

Алгоритмы и структуры данных (ФПМИ базовый поток 2020-2021)

Объединение базового потока

January 2021

1 Алгоритм: анализ алгоритма, сложность работы по времени и по памяти, асимптотика (O-большое), RAM-модель вычислений.

1.1

Сложность алгоритма можно оценивать по параметру $M(n)$ — это размер дополнительной памяти и $T(n)$ — время выполнения, где n — характеристический размер входа.

Def. Обозначение O-большое:

$f(n) = O(g(n)) \Leftrightarrow f(n) \leq Cg(n)$, где C — некая константа, для любых $n > N$.

1.2 RAM-модель вычислений (Random Access Memory)

У нас есть юнит, который читает программу и ее выполняет, есть память, чей размер задан (соответственно она может быть переполнена).

- к любой ячейке памяти мы имеем доступ за единицу (чтение, запись — $O(1)$);
- предполагается, что у юнита константное количество регистров, с которыми юнит выполняет арифметические действия (арифметика за $O(1)$).

2 Сортировки. Стабильные сортировки. Сортировки без дополнительной памяти. Сортировки “на месте” (in-place). Быстрая сортировка (Хоара, Ломуто). Вывод среднего времени работы быстрой сортировки.

2.1

Def. Стабильная сортировка — такая сортировка, что в ней элементы, которые эквивалентны по сравнению, не перемещаются относительно друг друга ($i < j, a_i = a_j \Rightarrow \pi(i) < \pi(j)$). Пример:

2, 1', 3, 1'' \rightarrow 1', 1'', 2, 3 — стабильная сортировка

2, 1', 3, 1'' \rightarrow 1'', 1', 2, 3 — не стабильная.

Def. Сортировка без дополнительной памяти означает, что дополнительные расходы памяти составляют $O(1)$ (на всякие swap'ы, сохранение какой-то промежуточной информации и т.п.)

Def. Сортировка in-place означает, что данные сортируются “на месте”, без привлечения дополнительных хранилищ и структур данных (то есть элементы перемещаются только внутри выданного массива). Пример: алгоритм QuickSort - он inplace, так как элементы меняются местами только друг с другом, но он требует доп.память размера $O(\log n)$ (стек рекурсии).

2.2 Быстрая сортировка.

2.2.1 Идея алгоритма

1. Разделим массив на 2 части таким образом, что элементы в левой части \leq элементов в правой части
2. Применим эту процедуру рекурсивно к левой и правой части

2.2.2 Partition - Ломуто

Разделим массив A. Выберем разделяющий элемент - пивот. Пусть пивот лежит в конце массива.

1. Установим 2 указателя: i,j в начало массива
2. Двигаем j вправо, пока не встретим элемент \leq пивота
3. Если встретили элемент \leq пивота, то меняем A[i] и A[j] местами(если $i \neq j$), двигаем i и j вправо
4. Меняем A[i] и A[n-1] (пивот)

2.2.3 Partition - Хоара

Разделим массив A. Выберем разделяющий элемент - пивот. Пусть пивот лежит в середине. Это разбиение эффективнее, так как происходит меньше swap'ов.

1. Установим 2 указателя: i,j в начало массива и в конец
2. Двигаем i вправо, пока не встретим элемент \geq пивота
3. j влево, пока не встретим элемент \leq пивота
4. Если $i \geq j$, то возвращаем j (последний элемент меньше пивота), иначе свопаем a[i], a[j]
5. Продолжаем цикл

2.2.4 Анализ быстрой сортировки

- Если Partition всегда делит пополам, то

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

Следовательно, $T(n) = O(n \log(n))$

- Если массив упорядочен, pivot = $A[n - 1]$, то массив делится в отношении n-1 : 0.

$$T(n) \leq T(n - 1) + cn \leq T(n - 2) + c(n + n - 1)$$

Следовательно, $T(n) = O(n^2)$

Утверждение. В среднем по обоим распределениям, при условии различности элементов $T(n) = O(n \log(n))$

Доказательство:

Пусть X полное количество сравнений элементов с опорным за время работы сортировки. Нам необходимо вычислить полное количество сравнений.

Переименуем элементы массива как $z_1 \dots z_n$, где z_1 наименьший по порядку элемент. Также введем множество $Z_{ij} = \{z_i, z_{i+1} \dots z_j\}$.

Заметим, что сравнение каждой пары элементов происходит не больше одного раза, так как элемент сравнивается с опорным, а опорный элемент после разбиения больше не будет участвовать в сравнении.

Поскольку каждая пара элементов сравнивается не более одного раза, полное количество сравнений выражается как

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}, \text{ где } X_{ij} = 1 \text{ если произошло сравнение } z_i \text{ и } z_j \text{ и } X_{ij} = 0, \text{ если сравнения не произошло.}$$

Применим к обеим частям равенства операцию вычисления матожидания и воспользовавшись ее линейностью получим

$$E[X] = E \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n Pr\{z_i \text{ сравнивается с } z_j\}$$

Осталось вычислить величину $Pr\{z_i \text{ сравнивается с } z_j\}$ вероятность того, что z_i сравнивается с z_j . Поскольку предполагается, что все элементы в массиве различны, то при выборе x в качестве опорного элемента впоследствии не будут сравниваться никакие z_i и z_j для которых $z_i < x < z_j$. С другой стороны, если z_i выбран в качестве опорного, то он будет сравниваться с каждым элементом Z_{ij} кроме себя самого. Таким образом элементы z_i и z_j сравниваются тогда и только тогда когда первым в множестве Z_{ij} опорным элементом был выбран один из них.

$Pr\{z_i \text{ сравнивается с } z_j\} = Pr\{\text{первым опорным элементом был } z_i \text{ или } z_j\} = Pr\{\text{первым опорным элементом был } z_i\} + Pr\{\text{первым опорным элементом был } z_j\} =$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$$

Матожидание времени работы быстрой сортировки будет $O(n \log n)$.

3 Сортировка слиянием. Сортировка слиянием без дополнительной памяти (стабильный вариант за $\mathcal{O}(N \log^2 N)$). Сортировка слиянием без дополнительной памяти (нестабильный вариант за $\mathcal{O}(N \log N)$)

3.1 Определение

Def. Сортировка слиянием (англ. *Merge sort*) — алгоритм, выполняющий сортировку массивы засчёт слияния отсортированных подмассивов. Алгоритм использует $\mathcal{O}(N)$ дополнительной памяти и работает за $\mathcal{O}(N \log N)$ времени.

3.2 Классический вариант за $\mathcal{O}(N)$ доп. памяти

3.2.1 Принцип работы:

Алгоритм использует принцип «разделяй и властвуй»: задача разбивается на подзадачи меньшего размера, которые решаются по отдельности, после чего их решения комбинируются для получения решения исходной задачи. Конкретно процедуру сортировки слиянием можно описать следующим образом:

1. Если в рассматриваемом массиве один элемент, то он уже отсортирован — алгоритм завершает работу.
2. Иначе массив разбивается на две части, которые сортируются рекурсивно.
3. После сортировки двух частей массива к ним применяется процедура слияния, которая по двум отсортированным частям получает исходный отсортированный массив.

3.2.2 Процедура слияния(Merge):

У нас есть два отсортированных массива a и b . Нам надо получить из них же отсортированный массив, но уже с размером $|a|+|b|$. Для этого можно применить процедуру слияния. Эта процедура заключается в том, что мы сравниваем элементы массивов (начиная с начала) и меньший из них записываем в финальный. И затем, в массиве у которого оказался меньший элемент, переходим к следующему элементу и сравниваем теперь его. В конце, если один из массивов закончился, мы просто дописываем в финальный другой массив. После мы наш финальный массив записываем вместо двух исходных и получаем отсортированный участок.

Псевдокод:

```
def merge(A, B, C : int[n]):  
    i_a = i_b = 0  
    while i_a < A and i_b < B:  
        if A[i_a] ≤ B[i_b]:  
            C[i_c] = A[i_a]  
            i_a += 1, i_c += 1  
        else:  
            C[i_c] = B[i_b]  
            i_b += 1, i_c += 1  
    while i_a < |A|:  
        C[i_c] = A[i_a]  
        i_a += 1, i_c += 1  
    while i_b < |B|:  
        C[i_c] = B[i_b]  
        i_b += 1, i_c += 1
```

3.2.3 Рекурсивный алгоритм:

```
def mergeSort(A : int[n]; left, right : int):  
    if right - left ≤ 1  
        return  
    mid = (left + right) / 2  
    mergeSort(A, left, mid)
```

```
mergeSort(A, mid, right)
merge(A, left, mid, right)
```

3.2.4 Время работы

Чтобы оценить время работы этого алгоритма, составим рекуррентное соотношение. Пускай $T(N)$ — время сортировки массива длины N , тогда для сортировки слиянием справедливо $T(N) = 2T(N/2) + \mathcal{O}(N)$ — время, необходимое на то, чтобы слить два массива длины N .

Распишем это соотношение:

$$T(N) = 2T(N/2) + \mathcal{O}(N) = 4T(N/4) + \mathcal{O}(N) = \dots = T(1) + \log(N)\mathcal{O}(N).$$

$$\Rightarrow T(N) = \mathcal{O}(N\log N)$$

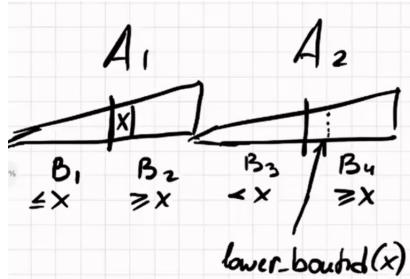
3.3 Стабильный вариант за $\mathcal{O}(N \log^2 N)$

3.3.1 Поменяется только процедура слияния(Merge → InplaceMerge):

```
def inplaceMerge(A1, A2 : int[n]):
    if |A1| == 0 or |A2| == 0:
        return
    if |A1| == 1 and |A2| == 1:
        if A1[0] > A2[0]:
            swap(A1[0], A2[0])
    m = |A1|
    B1 = A1[0, ..., m/2]
    B2 = A1[m/2, ..., m]
    x = B2[0]
    B3 = A2[m, ..., l_b(x)]
    B4 = A2[l_b(x), ..., n]
    rotate(B2, B3)
    inplaceMerge(B1, B3)
    inplaceMerge(B2, B4)
```

Комментарий:

Если $|A_1| < |A_2|$, то ищём центральный элемент x уже в правом массиве, а в левом берём $\text{upper_bound}(x)$.



Пояснительный рисунок 1.

3.3.2 Время работы

$$|B_1| \geq \frac{n}{4}$$

$$|B_2| \geq \frac{n}{4}$$

$$|B_1 + B_3| \geq n/4$$

$$\Rightarrow \text{правая часть: } |B_2 + B_4| \leq \frac{3n}{4}.$$

$$\text{Аналогично, левая часть: } |B_1 + B_3| \leq \frac{3n}{4}$$

То есть разделили $A = A_1 \cup A_2$ на 2 части по $\leq \frac{3n}{4}$, а значит изначальный размер уменьшился в $\frac{4}{3}$ раза. Поэтому делаем вывод, что рекурсия требует $\mathcal{O}(\log_{\frac{4}{3}}(N))$ памяти стека. Циклический сдвиг потребует ещё $\mathcal{O}(N)$.

Итого получим:

$$T(N) = T_{\text{Merge}}(N) \cdot \log(N) = \mathcal{O}(N \log^2(N))$$

3.4 Нестабильный вариант без дополнительной памяти

3.4.1 Алгоритм Inplace Merge Sort:

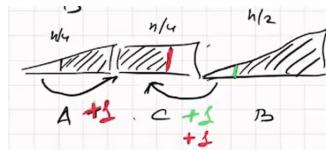
1. Отсортируем вторую половину массива, используя первую как дополнительную память.
2. Отсортируем первую четверть, используя вторую четверть.
3. Сольём первую четверть со второй половиной(обе сортированные), используя промежуточную(вторую) четверть.
4. Аналогично действуем с первой $\frac{1}{8}$, используя вторую $\frac{1}{8}$.
5. Далее по аналогии.

Замечание:

В процедуре merge(...) необходимо сделать замену:
 $C[i_c] = A[i_a] \rightarrow swap(C[i_c], A[i_a])$

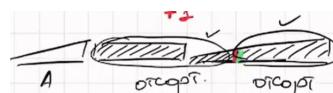
Корректность:

В процессе слияния конец промежуточного массива приближается к правому массиву тогда и только тогда, когда мы переносим объект из левого в промежуточный, в других случаях расстояние между промежуточным и правым не меняется:



Пояснительный рисунок 2.

Представим худший случай, когда промежуточный массив "догнал" правый. Но тогда получается, что их суммарная длина - есть уже отсортированный массив, причём все элементы из правого не меньше, чем элементы промежуточного, а значит алгоритм корректно выполняет слияние на каждом шаге, что свидетельствует о его корректности.



Пояснительный рисунок 3.

Время работы

Наибольший вклад вносят слагаемые, отвечающие за сортировку подмассивов и слияние:

$$\sum_{k=0}^{\log_2 N-1} \frac{N}{2^k} \log_2 \left(\frac{N}{2^k}\right) + \mathcal{O}(N) = \mathcal{O}\left(\sum_{k=0}^{\log_2 N-1} \frac{N}{2^k} \mathcal{O}(\log_2 N) + \mathcal{O}(N)\right)$$

Сумма ряда является суммой геометрической прогрессии:

$$\sum_{k=0}^{\log_2 N-1} \frac{1}{2^k} = \dots = 2 - \frac{4}{N}$$

$$T(N) = \mathcal{O}(N \cdot \mathcal{O}(\log_2 N)(2 - \frac{4}{N}) + \mathcal{O}(N))$$

$$\Rightarrow T(N) = \mathcal{O}(N \log N)$$

STL

В языке C++ для этого алгоритма существует готовая реализация `std::stable_sort`, которая работает за $\mathcal{O}(N \log N)$, если памяти, требуемой алгоритмом, достаточно и за $\mathcal{O}(N \log^2 N)$ иначе.

4 Хеш-таблицы. Коллизии. Разрешение коллизий методом цепочек. Анализ времени работы в случае простого равномерного хеширования. Универсальные семейства хеш-функций.

4.1

Пусть у нас есть \mathcal{K} — множество ключей. Хотим построить такую структуру данных, чтобы она могла выполнять быстро операции $\text{Insert}(\text{key})$, $\text{Erase}(\text{key})$, $\text{Find}(\text{key})$. Если $\mathcal{K} = 0, 1..n - 1$, n — достаточно мало, то мы можем использовать просто массив, где стоит метка, если элемент содержится и иначе.

Но при больших числах это затратно, поэтому введем функцию $h : k \rightarrow 0, 1, \dots, m - 1$, соответственно m — размер таблицы.

Получается по $A[h(x)]$ можем получить весь наш объект. Но если \mathcal{K} — большое, а m меньше $|\mathcal{K}|$, то возникает проблема — возможность коллизий. Рассмотрим способ решения этой проблемы — метод цепочек.

4.2

Def. Коллизия — ситуация когда $h(x) = h(y), x \neq y$.

Метод цепочек заключается в том, что если у нас произошла коллизия, то мы в ячейке $A[h(x)]$ имеет список, куда мы записываем все элементы, которые попадают в $A[h(x)]$. Соответственно алгоритм вставки — считаем $h(x)$, заходим в ячейку $A[h(x)]$, проходим по списку, если не находим такого элемента, то вставляем в конец x . (Остальное аналогично).

4.3

Проанализируем время работы метода цепочек, понятно, что это сильно зависит от функции. Например функция $h(x) \equiv 0$ очевидно плохая, т.к. все операции будут выполняться за линейное время.

Рассмотрим простое равномерное хеширование.

Def. Простое равномерное хеширование — когда h выбирается случайно и равномерно из множества $0..m - 1$. То есть мы можем считать, что значение $h(x)$ выбирается случайно.

Утверждение. Пусть $x \neq y \Rightarrow P(h(x) = h(y)) = \frac{1}{m}$, в случае равномерного хеширования.

Th. В случае равномерного хеширования время работы Insert , Erase , Find — $O(1 + \alpha)$, где $\alpha = \frac{n}{m}$ — load factor.

Доказательство: Пусть вставленные элементы это $\mathcal{X} = x_1 \dots x_n$.

По сути все эти операции работают за время Find , так как они заходят в ячейку за $O(1)$ там пробегают по всем элементам в ячейке и за единицу вставляют/удаляют элемент. Поэтому определим время работы Find . А его время работы пропорционально размеру одной ячейки + 1 (это константные операции см выше).

Пусть ищем элемент x в корзине длины \mathcal{L} . Посчитаем средний размер этой корзины.

$E(\mathcal{L}) = E(\sum_{i=1}^n I(h(x_i) = h(x))) = \sum_{i=1}^n EI(h(x_i) = h(x)) = \sum_{i=1}^n P(h(x_i) = h(x)) = \sum_{i=1}^n 1$, если $x = x_i; \frac{1}{m}$ иначе $= \frac{n-1}{m} + 1, x \in \mathcal{X}; \frac{n}{m}$, иначе $= O(\alpha)$.

Получается $O(\alpha + 1)$.

4.4

Но у нас появляется проблема равномерного хеширования, заключающаяся в том, что на практике его реализовать невозможно. Потому что простое равномерное хеширование заключается в случайной равномерной генерации хеш-значения для каждого ключа (идеальная с криптографической точки зрения хеш-функция). Допустим, умеем генерировать простой равномерный хеш. Он устроен так: приходит ключ, если он новый, то генерируем случайный хеш, если старый (ранее уже встречался), то берем значение, которое уже сгенерировали ранее. А теперь представьте, что мы таким образом захешировали 1000000 ключей. Приходит новый ключ и нам надо понять — встречался он ранее (входит ли он в тот список 1000000) или нет. То есть нужен быстрый поиск по ключам. Но погодите, ровно эту задачу мы и хотим решить с помощью хеш-функций. То есть, чтобы создать хеш-функцию, надо создать хеш-функцию (либо придумать другую структуру данных, которая умеет делать быстрый поиск, но, если мы такую придумаем, то зачем нам тогда хеши). Вот и получается «замкнутый круг». Да и вообще мощность множества всех возможных хеш-функций $|H| = |1..m - 1|^K >> 2^{64}$. Давайте тогда выделим подмножество H' , которые легко представить в ЭВМ, например $h(x) = ax^2$, так как нам достаточно сохранить a .

Def. H' — универсальное семейство хеш-функций если для любых $x, y \in \mathcal{K}, x \neq y, P(h(x) = h(y)) \leq \frac{1}{m}$, при

условии случайного выбора h из H' .

Понятно что для этого множества выполняется условие теоремы.

Рассмотрим пример универсального семейства: $K = \{0, 1 \dots p - 1\}$, где p - большое простое число. Заметим, что $K = Z_p$ — система вычетов по модулю p .

$h_{a,b}(x) = [(ax + b)\%p]\%m$, где $a \neq 0, a \in Z_p, b \in Z_p, m < p$.

Th. Это универсальное множество.

Доказательство: Зафиксируем $x, y \in K, x \neq y$. Докажем что $P(h(x) = h(y)) \leq \frac{1}{m}$ при случае (a, b).

Рассмотрим $x_1 = (ax + b)\%p, y_1 = (ay + b)\%p$

1. $x_1 \neq y_1: x_1 - y_1 = (ax + b - ay - b) = a(x - y) \neq 0(modp)$

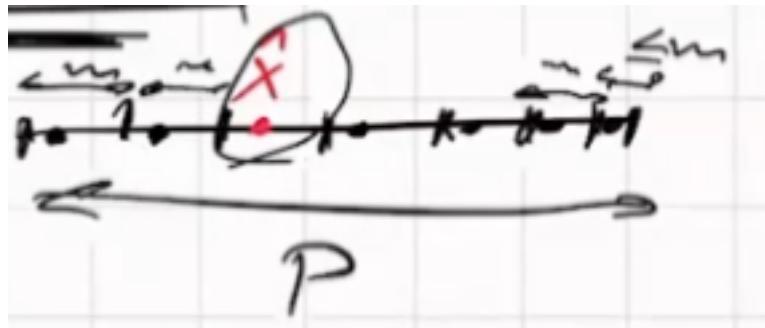
2. $(a, b) = p(p - 1)$

$(x_1, y_1) = p(p - 1)$

Покажем, что $(a, b) \rightarrow (x_1, y_1)$ — биекция.

3. Достаточно инъекции, то есть: для любых неравных x_1, y_1 существует единственная пара (a, b) , такая что $x_1 = (ax + b)\%p, y_1 = (ay + b)\%p$. Данная система имеет единственное решение(б/д). 4. Так как биекция, значит сгенерировать (a, b) — то же самое что и сгенерировать x_1, y_1 .

5. то есть $h(x) = x_1 \% m, h(y) = y_1 \% m$, ну а сколько таких пар, что $x_1 \% m = y_1 \% m$ их $\leq p(\lceil \frac{p}{m} \rceil - 1)$ — это потому что (см рисунок) мы выбираем у только в соседних отрезочках.



Пояснительный рисуночек 1.

$$\leq p(\frac{p+m-1}{m} - 1) = \frac{p(p-1)}{m}.$$

$$6. P(h(x) = h(y)) = P([(ax + b)\%p]\%m = [(ay + b)\%p]\%m) = \\ = P(x_1 \% m = y_1 \% m) \leq \frac{p(p-1)}{mp(p-1)} \leq \frac{1}{m}$$

5 Идеальное хеширование статического множества. Эффективный алгоритм построения структуры данных с константным (в худшем случае) доступом.

5.1 Определение

Def. Идеальная хеш-функция (англ. *perfect hash function*) — хеш-функция, которая без коллизий отображает различные элементы из множества объектов на множество ключей за $\mathcal{O}(1)$ времени в худшем случае.

5.2 Основная идея

Идеальное хеширование используется в задачах со статическим множеством ключей (т.е. после того, как все ключи сохранены в таблице, их множество никогда не изменяется) для обеспечения хорошей асимптотики даже в худшем случае. При этом мы можем дополнительно хотеть, чтобы размер таблицы зависел от количества ключей линейно.

В таком хешировании для доступа к данным потребуется лишь вычисление хеш-функций (одной или нескольких), что делает данный подход наибыстрейшим для доступа к статическим данным. Данная технология применяется в различных словарях и базах данных, в алгоритмах со статической (известной заранее) информацией.

Будем использовать двухуровневую схему хеширования с универсальным хешированием на каждом уровне:

5.2.1 Первый уровень

Используется тот же принцип, что и в случае хеширования с цепочками: n ключей хешируются в m ячеек с использованием хеш-функции $h(k) = ((a \cdot k + b) \bmod p) \bmod m$, случайно выбранной из семейства универсальных хеш-функций $H_{p,m}$, где p — простое число, превышающее m .

5.2.2 Второй уровень

На данном уровне вместо создания списка ключей будем использовать вторичную хеш-таблицу S_j , хранящую все ключи, хешированные функцией h в ячейку j , со своей функцией $h_j(k) = ((a_j \cdot k + b_j) \bmod p) \bmod m_j$, выбранной из множества H_{p,m_j} . Путем точного выбора хеш-функции h_j мы можем гарантировать отсутствие коллизий на этом уровне. Для этого требуется, чтобы размер m_j хеш-таблицы S_j был равен квадрату числа n_j ключей, хешированных функцией h в ячейку j .

Несмотря на квадратичную зависимость, ниже будет показано, что при корректном выборе хеш-функции первого уровня количество требуемой для хеш-таблицы памяти будет $\mathcal{O}(n)$.

5.3 Теоретическое обоснование

Теорема 5.1. Если n ключей сохраняются в хеш-таблице размером $m = n^2$ с использованием хеш-функции h , случайно выбранной из универсального множества хеш-функций, то математическое ожидание числа коллизий не превышает $\frac{1}{2}$.

Доказательство. Всего имеется C_n^2 пар ключей, которые могут вызвать коллизию. Если хеш-функция выбрана случайным образом из универсального семейства хеш-функций H , то для каждой пары вероятность возникновения коллизии $\leq \frac{1}{m}$. Пусть X — случайная величина, которая подсчитывает количество коллизий. Если $m = n^2$, то математическое ожидание числа коллизий равно $\mathbf{E}[X] = C_n^2 \cdot \frac{1}{n^2} = \frac{n^2 - n}{2} \cdot \frac{1}{n^2} < \frac{1}{2}$ ■

Теорема 5.2. Если мы сохраняем n ключей в хеш-таблице размеров $m = n$ с использованием хеш-функции h , выбираемой случайным образом из универсального множества хеш-функций, то $\mathbf{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$, где n_j — количество ключей, хешированных в ячейку j .

Доказательство. $\mathbf{E}\left[\sum_{j=0}^{m-1} n_j^2\right] = \mathbf{E}\left[\sum_{j=0}^{m-1} (n_j + 2C_{n_j}^2)\right] = \mathbf{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\mathbf{E}\left[\sum_{j=0}^{m-1} C_{n_j}^2\right] = \mathbf{E}[n] + 2\mathbf{E}\left[\sum_{j=0}^{m-1} C_{n_j}^2\right] = n + 2\mathbf{E}\left[\sum_{j=0}^{m-1} C_{n_j}^2\right]$

Первый переход в равенстве мы совершили благодаря формуле $a^2 = a + 2 \cdot C_a^2$. Далее мы воспользовались свойствами математического ожидания, в частности - линейности. Очевидно, что $\mathbf{E}\left[\sum_{j=0}^{m-1} C_{n_j}^2\right]$ - просто общее

количество коллизий, поэтому по свойству универсального хеширования математическое ожидание значения этой суммы не превышает $C_n^2 \cdot \frac{1}{m} = \frac{n(n-1)}{2m} = \frac{(n-1)}{2}$.

А так как $m = n$, то $\mathbf{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \leq n + 2 \cdot \frac{(n-1)}{2} = 2n - 1 < 2n$, ч.т.д. ■

Теперь выведем следствие из этой теоремы.

Теорема 5.3 (Следствие). *Если мы сохраняем n ключей в хеш-таблице размером $m = n$ с использованием хеш-функции h , выбираемой случайным образом из универсального множества хеш-функций, и устанавливаем размер каждой вторичной хеш-таблицы равным $m_j = n_j^2$ ($j = 0, 1, \dots, m-1$), то вероятность того, что общее количество необходимой для вторичных хеш-таблиц памяти не менее $4n$, меньше чем $\frac{1}{2}$.*

Доказательство. Применим неравенство Маркова $P(X \geq t) \leq \mathbf{E}[X]/t$

Пусть $X = \sum_{j=0}^{m-1} m_j$ и $t = 4n$.

Тогда $P\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} \leq \mathbf{E}\left[\sum_{j=0}^{m-1} m_j\right] \cdot \frac{1}{4n} < \frac{2n}{4n} = \frac{1}{2}$, ч.т.д. ■

5.4 Алгоритм построения статической хеш-таблицы

1. Строим хеш-таблицу 1-го уровня размера n так, чтобы $\sum_{i=1}^n n_i^2 \leq 4n$ (для этого генерируем h) ~ 2 итерации по теореме 5.3 .
2. Для каждого i строим хеш-таблицу размера n_i^2 так, чтобы не было коллизий (для этого генерируем h_i) ~ 2 итерации теореме 5.1 .

Таким образом, получим отсутствие коллизий за $\mathcal{O}(N)$ времени в среднем и $\mathcal{O}(N)$ доп. памяти.

6 Графы. Представление графов в памяти. Обход в ширину. Поиск кратчайших путей с помощью обхода в ширину.

6.1 Графы

Def. Любой *граф* можно определить как некоторый набор $a = (V, E)$, где:

- V - множество вершин
- $E \subseteq V \times V$ - множество рёбер

Def. *Ориентированный граф*(орграф) - граф, рёбрам которого присвоено направление.

Def. *Неориентированный граф*(неорграф) - граф, ни одному ребру которого не присвоено направление.

6.2 Представление графов в памяти

1. Список ребер $<(U_1, U_2), (U_2, U_4), (U_5, U_2), \dots>$

- Проверка наличия ребра - $\mathcal{O}(E)$
- Узнать соседей U вершины V : $(V, U) \in E$. - $\mathcal{O}(E)$
- $M = \mathcal{O}(E)$

2. Матрица смежности $A = V \times V$

$a_{ij} = I((i, j) \in E)$ - элемент матрицы, являющийся индикатором наличия ребра или количеством таких рёбер

- Проверка наличия ребра - $\mathcal{O}(1)$
- Узнать соседей U вершины V : $(V, U) \in E$. - $\mathcal{O}(V)$
- $M = \mathcal{O}(V^2)$

3. Списки смежности(список списков смежных вершин)

- Проверка наличия ребра - $\mathcal{O}(\deg V)$
- Узнать соседей U вершины V : $(V, U) \in E$. - $\mathcal{O}(\deg V)$
- $M = \mathcal{O}(\sum_{v \in V} \deg_+ V) = \mathcal{O}(E)$

6.3 Обход в ширину(англ. *Breadth-First Search*, BFS)

6.3.1 Описание алгоритма

Пусть задан невзвешенный неориентированный граф $G = (V, E)$, в котором выделена исходная вершина s . Требуется найти длину кратчайшего пути (если таковой имеется) от одной заданной вершины до другой. Частным случаем указанного графа является невзвешенный ориентированный граф.

Для алгоритма нам потребуются очередь и множество посещенных вершин q , которые изначально содержат одну вершину s . На каждом шагу алгоритм берет из начала очереди вершину u и добавляет все непосещенные смежные с u вершины в q и в конец очереди. Если очередь пуста, то алгоритм завершает работу.

6.3.2 Псевдокод

```
def bfs(G, s):  
    q = []  
    d[0, ..., |V| - 1] = infinity  
    p[0, ..., |V| - 1] = None  
  
    q.push(s)  
    d[s] = 0  
    while q is not []:  
        u = q.pop()  
        for v in neighbours(u):  
            if d[v] > d[u] + 1
```

```

d[v] = d[u] + 1
p[v] = u
q.push(v)

```

6.3.3 Анализ времени работы

- Список ребер

$$T(V, E) = V(\text{while}) \cdot E(\text{for}) = \mathcal{O}(VE)$$

- Матрица смежности

$$T(V, E) = V(\text{while}) \cdot V(\text{for}) = \mathcal{O}(V^2)$$

- Списки смежности(список списков смежных вершин)

$$T(V, E) = V(\text{while}) \cdot \deg V(\text{for}) = \mathcal{O}(\sum_{v \in V} (1 + \deg V)) = \mathcal{O}(V + \sum \deg V) = \mathcal{O}(V + E)$$

Если граф *плотный*(содержит много рёбер) $\Rightarrow C_V^2 = \mathcal{O}(V^2) \Rightarrow E \sim V^2$

Если граф *разреженный*, то $E \sim V$

Теорема 6.1 (Корректность BFS). В очереди поиска в ширину расстояние вершин до s монотонно неубывает.

Доказательство. **База:** для s $d[s] = 0 = \rho(s, s)$.

Пусть мы верно нашли и построили все кратчайшие пути длины $\leq n$. $(*) (d[v] \leq n \text{ или } \rho(s, v) \leq n \Rightarrow d[v] = \rho(s, v))$

Переход

- Пусть $d[v] = n + 1$ (нашли путь длины $n + 1$) $\Rightarrow (*) \rho(s, v) \geq n + 1 \Rightarrow d[v] = \rho(s, v) = n + 1$
- Пусть $\rho(s, v) = n + 1$

Рассмотрим кратчайший путь из s в v . Пусть $u = p[v]$. $\Rightarrow \rho(s, u) = n \Rightarrow (*) d[u] = n \Rightarrow d[v] \leq n + 1$

Т.к. при обработке U (предка V) мы встретили $V \Rightarrow \rho(s, v) = d[v] = n + 1 (d[v] \geq \rho(s, v))$

■

6.4 Поиск кратчайших путей с помощью обхода в ширину.

Пусть в графе у \forall вершины есть "стоимость" прохождения по ней.

- "0 - 1" граф

Решение: вместо очереди будем использовать дек и вершины, до которых ведут ребра "0" будем класть в начало, а "1" в конец.

- "0 - K" граф($\text{веса} \in \{0, 1, \dots, k\}$)

Решение 1: $T = \mathcal{O}(V + kE)$

Возьмём ребро веса k и разделим его на k фиктивных рёбер веса 1 и применим стандартный алгоритм.

- $V' \leq V + (k - 1)E$
- $E' \leq kE$

Решение 2: $T = \mathcal{O}(kV + E)$

- Заводим массив очередей q размера $k(|V| - 1)$. $q[d]$ хранит все вершины, для которых существует путь из s длины, равной d (необязательно кратчайший). Изначально $\forall i \rightarrow d[i] = \infty$
- $q[0].push(s)$, $d[s] = 0$
- идём по всем очередям, начиная с 0: достаём вершину и смотрим на всех её соседей и если новое расстояние меньше текущего, то обновляем его и добавляем в соответствующую очередь.
- повторяем алгоритм, пока не обработаем все очереди путей от 0 до $k(|V| - 1) + 1$ включительно.

7 Обход в глубину. Лемма о белых путях и следствия из нее. Лес деревьев обхода в глубину. Классификация ребер графа.

7.1 DFS.

Цвета вершин:

- *White(undiscovered)* — вершины, до которых *DFS* еще не дошел.
- *Grey(discovered)* — обнаруженные вершины.
- *Black(processed)* — полностью обработанные вершины, из них уже нет ребер в белые вершины.

Массивы времен:

- t_in — массив времен обнаружения вершин (входа).
- t_out — массив времен завершения обработки вершин (выхода).

Псевдокод:

```
def DfsVisit(G, v, color) {
    color[v] = Grey;
    t_in[v] = ++time;
    for u in G.neighbours[v]:
        if (color[u] == white):
            DfsVisit(G, u);

    color[v] = Black;
    t_out[v] = ++time;
}
```

Так как каждая вершина посещена не более одного раза

$$T = \mathcal{O}(\sum_{v \in V} (1 + \deg V)) = \mathcal{O}(V + \sum \deg V) = \mathcal{O}(V + E) \text{ в случае списка смежности.}$$

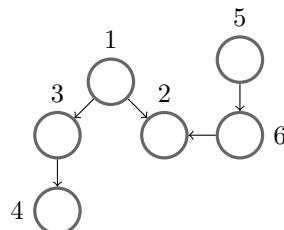
Def. Дерево (в неор. графике) — связный график без циклов.

Def. Дерево (в ор. графике) — слабосвязный график, у которого степень захода каждой вершины (количество ребер, входящих в вершину) равно одному (кроме корня дерева, у него степень захода равно нулю).

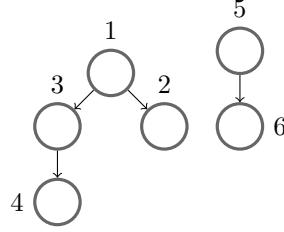
Def. Лес деревьев обхода в глубину — множество деревьев, полученное в результате вызова *DfsVisit* от нескольких разных вершин. Каждое отдельное дерево в таком случае характеризует вызов *DfsVisit* от корня этого дерева. Например, если запустить привычный код:

```
def Dfs(G):
    color = [white for vertex in G]
    for vertex in G:
        if color[vertex] == white:
            DfsVisit(G, vertex, color);
```

на графике



будет получен лес (он получен вызовом от вершин 1 и 5).



7.2 Классификация ребер:

- *Ребра дерева* — ребра, по которым обнаруживаются новые вершины (ведут в белые вершины).
- *Обратные ребра* — ребра, ведущие от потомков к предкам (ведут в серые вершины).
- *Прямые ребра* — ребра, ведущие от предков к потомкам, но не являющиеся ребрами дерева (ведут в черные вершины, но вход в вершину-конец ребра был совершен после входа в вершину-начало ребра).
- *Перекрестные ребра* — ребра, связывающие разные поддеревья (ведут в черные вершины, но вход в вершину-конец ребра был совершен до входа в вершину-начало ребра).

7.3 Лемма о белых путях.

Лемма. v — потомок u (то есть v будет посещена при вызове $DfsVisit(u)$) \Leftrightarrow существует путь из u в v , проходящий только по белым вершинам.

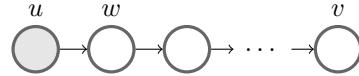
Доказательство:

- \Rightarrow

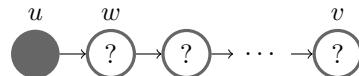
Достаточно рассмотреть последовательность вызовов $DfsVisit$ на пути от v до u .

- \Leftarrow

Существует путь по белым вершинам, допустим этот путь не обнаружен.



После вызова $DfsVisit(u)$:



“?” означает, что цвет вершины неизвестен.

Так как u черная, все вызовы $DfsVisit$ от ее белых соседей (в том числе от соседа w) завершены, значит, эти соседи (которые раньше были белыми, в том числе w) теперь черные. Аналогично из черной вершины не может быть ребра в белую, так как иначе вызов $DfsVisit$ не завершился бы, значит, все вершины на пути $u \rightarrow v$ черные, то есть посещены.

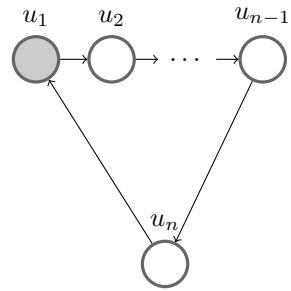
Следствие 1: Вызов $DfsVisit(v)$ посещает все вершины, достижимые из v . Для неориентированных графов такой вызов найдет компоненту связности, в которой лежит v .

Следствие 2 (проверка наличия цикла): В процессе работы DFS встречена серая вершина \Leftrightarrow в графе есть цикл.

Доказательство. • \Rightarrow

Очевидно.

- \Leftarrow



Без ограничения общности u_1 посещена первой из цикла. По лемме о белых путях так как существует путь $u_1 \rightarrow u_n$ только по белым вершинам, u_n будет посещена. Когда она посещена, от нее будет вызван $DfsVisit$, а так как из нее в u_1 ведет ребро, алгоритм придет в серую вершину u_1 . ■

8 Топологическая сортировка. Сильно связные компоненты. Конденсация графа. Алгоритм Косарайю.

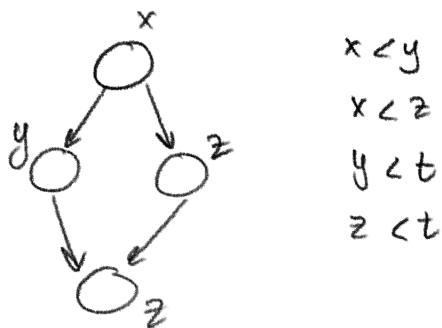
8.1 Топологическая сортировка.

Note. Обычная сортировка работает с линейно упорядоченными множествами:

$(a_1 < a_2 < \dots < a_n, \forall a_i, a_j : a_i < a_j \text{ либо } a_j < a_i, a_i \neq a_j)$ Топологическая сортировка работает с частично упорядоченными множествами (не все сравнимы)

8.1.1 Идея.

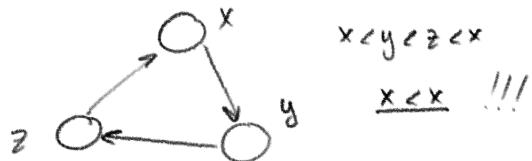
Построим граф, в котором V - элементы множества, $(u, v) \in E : u < v$ (для упорядоченного множества этот граф — полный)



Пояснительный рисунок 1.

Def. Топологически отсортировать массив - расположить элементы так, чтобы все ребра шли слева направо.

Note. Всегда ли корректно определена топологическая сортировка? Топологическая сортировка имеет смысл только для ациклических графов.



Пояснительный рисунок 2.

8.1.2 Алгоритм.

```

TopSort(G):
    color[G] = white;
    answer = [];
    for v in G.V:
        if color[v] == white:
            DFSVisit(G, v, answer, color);
    return answer;

DFSVisit(G, v, answer, color):
    color[v] = grey;
    for u in G.neighbours(v):
        if color[u] == grey:
            error;
        if color[u] == white:

```

```

    DFSVisit(G, u, answer, color);
    color[v] = black;
    answer.push_front(v);

```

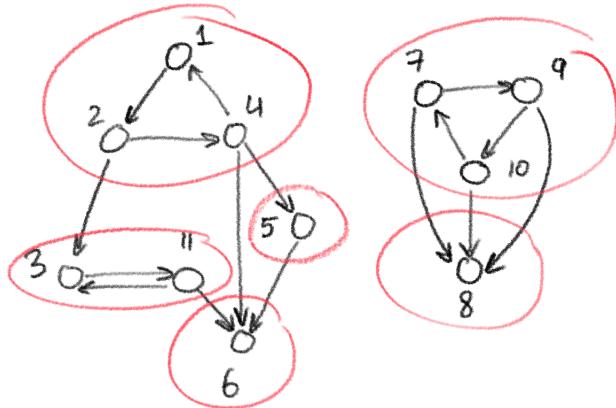
8.1.3 Корректность.

1. *TopSort* либо определяет наличие цикла,
 2. либо строит корректную топологическую сортировку.
1. Следует из критерия ацикличности.
2. Допустим циклов нет. Если в ”*for u in G.neighbours(v)*” встретил черную вершину, то она уже в списке → это ребро (v, u) идет слева направо. Если —— встретил белую, то запускаем $DFS(u)$ и ждем когда завершится этот вызов. → когда-нибудь эта вершина покраснеет и добавится в список, и только потом положил в список $v \rightarrow$ ребро (v, u) идет слева направо.

8.2 Сильно связные компоненты.

Def. *StronglyConnectedComponents(SCC)*

1. Слабая связность - если при замене всех ребер на неориентированные, получается связный граф.
2. Сильная связность - можно добраться в графе из x в y и наоборот (компоненты сильной связности - класс эквивалентности по отношению взаимной достижимости или же наибольший подграф, из любой вершины которого можно добраться до любой другой).



Пояснительный рисунок 3.

Def. Конденсация графа G - граф, в котором вершины - $SCC G$, а ребро $\exists \Leftrightarrow$ есть ребро из одной SCC в другую.
Note. Конденсации всегда ацикличны.

8.3 Алгоритм Косарайю(Kosaraju).

8.3.1 Идея.

1. Поиск в глубину на G и сортируем вершины в порядке, обратном времени завершения (по убыванию $tout$); *TopSort* без проверки на цикличность ($O(V + E)$)
2. Строим транспонированный граф G^T (обращаем все ребра) ($O(V + E)$)
3. Запустим DFS на G^T в порядке, полученном на шаге 1. Если обошли не все вершины, то запускаемся снова от первой(согласно пункту 1) еще не посещенной вершины.
4. Каждая найденная компонента в пункте 3 - это сильно связная компонента.

8.3.2 Корректность.

8.4 Лемма.

Если \exists ребро из $SCCu$ в $SCCv$, то \exists вершина $u \in SCCu$: она будет расположена левее всех вершин $SCCv$ после шага 1.

Пусть u - первая вершина из $SCCu$, которую нашел DFS . Возможны 2 случая:

1. Еще не посещали $SCCv$, тогда по лемме о белом пути посетим все вершины из $SCCu$ и $SCCv$. \rightarrow все эти вершины окажутся в конце списка, а потом положим вершину $u \rightarrow u$ левее всех остальных
2. Уже посещали $v \in SCCv$. По лемме о белых путях посетили все вершины из $SCCv$ \rightarrow они все уже в списке $\rightarrow u$ заведомо лежит левее

8.4.1 Корректно найдем $SCCu$.

По лемме u лежит левее $SCCv \rightarrow u$ обработают раньше, чем $SCCv$ (на шаге 3). Запускаем $DFS(u) \rightarrow$ по лемме о белых путях побываю во всех вершинах из $SCCu$, так как транспонирование не влияет на связность, и не попадаем в $SCCv$, так как ребро не изменилось \rightarrow посетим только вершины из $SCCu$.

9 Мосты и точки сочленения. Алгоритм поиска. Теорема Роббинса.

9.1 Мосты и точки сочленения

Def. Точки сочленения (*Cut Vertecies*) - вершины графа, при удалении которых граф распадается на большее число компонент связности (верно для неориентированных графов).

9.2 Алгоритм поиска.

Пусть G - связный граф и запустили DFS со стартовой вершиной r

1. если $u = r$, то u - точка сочленения \Leftrightarrow у $u \geq 2$ непосредственных потомка в дереве.

Доказательство: если ≥ 2 потомков. В неориентированном графе нет перекрестных ребер \rightarrow между двумя потомками нет ребер, то есть r - единственная вершина, соединяющая их $\rightarrow r$ - точка сочленения.

Если r - точка сочленения, тогда у нее ≥ 2 . От противного: у корня r 1 потомок (0 - очевидно). Тогда $\rightarrow r$ связан с оставшимся подграфом одним ребром дерева и, возможно, обратными рёбрами, но они на связность не влияют \rightarrow число компонент связности не увеличивается.

2. если $u \neq r$, тогда u - точка сочленения \Leftrightarrow у $u \exists$ потомок, из которого нет пути в предка u .

Доказательство: если u - точка сочленения: от противного. Допустим \exists путь из любого потомка в предка $u \rightarrow$ потомки u связаны и ее же предками путями не через u .

Если есть потомок, из которого нет пути в предка u . Тогда \rightarrow нет пути из предков и потомки не через $u \rightarrow u$ - это точка сочленения .

Def. Через tin - время захода (первое обнаружение вершины в DFS). Для предков $tin < tin$ для потомков.

Поэтому tin - это высота вершины в дереве (чем меньше, тем выше).

Пусть tup - это высота, но которую можно забраться из данной вершины, двигаясь только по ребрам дерева и одному обратному ребру. Пункт 2 Алгоритма поиска $\leftrightarrow tin[u] \leq tup[v]$, где v - потомок u .

9.2.1 Алгоритм.

```
CutVertices(G):
    cut_vertices = []
    color[0, ..., |V| - 1] = white
    for v in V:
        if color[v] == white:
            DFSVisit(G, v, is_root = true, cut_vertices)
    return cut_vertices;

DFSVISIT(G, v, is_root, answer):
    color[v] = grey
    tin[v] = tup[v] = ++t
    // (tup[v] = min(tin[v], tin[back], tup[child]))
    nChildren = 0
    for u in G.neighbours(v):
        if color[u] != white:
            tup[v] = min(tup[v], tin[u]);
        if color[u] == white:
            DFSVisit(G, u, is_root = false, answer)
            ++nChildren
            tup[v] = min(tup[v], tup[u])
            if tin[v] <= tup[u]:
                if not is_root:
                    answer.push[v]
    color[v] = black
    if (is_root and nChildren >= 2):
        answer.push[v]
```

Def. Мост (только для неориентированных графов) - это ребро, при удалении которого увеличивается число компонент связности. **Note.** Алгоритм и доказательство аналогичны точкам сочленения со следующими отличиями:

1. нет случайя корня
2. $\exists v$ потомок $u : tin[u] < tup[v], (u, v)$ - мост

9.3 Теорема Роббинса.

Неориентированный граф можно сильно ориентировать (то есть сохранить сильную связность) \leftrightarrow в G нет мостов.
Доказательство: \rightarrow От противного: Пусть есть мост. Ориентируем его. Пусть $(u, v) \rightarrow$ нет пути (v, u) **Доказательство:** \leftarrow Запустим DFS . Ориентируем ребра согласно ребрам дерева (вниз) и ребрам обратно (вверх). Рассмотрим вершину $v : tin[v] \geq tup[childrenv]$ \rightarrow из любого потомка v можно добраться выше (в предка v) \rightarrow из любой вершины можно добраться до корня \rightarrow , а из корня можно добраться в любую другую.

10 Эйлеровы пути и циклы. Критерий эйлеровости и полуэйлеровости. Алгоритм построения эйлерова цикла.

10.1 Основные определения

Def. Эйлеров путь (эйлерова цепь) в графе — это путь, проходящий по всем рёбрам графа и притом только по одному разу.

Def. Эйлеров цикл — эйлеров путь, являющийся циклом, то есть замкнутый путь, проходящий через каждое ребро графа ровно по одному разу.

Def. Граф называется эйлеровым, если он содержит эйлеров цикл.

Def. Граф называется полуэйлеровым, если он содержит эйлеров путь, но не содержит эйлеров цикл.

10.2 Критерий эйлеровости

Утверждение. Для того, чтобы граф $G = (V, E)$ был эйлеровым необходимо чтобы:

1. Все вершины имели четную степень.
2. Все компоненты связности кроме, может быть одной, не содержали ребер.

Доказательство:

1. Допустим в графе существует вершина с нечетной степенью. Рассмотрим эйлеров обход графа. Заметим, что при попадании в вершину и при выходе из нее мы уменьшаем ее степень на два (помечаем уже пройденные ребра), если эта вершина не является стартовой(она же конечная для цикла). Для стартовой(конечной) вершины мы уменьшаем ее степень на один в начале обхода эйлерова цикла, и на один при завершении. Следовательно вершин с нечетной степенью быть не может. Наше предположение неверно.
2. Если в графе существует более одной компоненты связности с ребрами, то очевидно, что нельзя пройти по их ребрам одним путем. На случай ориентированного графа доказательство продолжается аналогично.

10.3 Критерий полуэйлеровости

10.3.1 Подводящая теорема

Утверждение. В графе $G = (V, E)$ существует эйлеров цикл тогда и только тогда, когда:

1. Все вершины имели четную степень.
2. Все компоненты связности кроме, может быть одной, не содержат ребер.

Доказательство: Необходимость мы доказали ранее. Докажем достаточность, используя индукцию по числу вершин n .

База индукции: $n = 0$ цикл существует.

Предположим, что граф, имеющий менее n вершин содержит эйлеров цикл. Рассмотрим связный граф $G = (V, E)$ с $n > 0$ вершинами, степени которых четны. Пусть v_1 и v_2 — вершины графа. Поскольку граф связный, то существует путь из v_1 в v_2 . Степень v_2 — чётная, значит существует неиспользованное ребро, по которому можно продолжить путь из v_2 . Так как граф конечный, то путь, в конце концов, должен вернуться в v_1 , следовательно мы получим замкнутый путь(цикл).

Назовем этот цикл C_1 . Будем продолжать строить C_1 через v_1 таким же образом, до тех пор, пока мы в очередной раз не сможем выйти из вершины v_1 , то есть C_1 будет покрывать все ребра, инцидентные v_1 . Если C_1 является эйлеровым циклом для G , тогда доказательство закончено. Если нет, то пусть G' — подграфа графа G , полученный удалением всех рёбер, принадлежащих C_1 . Поскольку C_1 содержит чётное число рёбер, инцидентных каждой вершине, то каждая вершина подграфа G' имеет чётную степень. А так как C_1 покрывает все ребра, инцидентные v_1 , то граф G' будет состоять из нескольких компонент связности.

Рассмотрим какую-либо компоненту связности G' . Поскольку рассматриваемая компонента связности G' имеет менее, чем n вершин, а у каждой вершины графа G' чётная степень, то у каждой компоненты связности G' существует эйлеров цикл. Пусть для рассматриваемой компоненты связности это цикл C_2 . У C_1 и C_2 имеется общая вершина a , так как G связен. Теперь можно обойти эйлеров цикл, начиная его в вершине a , обойти C_1 , вернуться в a , затем пройти C_2 и вернуться в a . Если новый эйлеров цикл не является эйлеровым циклом для G , продолжаем использовать этот процесс, расширяя наш эйлеров цикл, пока, в конце концов, не получим эйлеров цикл для G .

10.3.2 Критерий полуэйлеровости

Утверждение. В графе $G = (V, E)$ существует эйлеров путь тогда и только тогда, когда:

1. Количество вершин с нечетной степенью меньше или равно двум.
2. Все компоненты связности кроме, может быть одной, не содержат ребер.

Доказательство: Добавим ребро, соединяющее вершины с нечетной степенью. Теперь можно найти эйлеров цикл, после чего удалить добавленное ребро. Очевидно найденный цикл станет путем.

Утверждение. В ориентированном графе $G = (V, E)$ существует эйлеров путь если:

1. Входная степень любой вершины равна ее выходной степени, кроме двух вершин графа, для одной из которых разность входной и выходной степеней равна единице.
2. Все компоненты слабой связности кроме, может быть одной, не содержат ребер.

Доказательство: Соединим ориентированным ребром вершину с большей входящей степенью с вершиной с большей исходящей степенью. Теперь можно найти эйлеров цикл, после чего удалить добавленное ребро. Очевидно найденный цикл станет путем.

10.4 Алгоритм построения эйлерова пути(цикла)

10.4.1 Описание работы

Сначала необходимо проверить граф на полуэйлеровость(эйлеровость). Учитывая тот факт, что эйлеров цикл есть частный случай эйлерова пути, будем искать эйлеров путь. Если граф содержит эйлеров путь(цикл), найдем его следующим алгоритмом(работает корректно в случае ориентированного и неориентированного графа).

Чтобы построить Эйлеров путь, нужно запустить алгоритм из вершины с нечетной степенью. Чтобы построить эйлеров цикл нужно запустить алгоритм из любой вершины с выходной степенью четной и большей нуля(так как все вершины данной компоненты связности войдут в цикл, нет различия с какой именно вершиной начинать алгоритм). Алгоритм напоминает поиск в глубину. Главное отличие состоит в том, что пройденными помечаются не вершины, а ребра графа. Начиная со стартовой вершины v строим путь, добавляя на каждом шаге не пройденное еще ребро, смежное с текущей вершиной. Вершины пути накапливаются в стеке S . Когда наступает такой момент, что для текущей вершины w все инцидентные ей ребра уже пройдены, записываем вершины из S в ответ, пока не встретим вершину, которой инцидентны не пройденные еще ребра. Далее продолжаем обход по не посещенным ребрам.

Листинг 1: check function example

```
bool checkEulerianPath() {
    int oddVerticesCounter = 0;
    for (const auto& vertex : V) {
        if (GetDegree(vertex) % 2 == 1) {
            ++oddVerticesCounter;
        }
    }
    if (oddVerticesCounter > 2) {
        return false;
    }

    std::vector<bool> visited(numVertices, false);
    for (const auto& vertex : V) {
        if (GetDegree(vertex) > 0) {
            DFS(vertex, visited);
            break;
        }
    }

    for (const auto& vertex : V) {
        if (GetDegree(vertex) > 0 && !visited[vertex]) {
            return false;
        }
    }
    return true;
}
```

Листинг 2: algorithm example

```
auto findEulerianPath(bool hasOddDegrees=false) {
    Vertex_t startVertex;
    if (hasOddDegrees) {
        for (const auto& vertex : V) {
            if (GetDegree(vertex) % 2 == 1) {
                startVertex = vertex;
                break;
            }
        }
    } else {
        for (const auto& vertex : V) {
            if (GetDegree(vertex) > 0) {
                startVertex = vertex;
                break;
            }
        }
    }
}

std::stack<Vertex_t> S;

S.push(startVertex);
while (!S.empty()) {
    bool foundEdge = false;

    for (const auto& vertex : V) {
        Edge_t tempEdge = MakeEdge(S.top(), vertex);
        if (E.find(tempEdge) != E.end()) {
            //found a not traversed edge
            S.push(vertex);
            E.remove(tempEdge);
            foundEdge = true;
            break;
        }
    }

    if (!foundEdge) {
        //there are no edges connected to S.top,
        //which have not been traversed already

        std::cout << S.top();
        //S.top is a part of eulerian path
        S.pop();
    }
}
}
```

10.4.2 Корректность алгоритма

Утверждение. Данный алгоритм проходит по каждому ребру, причем ровно один раз.

Доказательство: Допустим, что в момент окончания работы алгоритма имеются еще не пройденные ребра. Поскольку граф связен, должно существовать хотя бы одно не пройденное ребро, инцидентное посещенной вершине. Но тогда эта вершина не могла быть удалена из стека S , и он не мог стать пустым. Значит алгоритм пройдет по всем рёбрам хотя бы один раз. Но так как после прохода по ребру оно удаляется, то пройти по нему дважды алгоритм не может. (Примечание: Вершина v , с которой начат обход графа, будет последней помещена в путь P . Так как изначально стек пуст, и вершина v входит в стек первой, то после прохода по инцидентным ребрам, алгоритм возвращается к данной вершине, выводит ее и опустошает стек, затем выполнение программы завершается.)

Утверждение. Напечатанный путь P — корректный маршрут в графе, в котором каждые две соседние вершины u_i и u_{i+1} будут образовывать ребро $(u_i, u_{i+1}) \in E$.

Доказательство: Будем говорить, что ребро (w, u) представлено в S или P , если в какой-то момент работы алгоритма вершины w и u находятся рядом. Каждое ребро графа представлено в S . Рассмотрим случай, когда из S в P перемещена вершина u , а следующей в S лежит w . Возможны 2 варианта:

1. На следующем шаге для вершины w не найдётся инцидентного ребра, тогда w переместят в P , и ребро (w, u) будет представлено в P .
2. Иначе будет пройдена некоторая последовательность ребер u_1, u_2, \dots, u_k , начинающаяся в вершине w и проходящая по ребру (w, u_1) . Докажем, что данный проход u_1, u_2, \dots, u_k закончится в вершине w :
 - (a) Ребро (u_{k-1}, u_k) не может быть инцидентно вершинам u_1, \dots, u_{k-2} , иначе степень вершины u_k окажется нечетной.
 - (b) Предположим, что (u_{k-1}, u_k) инцидентно вершине, пройденной при обходе графа из вершины u . Но это неверно, так как тогда бы данные вершины пройдены ранее.

Из этого следует, что мы закончим обход в вершине w . Следовательно, данная вершина первой поместится в P вслед за u , и ребро (w, u) будет представлено в P .

Утверждение. Данный алгоритм находит корректный эйлеров путь. **Доказательство:** Из предыдущих утверждений следует, что P — искомый эйлеров путь и алгоритм работает корректно.

10.4.3 Время работы

Если реализовать поиск ребер инцидентных вершине и удаление ребер за $\mathcal{O}(1)$, то алгоритм будет работать за $\mathcal{O}(E)$. Чтобы реализовать поиск за $\mathcal{O}(1)$, для хранения графа следует использовать списки смежных вершин; для удаления достаточно добавить всем ребрам свойство *deleted* логического типа.

11 Система непересекающихся множеств. Реализация с помощью леса непересекающихся множеств. Ранговая эвристика, анализ времени работы. Эвристика сжатия путей, анализ времени работы(б/д).

11.1 Система непересекающихся множеств

Def. Система непересекающихся множеств(DSU — Disjoined Set Union) — это иерархическая структура данных, позволяющая эффективно работать с множествами. А именно — выполнять следующий набор операций за оптимальное время:

1. *MakeSet* — добавлять новый элемент x , помещая его в новое множество, состоящее из одного него.
2. *UnionSets* — объединять два непересекающихся множества в одно новое.
3. *FindSet* — возвращать по элементу какого-либо множества идентификатор этого множества.

11.2 Реализация с помощью леса непересекающихся множеств

11.2.1 Наивная реализация

Основная идея заключается в том, чтобы представлять множества в виде деревьев. У каждого множества будет так называемый представитель — *parent*. Таким образом все элементы множества, включая *parent* этого множества, ссылаются на *parent* при вызове операции *FindSet*.

Поговорим о самой реализации. Заведем массив *parents*, в котором будем хранить представителя каждого элемента. При создании нового множества из одного элемента, в этот массив просто добавляется новый элемент, представителем которого является он сам.

Рассмотрим теперь операцию *UnionSets*(X, Y). Пусть представителем элементов множества X является элемент A , а представителем элементов множества Y является элемент B . Теперь произвольно выберем, например, B и сделаем его представителем элемента A . Таким образом мы как бы подвесим дерево Y к дереву X .

Теперь обсудим реализацию операции *FindSet*(x). Чтобы однозначно определить к какому множеству принадлежит элемент x в общем случае недостаточно обратиться к представителю x , так как этот представитель мог быть «подвешен» к другому дереву. Действовать нужно следующим образом. Изначально текущим элементом является x , обращаемся к *parents*[x] = y , затем текущим элементом становится y и так далее. Рекурсивно обращаемся к представителю текущего элемента до тех пор, пока представитель текущего элемента не станет равным самому текущему элементу — *parents*[v] == v . Значит текущий элемент и есть представитель-корень данного дерева. Это и есть результат операции *FindSet*(x).

При помощи такой структуры данных можно отвечать на запросы вида: «лежат ли два элемента x и y в одном множестве?» Этот запрос эквивалентен проверке условия *FindSet*(x) == *FindSet*(y).

11.2.2 Недостаток наивной реализации

Выше мы уже выяснили, что операция *FindSet* работает рекурсивно, проходясь по представителям текущих элементов. Можно подобрать примеры, когда операция *FindSet* будет работать за $\mathcal{O}(N)$, что далеко от желаемой асимптотики $\mathcal{O}(1)$. Например, если создать N элементов и объединить их по схеме так, чтобы представителем a_i был элемент a_{i+1} (кроме последнего элемента, представителем которого является он сам). Таким образом вызов *FindSet*(a_1) будет выполнен за время $\mathcal{O}(N)$.

11.2.3 Код наивной реализации

Листинг 3: naive DSU example

```
void make_set (int v) {
    parents[v] = v;
}

int find_set (int v) {
    if (v == parents[v])
        return v;
    return find_set (parents[v]);
}
```

```

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b)
        parents[b] = a;
}

```

11.3 Ранговая эвристика, анализ времени работы

11.3.1 Ранговая эвристика

Ранговая эвристика заключается в оптимизации операции *UnionSets*. Ранее мы выбирали какое дерево к какому «присоединять» случайным образом. Теперь мы можем заметить, что если «подвесить» меньшее по глубине дерево к большему, то глубина полученного дерева не изменится. Причем если «подвесить» большее дерево к меньшему, то глубина итогового дерева будет равна сумме глубины большего дерева и единицы, т.е. в этом случае глубина определяется так: $depth(Res) = \max(depth(X), depth(Y)) + 1$. Причем если глубина X равна глубине Y , то итоговая глубина равна сумме глубины любого из деревьев и единицы.

11.3.2 Реализация

Для реализации механизма ранговой эвристики заведем массив $rank$, в котором для каждого из элементов хранится глубина дерева, представителем которого он является. Затем при объединении деревьев мы будем сравнивать их ранги и выбирать большее дерево.

11.3.3 Код ранговой эвристики

Листинг 4: example

```

void make_set (int v) {
    parent[v] = v;
    rank[v] = 0;
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            ++rank[a];
    }
}

```

11.3.4 Время работы

Операция *UnionSets* работает, очевидно, за время выполнения двух операций *FindSet* плюс еще константное время. Осталось показать, что теперь операция *FindSet* работает за время $\mathcal{O}(\log N)$. Заметим, что *FindSet* работает за время «высоты дерева». Покажем, что при ранге дерева k , оно содержит минимум 2^k элементов. Докажем по индукции.

1. База — очевидно, $1 \geq 2^0$.
2. Шаг — если дерево ранга $k - 1$ увеличило свой ранг на 1, то это значит, что к нему было добавлено дерево такого же ранга. Попредположению индукции в новом дереве не менее $N = 2^{k-1} + 2^{k-1} = 2^k$ элементов.

Это означает, что *FindSet* будет работать за время $\mathcal{O}(\log N)$.

11.4 Эвристика сжатия путей, оценка времени работы

11.4.1 Эвристика сжатия путей

Эта эвристика оптимизирует операцию *FindSet*. Заметим, что при одном рекурсивном проходе при вызове *FindSet*(x) мы получим информацию о том, что представителем x , его предка, предка его предка и так далее, является результат операции *FindSet*(x) = p . Оптимизация заключается в том, что всем элементам, которые мы «встречаем» во время рекурсивного прохода, назначить представителя, который является итоговым результатом выполнения операции *FindSet*.

11.4.2 Код эвристики сжатия путей

Листинг 5: example

```
int find_set (int v) {
    if (v == parent[v])
        return v;
    parent[v] = find_set (parent[v]);
}
```

11.5 Код с использованием обеих эвристик сжатия

Листинг 6: example

```
void make_set (int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find_set (int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set (parent[v]);
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            ++rank[a];
    }
}
```

11.6 Оценка времени работы DSU(б/д)

Без доказательства

$\alpha(N)$ — обратная функция Аккермана, которая растёт очень медленно, настолько медленно, что для всех разумных ограничений N она не превосходит 4.

1. $MakeSet(x) = \mathcal{O}(1)$
2. $UnionSets(x, y) = \mathcal{O}(\alpha(N))$
3. $FindSet(x) = \mathcal{O}(\alpha(N))$

12 Минимальные оставные деревья. Лемма о безопасном ребре. Алгоритм Прима. Алгоритм Краскала.

12.1 Минимальные оставные деревья

Рассмотрим связный неориентированный взвешенный граф $G = (V, E)$, где V — множество вершин, E — множество ребер. Вес ребра определяется, как функция $w : E \rightarrow R$.

Def. Оставное дерево (англ. spanning tree) графа $G = (V, E)$ — ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины.

Def. Минимальное оставное дерево (англ. minimum spanning tree) графа $G = (V, E)$ — это его ациклический связный подграф, в который входят все его вершины, обладающий минимальным суммарным весом ребер.

Пусть G' — подграф некоторого минимального оставного дерева графа $G = (V, E)$.

Def. Ребро $(u, v) \notin G'$ называется безопасным (англ. safe edge), если при добавлении его в $G', G' \cup (u, v)$ также является подграфом некоторого минимального оставного дерева графа G .

Def. Разрезом (англ. cut) неориентированного графа $G = (V, E)$ называется разбиение V на два непересекающихся подмножества: S и $T = V \setminus S$. Обозначается как $\langle S, T \rangle$.

Def. Ребро $(u, v) \in E$ пересекает разрез $\langle S, T \rangle$, если один из его концов принадлежит множеству S , а другой — множеству T .

12.2 Лемма о безопасном ребре

Утверждение. Рассмотрим связный неориентированный взвешенный граф $G = (V, E)$ с весовой функцией $w : E \rightarrow R$. Пусть $G' = (V, E')$ — подграф некоторого минимального оставного дерева G , $\langle S, T \rangle$ разрез G , такой, что ни одно ребро из E' не пересекает разрез, а (u, v) — ребро минимального веса среди всех ребер, пересекающих разрез $\langle S, T \rangle$. Тогда ребро $e = (u, v)$ является безопасным для G' .

Доказательство: Рассмотрим разрез графа $\langle S, T \rangle$. Обозначим ребро минимального веса, пересекающее разрез, как $e = (u, v)$, где $u \in S$ и $v \in T$.

$G' \subseteq G$. Рассмотрим минимальное оставное дерево MST , подмножеством которого является G' .

Предположит, что ребро $e = (u, v)$ не лежит в MST . Значит, на пути из u в v есть ребро e' , пересекающее разрез. Мысленно удалим $e = (u, v)$ и возьмем в оставное дерево ребро e' . Это необходимо сделать, так как оставное дерево должно связывать все вершины графа. Добавим в дерево ребро e . Получим цикл, содержащий ребра e и e' .

По условию $w(e) \leq w(e')$. Если удалить ребро e' , то граф останется связным, и вес дерева не увеличится. Значит, мы получили, что если в оставное дерево вместо ребра e' взять ребро e , то вес оставного дерева не увеличится. Это утверждение справедливо для e и любого другого ребра, пересекающего разрез. Значит, e является элементом минимального оставного дерева.

12.3 Алгоритм Прима

12.3.1 Идея

Алгоритм относится к классу жадных. Это и определяет основную идею его работы.

На каждом этапе работы мы будем хранить список вершин, которые уже попали в MST . Оставшиеся вершины мы будем хранить в пирамиде по минимуму, где ключом будет выступать «расстояние» до MST . Под словом расстояние будем иметь в виду минимальную сумму весов ребер, по которым от текущей вершины можно достичь одну из вершин, содержащихся в MST , при условии что в этом пути только конечная вершина содержится в MST . На каждом шаге будем выбирать вершину-минимум из пирамиды, обновлять «расстояния» до MST для всех соседей выбранной вершины. Продолжаем процесс до тех пор, пока все вершины не окажутся выбранными в MST .

12.3.2 Корректность

Корректность этого алгоритма вытекает из леммы о безопасном ребре. На каждом шаге мы разрезаем граф и выбираем минимальное ребро, пересекающее разрез.

12.3.3 Код

Листинг 7: Prim's algorithm using adjacency matrix

```
void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }

    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
        int vertex = minHeap(key, mstSet);
        mstSet[vertex] = true;

        for (int temp = 0; temp < V; ++temp) {
            if (graph[vertex][temp] != 0
                && mstSet[temp] == false
                && graph[vertex][temp] < key[temp]) {
                parent[temp] = vertex;
                key[temp] = graph[vertex][temp];
            }
        }
    }
}
```

12.3.4 Оценка времени работы

Реализация с линейным поиском минимального ребра и матрицей смежности работает за время $\mathcal{O}(V^2)$, что удобно в случае плотных графов(т.е. $E = \mathcal{O}(V^2)$).

Реализация с поиском в пирамиде и со списком смежности работает за время $\mathcal{O}(E \cdot \log V)$, что удобно в случае разреженных графов(т.е. $E = \mathcal{O}(V)$).

12.4 Алгоритм Краскала

Идея

- Сортируем ребра по возрастанию весов. ($T(n) = \mathcal{O}(n \log n)$).
- В полученном порядке проверяем ребра на то, добавляют ли они в уже построенный MST цикл (проверка на основе DSU , каждое поддерево — отдельное множество вершин, ребро не образует цикла, если его концы лежат в разных множествах), если новое ребро безопасно, добавляем его, объединяя множества, к которым относятся его концы.
- Повторяем предыдущий шаг, пока будет построено дерево.

12.4.1 Код

Листинг 8: Kruskal's algorithm

```
void KruskalMST(Graph* graph)
{
    int V = graph->V;
    Edge result[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges
```

```

qsort(graph->edge, graph->E, sizeof(graph->edge[0]),  

      myComp);  
  

// Allocate memory for creating V subsets  

subset* subsets = new subset[(V * sizeof(subset))];  
  

// Create V subsets with single elements  

for (int v = 0; v < V; ++v)  

{  

    subsets[v].parent = v;  

    subsets[v].rank = 0;  

}  
  

// Number of edges to be taken is equal to V-1  

while (e < V - 1 && i < graph->E)  

{  

    // Step 2: Pick the smallest edge. And increment  

// the index for next iteration  

    Edge next_edge = graph->edge[i++];  
  

    int x = find(subsets, next_edge.src);  

    int y = find(subsets, next_edge.dest);  
  

    // If including this edge does't cause cycle,  

// include it in result and increment the index  

// of result for next edge  

    if (x != y) {  

        result[e++] = next_edge;  

        Union(subsets, x, y);  

    }  

    // Else discard the next_edge  

}
}

```

12.4.2 Оценка времени работы

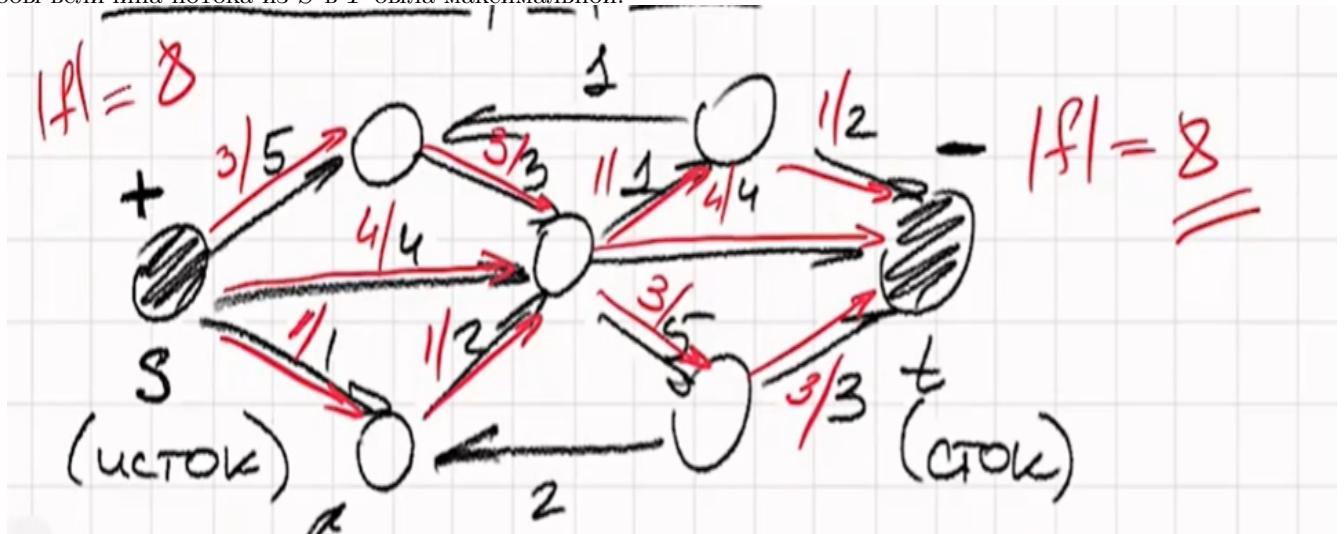
Больше всего времени тратится на сортировку ребер, алгоритм работает за $\mathcal{O}(E \log E + V)$. Корректность следует из теоремы о безопасном ребре.

13 Потоки в графах. Остаточная сеть. Задача о потоке максимальной величины. Теорема Форда-Фалкерсона. Алгоритм Форда-Фалкерсона. Алгоритм Эдмондса-Карпа.

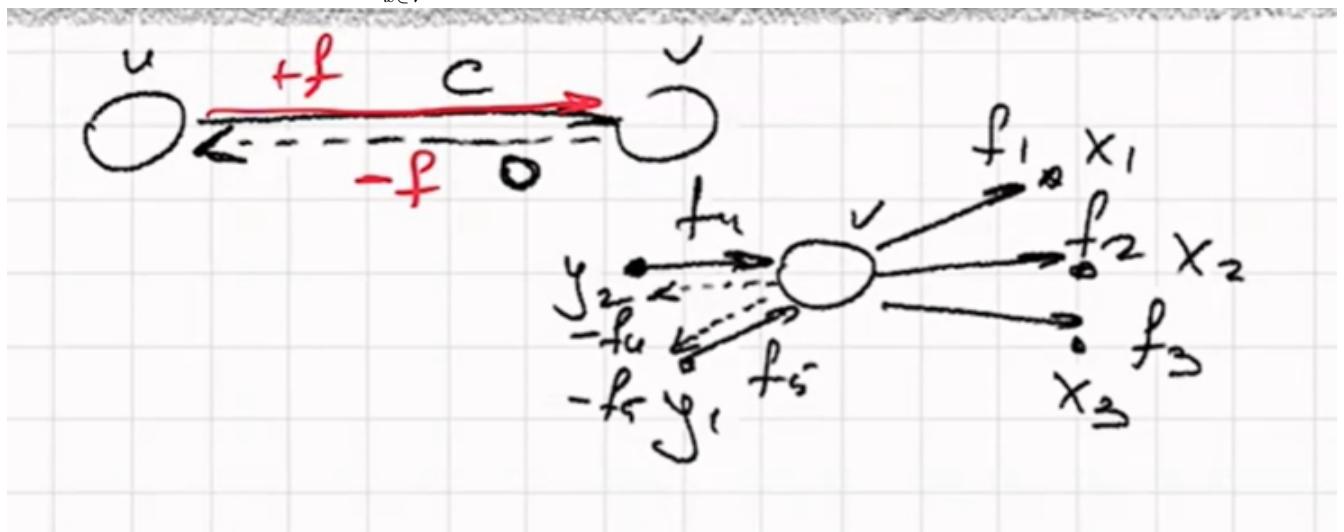
13.1 Потоки в графах

Постановка задачи: есть ориентированный граф. В нем определены 2 вершины — исток S и сток T . Из истока в сток течет жидкость, каждое ребро пропускает её только в одном направлении. Необходимо максимизировать величину потока из S в T . Проблема — у каждого ребра есть его пропускная способность (аналогия с шириной трубы и тем сколько воды пройдет через её сечение за единицу времени). В вершинах ничего не накапливается, сколько жидкости втекает, столько вытекает.

Задача состоит в том, чтобы максимизировать величину потока, т.е. на каждом ребре назначить поток так, чтобы величина потока из S в T была максимальной.



Для удобства вычислений каждому ребру назначим обратное, причем такое, что если из u в v течет поток f , то из v в u течет поток $-f$, но пропускная способность этого ребра 0. Теперь условие, что в вершине ничего не накапливается записывается, как $\sum_{x \in V} f(v, x) = 0$.



Def. f — поток в G , если $f: V \times V \rightarrow \mathbb{R}$, то

1. (антисимметричности) $\forall x, y \ f(x,y) = -f(y,x)$.
2. $\forall x, y \ f(x,y) \leq c(x,y)$. Если $(u,v) \notin E$, предполагается что $c(u,v) = 0$.
3. $\forall v \neq S,T \ \sum_{x \in V} f(v,x) = 0$.

Def. 1 Величиной потока называется $|f|_+ = \sum_{x \in V} f(s, x)$.

Def. 2 Величиной потока называется $|f|_- = \sum_{y \in V} f(y, t)$.

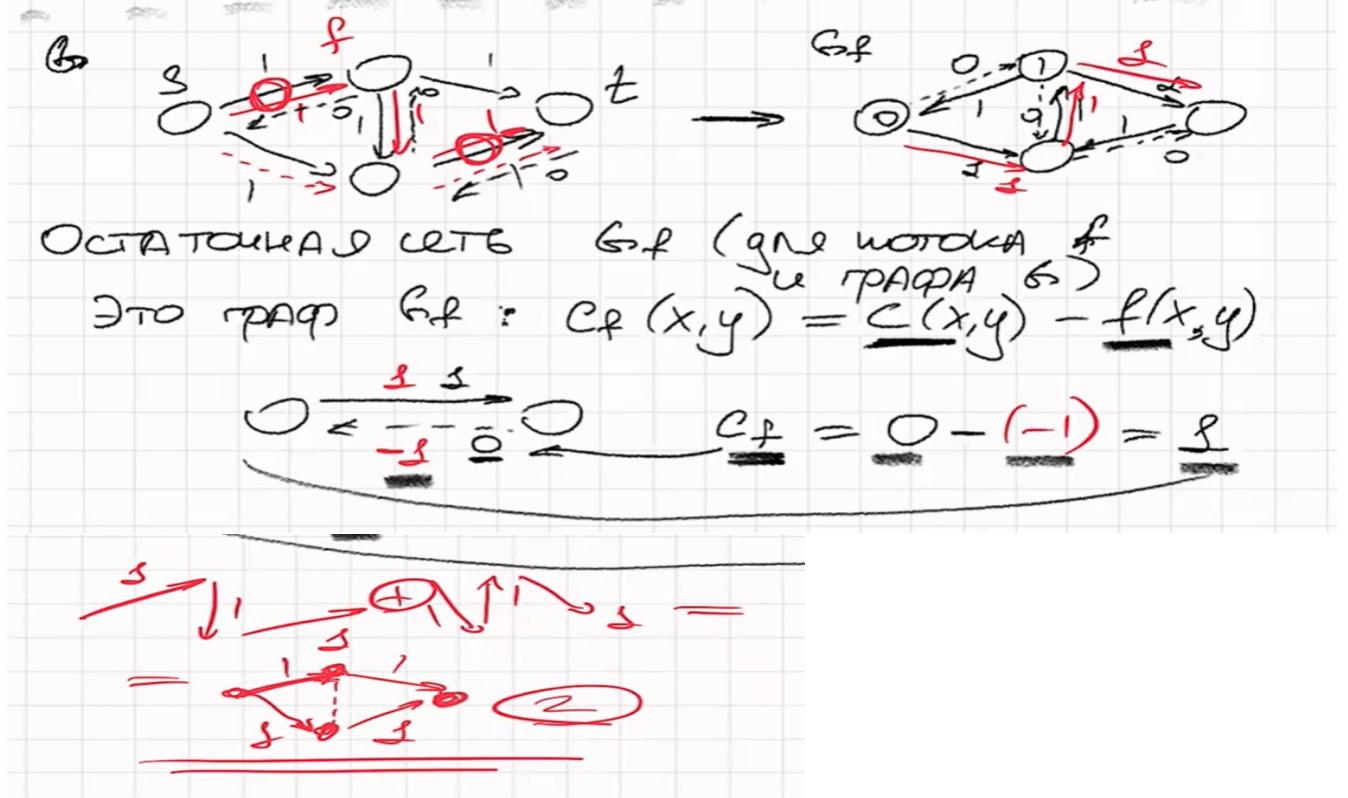
Утверждение. $|f|_+ = |f|_-$

Доказательство:

$$0 = \sum_x \sum_y f(x, y) = \sum_y f(s, y) + \sum_y f(t, y) + \sum_{x \neq s, t} \sum_y f(x, y) = (\text{воспользовались свойством 1}) = \sum_y f(s, y) + \sum_y f(t, y)$$

(воспользовались свойством 3)

Def. Остаточная сеть G_f для потока f и графа G — это граф $G_f : c_f(x, y) = c(x, y) - f(x, y)$, т.е., пропускная способность ребер которого равна разности пропускной способности исходного ребра и потока, который по немупущен в данный момент. Если вспомнить про то, что каждому ребру мы назначили обратное, то в остаточной сети могут появиться новые ребра, обозначающие направления, в которых можем исправить поток. Т.е. если мы выбрали не тот поток, то, пустив поток вдоль такого направления, можем вернуть себе ребро обратно. Таким образом остаточная сеть добавляет к графу реальные ребра, по которым мы можем такжепустить поток (пояснение на рисунке). Суммируя потоки, получаем поток большей величины.



Докажем, что можем суммировать потоки.

Лемма. Арифметика потоков

- Пусть f — поток в G , а f' в остаточной сети G_f . Тогда $f + f'$ — поток в исходном графе G , причем $|f + f'| = |f| + |f'|$

Доказательство:

Проверяем по определению, что $f + f'$ — поток.

(a) $\forall x, y f(x, y) + f'(x, y) = -f(y, x) - f'(y, x)$ (т.к. f и f' — потоки)

(b) Т.к. $f'(x, y) \leq c(x, y) - f(x, y) = c_f$. Значит $f(x, y) + f'(x, y) \leq c(x, y)$.

(c) $\forall s, t \sum_{y \in V} (f(v, y) + f'(v, y)) = 0$, потому что раз $f(v, y)$ и $f'(v, y)$ — потоки, то $\sum_{y \in V} f(v, y) = \sum_{y \in V} f'(v, y) = 0$.

Теперь покажем, что величина потока $f + f'$ совпадает с суммой величин потоков: $|f(x, y) + f'(x, y)| = \sum_{x \in V} f(s, x) + f'(s, x) = \sum_{x \in V} f(s, x) + \sum_{x \in V} f'(s, x) = |f(x, y)| + |f'(x, y)|$

- Пусть f_1 — поток в G и f_2 — поток в G , тогда $f_1 \overline{\exists} f_2$ — поток в G_{f_2} .

Доказательство: Аналогично.

13.2 Алгоритм Форда-Фалкерсона

Теорема 13.1. Поток f максимальен тогда и только тогда, когда в G_f нет пути из S в T .

(Суть: DFS-ом искать произвольный путь из S в T , потом пускать по нему поток, делать так несколько раз, пока можем его найти в остаточной сети, потом брать сумму потоков. В какой-то моменте мы не найдем путь из S в T . Этот момент означает, что мы уже построили максимальный поток.)

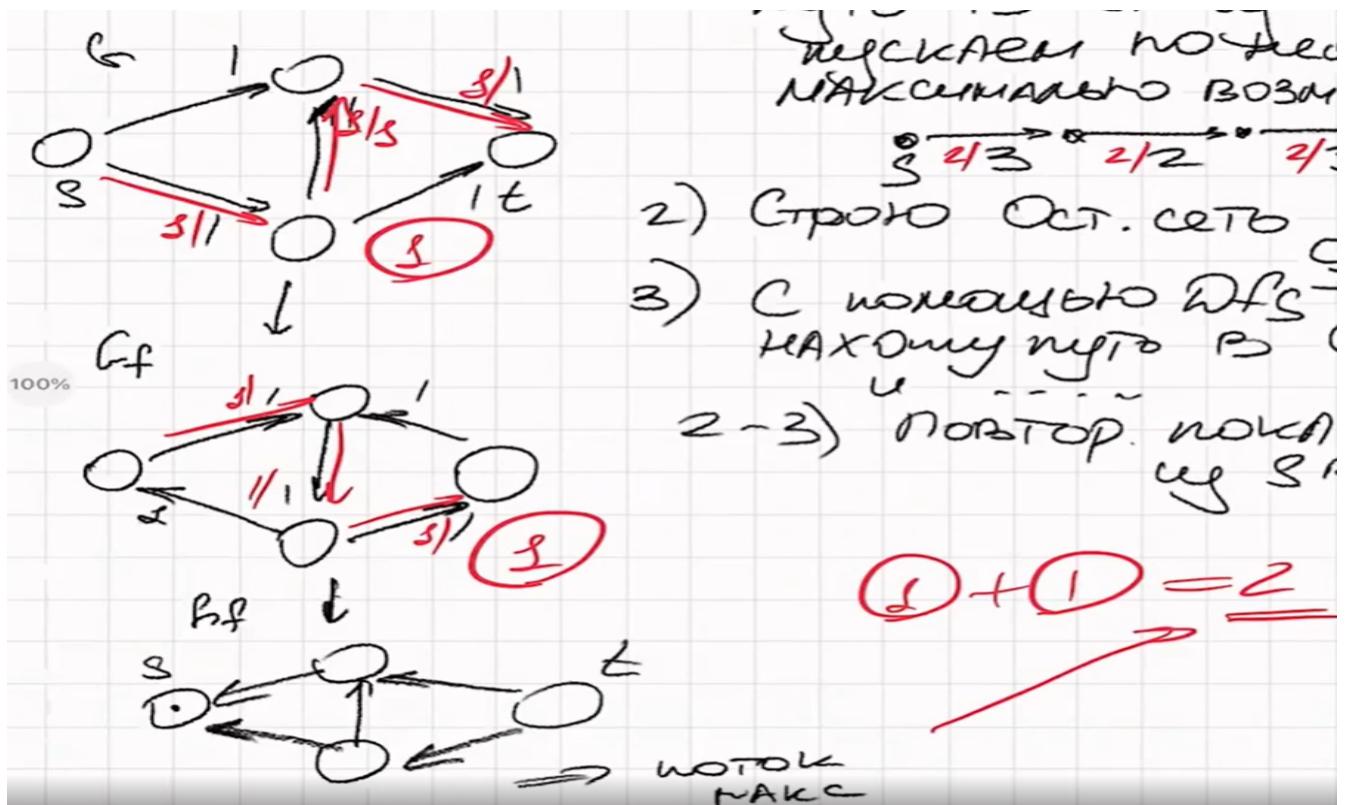
Доказательство:

Пусть поток f максимальен и есть путь в G_f из S в T . Тогда построим поток f' в G_f . По лемме 1 получаем, что $f + f'$ поток в G , причем $|f + f'| > |f|$. Получаем противоречие с тем, что поток f максимальен.

Пусть в G_f нет пути из S в T . Пусть f_{max} — максимальный поток, а f — поток, который нашли к данному моменту. Рассмотрим поток $f' = f_{max} - f$. По лемме 2 это поток в G_f , но раз в G_f нет пути из S в T , значит $|f'| \leq 0$. Значит $|f| \geq |f_{max}|$, то есть совпадает с максимальным потоком по величине $|f| = |f_{max}|$.

Алгоритм:

1. С помощью DFS находим путь в G из S в T и пускаем по нему максимально возможный поток (его величина равна минимальной пропускной способности ребра на пути).
 2. Строим остаточную сеть G_f $c_f = c - f$ (на семинарах говорилось, что то не обязательно).
 3. С помощью DFS находим путь в G_f из S в T и пускаем по нему максимально возможный поток.
- Пункты 2)-3) повторяются пока есть путь из S в T .



Время работы алгоритма: сложность DFS — $O(E)$ (считаем, что $E > V$), запускаем его несколько раз. В худшем случае количество раз — $O(|f|)$ (в худшем случае каждый раз поток увеличивается на единицу). Значит асимптотика $O(E \times |f|)$. Формально алгоритм нельзя считать аботающим за полиномиальное время, потому что он не работает за полиномиальное время от своих аргументов, а в нашем случае мы не знаем заранее величину $|f|$.

13.3 Алгоритм Эдмондса-Карпа

Меняем DFS из предыдущего алгоритма на BFS (поиск кратчайших путей). Это решает проблему со временем. Теперь ищем кратчайший путь из S в T и по нему пускать поток.

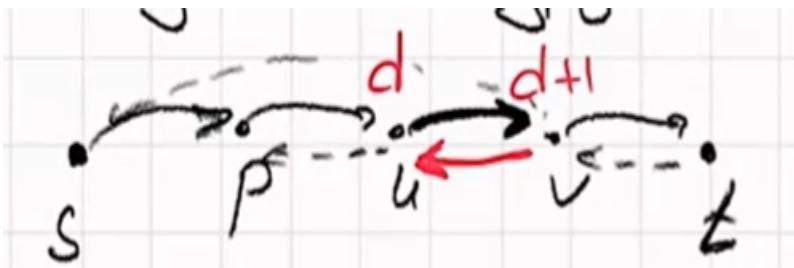
Время работы алгоритма: $O(V * E^2)$.

Корректность следует из теоремы Форда-Фалкерсона.

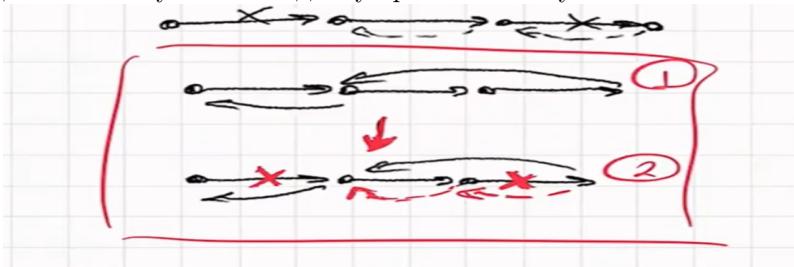
Лемма. На каждой операции алгоритма Эдмондса-Карпа кратчайшие пути от S до любой другой вершины не уменьшаются.

Доказательство:

1. Находим кратчайший путь из S в T . Пустим поток по этому пути. Если поток через ребро (u,v) неполный, в графе появляется ребро (v,u) , а ребро (u,v) остается в графе. Хотим доказать, что ребро (v,u) не будет принадлежать никакому кратчайшему пути. От противного пусть это не так, т.е. пусть (v,u) принадлежит кратчайшему пути из S в u . Пусть мы знаем, что расстояние от S до u равно d , значит расстояние от S до v тогда равно $d+1$. Если (v,u) принадлежит кратчайшему пути из S в u , значит расстояние по нему от S до v не меньше $d+1$, а до u $du = dv + 1 = d + 2 > d$. Значит (v,u) не лежит на кратчайшем пути от S до u , потому что полученное расстояние до u в таком случае увеличивается.
2. При удалении ребер кратчайший путь не уменьшается. Очевидно, т.к. если удалили ребро не на кратчайшем пути, то ничего не изменилось, а если убрали ребро на кратчайшем пути, то он точно не уменьшится, потому что мы уничтожили какой-то «хороший» кратчайший пути.



Почему применение обеих операций одновременно не может дать нового кратчайшего пути? Потому что из первого графа можно получить второй последовательным применением операций 1 и 2. Т.е. вместо одновременных действий в Эдмондсе-Карпе мы свели все к последовательность таких действий, а каждая из операций в отдельности не уменьшает длину кратчайшего пути.

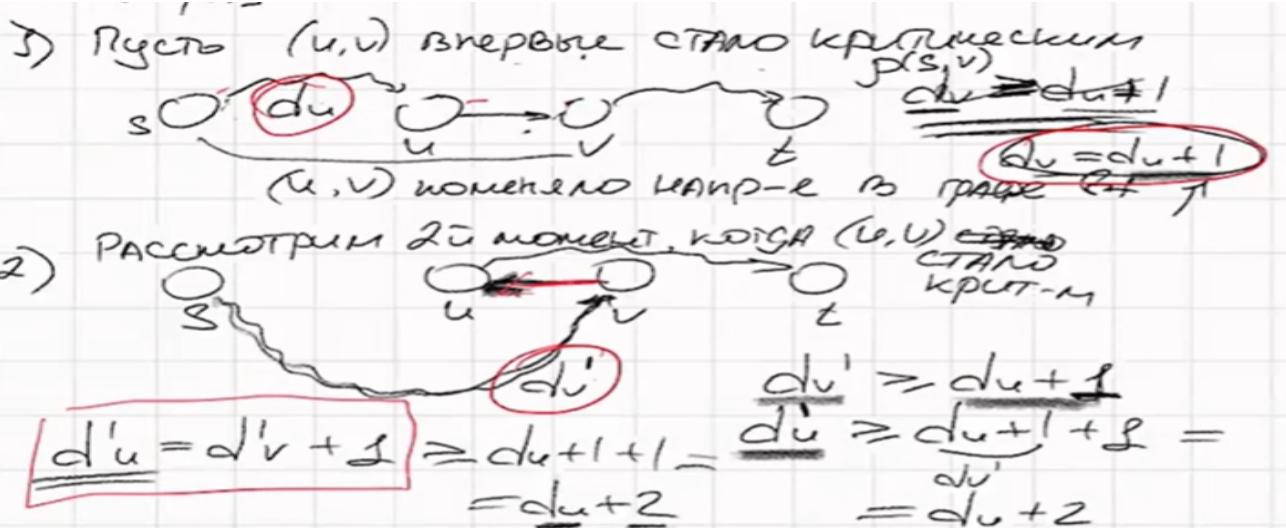


Теорема 13.2. Время работы алгоритма Эдмондса-Карпа $O(V \times E^2)$.

Доказательство:

Ребро $(u,v) \in E(G)$ — критическое, если по нему пускаем поток равный его максимальной пропускной способности. Покажем, что ребро может стать критическим не более V раз.

1. Пусть ребро (u,v) впервые стало критическим. Расстояние от S до u равно du . Про путь из S в v знаем, что его величина $dv = du + 1$, иначе бы du не было кратчайшим путем. Ребро меняет направление.
2. Рассмотрим второй момент, когда ребро (u,v) стало критическим. Оно меняет направление, когда находится на кратчайшем пути из S в T . Кратчайший путь выглядит так: есть путь из S в v dv' . В процессе работы с графом ребра только удаляются, значит путь не мог уменьшиться, тогда $dv' \geq dv = du + 1$ (следует из леммы, что в процессе работы алгоритма кратчайший путь не уменьшается), $dv' \geq dv' + 1 = du + 2$ потому что dv' — кратчайший путь, а (v,u) лежит на кратчайшем пути из S в T .



Таким образом расстояние до u может увеличиться не более $V/2$ раз, где V - количество вершин, потому что наибольшая длина в графе не может быть больше V . Значит $(u, v) \geq V/2$ раз критическое. Всего ребер в графе E . Каждый раз одно из ребер становится критическим, значит будет не более $V \times E/2$ операций. Значит время работы $T(BFS)n_{iter} \leq O(E) * V \times E \leq O(V \times E^2)$, где n_{iter} - число итераций.

Ещё существующие алгоритмы на потоках:

1. Алгоритм Диница за $O(V^2 \times E)$ — ищет все кратчайшие пути размера D и последовательно их обходит.
2. Алгоритм Трех индусов (имена Булат не назвал. Upd: Назвал. Upd: Назвал, но не на лекции. Их имена Malhotra, Kumar, Maheshwari) $O(V \times (V^2 + E))$ — оптимизация Диницы. Ищет сразу несколько коротких путей.

14 Кратчайшие пути из одной вершины. Алгоритм Форда-Беллмана. Алгоритм Дейкстры.

14.1 Кратчайший путь в графах

Постановка задачи: есть граф (неважно, ориентированный или нет). Есть вершина, называемая источником S , для всех ребер графа задана их длины (или вес), т.е. есть функция $w: E \rightarrow \mathbb{R}$ — вес ребра.

Задача. Single Source

Найти кратчайшие пути (КП) от S до любой другой вершины.

Note. BFS решает эту задачу: в графе, у которого все ребра весят по единице за $O(V + E)$, в 0-к графе за $O(kV+E)$. Проблема в том, что BFS работает только с целыми неотрицательными весами.

Лемма. Корректность постановки задачи о кратчайших путях

Задача поиска кратчайшего пути корректна тогда и только тогда, когда в графе нет отрицательных циклов.

Доказательство: В любом кратчайшем пути нет циклов, потому что:

1. Если бы был отрицательный цикл, его можно было бы бесконечное число раз уменьшать и исходный путь не был бы кратчайшим.
2. Если бы был положительный цикл, то значит его можно удалить из пути и путь станет только короче. Это противоречие с тем, что кратчайший путь найден корректно и его нельзя сделать меньше.

Пусть нет отрицательных циклов. Значит любой кратчайший путь из S в T не содержит циклов, потому что отрицательных нет, а положительные не имеют смысла (их можно удалить). Значит кратчайший путь обязательно простой. Всего простых путей из S в T конечное число, потому что он ограничено $V!$ (если граф полные, сначала выбираем первую вершину, потом вторую и т.д.). В конечном множестве всегда есть минимум, значит существует кратчайший путь из S в T .

14.2 Алгоритм Форда-Беллмана

Def. Пусть $d[u]$ и $d[v]$ — текущие оценки на кратчайшее расстояние от вершины S до u и v (т.е. мы нашли такие пути (необязательно кратчайшие) и их длина — оценка на кратчайшее расстояние). Тогда релаксация ребра (u,v) такая что, если $d[u] + w(u,v) < d[v]$, то мы можем обновить путь до v с помощью ребра (u,v) и $d[v] = d[u] + w(u,v)$

```
d[0...V-1]=∞  
d[s]=0  
for i in [0 to V-1]:  
    for (u,v) ∈ E:  
        Relax((u,v))
```

Псевдокод релаксации:

```
Relax((u,v)):  
    if d[v] > d[u]+w(u,v):  
        d[v] = d[u]+w(u,v)
```

Сложность алгоритма — $O(VE)$. Для разреженных графов он работает за $O(V^2)$, для плотных за $O(V^3)$.

Теорема 14.1 (Корректность алгоритма Форда-Беллмана). Алгоритм Форда-Беллмана корректно находит кратчайший путь за $V - 1$ шагов.

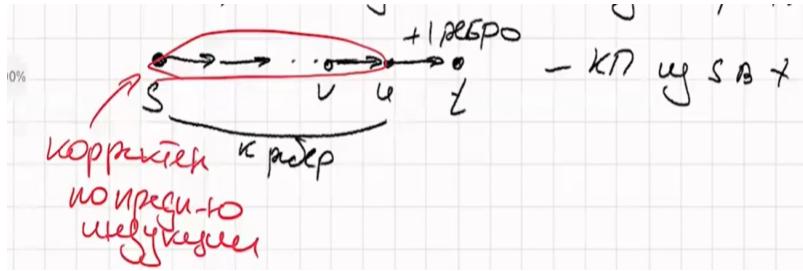
Доказательство: Индукция по длине (количеству ребер) пути из S в T .

База: $d[s] = 0$, потому что $\rho(s,s) = 0$.

Шаг индукции: Пусть найдены все кратчайшие пути из $\leq k$ ребер.

Докажем, что найдем все кратчайшие пути из $k + 1$ ребра:

Рассмотрим путь из S в t состоящий из $k + 1$ ребра. По предположению индукции путь из S в t корректен. На следующей итерации просмотрим все ребра (даже принадлежащие кратчайшим путям), в частности и ребро (u,t) . Так как оно принадлежит кратчайшему пути, то найдем кратчайший путь из S в t (успешно релаксируем).



Проверка на отрицательный цикл

```

for i in [0 to V-1]:
    for (u, v) ∈ E:
        Relax((u, v))
for (u, v) ∈ E:
    if(Relax((u, v))):
        Has_Negative_Cycle
    
```

Сложность алгоритма всё та же — $O(VE)$.

Теорема 14.2 (Критерий отрицательных циклов). *В G есть цикл отрицательного веса тогда и только тогда, когда на V -ой итерации произойдет релаксация $Relax$.*

Доказательство: Из теоремы о корректности алгоритма Форда-Беллмана следует, что раз на V -ой итерации произойдет релаксация $Relax$, то за $V - 1$ шаг мы не нашли все кратчайшие пути и задача некорректно поставлена. Такое может быть только в том случае, если в графе есть отрицательные циклы.

Теперь пусть в G есть цикл отрицательного веса и на V -ой итерации релаксации не произошло. Рассмотрим отрицательный цикл. Раз $Relax$ не было, то $d[v_2] \leq d[v_1] + w(v_1, v_2)$, $d[v_3] \leq d[v_2] + w(v_2, v_3)$ и т.д. $d[v_1] \leq d[v_n] + w(v_n, v_1)$. Если просуммировать все неравенства, получим $\sum_{i=1}^n d[v_i] \leq \sum_{i=1}^n d[v_i] + \sum_{i=1}^n w(v_i, v_{i+1})$, причем $v_{n+1} = v_1$.

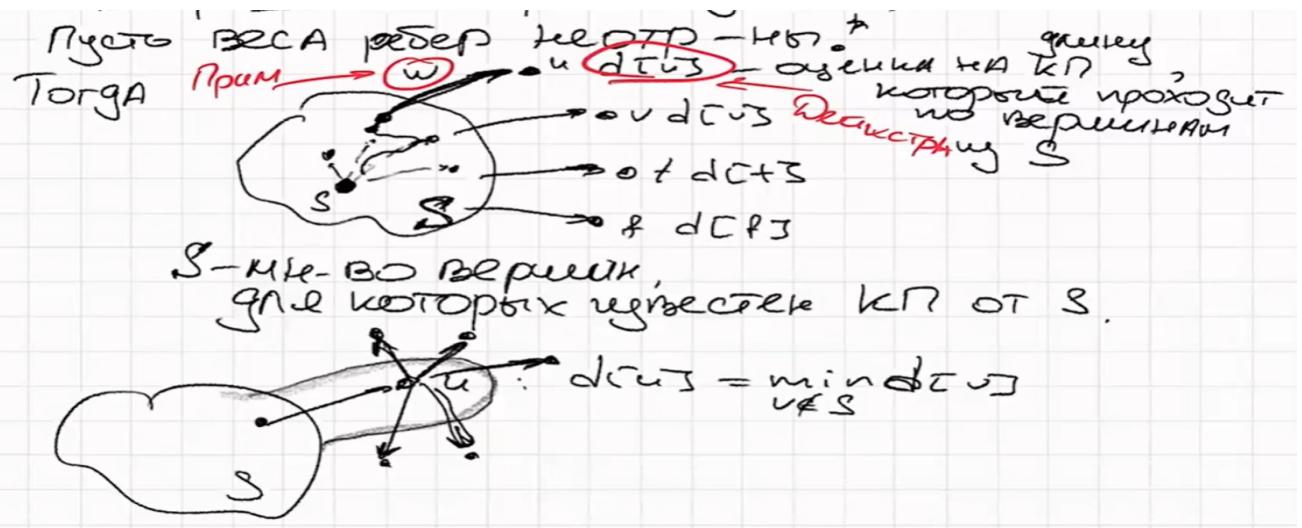
Сокращая, получаем $0 \leq \sum_{i=1}^n w(v_i, v_{i+1}) < 0$, потому что справа длина отрицательного цикла. Противоречие.

14.3 Алгоритм Дейкстры

Пусть веса ребер неотрицательны. Тогда пусть есть множество S вершин, для которых известен кратчайший путь от вершины s (понятно, что вершина s тоже лежит в этом множестве). Рассмотрим вершины не лежащие в этом множестве и ребра, ведущие к ним из s . Для каждой вершины есть оценка на длину кратчайшего пути, который проходит по вершинам из s . Выбираем вершину u с наименьшей оценкой и добавляем в множество S , релаксируем каждое ребро, выходящее из вершины u и продолжаем работу.

```

S={s}
d[0...V-1]=∞
d[s]=0
for i from 0 to V:
    Выберем v ∉ S, такую что d[v]-минимально
    Добавим v в S
    for (v, u) ∈ E:
        Relax(v, u)
    
```

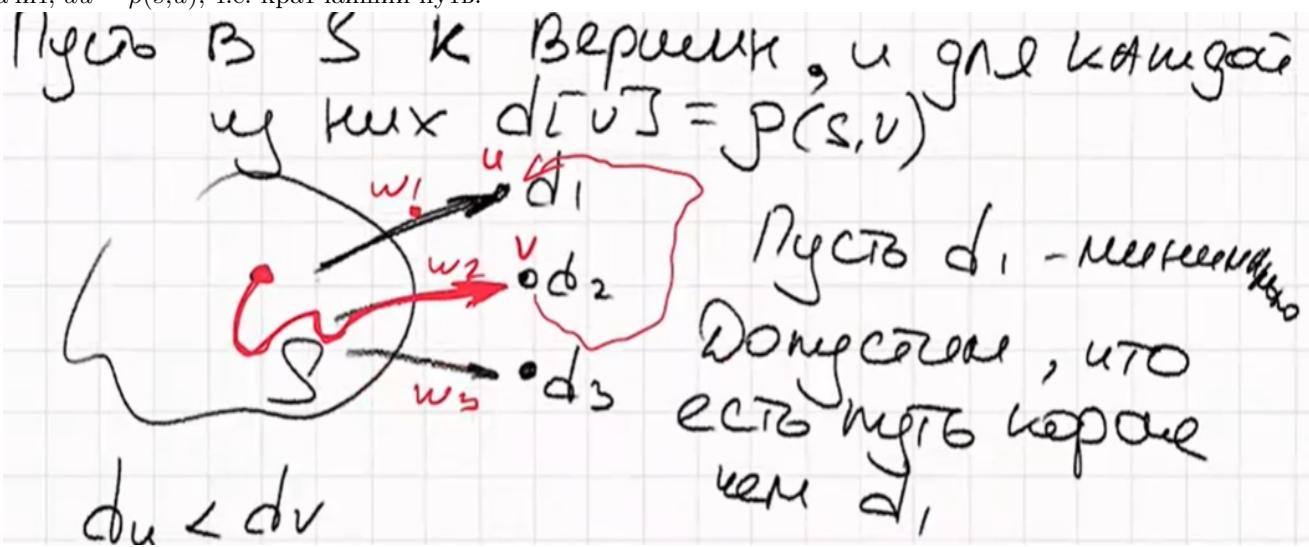


Теорема 14.3 (Корректность алгоритма Дейкстры). $\forall u \ d[u] = \rho(s, u)$ после алгоритма Дейкстры.

Доказательство:

Индукция по размеру S .

База $S = s$ $\rho(s, s) = 0 = d[s]$. Пусть в $S k$ вершин и для каждой из них $d[v] = \rho(s, v)$. Покажем, что мы корректно найдем расстояние и для других вершин. Пусть из всех путей d_1 ведущий в u — минимальный. Пусть есть путь в u короче, чем d_1 . Тогда мы выходим из множества, например, через ребро, соединяющее с вершиной v , до которой расстояние d_2 . Мы знаем, что $d_u < d_v$. Так как ребро w_2 лежит на кратчайшем пути от s до v , то d_v — кратчайший путь от s до v (очевидно, любой подпуть кратчайшего пути — кратчайший). $\rho(s, u) = d_v + \rho(v, u)$. Из неотрицательности ребер знаем, что $\rho(v, u) \geq 0$, тогда получаем $\rho(s, u) \geq d_v > d_u$. Получаем противоречие, т.к. считаем, что $\rho(s, u)$ — длина кратчайшего пути, а d_u — найденный путь, который проигнорировали, значит противоречие в предположении, что есть какой-то минимальный путь до u проходящий, например, через v . Значит, $d_u = \rho(s, u)$, т.е. кратчайший путь.



Сложность алгоритма.

Цикл с релаксацией, когда мы рассматриваем каждое ребро из добавленной вершины, работает за $O(E)$, потому что мы пройдем по всем ребрам. Если использовать бинарную кучу, то мы найдем минимум за $O(E \ln E)$ (построение минимального остовного дерева, как в алгоритме Прима), потому что при добавлении ребра мы тратим $O(\ln E)$. Тогда сложность $O(E \ln E)$, что на практике часто равно $O(E \ln V)$, потому что $E \sim V^\alpha$, где $\alpha \approx 1$ или 2 (для разреженных графов $V \ln V$, для плотных $V^2 \ln V$).

Note. Отрицательный цикл найти нельзя, т.к. нет отрицательных ребер **Note.** Если искать минимум линейным поиском по массиву, а не использовать кучу, то поиск минимального элемента будет за линию, т.е. за $O(V)$. Релаксация ребер будет за $O(E)$. Учитывая внешний цикл сложность будет $O(V^2 + E)$. Тогда для разреженных и плотных графов будет работать за $O(V^2)$, потому что в первом случае E порядка V , во втором порядка V^2 .

14.4 Улучшение алгоритма Форда-Беллмана

Не изменяем асимптотику, только улучшаем константу.

1. Если на какой-то итерации нет релаксации, то значит мы нашли все кратчайшие пути и нет смысла продолжать
2. Если на предыдущей итерации $d[v]$ не изменилась, то нет смысла рассматривать ребра, которые идут из v . То есть на каждой итерации рассматриваем только те вершины, для которых нашли новые кратчайшие расстояния, потому что тогда мы можем из неё что-то прорелаксировать

Алгоритм с такой идеей называется SPFA — Shortest Path Faster Algorithm.

```
d[0...V-1]=∞
d[s]=0
queue Q = {s} — очередь вершин с обновленным расстоянием
while (!Q.empty()):
    v = Q.pop()
    for (v,u) in G.E:
        if Relax(v,u) and u ∉ Q:
            Q.push(u)
```

Note.

1. SPFA по сути обобщение BFS (аналогии: relax — смотрим известно ли расстояние до вершины, очереди с обновленным расстоянием в обоих алгоритмах).
2. Худший случай $O(VE)$, но с лучшей константой и в среднем работает за $O(E)$ (в случае случайного графа).
3. Не работает с отрицательными циклами. В этом случае бесконечный while. С другой стороны, знаем, что Форд-Беллман за VE релаксаций находит все кратчайшие пути. Значит можно считать количество релаксаций и если их больше $(V - 1)E$ то есть отрицательный цикл.

15 Кратчайшие пути между всеми парами вершин. Алгоритм Флойда. Метод потенциалов. Алгоритм Джонсона.

Тупое решение задачи AllPairs — запуск алгоритма Форда–Беллмана или Дейкстры V раз (V — количество вершин). Асимптотика в таком случае будет $O(V \cdot VE)$ (Форд–Беллман), $O(V \cdot E \log V)$ (Дейкстры с кучей, выгоден на разреженных графах) или $O(V \cdot (V^2 + E))$ (Дейкстры с линейным поиском минимума, выгоден на плотных графах).

15.1 Алгоритм Флойда.

Заводим множество матриц $\{d^i \mid 0 \leq i \leq V - 1\}$, где $d^k(u, v)$ — кратчайший путь из u в v , в котором в качестве промежуточных можно использовать только вершины с номерами не превышающими k (для удобства нумерацию вершин начинаем с 1).

d^0 — матрица весов ребер, так как в ней не может быть никаких промежуточных вершин на пути, то есть длина кратчайшего пути между вершинами $< \infty \Leftrightarrow$ между ними есть ребро.

$d^{k+1}(u, v) = \min(d^k(u, v), d^k(u, k+1) + d^k(k+1, v))$, но так как $d^k(u, k) = d^{k-1}(u, k)$ и $d^k(k, v) = d^{k-1}(k, v)$, можно обойтись лишь одной матрицей, тогда алгоритм будет выглядеть следующим образом (w — матрица весов ребер):

```

d = w
for k from 1 to V:
    for u from 1 to V:
        for v from 1 to V:
            d(u, v) = min(d(u, v), d(u, k) + d(k, v));
    
```

Эта версия требует $O(1)$ дополнительной памяти и $O(V^3)$ времени.

Утверждение. Вершина $x \in c^-$, где c^- — отрицательный цикл $\Leftrightarrow d^V(x, x) < 0$.

Доказательство. Очевидно. ■

15.2 Метод потенциалов.

Def. $\phi : V \rightarrow \mathbb{R}$ — потенциал.

Определим граф $G_\phi = (G.V, G.E)$, но $w_\phi(x, y) = w(x, y) + \phi(x) - \phi(y)$ (меняется вес ребер).

Лемма. Пусть P — путь в графе G , а P' — эквивалентный в графе G_ϕ (то есть проходит в точности по тем же ребрам), тогда

$$\rho_{P'}(u, v) = \rho_P(u, v) + \underbrace{\phi(u) - \phi(v)}_{\text{не зависит от } P}$$

Доказательство.

$$\begin{aligned} \rho_{P'}(u, v) &= \sum_{e \in P'} w_\phi(e) = \sum_{e \in P} (w(e) + \phi(e.\text{from}) - \phi(e.to)) = \\ &= \sum_{e \in P} w(e) + \sum_{e \in P} (\phi(e.\text{from}) - \phi(e.to)) = \rho_P(u, v) + \phi(u) - \phi(v) \end{aligned}$$

Следствие 1: Если P — кратчайший путь в G , то P — кратчайший путь в G_ϕ , так все одинаковые пути в G и G_ϕ отличаются на константу.

Следствие 2: Веса циклов в G_ϕ не меняются относительно G , так как если $u = v$, $\phi(u) - \phi(v) = 0$.

Теорема 15.1. $\exists \phi \forall e \in G_\phi.E \quad w(e) \geq 0 \Leftrightarrow \neg \exists \text{ отрицательных циклов в } G$.

Доказательство. $\bullet \Rightarrow$

Веса всех циклов в $G_\phi \geq 0$, по следствию 2 все циклы в G также неотрицательны.

$\bullet \Leftarrow$

Добавим в граф вершину s^* и пусть из нее ребра с весом 0 во все вершины. Определим $\phi(v) = \rho(s^*, v) \leq 0$. Новые веса ребер: $w_\phi(u, v) = w(u, v) + \phi(u) - \phi(v) = w(u, v) + \rho(s^*, u) - \rho(s^*, v)$, но $w(u, v) + \rho(s^*, u) — длина$

некоторого пути из s^* в v (проходящего через u), а $\rho(s^*, v)$ — длина кратчайшего пути из s^* в v , значит, $w(u, v) + \rho(s^*, u) \geq \rho(s^*, v) \Leftrightarrow w(u, v) + \rho(s^*, u) - \rho(s^*, v) \geq 0$. ■

15.3 Алгоритм Джонсона.

1. Заводим вершину s^* , проводим ребра нулевого веса ко всем остальным вершинам.
2. Запускаем алгоритм Форда-Беллмана из s^* .
3. Строим $\phi(v) = \rho(s^*, v)$.
4. На полученном графе V раз запускаем алгоритм Дейкстры.

Асимптотика такого алгоритма полностью определяется асимптотикой реализации алгоритма Дейкстры и составляет $O(V \cdot E \log V)$ на разреженных графах и $O(V \cdot (V^2 + E))$ на плотных.

16 Эвристический поиск кратчайшего пути между парой вершин. Алгоритм A*. Условие оптимальности алгоритма (б/д).

Алгоритм A* является модификацией алгоритма Дейкстры, оптимизированной для случая, когда нужно найти кратчайшее расстояние от одной конкретной вершины до другой конкретной вершины. Одна из оптимизаций заключается в том, что вводится функция $h : V \rightarrow \mathbb{R}$, которая по поданной на вход вершине возвращает минимальную оценку расстояния от нее до вершины, путь в которую мы ищем (целевой). Естественно, значение этой функции должно вычисляться за $O(1)$.

Def. Эвристическая оценка $h(v)$ допустима, если для любой вершины v значение $h(v)$ не превосходит кратчайшего пути от v до целевой вершины.

Def. Эвристическая оценка $h(v)$ монотонна (преемственна), если для любой вершины v_1 и любого ее потомка v_2 разность $h(v_1)$ и $h(v_2)$ не превышает фактического веса ребра от v_1 к v_2 , а эвристическая оценка целевой вершины равна нулю.

Примеры эвристических функций:

- Если вершины графа – это некоторые точки координатной плоскости, а ребра имеют вес, равный расстоянию между его концами (в привычной евклидовой метрике), то в качестве функции h вполне логично взять расстояние от поданной на вход вершины до целевой в геометрическом смысле ($h(v) = \sqrt{(t_x - v_x)^2 + (t_y - v_y)^2}$, где t – целевая вершина).
- Если на этой плоскости все ребра горизонтальны или вертикальны, можно взять “манхэттенское” расстояние ($h(v) = |t_x - v_x| + |t_y - v_y|$).
- Если добавляются ребра, направленные по диагонали под углом 45° , можно взять расстояние Чебышева ($h(v) = \max |t_x - v_x|, |t_y - v_y|$).

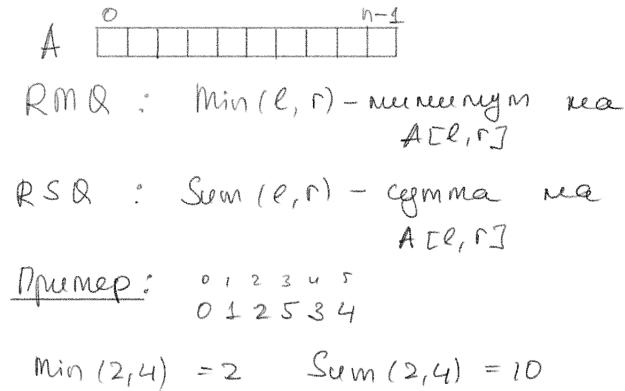
В качестве приоритета вершины v в алгоритме Дейкстры выступает минимальное расстояние $d(u)$ от старовой вершины до u , в алгоритме A* в качестве приоритета выбирается $d(u) + h(u)$. В остальном алгоритм совпадает с алгоритмом Дейкстры за исключением того, что A* завершается сразу по достижении целевой вершины (вторая логичная модификация, ведь без нее никакого выигрыша по времени не будет, так как все равно будут рассмотрены все вершины).

Условие оптимальности алгоритма: h монотонна \Rightarrow алгоритм A оптимален.*

17 Задачи static RMQ/RSQ. Префиксные суммы. Разреженная таблица. Дерево отрезков.

17.1 Static RMQ/RSQ

Def. *RangeMinimumQuery* и *RangeSumQuery*



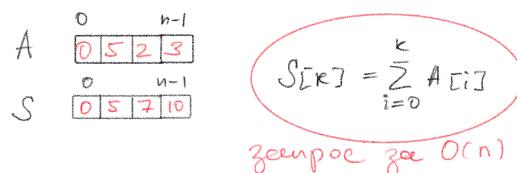
Пояснительный рисунок 1.

Note. Если решать в лоб, то каждый из запросов работает за $O(\text{length}(A))$, что очень долго.

Note. Массив НЕ меняется.

17.1.1 static RSQ Префиксные суммы.

Используется массив частичных (кумулятивных) сумм (*PartialSumArray* or *CumSumArray*) или же *PrefixSum* - префиксные суммы.



Пояснительный рисунок 2.

$$\text{Sum}(l, r) = \sum_{i=l}^r A[i] = S[r] - S[l-1]$$

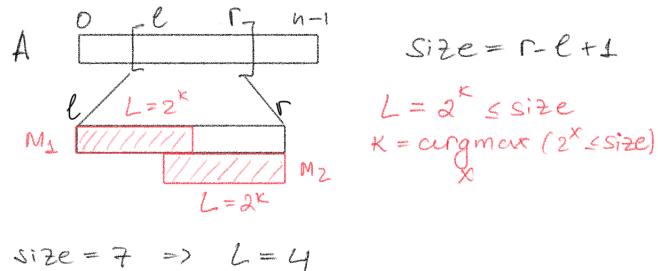
Каждый такой запрос работает за $O(1)$

17.1.2 static RMQ Разреженная таблица.

Такая задача уже не получится решить с помощью массива частичных сумм (так как минимум - необратимая операция, а для решения задач таким образом операция должна быть ассоциативна и обратима); **Note.** Слово дня - идемпотентность. Это свойство операции следующего вида:

$$x \oplus x = x$$

Минимум - идемпотентная операция.



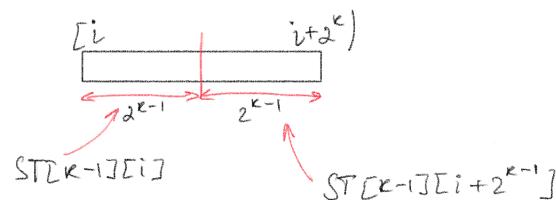
Пояснительный рисунок 3.

Минимум на всем отрезке = $\min(A[i, \dots, i + 2^k - 1])$. Идея:
 Посчитаем минимум на всех отрезках длины вида 2^k .
 $2^k = 1$ отрезков n штук
 $2^k = 2$ отрезков $n - 1$ штука
 \dots
 $2^k \leq n$ таких отрезков 1 или 2
 На каждом таком уровне $O(n)$ памяти, а всего уровней $\log n \rightarrow$ всего $O(n \log n)$ памяти

Def. Sparse Table: $ST[k][i] = \min(A[i, \dots, i + 2^k - 1])$ - это минимум на отрезке длины 2^k с началом в i

17.1.3 Заполнение таблицы

$ST[0][i] = \min(A[i, \dots, i + 2^0 - 1]) = A[i]$
 Пусть $ST[k-1][i]$ посчитано $\forall i \rightarrow ST[k][i] = \min(ST[k-1][i], ST[k-1][i + 2^{k-1}])$



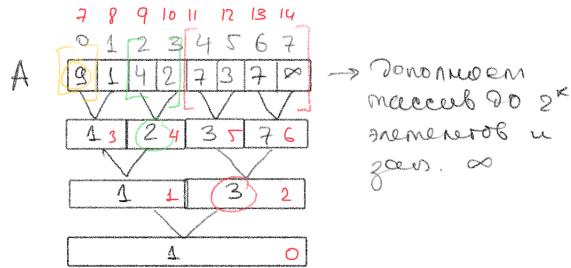
Пояснительный рисунок 4.

Query(l, r):
 $k = \lfloor \log r - l + 1 \rfloor$
 return $\min(ST[k][l], ST[k][r - 2^k + 1])$

17.1.4 Segment tree или Дерево отрезков

Умеет работать с обоими RMQ и RSQ, поддерживает операции изменения массива

17.1.5 Построение дерева отрезков.



Пояснительный рисунок 5.

1. Дополняем A до $2^k = n$
2. Заводим массив длины $2n - 1$
3. $T[n - 1, \dots, 2n - 1] = A // leaves$
4. for ($i = n - 1$ to 0):
 $T[i] = \text{Min}(T[\text{Left}(i)], T[\text{Right}(i)])$ //any other operation instead of min can be used

Утверждение. На каждом уровне ≤ 2 фундаментальных отрезка **Доказательство:** Допустим их хотя бы 3 → тогда хотя бы один не фундаментальный - центральный! Потому что можно перейти в родителя и получить лучшее разбиение → противоречие.

Note. Всего фундаментальных узлов $< 2 \log n$ или $O(\log n)$

17.1.6 Как искать фундаментальные узлы?

1. Запрос от листьям к корню

```
RMQ(l, r):
    l += n - 1
    r += n - 1
    left_res = 0
    right_res = 0
    while (l < r):
        if (l == Right(Parent(l))):
            left_res = min(left_res, T[left])
        l = Parent(l + 1)
        if (r == Left(Parent(r))):
            right_res = min(right_res, T[right])
        r = Parent(r - 1)
    if (l == r):
        left_res = min(left_res, T[left])
    return min(left_res, right_res)
```

18 Задачи dynamic RMQ/RSQ. Дерево отрезков с обновлением элемента. Дерево отрезков с обновлением на подотрезке. Многомерное дерево отрезков. Дерево Фенвика. Многомерное дерево Фенвика.

18.1 Dynamic RMQ/RSQ

Dynamic RMQ/RSQ отличается от static тем, что элементы массива меняются(есть запрос Update).

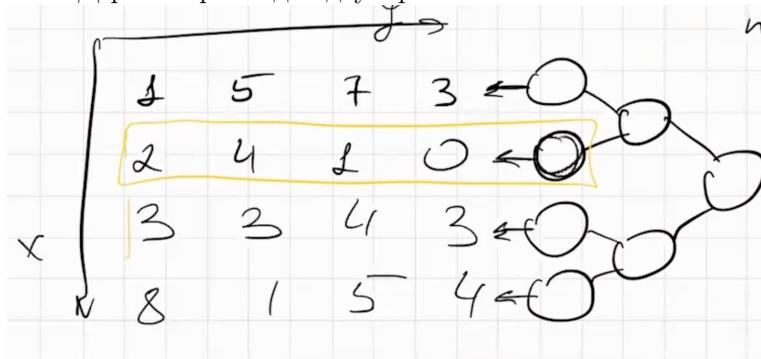
18.2 Дерево отрезков с обновлением элемента

Изменяя некоторый элемент мы изменяем листовой элемент дерева, а он в свою очередь влияет на своего родителя. Таким образом, поднимаясь от листа к корню мы по сути меняем логарифмическое число элементов. Выполняется за логарифмическое время.

```
Update(i, new_value):
    i+= n-1
    tree[i]=new_value
    while(i != parent(i)):
        i = parent(i)
        tree[i] = tree[left(i)] (+) tree[right(i)]
```

18.3 Многомерное дерево отрезков

Это экстраполяция задачи RMQ/RSQ на многомерный случай. Например найти минимум площадии в двумерном массиве. Дерево отрезка для двумерного массива:



Каждый лист соответствует одному ряду, и будет содержать внутри себя дерево отрезков для этого ряда. Например второй сверху лист содержит в себе дерево [2,4,1,0]. Более высокоуровневые элементы включающие в себя два листа, содержат в себе дерево отрезков из минимальных элементов этих двух рядов.



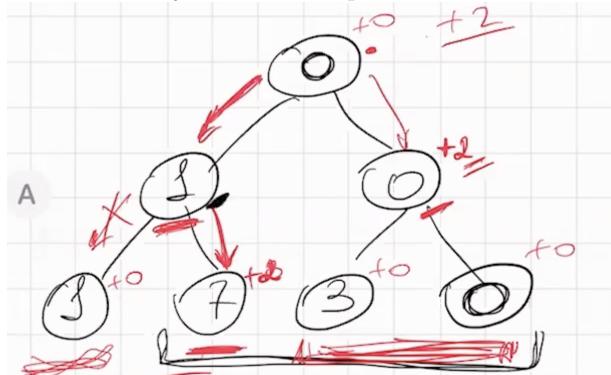
Для двумерного массива размера ($M \times N$) потребует $O(n^*m)$ памяти и времени построения, и на запрос и обновление $O(\log n * \log m)$.

18.4 Дерево отрезков с обновлением на подотрезке

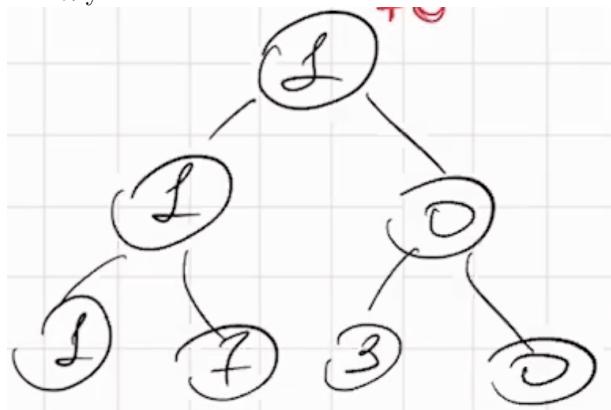
Бывают задачи когда нужно узнать минимум на отрезке или изменить элементы на отрезке. Последовательно изменять элементы не эффективно ($n \log n$), поэтому используем отдельные алгоритмы для группового изменения.

Пример

Например на отрезке $[1,7,3,0]$ хотим увеличить $[7,3,0]$ на два. Изменяем на нужное число фундаментальные узлы отвечающие за нужный нам отрезок.



И после изменений пересматриваем "родителей чье значение равно минимальному от новых значений "детей". Получим:



Псевдокод

Лор: Promise это изменения которые должны произойти на отрезке, те то на какую величину должны изменить элементы. Value - значение элемента дерева.

Push - это операция выполнения обещания.

(*) - операция Update (ассоциативна)

(+) - операция Query (дистрибутивна по Update $\rightarrow (a(+b)(*)c = (a(*)c) (+) (b(*)c))$)

```
Push(node):
    node.left.promise (*) = node.promise          //your left child should promise what you promise
    node.right.promise (*) = node.promise          //right child too
    node.value (*) = node.promise                 //your value increases by the size of the promise
    node.promise = 0                                //0 - neutral element
```

left, right - границы подотрезка на котором мы ищем минимум

Реализация Query:

```
Query(node, left, right):
    [a,b] = node.borders
    if ([a,b] dont cross [left,right]):
        return 0
    if ([a,b] in [left,right]):
        return node.value (*) node.promise
    else ([a,b] cross [l,r]):
        Push(node)
        return Query(node.left, left, right) (+)
            Query(node.right, l, r)
```

Реализация Update:

```
Update(node, l, r, Delta):
    [a,b] = node.borders
    if ([a,b] dont cross [l,r]):
        return
    if ([a,b] in [l,r]):
        node.promise (*) = Delta // increasing by Delta
    else:
        Push(node)
        Update(node.left, l, r, Delta)
        Update((node.right, l, r, Delta))
        node.value = Query(node.left) (+) Query(node.right)
```

18.5 Дерево Фенвика

Дерево Фенвика - структура данных требующая $O(n)$ памяти и позволяющая за $O(\log n)$ выполнять следующие операции:

1. изменять значение любого элемента в массиве
2. выполнять некоторую ассоциативную, коммутативную, обратимую операцию на отрезке

Решает задачу dynamic RSQ. Менее функционально, чем Дерево отрезков, нет групповых обновлений, но быстро пишется.

Допустим у нас есть массив a_0, a_1, \dots, a_{n-1} длины n нам надо посчитать сумму на подотрезке $(0, m)$. Построим такой массив $T : T[i] = \sum_{k=F(i)}^i a_k$. Где $F : \mathbf{R} \rightarrow \mathbf{R}$, $F(i) \leq i$.

В качестве $F(m)$ возьмем функцию которая заменяет последние единицы в бинарной записи числа на ноль. **Def. FenwickFunction:** $F(m) = m \& (m + 1)$, где $\&$ - битовое "и".

18.5.1 Query

Реализация Query $(0, m)$

```
Query(m):
    res = 0
    for (i=m; i >= 0; i = F(i) - 1)           // F(i) = i & (i+1)
        res += T[i]
    return res
```

$$\text{Query}(l, r) = Q(r) - Q(l-1)$$

18.5.2 Update

Рассмотрим как изменяется массив T при изменении элемента a_k .

Note. Для пересчёта дерева Фенвика при изменении величины a_k необходимо изменить элементы дерева T_i , для индексов i для которых верно неравенство $F(i) \leq k \leq i$.

Пр. Необходимо менять те $T[i]$ в которые попадает a_k , а значит необходимые i удовлетворяют условию $F(i) \leq k \leq i$.

Note. Все такие i , для которых меняется $T[i]$ при изменении a_k , можно найти по формуле $i_{m+1} = i_m \parallel (i_m + 1)$, $i_0 = k$.

12 ЛЕКЦИЯ 1:13

Построим возрастающую последовательность по i , которые надо обновлять. $i_0 = k$.

Допустим есть i_m , хотим i_{m+1} .

$$i_m = x01\dots1$$

$$k = x0? \dots ?$$

$$F(i_m) = x00..0$$

$$x_m >= x_{m+1} \text{ (иначе } F(i_{m+1}) > k\text{)}$$

получается увеличиваем за счет первого 0:

$i_{m+1} = x_m$???, но если где-то у нас будет 0, то тогда $F(i_{m+1}) = x_m$???

Получается i_{m+1} это i_m с заменой последнего 0 на 1: $i_m = i_m | (i_m + 1)$

Те чтобы обновить значение a_k достаточно обновить значение $i_0 = k$, и дальше все элементы преобразовать по формуле выше.

Реализация Update

```
Update(k, D):
    for (i=k, i<n, i=i | (i+1)):
        T[i] += D
```

18.6 Многомерное дерево Фенвика

Например двумерное дерево: $T[i,j] = \sum_{i=F(x)}^x \sum_{j=F(y)}^y a[x,y]$. Где внутренняя сумма - одномерное дерево Фенвика. Те в каждом элементе дерева - одномерное дерево Фенвика.

18.6.1 Query

```
Query(x, y):
    res = 0
    for (i=x, i<=0, i = i & (i+1) - 1):
        for (j=y, j<=0, j = j & (j+1) - 1):
            res += T[i][j]
    return res
```

Для двумерного случая для произвольного прямоугольника с координатами главной диагонали $(x_0, y_0), (x, y)$ поправилу исключений: $Q(\text{прямоугольника}) = Q(x, y) - Q(x_0, y) - Q(x, y_0) + Q(x_0, y_0)$.

18.6.2 Update

```
Update(x, y, Delta):
    for (i=x, i<n, i = i | (i+1)):
        for (j=y, j<m, j = j | (j+1)):
            T[i][j] += Delta
```

19 Декартово дерево. Основные операции. Анализ времени работы. Эффективные алгоритмы построения.

19.1 Декартово дерево(Cartesian Tree).

19.1.1 Binary Search Trees(BST).

Note. HashTable делает $Insert(x)$, $Erase(x)$, $Find(x)$ за амортизированную константу $O(\alpha + 1)$
 BST делает $Insert(x)$, $Erase(x)$, $Find(x)$ за $O(\log n)$

Def. Binary Tree - в левом поддереве корня r все элементы $< x$, а в правом $\geq x$

```
Find(tree, x):
    if (root == nullptr) return false
    if (root->x0 < x):
        Find(root->right, x)
    elif (root->x0 > x):
        Find(root->left, x)
    else: // root->x0 == x
        return true
```

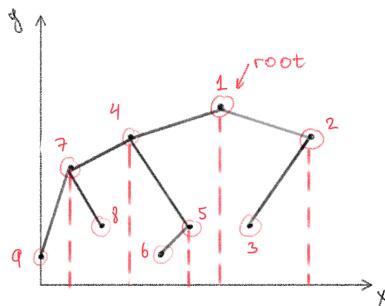
Def. BST - сбалансированное, если его глубина $O(\log n)$, то есть $\leq const * \log n$

Note. Несбалансированное BST - бамбук

Def. Cartesian Tree - дерево, которое хранит пары (x, y) и является BST по x и неполной бинарной кучей (или пирамидой) по y .

То есть $y_i > y_{i+1}$ и $y_i > y_{i+2}$, поэтому y называется приоритетом - в корне самый приоритетный элемент. Полукуча в силу того, что элементы отсутствуют не только на последнем уровне.

Утверждение. Если $\forall i, j: x_i \neq x_j, y_i \neq y_j$, то Декартово дерево строится однозначно.

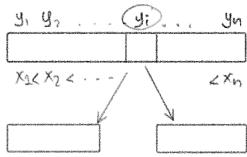


Пояснительный рисунок 1.

Note. Также называют Дерамида, дучи или курево (*Treap*).

Пусть $\{y_i\}_{i=0}^{n-1}$ - независимые, одинаково распределенные случайные величины, такие что $y_i \neq y_j$ и y не зависит от x . Тогда средняя глубина Декартового дерева - $O(\log n)$.

Решение. Так как приоритеты н. о. р. и не зависят от x , то каждый узел может стать корнем дерева с вероятностью $1/n$. Выбор корня \leftrightarrow выбор pivot в QuickSort. Рекурсивно строим левую и правую части. Так как в QuickSort - средняя глубина рекурсии $O(\log n)$, то по аналогии средняя глубина Декартового дерева тоже $O(\log n)$.



Пояснительный рисунок 2.

19.1.2 Эффективные алгоритмы построения.

Note. Используем Split/Merge. Дерево представляем в виде:

```

struct Node{
    keyT x;
    PrT y;
    Node* left;
    Node* right;
    Node* parent;
}
Treap = Node* root;

Setleft(node, left):
    if (node != nullptr):
        node -> left = left;
    if (left != nullptr):
        left -> parent = node;

Split(T, x0):
    if (T == nullptr):
        return(nullptr, nullptr)
    if (root -> x < x0):
        (T1, T2) = Split(T -> right, x0)
        Setright(T, T1)
        return (T, T2)
    elif (root -> x ≥ x0):
        (T1, T2) = Split(T -> left, x0)
        Setleft(T, T2)
        return (T1, T)

Merge(T1, T2):
    if (T1 == nullptr):
        return T2
    if (T2 == nullptr):
        return T1
    if (T1 -> y < T2 -> y):
        Setleft(T2, Merge(T1, T2 -> left))
        return T2
    else:
        Setright(T1, Merge(T1 -> right, T2))
        return T1
  
```

Note. Split и Merge работают в среднем за $O(\log n)$ **Note.** Все другие операции выражаются через Split and Merge:

```

Erase(T, x):
    T1, T2 = Split(T, x)
    T12, T22 = Split(T2, x + 0)
    //x + 0 == min x1 : x1 > x
    delete T12
    return Merge(T1, T22)
  
```

```

Insert(T, (x, y)):
    T0 = (x, y)
    T1, T2 = Split(T, x)
    return Merge(Merge(T1, T0), T2)

```

Note. Erase и Merge работают в среднем за $O(\log n)$

Build($\{x_i, y_i\}_{i=0}^{n-1}$):

```

root = nullptr
for (x, y) in {x_i, y_i}:
    root = Insert(root, (x, y))

```

Note. В среднем $O(n \log n)$. В худшем - $O(n^2)$

Если $\{x_i, y_i\}_{i=0}^{n-1}$ отсортирован по x . Тогда: Build($\{x_i, y_i\}_{i=0}^{n-1}$). Далее merge пар 0-1, 2-3 и так далее, merge новых пар и так пока не останется один массив, который и будет Treap. В худшем случае за $O(n \log n)$.

Build($\{x_i, y_i\}_{i=0}^{n-1}$):

```

root = nullptr
lastAdded = nullptr
for (x, y) in {x_i, y_i}:
    while ((lastAdded != nullptr) and (lastAdded->y < y)):
        lastAdded = lastAdded->parent
    if (lastAdded == nullptr):
        Setleft((x, y), root)
        root = (x, y)
    else:
        Setleft((x, y), lastAdded->right)
        Setright(lastAdded, (x, y))
    lastAdded = (x, y)

```

Note. В худшем случае - $O(n)$

20 Решение задач RMQ/RSQ с помощью декартова дерева. Декартово дерево по неявному ключу.

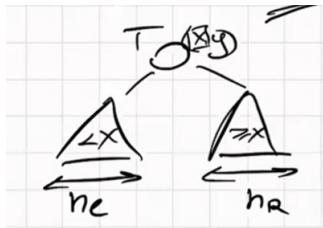
Рассмотрим несколько дополнительных возможностей декартовых деревьев

20.1 Индексация в декартовом дереве

$T \equiv \text{CartesianTree}$

Хотим иметь: $T[i] = x(i)$ - i -я порядковая статистика

Пусть дано следующее декартово дерево T , заданное своим корнем:



где n_L, n_R - кол-во элементов в левом и правом поддеревьях соответственно.

Заметим, что x тогда будет n_L -м элементом в порядке возрастания.

Чтобы поддерживать данную индексацию в узлах дерева необходимо дополнительно хранить информацию о том, какой размер поддерева храниться в данной вершине. Во-первых, в таком случае в узел следует добавить поле *size*.

Во-вторых, дописать вспомогательный метод в классе:

```
size_t size(node):
    if node is None:
        return 0
    else:
        return node->size
```

Поддерживается свойство:

```
size(node) == 1 + size(node->left) + size(node->right)
```

Ранее все операции с декартовыми деревьями выполнялись с помощью комбинирования *Split* и *Merge*. Если проанализировать их исполнение, то можно заключить, что на размер поддеревьев влияют только операции "подвешивания" *SetLeft* и *SetRight*, а значит для поддержания индексации необходимо переписать только их.

Обновлять информацию в узлах будем при помощи метода:

```
def updateNode(node):
    if node is not None:
        node->size = 1 + size(node->left) + size(node->right)
```

А теперь модернизируем ранее реализованные методы:

```
def setLeft(node, left):
    if node is not None:
        node->left = left
    if left is not None:
        left->parent = node
        updateNode(left)           ←

def setRight(node, right):
    if node is not None:
        node->right = right
    if right is not None:
        right->parent = node
        updateNode(right)         ←
```

Теперь мы научились корректно сохранять размер любого поддерева.

Вспомним, что мы стремились к индексации, то есть к возможности вычислять i -ю порядковую характеристику. Теперь мы в состоянии написать соответствующий метод:

```

value_type get(T, idx):
    if T is None:
        return 0
    if idx is size(T->left):
        return T->x
    elif size(T->left) < idx:
        return get(T->right, idx - (size(T->left) + 1))
    else: # size(T->left) > idx
        return get(T->left, idx)

```

В среднем время - $\mathcal{O}(\log N)$

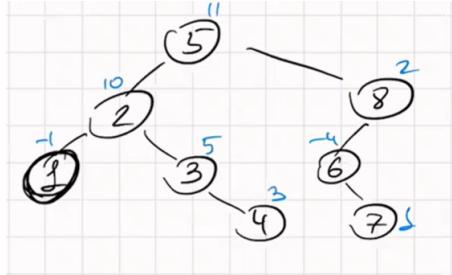
20.2 RMQ/RSQ

В дереве отрезков RMQ(L, R) - запрос минимума на элементах $x : L \leq x < R$

Добавим в узел поле *value*(получим подобие std::map)

Теперь $T[x] == \text{value}$.

Рассмотрим пример:



$$\text{RMQ}([2, 6]) = 3$$

$$\text{RSQ}([3, 8)) = 16$$

Для запросов на Д.Д. используется схожий с Д.О. подход. Для реализации нам понадобится ещё одно дополнительное поле в узле *result*, которое будет хранить результат операции на поддереве. Модифицируем метод *updateNode()*:

```

def updateNode(node):
    if node is not None:
        node->size = 1 + size(node->left) + size(node->right)
        node->result = Op(result(node->left), ←
                           node->value, result(node->right)) ←

```

где:

- Op - операция суммы, минимума, etc.
- result() - аналог size(), который мы реализовали ранее

Наконец реализуем метод запроса:

```

value_type RMQ(T, [L, R)):
    T_∞L, T_L∞ = split(T, L)
    T_LR, T_R∞ = split(T_L∞, R)
    answer = result(T_LR)
    T = merge(T_∞L, merge(T_LR, T_R∞))
    return answer

```

Note. правый предел всегда строгий, левый строгий только если $-\infty$. ∞ слева обозначает $-\infty$.

20.3 Групповое обновление

Обновление значений узлов происходит при помощи метода *update([L, R], delta)* - дистрибутивной по запросу операции, $x : L \leq x < R$. Т.е. в результате меняются не x , а *value*, лежащие по ключам x .

Т.е.:

$$\Rightarrow (a \oplus b) \odot c = (a \odot c) \oplus (b \odot c)$$

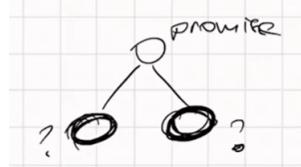
- \odot - update

- \oplus - query

Кроме того, добавим в узел поле *promise* - отложенную операцию обновления.

```
def push(node):
    if node is not None:
        if node->left is not None:
            node->left->promise ⊕= node->promise
        if node->right is not None:
            node->right->promise ⊕= node->promise
    node->value = node->value ⊕ node->promise
    node->result = node->result ⊕ node->promise
    node->promise = 0
```

Однако может возникнуть следующий *вишневый* случай:



Если мы захотим обратиться к одному из таких потомков, получим ложный ответ. Чтобы исключить эту ситуацию, нужно модифицировать методы *merge()*, *split()*, *setLeft()*, *setRight()*, *result()*:

```
value_type result(node):
    if node is not None:
        return node->result ⊕ node->promise ←

def merge(T1, T2):
    push(T1) ←
    push(T2) ←
    if T1 is None:
        return T2
    if T2 is None:
        return T1
    if T1->y < T2->y:
        setLeft(T2, merge(T1, T2->left))
        return T2
    else:
        setRight(T1, merge(T1->right, T2))
        return T1

def split(T, x0):
    push(T) ←
    if T is None:
        return nullptr, nullptr
    if root->x < x0:
        T1, T2 = split(T->right, x0)
        setRight(T, T1)
        return T, T2
    else # root->x ≥ x0:
        T1, T2 = split(T->left, x0)
        setLeft(T, T2)
        return T1, T

def setLeft(node, left):
    push(node) ←
    if node is not None:
        node->left = left
    if left is not None:
        left->parent = node
    updateNode(node)

def setRight(node, right):
    push(node) ←
    if node is not None:
        node->right = right
```

```

if right is not None:
    right->parent = node
updateNode(node)

```

И, наконец, *update()*:

```

def update(T, [L, R], delta):
    T∞L, T∞l = split(T, L)
    TLR, T∞r = split(T∞l, R)
    TLR->promise ⊕= delta
    merge(T∞L, merge(TLR, T∞r))

```

20.4 Д. Д. по неявному ключу

Давайте удалим из дерева все ключи \Rightarrow получим динамический массив со следующими свойствами:

1. insert - $\mathcal{O}(\log N)$
2. erase - $\mathcal{O}(\log N)$
3. get - $\mathcal{O}(\log N)$
4. RMQ/RSQ - $\mathcal{O}(\log N)$
5. update - $\mathcal{O}(\log N)$
6. shift - $\mathcal{O}(\log N)$ (циклический сдвиг на k позиций)
 $shift(k) \equiv split(k) \rightarrow T_L, T_R \rightarrow merge(T_R, T_L)$

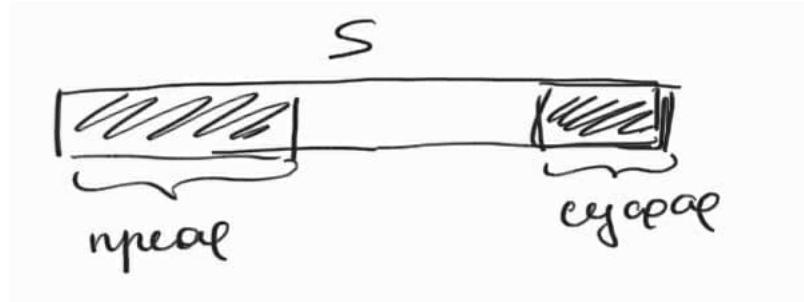
21 Поиск подстроки в строке. Префикс-функция. Z-функция. Алгоритм Кнута-Морриса-Пратта. Построение префикс-функции по Z-функции. Построение лексикографически наименьшей строки по префикс функции.

21.1

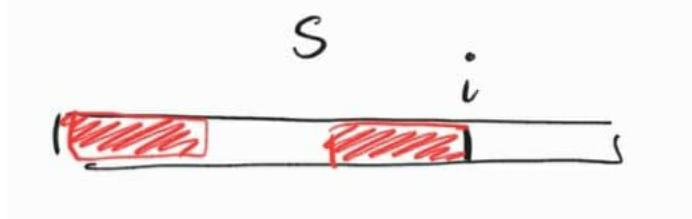
Задача поиска подстроки в строке. Пусть у нас есть строка S , подстрока P , задача найти такие i, j , что $S[i..j] = P$. Для самого наивного алгоритма - пробегаемся по всем элементам асимптотика будет $O(|P|(|S|-|P|))$. Рассмотрим далее более эффективный алгоритм для которого нужно будет ввести несколько понятий.

21.2 Префикс-функция

Def. Префикс-функция — ф-я от строки S и позиции $i (< n)$ такая, что $p(S, i) =$ длина наибольшего собственного префикса ($\neq S$), который совпадает с суффиксом $S[0...i]$.



Пояснительный рисунок 1.



Пояснительный рисунок 2.

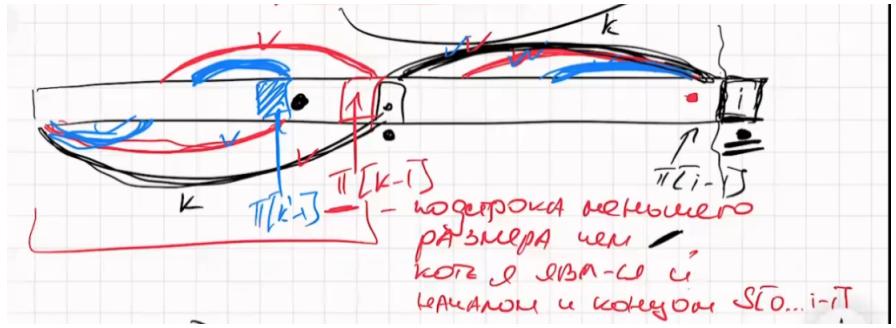
То есть (см рисунок 2) мы ищем максимальную длину, чтобы красные участки совпали. Пример: $S = abcababca \rightarrow p = 000121234$.

Понятно, что есть тривиальный алгоритм за $O(n^3)$.

Построим эффективный алгоритм.

Note. Заметим, что $p[i+1] \leq p[i] + 1$. Чтобы показать это, рассмотрим суффикс, оканчивающийся на позиции $i+1$ и имеющий длину $p[i+1]$, удалив из него последний символ, мы получим суффикс, оканчивающийся на позиции i и имеющий длину $p[i+1]-1$, следовательно неравенство $p[i+1] > p[i]+1$ неверно.

Note. Избавимся от явных сравнений строк. Пусть $k = p[i-1]$. Пусть мы вычислили $p[i-1]$, тогда, если $s[i] = s[p[i-1]]$, то $p[i] = p[i-1] + 1 = k + 1$. Если окажется, что $s[i] \neq s[p[i-1]]$, то нужно попытаться попробовать подстроку меньшей длины. Делаем это следующим образом. За исходное k необходимо взять $p[k-1]$. В случае, когда символы $s[k]$ и $s[i]$ не совпадают, $p[k-1]$ следующее потенциальное наибольшее значение k , что видно из рисунка. Последнее утверждение верно, пока $k > 0$, что позволит всегда найти его следующее значение. Если $k = 0$, то $p[i] = 1$ при $s[i] = s[1]$, иначе $p[i] = 0$.



Пояснительный рисунок 3.

```
int [] prefixFunction(string s):
p[0] = 0
for i = 1 to s.length - 1
    k = p[i - 1]
    while k > 0 and s[i] != s[k]
        k = p[k - 1]
    if s[i] == s[k]
        k++
    p[i] = k
return p
```

Утверждение. Время работы алгоритма - $O(n)$

Доказательство: Для доказательства этого нужно заметить, что итоговое количество итераций цикла while определяет асимптотику алгоритма. Теперь стоит отметить, что k увеличивается на каждом шаге не более чем на единицу, значит максимально возможное значение $k = n - 1$. Поскольку внутри цикла while значение k лишь уменьшается, получается, что k не может суммарно уменьшиться больше, чем $n - 1$ раз. Значит цикл while в итоге выполнится не более n раз, что дает итоговую оценку времени алгоритма $O(n)$.

21.3 Алгоритм Кнута-Морриса-Пратта

Это алгоритм поиска подстроки в строке.

1. $S' = P + \# + S$ // - элемент не присутствующий в P и S
2. $p(S')$ (или $z(S')$)
3. находим все позиции, где $p = |P|$.

Асимптотика $O(|S| + |P|) = O(|S|)$

21.4 Z-функция

Def. Z-функция — ф-я от строки S и позиции $i (< n)$ такая, что $z(S, i) =$ длина наибольшего префикса $S[i..n-1]$, который совпадает с префиксом строки S .

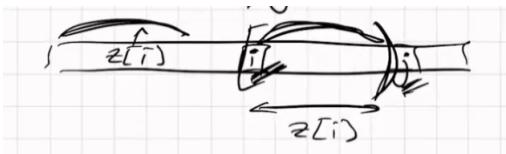
$z(S, 0) = 0$, либо $|S|$;



Пояснительный рисунок 4.

Понятно, что тривиальный можно за $O(n^2)$.

Введем понятие z-блока — пара (i, j) такая что $z[i] = j - i$.

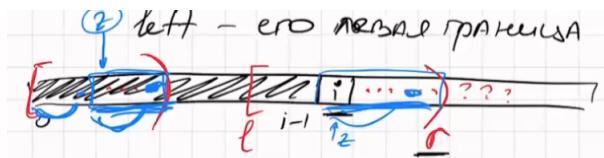


Пояснительный рисунок 5.

Эффективный алгоритм:

Пусть right — самая правая граница всех z-блоков, left — его левая.

Пусть мы посчитали все до $z[i]$.



Пояснительный рисунок 6.

Если $right > i$, то мы знаем что синие подстроки совпадают. Значит мы можем взять в качестве начального значения значение $z[\text{начало синего блока}]$, то есть $z[i] = \min(z[i - left], right - i)$ (\min так как мы не знаем ничего про элементы, которые выходят за границу красного блока).

Далее считаем просто по определению.

```
int [] zFunction(s : string):
int [] zf = int[n]
int left = 0, right = 0
for i = 1 to n - 1
    zf[i] = max(0, min(right - i, zf[i - left]))
    while i + zf[i] < n and s[zf[i]] == s[i + zf[i]]
        zf[i]++
    if i + zf[i] > right
        left = i
        right = i + zf[i]
return zf
```

21.4.1 Время работы

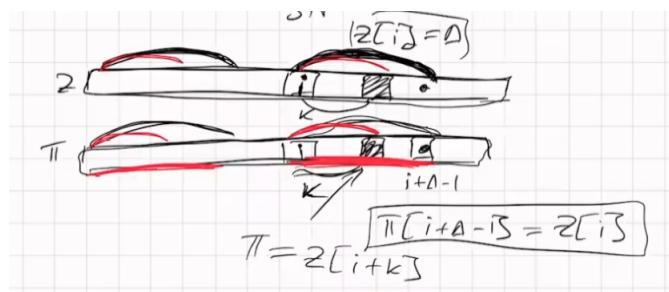
На любой итерации мы либо за единицу находим ответ, либо чекаем новые элементы, значит сдвигаем right. Но right не может сдвинуться более $|S|$ раз. Получается $O(|S|)$.

21.5 Z-функция → Префикс

Пусть у меня есть z функция, как тогда узнать префикс-ф.

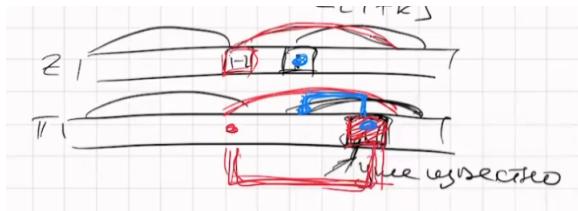
$p[i + \Delta - 1] = z[i]$;

Но также мы знаем значение для позиций внутри нашего черного отрезка.



Пояснительный рисуночек 7.

Причем, нам больше не нужно будет его обновлять.



Пояснительный рисуночек 8.

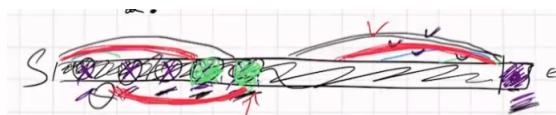
Посмотрим на рисунок. Мы вначале нашли одинаковые красные отрезки, для них проинициализировали значения, затем нашли черные. Рассмотрим позицию внутри, для нее мы при красных отрезках нашли значение, соответствующее красному блоку, а сейчас хотим обновить до синего, но оно заведомо меньше, значит обновлять бессмысленно.

```
int[] buildPrefixFunctionFromZFunction(int[] z) {
    int[] p = int[z.length]
    fill(p, 0)
    for i = 1 to z.length - 1
        for j = z[i] - 1 downto 0
            if p[i + j] > 0
                break
            else
                p[i + j] = j + 1
    return p
}
```

Так как присваиваем значения мы только один раз, а иначе выходим, то $O(|S|)$.

21.6 Prefix → str

Построим строку по префикс функции причем с наименьшим лексикографическим значением. Допустим мы построили $s[0..i-1]$. Рассмотрим $p[i] = k > 0$, тогда $s[i] = s[k-1]$. Иначе нам нужно добавить такой символ, который не продолжает не один из р-блоков — введем множество used и возьмем наименьший символ, не хранящийся в нем.



Пояснительный рисуночек 9.

```
int[] Prefix2Str(int[] z):
    s = ""
    s[0] = a;
    for int i = 0 to |p| - 1 {
        if (p[i] > 0) {
            s[i] = s[p[i] - 1]
        } else {
            std::set used = {};
            k = p[i - 1];
            while(k > 0) { // 
                used.add(s[k]);
                k = p[k - 1];
            }
            for (sym = 'b' to 'z') { // b
                if (sym not in used) {
                    s[i] = sym;
                    break;
                }
            }
        }
    }
```

```
    }  
}  
return s;
```

Асимптотика — $O(|S|)$.

Заметим что теперь мы можем взять два из трех (префикс-ф, z-ф, строку) и получить одного из другого, тк у нас есть цикл превращений $p \rightarrow str \rightarrow z \rightarrow p$