. Единственное отличие, что здесь копировать нужно не по одному, а по два элемента, тогда к моменту заполнения старого вектора все его элементы уже будут перенесены в новый, и старый можно будет удалить.
3. **HashTable**
   Аналогично с вектором элементы хештаблицы можно переносить до или после того, как их количество превысило допустимый предел. В первом случае, пока таблица целиком не скопировалась, операции запроса выполняются только на старом массиве, а добавлять придётся в оба, так как в отличие от вектора в хештаблице добавление происходит не в конец, а в произвольную ячейку. Если же мы создаём новую хештаблицу после заполнения старой, то наоборот добавлять можно только в новую, а запрос придётся совмещать из двух запросов к каждой таблице.
4. **Очередь с минимумом**
   Здесь мы рассматриваем очередь с минимумом на двух стеках. В обычной реализации очередь с минимумом работает за амортизированную константу, так как если при извлечении элемента, выходной стек окажется пуст, то операция отработает за $O(n)$. Для избавления от амортизации давайте воспользуемся уже знакомым по предыдущим пунктам методом ленивого копирования. Как только размер выходного стека становится меньше входного, начинаем копировать элементы входного стека в выходной. Давайте хранить стеки на массиве, причём обе границы стека могут в нём смещаться. Тогда, когда начинается копирование, мы отрезаем от входного стека все его элементы и приравниваем левую и правую границы $R + 1$. И копируем эти элементы "под дно"выходного стека. Тогда для восстановления нормального состояния очереди нам понадобится: a) скопировать n элементов из одного стека в другой, b)посчитать частичные минимумы на n копируемых элементах и на $\leq n$ элементах оставшихся в выходном стеке. Итого $3n$ операций и значит при обработке каждого *push* и *pop* необходимо выполнить 3 операции. Осталось научиться обрабатывать запросы минимума в процессе перестройки очереди. Для этого надо лишь отдельно запомнить минимум на копируемом участке, который заранее доступен на его вершине, и при обработке запроса брать минимум из обоих стеков и копируемого участка.

9.2. Куча(Heap)

Куча - бинарное дерево, которое, если обозначить вершины h_i, а вершина имеет номер 1, удовлетворяет свойствам:

$h_i \leq h_{2i}$

$h_i \leq h_{2i+1}$

Куча по определению выполняет две операции:
ExtractMin() - извлечь из кучи элемент с минимальным ключом
Add(x) - добавить в кучу элемент с ключом x
В stl структурой с такими операциями является priority_queue.
Однако, список возможностей кучи можно расширить ещё 2мя операциями:
Delete() - удалить элемент из кучи
DecreaseKey(y) уменьшить ключ элемента до y
Которые в свою очередь выражаются одна через другую:
Delete() = DecreaseKey($-\infty$) + ExtractMin()
DecreaseKey(y) = Delete() + Add(y)
Получившуюся структуру называют ExtendedHeap.

Реализация

Мы будем реализовывать кучу на массиве. У корня будет индекс 1, а индексы сыновей вершины i: $2i$ и $2i+1$, тогда индекс родителя jой вершины $\lfloor j/2 \rfloor$.

Заведём две вспомогательные функции: SiftUp(i) и SiftDown(i), которые будут просеивать iую вершину соответственно верх или вниз, пока она не займёт корректную позицию.

```
1
2  SiftUp(i)
3      while i>1 && h[i/2] > h[i]
4          swap(h[i], h[i/2])
5          i /= 2
6
7  SiftDown(i)
8      while i <= n
9          m = min(i, 2i, 2i+1) // min(h[i], h[2i], h[2i+1])
10         if(m == i)  break
11         swap(h[m], h[i])
12         i = m
```

Теперь с их помощью можно реализовать основные операции кучи.

```
1
2  Add(x)
3      h[++n] = x
4      SiftUp(n)
5
6  ExtractMin()
7      result = h[1]
8      swap(h[1], h[n--])
9      SiftDown(1)
10     return result
```

Теперь оценим время работы Add и ExtractMin. В худшем случае SiftUp и SiftDown пройдут всё дерево снизу вверх/сверху вниз. Высота бинарного дерева из n вершин $\lceil \log_2(n+1) \rceil$. Значит время работы Add и ExtractMin $O(\log n)$.

HeapSort

Одним из наиболее распространённых применений кучи является HeapSort(пирамидальная сортировка). Алгоритм представляет собой добавление всех элементов в кучу и извлечение их из неё. Тогда первый извлечённый элемент будет минимумом на всём массиве, второй извлечённый минимумом среди всех кроме первого, и так далее. Тогда, так как мы добавляем в кучу n элементов, а затем n элементов извлекаем, то суммарное время работы HeapSort $O(n \log n)$:

```
1
2 Heapsort()
3     For(i, n)
4         heap.Add(a[i])
5     For(i, n)
6         a[i] = h.ExtractMin()
```

Важной оптимизацией в HeapSort, и других алгоритмах, строящих кучу по известному количеству элементов, является построение за $O(n)$. Для этого будем строить кучу не в отдельном массиве, а прямо в массиве a (Inplace):

```
1
2 Build()
3     For i = n..1
4        Sift Down(i)
```

Время работы $= \sum_{k=1}^{\log n} k\lfloor\frac{n+1}{2^k}\rfloor \leq (\sum_{k=1}^{\infty}\lfloor\frac{k}{2^k}\rfloor)(n+1) = \Theta(1)\Theta(n) = \Theta(n)$

Лекция по алгоритмам #10
Пополняемые структуры

28 сентября

10.1. Add → Merge

Научимся делать *Merge* структур, для которых определена операция *Add*. Для этого достаточно добавить все элементы меньшей по размеру структуру в большую.

- **Merge**

```
1  Merge(A,B){
2    if A.size > B.size
3      swap(A,B)
4    foreach a in A
5      B.insert(a)
6    return B
7  }
```

Теорема 10.1.1. Если суммарное количество элементов в структурах не превосходит n, то любая последовательность вызовов *Merge* выполнится за $\mathcal{O}(n \log n \cdot Add)$, где *Add* - временная сложность операции добавления.

Доказательство. Каждый раз, когда для элемента вызывается функция *Add* размер множества, в котором он содержится увеличивается хотя бы в два раза. Таким образом, для каждого элемента *Add* не может быть вызван более $\log_2 n$ раз. Элементов n, следовательно всего вызовов не более $n \log_2 n$, ч.т.д. ■

10.2. Build, Get → Add

10.2.1. С помощью корневой оптимизации

Теперь научимся добавлять *Add* в структуру, которая умеет делать *Build* и *Get* (функция *Get* должна быть аддитивной).

Для этого разобьём структуру на несколько, каждая из которых будет размера m. В функции *Add* будем просто делать *Build* от того, что есть в текущей структуре и нового элемента. Временная сложность - $\mathcal{O}(Build(m))$

- **Add**

```
1  Add(parts,cur,x){
2    if parts[cur].size = m
3      cur++
4    Build(parts[cur], x)
5  }
```

В *Get* будем пробегать по всем структурам, вызывать *Get* от них, а затем брать *Get* от всех результатов. Временная сложность - $\mathcal{O}(\frac{n}{m} Get(m))$

- **Get**

```
1  Get(parts){
2    for A in parts
3      res = combine(Get(A), res)
4    return res
5  }
```

Рассмотрим этот подход на примере структуры $SortedArray$, для которой $Build = Sort$ и $Get = lower\_bound$. В этой структуре для Add можно использовать не $Build$, а обычную вставку в отсортированный массив за $\mathcal{O}(m)$. Тогда сложнность операции Get - $\mathcal{O}(\frac{n}{m}\log m)$, а Add - $\mathcal{O}(m)$. Возьмём m равное $\sqrt{n\log n}$. Тогда сложность обеих операций будет $\mathcal{O}(\sqrt{n\log n})$.

10.2.2. С помощью представления в двоичной системе счисления

Пусть в структуре n элементов. Рассмотрим разложение числа n в двоичной системе счисления - $n = \sum 2^{k_i}$. Будем поддерживать несколько (не более $\log_2 n$) структур размера 2^{k_i}, для каждого слогаемого из суммы.

Теперь $Add(x)$ будет выглядеть следующим образом: создадим новую структуру размера 1 из x, пока у нас есть структуры одного размера вызываем от них $Build$, добавляем его к нашим структурам, а те две удаляем.

Теорема 10.2.1. Пусть $Add = \log N + k_i$, где k_i - суммарное время работы вызовов $Build$ сделанных в этом вызове функции Add. Тогда для серии вызовов функции Add - $\sum k_i \le Build(N\log N)$

Доказательство. Пусть в процессе исползования структуры были сделаны вызовы $Build(b_1), Build(b_2), ..., Build(b_m)$. Будем использовать лемму $\sum Build(b_i) \le Build(\sum b_i)$ без доказательства. Каждый элемент входил в один из вызовов $Build$ не более $\log_2 N$ раз, всего элементов N. Тогда по лемме - $\sum Build(b_i) \le Build(\sum b_i) \le Build(N\log N)$, ч.т.д. ■

10.3. Get → Delete

Научимся удалять элемент из структуры, которая поддерживает Get (функция Get должна быть обратимой). Для этого просто заведём структурку $Deleted$ и при вызове $A.Get()$ будем возвращать $A.Get() - Deleted.Get()$.

Лекция по алгоритмам #11
Арифметические выражения

25 сентября

11.1. Без стека +, -, *, /, (,)

a. Найдем последнюю операцию
По скобкам храним балансы на префиксах. Тогда порядок выполнения операций такой: Максимальный баланс, максимальный приоритет, самая левая из возможных (так как все левоассоциативны)

b. 2 рекурсивных вызова - посчитаем правый и левый аргумент и применим операцию к ним
Время: $\mathcal{O}(n^2)$ в худшем случае

c. Случай: операции внутри скобок кончились $\rightarrow$ уберем их

11.2. Со стеком

11.2.1. Без скобок

Если приходит число, отправим его в стек чисел, операция — в стек операций. Пока у пришедшей операции приоритет не больше, чем у последней, выполняем последнюю для двух последних чисел, заменяя их на результат.

```
while (prior(oper.top()) >= prior(new)) {
  eval();
}
oper.push(new);
```

Еще может случиться, что у нас есть правоассоциативные операции. Тогда условие заменяется на:

```
(prior(top()) > prior(new) || (prior(top()) == prior(new) && type[new] == "L"))
```

11.2.2. Со скобками

```
// prior["("] = +inf;
// prior[")"] = 0;
while (top() != "(" && prior(top()) >= prior(new))
  eval();
if (new == ")")
  oper.pop();
else
  oper.push(new);
```

11.2.3. Унарный минус

a. $2- \sim 3$

```
while (unary(open.top()))
  x = eval(x);
values.push(x)
```

b. $12 = 1 - (-(-2))$ Если последний push в стек операций, то наш минус — унарный, иначе бинарный

11.3. Дерево выражений

Пусть у нас есть переменные. Тогда будем в стеке чисел хранить указатели на вершины дерева выражений. Можно будет подставить значения и посчитать ответ.

```
Node {
  int type;
  int value;
  Node *a, *b;
}
```

Лекция по алгоритмам #12
Сортировки

28 сентября

12.1. Что хотим?

Есть массив a[] (индексация [0, n)), хотим получать массив b[], являющийся такой перестановкой a[], что элементы в нём упорядочены (обычно по неубыванию).

12.2. Свойства сортировок

1. Stable (устойчивость)

   Сортировка устойчива, если равные элементы из a[] сохранили порядок следования относительно друг друга в b[].

2. Inplace (с использованием O(1) дополнительной памяти)

   Если сортировка использует O(1) дополнительной памяти (в частности, отсортированный массив сохраняется в a[]), то сортировка inplace.

12.3. Сортировки за O(n^2)

1. Selection sort (Сортировка выбором)

   Делаем n итераций. На i-той итерации меняем местами a[i] и a[k], где k - индекс максимального элемента среди {a[i], a[i + 1], ..., a[n - 1]}, то есть выбираем, какой элемент будет i-тым в отсортированном массиве.

   Псевдокод:

```
for i in [0, n)
  k = index_of_max(a[i], a[i + 1], ..., a[n - 1])
  swap(a[i], a[k])
```

   Количество сравнений - O(n^2).

   Количество копирований элементов (swap'ов, присваиваний и т.п.) - O(n).

2. Insertion sort (Сортировка вставками)

   Делаем n итераций. Элемент a[i] проталкиваем влево, пока он не окажется на своём месте в отсортированном массиве из первых i элеметов, то есть вставляем a[i] в нужное место отсортированного массива из первых (i - 1) элементов.

   Псеводкод:

```
for i in [0, n)
  j = i
  while (j > 0) && (a[j] < a[j - 1])
    swap(a[j], a[j - 1])
    j--
    //I--
```

Заметим, что каждый swap уменьшает на 1 кол-во инверсий I в массиве (инверсия - наличие такой пары a[i] и a[j], что (i < j) && (a[i] > a[j])).

Количество сравнений - O(n + I).

Количество копирований элементов (swap'ов, присваиваний и т.п.) - O(I).

• **Оптимизация** Так как мы вставляем в отсортированный массив, то можно позицию вставки искать бинпоиском (а затем "хвост"выталкивать вправо).

Количество сравнений - O(min(n + I, nlogn)).

Количество копирований элементов (swap'ов, присваиваний и т.п.) - O(I).

12.4. Теорема

Теорема 12.4.1. Не существует сортировок, использующих только сравнение «"(никакой дополнительной информации про элементы массива), работающих за o(nlogn).

Доказательство. Для простоты положим, что мы сортируем перестановку (то есть все элементы различны). Пусть наша сортировка сделала k сравнений (a[i] < a[j]), результат каждого - это 0 или 1, если False или True соответственно. Получили 2^k бинарных строк. Мы хотим уметь "отличать"друг от друга все перестановки нашего массива с помощью этих k сравнений, тогда должно выполняться $(n! \leq 2^k)$ (а иначе будут существовать 2 различных перестановки, которым соответствует одинаковый набор сравнений, а значит, мы не сможем эти две перестановки "отличить"). Логарифмируем обе части по основанию 2, получаем $\Theta(nlogn) \leq k$. То есть нам требуется хотя бы $\Theta(nlogn)$ сравнений, чтоб отсортировать массив.

12.5. Сортировки за O(nlogn) .

1. Merge sort (Сортировка слиянием)

   Спускаемся в левую и правую половину, сортируем их, сливаем две отсортированных половины.

   • **Рекурсивная реализация**

   Псевдокод:

```
1 Merge sort(l, r) // [l, r)
2   if (l + 1 >= r)
3     return
4   m = (l + r) / 2
5   Merge sort(l, m)
6   Merge sort(m, r)
7   Merge(l, m, r)  // [l,m) + [m, r) -> [l, r)
```

   Оценим время работы:

   $T(n) = 2 \cdot T(\frac{n}{2}) + O(n) = O(nlogn) + O(n) \cdot time\_for\_recursion$. Оценим дополнительную память:

   $Add\_memory = memory\_for\_merge + O(logn) \cdot memory\_for\_recursion$.

Избавимся от рекурсии.

• **Нерекурсивная реализация**

Будем поддерживать отсортированными все куски длиной 1, 2, 4, ..., $\frac{n}{2}$ на соответствующей итерации и сливать их.

Псевдокод:

```
for (k = 0; (1 << k) < n; k++)
  for (i = 0; i < n; i += 1 << (k + 1))
    Merge(i, min(i + (1 << k), n), min(i + (1 << (k + 1)), n)
```

Оценим время работы:

T(n) = O(nlogn)

Оценим дополнительную память:

Add_memory = O(n) (для слияния).

Нам достаточно двух массивов длины n. Поддерживаем куски длины 1 в a[], сливаем их в b[]. Теперь в b[] у нас куски длины 2. Сольём их в a[] и т.д.

12.5.1. Merge с подсчётом инверсий

Псевдокод:

```
Merge(an, a, bn, b, c) // [a, a + an) + [b, b + bn) -> c
  i = j = I = 0
  for k in [0, an + bn)
    if (i < an) && ((j == bn) || (a[i] <= b[j]))
      c[k] = a[i++]
    else
      c[k] = b[j++]
      I += an - i
```

2. Quick sort (Быстрая сортировка) Выбираем опорный элемент x, все элементы массива, которые <x, переносим в левую часть, все элементы массива, которые >x, переносим в правую часть, сортируем левую и правую части (partition).

Псевдокод:

```
Qsort (a)
  if (a.size() <= 1)
    return
  x = a[rand % a.size()]
  return Qsort(<x_from_a) + x * count(a, x) + Qsort(>x_from_a)
```

Оценим время работы (на перестановке):

$T(n) = \frac{\sum_{x=1}^{n-1}(T(x)+T(n-x-1))}{n} = O(nlogn)$

Лекция по алгоритмам #13
Сортировки, продолжение

28 сентября

13.1. Время работы рандомизированного QSort.

Теорема 13.1.1. *среднее время работы QSort:* $\Theta(n \log n)$

• **Доказательство #1:**

Докажем по индукции, что $T(n) \leq C \cdot n \cdot \ln n$

База : $T(2) \leq C \cdot 2 \cdot \ln 2$

Переход :

Оценим матожидание времени работы:

$T(n) \leq \frac{1}{n} \cdot \sum\limits_{x=1}^{n-1}(T(x) + T(n-x-1)) + n$

Заметим, что x и $n-x-1$ пробегают одни и те же значения.

$\Rightarrow T(n) \leq \frac{1}{n} \cdot \sum\limits_{x=1}^{n-1}(2 \cdot T(x)) + n$

$x < n \Rightarrow$ по предположению индукции $T(x) = C \cdot x \cdot \ln x$

$\Rightarrow T(n) \leq \frac{2}{n} \sum\limits_{x=1}^{n-1}(C \cdot x \cdot \ln x) + n \leq \frac{2}{n} \int\limits_{1}^{n}(C \cdot x \cdot \ln x)dx + n$

Посчитаем неопределенный интеграл:

$\int(x \ln x)dx = \frac{1}{2}x^2 \ln x - \int(\frac{1}{2}x)dx = \frac{1}{2}x^2 \ln x - \frac{1}{4}x^2$

Таким образом:

$T(n) \leq \frac{2}{n} \cdot C \cdot (\frac{1}{2}n^2 \ln n - \frac{1}{4}n^2 + \frac{1}{4}) = C \cdot (n \ln n - \frac{1}{2}n + \frac{1}{2n}) + n = Cn \ln n - \frac{Cn}{2} + \frac{C}{2n} + n$

$\lim\limits_{n \to \infty} \frac{C}{2n} = 0$

При $C = 2, n \to \infty : (-\frac{Cn}{2} + \frac{C}{2n} + n) = 0 \Rightarrow T(n) \leq 2 \cdot n \ln n$

• **Доказательство #2:**

Замечание: каждые два элемента сравниваются не более 1ого раза.

Обозначим за $sa[i]$ i'ый элемент отсортированного массива.

$T(n) =$ кол-во сравнений $(<) = \sum\limits_{i,j} \chi(i,j)$, где $\chi(i,j) = 1$, если элементы $sa[i]$ и $sa[j]$ сравнивались и $\chi(i,j) = 0$ в противном случае.

Заметим, что время работы зависит от какого-то рандома. Обозначим количество возможных исходов за α. Все исходы равновероятны. Тогда матожидание времени работы равно среднему времени по всем возможным исходам.

$T(n) = \frac{1}{\alpha} \sum\limits_{R=1}^{\alpha}(\sum\limits_{i,j}(\chi(i,j,R))) = \sum\limits_{i,j}(\frac{1}{\alpha} \sum\limits_{R=1}^{\alpha}(\chi(i,j,R))) = \sum\limits_{i,j} Pr(sa[i]$ сравнивалось с $sa[j])$, где $Pr(...)$ - вероятность события (англ. *Probability*).

Посчитаем $Pr(i,j)$:

Все сравнения происходят только с барьерным элементом (x) в Partition.

Рассмотрим момент, когда $sa[i]$ и $sa[j]$ попадают в разные ветви:

$min(sa[i], sa[j]) \le x \le max(sa[i], sa[j])$. Всего на этом отрезке $|i-j|+1$ чисел. Сравнение происходит только в случае, если $x = sa[i]$ или $x = sa[j]$. Итого $|i-j|+1$ равновероятных исхода, в 2 из которых происходит сравнение.

$Pr(i,j) = \frac{2}{|i-j|+1}$

Тогда получаем:

$T(n) = \sum\limits_{i,j}(\frac{2}{|i-j|+1}) = \Theta(n \cdot \ln n)$

13.2. Порядковая статистика.

Пусть дан массив a длиной n и число k. Задача заключается в том, чтобы найти в этом массиве k-ое по величине число, т.е. k-ую порядковую статистику.

В c++ : std::nth_element(a.begin(), a.end(), k)

Умеем получать за $\Theta(n \log n)$: $sorted(a)[k]$

Хотим за $\Theta(n)$

Главная идея: QSort с одной веткой.

Псевдокод:

```
nth_element(arr, k):
  x = arr[rand]
  l, r = partition(arr, x)

  #[0, l) - < x
  #[l, r) - = x
  #[r, len) - > x

  if k <= l:
    return nth_element(arr[0..l - 1], k)
  else if k > r:
    return nth_element(arr[r..len - 1], k - r)
  else:
    return x
```

Оценим среднее время работы(матожидание):

$T(n) = \frac{1}{n}(\sum\limits_{x=0}^{n-1} T(x)) + n \le \frac{n(n-1)}{2} \cdot \frac{C}{n} + n \le \frac{n-1}{2} \cdot C + n \le C \cdot n$

13.3. Детерменированный (почти) QSort.

IntroSort = std::sort()

QSort + HSort + ISort

Если глубина QSort $> 4n \Rightarrow HSort$

Если $n < 8 \Rightarrow ISort$

Лекция по алгоритмам #14
Сортировки

2 октября

14.1. Сортировки за $n \log n$

14.1.1. Сравнение

	Fast	Stable	Inplace	Deterministic
Quick	+	-	+	-
Merge	-	+	-	+
Heap	-	-	+	+

Для каждой сортировки существует стабильный вариант ценой дополнительной памяти. Например, можно сортировать не сами элементы, а пары {элемент, индекс}.

14.2. Сортировки, бывающие быстрее $n \log n$

14.2.1. Модифицированный HeapSort

Работает за $\mathcal{O}(n \log n)$, но бывает быстрее.

Сначала построим кучу H за $\mathcal{O}(n)$, а еще создадим кучу кандитов C (изначально содержит только минимальный элемент кучи H). Теперь на каждом шагу будем вытаскивать минимальный элемент из кучи C, вставлять на следующую позицию в отсортированом массиве и добавлять в кучу C детей текущего элемета из H.

Размер кучи C в худшем случае может быть $(n+1)/2$, но в лучшем (когда минимальные элементы в куче H лежат в порядке обхода DFS-а) он не превышает $\log n$. Следовательно, на некоторых входах можно добиться времени работы порядка $\mathcal{O}(n \log \log n)$.

14.2.2. Adaptive HeapSort

Алгоритм создан в 1992 году.

Сначала строем кучу, используя декартово дерево по неявному ключу (то есть представим каждую вершину в виде точки на плоскости, где x — значение в вершине, а y — индекс элемента). Существует алгоритм со стеком, который делает это за $\mathcal{O}(n)$. Теперь используем модифицированный HeapSort из прошлого пункта.

В некоторых случаях будет работать за $\mathcal{O}(n)$.

1. Когда вход уже отсортирован (причем в любую сторону).
2. Конкатенация k отсортированных массивов ($\mathcal{O}(n \log k)$) (на лекции было простое доказательство для $k = 2$).
3. Когда мало инверсий, ведь алгоритм работает за $\mathcal{O}(n \log \frac{\text{Inv}}{n})$, где Inv — количество инверсий.

14.3. Integer Sorting

Числа можно сортировать быстрее, потому что у них есть быстрая операция индексации и быстрая операция деления, которое монотонно (то есть если $a \le b$, то $a/x \le b/x$ для всех $x > 0$).

Замечание: все сортировки в этом разделе работают для чисел, помещающихся в машинное слово (размера 2^b, в современных компьютерах, как правило $b = 64$) и предполагают, что деление работает за $\mathcal{O}(1)$ (хотя на практике это не так).

14.3.1. CountSort

Стабильна, работает за $\mathcal{O}(n+m)$, где m такое, что $\forall i\ 0 \le a_i \le m$.

```
count = [0] * m
for e in a:
    count[e]++
for x in [0, m):
    pos[x+1] = pos[x] + count[x]
for i in [0, n):
    b[pos[a[i]]++] = a[i]
```

14.3.2. Digital (Radix) Sort

Сортируем бинарные строки длины m (числа от 0 до $2^m - 1 = M$).

Сначала [стабильно] сортируем по последнему элементу, потом по предпоследнему и т.д. Корректность легко доказать по индукции.

Время работы алгоритма $\mathcal{O}(nm) = \mathcal{O}(n \log M)$, где M — максимальное число.

Можно выполнять эту сортировку не только в двоичной системе, но и в любой другой. Тогда время работы будет $\mathcal{O}((n+k)\log_k M)$.

Теорема 14.3.1. Выгоднее всего брать $k = n$.

Доказательство. Случай 1: $k \ge n$ тогда $n + k = \mathcal{O}(k)$ $k \log_k M = \frac{k}{\log k} M$
Случай 2: $k \le n$ Тогда $n + k = \mathcal{O}(n)$ $n \log_k M = n \frac{\log M}{\log K}$ ■

14.3.3. BucketSort

Сортировка называется BucketSort, так как она раскладывает числа по ведрам (bucket — ведро). Она хороша тем, что работает за линию на любых случайных перестановках.

1. Найдем min и max за $\mathcal{O}(n)$.
2. Если min = max, все числа равны, поэтому отсортированы, выйдём.
3. Каждое x_i кладем в bucket (ведро) номер $\lceil \frac{x_i - \min}{\max - \min + 1} \cdot n \rceil$.
4. Теперь каждое ведро нужно отсортировать. Тут есть 2 варианта — рекурсивно вызвать BucketSort (BB) или использовать InsertionSort (BI). BI может быть выгоден, так как ведра маленькие.

Теорема 14.3.2. $BB \leq n \log_2 M$

Доказательство. Это верно, так как на каждом уровне рекурсии числа разбиваются минимум на 2 ведра, и на каждом уровне рекурсии чисел всего n. ■

Теорема 14.3.3. $M \leq n \Rightarrow BB = \Theta(n)$ и $BI = \Theta(n)$

Доказательство. Получается что-то очень похожее на CountSort и с таким же временем работы. ■

Теорема 14.3.4. На случайном входе и $M > n$ алгоритм работает за $\mathcal{O}(n)$

Доказательство. Время работы алгоритм $\sum k_i^2$, где k_i – размер i-го ведра. Если все числа случайные величины, порождённые равномерным распределением, то средняя величина $\sum k_i^2$ равна $\mathcal{O}(n)$. Доказательство последнего смотрите в лекции 16. ■

Лекция по алгоритмам #15
Простые алгоритмы

2 октября

15.1. Вектор – расширение и сужение.

1. Две основных характеристики вектора - size и capacity. size обозначает количество элементов в векторе, а capacity - количество выделенных ячеек памяти. Для того, чтобы не занимать много памяти и быстро работать при этом, методы PUSH и POP в векторе устроены интересным образом.

   PUSH:

```
1    if (size == capacity) {
2      capacity *= 2;
3    }
```

   POP:

```
1    if (size == capacity / 4) {
2      capacity /= 2;
3    }
```

2. Докажем, что эти методы работают за амортизированное $\mathcal{O}(1)$, найдем потенциал φ, который удовлетворяет следующим свойствам:

   (a) $a_i = t_i + \Delta\varphi$

   (b) $\begin{cases} \varphi_0 = 0 \\ \varphi_i \geq 0 \end{cases}$

   (c) PUSH : $\Delta\varphi = \begin{cases} +2 \\ -capacity \ [= size] \end{cases}$

   (d) POP : $\Delta\varphi = \begin{cases} +4 \\ -capacity \ [= 4size] \end{cases}$

   Рассмотрим функцию φ, которая будет меняться вышеописанным образом и возьмем её в качестве потенциала. Нетрудно заметить, что сохраняется условие $\varphi > 0$.

3. **Теорема 15.1.1.** Операции POP и PUSH работают в среднем за $\mathcal{O}(1)$.

   Доказательство.
   $\sum a_i = \sum t_i + \Delta\varphi$
   Заметим, что $\Delta\varphi = \varphi_{last} - \varphi_0 = (\geq 0) - 0 \geq 0$
   Таким образом, получаем $\sum a_i \geq \sum t_i$
   Тогда $a_i = \mathcal{O}(1) \Rightarrow \frac{\sum t_i}{n} = \mathcal{O}(1)$, что и требовалось доказать.

15.2. Два указателя.

Задача: находить количество различных чисел на отрезке [L, R] оффлайн за $\mathcal{O}(n\sqrt{m})$, m - количество запросов.

1. Предположим сначала, что запросы отличаются не сильно. Научимся обрабатывать запросы за $\mathcal{O}(1)$.

   L++, L−−, R++, R−−:

   Используем HashMap, $x \to count[x]$.

2. Пусть теперь $L_i \le L_{i+1}$ и $R_i \le R_{i+1}$. Научимся обрабатывать запрос за $\mathcal{O}(n+M)$.

   Храним 2 указателя L, R. При переходе к следующему отрезку применяем сколько надо раз пункт (a). Заметим, что указатели сдвинулись максимум на n. Отсюда оценка на $\mathcal{O}(n+M)$.

3. Теперь L и R всегда произвольные.

   Идея: Разделим все запросы на группы где L_i лежит в отрезках $[1,k],[(k+1),2k],\ldots$. В каждой группе упорядочим запросы по R.

   Заметим, что когда мы разбираемся с какой-то группой, R пробегает не более n, а L не более km_i, где m_i - количество запросов в i-й группе. Итого получаем $\mathcal{O}(n+km_i)$ на группу.

   Осталось просуммировать по i. Получаем $\mathcal{O}(n\frac{n}{k}+kM)$.

   Чтобы минимизировать ответ, надо взять такое k, что $n\frac{n}{k} = kM \Rightarrow k = \sqrt{\frac{n^2}{M}} = \frac{n}{\sqrt{M}}$.

   Подставляя, получаем ответ $\mathcal{O}(n\sqrt{M})$ $[+sort = \mathcal{O}(n+M)]$

15.3. k-я порядковая статистика.

Задача: по данному числу k находить k-й элемент в отсортированном массиве за $\mathcal{O}(n)$. Изначально массив не отсортирован.

1. Сразу напишем псевдокод функции.

```
1  KStat (a, L, R, k) {
2    x = a[*]
3    partition(x)
4    if (k < M1)
5      KStat(a, L, M1 - 1, k)
6    if (k > M2)
7      KStat(a, M2 + 1, R, k)
8  }
```

2. Будем искать * следующим образом: Поделим массив на подмассивы длины 5.Если n не делится на 5, заполним до делящегося произвольными элементами массива. Отсортируем подмассивы и рассмотрим массив их медиан. Его медиана, найденная применением рекурсивно этого алгоритма, и будет *. $n < 5$ - база рекурсии, перебираем руками.

Заметим, подмассивов у нас $n/5$, а среди них таких, что их медиана меньше *, следовательно, $n/10$. Тогда элементов в исходном массиве, меньших * хотя бы $n/10 * 3$. Хотя бы столько же и элементов больших *. Следовательно, рекурсивно (после partition(x)) алгоритм получит максимум $7n/10$ элементов.

3. $T(n) \leq T(n/5) + T(7n/10) + n = \Theta(n)$.

   Если рассмотреть дерево рекурсии, то на каждом уровне сумма коэффициентов рекурсивных вызовов не больше чем $(n/5 + 7n/10 < 1) \cdot a$, где a - сумма на предыдущем уровне. Получаем ограничение сверху как сумму геометрической прогрессии с коэффициентов меньше единицы. Значит, ответ $\Theta(n)$.

Лекция по алгоритмам #16
Kirkpatrick's Sort(1984)

5 октября

16.1. Время работы Bucket Insertion Sort(BI)

Доказательство. Т.к. мы разделяем весь массив на Backets, а затем сортируем каждый backet отдельно. Тогда суммарное время сортировки $n + \sum k_i^2$, где k_i^2 - размер i-го backet. ■

$\sum k_i^2 = \sum_{1 \leqslant i,j \leqslant n} IS(i,j)$, IS узнает лежат ли i, j в одном Backet
Обозначения:
Def 1: E - среднее или мат-ожидание
Def 2: Pr - вероятность (Probabilety)
$E(\sum k_i^2) = E(\sum_{1 \leqslant i,j \leqslant n} IS(i,j)) = \Pr(IS(i,j)) = 1 \cdot \frac{1}{n} + \frac{1}{n} \cdot n \cdot (n-1) = 2 \cdot n - 1$
Реализация:
1)Min, Max
2)$x_i \longrightarrow \lfloor \frac{x_i - Min}{Max - MIn + 1} \cdot n \rfloor$

16.2. Kirkpatrick's Sort (1984)

Задача: Имеется N целых чисел от 0 до $M-1$, необходимо отсортировать их за $\mathcal{O}(N \log_2 \log_2 M)$
Идея: У нас уже есть RadixSort, которая работает за $\mathcal{O}(N \log_2 M)$

Для такого спуска необходимо совершать рекурсивные вызовы.
a_i и b_i - 2 половины массива знаков, n - количество чисел, k - длина чисел(для удобства будет считать, что $k = 2^s$, если не хватает, добавить 0 в начало)

Псевдокод:

```
1 Sort(n, k, <a_i, b_i>)
2   if(k <= log_2(n)) {
3     CountSort
4     return
5   }
6   Sort(n, k/2, b_i)
7   Sort(n, k/2, a_i)
```

Оптимизация: Вместо 2-х сортировок сделаем одну.
Для ассимптотики берем n, т.к. k не меняется.
$T(k) = 2T(\frac{k}{2}) + n = n \cdot \log_2 M$
$\downarrow$
$T(k) = T(\frac{k}{2}) + n = n \cdot \log_2 k = n \cdot \log_2 \log_2 M$

1) $List[a_i] \leftarrow HashTable$ (Для каждого a_i - список b_i) $//\theta(n)$

```
1 for(int i = 0; i < n; i++)
2   List[a_i].push_back(b_i)
```

2)List[a_i].Find & Del Max Element (Для каждой находим и удаляем) $//\Theta(n)$
$\Rightarrow$ Кол-во списков(t) + Суммарное Длина(n - t) = n (итого n)
3)Рекурсивный вызов: Передаём сразу пару чисел $< b_i, a_i >$(n - t) или $< a_i, IND >$ (t) - 2 вида пар(IND - число которое не может точно попасться, для отличия пар)
Сортируем только по 1-й половине. Первые половины ключи.
4)Результат рекурсии
1.Sorted(a_i)
2.Sorted List[a_i](список отсортирован) - списоу без MAX
5)Sorted List[a_i] += Max[a_i] (сливаем с MAX)
6)

```
1 for a in Sorted(a_i)
2   Result += Sorted List[a]
```

Лекция по алгоритмам #17
Inplace алгоритмы

5 Октября

17.1. Уже известные.

1. QuickSort.
2. HeapSort.
3. SelectionSort.
4. Partition.
5. nth_element.
6. binary_search.
7. lower_bound.

17.2. Reverse.

Функция, разворачивающая массив задом наперед.

```
1  void reverse(int *a, int n)
2  {
3    int tmp;
4    for (int i = 0; i <= n >> 1; i++)
5    {
6      tmp = a[i];
7      a[i] = a[n - i - 1];
8      a[n - i - 1] = tmp;
9    }
10   return;
11 }
```

17.3. rotate.

17.3.1. 1 способ. 3 reverse

Пусть есть массив a длины n и нужно получить его циклический сдвиг на k. Это значит, что нужно, чтобы первые k элементов оказались в конце массива, сохранив порядок внутри, а остальные, соответственно, в начале, также сохранив порядок внутри. Для этого достаточно сделать reverse(a, k), reverse(a + k, n - k), reverse(a, n), так как после reverse(a, n) первые k элементов окажутся в конце, но при этом обе части массива - с n - k до n и с 0 до n - k - окажутся развернутыми.

```
1  void rotate(int *a, int n, int k)
2  {
3    reverse(a, k);
4    reverse(a + k, n - k);
5    reverse(a, n);
```

```
6   return;
7 }
```

17.3.2. Способ 2.

Пусть k = 1. Тогда можно запомнить первый элемент, затем положить по очереди во все элементы, кроме последнего, значение следующего элемента, затем присвоить последнему элементу запомненное значение. Примечание: пока k = 1, параметрps start и k в функции не нужны. Зато потом пригодятся.

```
1  void rotateone(int *a, int n, int k = 1, int start = 0)
2  {
3    int i = start, i1;
4    int tmp = *a;
5    do
6    {
7      i1 = i;
8      i += k;
9      if (i > n)
10       i -= n;
11     a[i1] = a[i];
12   }
13   while (i != start);
14   a[i1] = tmp;
15 }
```

Теперь избавимся от ограничения k = 1. Заметим, что все элементы массива разобьются на g = gcd(n, k) циклов, причем первые g элементов лежат в разных циклах. Также заметим, что внутри цикла произвести сдвиг можно, используя функцию rotateone.

```
1  void rotate(int *a, int n, int k)
2  {
3    g = gcd(n, k);
4    for (int i = 0; i < g; i++)
5    {
6      rotateone(a, n, k, i);
7    }
8  }
```

Также, в C++ реализована функция rotate, ее код можно посмотреть, при желании.

17.4. unique, intersect_sets, diff_sets

Первая функция принимает на вход мультимножество в виде отсортированного массива и складывает в начало массива все неповторяющиеся элементы мультимножества, также в отсортированном порядке. Поймем, что в отсортированном массиве одинаковые элементы могут идти только подряд. Тогда код unique становится очевиден:

```
1  void unique(int *a, int n)
2  {
3    int k = 0;
4    for (int i = 0; i < n; i++)
5      if (i == 0 || a[i] != a[i - 1])
6        a[k++] = a[i];
7    return;
```

```
8 }
```

intersect_sets и diff_sets реализуются аналогично.

17.5. merge

Слияние отсортированных массивов.

17.5.1. Пара общих идей

Во-первых, для всех следующих алгоритмов требуется, чтобы два сливаемых массива были записаны подряд (то есть nums[0:n] = A, nums[n:n + k] = B). Во-вторых, два подряд идущих массива можно поменять местами, используя rotate.

17.5.2. $O(N \log N)$, Stable

Предположим, что $|B| >= |A|$ (Если нет, их можно поменять местами за линейное время). Рассмотрим элемент $x = B[\frac{|B|}{2}]$. Каждый массив можно разделить на три части - элементы, меньшие x (A_1, B_1), элементы равные x (A_2, B_2) и элементы большие x (A_3, B_3). Расположим эти части в порядке $A_1B_1A_2B_2A_3B_3$, используя rotate. Теперь рекурсивно запустимся, чтобы слить A_1 с B_1 и A_3 с B_3. Теперь, если обозначить |A| + |B| за N, то время работы можно описать так: $T(N) = T(X) + T(Y) + O(N)$, то есть $O(N \log N)$, так как $X + Y \leq N$ и $max(X, Y) \leq \frac{3N}{4}$, так как хотя бы половина массива B больше x, а |B| - хотя бы $\frac{N}{2}$. Примечание: так как мы используем рекурсию глубиной $\log N$, мы таки используем $(\log N)$ памяти.

17.5.3. O(N), Unstable

Сначала предположим, что все элементы различны, а также, что существует k такое что $N = k^2$, а также $|B| \vdots k$ и $|A| \vdots k$. Тогда весь массив можно разделить на куски длины k. Пронумеруем для удобства куски массива A и куски массива B отдельно с единицы. Нулевым куском назовем пустой массив. Отсортируем их с помощью SelectionSort по первому элементу (Сложность - $O(\sqrt{N}^2) = O(N)$, но при этом только $O(\sqrt{N})$ операций копирования, каждая из которых делается за $O(\sqrt{N})$. Теперь достаточно поочередно слить соседние куски (Сначала первый со вторым, затем третий с тем, что лежит на месте второго и так далее). Доказательство: Пусть i и j - максимальные числа такие, что A_i и B_j уже подвергались слиянию (изначально одно из чисел равно 1, второе 0). Тогда так как массивы A и B изначально были отсортированы, все A_k для k < i и $_k$ для k < j также уже подвергались слиянию. Более того, среди всех элементов, подвергавшихся слиянию, первые (i + j - 1) * k уже стоят на своих местах. Пусть первый неслитый кусок - B_{j+1}. Тогда среди уже слитых элементов есть k * j элементов, меньших минимума из B_{j+1} - элементы B_k для k < j + 1. Также, поскольку куски были отсортированы по первым элементам, а также все куски внутри отсортированы, минимум из B_{j+1} больше, чем минимум из A_i, который, в свою очередь, больше всех элементов $_k$ для k < i. Значит, не более k элементов из начала стоят не на своих местах, причем это последние k элементов. Заметим, что на этом этапе алгоритм использует обычный merge за O(n), не inplace. Чтобы избавиться от этого, заметим, что последние 2 * k элементов можно использовать как буфер для merge, так как можно не копировать элементы, а менять местами. Именно в этот момент алгоритм перестает быть стабильным, так как элементы из буфера возвращаются туда в случайном порядке. Также, теперь последние 2 куска мы не сливаем с предыдущими. Теперь мы в конце просто сортируем SelectionSort-ом последние 3 * k элементов, что все равно работает за O(N).

Теперь избавимся от предположения, что N - точный квадрат. Теперь, если взять k максимальное такое, что его квадрат не больше N, и снова поделим массивы A и B на куски длины k, то получится, что в конце каждого массива остался "хвост". Воспользовавшись функцией rotate, перенесем оба "хвоста"в конец, а для остальной части массивов применим описанный выше алгоритм. Теперь рассмотрим суммарную длину "хвостов". Она небольшая ($(x + y) < 2 * k$), где x и y - длины "хвостов". Значит, можно просто:

1. поменять местами последние x + y элементов с предыдущими x + y элементами. Теперь последние x + y элементов отстоят от своих позиций в отсортированном массиве не более чем на 2 * k.
2. Отсортировать 2 * k элементов, лежащих перед новым "хвостом".
3. Использовать описанный выше merge для слияния начала массива с последними 2 * k элементами перед новым "хвостом".
4. Теперь можно просто отсортировать SelectionSort-ом последние 2 * k + x + y элементов.

Лекция по алгоритмам #18
Кучи

9 октября

18.1. k-heap

k-ичная куча (k-heap, d-heap) - структура данных в виде дерева, в котором в каждой вершине хранится число. Для этого дерева всегда выполняются условия:

1. в каждой вершине записан минимум в своём поддереве;

2. каждая вершина либо является листом, либо имеет ровно k сыновей.

k-ичную кучу можно хранить в массиве. 0-й элемент - корень, элементы с номерами $ki + 1, ki + 2, \ldots, ki + k$ - сыновья вершины i, элемент с номером $\lfloor \frac{i-1}{k} \rfloor$ - отец вершины i. Пусть для удобства реализации в несуществующих элементах лежит бесконечность.

В качестве k используется обычно степень двойки, т. к. в таком случае используемые при переходе по рёбрам дерева операции умножения и деления индексов сводятся к более быстрой операции - битовому сдвигу.

SiftUp и SiftDown

Операции SiftUp и SiftDown, как и для двоичной кучи, просеивают вершину вверх/вниз до тех пор, пока она не займёт корректную позицию. Оценим время работы.

SiftUp и SiftDown в худшем случае пройдут по всей высоте дерева. Высота дерева из n вершин равна $\lceil \log_k((k-1)n + 1) \rceil$, поэтому SiftUp работает за $\mathcal{O}(\log n)$. SiftDown на каждой своей итерации делает k сравнений. Поэтому SiftDown работает за $\mathcal{O}(k \log n)$.

Имея операции SiftUp и SiftDown, мы можем реализовать операции Add и ExtractMin.

```
1  sift_up(i)
2    while i > 0 && h[(i - 1) / k] > h[i]
3      swap(h[(i - 1) / k], h[i])
4      i = (i - 1) / k
5
6  sift_down(i)
7    while true
8      m = min_index(k * i + 1, ..., k * i + k)
9      if h[i] <= h[m] break
10     swap(h[i], h[m])
11     i = m
12
13 add(value)
14   h[end_heap] = value
15   sift_up(end_heap++)
16
17 extract_min()
18   res = h[0]
19   h[0] = inf
20   swap(h[0], h[--end_heap])
21   sift_down(0)
22   return res
```

18.2. MinMax heap

Пусть из двоичной кучи нам нужно уметь извлекать и максимум, и минимум. Можно использовать две кучи, в одной поддержка максимума, в другой - минимума, где каждый элемент имеет ссылку на него же в другой куче. Теперь сэкономим память: объединим в одну кучу.

На нулевом уровне (в корне) будем хранить минимум в дереве.

На первом уровне будем хранить максимумы в соответствующих поддеревьях.

На втором уровне будем хранить минимумы в соответствующих поддеревьях.

На третьем уровне будем хранить максимумы в соответствующих поддеревьях.

И так далее

SiftUp

Предположим, что вершина v, которую нам нужно поднять, находится на уровне минимумов (противоположный случай аналогичен).

Тогда отец v находится на уровне максимумов. Возможны следующие ситуации:

1. значение у v больше, чем у отца: меняем их местами и делаем для v из нового места обычный SiftUp с шагом $i = i/4$.

2. иначе делаем для v обычный SiftUp с шагом $i = i/4$.

Время работы - $\mathcal{O}(\frac{\log n}{2}) = \mathcal{O}(\log n)$.

SiftDown

Предположим, что вершина v, которую нам нужно спустить, находится на уровне минимумов (противоположный случай аналогичен).

Тогда дети v находятся на уровне максимумов. Возможны следующие ситуации:

1. среди внуков v есть те, что меньше v. Тогда найдём наименьшего внука v и поменяем его местами с v. Осталось проверить - если на новом месте v конфликтует со своим отцом, поменять их местами. Продолжаем SiftDown из места, где первоначально был наименьший внук v.

2. у v нет внуков. Тогда вручную обеспечим корректность для v и её детей.

На каждой итерации выполняется 5 сравнений. Время работы - $\mathcal{O}(\frac{5}{2}\log n) = \mathcal{O}(\log n)$.

MinMax кучу можно построить за линейное время inplace аналогично двоичной куче.

18.3. Leftist heap, skew heap

Пусть в дереве каждая вершина имеет степень 0, 1 или 2. Введём для каждой вершины функцию $d(v)$ - расстояние вниз до ближайшего отсутствия вершины.

Лемма: $d(v) \le \log_2(size + 1)$.

Доказательство. Очевидно, что $size \ge 2^{d(v)} - 1 \Leftrightarrow size + 1 \ge 2^{d(v)} \Leftrightarrow d(v) \le \log_2(size + 1)$.

Назовём левацкой кучей (leftist heap) структуру данных в виде дерева, в котором в каждой вершине хранится число. Для этого дерева выполняются условия:

1. в каждой вершине записан минимум в своём поддереве;

2. каждая вершина имеет степень 0, 1 или 2;

3. для каждой вершины $d(left(v)) \ge d(right(v))$. Для несуществующей вершины d примем равным нулю.

От предыдущих куч эта куча отличается возможностью быстрого слияния. Пусть EMPTY - пустая вершина: $left$ и $right$ указывают на неё же, $x = \infty$, $d = 0$.

```
1  Merge(a, b) {
2    if (a->x >= b->x)
3      swap(a, b);
4    if (b == EMPTY)
5      return a;
6    a->right = Merge(a->right, b);
7    if (a->right->d > a->left->d)
8      swap(a->right, a->left);
9    a->d = a->right->d + 1;
10   return a;
11 }
```

Время работы: на каждом шаге рекурсии $a \to d + b \to d$ уменьшается, поэтому:
$Time \leq a \to d + b \to d = \mathcal{O}(\log n)$.

Скошенная куча (Skew heap)

Уберём условие $d(left(v)) \geq d(right(v))$. В функции Merge уберём 7 и 9 строки. Полученная куча называется skew heap. Merge в ней работает за амортизированное $\mathcal{O}(\log n)$, доказательство в следующей лекции.

Лекция по алгоритмам #19
Кучи

9 октября

19.1. Доказательство Skew Heap.

Назовем ребенка вершины тяжелым, если размер его поддерева строго больше половины размера поддерева вершины.

Пусть ϕ - количество правых тяжелых детей. Заметим, что у каждой вершины не более одного тяжелого ребенка.

Посчитаем, сколько раз мы спустились в тяжелого правого сына. Наш путь мог проходить через легкие и тяжелые вершины. Заметим, что легких вершин на пути могло быть не больше $\log n$, так как при переходе в эту вершину размер поддерева уменьшается хотя бы в 2 раза.

Пусть тяжелых ребер на пути было k. Тогда $\Delta\phi = \mathcal{O}(\log n) - k$, так как каждый раз, когда мы спускаемся в тяжелого ребенка и merge-им его с чем-то, оно остается тяжелым, а, значит, когда мы сделаем swap, правый ребенок станет легким, и количество правых тяжелых детей уменьшится на 1.

$a_i = t_i + \Delta\phi = \mathcal{O}(\log n) + k + \mathcal{O}(\log n) - k = \mathcal{O}(\log n)$
$\phi_n - \phi_0 \leq n$

19.2. Leftist*

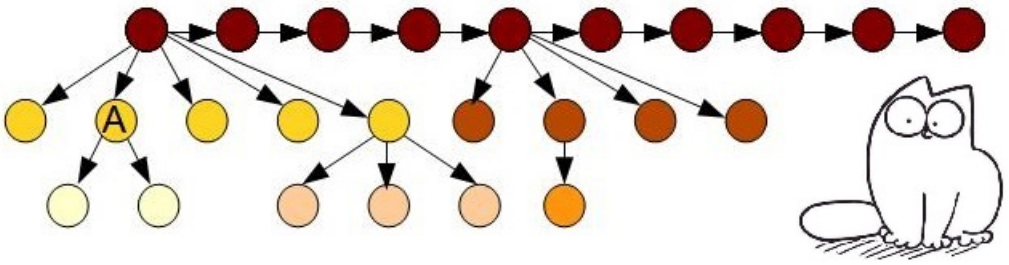
Заметим, что мы можем избавиться от условия $\forall v : d(l(v)) \geq d(r(v))$. Вместо этого будем спускаться в ребенка с меньшим d.

```
1    if (a->r->d < a->l->d)
2    a->r = Merge(a->r, b);
3    else
4    a->l = Merge(a->l, b);
```

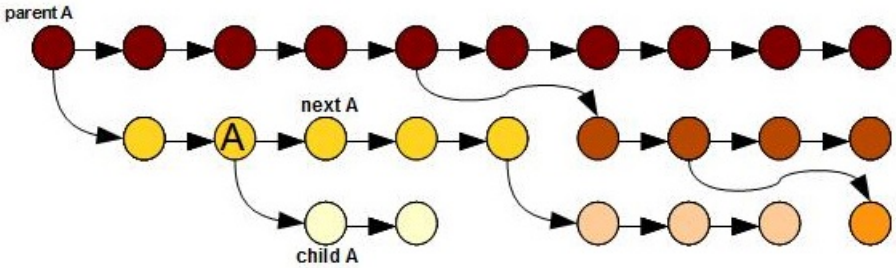
19.3. Pairng Heap.

Pairing Heap - структура данных, которая умеет делать Add, Merge и ExtractMin за $\mathcal{O}(\log n)$, а DecKey за $o(\log n)$.

Pairng Heap представляет из себя список корней нескольких деревьев, для каждого из которых выполняется свойство кучи, то есть значение в вершине не больше значения в ребенке.



Для каждой вершины будем хранить список ее детей. Для этого будем хранить указатель на первого ребенка и для каждого ребенка указатель на следующего в списке. Также для каждой вершины будем хранить указатель на ее отца.



Тогда Merge работает за $\mathcal{O}(1)$. Для этого мы просто добавляем в конец первого списка корней второй.

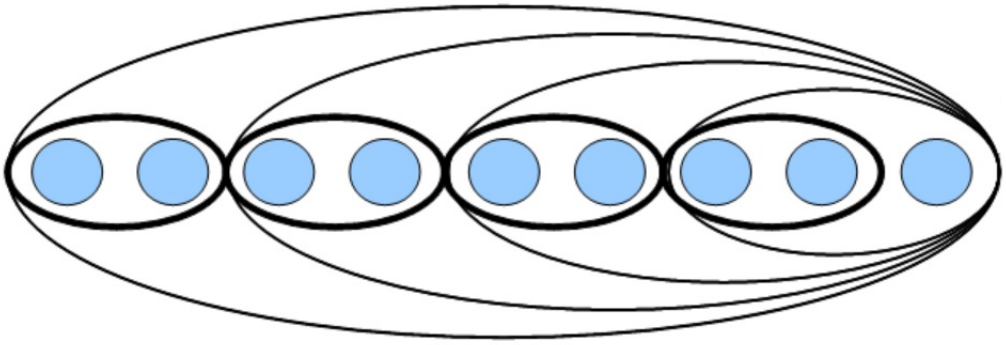
Add также работает за $\mathcal{O}(1)$. Для этого мы должны сделать merge имеющегося списка деревьев с новой вершиной.

DecKey тоже работает за $\mathcal{O}(1)$. Мы фиктивно удаляем вершину из списка детей ее отца и добавляем ее в список корней.

ExtractMin работает за k (длина списка корней) + Merge(Root, Min->child) + pairing.

Напишем функцию *Pairing*.

Эта функция получает список корней и переподвешивает вершины так, чтобы остался один корень, и при этом сохранилось свойство кучи. Для этого она сначала разбивает корни на пары, объединяет поддеревья в каждой паре, а затем по очереди объединяет все получившееся деревья в одно.



Для начала напишем функцию Pair(a, b), которая получает 2 дерева, берет из них то, что с меньшим ключом, и второе привешивает к нему.

```
1     Pair(a, b) {
2       if (a->x > b->x) swap(a, b);
3       b->Next = a->Child
4       a->Child = b
5     }
```

Теперь напишем саму функцию *Pairing*. Сначала запускаем *Pair* от двух корней. Потом рекурсивно получаем корень дерева, являющегося объединением остальных корней. А затем запускаем *Pair* от двух получившихся корней и да будет нам радость.

```
1     Pairing(a) {
2       if |a| = 1 return a[0];
3       if |a| = 2 return Pair(a[0], a[1]);
4       return Pair(Pair(a[0], a[1]), Pairing(a[2:]));
5     }
```

19.4. Van Emde Boas tree.

Пусть теперь все ключи лежат в диапазоне [0..C)
Разобьем все элементы на куски по $\sqrt{C}$.
i попадает в $[i/\sqrt{C}]$ кусок.

Назовем нашу структуру Heap.
1) Храним Min.
2) Для каждого i, $HashMap[i] = Heap(i)$.
3) Храним $index$ - $Heap$ номеров непустых кусов.

Мы отдельно обрабатываем случай, когда куча состоит из одного элемента. В этом случае мы не храним $HashMap$ и $index$, а храним только Min.

```
1  struct Heap {
2    int Min;
3    Heap *index;
4    unordered_map<int, Heap*> HashMap;
5    int ExtractMin();
6    void Add(int x);
7  };
```

Теперь напишем методы Add и ExtractMin.

```
1  Add(x, c) {
2    if EMPTY return Heap(x);
3    Heap* h = HashMap[x / sqrt(c)];
4    if (h == EMPTY) index.Add(x / sqrt(c), sqrt(c));
5    h->Add(x, sqrt(c));
6  }
7  ExtractMin() {
8    Heap* h = HashMap[index->Min];
9    h->ExtractMin();
10   if (h == EMPTY) index->ExtractMin();
11   Min = HashMap[index->Min].Min;
12   return Min;
13 }
```

Лекция по алгоритмам #20
Биномиальная куча

17 октября

20.1. Определения

Определим понятие "биномиальное дерево"рекурсивно.

Def 20.1.1. *Биномиальным деревом ранга* 0 *или* T_0 *будем называть одну вершину. Биномиальным деревом ранга* $n+1$ *или* T_{n+1} *будем называть дерево* T_n*, к корню которого подвесили еще одно дерево* T_n *(порядок следования детей не важен). При этом биномиальное дерево должно удовлетворять свойству кучи (значение в вершине не больше значения в предках).*

Выпишем несколько простых свойств биномиальных деревьев.

Lm 20.1.2. $|T_n| = 2^n$

Доказательство. Индукция по рангу дерева (далее эта фраза и проверка базы индукции будет опускаться). Для $n = 0$, действительно, дерево состоит из одной вершины. Тогда $|T_{n+1}| = |T_n| + |T_n| = 2^n + 2^n = 2^{n+1}$. ■

Lm 20.1.3. $degRoot(T_n) = n$

Доказательство. $degRoot(T_{n+1}) = degRoot(T_n) + 1 = n + 1$ ■

Lm 20.1.4. Сыновьями T_n являются деревья $T_0, T_1, .., T_{n-1}$.

Доказательство. К сыновьям T_n добавляется еще одно дерево T_n, поэтому для T_{n+1} сыновьями будут старые деревья $T_0, .., T_{n-1}$ и новый T_n. ■

Lm 20.1.5. $Depth(T_n) = n$

Доказательство. $Depth(T_{n+1}) = max(Depth(T_n), 1 + Depth(T_n)) = 1 + Depth(T_n) = 1 + n$ ■

Теперь определим понятие "биномиальная куча".

Def 20.1.6. *Биномиальная куча - список биномиальных деревьев различного ранга.*

20.2. Реализация функций для биномиальной кучи

20.2.1. Как хранить?

Мы будем хранить каждую вершинку кучи как структуру, у которой есть поля $next, child, x, rank$. Последние два параметра - это значение, хранимое в вершине и ранг дерева, соответсвующего наше вершине. Вместо того чтобы хранить список всех детей, мы будем хранить только указатель на первого ребенка (поле *child*). Также будем хранить указатель на следующую вершину-соседа (поле *next*). Таким образом, чтобы проитерироваться по детям вершины, надо взять вершину по указателю *child*, и переходить по указателям *next*, пока дети не закончатся.

```
struct Node{
  Node *next, *child;
  int x, rank;
};
```

20.2.2. Append

Пусть a - это первая вершина списка детей какой-то вершины (например, p). Эта функция добавляет в список детей вершины p вершину b (гарантируется, что b не лежит ни в чьем списке детей).

```
void Append(Node *&a, Node *b){
  b -> next = a;
  a = b;
}
```

Как это работает?

Вначале мы говорим, что у вершины b есть новый сосед - вершина a. Теперь b - это указатель на начало списка из вершины b и детей p. Присвоение в a значения b нужно, чтобы a оставалась вершиной-началом списка детей p. Мы передаем a в функцию через &, что гарантирует, что все измения с a также произойдут и с переменной, от которой вызывали функцию.

20.2.3. Join

Эта функция подвешивает дерево T_n к дереву T_n (очень важно, что они одинаковых рангов!), сохраняя свойство кучи.

```
Node *Join(Node *a, Node *b){
  if (a -> x > b -> x)
    swap(a, b);
  Append(a -> child, b);
  (a -> rank)++;
  return a;
}
```

Как это работает? В данной функции a и b - это корни (на самом деле указатели на корни, но я буду опускать это слово) двух деревьев. После операции *swap* в a будет корень дерева с меньшим значением. После мы добавляем b в список детей a с помощью уже имеющейся функции *Append* и увеличиваем ранг a на единицу.

20.2.4. ExtractMin

Как и любая куча, биномиальная умеет брать минимальное значение среди содержащихся в ней элементов. Данная функция принимает один параметр - указатель на начало списка деревьев, среди которых мы хотим найти (и удалить) минимум, а возвращает пару из указателя на новое начало списка и, собственно, самого минимума.

```
pair<Node *, int> ExtractMin(Node *a){
  Node **m = &a;
  for (Node **p = &a; *p; p = &((*p) -> next)){
    if ((*p) -> x < (*m) -> x)
      m = p;
  Node *res = *m;
```

```
7   *m = (*m) -> next;
8   return <UnionHeaps(a, res -> child), res -> x>;
9 }
```

Как это работает?

Что нам надо сделать? Нам надо пробежать по всем деревьям, и так как для любого дерева выполнено свойство кучи, достаточно рассмотреть значение в его корне. Далее надо взять минимальное из рассмотренных значений, и удалить вершину, которой соответствует этот минимум. В нашей функции будем поддерживать переменную m такую, что $m = \&(t\text{->}next)$, где t - это вершина, для которой сосед справа и есть минимум. Другими словами, в переменной m хранится как-бы указатель на вершину, которая является минимумом, но хранится он как адрес в памяти, где лежит значение $t\text{->}next$ (t - это левый сосед минимума), поэтому когда мы возьмем $(*m)$, то получим $(*(\&(t\text{->}next))) = t\text{->}next$, то есть минимум, что и требовалось. Переменная p - это то же, что и m, только там хранится не минимум, а текущее дерево. Цикл for в коде - это цикл, перебирающий корни всех деревьев ($*p$ - текущий корень). Обновление ответа в цикле - обычное сравнение. Далее в res сохраняется наш минимум. Пото мы делаем $*m = (*m)\text{->}next$, что равносильно $*(\&(t\text{->}next)) = (*(\&(t\text{->}next)))\text{->}next \Leftrightarrow t\text{->}next = t\text{->}next\text{->}next$. В результате теперь t ссылается не на m, а на вершину, на которую раньше ссылался ($t\text{->}next$), то есть наш минимум. Таким образом мы удалили минимум из списка корней деревьев, оставив все связи корректными. В итоге у нас остался список детей, оставшихся от минимума, которые надо прилепить к нашей куче. Это делает функция $UnionHeaps$, которая объединяет две кучи. О ней и пойдет речь в следующей главе.

P.S. Удаление минимума можно делать проще (в плане понимания), если хранить не только правого соседа, но и левого. Минус в том, что при таком подходе код получиться более громоздким.

20.2.5. UnionHeaps

Принимает на вход принимает указатели на кучи (на их начала), возвращает указатель на новую кучу. Для этого заведем массив векторов v, и в $v[i]$ будем хранить все деревья ранга i из обеих куч. Данная функция сначала заполняет массив v, а потом вызывает функцию $getHeap$, которая по массиву v строит новую кучу.

```
1 Node* UnionHeaps(Node *a, Node *b){
2   Have(a);
3   Have(b);
4   return getHeap();
5 }
```

Замечание 1. Обнулять массив v не надо, так как наша куча будет устроена так, что к моменту вызова $UnionHeaps$ массив v и так будет пуст. Замечание 2. Просто приписать один список корней к другому нельзя, так как по определению кучи, все деревья должны быть разных рангов.

20.2.6. Have

Функция, распихивающая деревья по векторам, соответсвующим их рангам.

```
1 void Have(Node *a){
2   for (;a;a = a -> next)
3     v[a -> rank].push_back(a)
4 }
```

20.2.7. getHeap

Как уже было сказано, по массиву v строит новую кучу. Для этого делаем следующее. Будем идти по списку деревьев от меньших рангов к большим. Если в данный момент мы встретили два дерева одинакого ранга, то сольем их в одно дерево с помощью функции $join$. При этом новое дерево будет иметь ранг, на один больше, чем у деревьев, из которых оно было получено. Добавим его в соответсвующий вектор в массиве v.

Можно заметить, что из деревьев одного ранга получаются деревья большего ранга, поэтому если мы закончили обработку вектора v_i, то для всех элементов массива от 0 до i уже все корректно и не испортится. В итоге мы получим массив v, в котором не может быть деревьев одинакового ранга (иначе мы бы применили к ним $join$), а следовательно, получили корректную биномиальную кучу.

```
k = log_2 (N+1);
node *getHeap(){
  node *res = NULL;
  for (int i = 0; i < k; i++){
    while ((int)v[i].size() >= 2){
      v[i + 1].push_back(join(v[i][(int)v[i].size() - 1], v[i][(int)v[i].size() -
          2]));
      v[i].pop_back();
      v[i].pop_back();
    }
    if ((int)v[i].size()){
      if (res == NULL)
        res = v[i][0];
      else
        res -> next = v[i][0];
      v[i].pop_back();
    }
  }
  return res;
}
```

Здесь N - максимальное количество вершин, которое может быть в куче. Так как размер T_n это 2^n, то ранг дерева можно оценить сверху двоичным логарифмом от количества вершин. Собственно, k - это и есть максимальный ранг, который может встретиться в куче. Еще можно заметить, после завершения итерации цикла вектор v_i становится пустым, то есть в конце массив v полностью опустеет.

Комментарий по коду. Мы перебираем ранги от меньших к большим. Когда мы стоим на ранге i, мы делаем $join$ двух деревьев этого ранга пока можем (для удобства все время берем два последних дерева). Если в конце осталось одно дерево ранга i, то добавим его в ответ.

20.3. Общие слова про стандартную реализацию биномильной кучи

Такая реализация кучи не дает никаких преимуществ по сравнению с ранее изученными. Основные операции $ExtractMin$ и Add (добавление одного элемента, реализуется как $UnionHeaps$ от кучи и нового элемента) используют в себе функцию $UnionHeaps$, которая работает за $O(\log n)$ (доказательство будет ниже), поэтому в такой реализациии биномиальная куча не

представляет особого интереса.

20.4. Быстрый Add или как сломать биномиальную кучу, чтобы она заработала быстрей

Переделаем наше определение биномаильной кучи, разрешив ситуацию существования нескольких куч одного ранга.

Сделаем новую функцию $merge$, которая сливает две кучи. Для этого будем хранить кучу не как указатель на начало, а как пара указателей: на начало списка и на конец. Теперь написать $merge$ легко - нам надо прицепить начало одной кучи к концу другой. Такой $merge$ работает за $O(1)$. Add (добавление одного элемента) выражается через $merge$ (сливаем старую кучу и новый элемент (на самом деле кучу из одного дерева T_0)). Поэтому Add теперь тоже работает за $O(1)$. При этом в $ExtractMin$ все еще будем вызывать $UnionHeaps$.

20.5. Время работы новой реализации с быстрым Add

Покажем, что несмотря на то что Add стал работать быстрее, $ExtractMin$ все еще работает за амортизированный $O(\log n)$.

Пусть в массиве v лежит всего α деревьев. Покажем, что тогда суммарное время работы всех $join$ при выполнении $getHeap$ это $O(\alpha)$ с помощью амортизационного анализа.

Пусть $\varphi = v.size()$, где $v.size()$ - это суммарное количество деревьев в массиве v. Тогда рассмотрим, что происходит при операции $join$. Он работает за $O(1)$, поэтому считаем $t_i = 1$. При этом два дерева удалились, и появилось одно новое, то есть $\Delta\varphi = -1$, следовательно, $a_i = 0$. Суммарно потенциал изменился ровно на $v.size()$, так как массив v полностью опустел. Итого $T = A - (\varphi_{end} - \varphi_{beg}) \leqslant 0 - (0 - v.size()) = \alpha$.

Теперь с помощью амортизационного анализа оценим время работы $ExtractMin$. Пусть на момент вызова $ExtractMin$ в куче было α деревьев. Возьмем $\varphi = 3roots$, где $roots$ - количество деревьев в куче. Заметим, что всего за $ExtractMin$ потенциал изменился с $3roots$ до $3\log n$ (или меньше), так как в новой куче будет не более $\log n$ деревьев.

Оценим реальное время работы. $ExtractMin$ работает за пробег по массиву корней, то есть за α. $Have$ пробегает по первому списку корней (это α), а потом по списку детей удаленного минимума (не более чем $\log n$, так как в дереве степень вершины не может больше $\log n$, так как у вершины только различные сыновья, то размер самого большого это 2^{rank}, поэтому если есть сын с рангом, большим, чем log, то размер этого сына больше n). $getHeap$ работает за $\log n$(итерация по всем степеням) $+ \alpha$ на все $join$. Получаем, что $a_i = t_i + \Delta\varphi = 3\alpha + 2\log n + (3\log n - 3\alpha) = O(\log n)$. Суммарное изменение потенциала это $O(n)$ (количество корней не может измениться больше, чем на n). Итого получили амортизационный логарифм на $ExtractMin$.

Лекция по алгоритмам #21
Фибоначчиева куча

12 котября

21.1. От биномиальных деревьев к фибоначчиевым

В предыдущей части мы научились делать биномиальную кучу с `Add` и `Merge` за $\mathcal{O}(1)$ и `ExtractMin` за амортизированное $\mathcal{O}(\log n)$. Немного расширим наши представления о ней, чтобы суметь сделать и уменьшение ключа за константу.

Def 21.1.1. *Неформально говоря, фибоначчиево дерево — обрезанное биномиальное дерево. Определим формально:*

- *Фибоначчиево дерево ранга* 0, F_0, *— просто вершина.*
- *Фибоначчиево дерево ранга* n, F_n $(n > 0)$, *— дерево, сыновьями корня которого являются* n *или* $n-1$ *фибоначчиевых деревьев меньших рангов, причем все ранги сыновей попарно различны.*
- *Естественно, фибоначчиево дерево удовлетворяет свойству кучи.*

Def 21.1.2. *Фиббоначиева куча — список фибоначчиевых деревьев.*

В фибоначчиевом дереве дети вершины не обязательно упорядочены по рангу. Для того, чтобы в дальнейшем можно было удалять у вершины сына, будем хранить детей в двусвязном списке.

Все операции, кроме удаления и уменьшения ключа элемента, происходят в фибоначчиевой куче так же, как и в улучшенной биномиальной куче.

Пока нет уменьшения ключа, наша куча не отличается от биномиальной и работает за такое же время.

21.2. Буду резать, буду удалять...

Добавим `DecreaseKey` за амортизированное $\mathcal{O}(1)$. Удаление выражается через него: уменьшим ключ удаляемого элемента до $-\infty$ и извлечем минимум.

Будем теперь каждую вершину помечать, если у нее недостает одного сына, то есть она имеет ранг n, но $n-1$ сыновей.

Тогда в `DecreaseKey(`v`)` после непосредственного уменьшения ключа v можно рассматривать следующие три случая:

1. Если инвариант кучи не нарушен, то больше ничего не делаем.
2. Иначе если отец v не помечен, то можно "вырезать" v, то есть удалить её из списка детей отца v и добавить в основной список деревьев кучи (так же, как в Pairing Heap). Теперь нужно пометить бывшего отца v.
3. Если отец v уже помечен, то все равно вырезаем v, а потом рекурсивно вырезаем отца v (в результате чего, возможно, придется вырезать и дедушку v, и следующих предков).

В любом случае, у всех только что вырезанных вершин нужно снять пометки, если они там были.

Сейчас может создаться ощущение, что мы поломали всю структуру, ведь, строго говоря, после удаления хотя бы двух сыновей у корня дерева, оно больше не попадает под наше определение фибоначчиева дерева.

Улучшим ситуацию, немного переопределив понятие ранга фибоначчиева дерева.

Def 21.2.1. *Ранг фибоначчиева дерева — степень его корня в последний момент времени, когда оно лежало в основном списке кучи.*

Чтобы такое определение было более интуитивным, рассмотрим дерево в тот момент, когда оно исчезает из списка корней кучи (т.е. подвешивается к дереву такого же ранга). Тогда до тех пор, пока оно опять не попадет в основной список, его степень будет не более, чем на 1, меньше его ранга — а ведь именно это свойство и является одним из ключевых в фибоначчиевом дереве.

21.3. А причём тут, собственно, Фибоначчи?

Lm 21.3.1. Пусть Fib_n — n-е число Фибоначчи. Тогда $Fib_n = 1 + \sum\limits_{i=1}^{n-2} Fib_n$

Доказательство. Докажем по индукции:

- База: $Fib_0 = Fib_1 = 1$.
- Переход: $Fib_n = Fib_{n-1} + Fib_{n-2} = 1 + \sum\limits_{i=1}^{n-3} Fib_n + Fibn - 2 = 1 + \sum\limits_{i=1}^{n-2} Fib_n$.

Lm 21.3.2. В фибоначчиевой куче у фибоначчиева дерева ранга n, можно так упорядочить детей корня, что ранг i-го из них будет не меньше i.

Доказательство. Доказать это можно индукцией по количеству операций над кучей. База — пустая куча — очевидна. Переход:

- При добавлении одного элемента это свойство не нарушается, при слиянии двух куч — тем более.
- При вырезании v, если она у своего отца имела номер i (в порядке, определённом условием леммы и существующем по предположению индукции), то у всех следующих сыновей отца v неравенство только усилилось.
- При подвешивании одного дерева ранга n к другому дереву такого же ранга мы можем сопоставить новому сыну номер n.

Lm 21.3.3. В фибоначчиевой куче размер фибоначчиева дерева ранга n не меньше Fib_n.

Доказательство. Доказать это можно аналогичной индукцией по количеству операций над кучей. Рассмотрим дерево которое получилось в результате очередной операции. По предыдущей лемме, можно упорядочить детей его корня так, чтобы ранг i-го из них будет не меньше i, значит, по предположению индукции, размер i-го хотя бы Fib_i. Пусть размер всего дерева равен S, а размер i-го из сыновей — S_i. Забудем теперь про n-го сына, если он есть, и оценим размер всего дерева: $S \geqslant 1 + \sum\limits_{i=0}^{n-2} S_i \geqslant 1 + \sum\limits_{i=0}^{n-2} Fib_n = Fib_n$.

Следствие 21.3.4. В фибоначчиевой куче размер дерева ранга n хотя бы φ^n, где φ — золотое сечение.

Следствие 21.3.5. Если куча содержит n элементов и в списке кучи все деревья имеют различные ранги, то их количество — $\mathcal{O}(\log n)$.

Оценим, наконец, время работы всех операций. Для этого введем потенциал, равный $2M + R$, где M — количество помеченных вершин, а R — количество корней в списке кучи. Сразу заметим, что изначально он равен 0, а после n операций точно не больше $3n$.

Итак,

- `Add`: Потенциал изменяется на 1, как реальное, так и амортизированное время работы — $\mathcal{O}(1)$
- `Merge`: Потенциалы двух сливаемых куч складываются, тоже $\mathcal{O}(1)$.
- `ExtractMin`: M не изменяется, поэтому значимым слагаемым в потенциале является только R. Имея его, аналогично биномиальной куче, амортизированное время работы — $\mathcal{O}(\log n)$.
- `DecreaseKey`: Пусть было вырезано k вершин, тогда хотя бы у $k-1$ из них (кроме самой первой), снялась пометка. Количество помеченных вершин уменьшилось хотя бы на $k-1$, а количество корней увеличилось на k, поэтому изменение потенциала не больше $2(1-k)+k = 2-k$. Так как реальное время работы — k, то амортизированное время работы — $\mathcal{O}(1)$.

21.4. Бонус: нужно больше сравнений!

Теорема 21.4.1. Любой корректный алгоритм построения бинарной кучи из n элементов делает не менее $1.3644n$ сравнений.

Доказательство. Будем оценивать на перестановках чисел от 1 до n.

Посчитаем, сколько существует бинарных куч. Сначала есть множество из $n!$ перестановок. На первом шаге мы обмениваем минимальный элемент с элементом в корне. Видно, что перестановка p могла получиться ровно из n других, так как в исходной перестановке минимум мог лежат на любом из n мест. Значит количество перестановок в множестве уменьшилось в n раз.

На i-м шаге обмениваем элемент на i-м месте с минимумом i-й вершины. Аналогично, количество перестановок уменьшилось в s_i раз, где s_i — размер поддерева i-й вершины ($s \simeq n, \frac{n}{2}, \frac{n}{4}, \frac{n}{4}, \frac{n}{4}, \frac{n}{4}, \ldots$).

В итоге получим множество всех корректных бинарных куч. Его размер равен $\dfrac{n!}{\prod\limits_{i=1}^{n} s_i}$.

Теперь предположим, что алгоритм делает k сравнений. Тогда перестановки разбиваются на 2^k групп, в одной группе к каждой перестановке применяется одна и та же последовательность обменов. Тогда существует группа размера хотя бы $\frac{n!}{2^k}$. Так как перестановки в ней различны, а последовательность обменов совпадает, все перестановки в ней превращаются в различные бинарные кучи. Получается, что количество различных бинарных куч не меньше, чем размер этой группы, то есть:

$$\frac{n!}{\prod_{i=1}^{n} s_i} \geqslant \frac{n!}{2^k} \Rightarrow 2^k \geqslant \prod_{i=1}^{n} s_i \Rightarrow k \geqslant 1.3644n$$

Последний переход шёл без доказательства.

Лекция по алгоритмам #22
Динамика
16 октября

Рассмотрим задачу калькулятор. Хотим получить число N из 1 за минимальное число шагов. У нас есть три вида операций: умножить на 2, умножить на 3 и прибавить единицу. Определим f[x] - минимальное число ходов, чтобы из 1 получить x.

22.1. Динамика назад

Состояния последовательно пересчитывается исходя из уже посчитанных. f[1] = 0 - база

$$f[x] = min \begin{Bmatrix} f[x-1] \\ f[x/2], & \text{если x} \vdots 2 \\ f[x/3], & \text{если x} \vdots 3 \end{Bmatrix} + 1$$

Выбираем минимум из уже посчитанных значений и прибавляем 1, т.к. сделали 1 ход. Таким образом, нашли оптимальный ответ, свели задачу в к меньшим подзадачам.
Перебираем x в порядке возрастания:

```
1  f[1] = 0
2  for x = 2...N
3        f[x] = ...
```

Асимптотика - $\Theta(N)$. Ответ лежит в ячейке f[N].

22.2. Динамика вперед

Обновляются все состояния, зависящие от текущего состояния. Пусть $f[i] = \infty \ \forall i$.
Положим f[1] = 0

```
1 void relax(int &a, int b){
2     a = max(a, b)
3 }
4 for x = 1..N - 1
5     f[x + 1] = relax(f[x + 1], f[x] + 1)
6     if 2x <= N
7         f[2x] = relax(f[2x], f[x] + 1)
8     if 3x <= N
9         f[3x] = relax(f[3x], f[x] + 1)
```

Корректность доказывается по индукции. От текущего состояния у нас могут зависеть до 3 других состояний, поэтому каждый раз мы их обновляем. Результат будет корректен, поскольку f[x] у нас посчитано корректно, для других состояний мы не ухудшаем ответ.

22.3. Ленивая динамика(рекурсия)

Добавим еще одну операцию в задачу - прибавить 2 к числу.
Напишем долгий перебор и оптимизуем его.
Заполним глобальный массив F бесконечностями.

```
int f(int x){
    if (x == 1) return 0
    int &res = F[x] // Запоминаем ссылку на res, в дальнейшем все действия с res будут изменять и F[x]
    if (res != + ∞) // наше значение уже было раньше посчитано, нет смысла его считать заново
      return res
    relax(res, f(x - 1))
    if (x % 2 == 0)
        relax(res, f(x/2))
    if (x % 3 == 0)
        relax(res, f(x/3))
    if (x >= 3)
        relax(res, f(x - 2))
    return res += 1
}
```

Время перебора без отсечений. Верхняя оценка: 4^N - Можем применить одну из 4 операций N раз. Нижняя оценка : n-ое число Фибоначчи(т.к. вызываемся от f(x - 1) и от f(x - 2)). $4^N \geqslant Time \geqslant FibN = \phi^N$
У ленивой динамики есть 2 преимущества:

- Не фиксированный порядок перебора
- Посещение только достижимых состояний

Все эти задачи можно свести к нахождению кратчайшего пути из 1 состояния в N-ое в ациклличном ориентированном графе, в котором вершины - состояния, ребро между состояниями - переход из одного в другое.

22.4. Что нужно знать для динамики

- Состояние динамики: параметр(ы), однозначно задающие подзадачу.
- Переходы между состояниями: формула пересчёта.
- Начало, т.е. значения начальных состояний.
- Порядок пересчёта.
- Положение ответа на задачу.

22.5. Граф состояний

Мы можем сформулировать термины ленивой, вперед, назад динамики через ацикличный граф. Каждый раз когда мы делали переход - можно сказать, что мы переходили из одной вершины графа(вершина - состояние) в другую.

- Ленивая динамика: происходит поиск в глубину из последней вершины к первой
- Динамика вперед: из каждой вершины мы релаксируем путь в другие в вершины.

- Динамика назад: Из всех путей ведущих в текущую вершину выбираем оптимальный.

22.6. Задача про путь в матрице

Пусть дана матрица a[n, m]. Хотим найти путь минимальной стоимости, максимальной стоимости, количество путей из точки [1,1] в точку [n, m].

- Состояние - [i, j], изначально все заполнено плюс или минус бесконечностями, в зависимости, что мы считаем: минимумальный или максимальный путь соотвественно.
- Переходы из [i, j] в [i + 1, j] и в [i, j + 1]
- Начало в [1, 1]
- Конец [n, m]
- Порядок: по порядку обрабатываем строчки сверху вниз.

Если мы считаем путь минимальной/максимальной стоимости то:

```
1  dp[i + 1, j] = min/max(dp[i, j] + a[i, j], dp[i + 1, j];
2  dp[i, j + 1] = min/max(dp[i, j] + a[i, j], dp[i, j + 1];
```

Если мы считаем количество путей, то тогда количество в следующую ячейку будет равнятся сумме двух ячеек, из которых можно прийти. Изначально dp[1, 1] = 1. В dp[n, m] будет лежать ответ.

Лекция по алгоритмам #23
Динамическое программирование

16 октября

23.1. Рюкзак

23.1.1. задача

Дано множество предметов $a_1, a_2, \ldots, a_n$
$a_i > 0,\ W > 0$
Нужно выбрать подмножество: $\sum a_i = W$

1. NP-hard
2. $n \cdot 2^n$
3. $\exists$ решения за $2^{n/2}$

   (meet in middle)

23.2. решение за O(nW)

Просмотрели первых i, взяли подмножество суммы w
$is[i, w] \quad \to is[i+1, w + a[i+1]]$
$\quad\quad\quad\ \to is[i+1, w]$

```
1     is[0,0] = 1:
2     for i = 1..n-1:
3       for w = 0..W:
4         if is[i, w]:
5           is[i + 1, w] = 1
6           is[i + 1, w + a[i]] = 2
```

23.2.1. восстановление ответа

```
1     i = 1, w = W
2     ans = []
3     while (i > 0):
4         if (is[i][W] == 2):
5               W -= a[i]
6               ans.append(a[i])
7         i--
```

23.2.2. оптимизируем память: храним 1-2 посл. строчек

```
1       for i = 0..n-1:
2         for w = W-a[i]..1:        // for w = 0..W-a[i] не верно!!!
3           if (is[w]):
4           is[w + a[i]] = 1
```

23.2.3. код с bitset

```
bitset<W + 1> is;
is[0] = 1;
for (int i = 0; i < n; i++) {
  is |= is << a[i];
}
```

$Memory = \Theta(\frac{W}{b}), \quad b = 64$
$Time = \Theta(\frac{n \cdot W}{b})$

23.2.4. Востановление ответа с O(W) памяти

$last[w]$ - номер последнего предмета в ответе для w

```
last = [0, 0, ...]
last[0] = -1
for i = 0..i-1:
  for w = W-a[i]..0:
    if (last[w] != 0 and last[w + a[i]] == 0):
      last[w + a[i]] = i + 1

w = W
while(w != 0):
  i = last[w] - 1
  w -= a[i], res[i] = 1
```

для каждого w хранится наименьший номер последнего элемета

//для рюкзака со стоимостями такой способ восстановления ответа не верен

23.3. Алгоритм Хиршберга

23.3.1. НОП

даны стороки a, b
$dp[i, j]$ - НОП для $a[:i]$ и $b[:j]$

```
relax(&x, y):
  x = max(x, y)

lcs(a,b):
  //тут должна быть инициализация dp нулями

  for i = 0..len(a):
    for j = 0..len(b):
      relax(dp[i+1,j], dp[i,j])
      relax(dp[i,j+1], dp[i,j])
      if a[i] == b[j]:
        relax(dp[i,j+1],dp[i,j] + 1)

  ans = str()
```

```
15          i = len(a)
16          j = len(b)
17          while (i > 0 or j > 0): восстановление ответа
18              if (dp[i,j] == dp[i-1,j]):
19                i--
20              else if (dp[i,j] == dp[i,j-1]):
21                j--
22              else if (dp[i,j] == dp[i-1,j-1] + 1):
23                ans = a[i] + ans
24                i--, j--
25          ans = ans.reverse()
26          return ans
```

23.3.2. НОП с восстановлением ответа и O(n + m) памяти

```
1   solve(la, ra, lb, rb):
2       if (min(ra - la, rb - lb) <= 1):
3           return lcs(a[la:ra], b[lb:rb]) //длина одной строк <= 1, lcs = общий символ
4
5       // надо хранить только 2 последние строчки dp
6       for i = la..ra:
7           for j = lb..rb:
8               relax(dp[i,j], dp[i,j-1])
9               relax(dp[i,j], dp[i-1,j])
10              if (i > la and j > lb and a[i - 1] == b[j - 1]):
11                  relax(dp[i,j], dp[i-1,j-1] + 1)
12              if (i == (la + ra) / 2):
13                  dp[i,j].second = j
14
15      mb = dp[ra,rb].second;
16      return solve(la, ma, lb, mb) + solve(ma, ra, mb, rb)
17
18    solve(0, len(a), 0, len(b)) //запуск
```

Хотим найти НОП(a,b)

- Пусть $min(len(a), len(b)) > 1$, ma = len(a) / 2.
- Находим такую клетку $dp[ma, mb]$, что через неё пройдёт восстановление ответа.
- решаем для ($a[:ma]$ и $b[:mb]$) + ($a[ma:]$ и $b[mb:]$).

Корректность:

- восстановление ответа проходит через хотя бы 1 клетку из dp[ma,*]
- $\exists j$, НОП(a,b) = НОП($a[:ma]$ и $b[:j]$) + НОП($a[ma:]$ и $b[j:]$)
- j = mb подойдёт. (буквы в НОП соотв переходам $dp[i,j] \to dp[i-1, j-1]$ в восстановлении ответа)

Время: $\mathcal{O}(nm),\ n = len(a),\ m = len(b)$
$T(1, m) = m,\ T(n, 1) = n$
$T(n, m) \leq (3n - 1)(3n - 1) - 2$
$T(n, m) \leq (n + 1)(m + 1) + T(\frac{n}{2}, m - i) + T(\frac{n}{2}, i) = (n + 1)(m + 1) + (\frac{3n}{2} - 1)(3m - 2) - 4$

Память: $M = \text{dp} + \text{ответ} + \text{стек рекурсии} = \mathcal{O}(2(m + 1)) + \mathcal{O}(n) + o(n) = O(n + m)$

Лекция по алгоритмам #24
Динамическое программирование

19 октября

24.1. Редакционное расстояние (расстояние Левенштейна)

24.1.1. Постановка задачи

Даны строки s и t. Разрешается изменять строку s с помощью трех операций:

1. Вставить новый символ в любое место в строке.
2. Удалить любой символ из строки.
3. Изменить любой символ строки.

Требуется за наименьшее число операций получить из строки s строку t.

24.1.2. Решение

Def 24.1.1. *Преобразованием будем называть некоторую последовательность операций над строкой s, в результате выполнения которых получается строка t.*

Теорема 24.1.2. В любом преобразовании, число операций в котором наименьшее возможное, над каждым символом строки s производится не более одной операции, а над каждым добавленным символом операции не производятся.

Доказательство. Зафиксируем некоторое преобразование с наименьшим числом операций P, любой символ строки s и посмотрим на то, что с этим символом происходило в преобразовании P. Обозначим этот символ — c. После выполнения всех операций c был либо удален, либо стал некоторым символом из строки t, причем возможно над c была применена операция изменения. Если c был удален и над ним применялись некоторые операции изменения, то их можно убрать из P и получить преобразование с меньшим числом операций. Если c не удалялся и изменялся несколько раз, то можно убрать из P все операции изменения c, кроме последней, и получить преобразование с меньшим числом операций.
Теперь зафиксируем любой добавленный операцией символ d (если такой существует). Если после выполнения всех операций d удален, то можно убрать из P все операции связанные с d и получить преобразование с меньшим числом операций. Если в P есть операции изменяющие d, то можно все эти операции убрать из P, а добавить d со знчением, на которое изменяла d последняя операция изменения. Тогда получится преобразование с меньшим числом операций. ■

Следствие 24.1.3. В любом преобразовании с наименьшим числом операций неважно в каком порядке выполнять операции.

Доказательство. Каждая операция выполняется над символом. Если поменять местами две операции над различными символами, то ничего не изменится. В преобразовании с наименьшим числом операций над каждый символом выполняется не больше одной операции, значит можно менять местами любые две операции, так как они производятся над разными символами. ■

Def 24.1.4. *$f_{i,j}$ — редакционное расстояние между префиксом строки s длинны i и префиксом строки t длинны j.*

Для того, чтобы ввести соотношение для $f_{i,j}$, нужно перебрать, какой символ после выполнения операций будет соответствовать символу t_j:

1. Добавленный символ. Тогда $f_{i,j} = f_{i,j-1} + 1$, то есть наименьшее число операций, чтобы преобразовать префикс s длины i в префикс t длины $j-1$ и еще одна операция добавления символа t_j.

2. s_i. Тогда если $s_i = t_j$, то $f_{i,j} = f_{i-1,j-1}$, иначе $f_{i,j} = f_{i-1,j-1} + 1$. То есть наименьшее число операций для оставшихся символом и возможно еще операция изменения символа s_i в t_j, если они различаются.

3. Символ из s, но не s_i. Тогда символ s_i не может быть не удален, так как тогда не понятно, какому символу из t он будет соответствовать. Значит $f_{i,j} = f_{i-1,j} + 1$, то есть наименьшее число операций для оставшихся символов и еще одна операция удаления.

Среди всех вариантов для $f_{i,j}$ нужно выбрать минимум.

- Так как $f_{i,j}$ зависит от значений f с меньшими индексами, то для вычисления f i и j нужно перебирать в порядке увеличения.
- Для восстановления ответа можно ходить по обратным ребрам в графе.
- Заметим, что к данному решению можно применить метод Хиршберга

24.2. Наибольший общий префикс (Longest Common Prefix, LCP)

24.2.1. Постановка задачи

Даны строки s и t. Требуется посчитать $f_{i,j}$ — наибольший общий префикс суффикса s длины i и суффикса t длины j.

24.2.2. Решение

Если $s_i \neq t_j$, то $f_{i,j} = 0$, иначе, если строки не кончились, то $f_{i,j} = f_{i+1,j+1} + 1$. То есть либо у строк нет общих префиксов, либо первые символы совпадают и каждый общий префикс является общим префиксом строк без первого символа.
Так как $f_{i,j}$ зависит от значений f с большими индексами, то для вычисления f i и j нужно перебирать в порядке уменьшения.

24.3. Наибольшая подпоследовательность палиндром

24.3.1. Постановка задачи

Дана строка s. Требуется найти самую длинную подпоследовательность s, которая является плиндромом.

24.3.2. Решение

Def 24.3.1. *$f_{l,r}$ — ответ на задачу для подстроки строки s, начинающейся с символа, стоящего на l-ой позиции, и заканчивающейся символом, стоящим на r-ой позиции.*

Для того, чтобы ввести соотношение для $f_{l,r}$, нужно перебрать, есть ли в ответе символы на l-ой и r-ой позициях:

1. Символ на l-ой позиции не входит в ответ. Тогда $f_{l,r} = f_{l+1,r}$.
2. Символ на r-ой позиции не входит в ответ. Тогда $f_{l,r} = f_{l,r-1}$.
3. Символы на l-ой и r-ой позициях входят в ответ. Но тогда они должны быть равны, так как один из них первый символ в ответе, а другой последний. Если они равны, то $f_{l,r} = f_{l+1,r-1} + 1$.

Среди всех вариантов для $f_{l,r}$ нужно выбрать максимальный.

- Так как $f_{l,r}$ зависит от значений f, у которых $r - l$ меньше, то сначала нужно перебирать $r - l$, а затем один из индексов в любом порядке.
- Для восстановления ответа можно ходить по обратным ребрам в графе.

24.4. Наибольшая возрастающая подпоследовательность (Longest Increasing Subsequence, LIS)

24.4.1. Постановка задачи

Дана последовательность a_n. Требуется найти у нее наибольшую строго возрастающую подпоследовательность.

24.4.2. Решение #1

Def 24.4.1. *f_i — наибольшая возрастающая подпоследовательность a_n, оканчивающаяся i-ым элементом.*

Для того, чтобы ввести соотношение для f_i, нужно перебрать предыдущий элемент подпоследовательности. Получится, что $f_i = \min\limits_{j<i,\ a_j<a_i} \{f_j\} + 1$.

- Так как f_i зависит только от значений f с меньшими индексами, то для вычисления f нужно перебирать i в порядке возрастания.
- Ответом для всей задачи является $\max\{f_i\}$.
- Для восстановления ответа можно ходить по обратным ребрам в графе.

24.4.3. Решение #2

Def 24.4.2. *$g_{i,j}$ — наименьший элемент (по значению) на который может оканчиваться возрастающая подпоследовательность длины j, состоящая из элементов префикса последовательности a_n длины i, либо $+\infty$, если подпоследовательностей с такими свойствами не существует.*

Возможно два случая:

1. Элемент a_i не принадлежит подпоследовательности. Тогда $g_{i,j} = g_{i-1,j}$.
2. Элемент a_i принадлежит подпоследовательности. Тогда если его убрать из подпоследовательности, то получится возрастающая подпоследовательность длины $j-1$ из первых $i-1$ элементов a_n, у которой последний элемент будет меньше, чем a_i. Наоборот, если существует возрастающая подпоследовательность первых $i-1$ элементов a_n длины $j-1$ такая, что ее последний элемент меньше a_i, то к ней можно добавить в конец a_i и получить возрастающую подпоследовательность длины j. Теперь, если $g_{i-1,j-1} < a_i$, то существует возрастающая подпоследовательность первых i элементов a_n длины j, оканчивающаяся a_i, в таком случае $g_{i,j} = a_i$. Если $g_{i-1,j-1} \geq a_i$, то искомой последовательности для $g_{i,j}$ не существует, так как в противном случае можно было бы убрать из этой подпоследовательности последний элемент a_i и получить возрастающую подпоследовательность длины $j-1$, у которой последний элемент меньше, чем $g_{i-1,j-1}$. Но это противоречит определению $g_{i-1,j-1}$.

Среди всех вариантов для $g_{i,j}$ нужно выбрать минимальный.

- Так как $g_{i,j}$ зависит только от значений g с меньшими индексами, то для вычисления g нужно перебирать i и j в порядке возрастания.
- Ответом для всей задачи является $\max\limits_{j|g_{n,j}\neq+\infty}\{j\}$.
- Для восстановления ответа можно ходить по обратным ребрам в графе.

24.4.4. Решение #3

Начнем оптимизировать второе решение.

Lm 24.4.3. При фиксированном i: $g_{i,j} \leq g_{i,j+1}$, то есть g_i не убывает.

Доказательство. Пусть $\exists j : g_{i,j} > g_{i,j+1}$, тогда $g_{i,j} \neq +\infty$, так как существует возрастающая подпоследовательность длины $j+1$, поскольку $g_{i,j+1} < g_{i,j} \leq +\infty$, и из нее можно убрать последний элемент и получить некоторую возрастающую подпоследовательность длины j.
Но тогда существуют соответствующие возрастающие подпоследовательноти a_k^{j+1} длины $j+1$ и a_l^j длины j. Но если последний элемент a_k^{j+1} меньше, чем последний элемент a_l^j, то предыдущий элемент $a_{k-1}^{j+1} < a_k^{j+1} < a_l^j$ тоже меньше последнего элемента a_l^j. Но тогда, последовательность a^{j+1} без последнего элемента является возрастающей подпоследовательностью длины j, и ее последний элемент меньше, чем $g_{i,j}$, но это противоречит определению $g_{i,j}$. ■

Lm 24.4.4. Для фиксированного i g_i отличается от g_{i-1} не более, чем в одном элементе, $g_{i,j_1} \neq g_{i-1,j_1}, g_{i,j_2} \neq g_{i-1,j_2} \Rightarrow j_1 = j_2$. Т.е. при пересчете g_i через g_{i-1} значение улучшится только в одном элементе.

Доказательство. Т.к. последовательность g_{i-1} не убывает, то ее можно поделить на три последовательные части: элементы меньшие a_i, элемент-$lower_bound$ a_i, все остальные элементы. Пусть $lower\_bound$ a_i — $g_{i-1,b}$. Все элементы меньшие a_i не могут быть улучшены, т.к. для каждого из них существует возрастающая подпоследовательность, оканчивающаяся на элемент меньший a_i. Значение $g_{i,b} = a_i$, по правилу пересчета, так как $g_{i-1,b-1} < a_i$ или, в случае, когда $b = 1$, не существует. Аналогично по правилу пересчета все остальные элементы не изменяются, так как $\forall j > b : g_{i-1,j-1} \geq a_i$. ■

Теперь, вместо линейного пересчета, при переходе от g_{i-1} к g_i, будем пользоваться бинарным поиском по g_{i-1}, чтобы найти элемент, который может измениться. Остается лишь считать g не в двухмерном массиве, а хранить только g_i и получать из него g_{i+1}, изменяя не более одного элемента.
Для восстановления ответа нужно поддерживать последовательность p_i, которая формируется следующим образом: когда пересчитывается g_i — $g_{i-1,b}$ это $lower\_bound$ a_i, тогда p_i равняется индексу элемента, который записан в $g_{i-1,b-1}$, или ничему, если $b = 1$. Таким образом поддерживается инвариант, что, если элемент a_k был записан в $g_{i,j}$, то p_k указывает на меньший индекс меньшего элемента, причем, пройдя по этим «ссылкам», можно получить возрастающую подпоследовательность длины j в обратном порядке.

24.5. Скобочная последовательность (Динамика по подотрезкам)

24.5.1. Постановка задачи

Дана скобочная последовательность s из разных типов скобок. Нужно удалить минимальное количество скобок так, чтобы оставшаяся последовательность стала правильной скобочной последовательностью.

24.5.2. Решение

Def 24.5.1. *$f_{l,r}$ — минимальное количество скобок, которое нужно удалить из подстроки s, начинающейся в позиции l и заканчивающейся в позиции r, чтобы получить из этой подстроки правильную скобочную последовательность.*

Для того, чтобы ввести соотношение для $f_{l,r}$, нужно перебрать, что произойдет со скобкой в позиции l:

1. Скобка будет удалена. Тогда $f_{l,r} = f_{l+1,r} + 1$, или $f_{l,r} = 1$, если $l = r$.
2. Скобка не будет удалена, тогда найдется парная ей скобка. Пусть эта парная скобка находится на позиции m, тогда из $s_{l+1,m-1}$ нужно удалить некоторые скобки, чтобы получить правильную скобочную последовательность и аналогично с $s_{m+1,r-1}$. То есть нужно перебрать парную скобку для скобки на позиции l и по всем вариантам взять минимум: $f_{l,r} = \min\limits_{m} f_{l+1,m-1}, f_{m+1,r-1}$.

Среди всех вариантов для $f_{l,r}$ нужно выбрать минимальный.

- Так как $f_{l,r}$ зависит от значений f, у которых $r - l$ меньше, то сначала нужно перебирать $r - l$, а затем один из индексов в любом порядке.
- Для восстановления ответа можно ходить по обратным ребрам в графе.

Лекция по алгоритмам #25
Динамика (продолжение)

19 октября

25.1. Умножение матриц

25.1.1. Собственно, умножение

Если есть 2 матрицы A и B размеров (n, m) и (m, k) соответственно (количество столбцов в первой обязательно равно количеству строк во второй), то мы умеем умножать их за nmk. Эта операция ассоциативна. Больше про умножение матриц мы ничего пока знать не должны.

25.1.2. Постановка задачи

Есть n матриц $A_1, A_2, \ldots, A_n$, $size(A_i) = (m_i, m_{i+1})$. Перемножить их за минимальное количество операций.

25.1.3. Решение

Суть задачи — оптимально расставить скобки. Это динамика по подотрезкам. Давайте переберём, какую операцию совершить последней. Пусть мы считаем ответ на отрезке $[L, R]$. Пусть мы выбрали знак умножения между матрицами A_M и A_{M+1}. После того, как мы полностью перемножим все матрицы на отрезках $[L, M]$ и $[M + 1, R]$, у нас слева останется матрица размерностей (m_L, m_{M+1}), а справа — (m_{M+1}, m_R). Их мы перемножим за $m_L \cdot m_{M+1} \cdot m_R$. Итак, ответ для отрезка — оптимум из значений $dp[L][M] + dp[M + 1][R] + m_L \cdot m_{M+1} \cdot m_R$. База: $dp[L][L] = 0$.

25.2. Арифметические выражения

25.2.1. Постановка задачи

Дано арифметическое выражение, в котором присутствуют только знаки + и $\cdot$, необходимо максимизировать результат. Для простоты: все числа натуральные.

25.2.2. Решение

Точно такая же динамика по подотрезкам. Перебираем последнюю операцию, которая разбивает наш отрезок на 2. Берём максимальный ответ на левом отрезке и на правом, применяем к ним нашу последнюю операцию. Из всех таких переходов выбираем максимум. $Time = \Theta(n^3)$

25.3. Задача о погрузке грузов

25.3.1. Постановка задачи

Есть бесконечное количество кораблей. У каждого корабля одна и та же грузоподъёмность W. Есть последовательность из n грузов, у каждого есть вес a_i. Мы можем брать какой-то префикс последовательности и какой-то суффикс и грузить эти грузы на корабль. При этом

сумма весов грузов на корабле не должна превышать W. Необходимо погрузить все грузы на минимальное количество кораблей.

25.3.2. $O(n^4)$

$dp[L][R]$ — минимальное количество кораблей, на которые мы можем погрузить все грузы с L по R. Перебираем, какой префикс и какой суффикс мы погрузим на следующий корабль. У нас $O(n^2)$ состояний и $O(n^2)$ переходов — за линию перебираем префикс, заканчивающийся в позиции i и за линию суффикс, начинающийся в j. Из $[L, R]$ переходим в $[i, j]$.

25.3.3. $O(n^3)$

Заметим, что нам не нужно перебирать и префикс, и суффикс, так как невыгодно грузить корабль не до конца. Итак, для фиксированного i мы берём такое j, чтобы поместить на корабль как можно больше грузов. Заметим, что при увеличении индекса i такой индекс j также увеличивается, т.е. мы можем применить метод двух указателей.

25.3.4. $O(n^2)$, измельчение переходов

А в чём, собственно, была проблема? В том, что каждый раз мы пытались загрузить корабль полностью. Давайте за один переход грузить всего один груз. Параметры отстанутся такими же — границы текущего отрезка. Но в динамике мы будем минимизировать пары: сколько кораблей уже отправлено и какой вес погружен на последний. Для начала рассмотрим переходы. Из состояния, где отправлено k кораблей, на неотправленный погружен вес w, а от грузов остался отрезок $[L, R]$, мы можем сделать 3 вещи. Первое: отправить корабль. Тогда k перейдёт в $k+1$, а w в 0. Мы можем погрузить левый груз, тогда w перейдёт в $w + a_L$, L в $L + 1$; или правый, тогда w перейдёт в $w + a_R$, а R в $R - 1$. Итак, у нас не n переходов, а всего 3. Правда, один из них делается в тот же отрезок, и это не очень приятно. Как бороться с переходом в себя? Можно сделать не 3, а 4 перехода. В одном из них мы грузим левый груз, не отправляя при этом корабль, в другом, напротив, отправляем; так же с правым грузом. Замечательно то, что мы можем получить ответ с $O(n)$ памяти, храня только последнюю строчку матрицы, где считаем динамику. Для этого L нужно перебирать в порядке возрастания, а R в порядке убывания, тогда всё будет считаться корректно и в нужном порядке.

25.3.5. Так почему же это работает?

Почему мы можем минимизировать пары? Давайте рассмотрим пары (k_1, w_1) и (k_2, w_2). Есть 2 очевидных случая. Пусть $k_1 = k_2$, $w_1 \neq w_2$. Здесь мы обращаем внимание только на одно значение и, естественно, пытаемся его минимизировать. Пусть теперь $k_1 < k_2$, $w_1 \leq w_2$. Здесь минимизация обоих значений также вопросов не вызывает. Теперь пусть $k_1 < k_2$, $w_1 > w_2$. Заметим, что из (k_1, w_1) мы можем перейти в $(k_1 + 1, 0)$ и $k_1 + 1 \leq k_2$, $0 \leq w_2$, значит, это состояние ничем не хуже (k_2, w_2). Итак, мы можем минимизировать пары, динамика работает.

Лекция по алгоритмам #26
Кучи и динамика

23 октября

26.1. Кучи. Две оптимизации куч

1. **Оптимизация операции Add**

Формулировка
Пусть есть куча, которая кмеет делать Add, $Merge$ и $ExtractMin$ за $O(\log(n))$ действий и $Build$ за $O(n)$ действий. Хотим из нее сделать кучу, которая делает $Merge$ и $ExtractMin$ за $O(\log(n))$ действий, Add за $O(1)$ действий и $Build$ за $O(n)$ действий.

Идея решения
Буферезированный Add.

Описание решения
Когда делаем Add, добавляем элемент в $buffer$. Когда делаем остальные операции, сначала делаем $Build(buffer)$ и $Merge$ результата с кучей, а потом саму операцию.

Реализация (псевдокод)

```
struct Q
    vector buffer;
    Heap heap;

Add(x)
    buffer.pop_back(x);

Merge(q)
    heap.Merge(Build(buffer));
    q.heap.Merge(Build(q.buffer));
    heap.Merge(q.heap);

ExtractMin()
    heap.Merge(Build(buffer));
    return heap.ExtractMin();

Build(vect)
    heap.Build(vect);
```

Доказательство асимптотики
Интуитивное:
Равномерно распределим действия, которые делает $Build$, между сделанными операциями Add. Тогда Add все еще делает $O(1)$ действий, а операции $Merge$ и $ExtractMin$ делают $O(\log(n))$ действий.

Строгое (метод потенциалов):
Пусть ϕ равно количеству k элементов в $buffer$, умноженному на константу c (нужно, чтобы

можно было сократить $O(k)$, которое делает $Build$ с $\delta\phi$). Тогда:
Add: $\delta\phi = c$; $t_i = O(1) \Rightarrow a_i = O(1)$
$Merge$: $\delta\phi = -c \cdot k$; $t_i = O(k) + O(\log(n)) \Rightarrow a_i = O(\log(n))$
$ExtractMin$: $\delta\phi = -c \cdot k$; $t_i = O(k) + O(\log(n)) \Rightarrow a_i = O(\log(n))$
Доказательство корректно, т.к. $0 \leq \phi_i \leq c \cdot n$;

2. Оптимизация Merge

Формулировка
Пусть есть куча, которая умеет делать $ExtractMin$ и $Merge$ за время $O(\log(n))$, Add за время $O(1)$ и $Build$ за время $O(n)$. Хотим сделать из нее кучу, которая умеет делать $ExtractMin$ за время $O(\log(n))$, Add и $Merge$ за время $O(1)$ и $Build$ за время $O(n)$.
Идея решения
На основе старой кучи типа Q создать новую кучу типа B, которая хранит минимум и кучу Q куч B

Описание решения
Куча типа B содержит поля min (минимальный эдемент) и q (куча типа Q куч типа B)

Def 26.1.1. $b_1 < b_2 \Leftrightarrow b_1.min < b_2.min$, *где* b_1 *и* b_2 *- кучи типа* B.

Add - это создание кучи типа B, состоящей из минимума, равного добавляемому элементу, и пустой кучи типа Q и добавление ее в q. ($q.Add$)
$Merge$ - пусть $b_1 < b_2$. Тогда Merge этих куч добавит в $b_1.q$ кучу b_2. ($b_1.q.Add$)
$ExtractMin$ - возвращает значение поля min на момент начала своего выполнения. Нужно получить новый минимум. Очевидно, что он хранится в $b_{tmp} = q.ExtractMin()$ (см. определение сравнения двух куч типа B). Теперь $min = b_{tmp}.min$, $q = Merge(q, b_{tmp})$.
$Build$ - Add, сделанный n раз.

Реализация (псевдокод)

```
template <class T>
struct B
    T min;
    Q< B<T>* >* q;


//Add == Merge

void Merge(B* b1, B* b2)
    if(b1->min > b2->min)
        swap(b1,b2);
    b1->q->Add(b2); //O(1)


T ExtractMin(B* b)
    T result = b->min;
    B* btmp = b->q->ExtractMin(); //O(log(n))
```

```
18      b->min = btmp->min;
19      b->q = QMerge(b2->q,b->q) //Merge two Q-type heaps. Q. Not B.
20      return result;
```

26.2. Динамическое программирование и перемножение матриц. Быстрое возведение в степень.

Задача про подсчет путей из a в b длины k

Формулировка

Дан ориентированный граф и таблица смежности $is_edge[a][b]$. Посчитать количество путей из вершины a в вершину b длины ровно k.

Решение

Динамика. Состояния: длина пути k, начальная вершина a, конечная вершина b.
Переход:
$f[k][a][b] = \sum\limits_c (f[k-1][a][c] \cdot is_edge[c][b])$

Заметим, что это то же самое, что и $f[k] = f[k-1] \cdot is_edge$, где $f[k]$, $f[k-1]$ и is_edge - матрицы.
$(C = A \cdot B \Leftrightarrow C_{i;j} = \sum\limits_k (A_{i;k} \cdot B_{j;k}))$
$\Rightarrow f[k] = f[0] \cdot is_edge^k = is_edge^k$

Время работы

Изначально динамика работала за $O(n^3 \cdot k)$. Теперь работает $O(n^3 \cdot \log(k))$ (используем быстрое возведение в степень).

Быстрое возведение в степень

```
1  Pow(A,n)
2      if(n == 0)
3          return E;
4      x = Pos(A, n / 2);
5      x = x*x;
6      if(n % 2 == 1)
7          x = x*A;
8      return x;
```

Задача про числа Фибоначчи

Формулировка

Посчитать n-ное число Фибоначчи.

Решение

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$$
$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

Лекция по алгоритмам #27
Матрицы и динамика

23 октября

27.1. Еще немножко о матрицах

1. **Постановка задачи:**

   Последовательность задана рекуррентным соотношением такого вида:

   $$F_n = \sum_{i=1}^{k} C_i F_{n-i} + b$$

   Элементы $\{F_1 \dots F_k\}$ даны. Требуется вычислить элемент послеовательности с номером N.

2. **Решение для случая $b = 0$:**
   Запишем переход от элементов $\{F_n \dots F_{n-k+1}\}$ к $\{F_{n+1} \dots F_{n-k}\}$ как домножение вектора на матрицу:

   $$\begin{pmatrix} F_{n+1} \\ F_n \\ F_{n-1} \\ \dots \\ F_{n-k} \end{pmatrix} = \begin{pmatrix} C_1 & C_2 & \dots & C_{k-1} & C_k \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_n \\ F_{n-1} \\ F_{n-2} \\ \dots \\ F_{n-k+1} \end{pmatrix} = M \cdot V$$

   Домножением первой строки матрицы на вектор получим элемент F_{n+1} в точности так, как он задан в соотношении. Следующие строки перенесут соответствующие элементы $F_n \dots F_{n-k}$ в новый вектор.
   Запишем формулу для получения вектора, содержащего элемент с номером N:

   $$\{F_{N-k+1} \dots F_N\} = \{F_1 \dots F_k\} \cdot M^{N-k}$$

3. **Случай $b \neq 0$:**
   Немного дополним матрицу и заметим, что все замечательно работает:

   $$\begin{pmatrix} F_{n+1} \\ F_n \\ \dots \\ F_{n-k} \\ 1 \end{pmatrix} = \begin{pmatrix} C_1 & C_2 & \dots & C_k & b \\ 1 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_n \\ F_{n-1} \\ \dots \\ F_{n-k+1} \\ 1 \end{pmatrix} = M \cdot V$$

   Теперь в первой строчке в сумму добавляется b, остальные строки все так же переносят соответствующие элементы V и единицу.

4. **Время работы:**

Матрица имеет размер $k \times k$, умножение происходит за $O(k^3)$. Используя быстрое возведение в степень, получаем время $O(k^3 \log N)$.

27.2. Задача о министерствах

1. **Постановка задачи:**

   Дана прямоугольная матрица $(n \times m)$, для каждой клетки известна стоимость ее посещения $cost_{i,j} \geq 0$. Изначально вы стоите в первой строке, в клетке с координатами (1 ; s). Нужно найти путь минимальной стоимости в клетку (n ; t). Разрешается ходить в клетку, соседнюю по стороне, кроме клетки в предыдущей строке.

2. **Полезные наблюдения:**

   Существует оптимальный ответ, в котором каждая клетка будет посещена не более 1 раза.
   Рассмотрим какой-нибудь оптимальный ответ. Возьмем в нем клетку, посещенную более одного раза. Удалим все переходы от ее первого до последнего вхождения в ответ. Ответ не ухудшился, так как мы вычли сумму неотрицательных чисел. Размер ответа уменьшился. Так как он конечный, за конечное число итераций сможем построить из любого оптимального ответа такой, в котором все клетки посещены не более 1 раза.

3. **Решение задачи:**

   Будем считать такую динамику: $dp_{i,j}$ – минимальная стоимость, за которую можно добраться до клетки (i ; j). Пересчитывать ее будем по строкам.
   Для очередной строки i сделаем два пробега – слева направо и справа налево. В первом случае будем релаксировать $dp_{i,j}$ переходом из $dp_{i,j-1}$, во втором – переходом из $dp_{i,j+1}$. Так как посещать одну клетку более одного раза бессмысленно (см. выше), все возможные пути, проходящие по этой строке тем самым обработаны. Перейдем в строку $i + 1$. Для этого значение в каждой клетке строки $i + 1$ прорелаксируем переходом в нее из предыдущей строки. Итого, обработка строки затрачивает $O(m)$ времени.
   Если не нужно восстанавливать ответ, то можно хранить значение динамики только в последних двух строках. Тогда получаем решение, затрачивающее $O(m)$ памяти и $O(nm)$ времени.

4. **Восстановление ответа:**

   Самый простой вариант – для каждой клетки хранить ее предка. Но такое решение будет использовать $O(nm)$ памяти, давайте улучшим:

   1. **Делим и властвуем:**

      Разделим строки на две половины (верх/низ). В каждой из них посчитаем динамику с линейной памятью, найдя при этом для каждой клетки из центральной строки предка из первой строки, а для клеток из последней – предка из центральной (аналогично алгоритму Хиршберга). Так найдем часть центральной строки, которая войдет в ответ. Ее можно хранить сжато (координаты клетки, в которой входим в центральную строку, и координаты клетки, в которой выходим). Затем вызовемся рекурсивно от каждой из половин, склеим их ответы с помощью посчитанного в центральной строке.

Память стала линейной - на каждом уровне рекурсии тратим $O(m)$ памяти на динамику, затем оставляем $O(1)$ на ответ для центральной строки. Но при этом время ухудшилось до $O(nm \log n)$, так как теперь каждая строка будет участвовать в пересчете динамики $\log n$ раз.

2. **Обобщим этот метод:**

   В предыдущем пункте мы поделили таблицу на две части, а что, если поделим на k? Как теперь будет работать динамика: делим строки на k групп, в каждой группе считаем динамику, поддерживая массив предков из последней строки предыдущей группы. Вызываемся рекурсивно от каждой группы, склеиваем полученные ответы.
   Теперь затраченная память – $O(km)$. Каждая строка теперь будет участвовать в динамике $\log_k n$ раз, итоговое время $O(nm \log_k n)$.

3. **Примеры выбора k:**

   - Если хранили всех предком, мы делили доску на n строк:
     $Mem = O(nm)$
     $Time = O(nm \log_n n) = O(nm)$

   - Если делили на 2 группы:
     $Mem = O(2nm) = O(nm)$
     $Time = O(nm \log_2 n)$

   - Если поделим на $\sqrt{n}$:
     $Mem = O(nm\sqrt{n})$
     $Time = O(nm \log_{\sqrt{n}} n) = O(nm)$

5. **Немножко выводов:**
   Алгоритм Хиршберга был применим только в задачах, где переходы из состояния (i, j) были в (i + 1, j), (i, j + 1) и (i + 1, j + 1). Описанный в этой задач метод восстановления ответа позволяет работать с еще двумя типами переходов – в (i, j - 1) и (i + 1, j - 1), это здорово.

Лекция по алгоритмам #28
Динамика по подмножествам

2 ноября

28.1. Представление множеств

Подмножества n-элементного множества (например, множества чисел от 0 до $n-1$) можно задавать последовательностями из n нулей и единиц естественным образом («0» — берём элемент, «1» — не берём). Поэтому в программе множество удобно представлять числом, битовая запись которого совпадает с заданием этого множества строкой из нулей и единиц.

28.2. Операции с множествами

Нам хочется, чтобы операции, которые мы умели производить над множествами, выражались и в новом представлении, то есть чтобы биекция была изоморфизмом.

28.2.1. Объединение

$C = A \cup B = \{x \mid x \in A \vee x \in B\}$

```
1  unsigned char a = 72;     // 01001000
2  unsigned char b = 184;    // 10111000
3  unsigned char c = a | b; // 11111000 = 248
```

28.2.2. Пересечение

$A \cap B = \{x \mid x \in A \wedge x \in B\}$

```
1  unsigned char a = 72;     // 01001000
2  unsigned char b = 184;    // 10111000
3  unsigned char c = a & b; // 00001000 = 8
```

28.2.3. Дополнение

$C = \overline{A}$

```
1  unsigned char a = 184; // 10111000
2  unsigned char c = ~a;  // 01000111 = 71
```

28.2.4. Симметрическая разность

$C = A \triangle B$

```
1  unsigned char a = 72;     // 01001000
2  unsigned char b = 184;    // 10111000
3  unsigned char c = a ^ b; // 11110000 = 240
```

28.2.5. Разность

Существует несколько способов найти разность множеств.
$A \setminus B = A \cap \overline{B} = A \triangle (A \cap B)$

```
1  unsigned char a = 184;          // 10111000
2  unsigned char b = 73;           // 01001001
3  unsigned char c = a ^ (a & b); // 10110000 = 176
4  unsigned char d = a & (~b);    // 10110000 = 176
```

28.2.6. Одноэлементные множества

Заметим, что одноэлементные множества задаются степенями двойки, которые легко вычислить с помощью операции битого сдвига.

$A = \{x\}$

```
1  int x = 4; // [0, 8)
2  unsigned char a = 1 << x; // 00010000 = 16
```

28.2.7. Добавление элемента в множество

Для того, чтобы добавить элемент x в множество A, нужно множество A объединить с одноэлементным множеством $\{x\}$.

$B = A \cup \{x\}$

```
1  int x = 6; // [0, 8)
2  unsigned char a = 184;          // 10111000
3  unsigned char b = a | (1 << x); // 11111000
```

Если элемент уже существовал в множестве, естественно, повторно он добавлен не будет.

28.2.8. Удаление элемента из множества

Для того, чтобы удалить x из множества A, нужно из A вычесть одноэлементное множество $\{x\}$. Или, если мы знаем, что элемент в множестве точно был, найти симметрическую разность A и $\{x\}$.

$B = A \setminus \{x\}$ или $B = A \triangle \{x\}$

```
1  int x = 5; // [0, 8)
2  unsigned char a = 184;          // 10111000
3  unsigned char b = a ^ (1 << x); // 10011000
```

28.2.9. Проверка принадлежности

$x \in A \Leftrightarrow A \cap \{x\} \neq \varnothing$

```
1  int x = 5; // [0, 8)
2  int y = 6; // [0, 8)
3  unsigned char a = 184;   // 10111000
4  bool inx = (1 << x) & a; // 00100000, true
5  bool iny = (1 << y) & a; // 00000000, false
```

28.3. Динамика по подмножествам

Основная идея в том, что мы можем посчитать значение функции для какого-то множества, зная значения функци для одного или нескольких его собственных (не совпадающих со всем множеством) подмножеств.

Это означает, что перебирать множества нужно в таком порядке, чтобы все подмножества перебираемого на текущем шаге множества уже встречались на предыдущих шагах.

Например, если задавать подмножества n-элементного множества беззнаковыми целыми числами от 0 до $2^n - 1$ (что подробно описано в предыдущей части), нас устроит порядок $0 \dots 2^n - 1$, поскольку любое (собственное) подмножество описывается меньшим числом.

Кроме того, для пересчёта динамики нужно научиться перебирать подмножества. О том, как перебирать все подмножества, будет сказано в одной из следующих глав, а для тех задач, которые будут рассмотренны в этой главе, достаточно переходить к подмножеству без минимального элемента. Научимся делать это, извлекая младший значащий бит.

28.3.1. Младший бит

Пусть число a в двоичной системе имеет вид $'\dots\overbrace{10\dots0}^{k \text{ бит}}'$. Тогда число $\sim a + 1$ совпадает с a ровно в последнийх k битах, а в остальных отличается. Поэтому если применить побитовое «и» к a и $\sim a + 1$, мы как раз получим последний кусок $'10\dots0'$.

```
1  unsigned char a = 184;           // 10111000
2  unsigned char b = a & (~a + 1); // 00001000
3 /*
4             a = 10111000
5            ~a = 01000111
6        ~a + 1 = 01001000
7 a & (~a + 1) = 00001000
8 */
```

Однако избавляться от младшего бита можно несколько проще: $a\&(a-1)$. Корректность этой операции легко проверить аналогичнми рассуждениями.

```
1  unsigned char a = 184;           // 10111000
2  unsigned char b = a & (a - 1);  // 10110000
3 /*
4             a = 10111000
5         a - 1 = 10110111
6  a & (a - 1) = 10110000
7 */
```

Рассмотрим несколько примеров.

28.3.2. Суммы на подмножествах

Даны n предметов. Вес i-го предмета — w_i. Требуется найти суммарный вес каждого подмножества предметов.

Сложность алгоритма — $\mathcal{O}(2^n)$.

```
1  sum[0] = 0;
2  for (int i = 0; i < n; ++i)
3    sum[1 << i] = w[i];
4
5  for (int mask = 1; mask < (1 << n); ++mask) {
6    int e = mask & (~mask + 1);
7    sum[mask] = sum[e] + sum[mask ^ e];
8  }
```

28.3.3. Количество единиц в числе

Для каждого числа от 0 до $2^n - 1$ посчитать количество единиц в его битовой записи. Задачу можно свести к предыдущей: теперь вес каждого предмета — 1. Но рассмотрим немного другое решение, тоже за $\mathcal{O}(2^n)$.

```
1  sum[0] = 0;
2
3  for (unsigned int mask = 1; mask < ((unsigned int)1 << n); ++mask) {
4    sum[mask] = sum[mask >> 1] + (mask & 1);
5  }
```

Здесь мы перешли не к подмножеству, а просто уменьшили число, сдвинув его на 1 вправо. С операцией '>>' нужно работать аккуратно, поскольку для знаковых чисел такой сдвиг — Implementation-defined behaviour.

28.4. Гамильтонов путь и цикл

Def 28.4.1. *Гамильтонов путь — простой путь, проходящий через каждую вершину графа ровно один раз.*

Def 28.4.2. *Гамильтонов цикл — замкнутый гамильтонов путь.*

28.4.1. Гамильтонов путь

Дан неориентированный граф на n вершинах. Найти в нём гамильтонов путь.

Простая динамика: $is_{A,v}$ — есть ли гамильтонов путь в подграфе, индуцированном на множестве вершин A (A — маска), оканчивающийся в вершине v ($v \in A$).

База: $A_{\{v\},v} = 1$; Переход: переберём все смежные с v вершины, лежащие в множестве A. $is_{A,v} = 1$ только в том случае, если для какой-то перебираемой вершины u (смежной с v) существовал гамильтонов путь, оканчивающийся в u и проходящий по множеству $A \setminus \{v\}$.

Сейчас требуется $\Theta(2^n n^2)$ времени и $\Theta(2^n n)$ памяти. Память можно сократить до $\Theta(2^n)$, если вместо двумерного массива is нулей и единиц хранить одномерный массив масок fin: fin_A — маска вершин, в которых может заканчиваться гамильтонов путь по множеству A.

Чтобы соптимизировать время до $\Theta(2^n n)$, будем хранить в битовом виде ещё и матрицу смежности: adj_v — множество вершин, смежных с v. Теперь заменим перебор всех соседей вершины на одну битовую операцию: $fin_{A,v} = fin_{A \setminus v} \& adj_v$ ($fin_{A,v}$ — v-й бит маски fin_A).

```
1  // fin = {0, ..., 0}
2  for (int i = 0; i < n; ++i)
3    fin[1 << i] = 1 << i;
4
5  for (int mask = 1; mask < (1 << n); ++mask)
6    for (int v = 0; v < n; ++v)
7      if (mask & (1 << v) && mask != (1 << v))
8        if (fin[mask ^ (1 << v)] & adj[v])
9          fin[mask] |= (1 << v);
```

28.4.2. Гамильтонов цикл

Зафиксируем вершину 0. В графе есть гамильтонов цикл тогда и только тогда, когда существует гамильтонов путь, начинающийся в вершине 0 и заканчивающийся в одной из смежных с ней вершин.

Задача нахождения гамильтонова пути, начинающегося в вершине 0 решается точно такой же динамикой.

Единственным отличием будет база:

```
1  fin[1] = 1;
```

28.4.3. Восстановление ответа

Поймём, как по посчитанной динамике восстановить ответ.

Мы для каждого подмножества знаем те вершины, в которых заканчивается гамильтонов путь, проходящий по этому подмножеству. Назовём такие вершины конечными для данного подмножества. Возьмём множество всех вершин и удалим из него любую конечную, предварительно добавив в ответ. Мы перешли к множеству, в котором меньше вершин на одну. Следующей вершиной в пути должна быть вершина, смежная с предыдущей, причём конечная для текущего подмножества. Такая, конечно, найдётся, ведь динамика посчитана корректно. Добавим её в ответ и удалим. Будем так делать до тех пор, пока вершин не останется.

Гамильтонов цикл восстанавливается аналогично, за исключением того, что первой вершиной выбирается не любая конечная, а смежная с 0.

Лекция по алгоритмам #29
Динамика на подмножествах. Раскраска графа.

26 октября

29.1. Постановка задачи

Дан граф на n вершинах. Требуется раскрасить вершины в минимальное количество цветов так, чтобы концы любого ребра имели различные цвета.

29.2. Решение за $\mathcal{O}(4^n)$

Назовём множество вершин "хорошим если не существует ребра, оба конца которого лежат в этом множестве. Заметим, что вершины из хорошего множества всегда можно покрасить в один цвет. Сначала предподсчитаем для каждого подмножества вершин, является ли оно хорошим. Сделаем это в лоб: переберём все подмножества вершин, переберём все возможные пары вершин из этого подмножества и проверим наличие ребра. Вершин в подмножестве не больше n, подмножеств 2^n, поэтому алгоритм будет работать за $\mathcal{O}(2^n n^2)$.

Теперь решим исходную задачу. Рассмотрим некотое подмножество вершин A. $dp[A]$ - в какое минимальное число цветов можно раскрасить подграф на вершиинах из A. Давайте выберем, какие вершины из A мы покрасим в первый цвет. Переберём все подмножества вершин и, если текущее (назовём его X) является подмножеством A и $good[X] == true$ (критерий того, граф на вершинах из X можно покрасить в 1 цвет), то прорелаксируем ответ для A значением $dp[A \backslash X] + 1$.

Итоговая асимптотика $\mathcal{O}(2^n \cdot n^2 + 2^n \cdot 2^n) = \mathcal{O}(4^n)$.

29.3. Эффективный перебор всех подмножеств заданного множества

Пусть надо перебрать все непустые подмножества множества A. Сделать это можно следующим кодом:

```
1    for (int X = A; X > 0; X = (X - 1) & A) {
2        ...
3    }
```

1) X точно подмножетво A т.к. мы на каждой итерации оставляем только те биты, которые были в A

2) На каждой итерации X уменьшается т.к. из него вычитают 1 и делают & с A (а эта операция не может увеличить число). Из этого, в частности, следует, что код отработает за конечное время.

3) Заметим, что на каждой итерации мы берём лексикографически максимальное подмножество, меньшее текущего (если сравнивать множества как двоичные записи кодирующих их чисел). Действительно, вычитая единицу мы заменяем последнюю 1 на 0, не трогаем всё что было до неё (следовательно новое подмножество меньше), а все доступные биты (т.е. те, которые были в A) после неё делаем 1 (поэтому среди всех меньших, полученное множество лексикографически максимальное). Тогда раз мы начали с лексикографически максимального и закончили минимальным, то мы переберём все подмножества.

A = 10111010011010
X = 10101010010000
$X-1$ = 10101010001111
$(X-1)\&A$ = 10101010001010

Этот алгоритм позволяет перебирать подмножества за их количество, а не за 2^n.

Замечание: пустое множество, если оно требуется, придётся разобрать отдельно.

29.4. Решение за $\mathcal{O}(3^n + 2^n n^2)$

Давайте теперь при переходе динамики перебирать только подмножества текущего множества. Оценим время работы. Если множество содержало k элементов, то количество его подмножеств 2^k. Сколько раз мы будем рассматривать множество, содержащее k элементов? C_n^k. Тогда суммарное время работы можно оценить как $\sum\limits_{k=0}^{n} 2^k \cdot C_n^k = \sum\limits_{k=0}^{n} 1^{n-k}2^k \cdot C_n^k = (1+2)^n = 3^n$.

С учётом предподсчёта (good[i]) итоговая асимптотика будет $\mathcal{O}(3^n + 2^n n^2)$.

29.5. Эффективный подсчёт массива good ($\mathcal{O}(2^n)$)

Примечнаие: Мы умеем находить за $\mathcal{O}(1)$ младший единичный бит числа, но не умеем находить его номер. Пока мы будем использовать массив index, который по числу будет хранить какой степенью двойки оно является ($index[2^n] = n$). т.к. мы будем использовать только n ячеек этого массива, то можно заменить его на HashMap. В одном из следующих конспектов будет рассказано, как обойтись без этого массива.

Заметим, что можно считать массив good более эффективно: выберем одну конкретную вершину из текущего множества, проверим, что из неё нет рёбер в это множество (если хранить соседей вершиы в таком же виде, как и множества, то это можно делать за 1 - просто проверить что пересечение этих множеств пусто) и проверим, с помощью обращения к уже посчитанной ячейке good, что множество без этой вершины хорошее. Итоговая асимптотика $\mathcal{O}(2^n)$, что позволяет уменьшить асимптотику решения основной задачи до $\mathcal{O}(3^n + 2^n) = \mathcal{O}(3^n)$

29.6. Перебор всех максимальных по включению независимых подмножеств

Вершины степени 0 входят во все максимальные независимые подмножества. Их возьмём сразу. Выберем вершину v с минимальной степенью. Пусть её степень d. Мы либо возьмём её в ответ (делаем рекурсивный вызов, выкинув её и её соседей из графа => вершин останется n - d - 1), либо возьмём одного из её соседей (степени соседей не меньше d, для каждого делаем рекурсивный вызов без вершины v, без текущего соседа и без его соседей => вершин останется не больше n - d - 1). Итого, если обозначить время работы на компоненте размера n, то $T(n) <= (d+1)T(n-d-1)$.

Докажем, что $T(n) = O(3^{n/3})$. Докажем по индукции.

$T(n) <= (d+1)T(n-d-1) <= (d+1) \cdot C \cdot 3^{\frac{n-d-1}{3}} = C \cdot 3^{\frac{n}{3}} \cdot (d+1)3^{-\frac{d+1}{3}} <= C \cdot 3^{\frac{n}{3}}$

Докажем последнее неравенство. Рассмотри функцию $f(x) = x \cdot 3^{-\frac{x}{3}}$.

$f'(x) = (x \cdot 3^{-\frac{x}{3}})' = 3^{-\frac{x}{3}-1} \cdot (3 - x\ln 3)$

Следовательно $f(x)$ убывает при x больших $\frac{3}{\ln 3}$ (это примерно 2,7).

Заметим, что $1 = f(3) > f(2) > f(1)$. Т.к. после 3 функция убывает, то и все последующие значения меньше 1.

29.7. Решение задачи о покраске графа за $\mathcal{O}(2.44^n)$

Теперь для каждого рассматриемого множества будем перебирать только его максимальные по включению независимые подмножества. Тогда время работы можно оценить как $\sum\limits_{k=0}^{n} 1.44^k \cdot C_n^k = \sum\limits_{k=0}^{n} 1^{n-k} 1.44^k \cdot C_n^k = (1 + 1.44)^n = 2.44^n$. Т.е. асимптотика станет $\mathcal{O}(2.44^n)$.

Замечание: 1.44 на самом деле $\sqrt[3]{3}$, 2.44, соответственно, $\sqrt[3]{3} + 1$

Лекция по алгоритмам #30
2 указателя

30 октября

Задача: есть n точек. Необходимо выбрать k точек так, чтобы сумма расстояний от каждой из исходных n точек до ближайшей выбранной была минимально возможной. Ранее мы решили задачу, используя следующую теорему: существуют оптимальные наборы точек, для которых выполняется условие: $p[k-1,n] \le p[k,n] \le p[k,n+1]$, где $p[k,n]$ - это координата середины отрезка между последней и предпоследней точками.

30.1. Доказательство

Для начала поймём, что каждая выбрання точка будет ближайшей для тех точек, которые находятся между серединами отрезков от неё до соседей назовём такой отрезок областью влияния выбранной точки.

Сведём первое неравенство ко второму. Предположим, что оно ложно, то есть в оптимальных наборах k и $k-1$ точек для некоторого исходного набора $p[k-1,n] > p[k,n]$. Возмём один набор точек и выберем для него сначала k, потом $k-1$ точку. Развернём оба набора точек. Посмотрим на границы отрезков влияния выбранных точек. Первая граница правее у второй выборки точек. Ясно, что при последовательном переходе к следующим границам настанет момент, когда либо границы отрезков в разных выборках сравняются, либо граница в первой окажется правее соответствующей границы во второй. В случае, наличия совпадающих границ отрезки левее них представляют собой различные решения одной и той же задачи, поэтому можно заменить часть точек в одной из выборок так, что она останется оптимальной, а разбиение на отрезки до совпадающих границ совпадёт со второй выборкой, то есть первое неравенство выполнится. Если же граница для первой выборки обгоняет вторую, отбросим точки правее соответствующей границы во второй выборке и получим противоречие с первым неравенством.

Теперь докажем второе неравенство ($p[k,n] \le p[k,n+1]$). Пусть q - координата последней выбранной точки.

Рассмотрим случай $q_2 \ge q_1$. Возьмём четыре набора точек. Первый - оптимальный выбор k точек для n исходных. Второй - $n+1$ исходная точка, $p = p_1$. Третий - оптимальный выбор k точек для $n+1$ исходной $p = p_2$, $q = q_2$. Четвёртый - снова n точек дано, $q = q_1$, $p = p_2$.

Обозначим f_i сумму расстояний от каждой точки до ближайшей выбранной в i-м наборе; F_1 - сумму расстояний для точек левее p_1 (то есть для точек до p в первых двух наборах), F_2 - сумму расстояний для точек левее p_2, (то есть для точек до p в двух последних наборах).

$$f_1 - f_4 = (F_1 - F_2) + (\sum_{i=p1}^{n} dist(p_i, q_1) - \sum_{i=p1}^{n} dist(p_i, q_1)) - \sum_{i=p_2}^{p_1-1} dist(i, q_1)$$

$$f_2 - f_3 = (F_1 - F_2) + (\sum_{i=p1}^{n+1} dist(p_i, q_1) - \sum_{i=p1}^{n+1} dist(p_i, q_1)) - \sum_{i=p_2}^{p_1-1} dist(i, q_2)$$

$((p_1..n - q_1) - (p_1..n - q_1)) = 0 = ((p_1..n + 1 - q_1) - (p_1..n + 1 - q_1))$ - расстояния от q_i до точек правее p_1.

$\sum_{i=p_2}^{p_1-1} dist(i, q_2) < \sum_{i=p_1}^{p_1-1} dist(i, q_1)$, т.к. $q_2 \ge q_1$ - расстояния от q_i до точек правее.

Отсюда $(f_1 - f_4) \ge (f_2 - f_3)$. Из оптимальности третьего набора $f_2 - f_3 \ge 0$. По транзитивности $f_1 - f_4 \ge 0 \Rightarrow f_1 \ge f_4$, следовательно, четвёртый набор оптимален. Мы нашли такой набор длины

n, для которого $p[n,k] \le p[n+1,k]$. Неравенство доказано.

Теперь рассмотрим случай $q_2 < q_1$. Снова возьмём черыре набора точек. Первый - оптисмальное решшение задачи для n точек. Второй - $n+1$ точка дана, $p = p_1$. Третий - решение задачи для $n+1$ точки. Четвёртый - n точек дано, выббранные - как в третьем. p_2 - p для двух последних наборов.

Изменения при переходе от первого набока ко второму - добавление одной точки, поэтому увеличивается на $x - q_1$, где x - координата добавленной точки. При переходе от второго набора к третьему сумма расстояний может только уменьшится, т.к. третий оптимален. При переходе от третьего набора к четвертому сумма расстояний опять уменьшается за счёт исчезновения последне точки, причём, поскольку $q_2 < q_1 < x$, модуль этого изменения больше, чем при первом переходе. Отсюда видно, что сумма расстояний в последнем наборе меньше, чем в первом, значит, мы нашли оптмальный ответ с $p \le p_2$. Неравенство доказано. document

Лекция по алгоритмам #31
Больше динамики по подмножествам

30 октября

31.1. Старший бит

Найти старший бит числа от 0 до $2^n - 1$.

a) Посчитать для всех. up[i] - старший бит числа i. $\mathcal{O}(2^n)$.

```
up[1] = up[0] = 0;
for (int i = 2; i < (1 << n); ++i)
  up[i] = up[i >> 1] + 1;
```

b) Только для X. $\mathcal{O}(\log X)$.

```
up = 0;
for (int i = 2; i <= X; i *= 2)
  up++;
```

31.2. Независимость

Def 31.2.1. *Независимое множество - такое множество вершин графа, что никакие 2 вершины из этого множества не соединены ребром.*

Для каждого множества вершин проверить, правда ли, что оно независимо.

is[A] - независимость множества с маской A, up - вершина в A с наибольшим номером, g[A] - соседи вершины up. A независимо, если A up независимо и из up в A нет рёбер, то есть is[A] = (is[A^(1 << up)] && ((g[up] & A) == 0)).

31.3. Сумма на подмножестве, рекурсивная версия

go - рекурсивная процедура, которая перебирает все множества и в процессе добавления элемента в множество пересчитывает сумму. Глубина рекурсии - n. Время - то же $\mathcal{O}(2^n)$.
i - номер элемента, A - текущее множество, S - текущая сумма.

```
go (i, A, S) {
  if (i == n)
    return;
  go (i + 1, A, S);
  go (i + 1, A | (1 << i), S + a[i]);
}
```

Минус - рекурсия даёт большую константу. Плюс - не нужно хранить все суммы.
Из-за меньшего объёма используемой памяти с большими данными будет работать быстрее.

31.4. Set Cover (аналог Рюкзака)

Дано большое множество A и k его подмножеств B_1, B_2, ..., B_k.
Нужно представить A в виде объединения минимального количества каких-то B.

Пусть в A m элементов. Можно рекурсией аналогично предыдущей задаче за $\mathcal{O}(2^k)$.
A можно за $\mathcal{O}(2^m k)$. f[i, C] - количество множеств из первых i, которыми можно покрыть C.
relax(f[i + 1, C], f[i, C]) - когда не берём B_i, relax(f[i + 1, C | B_i], f[i, C]) - когда берём B_i.

31.5. Динамика по надмножествам

```
1 for (A = 0; A < (2 << n); ++A)
2   for (B = A; B < (2 << n); B = (B + 1) | A)
3     ;
```

1) Любое B - надмножество соответствующего A. $B = ...|A \supset A$.

2) Для одного A каждое следующее B больше предыдущего. B = (B + 1) | ... $\geq$ B + 1 > B.

3) Корректность. Для каждого A соответствующие ему B - ровно все его надмножества:
Пусть A_1, ..., A_s - реальные упорядоченные по возрастанию (битовых масок) надмножества A. Тогда посмотрим, как из A_i получается A_{i+1}. Разделим A_i на старшие и младшие биты так, чтобы старшие заканчивались на 0, а все младшие были равны 1. Теперь разделим A и A_{i+1} на старшие и младшие биты так, чтобы их количества совпадали. Грубо говоря, выбрали у A, A_i и A_{i+1} одинаковые хвосты так, чтобы у A_i хвост был максимальным из состоящих только из единиц. Тогда старшие биты у A_{i+1} такие же, как у A_i, только последний из них - 1, а не 0, иначе A_{i+1} не было бы больше, чем A_i. При этом младшие биты A_{i+1} (хвост) такие же, как у A, иначе A_{i+1} не было бы минимальным включающим A из множеств с такими старшими битами.
Заметим, что это в точности соответствует методу выбора следующего B из цикла:
$A_{i+1} = (A_i + 1) \mid A$. $A_i + 1$ обеспечивает правильные старшие биты, а | A - правильные младшие.

4) Время - $\mathcal{O}(3^n)$. Заметим, что у каждого элемента на любом шаге может быть одно из трёх состояний: он в A (и, соответственно, в B); он не в A, но в B; он не в A и не в B. Значит, всего комбинаций состояний 3^n, то есть будет сделано 3^n шагов.

Если мы умеем перебирать подмножества, то умеем и перебирать надмножества:
Для любого B надмножества A: (Всё \ B) - подмножество (Всё \ A). И наоборот.

Кроме того, если в множестве A m элементов, то у A ровно 2^{n-m} надмножеств.

Лекция по алгоритмам #32
Динамика по скошенному профилю

30 октября

32.1. Разбиение доски на доминошки

Дана доска со сторонами W и H. Посчитать количество способов разбить её на доминошки.

```
int go (x, y) {
  if (x + 1 == W) {              \\ 1)
    x = 0;
    y++;
  }
  if (y == H)                   \\ 2)
    return 1;
  if (A[x, y])                  \\ 3)
    return go(x + 1, y);
//  if (m.count(x, y, A))            \\ *)
//    return M[x, y, A];
  int res = 0;
  if ((x + 1 < W) && (A[x + 1, y] == 0)) {  \\ 4)
    A[x, y] = A[x + 1, y] = 1;
    res += go(x + 1, y);
    A[x, y] = A[x + 1, y] = 0;
  }
  if ((y + 1 < H) && (A[x, y + 1] == 0)) {  \\ 5)
    A[x, y] = A[x, y + 1] = 1;
    res += go(x, y + 1);
    A[x, y] = A[x, y + 1] = 0;
  }
  return M[x, y, A];
}
```

Возвращаемое значение - количество способов покрыть всё оставшееся доминошками.
A - массив покрытости клеток. M - хэш таблица, в которой храним ответы.
Рекурсивный перебор за $\mathcal{O}(2^{\frac{HW}{2}})$. Идем по строкам. Покрываем клетку (x, y).

1) Если дошли до конца строки - переходим на следующую.

2) Если строки закончились - конец. Покрыть ничто можно одним способом.

3) Если клетка уже покрыта - возвращаем результат для следующей.

4) Если есть свободная клетка справа, то можно поставить горизонтальную доминошку: Считаем клетки этой доминошки покрытыми, запускаем рекурсию, освобождаем клетки.

5) Если есть свободная клетка снизу, то можно поставить вертикальную доминошку: Аналогично.

*) Если проверять, посчитано ли уже это значение, то будет не перебор, а (ленивая) динамика по профилю, которая работает уже за $\mathcal{O}(2^W HW)$.

Про M:

1) Можно хранить весь A как число (бит - покрытость клетки), если $W \cdot H \leq 128$ (__int128).

2) A - vector<vector<int>>, M - map<vector<vector<int>>, int>.

3) M[A][x][y] или tuple<__int128, int, int>. Полиномиальный хэш.

32.2. Красивые матрицы

Def 32.2.1. *Красивая матрица - матрица, в которой нет квадрата 2x2 из нулей.*

Есть длинная матрица $5 \cdot 10^9$ из нулей и единиц. Сколько в ней красивых матриц?

Динамика. f[i,C] - сколько способов заполнить оставшуюся часть матрицы, если уже заполнено i столбцов, причём последний из них - C. is[C_1, C_2] - можно ли после столбца C_1 поставить столбец C_2. Считаем количество матриц, в которых за i-м столбцом C_1 следует i+1 -й C_2, если до i включительно всё посчитано.
f[i + 1, C_2] = $\sum_{C_1}$(f[i, C_1] · is[C_1, C_2]), то есть f[i + 1] = f[i] · is для матриц f и is.
f[10^9] = f[0] · (is$^{(10^9)}$), а это можно сделать за $(2^5)^3 \cdot \log_2 10^9$ (2^5 - размер матрицы is).

Лекция по алгоритмам #33
Топологическая сортировка и т.п.

9 ноября 2015г.

33.1. Что хотим?

Дан ориентированный граф. Хотим найти такую перестановку вершин, что любое ребро идёт "слева направо". Она и называется топологической сортировкой.

Сразу заметим, что если в графе есть цикл, то топологической сортировки не существует. Пусть у нас цикл v -> u —> v (ребро v -> u, какой-то путь u —> v).

Пусть правильный порядок - это ... v ... u ... Но все рёбра пути u —> v должны идти слева направо, то есть правильный порядок - это ... u ... v ... Но есть ребро v -> u, значит, правильный порядок - это всё-таки ... v ... u ...

В общем, в любом случае противоречие, так что в графе с циклами топологической сортировки действительно нет.

33.2. Наивный алгоритм ($\Theta(n^2)$)

Давайте как-нибудь научимся искать топологическую сортировку. Сделаем n итераций, на каждой выберём вершину с входящей степенью 0 (степень, разумеется, считаем по вершинам, ещё не добавленным в топологическую сортировку), добавляем её в топологическую сортировку. Если есть несколько вершин входящей степени 0, то можем выбрать ту, которая нам нужна, может быть, по условию (например, с минимальным номером, или, опять же,по критерию, указанному в условии задачи). Вершину входящей степени 0 мы берём просто потому, что все вершины, которые будут идти позже выбранной, не имеют рёбер в выбранную (просто потому, что в выбранную вообще сейчас нет рёбер).

```
for (int it = 0; it < n; it++)
{
    int best = inf;
    for (int j = 0; j < n; j++)
      if (!deg_in[j] && (best == inf || our_comporator))
        best = j;
    for (int j = 0; j < n; j++)
      deg_in[j] -= e[best][j]; //e[i][j] is a number of edges from i to j
    cout << best << "\n";
    deg_in[best] = inf;
}
```

33.3. Более быстрый алгоритм нахождения топологической сортировки ($\Theta(n+m)$)

Заметим, что для любой вершины все её соседи идут позже в топологической сортировке. Получаем алгоритм

```
void dfs(int v)
{
  used[v] = 1
  for (int u : e[v])
    if (!used[u])
```

```
6       dfs(u);
7   topsort.push_front(v);
8   // or push_back and reverse(topsort.begin(), topsort.end()); after dfs
9 }
```

33.4. Проверка на наличие циклов

Пусть не гарантируется, что граф ацикличен. Научимся одним dfs'ом проверять наличие циклов и строить топологическую сортировку.

```
1  bool dfs(int v)
2  {
3    used[v] = 1
4    for (int u : e[v])
5      if (!used[u])
6        dfs(u)
7      else
8        if (used[u] == 1)
9          return 0
10   topsort.push_front(v)
11   used[v] = 2
12 }
```

Красим вершины в 3 цвета: 0 - ещё не заходили (белый), 1 - уже зашли, но не вышли (серый), 2 - уже вышли (чёрный). Тогда понятно, что цикл есть, если мы из серой вершины попали в серую. dfs возвращает bool - правда ли, что граф ацикличен.

33.5. Топологическая сортировка и динамическое программирование

Пусть мы хотим найти количество путей из start в end. Понятно, что задача решается динамическим программированием. Также понятно, что правильный порядок обхода состояний - это топологическая сортировка (или её развёрнутый порядок).

33.5.1. DP

```
1  cnt[end] = 1;
2  for (int i = n; i >= 1; i--)
3  {
4    v = topsort[i];
5    for (int u : e[v])
6      cnt[v] += cnt[u]
7  }
```

33.5.2. Lazy DP

```
1  fill(cnt, cnt + n, -1);
2  cnt[end] = 1;
3  int dfs(int v)
4  {
5    if (cnt[v] != -1)
6      return cnt[v];
7    cnt[u] = 0;
```

```
8   for (int u : e[v])
9     cnt[v] += dfs(u)
10  return cnt[v];
11 }
```

33.6. Сильная связность

Ориентированный граф сильно связен, если между любой (упорядоченной) парой вершин есть путь.

33.7. Алгоритм нахождения компонент сильной связности

1. Запустим поиск топологической сортировки, игнорируя то, что её может и не существовать.

2. Будем красить по обратным рёрбам. То есть берём непокрашенную вершину, красим её и всё, что из неё достижимо по обратным рёбрам (в новый цвет).

```
1  void dfs(int v)
2  {
3    clr[v] = color;
4    for (int u : ee[v])
5      if (!clr[u])
6        dfs(u);
7  }
8  color = 0;
9  for (int i = 1; i <= n; i++)
10   if (!clr[topsort[i]])
11   {
12     color++;
13     dfs(topsort[i]);
14   }
```

33.8. Почему работает?

Давайте поддерживать инвариант:

a: все вершины v, для которых scc(v) = scc(j) (j < i) уже покрашены до того, как мы начали работать с i-той вершиной в порядке топологической сортировки.

b: Никакая u: ∀ j < i: scc(u) != scc(j) не покрашена.

Смотрим на i-тую вершину в порядке топологической сортировки (для удобства называть её тоже будем i). Если i не покрашена, то вся scc(i) достижима из i по обратным рёбрам. Покрасим scc(i). Покажем, что лишние вершины не будут покрашены.

От противного: ∃ x, покрашенная из i, но scc(x) != scc(i).

Смотрим на scc(x). Пусть y - вершина, в которую мы зашли первым dfs'ом (тем, который собирает topsort) раньше всего. Покажем, что y в topsort раньше, чем scc(i). Вошли первым dfs'ом в y. scc(i) либо целиком чёрная, либо целиком белая. Если scc(i) целиком серая, то из scc(i) есть путь в y, которая находится в scc(x), из которой есть путь в scc(i). Получаем противоречие. Частичные цвета тоже невозможны, т.к. компоненты мы красим сразу и целиком.

Пусть scc(i) вся чёрная => в топологической сортировке y будет раньше => y уже посещён, а значит, scc(y) уже покрашена.

Пусть scc(i) вся белая => есть путь из y в scc(i), поэтому scc(i) станет чёрной раньше, чем y => в топологической сортировке y идёт раньше, чем scc(i).

В любом случае имеем противоречие (с инвариантом и в целом с ситуацией, которую мы рассматриваем).

Лекция по алгоритмам #34
Эйлеровые графы, циклы и сопутствующие алгоритмы

13 ноября

Def 1: Эйлеров цикл - цикл по всем рёбрам графа **Def 2:** Эйлеров путь - путь по всем рёбрам графа

Критерии наличия эйлерова пути:

s, t - вершины ориентированного графа

1)$deg_{in}(t) - deg_{out}(t) = 1$

$deg_{out}(t) - deg_{in}(s) = 1$

$\forall x$(x - оставшиеся вершины графа) $deg_{out}(x) - deg_{in}(x) = 0$

2)Связность графа

Док-во: 1) Из s в t можно пройти.(если мы вошли в какую-то вершину, то из неё мы можем выти, возможно за исключением случая вершины t, но бесконечного пути быть не может, значит он закончится в t)

2)Возьмем максимальный путь и скажем, что он эйлеров(От противного, пусть нет, тогда т.к. граф связен, то в одну из вершин не взятых в путь есть входящие и сходящие рёбра, но теперь т.к. у всех вершин $deg_{out}(x) - deg_{in}(x) = 0$, то идя по невзятым рёбрам мы вернемся в данную вершину, а значит сможем увеличить путь).

Алгоритм поиска:

```
1  struct Edge{
2    int to, id;
3  }
4
5  deque<int> path;
6  vector<Edge> g[N];
7
8  dfs(v){
9    while(g[v].size() > 0){
10     Edge e = g[v].back();
11     g[v].pop_back;
12     dfs(e.to);
13     path.push_front(e.id);
14   }
15 }
```

Что же делать, если граф неориентированный?

1)Заметим, что он останется связным.

2) Запоминаем номера одинаковых рёбер.

3)При необходимости лениво удаляем.

34.1. Задача(De Brujiu sequence):

Имеется Алфавит, все слова которого сроки длины n, состоящие из 0 и 1. Существует ли строка содержащая все возможные слова как подстроки.

Решение:

Ответ: Содержит.

Почему:

Доказательство. 1)Рассмотрим граф, в котором вершины - это все возможные подстроки длины $n-1$, пусть из вершины в вершину есть рёбро, если одна является префиксом слова, а другая суффиксом.

2)Заметим, что из каждой вершины выходит 2 ребра и в каждую входит. Кроме того граф связен, т.к. можно за n действий всегда превратить одну строку в другую.

3)начит у нас существует эйлеров путь по всем рёбрам(Считаем началом концом одну и ту же вершину, можно разделить мысленно на 2, из одной будет исходить 2 ребра в входить одно, а во вторую будет входить оставшееся). В графе 2^{n-1} вершинб из каждой выходит 2 ребра.

4)Всего рёбер 2^n. Заметим, что каждое ребро задает строку длины n. Это значит, что наши рёбра это ровно 2^n слов алфавита т.к. префикс и суффикс в паре уникальны.

Итого: Раписав начальную вершину и при переходе записывая соответствующий новую цифру мы получим нашу строку длины $(n-1)+2^n$.

■

Лекция по алгоритмам #35
Компоненты двусвязности

13 ноября

35.1. Поиск компонент реберной двусвязности.

Def 35.1.1. *Реберная двусвязность:*
Две вершины называются ***реберно двусвязными****, если между ними существуют два реберно непересекающихся пути*

⇔ *они лежат на реберно простом цикле.*

⇔ *при удалении любого ребра они останутся в одной компоненте связности.*

Теорема 35.1.2. *Реберная двусвязность - отношение эквивалентности.*

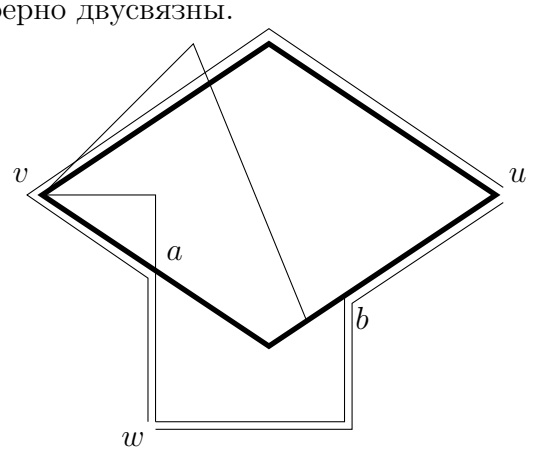
• **Доказательство:**

Рефлексивность и симметричность очевидны.

Докажем транзитивность.
(u, v), (v, w) - реберно двусвязны.

Доказать: (u, w) - реберно двусвязны.

Из w в v есть два реберно непересекающихся пути, P_1 и P_2 соответственно. Обозначим за C объединение двух реберно непересекающихся путей из u в v. C будет реберно-простым циклом. Пусть вершины a и b - первые со стороны w вершины на пересечении P_1 и P_2 с C соответственно. Рассмотрим два пути $w - a - u$ и $w - b - u$, такие, что части $a - u$ и $b - u$ идут в разные стороны по циклу C. Наличие двух таких реберно непересекающихся путей очевидно, а значит u и w реберно двусвязны.



$\forall e = (u, v)$: e - мост ⇔ u и v в разных компонентах реберной двусвязности.

• **Алгоритм 1:**

Нашли мосты, удалили мосты. Компоненты связности в итоговом графе - это компоненты реберной двусвязности в исходном.

$\mathcal{O}(V + E)$

• **Однопроходный алгоритм со стеком:**

Алгоритм строится на базе алгоритма поиска мостов. Во-первых, создадим глобальный стек, и при спуске по дереву dfs добавляем в него вершины. Во-вторых, когда возвращаемся назад, проверяем не является ли ребро мостом. Если это так, то все вершины, находящиеся до текущего потомка в стеке, принадлежат одной компоненте. Заметим, что эта компонента будет висячей вершиной в дереве конденсации, так как обходили граф поиском в глубину. Значит, ее можно выкинуть и продолжить поиск в оставшемся графе. Действуя по аналогии в получившемся графе, найдем оставшиеся компоненты реберной двусвязности.

Код:

```
    void newComp(size_t size = 0) {
        comp.emplace_back(); // create new empty component
        while (st.size() > size) {
            comp.back().push_back(st.back());
            st.pop_back();
        }
    }

    void find_bridges(int v, int parentEdge = -1) {
        if (up[v] != 0) { // visited
            return;
        }

        up[v] = tin[v] = ++timer;
        st.push_back(v); // st - stack

        for (edge e : g[v]) {
            if (e.id == parentEdge) {
                continue;
            }

            int u = e.to;

            if (tin[u] == 0) {
                int size = st.size();
                find_bridges(u, e.id);
                if (up[u] > tin[v]) {
                    newComp(size);
                }
            }

            up[v] = min(up[v], up[u]);
        }
    }

    int main() {
        ...
        fori (n) {
            if (up[i] == 0) {
                find_bridges(i);
                newComp(); // dfs root component
```

```
42              }
43          }
44          ...
45      }
```

$\mathcal{O}(V+E)$

35.2. Поиск компонент вершинной двусвязности.

Def 35.2.1. *Вершинная двусвязность:*
Два ребра графа называются ***вершинно двусвязными****, если между ними существует два [вершинно] непересекающихся пути*
$\Leftrightarrow$ *они лежат в одном простом цикле.*
$\Leftrightarrow$ *при удалении любой вершины они останутся в одной компоненте связности.*

Пусть $e=(u,v),\ g=(v,w)$ // два смежных ребра
e и g в разных компонентах вершинной двусвязности $\Leftrightarrow$ нет пути $(u-w)$ без v.

Доказательство:

$\Leftarrow$:
(очевидно)

$\Rightarrow$:
От противного: пусть они в одной компоненте $\Rightarrow \exists$ два вершинно-непересекающихся пути $\Rightarrow$ один не содержит v.

Замечание:
e и g в разных компонентах $\Rightarrow v$ - точка сочленения.
v - точка сочленения $\not\Rightarrow$ e и g в разных компонентах.

Теорема 35.2.2. *Вершинная двусвязность - отношение эквивалентности.*

- **Доказательство:**

Рефлексивность и симметричность очевидны. Доказательство транзитивности аналогично с отношением реберной двусвязности. Идем до первого пересечения с циклом, после чего идем по циклу в разные стороны.

- **Алгоритм поиска за $\mathcal{O}(V+E)$:**

Алгоритм строится на базе алгоритма поиска точек сочленения. Во-первых, первый раз проходя по ребру, будем класть его на стек. Во-вторых, если видим, что спустившись по ребру нельзя подняться выше текущей вершины по дереву dfs(это говорит, что вершина - точка сочленения), то помечаем все, что лежит на стеке до этого ребра как новую компоненту. Отдельно, как и в алгоритме поиска точек сочленения, стоит рассмотреть случай корня dfs.

Код:

```
1   void newComp(size_t size = 0) {
2       while (st.size() > size) {
3           color[st.back()] = compCnt;
4           st.pop_back();
```

```
5         }
6         ++compCnt;
7     }
8
9     void find_points(int v, bool root = true) {
10        if (up[v] != 0) {
11            return;
12        }
13
14        up[v] = tin[v] = ++timer;
15        int cnt = 0;
16
17        for (edge e : g[v]) {
18            if (used[e.id]) {
19                continue;
20            }
21
22            int u = e.to;
23            st.push_back(e.id); // st - stack
24            used[e.id] = true;
25
26            if (tin[u] == 0) {
27                ++cnt;
28                size_t size = st.size() - 1; // minus this edge
29
30                find_points(u, false);
31
32                if (up[u] >= tin[v] && (!root || cnt >= 2)) {
33                    newComp(size);
34                }
35
36                up[v] = min(up[v], up[u]);
37            } else {
38                up[v] = min(up[v], tin[u]);
39            }
40        }
41
42        if (root && !g[v].empty()) {
43            newComp();
44        }
45    }
46
47    int main() {
48        ...
49        fori (n) {
50            find_points(i);
51        }
52        ...
53    }
```

35.3. Про 2-sat:

Есть формула в 2-КНФ.
Хотим найти выполняющий набор или сказать, что формула противоречива.

Построим граф. Вершины графа - литералы (x или $\neg x$, где x - переменная).
Для каждого дизъюнкта $(a \vee b)$ добавим в граф ребра:
$\bar{a} \to b$
$\bar{b} \to a$

Формула противоречива $\Leftrightarrow \exists x$: x и $\neg x$ лежат в одной компоненте сильной связности.

• Замечание 1:

В таком графе если есть путь из литерала la в литерал lb, то есть и путь из $\neg lb$ в $\neg la$.

• Замечание 2:

По сути, мы заменили один дизъюнкт на две импликации. Таким образом, набор значений переменных выполняющий $\Leftrightarrow$ нет ребер из выполненных литералов в невыполненные.

• Решение:

Сделаем конденсацию графа.
Проверим формулу на выполнимость(что $\forall x$: x и $\neg x$ в разных компонентах сильной связности).
Оттопсортим.
Пойдем по компонентам в обратном топсорту порядке.
Берем любой литерал (lx). Хотим его выполнить(если $lx = x$, то установить $x = 1$, иначе $(lx = \neg x)$, установить $x = 0$). Если нет противоречий, выполняем все литералы этой компоненты. Иначе - не трогаем эту компоненту и идем дальше.

• Утверждение 1:

Или мы сможем выполнить все литералы одной компоненты, или ни одного(для всех переменных этой компоненты уже были установлены противоположные значения).

Доказательство:
$\forall x, y : lx$ и ly в одной компоненте $\Rightarrow \neg ly$ и $\neg lx$ тоже лежат в одной компоненте.

• Утверждение 2:

Если мы не смогли выполнить компоненту A, то мы не сможем выполнить и ни одну компоненту, из которой есть ребра в A.

Доказательство:
Предположим обратное. Рассмотрим первый случай, когда все плохо.
$\forall lx \in A$ знаем, что $\neg lx$ встречался ранее. Значит, для любого ребра $ly \to lx : \neg ly$ тоже встречался ранее. Причем существует ребро $\neg lx \to \neg ly$ и $\neg lx$ был выполнен. Тогда и $\neg ly$ был выполнен, т.к. мы рассматриваем первый плохой момент. $\Rightarrow ly$ не будет выполнен.

Получили, что для любого ребра $lx \to ly$ если выполнен литерал lx, то выполнен и $ly \Rightarrow$ полученный набор значений переменных выполняющий.

Лекция по алгоритмам #36
BFS, Dijkstra

20 ноября

36.1. Рекламная пауза

А что мы знаем о поиске кратчайших путей?

1. Невзвешенный граф $\Rightarrow \mathcal{O}(V+E)$ — BFS
2. Взвешенный
   a. $w_i \geqslant 0$ — Дейкстра (Dijkstra):
      1. $E^2 + V$
      2. $E \log V$ (heap)
      3. $E + V \log V$ (Fib. heap)
      4. $(E + V \log C)$ OR $(E + V\sqrt{\log C})$ ($w_i \in \{1 \dots C\}$, radix heap)
   b. $w_i \in \mathbb{Z}+$ Неорграф $\Rightarrow \mathcal{O}(V+E)$ — Thorup (2002)
   c. $w_i \in \mathbb{Z}+$ Орграф $\Rightarrow \mathcal{O}(E + V \log\log V)$
   d. Нет отр. циклов - алгоритм Форда-Беллмана ($\mathcal{O}(VE)$) $\rightarrow$ алгоритм Гольдберга (1993) ($\mathcal{O}(E\sqrt{V}\log N)$)

36.2. BFS

Ищет путь от заданной вершины s до всех остальных на невзвешенном графе. На i-м шаге рассматриваются вершины множества A_i - вершины на расстоянии i от s. Реализация:

1. Так делать не нужно

```
1  vector<vector<int>> a(n);
2  a[0].push_back(s), d[s] = 0;
3  for (int i = 0; i < n; ++i) {
4      // Calc for a[i + 1] using a[i]
5  }
```

2. На очереди

```
6  memset(d, -1, sizeof(d));
7  queue<int> q;
8  for (q.push(s), d[s] = 0; !q.empty(); ) {
9      int u = q.front(), q.pop();
10     for (int v: edges[u]) {
11         if (d[v] == -1) {
12             d[v] = d[u] + 1;
13             q.push(v);
14         }
15     }
16 }
```

3. Заметим, что на основе этого можно сделать Недо-DFS без рекурсии

```
17 memset(d, -1, sizeof(d));
18 stack<int> q;
19 for (q.push(s), d[s] = 0; !q.empty(); ) {
20     int u = q.back(), q.pop();
21     for (int v: edges[u]) {
22         if (d[v] == -1) {
23             p[v] = u;
24             d[v] = 0;
25             q.push(v);
26         }
27     }
28 }
```

Time: $\mathcal{O}(V + E)$

36.2.1. Граф кратчайших путей из s в t

1. $BFS(s)$
2. оставим ребра, такие что $d[v] = d[u] + 1$
3. $DFS\_reversed(t)$;

36.2.2. Матрица расстояний ($d[s][t]$), транзитивное замыкание ($exists\_path[s][t]$)

Очевидно: $\mathcal{O}(V \cdot (V + E))$ — BFS из каждой вершины

Давайте искать замыкание чуть быстрее — за $\mathcal{O}(\frac{V \cdot (V+E)}{\omega})$, где ω - машинное слово

1. bitset<N> exists_path[N];
2. решим для конденсации (которую построим, конечно же, за $\mathcal{O}(V + E)$)
3. получен ориентированный ациклический граф, будем считать по нему динамику: $is[u] = \left(\bigcup\limits_{u \to v} is[v]\right) \cup u$

36.2.3. Форд-Беллман

Испортим BFS так, что он превратится в алгоритм Форда-Беллмана:

```
29 memset(d, -1, sizeof(d));
30 queue<int> q;
31 for (q.push(s), d[s] = 0; !q.empty(); ) {
32     int u = q.front(), q.pop();
33     for (int v: edges[u]) {
34         if (d[v] == -1 || d[v] > d[u] + w[u][v]) {
35             d[v] = d[u] + w[u][v];
36             q.push(v);
37         }
38     }
39 }
```

36.2.4. 0-1-BFS ($w_i \in \{0, 1\}$)

Сначала обрабатываем нулевые ребра, затем единичные, то есть:
$w[u][v] = 0 \to q.push\_front(v)$
$w[u][v] = 1 \to q.push\_back(v)$
$\mathcal{O}(V + E)$, так как мы добавим не более одного раза с каждой стороны

36.2.5. 1-K-BFS ($w_i \in \{1, \dots K\}$)

1. Делим ребро длины l на путь длины l через фиктивные вершины и пускаем по ним обычный BFS ($\mathcal{O}(V + KE)$)

2. $K + 1$ очередь. Последовательно рассматриваем пути длины от 0 до $(V - 1)K$. Пусть мы сейчас смотрим на путь длины d, тогда наши очереди представляют собой очереди для $d, d + 1, \dots, d + K$. Идем по ребрам вершин, для которых ответ не больше d, в очереди для d и закидываем потомков в очереди $d + \underset{u \to v}{w}$, обновляя ответ.

Лекция по алгоритмам #37
Алгоритм Дейкстры, A*

20 сентября

37.1. Анонс

Алгоритм Дейкстры ищет расстояния от данной вершины s до всех вершин в графах без отрицательных ребер. Наивная реализация работает за $\mathcal{O}(|V|^2)$, но её очень просто ускорить, используя известные нам структуры данных. В общем-то алгоритм Дейкстры - очень похож на известные нам bfs, но вершины просматриваются стого в порядке расстояния от стартовой, и засчёт этого каждая вершина просматривается ровно один раз.

Я тут всё расписываю максимально подробно. Если кому-то это не надо - смотрите только код, там есть поясняющие комментарии. Если кому-то надо ещё подробнее - пусть обратится к психиатру.

37.2. Алгоритм Дейкстры, преамбула

- $d[v]$ - минимальное найденное расстояние от стартовой вершины s до вершины v. Изначально полагаем $d[v] = \infty$ $d[s] = 0$. $d[v]$ является верхней оценкой на расстояние, то есть $d[v]$ никогда не меньше истинного расстояния. В конце работы алгоритма все $d[v]$ равны истинным расстояниям.
- "Обработать вершину u" значит для всех вершин v, смежных с u положить $d[v] = min(d[v], d[u] + w)$, где w - вес ребра из u в v ($relax$ по всем соседям). **Мы будем обрабатывать вершину только если расстояние до неё уже посчитано окончательно, т.е. соответствует истинному.**
- $used[v]$ - флаг, который говорит, что вершина v уже обработана.
- $e[v]$ - множетво ребер, исходящих из вершины v. $e[v][i].to$ - конец i-го ребра, $e[v][i].weight$ - вес i-го ребра.
- A - множество обработанных вершин.

Предыдущие алгоритмы поиска кратчайших путей обрабатывали каждую вершину много раз и работали в худшем случае $\mathcal{O}(VE)$. Алгоритм Дейксты обрабатывает каждую вершину только один раз и поэтому работает заметно быстрее.

Идея алгоритма основана на следующем утверждении:

Лемма: Пусть $u \notin A$, $d[u]$ минимально среди всех ещё не обработанных вершин. Тогда расстояние $d[u]$ уже посчитано окончательно, то есть соответствует истинному расстоянию.

В самом деле, рассмотрим любой путь $s \rightsquigarrow u$. Пусть v - первая не обработанная вершина на этом пути, w - последняя обработанная вершина на этом пути, а l - длина пути.

1. $v = u$. Так как w уже обработана, $l \geq d[w] + e[w \to v] \geq d[u]$.
2. $v \neq u$. Наш путь состоит из двух частей: $s \rightsquigarrow v$ и $v \rightsquigarrow u$. Длина первой части пути не меньше $d[u]$ - так как в противном случае $d[v] < d[u]$, что противоречит минимальности $d[u]$. Длина второй части пути неотрицательна, так как все ребра в графе неотрицательны. Значит сумма длин двух частей не меньше $d[u]$.

Получается длина любого пути не меньше $d[u]$. Это, конечно, значит, что $d[u]$ посчитано окончательно.

37.3. Алгоритм Дейкстры, алгоритм

Из предыдущей леммы сразу видно алгоритм: на каждом шаге берём ещё не обработанную вершину u с минимальным $d[u]$ и обрабатываем. Вот псевдокод:

```
u = s; //изначально у нас одна вершина с правильно посчитанным расстоянием - s
while (u != NIL){
    for(edge in e[u])          //обрабатываем вершину u
        relax(edge.to, d[u] + edge.weight);

    u = find_min(); //находим новую вершину с минимальным d
}
```

Время работы алгоритма равно $\mathcal{O}(|V| \cdot find\_min + |E| \cdot relax)$. Действительно, циклы for перебирают все исходящие ребра всех вершин, их $\mathcal{O}(E)$. А перед каждым вызовом $find\_min$ количество обработанных вершин увеличивается на один, поэтому всего $\mathcal{O}(|V|)$ $find\_min$.

37.4. relax и find_min

Время работы алгоритмы зависит от этих двух функций, посмотрим, что мы можем с ними сделать. $relax$ - это, очевидно, $DecreaseKey$, а $find\_min$ - $ExtractMin$ (на множествах ещё не обработанных вершин).

1. **Дешево, сердито**. Никаких структур, $relax$ просто обновляем d, вершину с минимальным d находим простым циклом:

```
void relax(int v, int val){
    d[v] = min(d[v], val);
}

int find_min(){
   int res = 0; // 0-я вершина - фиктивная, d[0] - бесконечность
   for(int i = 1; i <= n; i++)
       if (!used[i] && d[i] < d[res]) res = i;
   used[res] = true;

   return res;
}
```

Плюсы: ничего сложного, если $|E| \approx |V|^2$, то время работы оптимально

Минусы: если ребер не слишком много, то, очевидно, это слишком медленно

2. **Двоичная куча** Тут некоторые сложности с $DecreaseKey$. Немного поколдовав над двоичной кучей, можно получить доступ к произвольным элементам, и тогда будет достаточно просто изменить нужный элемент и просеять его вверх. Но, сделав так, мы лишимся огромного плюса двоичной кучи: она есть в STL.

   Поэтому будем делать по другому. В куче будем хранить пары $(u, d[u])$. Вместо того, чтобы обновлять элемент, будем добавлять новый. То есть $relax$ выглядит так:

   ```
   void relax(int u, int val){
        Insert( {u, val} );
    }
   ```

   А $find\_min$ выглядит так:

   ```
   while (used[u]) u = ExtractMin()->first;
   used[u] = true;
   ```

   В куче теперь будут находиться лишние элементы. Но их не более чем $\mathcal{O}(|E|)$, потому что общее количесво вызовов $relax$ равно $\mathcal{O}(|E|)$. Действительно, каждая вершина u обрабатывается один раз и порождает $degree(u)$ вызовов $relax$.

   $|E| = \mathcal{O}(|V|^2)$, поэтому все запросы к куче будут по прежнему выполняться за $\mathcal{O}(\log |V|)$.

   Всего запросов будет $\mathcal{O}(|E|)$. Поэтому общее время работы будет $\mathcal{O}(|E| \log |V|)$

   Плюсы: не надо писать ничего лишнего, относительно хорошее время

   Минусы: время можно ещё улучшить

3. **k-Heap**

   Весьма достойный вариант в случае критичности времени. Всё то же, что и в предыдущем пункте. Но раз уж всё равно в STL нет, можно не жульничать и делать $DecreaseKey$ честно, поддерживая доступ к прозвольным элементам.

   $DecreaseKey$ - реализуется через $SiftUp$, $ExtractMin$ - через $SiftDown$. Так как $DecreaseKey$ у нас $\mathcal{O}(|E|)$ штук, а $ExtractMin$ - $\mathcal{O}(|V|)$, то нам как раз выгодно, чтобы $SiftUp$ работал быстро, а $SiftDown$ можно немного замедлить. Поэтому $k - Heap$ выигрывает перед обычной двоичной кучей.

   Плюсы: хорошее время на практике: $\mathcal{O}(|E| \log_k |V| + k|V| \log_k |V|)$

   Минусы: нет в STL, теоретически можно ещё быстрее

4. **Фибоначчиева куча** Используя Фибоначчиеву кучу, можно добиться асимптотики $\mathcal{O}(|V| \log |V| + |E|)$. Это в некотором смысле предел для алгоритма Дейкстры.

   Мы не можем не просмотреть всех ребер, поэтому время работы не может быть меньше $\mathcal{O}(|E|)$. Кроме того, в алгоритме Дейкстры мы обрабатываем вершины в порядке сортировки по расстоянию. В общем случае отсортировать быстрее, чем за $\mathcal{O}(|V| \log |V|)$ мы не можем.

То есть, не пользуясь никакой дополнительной информацией (например, что веса ребер - целые числа от 1 до C), невозможно реализовать алгоритм Дейкстры быстрее.

Плюсы: оптимальная асимптотика

Минусы: все мы знаем, насколько на самом деле быстры Фибоначчиевы кучи...

37.5. A*, преамбула

Модифицируем алгоритм Дейкстры, чтобы он быстро работал на реальных (случайных, с некоторыми дополнительными свойствами) графах. Часто нам нужно найти расстояние только между двумя вершинами, а не от одной до всех остальных.

В этом случае можно сделать так, что алгоритм будет не равномерно находить расстояния до всех вершин вокруг стартовой, а как бы двигаться в сторону нужной вершины, не уходя далеко в сторону. Тогда мы сможем найти нужное расстояние раньше.

Для этого нам понадобится функция f - оценка расстояния до конечной вершины.

То есть пусть мы ищем кратчайший путь от вершины s до вершины t. $d(u, v)$ - расстояние между вершинами u и v Тогда $f(v)$ - оценка на $d(v, s)$, которая обладает следующими свойствами:

1. **Неотрицательность** $f(v) \geq 0$
2. **Оценка снизу** $f(v) \leq d(v, t)$
3. **Неравенство треугольника** $\forall a, b$, связанных ребром $f(b) \leq f(a) + w$, где w - вес ребра.

Функция f подбирается, исходя из задачи. Если граф расположен в пространстве или на плоскости (например, города и дороги), то там существует естественная функция f - эвклидово расстояние. Легко проверить, что все три свойства в этом случае выполняются. Тогда алгоритм Дейкстры будет пытаться продвигаться именно по направлению к нужной точке.

37.6. A*, алгоритм

A* делает в общем-то то же, что и алгоритм Дейкстры, но рассматривает вершины не в порядке d, а в порядке $d + f$. В остальном всё будет происходить точно так же. Все обозначения означают то же, что и раньше.

Лемма: Пусть $u \notin A$, $d[u] + f[u]$ минимально среди всех ещё не обработанных вершин. Тогда расстояние $d[u]$ уже посчитано окончательно.

Док-во: Рассмотрим произвольный путь $s \rightsquigarrow u$. В общем случае путь может заходить в A и выходить оттуда несколько раз. Но так как до любой обработанной вершины w существует кратчайший путь $s \rightsquigarrow w$ только по обработанным вершинам, мы можем считать, что путь $s \rightsquigarrow u$ никогда не возвращается в A (наша цель сделать путь как можно короче).

Пусть v_1 - первая необработанная вершина на пути, v_2 - вторая, v_k - предпоследняя. w_i - вес ребра между v_i и v_{i+1} (считаем, что $v_{k+1} = u$).

1. По неравенству треугольника для f:

$$f[v_k] \leq f[u] + w_k$$

$$f[v_{k-1}] \leq f[v_k] + w_{k-1} \leq f[u] + w_{k-1} + w_k$$

$$\dots$$

$$f[v_1] \leq f[u] + w_0 + w_1 + \dots w_k$$

2. Мы выбрали u так, что $d[u] + f[u]$ минимально среди всех необработанных вершин. Так как весь путь $s \rightsquigarrow v_1$ проходит по обработанным вершинам длина этого пути не меньше $d[v_1]$.

   Тогда длина всего пути равна:

$$l = d[v_1] + w_1 + \dots w_k$$

$$l + f[u] = d[v_1] + (w_1 + \dots w_k + f[u]) \geq d[v_1] + f[v1] \geq d[u] + f[u]$$

$$l \geq d[u]$$

Итак, мы рассмотрели произвольный путь и поняли, что его длина не меньше $d[u]$. Это значит, что $d[u]$ посчитано окончательно.

37.7. A*, эпилог

Чтобы получить $A*$ из Дейкстры, надо добавить очень мало. А именно, *relax* теперь выглядит так:

```
void relax(int u, int val){
    Insert( {u, f[u] + val} );
}
```

А главный цикл - так:

```
u = s; //изначально у нас одна вершина с правильно посчитанным расстоянием - s
while (u != NIL && u != t){
    for(edge in e[u])          //обрабатываем вершину u
        relax(edge.to, d[u] + edge.weight);

    u = find_min(); //находим новую вершину с минимальным d + f
}
```

Тут предполагается использование некоторой кучи, и мы по-прежнему вставляем новые элементы вместо того, чтобы обновлять старые.

Время работы $A*$, очевидно, асимптотически такое же, как у Дейкстры. Однако на реальных графах $A*$ работает намного быстрее, так как не просматривает эвристически неоптимальные маршруты. Всё это, конечно, в предположении, что функция f считается за $\mathcal{O}(1)$

Лекция по алгоритмам #38
Алгоритмы Флойда и Беллмана-Форда

23 ноября

38.1. Алгоритм Флойда

Алгоритм Флойда состоит в том, чтобы найти матрицу $d[a, b]$ – минимальный путь от вершины a до вершины b. Алгоритм позволяет присутствовать в графе ребрам с отрицательными весами, но не циклам отрицательного веса. Чаще всего код алгоритма выглядит следующим образом:

```
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            relax(d[i, j], d[i, k] + d[k, j]);
```

Предварительно, нужно заполнить d бесконечностями, ячейки главной диагонали 0 и инициализировать матрицу данными нам ребрами.

Сложность алгоритма $\mathcal{O}(V^3)$, где V - количество вершин в графе.

Покажем его корректность. Для этого рассмотри вот такую матрицу: $d[k, a, b]$ - минимальный путь от a до b, в котором в качестве промежуточных вершин могут быть вершины от 0 до $k - 1$. Изначально $d[0, a, b] = w_{a,b}$, то есть, вес ребра между a и b или бесконечности, если такового нет. Переход делаем следующим образом: $d[k + 1, a, b] = min(d[k, a, b], d[k, a, k] + d[k, k, b])$. Действительно, когда мы позволяем вершине k быть промежуточной, мы можем либо не взять ее(первый переход), либо взять(второй переход). Убрав параметр k мы получим первоначальный вид алгоритма. Мы можем так поступить, ведь единственная проблема, с которой мы можем столкнуться - прохождение по вершине k несколько раз, но нам это не выгодно.

38.2. Алгоритм Флойда и восстановление пути

Для этого будем считать вспомогательный массив - $p[a, b]$, а именно: какую вершину мы брали в качестве промежуточной, когда обновляли ответ для $d[a, b]$?

```
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (d[i, j] > d[i, k] + d[k, j]) {
                d[i, j] = d[i, k] + d[k, j];
                p[i, j] = k;
            }
```

Процедура восстановления ответа выглядит следующим образом:

```
void restore_path(int a, int b)  {
    if (p[a, b] == -1) { //edge
        cout << a << " ";
        return;
    }
    restore_path(a, p[a, b]);
    restore_path(p[a, b], b);
}
```

Для корректности рекурсии покажем, что в итоге получается простой путь и рекурсия не зацикливается.
1) Пусть путь не простой. Докажем по индукции корректность. Пусть неверно для какого-либо пути (k, a, b). Тогда обозначит за x вершину, которая содержится и в пути от a до k, и от k до b(x не равна k). Заметим, что цикл $x \to k \to x$ неотрицательного веса, а следовательно путь $a \to x \to b$ не хуже, чем имеющийся. Также промежуточная вершина x меньше k, и поэтому мы бы нашли этот путь ранее.
2) Рекурсия не зацикливается, так как длина подзадач строго уменьшается.

Таким же образом можно восстанавливать и отрицательный цикл. Отрицательный цикл появляется, когда в какой-либо момент времени $d[a, a]$ становится меньше нуля. Это примем без доказательства.

38.3. Алгоритм Флойда и транзитивное замыкание

Тривиальный алгоритм за $\mathcal{O}(V^3)$:

```
for (auto e : edges)
    is[e.first][e.second] = true;
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            is[i, j] |= is[i, k] && is[k, j];
```

Можно написать это же с bitset-ом и получить алгоритм за $\mathcal{O}(\frac{V^3}{\omega})$, где ω - размер машинного слова.

```
bitset<n> is[n];
for (auto e : edges)
    is[e.first].set(e.second, 1);
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        if (is[i, k])
            is[i] |= is[k];
```

38.4. Алгоритм Беллмана-Форда

$d_1[k, b]$ - это min по путям от вершины s до b, состоящим ровно из k ребер. Изначально $d_1[0, s] = 0, d_1[0, b \neq s] = \infty$. Переход: $d_1[k, b] = \min\limits_{e:a->b}(d_1[k-1, a] + w_e)$. Но нам хотелось бы иметь min по путям не только из ровно k ребер, а и по тем, количество ребер которых меньше. Такова мотивация для введения $d_2[k, b] = min(d_1[k, b], d_2[k-1, b])$. Убрав параметр k, получаем знакомый всем вид этого алгоритма:

```
for (int k = 0; k < n - 1; k++) {
    for (e : a -> b)
        if (d[b] > d[a] + w_e) {
            d[b] = d[a] + w_e;
            p[b] = e;
        }
}
```

Мы также поддерживаем массив $p[x]$ - предыдущая вершина на пути от x до s. Сложность алгоритма $\mathcal{O}(VE)$.

38.5. Алгоритм Беллмана-Форда и break

Заметим, что мы сделаем следующую итерацию впустую, если у нас на текущей ничего не улучшилось. Эта наталкивает на мысль о переменной-индикаторе.

```
bool RUN = true;
for (int k = 0; k < n - 1 && RUN; k++) {
    RUN = false;
    for (e : a -> b)
        if (d[b] > d[a] + w_e) {
            d[b] = d[a] + w_e;
            p[b] = e;
            RUN = true;
        }
}
```

Мы получили время работы $\mathcal{O}(kE)$, где k - длина максимального по количеству ребер кратчайшего пути.

38.6. Алгоритм Беллмана-Форда и random_shuffle()

Теория вероятности утверждает, что если сделать $random\_shuffle()$ ребер перед выполнением алгоритма, то мы получим время работы $\mathcal{O}(\frac{k}{2}E)$. Интуитивное объяснение этому феномену в том, что нам с очень большой вероятностью повезет и мы за одну итерацию будем выкидывать сразу два ребра кратчайшего пути. Код при этом остается таким же, как и в предыдущем пункте.

38.7. Алгоритм Беллмана-Форда и очередь

Понаблюдав за поведением предыдущих реализаций алгоритма, можно осознать, что зачастую он пытается улучшать точно такую же пару вершин, как и на предыдущей итерации, хотя значения на концах этого ребра не менялись. Напрашивается введение множеств A_i - множество всех вершин, которые были улучшены на итерации i. Как и в алгоритме BFS эти множества удобно хранить в очереди.

```
while (q.size()) {
    int a = q.front();
    q.pop();
    in[a] = false;
    for (e : a -> b)
        if (d[b] > d[a] + w_e) {
            d[b] = d[a] + w_e;
            p[b] = e;
            if (!in[b]) {
                q.push(b);
                in[b] = true;
            }
        }
}
```

Массив $in[v]$ хранит информация о присутствии вершины v в очереди и позволяет не добавлять вершину на рассмотрение, если она уже лежит в очереди. Примечательно, что на случайных графах эта реализация алгоритма Беллмана-Форда работает за линейное время.

Лекция по алгоритмам #39
Цикл отрицательного веса и цикл минимального среднего веса.

23 ноября

39.1. Цикл отрицательного веса.

Добавим в граф фиктивную вершину, и направим из нее (но не в нее!) ребра нулевого веса во все остальные. Запустим из нее алгоритм Форда-Беллмана.

Как известно, за первую $n-1$ итерацию алгоритм Форда-Беллмана находит все кратчайшие пути, состоящие из не более чем $n-1$ ребер. Значит, если на n-ной итерации произошла релаксация, то в графе есть кратчайший путь длины n, а в пути длины n обязательно есть цикл. Так как этот путь кратчайший (причем цикл веса 0 не релаксируется алгоритмом Форда-Беллмана), то этот цикл имеет отрицательный вес (иначе без него путь бы стал короче).

Восстановление ответа: достаточно для каждой вершины хранить последнюю вершину на пути в нее. Когда найдена вершина, расстояние до которой улучшилось на n-ной итерации, достаточно идти из нее назад, пока не будет найден цикл.

Осталось доказать, что если в графе есть цикл отрицательного веса $v_0, v_1, \ldots, v_k$, где $v_0 = v_k$, то Форд-Беллман сделает хотя бы одну релаксацию на n-ном шаге. Предположим, что этого не произошло. Тогда после $n-1$ релаксации для всех i от 1 до k $d[v_i] \le d[v_{i-1}]+w(v_{i-1}, v_i)$. Если просуммировать это неравенство по всем i от 1 до k, получится, что $\sum\limits_{i=0}^{k-1} d[v_i] \le \sum\limits_{i=1}^{k} d[v_i] + \sum\limits_{i=1}^{k} w(v_{i-1}, v_i)$, а так как $v_0 = v_k$, это равносильно $0 \le \sum\limits_{i=1}^{k} w(v_{i-1}, v_i)$, то есть тому, что наш отрицательный цикл неотрицателен.

Сложность алгоритма равна сложности алгоритма Форде-Беллмана и равна $\mathcal{O}(VE)$.

Примечание: такой алгоритм находит только отрицательные циклы, достижимые из стартовой вершины алгоритма. Чтобы алгоритм работал корректно, нужно добавить фиктивную вершину, из которой провести ребра во все остальные (но не обратно!), и запускаться из нее.

39.2. Цикл минимального среднего веса.

39.2.1. Бинпоиск.

Воспользовавшись умением находить отрицательный цикл за $\mathcal{O}(VE)$, можно сделать следующее: перебрать бинпоиском минимальный средний вес (очевидно, он не меньше минимального веса ребра графа и не больше максимального). Для каждого значения минимального среднего веса вычтем за $\mathcal{O}(E)$ из весов всех ребер выбранное значение, а затем проверим, есть ли в графе отрицательный цикл. Сложность – $\mathcal{O}(VE \log M)$, где M – разность максимального и минимального весов ребер в графе.

39.2.2. Алгоритм Карпа.

Итак, задача: избавиться от логарифма. Для этого разобъем задачу на 2 части: нахождение минимального среднего веса цикла и нахождение самого этого цикла при известном минимальном среднем весе.

Вторая часть уже разобрана в предыдущем методе (вычитание найденного значения + ε из весов всех ребер и поиск цикла отрицательного веса).

Теперь, собственно, нахождение минимального среднего веса цикла в графе. Определю $d[k, v]$ как длину кратчайшего пути из вершины s в v по k ребрам (считается Фордом-Беллманом). Теперь определю μ как $\min\limits_{v}(\max\limits_{k=0}^{n-1}(\frac{d[n,v]-d[k,v]}{n-k}))$. Докажу, что это и есть искомый ответ. Для этого достаточно сказать, что:

1. При увеличении (уменьшении) весов всех ребер на одну и ту же константу, ни μ, ни ответ (обозначу его как X), не изменятся по определениям.

2. Если доказать, что при $X = 0 \cdot \mu = 0$, то при любом $X\mu = X$ (следует из предыдущего пункта).

3. Заметим, что при $X = 0$ граф без отрицательных циклов. Для него μ по определению неотрицательна, так как знаменатель ($\frac{d[n,v]-d[k,v]}{n-k}$ всегда положителен, а числитель $\max\limits_{k=0}^{n-1}(\frac{d[n,v]-d[k,v]}{n-k}) = 0$ равен разности кратчайшего пути из n вершин и кратчайшего пути вообще, что является неотрицательной величиной.

4. При $X = 0$ существует вершина v, для которой $\max\limits_{k=0}^{n-1}(\frac{d[n,v]-d[k,v]}{n-k}) = 0$. Так как $X = 0$, в графе существует 0-цикл. Возьмем некоторую вершину u на этом цикле. Пусть кратчайший путь до нее состоит из k ребер. тогда пойдем из этой вершины далее по нулевому циклу $n-k$ раз, пока не прийдем в некоторую вершину v. Обозначу сумму весов на пути по нулевому циклу из u в v как , а из v в u как . По определению $+ = 0$. С одной стороны $d[v]$ (кратчайший путь до вершины v) не больше $d[u]+A$. С другой стороны, если предположить, что $d[v] < d[n, v]$, при том что $d[n, v] \leq d[u] + A$, получаем: $d[u] \leq d[v] + B < d[u] + A + B = d[u]$, что ложно. Значит, $d[v] = d[n, v]$, что значит, что $\max\limits_{k=0}^{n-1}(\frac{d[n,v]-d[k,v]}{n-k}) = 0$, а так как μ неотрицательна, $\mu = 0$.

Из написанного выше видно, что $X = \mu$. Восстановление такое же, как и в алгоритме с бинпоиском. Сложность – $\mathcal{O}(VE)$.
Примечание: также как и в алгоритме с бинпоиском, надо добавить фиктивную вершину. Также, в этой задаче считалось, что если нет пути из s в v по k ребрам, $d[k, v] = \infty$. Также, $\infty - \infty = \infty$. Таким образом, если в графе нет циклов, то $d[n, v] = \infty$, то есть $\mu = \infty$.

Лекция по алгоритмам #40
Алгоритм Карпа и алгоритм Джонсона

28 сентября

40.1. Алгоритм Карпа

Задача: найти цикл минимального среднего веса.

Теорема 40.1.1. Величина цикла с минимальным средним весом равна $\mu = \min\limits_{v} \max\limits_{0 \le k \le n-1} \frac{d_{n,v} - d_{k,v}}{n-k}$

Lm 40.1.2. Если вес всех рёбер уменьшить на x, то величина цикла с минимальным средним весом уменьшится на x.

Таким образом, достаточно доказать, что в если величина ответа равна нулю, то $\mu = 0$. Пусть $z_v = \max\limits_{0 \le k \le n-1} \frac{d_{n,v} - d_{k,v}}{n-k}$. Тогда $\mu = 0 \Leftrightarrow \begin{cases} \forall v\ z_v \ge 0 \\ \exists v \colon z_v = 0 \end{cases}$

Докажем сначала первое. $z_v = \max\limits_{k} \frac{d_n - d_k}{n-k}$. $n - k > 0$, поэтому чтобы определить знак достаточно рассмотреть числитель. d_n - константа, поэтому $\max\limits_{k}\{d_n - d_k\} = d_n - \min\limits_{0 \le k \le n-1} d_k \ge 0$. Последнее неравенство верно, т.к. $\min\limits_{k=0}^{n-1} d_k$ - это кратчайший путь до вершины, а d_n - какой-то путь.

Теперь докажем вторую часть. Пусть u - какая-то вершина цикла длины 0. Пусть $p(u)$ - кратчайший путь в эту вершину, а количество рёбер в нём равно k. Пройдём от этой вершины $n - k$ рёбер вдоль цикла. Пусть мы попали в вершину v. Тогда $d_{n,v} = d_{u,k}$. Кроме того, кратчайший путь в вершину u равен кратчайшему пути в вершину v. Следовательно, $\exists k \colon d_{n,v} = d_{k,v} \Rightarrow z_v = 0$, ч.т.д.

Алгоритм нахождения цикла минимального веса:

1. Посчитаем с помощью Форда-Беллмана матрицу $d_{n,k}$.
2. Вычислим μ.
3. Вычтем из всех рёбер $\mu + \epsilon$.
4. Найдём отрицательный цикл стандартным алгоритмом.

Сложность данного алгоритм исходит из сложности ФБ и равна $\mathcal{O}(VE)$.

40.2. Алгоритм Джонсона

Задача: посчитать расстояние между всеми парами вершин быстрее, чем это умеет делать Флойд. В графе могут быть отрицательные рёбра

На графе с рёбрами неотрицательного веса эту задачу можно решить запустив из каждой вершины алгоритм Дейкстры за $\mathcal{O}(V^2 \log V)$. Алгоритм Джонсона состоит из трёх шагов:

1. Сделать преобразование над рёбрами, которое делает их веса неотрицательными.
2. Найти кратчайшие пути между всеми парами вершин на новом графе описаным выше методом.

3. Восстановить пути в исходном графе по подсчитанным.

Def 40.2.1. *Потенциал вершины – функция* $p\colon V \to \mathbb{R}$.

Def 40.2.2. *Применить потенциал – провести с весом каждого ребра* $e(a,b)$ *преобразование* $w_e = w_e + p(a) - p(b)$.

Lm 40.2.3. Кратчайший путь из a в b увеличится на $p(a) - p(b)$.

Доказательство. Вес пути - это сумма весов рёбер. Заметим, что все изменения примененные к рёбрам сократятся, кроме тех, что произошли на краях. ■

Следствие 40.2.4. Кратчайшие пути исходного графа переходят в кратчайшие пути в новом графе.

Чтобы, после вычисления кратчайших путей в новом графе, получить пути в исходном нужно к для каждой пары вершин (a,b) сделать $d[a,b] = d[a,b] + p(b) - p(a)$. Это прямо следует из леммы 40.2.3.

• **Как выбрать потенциалы, чтобы новые веса рёбер были неотрицательными?**

Добавляем фиктивную вершину s и из неё рёбра во все веса 0. Потенциал v – расстояние от фиктивной вершины s до v.

Лекция по алгоритмам #41
Алгоритм Гольдберга

27 ноября

41.1. Алгоритм Гольдберга

Задача: В графе без отрицательных циклов найти такой потенциал $p : V \to \mathbb{R}$, что $\forall e(a,b) \in E : w'_e = w_e + p(a) - p(b) \geq 0$.

Пусть в графе веса всех ребер ограничены снизу. Т.е. $\forall e \in E : w_e \in \mathbb{Z} \cap [-N; +\infty]$ для некоторого $N > 0$. Тогда алгоритм Гольдберга может решить данную задачу за время $O(E\sqrt{V}\log N)$.

Научимся решать задачу для случая $N = 1$. Все ребра графа разбиваются на 3 класса:

$$w_e = \begin{cases} -1 \\ 0 \\ x, \quad x > 0 \end{cases}$$

Решение за $O(EV)$.

Def 41.1.1. *Назовем вершину плохой, если в нее входит ребро веса -1.*

Def 41.1.2. *Ребро $e(a,b)$называется исходящим ребром множества вершин A,если $a \in A, b \notin A$.*

Def 41.1.3. *Ребро $e(a,b)$ называется входящим ребром множества вершин A, если $a \notin A, b \in A$.*

Def 41.1.4. *Ребро $e(a,b)$ называет внутренним ребром множества вершин A, если $a \in A, b \in A$.*

Def 41.1.5. *Ребро $e(a,b)$ называет внешним ребром множества вершин A, если $a \notin A, b \notin A$.*

Алгоритм будет итеративым. Каждая итерация состоит из 5 шагов:

1. Находим плохую вершину v. Пусть $e(x,v) : w_e = -1$.
2. Запускаем dfs из v по ребрам с весами -1 и 0. Все посещенные вершны образуют множество $A \subset V$. Если $x \in A$, то существует неположительный путь из v в x, а значит есть и отрицательный цикл (т.е. у нас уже есть отрицательное ребро из x в v).
3. Для всех $a \in A$ делаем $p(a) - = 1$. При этом w'_e внутренних ребер A не изменился (+1 с одного конца и -1 с другого), входящих увеличился на 1, а исходящих уменьшился на 1. Но т.к. все исходящие ребра имеют положительный вес (иначе мы прошли бы по этому ребру в dfs-е), то все исходящие ребра остались неотрицательными.

4. Вершина v перестала быть плохой. Новых плохих вершин не образовалось, т.к. не появилось новых отрицательных ребер.

5. Если в графе не осталось плохих вершин, то заканчиваем – у нас не осталось отрицательных ребер.

В худшем случаи мы сделаем V итераций, на каждой dfs отработает за $O(V + E)$. Получаем время работы $O(VE)$.

Решение за $O(E\sqrt{V})$).

Lm 41.1.6. Если из некоторого множества A пошагово удалять $\geq \sqrt{|A|}$ элементов ($|A|$ – размер множества на текущем шаге), то множество станет пустым через $O(\sqrt{|A_0|})$ шагов ($|A_0|$ – изначальный размер).

Доказательство. Представим размер множества как клетчатый прямогульник со сторонами примерно $\sqrt{|A_0|}$. Тогда каждый шаг удаления можно соотнести с отрезанием от прямоугольника строки/столбца (чередуются). Кол-во строк и столбцов в этом прямоугольнике $O(\sqrt{|A_0|})$. После удаления всех строк и слобцов наше множество станет пустым. ■

Алгоритм будет итеративым. Каждая итерация состоит из n шагов:

1. Сконденсируем граф по -1-м и 0-м ребрам. Если внутри какой-то компоненты оказалось ребро веса -1, то мы нашли отрицательный цикл внутри этой компоненты.

2. Добавим фиктивную вершину S и проведем из нее ребра веса 0 во все другие вершины.

3. Найдем расстояние от вершины S до всех других по -1, 0 ребрам. Т.к. в графе нет циклов, то это можно сделать с помощью динамики за $O(E)$.

4. Разобьем все вершины на слои в зависимости от посчитанного расстояния. Если вершина v лежит в $i-$м слое, то существует путь из S в v, в котором будет i ребра веса -1. Обозначим за k кол-во оставшихся плохих вершин. Тогда, по принципу Дирихле, есть или слой, в котором $\geq \sqrt{k}$ плохих вершин, или кол-во слоев $\geq \sqrt{k}$ (не считая 0-го, т.к. в нем нет плохих вершин).

5. Рассмотрим 2 случая:

   (a) Есть слой, в котором $\geq \sqrt{k}$ плохих вершин. Тогда, запустим dfs из всех этих вершин аналогично 1-му решению, образуем множество A. Заметим, что вершины более ранних слоев не могут лежать в этом множестве. Применим $p(v)- = 1$ для всех вершин множества. Все исходящие ребра мы не испортим. Т.к. все ребра веса -1, которые входили в плохие вершины, шли из более раннего слоя, то после изменения потенциала их вес станет равен 0. Т.е. все плохие вершины из этого слоя хорошими, а новых не образовалось.

   (b) Кол-во слоев $\geq \sqrt{k}$. Тогда, по сказанному в 4 пункте, есть простой путь, в котором $m \geq \sqrt{k}$ ребер веса -1, значит и $m \geq \sqrt{k}$ плохих вершин.
   Рассмотрим этот путь. Пронумеруем плохие вершины этого пути с конца (т.е. путь будет идти в порядке $v_m, v_{m-1}, \ldots, v_2, v_1$).
   Запустим dfs из v_1, получим множество A_1. Для всех вершин пометим, что они были добавлены в 1 множество. Теперь мы должны были бы запустить dfs из v_2 и получить множество A_2. Но это долго. Заметим, что $A_1 \subset A_2$ (т.к. v_1 достижим из v_2). Множество A_2 состоит из 3-х частей: вершин, которые достижимы из v_2, то не достижими из v_1; вершины, которые лежат в A_1; вершины, которые достижимы из v_1, но в момент запуска dfs из v_1 в них вели положительные ребра (при применении $p(v)- = 1$ к A_1 веса исходящих ребер уменьшились и могли стать 0). Чтобы обработать 1-ю группу просто запустим dfs из v_2, ходить будем только по непосещенным вершинам.

Чтобы обрабатывать 3-ю группу, заведем магическую структуру $Vector$, которая по номеру плохой вершины возвращает список ребер, которые могли "обнулиться"при обработке предидущей плохой вершины.
Получаем следующий порядок действий: при обработке v_i запускаем dfs по еще не посещенным вершинам, а также из концов ребер, которые могли обнулиться на текущем шаге (если вершина на другом конце ребра уже лежит в каком-то множестве или ребро не обнулилось, то ничего не делаем); помечаем посещенные вершины, что они принадлежат i-му множеству; добавляем в структуру новые исходящие ребра-кандидаты.
Заметим, что после i-го запуска мы сделали вершину v_i хорошей. Т.е. мы сделалем все m вершин хорошими.
Само значение потенциала для вершин можно изменять не на каждом шаге, а в конце, по номеру множества, в которое его добавили (это будет 1-е попадание, на последующих шагах мы всегда делали $-=1$ для этой вершины).

Опишем структуру $Vector$. Это будет вектор векторов размера E. Ребро веса a, которое добавили на шаге b, кладем в список $a+b$. При обработке i-й плохой вершины мы смотрим в i список. Для ребер этого списка верно, что с момента их добавления прошло a (вес ребра) шагов, а значит мы могли a раз уменьшить его вес на 1. Ребра веса больше E можно игнорировать (т.к. оно не успеет стать нулевым).

На каждой интерации мы делали хорошими $\geq \sqrt{k}$ плохих вершин, а значит мы сделаем $O(\sqrt{k})$ итераций, что в худшем случаи $O(\sqrt{V})$. На каждой итерации мы совершали $O(V+E)$ операций, а значит время работы $O(E\sqrt{V})$.

Обобщение решения. Покажем, как использовать полученное решение для любого N.
Пусть $N=2d+1$. Будем использовать решение для $N=1$, чтобы перейти к $N=d$. Для этого запустим его с учетом следующих переобозначений:

$w_e \in [-(2d+1);-(d+1)]$	-1
$w_e \in [-d;0]$	0
$w_e > 0$	$\lceil \frac{w_e}{d+1} \rceil$ (вес при добавлении в структуру)
$p(v)-=(d+1)$	$p(v)-=1$

Исходный алгоритм для всех ребер веса -1 ставил вес ≥ 0. Теперь алгоритм для всех ребер из $[-(2d+1);-(d+1)]$ сделает вес $\geq d$. Таким образом, за $\log N$ применений этого алгорима все ребра будут иметь вес ≥ 0.
Получаем время работы $O(E\sqrt{t}\log N)$.

Лекция по алгоритмам #42
Алгоритм Йена и введение в Radix Heap

30 ноября

42.1. Алгоритм Йена

42.1.1. Постановка задачи

Есть взвешенный граф с неотрицательными весами рёбер, найти в нём k-тый по длине простой путь между вершинами s и t.

42.1.2. 2-ой по длине путь

Рассмотрим кратчайший путь между s и t. Второй по длине путь между ними своими начальными рёбрами совпадает с первым, а затем в каком-то месте появляется первое отличие. Давайте переберём ребро, в котором путь будет отличаться, т.е. по которому мы не пойдём. Назовём это ребро запрещённым. Найдём из вершины, где мы остановились, кратчайший путь до t, запретив ходить в вершины, которые были в начале пути, а также по запрещённому ребру. Т.к. в кратчайшем пути максимум $V - 1$ ребро, то мы получили время работы $V \cdot Dijkstra$

42.1.3. k-тый путь

Разобьём пути между s и t на классы. Один класс определяется парой: префикс пути, запрещённые рёбра. Переберём префикс, как в предыдущем пункте, получим набор классов. Выберем из каждого класса по минимальному представителю.
Будем хранить Heap классов. Достанем минимум (обозначим его X) и, попытавшись разными способами увеличить префикс на одно ребро, раздробим таким образом класс X на другие классы, объединение которых даёт класс X. Сделав так k раз, получим k минимальных путей.

$Time = k \cdot (V \cdot Dijkstra + \log)$
$Memory = k \cdot V \cdot V$. Можно уменьшить память до $k \cdot V$, так как в каждый момент времени можно хранить только лучшие k классов.

42.2. Введение в Radix Heap

42.2.1.

Radix Heap = k-bfs + Dijkstra + Magic,
и с целыми весами рёбер от 1 до k мы получаем кратчайшие пути от одной вершины за $O(E + V \log k)$

42.2.2.

Здесь нам тоже нужна $k + 1$ очередь...

42.2.3.

...и вообще стоит заметить, что 1-k bfs и Dijkstra очень похожи...

42.2.4.

...и то, что кратчайшие пути за $O(E \log k)$ мы уже получаем совершенно бесплатно следующим образом: давайте в 1-k bfs выбирать следующую очередь за log, храня Heap непустых очередей...

42.2.5.

...а дальше получим $E+V\sqrt{k}$ с помощью sqrt-декомпозиции: разобьём очереди на $\sqrt{k}$ блоков по $\sqrt{k}$ очередей, пробежим циклом по блокам, затем пробежим по первому непустому.

Лекция по алгоритмам #43
Radix Heap

30 ноября

43.1 Radix Heap

Мы хотим написать кучу для реализации алгоритма Дейкстры.

Radix Heap работает за $\mathcal{O}(E + V \log k)$ при условии, что веса всех ребер от 1 до k.

Будем хранить все расстояния в $m + 1$ бакетах, где $m = \lceil \log k \rceil$.

Зададим диапазоны $d_0 \le d_1 \le \cdots \le d_m \le d_{m+1} \le \infty$. Тогда в первом бакете будут храниться числа из промежутка $[d_0 \dots d_1)$, во втором – из промежутка $[d_1 \dots d_2)$, в i-ом бакете хранятся числа из промежутка $[d_{i-1} \dots d_i)$. В $(m+1)$-ом бакете изначально будут лежать все расстояния. Внутри каждого бакета будем хранить расстояния в любом порядке.

Зададим ограничения на диапазоны d_i. Пусть $x_0 = 1$ и $\forall 1 \le i \le m$ $x_i = 2^{i-1}$ Скажем, что $\forall i$ $d_{i+1} - d_i \le x_i$.

Создадим очереди q_0, q_1, …, q_m, соответствующие бакетам.

Изначально $d_0 = 0$, $d_1 = 1$, $d_{i+1} - d_i = x_i$, $d_{m+1} = \infty$, $q_0 = \{start\}$, q_m – остальные вершины

Для реализации алгоритма Дейкстры мы должны уметь делать Extract Min и Decrease Key.

- **EMin**

  Пусть q_0, q_1, …, q_{i-1} – пустые.

  q_i – первая непустая очередь.

  Если $i \le 1$, то вытащим любой элемент из этой очереди – он и будет минимальным.

  Если нет, то нужно все элементы очереди раскидать по предыдущим очередям. Заметим, что $x_0 + \cdots + x_{i-1} = x_i \ge d_{i+1} - d_i$. Это означает, что мы можем все элементы из очереди раскидать по предыдущим. Проставим следующие диапазоны: $d_0 = 1$, $d_1 = 2$, $d_{j+1} - d_j = x_j$ или $d_{j+1} = d_{i+1}$, если $d_j + x_j > d_{i+1}$, для всех предыдущих очередей и $d_{i+1} = d_i$. Тогда все элементы распределились корректным образом.

  Теперь мы хотим, чтобы q_0 оказалась непустой. Пусть a – минимальный элемент в очереди. Тогда будем раскидывать элементы также, только теперь $d_0 = a$.

  q_0 стала непустой, вытаскиваем из нее любой элемент.

  Первую непустую очередь мы ищем за $\mathcal{O}(\log k)$. Каждый элемент в итоге переместится в предыдущую очередь не более $\log k$ раз. Значит, всего EMin работает за $\mathcal{O}(V \log k)$.

- **DecKey**

  Уменьшаем элемент. Пока он не соответствует диапазону текущей очереди, перемещаем его в предыдущую. Каждый элемент мы переместим не более $m = \log k$ раз, поэтому суммарно Decrease Key отработает за $\mathcal{O}(E + V \log k)$.

43.2 Two Level Radix Heap

$\mathcal{O}(E + V\frac{\log k}{\log\log k})$

Возьмем некоторое t. Разобьем весь диапазон на $\log_t k$ очередей, каждую очередь разобьем на t кусков. Зададим диапазоны $d_{i+1} - d_i \le t^i$.

- **EMin**

  За $\mathcal{O}(\log_t k)$ найдем первую непустую очередь, за $\mathcal{O}(t)$ найдем первый ненулевой кусок в ней.

  Заметим, что $t^i = t^{i-1} \cdot t$, что означает, что диапазон одного куска i-ой очереди равен диапазону $(i-1)$-ой очереди. Тогда для того, что бы распределить кусок по предыдущим очередям, мы можем переместить все элементы из него в $(i-1)$-ую очередь и поменять диапазоны.

  Тогда EMin работает за $\mathcal{O}(\log_t k + t)$. Суммарно за $\mathcal{O}(V \cdot (\log_t k + t))$.

- **DecKey**

  Уменьшим значение элемента. Пока число не попадает в диапазон текущей очереди, будем перемещать его в предыдущую. Каждый элемент переместится не более, чем на $\log_t k$. Значит, суммарно Deckey работает за $\mathcal{O}(E + V \log_t k)$.

Возьмем $t = \frac{\log k}{\log\log k}$. Тогда алгоритм работает за $\mathcal{O}(E + V\frac{\log k}{\log\log k})$.

Лекция по алгоритмам #44
Система непересекающихся множеств

4 декабря

44.1. Система непересекающихся множеств

44.1.1. Определение

CHM - Система непересекающихся множеств
DSU - Disjoint Set Union

CHM - это структура данных, поддерживающая две операции:
int get(int v) - получить номер множества, которому принадлежит элемент v
void join(int u, int v) - объединить множества, в которых лежат элементы u и v
Также часто бывает полезной is(u,v): {get(u)==get(v)}, проверяющая элементы u,v на принадлежность одному множеству.

44.1.2. CHM на списках

color[v] - номер множества, которому принадлежит v
list[col] - вершины множества под номером col
get: return color[v] $\mathcal{O}(1)$
join: if color[u] != color[v]: merge(list[color[u]], list[color[v]])
merge: 1)Перекрасим вершины меньшего множества.
2)Объединим списки.
Здесь мы пользуемся уже знакомым приемом присоединения меньшего множества к большему, благодаря которому каждый элемент будет перекрашен не более $\log n$ раз. Таким образом асимптотика последовательного объединения n одноэлементных множеств в одно $\mathcal{O}(n \log n)$.

44.1.3. CHM на деревьях

Давайте хранить множества в виде деревьев. Корень каждого дерева - какой-то элемент множества, выполняющий роль его идентификатора.
parent[v] - предок элемента v
parent[root] = root
join(u,v): p[get(u)]=get(v)
get(v): return v == p[v] ? v : get(p[v])
Время: join=$1 + 2 \cdot get$, get=$\mathcal{O}(n)$

44.2. Эвристики

44.2.1. Сжатие путей

Каждый раз когда вызывается get будем переподвешивать к корню все вершины на пути между ним и v.
get(v): return v == p[v] ? v : (p[v] = get(p[v]))

Теорема 44.2.1. Сложность m join'ов из n одноэлементных множеств с эвристикой сжатия путей равна $(m+n)\log n$

Доказательство. Назовем легкими ребра ab(ребра направлены вверх по дереву), для которых $size[a] < \frac{1}{2}size[b]$, соответственно тяжелыми будут ребра, для которых $size[a] \geq \frac{1}{2}size[b]$.

За один get мы пройдём не более $\log n$ легких ребер. Всего $m\log n$ легких ребер во всех запросах.

Теперь оценим количество тяжелых ребер. Так как эвристика сжатия путей переподвешивает вершины к корню, то мы будем удалять все пройденные ребра. Так как у каждой вершины можно удалить тяжелое ребро не более $\log n$ раз, то всего мы обработаем не более $n\log n$ тяжелых ребер.

$m\log n + n\log n = (m+n)\log n$ ■

44.2.2. Ранговая

Храним для каждого дерева его размер. При объединении меньшее дерево подвешивается к большему. Соответственно $depth \leq \log n$, $size[depth] \geq 2^{depth}$.

Вместо размеров можно хранить "ранги"деревьев. Изначально ранги всех одноэлементных деревьев равны 0. При объединении дерево меньшего ранга подвешивается к дереву большего ранга, а при равенстве порядок объединения произвольный, и ранг получившегося дерева увеличивается на 1.

$rank[v] \leq \log n$

$size[rank] \geq 2^{rank}$

Ранги требуют log памяти, которая нужна на хранение размеров.

Стоит упомянуть реализацию $join$, подвешивающую деревья в случайном порядке. Такая реализация дает лишь константное ускорение работы примерно в 2 раза.

Лекция по алгоритмам #45
Хорошие оценки на время работы СНМ

4 Декабря

45.1. Введение

В СНМ содержится n элементов. Подразумевается, что СНМ использует обе эвристики: сжатие путей и ранговую. За все время работы к СНМ было сделано m запросов типа *get* и q запросов типа *join*.

Def 45.1.1. *Ранг. Каждому элементу e сопоставляется число $r_e \in \mathbb{N} \cup \{0\}$, которое называется рангом элемента e. Изначально $\forall e : r_e = 0$. Если объединяются два множества с представителями a и b, причем $r_a = r_b = r$, то ранг представителя объединения, будь то a или b, становится равным $r + 1$, то есть увеличивается на 1.*

Def 45.1.2. *Предок. Если элемент e не является представителем множества A, в котором он лежит, то, в соответствие со структурой СНМ, элемент e ссылается на другой элемент p_e множества A, который называется предком элемента e.*

Lm 45.1.3. Если элемент e не является представителем своего множества, то $r_{p_e} > r_e$.

Доказательство. Докажем это утверждение, как инвариант:

- Такой инвариант выполнен в самом начале, когда каждый элемент образует одноэлементное множество.
- Когда выполняется операция *join*, инвариант сохраняется, так как либо элемент с меньшим рангом стал ссылаться на элемент с большим рангом, либо ранги представителей объединяемых множеств были равны, но тогда ранг нового представителя станет больше на 1 и будет больше, чем ранг представителя второго множества.
- Когда выполняется операция *get*, инвариант сохраняется, если он был выполнен до этого, так как в таком случае ранг представителя множества больше, чем ранг любого другого элемента этого множества, а операции *get* просто перенаправляет ссылки некоторых элементов на представителя.

Получаем, что данный инвариант сохраняется на протяжении всей работы с СНМ. ■

Def 45.1.4. *Размер множества. Если элемент e является представителем своего множества A, то $s_e := |A|$.*

Lm 45.1.5. Если элемент e является представителем своего множества, то $2^{r_e} \leq s_e$.

Доказательство. Докажем это утверждение, как инвариант:

- Такой инвариант выполнен в самом начале, когда каждый элемент образует одноэлементное множество: $2^{r_e} = 2^0 = 1 = s_e$.
- Когда выполняется операция *join* (обозначим за a и b представителей объединяемых множеств), возможны два варианта:

1. Ранги представителей различны, тогда: $2^{r_a} \le s_a, 2^{r_b} \le s_b \Rightarrow \max\{2^{r_a}, 2^{r_b}\} \le \max\{s_a, s_b\} \le s_a + s_b$.
2. Ранги представителей совпали, тогда: $r_a = r_b = r, 2^{r_a} \le s_a, 2^{r_b} \le s_b \Rightarrow 2^{r+1} = 2^r \cdot 2 = 2^r + 2^r = 2^{r_a} + 2^{r_b} \le s_a + s_b$.

- Когда выполняется операция *get*, ни с размером множества, ни с рангом представителя, ничего не происходит, значит инвариант сохраняется.

Получаем, что данный инвариант сохраняется на протяжении всей работы с СНМ. ■

45.2. Крутые ребра и логарифм со звездочкой

Def 45.2.1. *Крутое ребро. Если элемент e не является представителем своего множества и $r_{p_e} \ge 1.7^{r_e}$, то ссылка элемента e на p_e называется крутым ребром (ребро в графе ссылок).*

Lm 45.2.2. Крутых ребер на пути по ссылкам от $\forall$ элемента $e \in A$ до представителя множества A не больше чем $2\log_2^*(n)$.

Доказательство. Докажем от противного, что крутых ребер не больше чем $\log_{1.7}^*(n)$: пусть крутых ребер на пути $E > \log_{1.7}^*(n)$.
Обозначим за R представителя A. Докажем, что $r_R \ge 1.7^{\cdots^{1.7^{r_e}}}$, где количество 1.7 равно E. Ранг элемента e равен r_e, ранги следующих элементов строго больше (по лемме 1.3). Если ранг некоторого элемента на пути равен x, то $x \ge r_e$, а если следующее ребро на пути является первым крутым ребром на пути, то для ранга следующего на пути элемента a верно, что $r_a \ge 1.7^x \ge 1.7^{r_e}$. Проводя данное рассуждение индукционно, получаем требуемое неравенство: $r_R \ge 1.7^{\cdots^{1.7^{r_e}}}$.
Из полученного неравенства следует что, если $r_e > 0$, то $\log_{1.7}^*(r_R) \ge E > E - 1$, а если $r_e = 0$, то $\log_{1.7}^*(r_R) \ge E - 1$. То есть в любом случае $\log_{1.7}^*(r_R) \ge E - 1 > \log_{1.7}^*(n) - 1$. Так как это неравенство в целых числах, то из него следует, что $\log_{1.7}^*(r_R) \ge \log_{1.7}^*(n) \Rightarrow r_R \ge n \Rightarrow 2^{r_R} \ge 2^n > n \ge s_R$. Полученное неравенство противоречит лемме 1.5, значит предположение неверно и $E \le \log_{1.7}^*(n)$.
Имеет место неравенство $\forall x : 1.7^{1.7^x} > 2^x$. Если $n = 1.7^{\cdots^{1.7^{\alpha}}}$, где $0 < \alpha \le 1$ и количество 1.7 четное (обозначим его за $T_{1.7} = \log_{1.7}^*(n)$), то из неравенства следует, что $n > 2^{\cdots^{2^{\alpha}}}$, где количество 2 (обозначим за T_2) в два раза меньше $T_{1.7}$, то есть равно $\frac{T_{1.7}}{2}$. Если $T_{1.7}$ нечетно, то $T_2 = \frac{T_{1.7}-1}{2}$. В любом случае $n > 2^{\cdots^{2^{\alpha}}}$, где $T_2 \ge \frac{T_{1.7}-1}{2}$, но тогда $\log_2^*(n) > T_2 \ge \frac{T_{1.7}-1}{2} = \frac{\log_{1.7}^*(n)-1}{2} \Rightarrow 2\log_2^*(n) > \log_{1.7}^*(n) - 1 \Rightarrow 2\log_2^*(n) \ge \log_{1.7}^*(n)$.
Получаем, что $E \le \log_{1.7}^*(n) \le 2\log_2^*(n)$. ■

Lm 45.2.3. Если элемент e не является представителем своего множества, то после любой операции (*get* или *join*) значение величины r_{p_e} не уменьшилось.

Доказательство. Отдельно для каждой операции:

1. *get*. После операции либо предок элемента не изменился, тогда r_{p_e} тоже, так как после операции *get* ранги элементов не меняются, либо предком элемента стал представитель множества, но у представителя множества ранг не меньше, чем у всех элементов множества (по лемме 1.3), включая бывшего предка e.

2. *join*. После операции предок элемента не изменился, но мог измениться ранг предка. Ранг одного элемента по определению не может уменьшится.

■

Lm 45.2.4. Если элемент e не является представителем своего множества, то e уже никогда не станет представителем своего множества, а величина r_e уже никогда не изменится.

Доказательство. После операции *get* e разве что может ссылаться на представителя своего множества. После операции *join* один из двух представителей перестает быть представителем, а новых не появляется.
Ранг элемента меняется только когда происходит операция *join*, причем изменяется ранг элемента, который был представителем своего множества. Та как e уже не станет представителем своего множества, а ранг меняется только у представителя, то r_e больше не изменится. ■

Lm 45.2.5. Пусть некоторый элемент e после операции *join* перестал быть представителем своего множества. Тогда согласно лемме 2.4 e никогда не станет представителем своего множества, а величина r_e не изменится. После 1.7^{r_e} операций *get*, которые проходили по ссылке из e, когда p_e не являлся представителем своего множества, ссылка на p_e обязательно станет крутым ребром и останется им навсегда.

Доказательство. Если операция *get* проходит по ссылке из e не в представителя множества, то после операции ссылка элемента e изменится на представителя множества, у которого ранг больше хотя бы на 1 (лемма 1.3). Так как величина r_{p_e} не уменьшается (лемма 2.3), то после 1.7^{r_e} таких операций *get* r_{p_e} будет хотя бы 1.7^{r_e}. Значит через 1.7^{r_e} таких операций *get* будет выполнено неравенство $r_{p_e} \geq 1.7^{r_e}$, что означает, что ссылка на p_e является крутым ребром. Так как r_{p_e} не уменьшается, а 1.7^{r_e} не изменяется (лемма 2.4), то неравенство будет только усиливаться, а значит ссылка элемента e навсегда останется крутым ребром. ■

Теорема 45.2.6. Суммарное время работы всех операций *get* пропорционально $m \log^*(n) + n + m$.

Доказательство. Время работы i-ой операции *get* пропорционально $\log^*(n) + t_i + 1$:

1. $\log^*(n)$. По лемме 2.2 количество крутых ребер на пути операции *get* не больше чем $2\log^*(n)$.

2. t_i означает количество не крутых ребер на пути i-ой операции *get*, не учитывая последней ссылки в представителя множества. То есть это как раз такие ссылки, которые используются в лемме 2.5.

3. 1. Некоторое константное количество действий для каждой операции *get*. Сюда включен проход по последней ссылке в представителя множества, так как это действие не учитывается в пункте 2 и не всегда учитывается в пункте 1.

Значит суммарное время работы всех операций *get* пропорционально $\sum\limits_i (\log^*(n) + t_i + 1) = \sum\limits_i \log^*(n) + \sum\limits_i t_i + \sum\limits_i 1 = m\log^*(n) + \sum\limits_i t_i + m$.
$\sum\limits_i t_i = \sum\limits_e c_e$, где c_e суммарное количество переходов по ссылке элемента e всеми операциями *get*, когда ссылка элемента e была не крутым ребром, и p_v не являлся представителем своего

множества. Но по лемме 2.5 величина $c_e \le 1.7^{r_e}$ (для каждого элемента берется его ранг в момент, когда элемент перестал быть представителем своего множества, по лемме 2.4 его ранг уже не изменится), так как после такого количества переходов по ссылке из e ссылка гарантировано станет крутым ребром и останется им навсегда. Значит $\sum\limits_e c_e \le \sum\limits_e 1.7^{r_e}$. Если за $count(r)$ обозначить количество элементов с рангом r, то $\sum\limits_e 1.7^{r_e} = \sum\limits_r 1.7^r \cdot count(r)$.

Докажем, что $count(r) \le \frac{n}{2^r}$. От противного: пусть $count(r) > \frac{n}{2^r}$. Рассмотрим в хронологическом порядке появление каждого элемента e с рангом r. Если появляется элемент с рангом r, то значит произошла операция $join$, где элемент e, переставший быть представителем своего множества, имел ранг r. Тогда посчитаем по всем таким событиям суммарный размер множеств, представителями которых были эти элементы e. Эти множества не пересекаются, так как по лемме 1.3 r наибольший ранг элемента в множестве, а значит одно множество не могло содержать предыдущее, так как в предыдущем уже есть элемент с рангом r, а пересекаться множества не могут, так как в СНМ множества не разбиваются. Значит суммарный размер таких множеств не превосходит количества всех элементов, то есть n. С другой стороны по лемме 1.5 размер каждого такого множества хотя бы 2^r, но тогда суммарный размер этих множеств хотя бы $2^r \cdot count(r) > 2^r \cdot \frac{n}{2^r} = n$. Полученное противоречие означает, что предположение неверно, и $count(r) \le \frac{n}{2^r}$.

Используя результат предыдущих рассуждений получаем, что $\sum\limits_r 1.7^r \cdot count(r) \le \sum\limits_r 1.7^r \cdot \frac{n}{2^r} = n \cdot \sum\limits_r (\frac{1.7}{2})^r$. Так как $\frac{1.7}{2} < 1$, то сумма геометрической прогрессии $\sum\limits_r (\frac{1.7}{2})^r$ не превосходит некоторой константы C, а значит $n \cdot \sum\limits_r (\frac{1.7}{2})^r \le n \cdot C$. Получили, что $\sum\limits_i t_i \le C \cdot n$, то есть, что сумма t_i пропорциональна n.

Получаем, что суммарное время работы всех операций get пропорционально $m \log^*(n) + n + m$. ■

45.3. Улучшение оценки, растущая крутизна

Def 45.3.1. *Крутизна.*

$$f(x,y) = \begin{cases} 0, \text{ если } x < 1.7^y \\ 1 + f(\log_{1.7}(x), y), \text{ иначе} \end{cases}$$

Крутизной элемента e называется число $h_e = f(r_{p_e}, r_e)$. Для представителей множеств h_e равно 0. Если говорить не строго, то h_e означает максимальное количество крутых ребер, что пройдя по этим крутым ребрам ранг увеличится не больше, чем если пройти по ссылке из e в p_e.

Lm 45.3.2. Для данного элемента e величина h_e не уменьшается.

Доказательство. По лемме 2.3 величина r_{p_e} не уменьшается. Так как функция f из определения крутизны монотонна при фиксированном втором параметре (он не изменяется по лемме 2.4), то h_e не уменьшается. ■

Lm 45.3.3. Для данного элемента e после операции get величина h_e выростет хотя бы на количество крутых ребер на пути из p_e до представителя множества.

Доказательство. Воспользуемся обозначениями и вспомогательным утверждением леммы 2.2. $r_R \geq 1.7^{\cdot^{\cdot^{\cdot^{1.7^{r_{p_e}}}}}}$, где количество 1.7 равно E — количеству крутых ребер на пути из p_e до R. Тогда новая крутизна $h_e = f(r_R, r_e) \geq f(1.7^{\cdot^{\cdot^{\cdot^{1.7^{r_{p_e}}}}}}, r_e) = E + f(r_{p_e}, r_e)$, то есть увеличилась хотя бы на E. ■

Lm 45.3.4. Для любого элемента e величина $h_e \leq 2\log^*(n)$.

Доказательство. Доказательство аналогично лемме 2.2, именно в этом смысле крутизна эквивалентна h_e крутым ребрам. ■

Следствие 45.3.5. $\sum\limits_e h_e \leq 2n\log^*(n)$.

Следствие 45.3.6. Для данного элемента e после операции *get* величина h_e выростет хотя бы на суммарную крутизну ребер на пути из p_e до представителя множества.

Доказательство. Аналогично лемме 3.3. ■

Теорема 45.3.7. Суммарное время работы всех операций *get* пропорционально $n\log^*(n)+n+m$.

Доказательство. Время работы i-ой операции *get* пропорционально $u_i + t_i + 1$:

1. u_i означает «количество крутых ребер на пути операции минус 1» или 0, если крутых ребер нет. Логически из рассмотрения выкидывается последнее крутое ребро на пути операции.
2. t_i аналогично теореме 2.6.
3. 1. Аналогично теореме 2.6, но сюда также включен проход по последнему крутому ребру на пути операции.

Значит суммарное время работы всех операций *get* пропорционально $\sum\limits_i (u_i + t_i + 1) = \sum\limits_i u_i + \sum\limits_i t_i + \sum\limits_i 1 = \sum\limits_i u_i + \sum\limits_i t_i + m$. Из теоремы 2.6 сумма $\sum\limits_i t_i$ пропорциональна n.
У каждого элемента, ссылка которого учтена в u_i, крутость после операции *get* увеличится хотя бы на 1 (лемма 3.3). То есть после i-ой операции *get* суммарная крутость возрастет хотя бы на u_i, так как крутости остальных элементов не уменьшаются (лемма 3.2). То есть после всех операций *get* суммарная крутость будет хотя бы $\sum\limits_i u_i$. Но по следствию 3.5: $\sum\limits_i u_i \leq \sum\limits_e h_e \leq 2n\log^*(n)$.
Получили, что суммарное время работы всех операций *get* пропорционально $n\log^*(n) + n + m$. ■

45.4. Альтернативная оценка, классы крутизны

Def 45.4.1. *Класс крутизны. Каждый элемент e в зависимости от его крутизны попадает в некоторый класс крутизны, который обозначается* c_e. *А именно:* $c_e = [\log(h_e)]$.

Следствие 45.4.2. Из определения сразу следует, что классов крутизны не больше, чем $\log(\max\limits_e\{h_e\})$, что не больше, чем $\log(2\log^*(n)) = \log(\log^*(n)) + 1$ (по лемме 3.4).

Lm 45.4.3. Для данного элемента e величина c_e не уменьшается.

Доказательство. По лемме 3.2 величина h_e не уменьшается. Так как log монотонная функция, то c_e не уменьшается. ■

Lm 45.4.4. Если во время операции *get* над крутым ребром элемента e_1 было крутое ребро элемента e_2 (то есть e_2 было на пути до корня позже, чем e_1), и $c_{e_1} \leq c_{e_2}$, то класс крутизны элемента e_1 после операции *get* строго увеличится.

Доказательство.] c'_{e_1} и h'_{e_1} — это новый класс крутизны и новая крутизна элемента e_1 соответственно. Тогда по следствию 3.6: $h'_{e_1} \geq h_{e_1} + h_{e_2} = 2^{\log(e_1)} + 2^{\log(e_2)} \geq 2^{[\log(e_1)]} + 2^{[\log(e_2)]} = 2^{c_{e_1}} + 2^{c_{e_2}} \geq 2^{c_{e_1}} + 2^{c_{e_1}} = 2 \cdot 2^{c_{e_1}} = 2 \cdot 2^{[\log(h_{e_1})]}$. Тогда $c'_{e_1} = \log(h'_{e_1}) \geq \log(2 \cdot 2^{[\log(h_{e_1})]}) = 1 + [\log(h_{e_1})] = c_{e_1} + 1 \Rightarrow c'_{e_1} > c_{e_1}$. ■

Теорема 45.4.5. Суммарное время работы всех операций *get* пропорционально $(n + m)\log((\log^*(n))$.

Доказательство. Время работы i-ой операции *get* пропорционально $\log((\log^*(n)) + v_i + t_i + 1$:

1. $\log((\log^*(n))$. Крутых ребер на пути, у которых класс крутости не поменялся будет не больше, чем $\log(\log^*(n)) + 1$. Пусть это не так. Но по следствию 4.2 среди таких ребер существуют два с одинаковым классом крутизны, а по лемме 4.4 одно из них поменяло свой класс крутизны, получили противоречие.

2. v_i означает количество крутых ребер на пути, у которых класс крутости поменялся.

3. t_i аналогично теореме 2.6.

4. 1. Аналогично теореме 2.6.

Значит суммарное время работы всех операций *get* пропорционально $\sum\limits_i (\log((\log^*(n)) + v_i + t_i + 1) = \sum\limits_i \log((\log^*(n)) + \sum\limits_i v_i + \sum\limits_i t_i + \sum\limits_i 1 = m\log((\log^*(n)) + \sum\limits_i v_i + \sum\limits_i t_i + m$. Из теоремы 2.6 сумма $\sum\limits_i t_i$ пропорциональна n.
Так как у каждого элемента класс крутизны только увеличивается (лемма 4.3), а классов крутизны не больше чем $\log((\log^*(n)) + 1$ (следствие 4.2), то каждый элемент будет изменять свой класс крутизны не больше чем $\log((\log^*(n)) + 1$ раз. Значит $\sum\limits_i v_i \leq n\log((\log^*(n))$.
Получили, что суммарное время работы всех операций *get* пропорционально $(n + m)\log((\log^*(n))$. ■

45.5. Функция Аккермана

Def 45.5.1. *Функция Аккермана.*
$A_k(t) : \mathbb{N} \to \mathbb{N}$

$$A_k(t) = \begin{cases} t + 1, \text{при } k = 0 \\ A_{k-1}^{(t+1)}(t), \text{иначе} \end{cases}$$

Где $f^{(t)} = f \circ \cdots \circ f$ t раз, то есть композиция функции f с собой t раз.

Lm 45.5.2. $A_k(t) > t$, A_k монотонно возрастает.

Доказательство. Доказательство по индукции.
База $k = 0$ — тривиально.
Переход $k-1 \to k$. $A_k(t) = A_{k-1}^{(t+1)}(t)$. Каждое очередное применение функции A_{k-1} дает большее чем аргумент число, значит результат больше, чем t, то есть $A_k(t) = A_{k-1}^{(t+1)}(t) > t$.
Так как $t < A_{k-1}(t)$, то $A_{k-1}^{(t+1)}(t) < A_{k-1}(A_{k-1}^{(t+1)}(t)) = A_{k-1}^{(t+2)}(t)$. Но а так как A_{k-1} монотонна, то $A_k(t) = A_{k-1}^{(t+1)}(t) < A_{k-1}^{(t+2)}(t+1) = A_k(t+1)$. ■

Lm 45.5.3. $f(t) = A_t(1)$ монотонно возрастает.

Доказательство. $A_{t+1}(1) = A_t(A_t(1)) > A_t(1)$, так как $A_t(1) > 1$, а A_t монотонна (лемма 5.2) ■

Выпишем первые несколько функций явно:

1. $A_1(t) = A_0^{(t+1)}(t) = 2t + 1$. Последнее равенство доказывается по индукции.
2. $A_2(t) = A_1^{(t+1)}(t) = 2^{t+1}(t+1) - 1 \geq 2^t$. Последнее равенство доказывается по индукции.

Def 45.5.4. *Обратная функция Аккермана.* $\alpha(t) = \min\{k \mid A_k(1) \geq t\}$

Посчитаем несколько первых значений α:

0. $A_0(1) = 1 + 1 = 2$.
1. $A_1(1) = 2 \cdot 1 + 1 = 3$.
2. $A_2(1) = 2^2 \cdot 2 - 1 = 7$.
3. $A_3(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2047$.
4. $A_4(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047) = A_2^{(2047)}(A_2(2047)) > A_2^{(2047)}(2^{2047}) > \cdots > 2^{2^{\cdots^{2^{2047}}}}$, где количество 2 равно 2048. Последняя серия неравенст следует из монотонности A_2 (лемма 5.2). Получившаяся оценка снизу на $A_4(1)$ невероятно большое число, превышающее число атомов в наблюдаемой части Вселенной.

Последнее высказывание означает, что для чисел, которые встречаются на практике $\alpha(t) \leq 4$.

45.6. СНМ и Аккерман

Def 45.6.1. *Порядок элемента. Для каждого элемента e, который не является представителем своего множества, определим величину $level(e) = \max\{k \mid r_{p_e} \geq A_k(r_e)\}$.*

Def 45.6.2. *Характеристика порядка. Для каждого элемента e, который не является представителем своего множества, определим величину $iter(e) = \max\{k \mid r_{p_e} \geq A_{level(e)}^{(k)}(r_e)\}$.*

Lm 45.6.3. Для любого элемента e, не являющегося представителем своего множества, верно, что $0 \leq level(e) < \alpha(n)$.

Доказательство. $level(e) \geq 0$, так как $A_0(r_e) = r_e + 1 \leq r_{p_e}$ по лемме 1.3.
Пусть $level(e) \geq \alpha(n)$. $k = level(e)$. По определению $r_{p_e} \geq A_k(r_e) \geq A_k(1)$, так как A_k монотонна (лемма 5.2).
Но ранг любого элемента не превосходит $n-1$, так как ранг одного элемента увеличивается не больше чем на 1 за одну операцию $join$, но всего операций $join$, которые действительно соединяют два множества, может быть не больше чем $n-1$, так как за одну операцию $join$ количество множеств уменьшается на 1.
Получаем, что $n > r_{p_e} \geq A_k(1)$. Но по предположению $k \geq \alpha(n) \Rightarrow A_k(1) \geq A_{\alpha(n)}(1) \geq n$, первое неравенство по лемме 5.3. Получили противоречие. Значит $level(e) < \alpha(n)$. ■

Lm 45.6.4. Для любого элемента e, не являющегося представителем своего множества, верно, что $1 \leq iter(e) \leq r_e$.

Доказательство. Так как $r_{p_e} \geq A_{level(e)}(r_e)$, то $iter(e) \geq 1$.
$r_{p_e} < A_{level(e)+1}(r_e) = A^{(r_e+1)}_{level(e)}(r_e) \Rightarrow iter(e) < r_e + 1 \Rightarrow iter(e) \leq r_e$. ■

Будем доказывать время работы CHM методом амортизационного анализа.

Def 45.6.5. *Потенциал элемента. Определим для любого элемента e величину* $\varphi(e)$.
$$\varphi(e) = \begin{cases} \alpha(n) \cdot r_e, \text{если } e \text{ представитель своего множества} \\ (\alpha(n) - level(e)) \cdot r_e - iter(e), \text{иначе} \end{cases}$$

Def 45.6.6. *Потенциал. Для амортизационного анализа возьмем потенциал для i-ой операции* $\varphi_i = \sum\limits_e \varphi(e)$.

Lm 45.6.7. $\varphi_0 = 0$, $\varphi_i \geq 0$.

Доказательство. Так как в начальный момент времени все элементы являются представителями своих одноэлементных множеств, то $\forall e : \varphi(e) = \alpha(n) \cdot r_e = \alpha(n) \cdot 0 = 0 \Rightarrow \varphi_0 = \sum\limits_e \varphi(e) = \sum\limits_e 0 = 0$.
Так как ранг и α величины неотрицательные, то $\alpha(n) \cdot r_e \geq 0$. Так как $level(e) < \alpha(n)$ (лемма 6.3), то $(\alpha(n) - level(e)) \cdot r_e \geq r_e$, а так как $iter(e) \leq r_e$ (лемма 6.4), то $(\alpha(n) - level(e)) \cdot r_e - iter(e) \geq r_e - iter(e) \geq 0$. Получили, что $\forall e : \varphi(e) \geq 0$. Значит $\varphi_i = \sum\limits_e \varphi(e) \geq 0$. ■

Теорема 45.6.8. Для любой i-ой операции, имеющей тип $join$, амортизированное время работы $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$

Доказательство. Так как φ_i зависит от $\varphi(e)$, который зависит от r_e, $level(e)$ и $iter(e)$, которые зависят от r_{p_e}, то потенциал изменится только у тех элементов, у которых изменился ранг или ранг предка, а также у тех элементов, которые перестали быть представителями своего множества. Так как операция $join(e_1, e_2)$ может поменять ранг одного элемента, пусть e_1 (то есть он стал представителем объединения множеств), а элемент e_2 перестал быть представителем своего множества, то $\varphi(e)$ мог поменяться у e_1, e_2 и тех элементов, у которых $p_e = e_1$ (обозначим множество таких элементов за S).
Обозначим за $\varphi'(e)$, $level'(e)$ и $iter'(e)$ потенциал, порядок и характеристику порядка элемента e после операции $join$ соответственно. Тогда $\Delta\varphi(e_2) = \varphi'(e_2) - \varphi(e_2) = (\alpha(n) - level(e_2)) \cdot r_{e_2} -$

$iter(e_2) - (\alpha(n) \cdot r_{e_2}) = -level(e_2) \cdot r_{e_2} - iter(e_2)$, так как $level$ неотрицательный, а $iter$ положительный (леммы 6.3 и 6.4), то $\Delta\varphi(e_2) < 0$.
Если ранг e_1 остался прежним, то $\forall e \in S : \Delta\varphi(e) = 0$, так как r_e, $level(e)$ и $iter(e)$ остались прежними.
Если ранг e_1 поменялся, то если для некоторого элемента $e \in S$ $level(e)$ остался прежним, то $iter(e)$ не уменьшился, так как иначе бы это означало, что r_{p_e} уменьшился, что противоречит тому, что он увеличился. Если $iter(e)$ не изменился, то $\varphi(e)$ остался прежним, так как величины от которых он зависит не изменились. Если $iter(e)$ уменьшился, то $\Delta\varphi(e) = \varphi'(e) - \varphi(e) = (\alpha(n) - level'(e)) \cdot r_e - iter'(e) - ((\alpha(n) - level(e)) \cdot r_e - iter(e)) = (\alpha(n) - level(e)) \cdot r_e - iter'(e) - (\alpha(n) - level(e)) \cdot r_e + iter(e) = iter(e) - iter'(e) < 0$.
Если ранг e_1 поменялся, то если для некоторого элемента $e \in S$ $level(e)$ тоже поменялся, то $\Delta\varphi(e) = \varphi'(e) - \varphi(e) = (\alpha(n) - level'(e)) \cdot r_e - iter'(e) - ((\alpha(n) - level(e)) \cdot r_e - iter(e)) = (\alpha(n) - level'(e)) \cdot r_e - iter'(e) - (\alpha(n) - level(e)) \cdot r_e + iter(e) = r_e \cdot (level(e) - level'(e)) + iter(e) - iter'(e) \leq -r_e + iter(e) - iter'(e)$, так как $iter(e) \leq r_e$ (лемма 6.4), то $\Delta\varphi(e) = -r_e + iter(e) - iter'(e) \leq -iter'(e) < 0$, так как $iter'(e) > 1$ (лемма 6.4).
Получили, что изменение потенциала у всех элементов, кроме e_1 неположительное.
Если r_{e_1} не изменился, то $\Delta\varphi(e_1) = 0$, иначе $\Delta\varphi(e_1) = \varphi'(e_1) - \varphi(e_1) = \alpha(n) \cdot (r_{e_1} + 1) - \alpha(n) \cdot (r_{e_1}) = \alpha(n)$.
Получили, что $\Delta\varphi = \sum\limits_e \Delta\varphi(e) \leq \Delta\varphi(e_1) \leq \alpha(n)$. Но так как $t_i = \mathcal{O}(1)$, то $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$. ■

Теорема 45.6.9. Для любой i-ой операции, имеющей тип get, амортизированное время работы $a_i = t_i + \Delta\varphi = \mathcal{O}(\alpha(n))$

Доказательство. Аналогично рассуждениям из теоремы 6.8, потенциал меняется только у элементов на пути операции get, причем потенциал не увеличивается. i-ая операция get работает пропорционально $w_i + x_i + 1$:

1. w_i означает элементы e на пути операции get, у которых $\Delta\varphi(e) < 0$.
2. x_i означает элементы e на пути операции get (кроме представителя множества), у которых $\Delta\varphi(e) = 0$.
3. 1. Некоторое константное количество действий. Сюда включено посещение представителя множества.

Если порядок некоторого элемента e или характеристика порядка изменились, то они увеличились, так как величина r_{p_e} увеличилась, но тогда аналогично рассуждениям из теоремы 6.8 величина $\Delta\varphi(e) < 0$.
Пусть $x_i > \alpha(n)$. Тогда так как количество различных порядков $\alpha(n)$ (лемма 6.3), то $\exists e_1, e_2$ на пути операции get, такие, что $level(e_1) = level(e_2)$. Пусть e_2 находится ближе к представителю множества. Обозначим представителя множества за R, а $\varphi'(e)$, $level'(e)$ и $iter'(e)$ аналогично теореме 6.8. Тогда $r_R \geq r_{p_{e_2}} \geq A_{level(e_2)}(r_{e_2}) = A_{level(e_1)}(r_{e_2}) \geq A_{level(e_1)}(r_{p_{e_1}})$ значит если $level'(e_1) = level(e_1)$, то $iter'(e_1) > iter(e_1)$, так как в этом случае $r_{p_{e_1}} \geq A^{(iter(e_1))}_{level(e_1)}(r_{e_1}) = A^{(iter(e_1))}_{level'(e_1)}(r_{e_1}) \Rightarrow r_R \geq A_{level(e_1)}(r_{p_{e_1}}) = A_{level'(e_1)}(r_{p_{e_1}}) \geq A_{level'(e_1)}(A^{(iter(e_1))}_{level'(e_1)}(r_{e_1})) = A^{(iter(e_1)+1)}_{level(e_1)}(r_{e_1})$.
Получаем противоречие, так как у элемента e_1 величина $\Delta\varphi(e_1) < 0$. Значит $x_i \leq \alpha(n)$.
Получаем, что величина a_i пропорциональна $w_i + x_i + 1 + \Delta\varphi$, но сумма каждого элемента e, учтенного в w_i, со своим $\Delta\varphi(e)$ неположительна, а $x_i \leq \alpha(n)$, значит $a_i = \mathcal{O}(\alpha(n))$. ■

Теорема 45.6.10. Суммарное время работы всех операций с СНМ пропорционально $(m+q) \cdot \alpha(n)$.

Доказательство. $\sum\limits_i t_i = \sum\limits_i a_i - \varphi_{m+q} + \varphi_0$. Но по лемме 6.7 величина $\varphi_0 = 0$, а $\varphi_{m+q} \geq 0$, а значит $\sum\limits_i t_i = \sum\limits_i a_i - \varphi_{m+q} \leq \sum\limits_i a_i$, но сумма $\sum\limits_i a_i$ пропорциональна $(m+q) \cdot \alpha(n)$. ■

Лекция по алгоритмам #46
Минимальное остовное дерево (MST)

7 декабря

46.1. Определение

Дан связный неориентированный взвешенный граф. Остовное дерево (spanning tree) - ациклический связный подграф данного графа, содержащий все его вершины. Минимальное остовное дерево (minimum spanning tree, MST) - остовное дерево минимального веса.

46.2. Алгоритмы поиска

46.2.1. Алгоритм Краскала

Отсортируем все рёбра в порядке возрастания весов. Заведём для вершин СНМ. Изначально каждая вершина находится в своей компоненте. Перебираем рёбра: если оба конца ребра находятся в одной компоненте СНМ, то проигнорируем ребро, иначе объединим эти компоненты и добавим ребро в MST.

```
dsu s(n);
vector <Edge> mst;
sort(edge.begin(), edge.end(), [](Edge x, Edge y){return x.weight < y.weight;});
for (auto e: edge) {
  if (s.get_parent(e.u) != s.get_parent(e.v)) {
    s.join(e.u, e.v);
    mst.push_back(e);
  }
}
```

Время работы: $\mathcal{O}(E \log E)$ - сортировка, $\mathcal{O}(E \log V)$ или $\mathcal{O}(E\alpha(V))$ - операции с СНМ. Отсюда $T = \mathcal{O}(E \log E) = \mathcal{O}(E \log V)$ (т. к. в графах без кратных рёбер $E \leq V^2$).

46.2.2. Алгоритм Прима

Изначально MST состоит из одной вершины. Для каждой вершины v поддерживаем ребро минимального веса из построенной части MST в v. $d[v]$ и $par[v]$ - соответственно вес ребра и номер другого конца.

1. Добавляем минимальное ребро, выходящее из построенной части MST.

2. Обработаем новую вершину u, добавленную только что в MST: для всех ещё не добавленных вершин v, смежных с u, прорелаксируем $d[v]$ и $par[v]$ ребром (u, v).

3. Возвращаемся в пункт 1, и так до тех пор, пока не добавим все вершины.

Алгоритм Прима очень похож на алгоритм Дейкстры, единственное отличие - релаксация.

```
//Dijkstra
if (d[u] + weight(u, v) < d[v]) {d[v] = d[u] + weight(u, v); par[v] = u;}
//Prim
if (weight(u, v) < d[v]) {d[v] = weight(u, v); par[v] = u;}
```

Время работы:

Базовая реализация: $\mathcal{O}(V^2 + E)$ (оптимально для полных и почти полных графов).

Для разреженных графов выигрыш по времени даёт использование кучи для быстрого нахождения минимального ребра. Как и в алгоритме Дейкстры, потребуется $\mathcal{O}(V)$ операций ExtractMin и $\mathcal{O}(E)$ операций DecreaseKey.

Двоичная куча: $\mathcal{O}((V+E)\log V)$ (нет смысла использовать, т. к. алгоритм Краскала пишется быстрее и работает не хуже).

Фибоначчиевая куча: $\mathcal{O}(V\log V + E)$.

Radix Heap использовать нельзя, т. к. в отличие от алгоритма Дейкстры, извлекаемые из кучи значения не обязательно возрастают.

46.2.3. Алгоритм Борувки

1. Для каждой вершины найдём минимальное инцидентное ей ребро. Добавим все такие рёбра в MST.

2. Сжимаем компоненты связности добавленных рёбер в одну вершину, избавляемся в сжатом графе от петель и кратных рёбер (мы это умеем делать за $\mathcal{O}(V+E)$).

3. Возвращаемся в пункт 1 и проделываем то же самое на сжатом графе, и так до тех пор, пока в графе больше одной вершины.

Минимизировать в пункте 1 нужно пару <вес ребра, id ребра>, чтобы никакие два ребра не были равными. Иначе могут возникнуть проблемы, к примеру, на полном графе из трёх вершин с одинаковыми весами всех рёбер.

Время работы:

1. $T = \mathcal{O}((V+E)\log V)$

На каждой итерации число вершин уменьшается хотя бы в два раза.

2. $T = \mathcal{O}(V^2)$

Пусть $E = V^2$. Тогда (т. к. мы избавляемся от кратных рёбер) $T = V^2 + (\frac{V}{2})^2 + (\frac{V}{4})^2 + \ldots = \mathcal{O}(V^2)$.

3. $T = \mathcal{O}((V+E)\log\frac{V^2}{E})$

Начиная с некоторого k, $(\frac{V}{2^k})^2 \le E$. Рассмотрим шаги алгоритма:

$V \to \frac{V}{2} \to \frac{V}{4} \to \ldots \to \frac{V}{2^k} \to \ldots$

До k-го шага будет выполнено $\mathcal{O}((V+E)\log\frac{V^2}{E})$ действий (по утверждению 1).

Начиная с k-го шага, будет выполнено $\mathcal{O}(E)$ действий (по утверждению 2).

Отсюда $T = \mathcal{O}((V+E)\log\frac{V^2}{E})$.

46.2.4. Доказательство корректности

Лемма о разрезе:

Пусть у нас есть некоторый разрез графа: разбиение вершин на множества S и $V \setminus S$. Ребро проходит через разрез, если его концы лежат в разных множествах. Предположим, что у нас есть некоторое множество F рёбер, не проходящих через разрез, и известно, что F является подмножеством рёбер некоторого MST. Пусть e - минимальное ребро, проходящее через разрез. Тогда множество $F \cup \{e\}$ тоже является подмножеством рёбер некоторого MST.

Доказательство:

Предположим, что есть MST, не содержащее e и содержащее F. Значит, через разрез проходит некоторое ребро e'. Добавим к MST ребро e, получаем граф с циклом, проходящим через e и e'. Удалим e'. Получится вновь дерево, причём (т. к. $weight(e) \le weight(e')$) вес полученного дерева не больше веса MST. Отсюда существует MST, содержащее $F \cup \{e\}$. Лемма доказана.

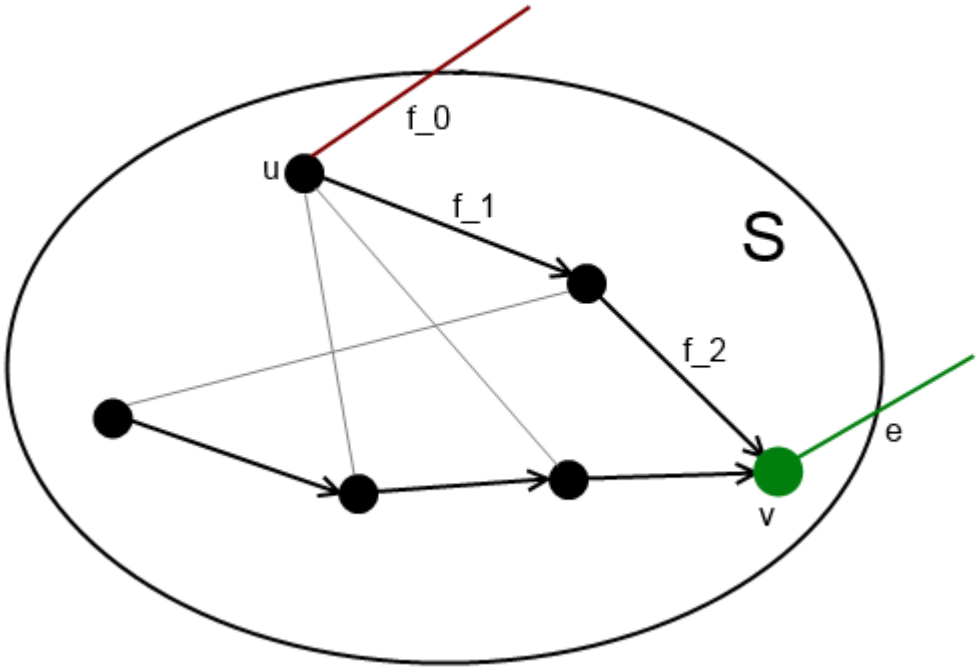
Применение леммы о разрезе к доказательству алгоритмов:

Алгоритм Краскала: множество S - компонента СНМ, в которой находится один из концов добавляемого ребра.

Алгортим Прима: множество S - вершины построенной части MST.

Алгоритм Борувки: здесь уже сложнее, т. к. мы одновременно добавляем несколько рёбер. Пусть у нас уже добавлены некоторые рёбра. Хотим добавить минимальное ребро, инцидентное некоторой вершине v (назовём его e), если оно ещё не добавлено. В качестве множества S рассмотрим множество вершин, из которых по добавленным рёбрам достижима вершина v.

Нужно доказать, что e удоволетворяет лемме о разрезе: т. е. является минимальным ребром, проходящим через разрез. Рассмотрим произвольную вершину $u \neq v$ множества S и произвольное ребро f_0, инцидентное u и проходящее через разрез. Очевидно, что f_0 больше взятого из вершины u ребра (обозначим f_1). Очевидно, каждое следующее ребро на пути от u к v по добавленным рёбрам $(f_2, f_3, \ldots)$ будет меньше предыдущего (по этой же причине в алгоритме не может получиться цикл). По транзитивности получаем $f_0 < f_1 < f_2 < \ldots < e$, т. е. минимальное ребро, проходящее через разрез - ребро e, и по лемме о разрезе мы можем его взять.



Лекция по алгоритмам #47
Хаффман и жадность

11 декабря

47.1. Задача о шахматном коне (Правило Варнсдорфа)

47.1.1. Постановка задачи

Найти гамельтонов путь на шахматном поле n на n ходом коня.

47.1.2. Правило Варнсдорфа

Давайте строить путь жадно. На каждой итерации следует добавлять вершину наименьшей остаточной степени (из ещё не посещённых и доступных из последней клетки пути). Утверждается, что с большой вероятностью мы найдём гамельтонов путь.

47.1.3. Куда идти, если вершин наименьшей степени несколько?

1) “Куда-нибудь” - плохо. Возможна ситуация, когда такой алгоритм не найдёт путь.

2) Random - хорошо. С большой вероятностью путь будет найден (а если нет, можно попробовать ещё раз т.к. алгоритм недетерминированный).

3) Утверждается, что хорошо работает стратегия “прижиматься к краю” т.е. при равенстве остаточных степеней выбирать вершину, расстояние от которой до ближайшего края меньше. При равенстве расстояний - случайную.

47.1.4. Откуда начинать?

1) Можно пробовать стартовать со случайных клеток.

2) Можно перед этим попробовать начать с “особенных” клеток (угловой, крайней, центральной).

47.1.5. Обобщение для произвольного графа (всё ещё жадность)

Берём вершину минимальной степени, среди таких - случайную. Начинаем со случайной.

47.1.6. Улучшение

1) Остаток должен быть связен и все вершины должны быть достижимы из текущей вершины (dfs). Иначе гамельтонов путь с таким префиксом не получится.

2) Существует не более одной вершины степени 1. Иначе гамельтонов путь с таким префиксом не получится.

3) Пробовать ходить в первые k вершин в порядке увеличения степени. Чем меньше k тем больше вероятность не найти путь, чем больше k тем дольше алгоритм работает.

47.2. Хаффман и сжатие данных

47.2.1. Идеальный архиватор

Не существует алгоритма, который сжимает любую битовую строку длины n (без потери информации).

Действительно, строк длины n ровно 2^n, а строк строго меньшей длины ровно $2^n - 1$, что, очевидно, меньше.

47.2.2. Кодирование

$\sum$ - алфавит, $|\sum| = n$.

Не любой набор кодов подходит для кодирования. Например, если мы выбрали коды для символов так: a -> 0, b -> 1, c -> 01, то запись 01 нельзя будет декодировать однозначно (подойдёт и “ab”, и “c”).

Набор слов называется беспрефиксным, если никакое слово не является префиксом другого.

Заметим, что если коды обладают этим свойством (беспрефиксностью), то любую корректную запись можно однозначно декодировать. Для эффективной расшифровки можно использовать структуру данных Бор, в которой будут храниться коды.

Мы будем кодировать текст следующим образом: первые k бит (заранее известное число) - длина закодированного текста, потом какое-то количество бит отведено под кодирование дерева кодов, потом последовательная запись кодов символов.

47.2.3. Алгоритм Хаффмана

Алгоритм Хаффмана подбирает такой беспрефиксный набор кодов к алфавиту, что последовательная запись кодов символов текста, имеет минимальную возможную длину. Иными словами, если cnt_i - количество вхождений i-ого символа, а len_i - длина кода, соответствующего i-ому символу, то $\sum(cnt_i \cdot len_i) \to min$.

47.2.4. Наблюдения

1) Из двух различных кодов более короткий следует отдать символу, который используется чаще. (Действительно, попробуем сделать наоборот, и заметим, что суммарная длина неуменьшится).

2) Заметим, что коды могут заканчиваться только в листях бора. Действительно, иначе такой код будет префиксом другого кода (вершина имела детей => спустившись вниз по бору мы получим ещё какой-то код).

3) Рассмотрим бор из кодов. Заметим, что любая вершина должна быть либо листом, либо иметь два ребёнка. Действительно, пусть какая-то вершина имеет одного ребёнка. Удалим её, а её ребёнка подвесим к предку. Беспрефиксность не сломалась, а какой-то код (как минимум один, к которому вела ветка) стал короче.

47.2.5. Алгоритм

Попытаемся выбрать оптимальное дерево кодов.

Представим, что мы уже нашли ответ. Рассмотри самую глубокую ветку. Предпоследняя вершина в этой ветке обязана иметь два ребёнка (а т.к. рассматриваемая ветка была самая глубокая, то второй ребёнок тоже обязан быть листом). Из наблюдения №1 следует, что коды, соответствующие этим листям, должны использоваться для кодирования самых редких символов (Если таких несколько - выберем любые два). Пусть эти символы c_1 и c_2. Удалим оба листа

из дерева, а их предку поставим в соответствие мнимый символ, количество вхождений которого будет равно сумме частот вхождений c_1 и c_2. Пусть мы теперь найдём оптимальное решение для новой задачи. Заметим, что если мы теперь вернём листы с $c1$ и $c2$ в дерево (проведём из мнимой вершины рёбра 0 и 1 в c_1 и c_2 соответственно), то величина $[\sum(cnt_i \cdot\ len_i)]$ изменится на константу (на суммарное количество вхождений c_1 и c_2). Тоже самое происходило и при удалении этих вершин (указанная величина тоже менялась на константу). Таким образом, из оптимального дерева для новой задачи должно получаться оптимальное решение для старой задачи. Тогда мы можем решать задачу итеративно, уменьшая на каждом этапе количество символов в алфавите на 1. Последнее наблюдение - алфавиту из одного символа должно соответствовать дерево из одной вершины (Действительно, рассмотрим корень дерева. Двух детей у него быть не может т.к. дерево должно содержать только один код; одного ребёнка быть не может по наблюдению №3, а значит, корень не имеет детей).

47.2.6. Сохранение дерева кодов

К сожалению, для декодирования текста нам нужно опять построить дерево кодов. Существует несколько способов передачи дерева:

1) Можно передавать всю таблицу количества вхождений. Это займёт $n \cdot k + 32$ бита, где k - минимальное количество бит, необходимое для кодирования количества вхождений ($k = \log_2 MAX\_CNT$ с округлением вверх, где MAX_CNT - количество вхождений самого частого символа). 32 бита (например) уйдёт на кодирование дополнительной информации (k или длины всей таблицы).

2) Можно закодировать сразу само дерево. Например, так:

Закодируем поддерево вершины A. Если A имеет детей, то запишем “0(код левого поддерева)(код правого поддерева)”, иначе (A - лист) “1(код символа, которому соответствует этот лист)”.

Лекция по алгоритмам #48
Жадность или получаем зачёты оптимальным образом

11 декабря

48.1. Методы решения некоторых задач теории расписаний

Решим несколько задач, в которых нужно составить расписание выполнения некоторых абстрактных работ, которое удовлетворяет некоторым условиям и/или оптимизирует некую величину.

• t_i — **время выполнения i-й работы, выполнить максимальное количество работ, чтобы суммарное время выполнения не превышало T**

Решение: отсортируем работы по t_i и будем набирать, начиная с меньших, пока возможно. $\mathcal{O}(Sort)$.

Набросок доказательства: можно заменить работу с большим t_i на меньшее.

Замечание: можно за $\mathcal{O}(n)$ модифицированным алгоритмом поиска порядковой статистики.

• t_i, d_i — **дедлайн (максимальный момент времени, до которого нужно закончить выполнение работы), можно ли выполнить все работы**

Решение: нужно пробовать выполнять все работы в порядке увеличения дедлайна.

Доказательство: пусть есть какой-то корректный порядок выполнения (если он есть), тогда в нём можно переместить работу с минимальным дедлайном в начало списка выполнения. Действительно, идущие перед ней работы теперь будут заканчиваться не позже, чем заканчивалась она, но её дедлайн — минимальный, поэтому все работы все ещё можно выполнить. Дальше делаем аналогично.

• t_i, d_i — **дедлайн, но не конца, а начала времени выполнения**

Решение: отсортировать в порядке увеличения $(t_i + d_i)$.

Доказательство: очевидное доказательство — свести к предыдущей задаче. Тем не менее, для расширения сознания, приведём другой способ решения этой задачи. Пусть j-я работа выполняется последней. Тогда $(\sum t_i) - t_j \le d_j \Rightarrow CONST \le t_j + d_j$. Тогда заведомо не хуже выбрать в качестве последней работы ту, у которой $(t_i + d_i)$ максимально.

• t_i, $fee_i > 0$ — **штраф, минимизировать $\sum T_i \cdot fee_i$, где T_i — момент окончания выполнения i-й работы**

Решение: выполнять работы в порядке уменьшения $\dfrac{fee_i}{t_i}$.

Доказательство: возьмём произвольный порядок выполнения работ, пусть есть две соседние работы A и B (A прямо перед B), такие что $\dfrac{fee_A}{t_A} < \dfrac{fee_B}{t_B} \Rightarrow t_B fee_A < t_A fee_B$, тогда рассмотрим, как поменяется целевая функция если поменять эти две работы местами:

$$t_A \cdot fee_A + (t_A + t_B) \cdot fee_B + CONST \to t_B fee_B + (t_A + t_B) fee_A + CONST$$

$$t_A fee_B \to t_B fee_A$$

Значит, таким действием мы только улучшим ответ.

Чтобы доказательство методом *swap*-ов работало, нужно доказать, что при равенстве сравниваемых величин (в нашем случае $\frac{fee_i}{t_i}$) не важно, в каком порядке идут работы A и B. В данной задаче это очевидно.

Многие задачи, подобные рассмотренным, решаются сортировкой работ в соответствии с некоторые компаратором $less(a, b)$, который проверяет, правда ли выгоднее поставить работу a перед работой b. Более того, этот компаратор имеет вид

```
1    less(a, b): return F(a, b) < F(b, a)
```

, где $F(a, b)$ — какая-то "величина оптимальности", если поставить a перед b.

Например, решим третью задачу про дедлайн начала таким способом.

Пусть есть две соседние задачи A и B, а суммарное время выполнения всех задач перед ними равно T. Если задача A идёт перед задачей B, то должны выполняться условия $T \le d_A$ и $T + t_A \le d_B$, то есть $T \le \min(d_A, d_B - t_A)$. Аналогично, если B идёт перед A, то $T \le \min(d_B, d_A - t_B)$. Видно, что выгоднее выбрать такой вариант, в котором ограничение сверху на T как можно более свободное, то есть тот, у которого соответствующий минимум максимален.

К сожалению, на паре не было времени, чтобы показать, что такой порядок транзитивен.

48.2. Задача о двух станках или $F_2||C_{max}$

Есть два станка и несколько деталей, каждую из которых нужно обработать сначала на первом станке, а затем на втором. Время обработки i-й детали на первом станке равна a_i, а на втором — b_i. Необходимо минимизировать момент завершения обработки всех деталей.

Утверждение: выполнять работы на обоих станках надо в одном и том же порядке. Действительно, пусть какой-то префикс работ на обоих станках совпадает, а затем на втором станке обрабатывается i-я деталь. Тогда можно и на первом станке i-ю деталь обрабатывать сразу после найденного префикса (все детали, которые сдвинулись на первом станке всё равно будут обработаны, так как они обрабатываются после i-й). Тем самым мы увеличим префикс. Такими действиями дойдём до того, что порядок обработки деталей на обоих станках одинаков.

Рассмотрим две соседние в этом порядке детали x и y. На минутку забудем, что есть другие детали. Рассмотрим время выполнения этих работ, оно равно $a_x + \max(a_y, b_x) + b_y$ (через $\max(a_y, b_x)$ единиц времени сможет начать обрабатываться на втором станке деталь y). $a_x + b_y + max(a_y, b_x) = a_x + b_y + min(a_y, b_x) + max(a_y, b_x) - min(a_y, b_x) = a_x + a_y + b_x + b_y - min(a_y, b_x)$. Теперь наш компаратор принимает вид:

```
1    less(x,y): return min(a_y,b_x) > min(b_y,a_x)
```

В общем случае, нужно рассмотреть несколько случаев, чтобы показать, что, следуя данному компаратору, выгодно обменять (или не обменивать) две соседние детали местами.

Опять же, окончательное доказательство корректности такого алгоритма приведено не было, либо автор не помнит такого.

48.3. Задача о выполнении максимального количества работ

Рассмотрим ещё одну задачу: есть несколько работ, у каждой своё время выполнения t_i и дедлайн d_i, но теперь нужно не проверить, можно ли выполнить их все, а выполнить как можно больше из них.

Сперва отсортируем все задачи по возрастанию d_i, как было в прошлой задаче. Теперь рассмотрим два решения. Первое работает медленнее и вряд ли пригодится нам с практической точки зрения, но оно поможет доказать второе.

• **Решение динамическим программированием.** Рассмотрим первые i работ и все способы выбрать ровно k из них, чтобы все k можно было выполнить. Тогда пусть $T[i,k]$ — минимальное по всем таким способам суммарное время выполнения этих k работ ($T[i,k] = +\infty$, если таких способов нет). Теперь рассмотрим $(i+1)$-ю работу и прорелаксируем динамику. Либо мы оставляем эту работу в покое ($relax(T[i+1,k], T[i,k])$), либо мы выполняем её ($relax(T[i+1,k+1], T[i,k] + t_{i+1})$), но это возможно только при условии $T[i,k] + t_{i+1} \le d_{i+1}$. Ответ — максимальное k, такое что $dp[n][k] \neq +\infty$.

• **Жадное решение.** Опять же будем рассматривать работы по очереди и для каждого префикса работ строить оптимальное решение определенным образом. Пусть мы построили какой-то ответ и рассматриваем i-ю работу. Есть три случая:

1. Если мы можем добавить её к текущему ответу, то есть $S + t_i \le d_i$, где S — суммарное время выполнения выбранных работ, то просто делаем это.

2. Иначе пробуем качественно улучшить текущий ответ. Если существует среди выбранных такая работа, что её время выполнения больше, чем t_i, то мы можем избавиться от этой работы, а взамен неё поставить i-ю. Ясно, что новый набор работ тоже можно выполнить. (*Реализационное замечание: здесь надо использовать кучу*)

3. Иначе ничего не происходит.

Пусть на i-м шаге k_i — количество выбранных работ, а $a_1 \le a_2 \le \ldots \le a_{k_i}$ — отсортированные времена выполнения выбранных работ. Утверждается, что k_i — максимальное k, такое что $T[i,k] \neq +\infty$, и что $T[i,m] = \sum_{t=1}^{m} a_t$. Из этого утверждения, в частности, очевидным образом следует коррекность алгоритма.

Докажем его по индукции.

База: $i = 0 \Rightarrow k = 0$.

Переход: $i \to i+1$

1. Пусть OPT_j — оптимальный ответ на j-м шаге. Знаем, что $OPT_i = k_i$. Надо доказать, что $OPT_{i+1} = k_{i+1}$. Если $OPT_{i+1} = OPT_i$, то наш алгоритм не мог улучшить ответ, поэтому всё хорошо. Если $OPT_{i+1} > OPT_i$, то, во-первых, $(i+1)$-я работа точно должна быть выполнена (иначе $OPT_i = OPT_{i+1}$), и наш алгоритм её сможет взять, так как для неё останется времени не меньше, чем в любом оптимальном ответе (ибо $T[i,k]$ — минимум).

2. Заметим, что $T[i+1,m]$ равно либо $T[i,m-1]+t_i$, либо $T[i,m] = T[i,m-1]+a_m$ (последнее равенство по предположению индукции). Теперь легко доказать, вторую часть утверждения. В зависимости от случая (1)-(3) t_{i+1} как-то встраивается (или не встраивается) в последовательность $a_1, \ldots, a_{k_i}$, и внутренней индукцией, в которой база и переходы получаются из вышеописанного пересчёта динамики, показывается что сумма m минимальных работ из новых выбранных равна $T[i+1,m]$.

Лекция по алгоритмам #49
Модификация Фибоначчевой кучи

18 декабря

49.1. Структура

Фибоначчева куча работает быстро. Add и DecreaseKey работают за $\mathcal{O}(1)$, а ExtractMin работает за $\mathcal{O}(\log n)$. Казалось бы, что лучшего и желать нельзя, но все-таки Фибоначчеву кучу можно ускорить, делая ExtractMin за $\mathcal{O}(\log c)$, где c — количество различных весов в куче.

Как же это сделать? Во-первых, вершины кучи будут парами (w, list_w), где w — это вес, а list_w — множество id веса w, хранимое в виде списка (пример: при написании алгоритма Дейкстры в куче будут хранится пары $(\text{dist}[v], \{v : \text{dist}[v] == w\})$). Во-вторых, будем поддерживать массивы HeapNode (HeapNode[w] хранит указатель на вершину веса w) и ListNode (ListNode[id] хранит указатель на id в каком-то list_w).

49.2. Операции

Напишем теперь новые операции для нашей новой кучи (они, в отличие от обычных операции Фибоначчевой кучи, будут отмечатся звездочкой). Для простоты будем считать, что все нужные массивы пересчитываются сами собой (на практике пересчет не должен вызвать проблем) и использовать только массив count, который хранит количество id данного веса.

49.2.1. Add*

```
1   Add*(id, w):
2       if count[w] > 0:
3           count[w] += 1
4       else:
5           Add(id, w)
```

49.2.2. DecreaseKey*

```
1   DecreaseKey*(id, oldW, newW):
2       if count[oldW] > 1:
3           --count[oldW]
4           Add*(id, newW)
5       else:
6           if count[newW] == 0: // Do not create or delete any nodes
7               DecreaseKey(oldW, newW)
8               swap(count[oldW], count[newW])
9           else:
10              count[oldW] = 0
11              ++count[newW]
```

49.2.3. ExtractMin*

```
1   ExtractMin*():
2       while True:
3           w = GetMin()
```

```
4               if count[w] > 0:
5                   break
6               ExtractMin(); // Spend 1 coin
7           if count[w] > 0:
8               --count[w]
9           else:
10              ExtractMin(); // Spend 1 coin
11          // Borrow 2 coins
```

49.3. Доказательство

Легко понять, что новые операции корректны, осталось понять, почему куча стала работать быстрее, то есть доказать, что ExtractMin* работает за $\mathcal{O}(\log c)$.

Воспользуемся методом банковского учета. Обозначим количество монет буквой ϕ (изначально $\phi = 0$). Пусть каждая монета стоит $\log c$. При каждом запуске ExtractMin* мы берем две монеты, а на каждый вызов ExtractMin тратим одну монету.

Нужно доказать, что монет всегда хватает, когда они тратятся, то есть $\phi \geq 0$.

В момент появления в куче вершины веса w отметим $e = \text{list}_w[0]$. Понятно, что e будет удален раньше самой вершины. В момент удаления e возьмем монету, которую потратим в момент удаления вершины.

Лекция по алгоритмам #50
Применение модифицированной кучи Фибоначчи

18 декабря

50.1. Применение в алгоритме Дейкстры непосредственно

Алгоритм Дейкстры, как известно, работает за время $\mathcal{O}(E \cdot \texttt{DecreaseKey} + V \cdot \texttt{ExtractMin})$. Модифицированная куча Фибоначчи позволяет делать операцию `ExtractMin` не за $\mathcal{O}(\log V)$, а за $\mathcal{O}(\log C)$, где C — количество различных ключей, находящихся в данный момент в куче.

Но если веса рёбер в графе — целые числа от 1 до C, то в алгоритме Дейкстры в куче будет как раз таки не более $C + 1$ различных ключей в каждый момент времени (просто потому что в момент релаксации вдоль ребра веса не больше C мы не получим ключа, превосходящего минимальный более, чем на C).

Таким образом, мы добились асимптотики $\mathcal{O}(E + V \log C)$.

50.2. Применение в Two-Level Radix Heap

Two-Level Radix Heap без модификаций работает за время $\mathcal{O}(n(\log_t C + t))$, где $\log_t C$ берётся из поиска непустого бакета, а t — из поиска непустого подбакета. Если искать непустой подбакет с помощью модифицированной кучи Фибоначчи, можно получить время $\mathcal{O}(n(\log_t C + \log t))$. Асимптотического минимума функция достигает, когда $\log t = \sqrt{\log C}$, при этом получаем $\mathcal{O}(n\sqrt{\log C})$.

Для того, чтобы `ExtractMin` работал за $\sqrt{\log C}$, нужно удалять все вершины найденного подбакета разом. Добиться этого можно, если в куче Фибоначчи весу вершины будет соответствовать индекс подбакета, тогда нужно просто сделать в куче `ExtractMin` (не `ExtractMin`*). Но в этом случае нарушается потенциал, введённый при доказательстве времени работы модифицированной кучи Фибоначчи. Тем не менее, утверждается, что на асимтотику это не повлияет (доказательство остаётся читателям в качестве упражнения).

Таким образом, получили Дейкстру за $\mathcal{O}(E + V\sqrt{\log C})$.