Лекции по алгоритмам
Семестр 1

Лектор: Александр Куликов

Автор конспекта: Ольга Черникова

Собрано 6 сентября 2015 г. в 04:51

Содержание

1 Числа Фибоначчи **2**
- 1.1 Экспоненциальный рекурсивный алгоритм . . . 2
- 1.2 Полиномиальный алгоритм . . . 2

2 Скорость роста функций **3**
- 2.1 $\mathcal{O}$ — символика . . . 3
- 2.2 Рост функций . . . 3

3 Простейшие структуры данных **3**
- 3.1 Стек . . . 3
- 3.2 Стек с поддержкой максимума . . . 4
- 3.3 Очередь . . . 4
- 3.4 дек . . . 4
- 3.5 Списки . . . 4
- 3.6 Деревья . . . 4

4 Расширяющийся массив **4**
- 4.1 Метод потенциалов . . . 5

5 Сортировки **5**
- 5.1 Квадратичные сортировки . . . 5
- 5.2 Подсчетом . . . 6

6 Нижняя оценка $\Omega(n \log n)$ для сортировки **6**
- 6.1 стабильная сортировка . . . 6
- 6.2 цифровая сортировка . . . 6

7 Сортировка слиянием **6**
- 7.1 с рекурсией . . . 6
- 7.2 без рекурсии . . . 7

8 Быстрая сортировка **7**
8.1 Среднее время работы . . . 8
8.2 IntroSort . . . 8
8.3 Создание небольшой глубины рекурсии . . . 8
8.4 QuickSort3 . . . 9

9 Куча **9**
9.1 heap sort . . . 9
9.2 Частичная сортировка . . . 9
9.3 очередь с приоритетами(priority queue) . . . 9
9.4 Двоичная куча . . . 9
9.5 Построение кучи за линейное время . . . 10
9.6 k-ичная куча . . . 10

10 Порядковая статистика **10**

11 Динамическое программирование **11**
11.1 Динамика вперед/назад/ленивая . . . 11
11.1.1 Назад . . . 11
11.1.2 Вперед . . . 11
11.1.3 Ленивая . . . 11
11.2 Максимальная возрастающая подпоследовательность . . . 12
11.3 Рюкзак . . . 12
11.3.1 Рюкзак. Восстановление ответа . . . 12
11.4 Оптимальная триангуляция . . . 13
11.5 Независимые множества в деревьях . . . 14

12 Алгоритм Хиршберга нахождение расстояния редактирования(Расстояние по Левенштейну) **14**

13 Динамика на подмножествах **15**
13.1 Задача о гамильтоновом цикле . . . 15

14 Поиск в глубину **15**
14.1 Топологическая сортировка . . . 16

15 Выделение компонент сильной связности **16**

16 Поиск в ширину **16**

17 Дейкстра **17**
17.1 оценка времени работы . . . 17

18 алгоритм Форда-Беллмана **17**
18.1 отрицательный цикл . . . 18

19 Кратчайшие пути в ациклических ориентированных графах **18**

20 алгоритм Флойда **18**

21 алгоритм Джонсона **18**

22 Система непересекающихся множеств **19**

23 СНМ за log^*n **19**

24 Алгоритм Прима **19**

25 Алгоритм Крускала **20**

26 Коды Хаффмена **20**

27 Выполнимость хорновской формулы **21**
27.1 Задача выполнимости (SAT) 21
27.2 Хорновская формула 21

28 Задача о покрытии множествами **21**

29 Деревья поиска **22**
29.1 АВЛ-дерево 22

1 Числа Фибоначчи

$0, 1, 1, 2, 3, 5, 8, 13 \ldots$ — последовательность Фибоначчи.

$F_k = F_{k-2} + F_{k-1}$

1.1 Экспоненциальный рекурсивный алгоритм

```
Fib(k):
    if (k == 0):
        return 0
    if (k == 1):
        return 1
    return Fib(k - 1) + Fib(k - 2)
```

$T(k) \geq T(k-1) + T(k-2) + 3 \geq F_k$

Скорость роста чисел Фибоначчи экспоненциальная.

$F_k \sim \Phi^k \sim 1,618^k$

$F_k \geq 2^{\frac{k}{2}}$ - для $k \geq 6$ (экспоненциальная скорость роста).

Доказательство:

Докажем по индукции.

База: k = 6, k = 7;

Переход: $k \to k+1$

$F_{k+1} = F_{k-1} + F_k \geq 2^{\frac{k-1}{2}} + 2^{\frac{k}{2}} = 2^{\frac{k-1}{2}} \left(1 + 2^{\frac{1}{2}}\right) \geq 2^{\frac{k+1}{2}}$

■

1.2 Полиномиальный алгоритм

```
F[0...k]
F[0] = 0
F[1] = 1
for i = 2 ... k
    F[i] = F[i - 1] + F[i - 2]
return F[k]
```

Арифметических операций - $\mathcal{O}(k)$

Битовых операций - $O(k^2)$

2 Скорость роста функций

2.1 $\mathcal{O}$ — символика

T(n) — время работы алгоритма.
T: $\mathbb{N}_0 \to \mathbb{N}_0$

1. T(n) = $\mathcal{O}(f(n))$, если $\exists c > 0 : \forall n T(n) \leq cf(n) (\leq)$
2. T(n) = $\Omega(f(n))$, если $f(n) = \mathcal{O}(T(n))$(" $\geq$ ")
3. T(n) = $\Theta(f(n)), T(n) = \mathcal{O}(f(n)) T(n) = \Omega(f(n))(==)$
4. T(n) = $o(f(n)), \frac{T(n)}{f(n)}_{n \to \text{inf}} \to 0(<)$ (T растет медленнее f)

2.2 Рост функций

1. $n^2 = o(n^3)$
   $\sqrt{(n)} = o(n)$
   $n^a = o(n^b)$ если a < b
   $3n^3 + 5n^2 + n + 20 = \mathcal{O}(n^3)$
2. $(\log_2 n)^{20} = o(n^{\frac{1}{10}})$
   $(\log_2 n)^a$(полилогарифм) $= o(n^b)$
3. $n^{10} = o(2^n)$ (экспонента)
   полилогорифм < многочлен < экспонента
   $\log n < \log^2 n < \sqrt{(n)} < \sqrt{(n)} \log n < n^{\frac{2}{3}} < n < n \log n < n^2 < n^3 < 2^n < 3^n$
   $n! \sim (\frac{n}{e})^n$
   $\log(n!) \sim n \log n$

3 Простейшие структуры данных

3.1 Стек

Стек: |ooooooo|→ _ _ _ _ _ _ _ _ _ _ _ _ FILO (stack) $\mathcal{O}(1)$ — время работы.
Pop()
Push(x)
IsEmpty()

3.2 Стек с поддержкой максимума

Храним два стека.

В первом элементы, во втором максимум на префиксе.

[3, 2, 4, 5 →] (стек)

[3, 3, 4, 5 →] (стек с максимумом)

Max()

3.3 Очередь

Очередь:_ _ →|ooooooo|→ _ _ FIFO (queue)

Push_back(x)

Pop_front

Очередь с помощью двух стеков.

→(1)] [(2) →

Кладем элемент в первый стек

Достаем элементы из второго стека.

Если во (2) стеке кончились элементы перекладываем все элементы из первого стека.

Суммарное время работы $\mathcal{O}(N)$ (каждый элемент перекладываем три раза)

3.4 дек

Указатель на начало, указатель на конец. Закольцованный массив.

3.5 Списки

$\circ \to \circ \to \circ \to$

$\rightleftharpoons \circ \rightleftharpoons_{prev} \circ \rightleftharpoons^{next} \circ \rightleftharpoons \circ$

$\downarrow$ удаление элемента

$\rightleftharpoons \circ \rightleftharpoons \circ \rightleftharpoons \circ$

3.6 Деревья

Бинарное дерево(не более двух детей)

Можно у каждой вершины хранить две ссылки — левый ребенок и правый сосед

Почти полное бинарное дерево

[2i] ← [i] →[2i +1]

Вместо дерева можно хранить в массиве [1, 2, 3, 4 ...

4 Расширяющийся массив

Когда место заканчивается, создаем новый массив большего размера и копируем туда элементов.

Осталось понять, насколько надо увеличивать массив.

1. адетивная схема реалокации $(l \to l + 10)$

   $c + 2c + 3c + \ldots + n = c(1 + 2 + \ldots + \frac{n}{c}) = \Theta(n^2)$

   (не подходит)

2. мультипликативна схема$(l \to 1,5l)$

   $c + c^2 + c^3 + \ldots + n = \Theta(n)$

4.1 Метод потенциалов

$c_1 + c_2 + \ldots + c_n \leq n * \max_{1 \leq i \leq n} c_i$

c_i $---$ ()

$c'_i = c_i + \delta\Phi_i = c_i + \Phi_i - \Phi_{i-1}$ — учетная стоимость.

$\sum_{i=1}^{n} c'_i = \sum_{i=1}^{n} c_i + (\Phi_n - \Phi_0)$

l - длина, k - количество занятых ячеек

$\Phi = 2k - l \geq 0$

1. добавление без перевыделения

   $\Phi_{i-1} = 2k - l$

   $\Phi_i = 2k - l + 2$

   $c'_i = c_i + \delta\Phi = 3$

2. с перевыделением

   $c'_i = c_i + \delta\Phi_i = (k + 1) + (2(k + 1) - 2k) - (2k - k) = k + 1 + 2k + 2 - 2k - 2k + k = 3$

   $3n = \sum c_i + (\Phi_n - \Phi_0) \Rightarrow \sum c_i \sim \mathcal{O}(n)$

5 Сортировки

A[] A'[]

$A'[1] \geq A'[2] \geq A'[3] \geq A'[4] \geq \ldots \geq A'[n]$

5.1 Квадратичные сортировки

Время работы $\mathcal{O}(n^2)$

Просты в реализации.

Маленькая константа, быстро работают при маленьком размере массива.

5.2 Подсчетом

A[] — исходный массив
B[2, 2, 1] — количество элементов данного вида
C[0, 2, 4] — с какого места надо начинать записывать.

6 Нижняя оценка $\Omega(n \log n)$ для сортировки

.

Построим дерево вопросов и возможных итоговых ответов.
Ответы - листья дерева.
Время работы - глубина дерева
d - глубина.
Количество листьев $\leq 2^d$
глубина = $\Omega(\log_2$(количество листьев))
$\log_2(n!) = \Theta(n \log n)$
В среднем случае оценка тоже верна.

6.1 стабильная сортировка

Если a == b и a было раньше b изначально, то и в отсортированном массиве a будет идти раньше.

Стабильные сортировки: подсчетом, вставками, слиянием.

6.2 цифровая сортировка

сортируем сначала по последней цифре, потом по предпоследний и т.д.

Результат будет корректный, поскольку сортировка подсчетом стабильная.

Время работы $\mathcal{O}(d * n)$, d - длина, n - количество элементов.

7 Сортировка слиянием

7.1 с рекурсией

Метод "разделяй и властвуй"

```
MergeSort(A, l, r)
    if r-l <= 30
        InsertSort(A, l, r)
    m = (l + r)/2
    MergeSort(A, l, m)
    MergeSort(A, m + 1, r)
```

```
    Merge(A, l, m, r)
```

$T(n) \leq 2T(n/2) + \mathcal{O}(n) \Rightarrow T(n) = \mathcal{O}(n \log n)$

7.2 без рекурсии

```
Q = Queue()
for i = 1 to n:
    Q.push_back({A[i]})
while Q.size() > 1
    A1.PopFront()
    A2.PopFront()
    B=Merge(A1, A2)
    Q.push_back(B)
```

8 Быстрая сортировка

```
QuickSort(A, l, r):
    m=Partition(A, l, r)
    QuickSort(A, l, m - 1)
    QuickSort(A, m + 1, r)

Partition(A, l, r):
    x = A[l]
    j = l;
    for i = l + 1 to r:
        if(A[i] <= x)
            swap(A, i, j + 1)
            j = j + 1
    swap(A, l, j)
    return j
```

8.1 Среднее время работы

Теорема 8.1. Математическое ожидание времени работы QuickSort есть $\mathcal{O}(n \log n)$. В худшем случае $-\mathcal{O}(n^2)$.

Доказательство: Время работы $\sim$ количество сравнений $= \Theta$(количества сравнений)

E(время работаты) $= \Theta(E())$

E(количества сравнений) $= E \sum_{1 \le i < j \le n} x_{ij} = \sum_{1 \le i < j \le n} Ex_{ij} = \sum(P(x_{ij} = 1)) = \sum \frac{2}{j-i+1} = \Theta(n \log n)$

$$x_{ij} = \begin{cases} 1, & \text{если произошло сравнение A'[i] и A'[j], A' — отсортированный массив;} \\ 0, & \text{иначе} \end{cases}$$

Лемма: $P(x_{ij} = 1) = \frac{2}{j-i+1}$

Доказательство: $P(x_{ij} = 1) = P(A'[i \dots j]A'[i]A'[j])$

Утверждение $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \Theta \log n$

$\frac{\log n}{2} \le \frac{1}{2} + \frac{1}{4} + \frac{1}{4} \ldots + \frac{1}{n} \le 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \le 1 + \frac{1}{2} + \frac{1}{2} + \ldots + \frac{1}{n} \le \log n + 1 = \Theta \log n$

8.2 IntroSort

Если работает дольше $c \log n$ прерываем qsort и сортируем heap sort

8.3 Создание небольшой глубины рекурсии

```
QuickSort(A, l, r):
    m=Partition(A, l, r)
    QuickSort(A, l, m - 1)
    QuickSort(A, m + 1, r)  -- хвостовая рекурсия, tail recursion
```

Для маленькой глубины рекурсии.

```
QuickSort(A, l, r):
    while l < r
        m = Partition(A, l, r)
        идем в меньший
            QuickSort(A, l, r - 1)
            l = m + 1
```

Глубина не более $\log n$.

8.4 QuickSort3

Делим массив на 3 части. Элементы, которые меньше нашего, которые равны нашему и которые больше нашего.

9 Куча

9.1 heap sort

Строим кучу за линейное время, после этого вытаскиваем элементы.

9.2 Частичная сортировка

Хотим узнать в отсортированном порядке первые k элементов и что бы остальные шли както.

Строим кучу за линейное время и вытаскиваем первые k.

Время работы $\mathcal{O}(n) + \mathcal{O}(k \log(n))$

9.3 очередь с приоритетами(priority queue)

Getmin() $\mathcal{O}(1)$

Extractmin() $\mathcal{O}(h)$

Insert(x) $\mathcal{O}(h)$

Delete(ptr) $\mathcal{O}(h)$

Change Priority$\mathcal{O}(h)$

9.4 Двоичная куча

Двоичное дерево.

Для любого поддерева верно, что элемент в корне больше/меньше всех элементов в поддереве.

ShiftDown $\mathcal{O}(h)$

Просеивание вниз. Если в корне не верный элемент, а для обоих поддеревьев куча верная. Хотим построить правильную кучу.

Для этого выбираем min/max из вершины и детей. Меняем корень и меньшего местами и идем в меньшего ребенка.

```
Extractmin()
    swap(root, с последним элементом)
    SiftDown()
    return последний элемент
```

SiftUp() $\mathcal{O}(h)$
Insert(x)
добавляем в конец.
SiftUp();
Delete()
меняем с последним элементом просеиваем вниз.
храним дерево в массиве

9.5 Построение кучи за линейное время

Сначала построим дерево из элементов массива не обращая внимание на из порядок, потом для всех элементов, у которых есть потомок вызовем просеивание вниз. Просеивание вниз будем вызывать в порядке от листьев к корню.

$n = 1 + 2 + 4 + \ldots + 2^k$

Время работы $2^k + 2 * 2^{k-1} + 3 * 2^{k-2} + \ldots + k = n + n/2 + n/4 + \ldots + 1 = \Theta(n)$

9.6 k-ичная куча

У каждого элемента до k детей.

SiftDown - $\mathcal{O}(d \log_d n)$

SiftUp - $\mathcal{O}(\log_d n)$

10 Порядковая статистика

Хотим узнать, кто стоит на k-ой позиции в отсортированном массиве

$E(T(n)) = \mathcal{O}(n)$

$T(n) \leq T_o(n \to \frac{3n}{4}) + T(\frac{3n}{4})$

$E(n) \leq E(n \to \frac{3n}{4}) + E(\frac{3n}{4})$

Лемма: среднее количество подбрасывания честной монетки до первого выкидывания орла = 2.

Доказательство:

$E = \sum_{i=1}^{\inf} i * P[i] = \sum_{i=1}^{\inf} i \frac{1}{2^i} = 2$

■

$E(n) \leq E(\frac{3n}{4}) + \mathcal{O}(n) \Rightarrow E(n) = \mathcal{O}(n)$

$E(n) \leq cn + E(\frac{3n}{4}) \leq cn + c\frac{3n}{4} + E(\frac{9n}{16}) = cn(1 + \frac{3}{4} + \frac{9}{16})$убывающая геометрическая последовательность $\Rightarrow E(n) = \mathcal{O}(n)$

11 Динамическое программирование

11.1 Динамика вперед/назад/ленивая

$x \to x + 1$

$x \to 2x$

$x \to 3x$

Найти минимальное количество ходов, чтобы получить нужное число.

11.1.1 Назад

$$d[x] = \text{сколько нужно ходов, что бы получить x.}$$

$$ans = d[n]$$

$$d[1] = 0$$

$$d[x] = 1 + min \begin{cases} d[x-1] \\ d[x/2], & \text{если кратно 2} \\ d[x/3], & \text{если кратно 3} \end{cases}$$

11.1.2 Вперед

```
d[] = inf
d[1] = 0
for x = 1 to n
    relax(x + 1, d[x] +1)
    relax(2x, d[x] +1)
    relax(3x, d[x] + 1)
```

11.1.3 Ленивая

```
int f(int x)
    if(marked[x])
        return d[x]
    marked[x] = 1
    d[x] = f(x - 1)
    if(x mod 2 = 0)
        d[x] = min(d[x], f(x/2))
    if(x mod 3 = 0)
        d[x] = min(d[x], f(x/3))
    return ++d[x]
```

Бонусы ленивой динамики:

1. перебирает только то, что нужно.
2. не думаем, в каком порядке надо перебирать вершины.

11.2 Максимальная возрастающая подпоследовательность

$A[1 \dots n]$ $1 \le i_1 < i_2 < i_3 < \dots < i_k \le n$ $A[i_1] < A[i_2] < A[i_3] < \dots < A[i_k]$

$D[1 \dots n]$ D[i]= длина максимальной возрастающий подпоследовательности A, заканчивая в A[i].

$D[i] = \max_{i \le j \le i-1}[D[j] : A[j] < A[i]] + 1$

11.3 Рюкзак

Есть предметы определенного веса и рюкзак вместимостью W. Нужно поместить как можно больше предметов.

$\sum_i a_i \le W$
$\sum_i a_i \to max$
назад

```
is[0, 0] = 1
for i = 1 to n
    for w = 0 to W
        is[i, w] = is[i - 1, w] or is[i - 1, w - a[i]]
ans = is[n, w]
```

Можно хранить только два массива.

11.3.1 Рюкзак. Восстановление ответа

```
is[0, 0] = 1
for i = 1 to n
    for w = 0 to W
        is[i, w] = is[i - 1, w]
        p[i, w] = 0
        if (w >= a[i] and is[i - 1, w - a[i]])
            is[i, w] = 1
            p[i, w] = 1

if (is[n, W])
```

```
    for i = n to 1
        if(p[i, w] = 1)
            print(a[i])
            w -= a[i]
```

Уменьшим количество памяти. Динамика вперед

```
is[0] = 1
for i = 0 to n - 1
    for w = W to 0
        if is[w]
            is[w + a[i]] = 1
ans = is[w]
```

```
is[0] = 1
for i = 0 to n - 1
    for w = W to 0
        if is[w] and !is[w + a[i]](важное условие, тест 1 2 3)
            is[w + a[i]] = 1
            p[w + a[i]] = 1
if (is[w])
    while(w > 0)
        i = p[w]
        print(a[i])
        w -= a[i]
```

11.4 Оптимальная триангуляция

(картинка)

```
D[i, j] = первая и последнее вершина, которая задает многоугольник
for i = 1 to n - 2:
    D[i, i + 2] = 0
for l = 3 to n
    for i = 1 to n - l
```

```
j = i + l
D[i, j] = min_{i<k<j}(D[i, k] + D[k, j] + d(i, k) + d(k, j))
```

11.5 Независимые множества в деревьях

(картинка)

$I \subseteq V(G)$ - независимо, если $\forall u, v, \in I : (u, v) \notin E(G)$

Взвешенное дерево.

$D[u]$ - ответ для дерева с корнем в u.

$D[u] = \max(w[u] + \sum D[\text{внуков}], \sum D[\text{сынов}])$

12 Алгоритм Хиршберга нахождение расстояния редактирования(Расстояние по Левенштейну)

x[n]

y[m]

Строчки, можем удалять, заменять элементы.

```
EditDistance(x[1..n], y[1..m])
D[n + 1, m + 1]
for i = 0 to n
    D[i, 0] = i
for j = 0 to m
    D[0, j] = j

for i = 1 to n
    for j = 1 to m
        D[i, j] = min(1 + D[i, j - 1], 1 + D[i - 1, j])
        if (x[i] == y[j])
            D[i][j] = min(D[i][j], D[i - 1][j - 1]);
```

Достаточно хранить только два последних слоя.

Памяти - $\mathcal{O}(n)$

Время - $\mathcal{O}(nm)$

Хотим найти ответ от 0 0 до n m.

```
Hirschbery(x, y)
    найдем k
    k = argmin_{1≤i≤m}(dist((0,0) → (i,n/2)) + dist((i,n/2) → (m,n)))
    Hirschberg(x[1...n/2], y[1..k])
    Hirschberg(x[n/2...n], y[k + 1, m])
```

$T(n,m) \leq \mathcal{O}(nm)+T(n/2,k)+T(n/2,m-k) \leq cnm+c(\frac{n}{2}k)+c\frac{n}{2}(m-k) \leq cnm \Rightarrow T(n,m) = \mathcal{O}(nm)$

13 Динамика на подмножествах

13.1 Задача о гамильтоновом цикле

Пройти по всем вершинам ровно по одному разу. $\mathcal{O}(2^n n^2)$

D[S, i] - оптимальный путь из 1 в i проходящий по вершинам из S по разу.

14 Поиск в глубину

Способы хранения графа:

1. Список смежности
2. Матрица смежности
3. Список ребер

```
Explore(v)
    visited[v] = true
    for (u, v) ∈ E
        if visited[u] == false
            Explore(u)

DFS()
    for v ∈ V
        visited[v] = 0
    for v ∈ V
        if (visited[v] = false)
            Explore(v)
            ++cc - количество компонент
```

Время работы $\mathcal{O}(V+E)$

14.1 Топологическая сортировка

Ациклический ориентированный графы DAG.

Лемма: у графа есть топологическая сортировка тогда и только тогда граф ацикл.

Доказательство:

У ацикл. графа всегда есть исток и сток.

Сток можем поставить в конец и из графа удалить. Таким образом ничего не испортится. (Доказательство, по количеству вершин)

15 Выделение компонент сильной связности

Отсортируем вершины топ сортом, не смотря на то, что топологической сортировки там может и не быть.

Будем перебирать вершины в порядке топологической сортировки и запускать dfs на инвентированном графе. То что выделит dfs и будет компонента сильной связности.

Доказательство корректности Вершины s и t лежат в одной компоненте тогда и только тогда, когда они в итоге в одном дереве dfs.

$\Rightarrow$

Если s и t взаимодостижимы, то в итоге они будут лежать в одном дереве обхода в глубину.(Из t запустим dfs и он дойдет до s)

$\Leftarrow$

1. Вершины s и t лежат в одном дереве, значит в инвентированном графе они обе достижимы из корня r.
2. r шло раньше в порядке топ сорта, чем s и t. Рассмотрим два случая.
   (a) в прямом графе s и t достижимы из r, тогда все ок.
   (b) хотя бы одна, пусть t из r не достижима. При этом известно, что r достижима из t, но тогда бы r шло позже t по построению. Противоречие.

16 Поиск в ширину

Невзвешенный граф.

```
BFS(s)
    dist[1..N] = inf
    dist[s] = 0
    Q.pushBack(s)
    while(Q.size() > 0)
        u = Q.popFront()
        for (u, v) ∈ E
```

```
            if dist[v] = inf
                dist[v] = dist[u] + 1
                prev[v] = u
                Q.pushBack(v)
```

17 Дейкстра

```
Dijkstra(s)
    dist[1..N] = inf
    prev[1..N]
    dist[s] = 0
    while(Q.size() > 0)
        u = Q.ExstractMin()
        for (u, v) ∈ E
            if v ∈ Q and dist[v] > dist[u] + w(u, v)
                dist[v] = dist[u] + w(u, v)
                prev[v] = u
                меняем очередь
```

17.1 оценка времени работы

Время работы	k-ичная куча	двоичная куча	массив
V*T(Insert)	$\mathcal{O}(v \log_k v)$	$\mathcal{O}(v \log v)$	$\mathcal{O}(v)$
V*T(Extract Min)	$\mathcal{O}(vk \log_k v)$	$\mathcal{O}(v \log v)$	$\mathcal{O}(v^2)$
E*T(Change Prio)	$\mathcal{O}(E \log_k v)$	$\mathcal{O}(E \log v)$	$\mathcal{O}(E)$

18 алгоритм Форда-Беллмана

```
FordBelman(G, s)
    инициальзация
        повторить v - 1 раз
            для всех (u, v) ∈ E
                update(u, v)
```

время работы $\mathcal{O}(VE)$

18.1 отрицательный цикл

Если на V-ой итерации что-то срелаксировалось,значит есть цикл отрицательного веса.

Доказательство

Если что-то срелаксировалось, то цикл точно есть.

Если есть отрицательный цикл $a_1, a_2, a_3, \ldots, a_n$.

Пусть ничего не срелаксировалось, тогда

$d[a_2] \leq d[a_1] + w_1$

$d[a_3] \leq d[a_2] + w_2$

...

$d[a_1] \leq d[a_n] + w_n$

$\Rightarrow 0 \leq \sum w_i$ Противоречие

19 Кратчайшие пути в ациклических ориентированных графах

Отсортируем вершины топ сортом.

Можем запускать динамику в данном порядке.

20 алгоритм Флойда

```
Floyd(G)
    for i = 1 to n
        for j = 1 to n
            D[i][j]= w(если есть такое ребро)
            D[i][j] = inf(иначе)
    for k = 1 to n
        .for i, j
            D[i][j] = min(D[i][j], D[i][k] + D[k][j])
```

21 алгоритм Джонсона

Деикстра не работает для ребер с отрицательными весами.

Введем потенциалы, тогда $w'[u, v] = w[u, v] + Ph[u] - Ph[v]$

Тогда $dist[s, f] = dist'[s, f] - Ph[s] + Ph[f]$

Осталось придумать потенциалы так,что бы веса были положительные.

Потенциал будет равен расстоянию от фиктивной вершины, с ребром во все вершины веса 0.

22 Система непересекающихся множеств

(картинка)

Хотим объединять множества и понимать, принадлежат ли вершины одному множеству.

MakeSet(x)

Find(x)

Union(x, y)

Для каждой вершины будем хранить цвет множества, где лежит вершина.

При сливание двух множеств перекрашиваем полностью меньшее множество.

Каждая вершина, когда перекрашивается попадает в множества в два раза большее. $\Rightarrow$ время работы$\mathcal{O}(n \log n)$

23 СНМ за $log^* n$

Будем использовать две эвристики: сжатие путей и ранговая эвристика.

Для каждой вершинки храним ссылку на представителя.

Представитель множества - корень.

Ранговая эвристика - для каждой вершины храним высоту поддерева и меньшее поддерево привешиваем к большему.

Корневая - каждый раз, когда получаем запрос об узнавание цвета подвешиваем все вершины на пути к корню.

Теорема: суммарное время m операций СНМ из которых n $\leq$ m - это MakeSet, есть $\mathcal{O}(m \log^* n)$

Доказательство:

Суммарное время Find $\sim$ количество переходов от вершины к родителю = количество переходов к корню + количество переходов к родителю из того же отрезка + количества переходов к родителю из другого отрезка = ($\mathcal{O}(m)$ - переходов к корню) + ($\mathcal{O}(n \log^* n)$ - переходов к родителю внутри отрезка) + ($\mathcal{O}(m \log^* n)$ - переходов к родителю из другого отрезка)

[1...1][2...2][3...4][5...16][17...2^{16}][$2^{16}+1$ ... $2^{2^{16}}$]

[k + 1...2^k] В каждом отрезке $\log^*$ имеет одинаковое значение.

Переходов внутри одного отрезка [k + 1...2^k] - $\mathcal{O}(n)$

Вершин ранга $k+1 \leq \frac{n}{2^{k+1}}$ Вершин ранга $k+2 \leq \frac{n}{2^{k+2}}$ ...

$\Rightarrow$ количество вершин попавших в отрезок $\leq \frac{n}{2^k}$

Для каждой вершины в отрезке мы переходим к ее предку внутри отрезка не более 2^k раз(после каждого перехода ранг предка увеличивается).

■

24 Алгоритм Прима

Дан неориентированный взвешенный связный граф. Хотим выбрать остовное дерево минимального веса.

Лемма о разрезе: $T \subseteq E$ - часть некоторого МОД $S \subseteq V$ - такой разрез, что T не пересекает разрез, e минимальное ребро графа G, пересекающее разрез. $\Rightarrow T \cup \{e\}$ - часть некоторого(возможно другого) МОД.

Доказательство:

Рассмотрим дерево и добавленное нами ребро.

Если добавленное нами ребро принадлежит дереву,значит все хорошо.

Если не принадлежит, то образовался цикл. Существует ребро которое входит из V§в S и принадлежит циклу. Это ребро не меньше ребра e(иначе мы бы выбрали его), значит можем его выкинуть и добавить e, от этого вес остовного дерева только улучшится.

■

Алгоритм Прима

Изначальна у нас есть компонента из одной вершины. Пытаемся ее расширить. Для этого для данной компоненты поддерживаем вес ребер из S до V/S.

Алгоритм пишется как Деикстра с заменой суммы на минимум.

Время работы $\mathcal{O}(E \log V)$

25 Алгоритм Крускала

1. отсортировать ребра по весу
2. перебираем ребра в порядке увеличения веса
3. если ребро не создает цикла, добавляем его

```
Kruskal(G)
sort()
T= []; for v ∈ V:
    MakeSet()
for (u, v) в порядке сортировки по увеличению веса
    if(Find(u) != Find(v))
        T = T ∪ (u, v)
        Union(u,v)
```

Для работы с Find и Union используется СНМ.

Время работы $\mathcal{O}(ElogE) = \mathcal{O}(E \log E) + \mathcal{O}(V) + \mathcal{O}(E \log^*(V))$

26 Коды Хаффмена

У нас есть текст,хотим его записать в бинарном виде.

Если для каждой буквы выделять одинаковое количество бит, результат может получится далеко не оптимальным. Например одна буква встретилась один раз, вторая повторяется очень много.

Посчитаем частоту для каждой буквы.

Построим бинарное дерево и в листьях запишем буквы.

Для каждой вершины(кроме корня) запишем сумму весов в поддереве. Тогда длина записи = сумме весов всех вершин.

Теперь поймем, как построить оптимальное дерево.

Для этого отсортируем все частоты в порядке увеличения.

Существует оптимальное дерево, в котором две самые редкие вершины братья. Тогда объединим их и повторим алгоритм.

Время построения дерева $\mathcal{O}(n \log n)$

27 Выполнимость хорновской формулы

27.1 Задача выполнимости (SAT)

$x_1, x_2, \ldots, x_n \in \{0, 1\} = \{false, true\}$

Формула в КНФ. Проверяется выполнимость за $\mathcal{O}(m2^n)$

27.2 Хорновская формула

В каждом дизъюнкте не более одной положительной переменной

$x_1, x_2, \ldots, x_n = 0$
`while (есть неверный дизъюнкт содержащий положительную переменную` x_i`)`
    x_i `= 1`
`если формула выполнима`
    `вернуть` $x_1, x_2, \ldots, x_n$
`иначе`
    `вернуть NO`

Время работы $\mathcal{O}(N)$

28 Задача о покрытии множествами

У нас есть набор множеств.

Требуется выбрать минимальное количество множеств, так, чтобы покрыть все элементы.

Жадный алгоритм: на каждом шаге выбираем множество, которое покрывает максимальное количество из еще не покрытых элементов.

Лемма: Жадный алгоритм является ln n приближенным. (n - количество элементов, которые надо покрыть)

Доказательство: n_i - количество непокрытых элементов после шага i.

Хотим доказать, что n_i - убывает быстро.

k - размер оптимального покрытия.

$i-1 \to i$

$n_i \le n_{i-1}(1-\frac{1}{k})$ (на каждом шаге мы должны уметь покрывать хотя бы 1/k часть от оставшихся.Такой элемент существует, поскольку n_{i-1} мы можем покрыть k множествами).

$n_i \le n_{i-1}(1-\frac{1}{k}) \le n_{i-2}(1-\frac{1}{k})^2 \le \ldots \le n(1-\frac{1}{k})^i < 1 \Leftrightarrow (1-\frac{1}{k})^i < \frac{1}{n}$

$1+x < e^x$

$(1-\frac{1}{k})^i < (e^{-\frac{1}{k}}) = e^{-\frac{i}{k}}$

$e^{-\frac{i}{k}} = \frac{1}{n}$

$i = k \ln n$

■

29 Деревья поиска

В левом поддереве все элементы меньше корня, в правом - больше.

Это верно для любого поддерева.

Печать элементов: Распечатать левое поддерево, распечатать нас, распечатать правое поддерево.

Вывод за $\mathcal{O}(N) \Rightarrow$

Find/lowerBound/UpperBound $\mathcal{O}(h)$

Insert $\mathcal{O}(h)$

Min/Max $\mathcal{O}(h)$

Succ/Prev $\mathcal{O}(h)$

Delete $\mathcal{O}(h)$

Удаление

легко удалить лист, вершину, у которой один ребенок.

Меняем местами нас с самым правым в левом поддереве и удаляем.

29.1 АВЛ-дерево

Для любой вершины высота ее поддеревьев различается не более чем на 1.

Лемма Высота АВЛ-дерева = $\mathcal{O}(\log n)$

Доказательство В дереве высоты h $\ge c^h$ элементов.

база h = 0

переход: $h-1 \to h$

количество $\ge 1 + c^{h-1} + c^{h-2} \ge c^h \Leftarrow c+1 \ge c^2$

КОНЕЦ