# Курс

# "Алгоритмы и структуры данных"

2 семестр ФПМИ МФТИ

Даниил Гагаринов

Артур Кулапин

github.com/yaishenka

github.com/kulartv

Весна 2019

# Содержание

# Графы. Алгоритмы обхода

## Понятия графов

**Def.** Граф — упорядоченная пара $G(V, E)$, где $V$ — множество вершин, а $E \subset (V \times V)$ — ребра. Далее будем считать, что $|V| < \infty$.

**Def.** Мультиграф — граф, в котором хотя бы одна пара вершин соединена более чем одним ребром.

**Def.** Псевдограф — мультиграф, в котором присутствуют петли.

**Def.** Граф ориентированный, если для каждого ребра задана его ориентация, то есть для каждого ребра есть стартовая и конечная вершины.

**Def.** Две вершины смежны, если они соединены ребром.

**Def.** Две вершины $u, v$ достижимы, если найдется такая цепь из вершин $v, v_1 \dots v_n, u$, что $v_i$ смежна с $v_{i+1}$, а также смежны $v, v_1$ и $v_n, u$. Для ориентированного графа также необходимо, чтобы $v_i$ была стартовой вершиной, а $v_{i+1}$ была конечной.

**Def.** Неориентированный граф связен, если для произвольной вершины все остальные вершины достижимы из нее.

**Def.** Назовем неориентированной копией $G'$ ориентированного графа $G$ граф, построенный на тех же вершинах и ребрах, но ориентация ребер отсутствует.

**Def.** Ориентированный граф слабо связный, если он сам не связен, но связна его неориентированная копия.

**Def.** Ориентированный граф сильно связен, если для произвольной вершины все остальные вершины достижимы из нее.

**Def.** Компонента слабой (сильной) связности в графе $G$ — подграф $G'$, который слабо (сильно) связен.

## Алгоритмы, производные от DFS

**Algorithm.** Обход в глубину (DFS)

1. Выбираем любую вершину из еще не пройденных.
2. Помечаем эту вершину, как пройденную.
3. Для каждой смежной вершины повторяем шаги 1 и 2.

**Def.** Цвета вершин во время DFS:

- Белый — вершина еще не была посещена
- Серый — вершина находится в процессе обхода
- Черный — для данной вершины пройдены все смежные

**Def.** Назовем временами входа и выхода для вершины $u$ пару чисел $entry[u], leave[u]$. Массивы $leave, entry$ будем заполнять в ходе модифицированного DFS.

**Algorithm.** Модифицированный DFS

Изначально все вершины белые.

1. Заведем переменную $time = 0$.
2. Выберем произвольную вершину $u$, $entry[u] = time$
3. Красим вершину $u$ в серый цвет, $++time$.
4. Для каждой белой вершины, смежной с $u$ запускаем DFS.
5. Красим $u$ в черный цвет, $++time$, $leave[u] = time$.

**Def.** Дерево обхода, типы ребер

Рассмотрим подграф предшествования обхода в глубину $G_p(V, E_p)$, где $E_p = \{(p[u], u) : u \in V, p[u] \neq NIL\}$, где в свою очередь $p[u]$ — вершина, от которой был вызван $dfs(u)$ (для вершин, от которых $dfs$ был вызван нерекурсивно это значение соответственно равно $NIL$). Подграф предшествования поиска в глубину образует лес обхода в глубину, который состоит из нескольких деревьев обхода в глубину. С помощью полученного леса можно классифицировать ребра графа $G$.

- Ребро древесное, если оно лежит в $G_p$. (ребро вниз)

- Ребро обратное, если оно соединяет вершину и ее предка (для неориентированного графа предок не должен быть родителем). (ребро вверх)
- Ребро прямое, если оно не древесное и соединяет вершину с ее потомком (в неориентированном графе прямое ребро является обратным). (длинное ребро вниз)
- Остальные ребра перекрестные. (ребро в сторону)

**Th.** Лемма о белых путях

Пусть дан граф $G$. Запустим $dfs(G)$. Остановим выполнение процедуры $dfs$ от какой-то вершины $u$ в тот момент, когда вершина $u$ была выкрашена в серый цвет (назовем его первым моментом времени). Заметим, что в данный момент в графе $G$ есть как белые, так и черные, и серые вершины. Продолжим выполнение процедуры $dfs(u)$ до того момента, когда вершина $u$ станет черной (второй момент времени). Тогда вершины графа $G \setminus u$, бывшие черными и серыми в первый момент времени, не поменяют свой цвет ко второму моменту времени, а белые вершины либо останутся белыми, либо станут черными, причем черными станут те, что были достижимы от вершины $u$ по белым путям.

**Доказательство:** Черные вершины останутся черными, потому что цвет может меняться только по схеме белый, серый, черный. Серые останутся серыми, потому что они лежат в стеке рекурсии и там и останутся.

Заметим, что не существует такого момента в процессе обхода, что существует ребро из черной $v$ вершины в белую $u$. Действительно, запустим $dfs(v)$. В этот момент $v$ стала серой, а $u$ была белой. Далее будет запущен $dfs(u)$, так как $u$ была белой. По алгоритму вершина $v$ будет покрашена в черный цвет тогда, когда завершится обход всех вершин, достижимых из нее по одному ребру, кроме тех, что были рассмотрены раньше нее. Таким образом, вершина $v$ может стать черной только тогда, когда $dfs$ выйдет из вершины $u$, и она будет покрашена в черный цвет. Получаем противоречие.

Теперь заметим, что если вершина достижима по пути из белых вершин в первый момент времени, то она стала черной ко сторому моменту времени (из абзаца выше следует).

Заметим, что это верно и в обратную сторону. Рассмотрим момент, когда вершина $v$ стала черной: в этот момент существует серый путь из $u$ в $v$, а это значит, что в первый момент времени существовал белый путь из $u$ в $v$.

Отсюда следует, что если вершина была перекрашена из белой в черную, то она была достижима по белому пути, и что если вершина как была, так и осталась белой, она не была достижима по белому пути, что и требовалось доказать.

**Algorithm.** Проверка неориентированного графа на связность.

Модифицируем $DFS$ так, чтобы он возвращал число посещенных вершин. Тогда запустим его от произвольной вершины, и если возвращенное число равно $|V|$, то граф связный, иначе — нет.

**Algorithm.** Проверка графа на цикличность

Будем решать задачу с помощью поиска в глубину.

В случае ориентированного графа произведём серию обходов. То есть из каждой вершины, в которую мы ещё ни разу не приходили, запустим поиск в глубину, который при входе в вершину будет красить её в серый цвет, а при выходе из нее — в чёрный. И, если алгоритм пытается пойти в серую вершину, то это означает, что цикл найден.

В случае неориентированного графа, одно ребро не должно встречаться в цикле дважды по определению. Поэтому необходимо дополнительно проверять, что текущее рассматриваемое из вершины ребро не является тем ребром, по которому мы пришли в эту вершину.

**Algorithm.** Топологическая сортировка

Топологическая сортировка ориентированного ациклического графа $G(V, E)$ представляет собой упорядочивание вершин таким образом, что для любого ребра $(u, v) \in E$ номер вершины $u$ меньше номера вершины $v$.

Предположим, что граф ацикличен, т.е. решение существует. Что делает обход в глубину? При запуске из какой-то вершины $v$ он пытается запуститься вдоль всех рёбер, исходящих из $v$. Вдоль тех рёбер, концы которых уже были посещены ранее, он не проходит, а вдоль всех остальных — проходит и вызывает себя от их концов.

Таким образом, к моменту выхода из вызова $DFS(v)$ все вершины, достижимые из $v$ как непосредственно (по одному ребру), так и косвенно (по пути) — все такие вершины уже посещены обходом. Следовательно, если мы будем в момент выхода из $DFS(v)$ добавлять нашу вершину в начало некоего списка, то в конце концов в этом списке получится топологическая сортировка.

## Нахождение компонент сильной связности

**Algorithm.** Алгоритм Косарайю

1. Строим граф $H$ с инвертированными ребрами
2. Выполним $DFS$ на $H$, вычисляющий для каждой вершины время выхода $DFS$ из нее (этот массив обозначим за $f$).
3. Выполняем $DFS$ на исходном графе, перебирая вершины в порядке убывания $f(u)$.

**Доказательство:** Докажем, что вершины $s, t$ взаимно достижимы тогда и только тогда, когда после выполнения алгоритма они принадлежат одному дереву обхода.

$\Rightarrow$

Если вершины $s, t$ были взаимно достижимы в графе $G$, то на третьем этапе будет найден путь из одной вершины в другую, это означает, что по окончанию алгоритма обе вершины лежат в одном поддереве.

$\Leftarrow$

1. Вершины $s, t$ лежат в одном и том же дереве поиска в глубину на третьем этапе алгоритма. Значит, что они обе достижимы из корня $r$ этого дерева.

2. Вершина $r$ была рассмотрена вторым обходом в глубину раньше, чем $s$ и $t$, значит время выхода из нее при первом обходе в глубину больше, чем время выхода из вершин $s$ и $t$. Из этого мы получаем 2 случая:

   - Обе эти вершины были достижимы из $r$ в $H$. А это означает взаимную достижимость вершин $s, r$ и взаимную достижимость вершин $r, t$. А складывая пути мы получаем взаимную достижимость вершин $s, t$.
   - Хотя бы одна не достижима из $r$ в $H$, например $t$. Значит и $r$ была не достижима из $t$ в $H$, так как время выхода $r$ больше . Значит между этими вершинами нет пути, но последнего быть не может, потому что $t$ была достижима из $r$ по пункту 1.

   Значит $s, t$ достижимы в обоих графах.

**Algorithm.** Алгоритм Тарьяна нахождения компонент сильной связности

Во-первых вспомним, что если мы попали в компоненту сильной связности через вершину $v$ (назовём такую вершину корневой вершиной её компоненты сильной связности), то после выхода из этой вершины, вся её компонента будет обойдена.

Таким образом все вершины из той же компоненты сильной связности будут лежать в поддереве с корнем в $v$. Было бы здорово, если б мы умели как-то определять, что мы находимся именно в такой вершине.

Тогда наш алгоритм будет работать так:

1. Заводим стек и когда в процессе обхода в глубину впервые заходим в вершину, кладём её в этот стек.

2. Когда мы завершаем обработку какой-то вершины $v$, мы проверяем, является ли она первой обработанной из своей компоненты сильной связности.

3. Если является, то мы вынимаем из стека все вершины вплоть до $v$ и объявляем их одной компонентой. И это будет соответствовать действительности.

**Note.** В данном алгоритме $d$ — массив времен входа, а $f$ — массив времен выхода

Там будут все вершины из данной компоненты, так как все они потомки $v$, и там не будет ничего лишнего. Действительно, если встретилась вершина u из другой компоненты, то это значит, что она находится где-то в поддереве, начинающемся в $v$, вместе с вершиной $w$, через которую наш обход

зашёл в эту компоненту (иначе, $w$ предок $v$ и существует цикл, содержащий $w$, $v$ и u, то есть все они лежат в одной компоненте). А это значит, что вершину u мы вытащим из стека, когда закончим обработку вершины $w$.

Чтобы определять, является ли вершина $v$ корневой,будем считать в каждой вершине число $l[v] = min(d[v], l[u], d[w])$, где $u$ пробегает по всем вершинам, в которые есть рёбра дерева из $v$, а $w$ пробегает по всем вершинам, находящимся в данный момент в стеке, в которые из $v$ ведут обратные и перекрёстные рёбра. Эта величина считается в момент окончания обработки вершины $v$. Как только мы видим, что $l[v] = d[v]$, мы говорим, что вершина $v$ — корневая, и начинаем вынимать вершины из стека.

**Утверждение.** Докажем индукцией по времени окончания обработки, что:

1. В стеке нет вершин из полностью чёрных компонент
2. $l[v] \geqslant d[r]$, где $r$ — корень компоненты, содержащей $v : r = root(v)$.

**Доказательство:** $l[v] = min(d[v], l[u], d[w])$, докажем сначала второе утверждение:

1. По определению массива времен входа $d$, $d[v] \geq d[r]$
2. Заметим, что $f(u) < f(v)$ — времена выхода. Если $u \in SCC(v)$, то $l[u] \geq d[root(u)] = d[r]$. Если же $u \notin SCC(v)$, то $u = root(u)$, тогда $l[u] \geq d[u]$, но $d[u] \geq d[v] \geq d[r]$, то есть $l[u] \geq d[r]$
3. $w$ — в стеке, значит либо она сама серая, либо она чёрная, и тогда, по предположению (1) в стеке есть серая вершина $t$ из той же компоненты. Значит, есть путь, начинающийся в $v$, проходящий через $w$, затем $t$ и возвращающийся обратно в v (раз $t$ серая, то она предок $v$ по дереву обхода). Значит, $w \in SCC(v)$, откуда $d[w] \geq d[r]$.

Таким образом, получаем, что $l[v] \geq d[r]$.

Докажем первую часть утверждения: до окончания обработки вершины $v$ в ее поддереве не может быть ни одной полностью черной компоненты. Теперь заметим, что полностью черной компонента может стать только если $v$ была последней необработанной (не черной) вершиной в своем поддереве, а это возможно если она была корневой для своей компоненты. Тогда $l[v] \geq d[v]$, но в ходе алгоритма получим, что мы обошли все ее поддерево, откуда нетрудно понять, что $l[v] = d[v]$.

Тогда по алгоритму именно в этот момент мы начинаем раскручивать стек и получать компоненту связности, за которую отвечает вершина $v$. А значит если $v$ — корневая, то мы вытащим всю компоненту связности, за которую она отвечает.

**Следствие** Если вершина $v$ — корневая, то наш алгоритм выкинет из стека все вершины этой компоненты в момент окончания обработки $v$.

Осталось доказать, что алгоритм не начнёт выкидывать вершины, если $v$ — не корневая. То есть, что если $l[v] = d[v]$, то $v = root(v)$.

**Утверждение.** Докажем индукцией по времени окончания обработки следующее утверждение:

1. Если $l[v] = d[v]$, то $v = root(v)$
2. От вершины стека до текущей вершины в стеке лежат только вершины из той же компоненты
3. Вершина вынимается из стека только когда заканчивается обработка корня соответствующей компоненты.

**Доказательство:**

**(1)** Пусть $u = root(v) \neq v$. В момент обработки $v$, $u$ — серая. На пути из $v$ в $u$, найдём первую вершину $t$, не являющуюся потомком $v$ в дереве обхода, а за $w$ обозначим вершину, которая ей предшествует на этом пути. Ребро $(w, t)$ — обратное (если $t$ — серая) или перекрёстное (если t — чёрная), в любом случае $d[t] < d[v]$. И в обоих случаях $t$ лежит в стеке. Если $t$ серая, то она в стеке, так как выше текущей в стеке лежат только чёрные вершины, откуда следует, что из стека не вынимаются серые вершины. Если $t$ чёрная, то она в стеке по предположению (3). Так как $w$ потомок $v$ в дереве обхода, то существует путь из $v$ в $w$ по рёбрам дерева, а значит $l[v] \leqslant d[t] < d[v]$.

**(2)** Так как мы вынимаем из стека вершины от текущей до самого верха, то над текущей вершиной может лежать только один из её сыновей. Если он из другой компоненты, то он сам и всё что выше него должны были быть убраны в момент окончания его обработки. Значит он из той же компоненты, что и текущая вершина. А всё что выше него из той же компоненты по предположению индукции

**(3)** Пусть обработав вершину v мы решили, что пора вынимать вершины из стека. Тогда во-первых, из (2) следует, что выше по стеку будут лежать только вершины из $SCC(v)$, так что вершины из других компонент мы не вытащим, а во-вторых, из (1) следует, что $v$ — корень $SCC(v)$, то есть наш алгоритм работает ровно так как мы хотели, и, в частности, выполняется (3).

## Реберная двусвязность

**Def.** Две вершины реберно двусвязны, если между ними существуют два пути без общих ребер.

Нетрудно показать, что отношение реберной двусвязности является отношением эквивалентности, тогда введем понятие ниже

**Def.** Компонентами рёберной двусвязности графа называют его подграфы, множества вершин которых — классы эквивалентности рёберной двусвязности, а множества рёбер — множества ребер из соответствующих классов эквивалентности.

**Def.** Мост — ребро, соединяющее две компоненты реберной двусвязности.

**Def.** Мост — ребро, при удалении которого число компонент связности увеличивается.

**Утверждение.** Пусть $T$ — дерево обхода в глубину графа $G$. Ребро $(u, v)$ является мостом тогда и только тогда, когда $(u, v) \in T$ и из вершины $v$ и любого ее потомка нет обратного ребра в вершину

$u$ или предка $u$.

**Def.** Функция $ret$

$$ret(v) = min \begin{cases} enter(v), & \text{Время входа в } v \\ enter(p), & p \text{ — конец обратного ребра } (v,p)\ v \\ ret(to), & to \text{ — прямой потомок } v \end{cases}$$

**Утверждение.** Ребро $(u, v)$ является мостом тогда и только тогда, когда $(u, v)$ принадлежит дереву обхода в глубину и $ret(v) > enter(u)$.

**Доказательство:** Рассмотрим вершину $v$ или её потомка. Из нее есть обратное ребро в предка $v$ тогда и только тогда, когда найдется такой сын $t$, что $ret[t] \leq enter[v]$. Если $ret[t] = enter[v]$, то найдется обратное ребро, приходящее точно в $v$. Если же $ret[t] < enter[v]$, то это означает наличие обратного ребра в какого-либо предка вершины $v$.

Таким образом, если для текущего ребра $(v, t)$ (принадлежащего дереву поиска) выполняется $ret[t] > enter[v]$, то это ребро является мостом; в противном случае оно мостом не является.

## Вершинная двусвязность

**Def.** Два ребра графа называются вершинно двусвязными, если существуют вершинно непересекающиеся пути, соединяющие их концы.

Нетрудно показать, что отношение реберной двусвязности является отношением эквивалентности, тогда введем понятие ниже

**Def.** Компонентами вершинной двусвязности графа, называют его подграфы, множества ребер которых — классы эквивалентности вершинной двусвязности, а множества вершин — множества всевозможных концов ребер из соответствующих классов.

**Def.** Точка сочленения графа $G$ — вершина, при удалении которой в $G$ увеличивается число компонент связности.

**Algorithm.** Поиск точек сочленения

Пусть $tin[u]$ — время входа поиска в глубину в вершину $u$. Через $up[u]$ обозначим минимум из времени захода в саму вершину $tin[u]$, времен захода в каждую из вершин $p$, являющуюся концом некоторого обратного ребра $(u, p)$, а также из всех значений $up[v]$ для каждой вершины $v$, являющейся непосредственным сыном $u$ в дереве поиска.

Тогда из вершины $u$или её потомка есть обратное ребро в её предка $\Leftrightarrow$ существует такой сын $v$, что $up[v] \geq tin[u]$.

Таким образом, если для текущей вершины $u \neq root$ существует непосредственный сын $v : up[v] \geq$

$tin[u]$, то вершина $u$ является точкой сочленения, в противном случае она точкой сочленения не является.

**Утверждение.** Пусть $T$ — дерево обхода в глубину, $root$ — корень $T$.

- Вершина $u \neq root$ — точка сочленения равносильно тому, что найдется $v \in T$ — сын $u$: из $v$ или любого потомка вершины $v$ нет обратного ребра в предка вершины $u$.
- $root$ — точка сочленения, что равносильно тому, что $root$ имеет хотя бы двух сыновей в $T$.

**Доказательство:**

$\Rightarrow$

1. Удалим $u$ из $G$. Докажем, что не существует пути из $v$ в любого предка вершины $u$. Пусть это не так. Тогда найдется $x \in T$ — предок $u$ такой, что существует путь из $v$ в $x$ в $G \setminus u$. Пусть $w$ — предпоследняя вершина на этом пути, $w$ — потомок $v$. $(w, x)$ — не ребро дерева $T$ (в силу единственности пути в дереве). Тогда $(w, x)$ — обратное ребро, что противоречит условию.
2. Пусть у $root$ хотя бы два сына. Тогда при удалении $root$ не существует пути между его поддеревьями, так как не существует перекрестных ребер, то есть $root$ — точка сочленения.

$\Leftarrow$

1. Докажем что из отрицания второго утверждения следует отрицание первого. Обозначим через $G'$ граф, состоящий из вершин, не являющихся потомками $u$. Удалим вершину $u$. Очевидно, что граф $G'$ и все поддеревья вершины $u$ останутся связными, кроме того из каждого поддерева есть ребро в $G'$, значит $G \setminus u$ связный, то есть $u$ — не точка сочленения.
2. Пусть $root$ — точка сочленения и у него есть только один сын. Тогда при удалении $root$ остается дерево с корнем в его сыне, содержащее все остальные вершины графа, то есть оставшийся граф связен — противоречие с тем, что $root$ — точка сочленения.

# Асимптотика всех алгоритмов выше $O(V + E)$

## Волновой алгоритм (BFS)

BFS — метод обхода графа в котором сначала обходятся все вершины на расстоянии $K$ от корня обхода, потом все на расстоянии $K + 1$.

**Algorithm.** $BFS$

1. Поместить вершину начальную $v$ в очередь.

2. Извлечь из очереди вершину $u$ и пометить ее как развернутую.
3. Добавить в очередь все неразвернутые смежные с $u$ вершины в очередь, которые еще не в ней.
4. Если очередь пуста, то пройдены все достижимые из $v$ вершины. И необходимо запустить от нерассмотренных вершин BFS.
5. Повторить шаги со второго.

## Эйлеровы графы

**Def.** Эйлеров путь — путь, проходящий по всем рёбрам графа и притом только по одному разу.

**Def.** Граф полуэйлеров, если в нем есть эйлеров путь.

**Def.** Эйлеров цикл — замкнутый путь, проходящий через каждое ребро графа ровно по одному разу.

**Def.** Граф эйлеров, если в нем есть эйлеров цикл.

Будем считать граф связным (для несвязного нетрудно самому получить дополнение к теореме)

**Th.** Критерий эйлеровости неориентированного графа.

Неориентированный граф эйлеров тогда и только тогда, когда все его вершины имеют четную степень.

$\Rightarrow$

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 пройденые 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.

$\Leftarrow$

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 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 $v_2$. $deg(v_2)$ 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 $G'$ — подграф 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 $C_1$nпокрывает 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 связноти 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 $G$.

**Th.** Критерий полуэйлеровости неориентированного графа

Граф $G$ полуэйлеров тогда и только тогда, когда количество вершин с нечетной степенью меньше или равно двум.

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

**Th.** Критерии эйлеровости ориентированного графа

Ориентированный граф $G$ эйлеров тогда и только тогда, когда для каждой вершины верно, что входная степень равна выходной.

**Доказательство:** Аналогично неориентированному графу.

**Th.** Критерий полуэйлеровости ориентированного графа

Граф $G$ полуэйлеров тогда и только тогда, когда количество вершин с нечетной степенью равно двум, при этом для одной из них разность входной и выходной степеней равна 1, а для другой равна -1.

**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.

**Algorithm.** Алгоритм поиска эйлерова цикла

Будем жадно идти по ребрам и удалять их, тогда если граф эйлеров (что необходимо сначала проверить). Можно в этом узреть $DFS$ по ребрам

1. Заведем стек, в него положим любую вершину

2. Если степень вершины ноль, то снимаем вершину со стека и добавляем в ответ, иначе берем любое ребро из этой вершины, удаляем его из графа, а второй конец ребра кладем на стек

3. Повторять, пока стек не опустеет

# Алгоритмы поиска кратчайших путей в графе

## Алгоритм Дейкстры и его производные

Граф $G(V, E)$ взвешенный, если определена весовая функция $w : E \to \mathbb{R}$ такая, что каждому ребру сопоставляет его вес.

Назовем длиной пути $v_1, \dots v_n$ следующую величину: $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$.

**Algorithm.** Алгоритм Дейкстры

Алгоритм Дейкстры ищет кратчайшие пути в графе от заданной вершины до всех, если веса всех ребер неотрицательны.

В алгоритме поддерживается множество вершин $U$, для которых уже вычислены длины кратчайших путей до них из $s$. На каждой итерации основного цикла выбирается вершина $u \notin U$, которой на текущий момент соответствует минимальная оценка кратчайшего пути. Вершина $u$ добавляется в множество $U$ и производится релаксация всех исходящих из неё рёбер. Релаксация — процесс, в котором ищется минимальный путь до данной вершин от всех смежных вершин.

**Th.** Пусть $G(V, E)$ — ориентированный взвешенный граф, вес рёбер которого неотрицателен, $s$ — стартовая вершина. Тогда после выполнения алгоритма Дейкстры $d(u) = \rho(s, u)$ для всех $u$, где $\rho(s, u)$ — длина кратчайшего пути из вершины $s$ в вершину $u$.

**Доказательство:**

Докажем по индукции, что в момент посещения любой вершины $u$, $d(u) = \rho(s, u)$.

- На первом шаге выбирается $s$, для неё выполнено: $d(s) = \rho(s, s) = 0$.

- Пусть для $n$ первых шагов алгоритм сработал верно и на $n+1$ шагу выбрана вершина $u$. Докажем, что в этот момент $d(u) = \rho(s, u)$. Для начала отметим, что для любой вершины $v$, всегда выполняется $d(v) \geq \rho(s, v)$ (алгоритм не может найти путь короче, чем кратчайший из всех существующих). Пусть $P$ — кратчайший путь из $s$ в $u$, $v$ — первая непосещённая вершина на $P$, $z$ — предшествующая ей (следовательно, посещённая). Поскольку путь $P$ кратчайший, его часть, ведущая из $s$ через $z$ в $v$, тоже кратчайшая, следовательно $\rho(s, v) = \rho(s, z) + w(z, v)$. По предположению индукции, в момент посещения вершины $z$ выполнялось $d(z) = \rho(s, z)$, следовательно, вершина $v$ тогда получила метку не больше чем $d(z)+w(z, v) = \rho(s, z)+w(z, v) = \rho(s, v)$,

следовательно, $d(v) = \rho(s, v)$. С другой стороны, поскольку сейчас мы выбрали вершину $u$, её метка минимальна среди непосещённых, то есть $d(u) \leq d(v) = \rho(s, v) \leq \rho(s, u)$, где второе неравенство верно из-за ранее упомянутого определения вершины $v$ в качестве первой непосещённой вершины на $P$, то есть вес пути до промежуточной вершины не превосходит веса пути до конечной вершины вследствие неотрицательности весовой функции. Комбинируя это с $d(u) \geq \rho(s, u)$, имеем $d(u) = \rho(s, u)$, что и требовалось доказать.

- Поскольку алгоритм заканчивает работу, когда все вершины посещены, в этот момент $d(u) = \rho(s, u)$ для всех $u$.

**Утверждение.** Асимптотика алгоритма Дейкстры

Заметим, что в ходе алгоритма нужно выполнять поиск минимума $V$ раз и проводить релаксацию $E$ раз. Тогда если релаксация имеет временную сложность $O(R)$ и поиск минимума $O(M)$ раз, тогда асимптотика составит $O(ER + VM)$.

Для наивной реализации: $O(R) = O(1), O(M) = O(V)$, то есть сложность: $O(V^2 + E)$.

При *set* можно получить, что сложность: $O(VlogV + ElogV) = O(ElogV)$.

**Def.** Дерево кратчайших путей — взвешенное дерево, на каждом ребре которого написана длина от вершины *root* до всех остальных с промежуточными вершинами (коряво сформулировано). Его строит алгоритм Дейкстры.

**Def.** Потенциал — функция $\pi : V \to \mathbb{R}$.

**Def.** Измененный вес ребра $w'(u, v) = w(u, v) + \pi(u) - \pi(v)$.

Нетрудно заметить, что если есть путь $p = \{v_1, \ldots, v_n\}$, то $w'(p) = w(p) + \pi(v_n) - \pi(v_1)$.

Заметим, что если путь для весовой функции $w$ был кратчайшим, то он останется кратчайшим и для функции $w'$, то есть если удалось подобрать такой потенциал, что измененные веса ребер неотрицательны, то можно использовать алгоритм Дейкстры на модифицированных весах.

Рассмотрим пример: $\pi(v) = \rho(v, t)$. Тогда данный потенциал обнуляет все ребра, лежащие на кратчайших путях к вершине $t$.

$w'(u, v) = w(u, v) + \rho(v, t) - \rho(u, t) \geq 0$. То есть на обновленном графе можно применить алгоритм Дейкстры для любой вершины, достижимой из $t$.

**Algorithm.** Алгоритм $A^*$

Алгоритм основан на функции $f(v) = g(v) + h(v)$, где $g$ — наименьшая стоимость пути в $v$ из стартовой вершины, $h$ — эвристическое приближение стоимости пути от $v$ до конечной цели.

Алгоритм работает подобно алгоритму Дейкстры, только при релаксации рассматривается функция $f$, а не значение $g$.

**Def.** Эвристическая оценка $h(v)$ допустима, если для любой вершины $v$ значение $h(v)$ меньше

или равно весу кратчайшего пути от $v$ до цели.

**Def.** Эвристическая функция $h(v)$ называется монотонной, если для любой вершины $v_1$ и ее потомка $v_2$ разность $h(v_1)$ и $h(v_2)$ не превышает фактического веса ребра $w(v1, v2)$, а эвристическая оценка целевого состояния равна нулю.

**Утверждение.** Монотонная эвристика допустима (обратное неверно)

**Утверждение.** Если $h(v)$ монотонна, то последовательность значений $f(v)$ на любом пути не убывает.

Примеры эвристик:

- Пусть задана координантная сетка, по которой можно перемещаться в четырех направлениях, тогда в качестве эвристики стоит выбрать манхэттонское расстояние.
- Пусть перемещение происходит на плоскости, тогда в качестве эвристики можно выбрать евклидово расстояние.

**Algorithm.** Двунаправленный алгоритм Дейкстры

Мы можем уменьшить количество посещённых вершин в алгоритме Дейкстры, просто запустив его и из начальной и из конечной вершины. Такая эвристика не испортит скорость работы в худшем случае.

Создадим две приоритетных очереди и запустим на одной из них алгоритм Дейкстры, ищущий $d_{forward}(v)$ из $s$, а на другой — ищущий $d_{reverse}(v)$ из $t$. Алгоритм завершит свою работу, когда какая-нибудь вершина $z$ будет удалена из обеих очередей.

Тонкость этого алгоритма заключается в том, что кратчайший путь $s \to t$ не обязательно пройдёт через вершину $z$. Поэтому после остановки двунаправленного поиска, нам необходимо перебрать все рёбра из вершин, имеющих $d_{forward}(u)$ в вершины с $d_{reverse}(v)$ и найти ребро $(u, v)$ с минимальным $d_{forward}(u) + w(u, v) + d_{reverse}(v)$. Если эта величина меньше, чем длина первоначально найденного пути, то это и есть результат работы алгоритма.

На практике, такой двунаправленный поиск быстрее обычного алгоритма Дейкстры примерно в два раза.

## Алгоритм Форда-Беллмана

Алгоритм Форда-Беллмана решает задачу поиска кратчайших путей для графа с произвольной весовой функцией (необязательно неотрицательной).

**Algorithm.** Алгоритм Форда-Беллмана

Количество путей длины $k$ рёбер можно найти с помощью метода динамического программирования. Пусть $d[k][u]$ — количество путей длины $k$ рёбер, заканчивающихся в вершине $u$. Тогда

$d[k][u] = \sum\limits_{(v,u)\in E} d[k-1][v]$.

Аналогично посчитаем пути кратчайшей длины. Пусть $s$ — стартовая вершина.

Тогда $d[k][u] = \min\limits_{v:(v,u)\in E}(d[k-1][v] + w(u,v))$, при этом $d[0][s] = 0$, а $d[0][u] = \infty$.

Тогда заметим, что искомая длина пути из $s$ в $t$: $\rho(s,t) = \min\limits_{k\in[0,n-1]} d[k][t]$. Корректность следует из определения матрицы $d$.

Псевдокод алгоритма:

```
bool fordBellman(s):
    for v ∈ V
        d[v] = 1
    d[s] = 0
    for i = 0 to |V| − 1
        for (u, v) ∈ E
            if d[v] > d[u] + w(u, v)
                d[v] = d[u] + w(u, v)
    for (u, v) ∈ E
        if d[v] > d[u] + w(u, v)
            return false
    return true
```

**Утверждение.** Асимптотика алгоритма: $O(VE)$.

**Лемма.** Пусть $G(V,E)$ — взвешенный ориентированный граф, $s$ — стартовая вершина. Тогда после завершения $|V|-1$ итераций цикла для всех вершин, достижимых из $s$, выполняется равенство $d[v] = \delta(s,v)$ — фактический вес кратчайшего пути из $s$ в $v$.

**Доказательство:** Рассмотрим произвольную вершину $v$, достижимую из $s$. Пусть $p = (v_0 \dots, v_k)$, где $v_0 = s$, $v_k = v$ — кратчайший ациклический путь из $s$ в $v$. Путь $p$ содержит не более $|V|-1$ ребер. Поэтому $k \le |V| - 1$. Докажем следующее утверждение:

После $n : (n \le k)$ итераций первого цикла алгоритма, $d[v_n] = \delta(s, v_n)$. Воспользуемся индукцией по $n$ (база очевидна по определению матрицы $d$). Индукционный переход:

Пусть после $n : (n < k)$ итераций, верно что $d[v_n] = \delta(s, v_n)$. Так как $(v_n, v_{n+1})$ принадлежит кратчайшему пути от $s$ до $v$, то $\delta(s, v_{n+1}) = \delta(s, v_n) + \omega(v_n, v_{n+1})$. Во время $l+1$ итерации релаксируется ребро $(v_n, v_{n+1})$, следовательно по завершению итерации будет выполнено, что $d[v_{n+1}] \leqslant d[v_n] + \omega(v_n, v_{n+1}) = \delta(s, v_n) + \omega(v_n, v_{n+1}) = \delta(s, v_{n+1})$.

Ясно, что $d[v_{n+1}] \geqslant \delta(s, v_{n+1})$, поэтому верно что после $l+1$ итерации $d[v_{n+1}] = \delta(s, v_{n+1})$. Индукционный переход доказан. Итак, выполнены равенства $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$.

**Th.** Пусть $G = (V, E)$ — взвешенный ориентированный граф, $s$ — стартовая вершина. Если граф

$G$ не содержит отрицательных циклов, достижимых из вершины $s$, то алгоритм возвращает $true$ и для всех $v \in V$ $d[v] = \delta(s, v)$. Если граф $G$ содержит отрицательные циклы, достижимые из вершины $s$, то алгоритм возвращает $false$.

**Доказательство:** Пусть граф $G$ не содержит отрицательных циклов, достижимых из вершины $s$.

Тогда если вершина $v$ достижима из $s$, то по лемме $d[v] = \delta(s, v)$. Если вершина $v$ не достижима из $s$, то $d[v] = \delta(s, v) = \infty$ из несуществования пути.

Теперь докажем, что алгоритм вернет значение $true$. После выполнения алгоритма верно, что для всех $(u, v) \in E$, $d[v] = \delta(s, v) \leqslant \delta(s, u) + \omega(u, v) = d[u] + \omega(u, v)$, значит ни одна из проверок не вернет значения $false$.

Пусть граф $G$ содержит отрицательный цикл $c = (v_0, ..., v_k)$, где $v_0 = v_k$, достижимый из вершины $s$. Тогда $\sum\limits_{i=1}^{k} \omega(v_{i-1}, v_i) < 0$.

Предположим, что алгоритм возвращает $true$, тогда для $i = 1, \ldots, k$ выполняется $d[v_i] \leqslant d[v_{i-1}] + \omega(v_{i-1}, v_i)$.

Просуммируем эти неравенства по всему циклу: $\sum\limits_{i=1}^{k} d[v_i] \leqslant \sum\limits_{i=1}^{k} d[v_{i-1}] + \sum\limits_{i=1}^{k} \omega(v_{i-1}, v_i)$. Из того, что $v_0 = v_k$ следует, что $\sum\limits_{i=1}^{k} d[v_i] = \sum\limits_{i=1}^{k} d[v_{i-1}]$. Получили, что $\sum\limits_{i=1}^{k} \omega(v_{i-1}, v_i) \geqslant 0$, что противоречит отрицательности цикла $c$.

**Algorithm.** Поиск отрицательного цикла.

Чтобы найти сам цикл, достаточно хранить вершины, из которых производится релаксация.

Если после $|V|-1$ итерации найдется вершина $v$, расстояние до которой можно уменьшить, то эта вершина либо лежит на каком-нибудь цикле отрицательного веса, либо достижима из него. Чтобы найти вершину, которая лежит на цикле, можно $|V| - 1$ раз пройти назад по предкам из вершины $v$. Так как наибольшая длина пути в графе из $|V|$ вершин равна $|V| - 1$, то полученная вершина $u$ будет гарантированно лежать на отрицательном цикле.

Зная, что вершина $u$ лежит на цикле отрицательного веса, можно восстанавливать путь по сохраненным вершинам до тех пор, пока не встретится та же вершина $u$. Это обязательно произойдет, так как в цикле отрицательного веса релаксации происходят по кругу.

## Алгоритм Флойда

Дан взвешенный ориентированный граф $G(V, E)$, в котором вершины пронумерованы от 1 до $n$.

$$\omega_{uv} = \begin{cases} \text{w(u, v)}, & \text{if } (u, v) \in E \\ +\infty, & \text{if } (u, v) \notin E \end{cases}$$

Требуется найти матрицу кратчайших расстояний $d$, в которой элемент $d_{ij}$ либо равен длине

кратчайшего пути из $i$ в $j$, либо равен $+\infty$, если вершина $j$ недостижима из $i$.

**Algorithm.** Алгоритм Флойда-Уоршелла

Обозначим длину кратчайшего пути между вершинами $u$ и $v$, содержащего, помимо $u$ и $v$, только вершины из множества $\{1 \dots i\}$ как $d_{uv}^{(i)}$, $d_{uv}^{(0)} = \omega_{uv}$.

На каждом шаге алгоритма, мы будем брать очередную вершину (пусть её номер — $i$) и для всех пар вершин $u$ и $v$ вычислять $d_{uv}^{(i)} = \min(d_{uv}^{(i-1)}, d_{ui}^{(i-1)} + d_{iv}^{(i-1)})$. То есть, если кратчайший путь из $u$ в $v$, содержащий только вершины из множества $\{1 \dots i\}$, проходит через вершину $i$, то кратчайшим путем из $u$ в $v$ является кратчайший путь из $u$ в $i$, объединенный с кратчайшим путем из $i$ в $v$. В противном случае, когда этот путь не содержит вершины $i$, кратчайший путь из $u$ в $v$, содержащий только вершины из множества $\{1 \dots i\}$ является кратчайшим путем из $u$ в $v$, содержащим только вершины из множества $\{1 \dots i-1\}$.

В итоге получаем, что матрица $d^{(n)}$ и является искомой матрицей кратчайших путей, поскольку содержит в себе длины кратчайших путей между всеми парами вершин, имеющих в качестве промежуточных вершины из множества $\{1 \dots n\}$, что есть попросту все вершины графа. Такая реализация работает за $\Theta(n^3)$ времени и использует $\Theta(n^3)$ памяти.

**Утверждение.** В течение работы алгоритма Флойда выполняются неравенства: $\rho(u, v) \leqslant d_{uv} \leqslant d_{uv}^{(i)}$.

**Доказательство:**

**Докажем второе неравенство индукцией по итерациям алгоритма.**

Пусть также $d'_{uv}$ — значение $d_{uv}$ сразу после $i-1$ итерации.

Покажем, что $d_{uv} \leqslant d_{uv}^{(i)}$, зная, что $d'_{uv} \leqslant d_{uv}^{(i-1)}$.

Рассмотрим два случая:

1. Значение $d_{uv}^{(i)}$ стало меньше, чем $d_{uv}^{(i-1)}$. Тогда $d_{uv}^{(i)} = d_{ui}^{(i-1)} + d_{iv}^{(i-1)} \geqslant$ (выполняется на шаге $i-1$, по индукционному предположению) $\geqslant d'_{ui} + d'_{iv} \geq$ (в силу выполнения 7-ой строчки алгоритма на $i$-ой итерации и невозрастания элементов массива $d$) $\geqslant d_{uv}$.
2. В ином случае всё очевидно: $d_{uv}^{(i)} = d_{uv}^{(i-1)} \geqslant d'_{uv} \geqslant d_{uv}$, и неравенство тривиально.

**Докажем первое неравенство от противного.**

Пусть неравенство было нарушено, рассмотрим момент, когда оно было нарушено впервые. Пусть это была $i$-ая итерация и в этот момент изменилось значение $d_{uv}$ и выполнилось $\rho(u, v) > d_{uv}$. Так как $d_{uv}$ изменилось, то $d_{uv} = d_{ui} + d_{iv} \geq$ (так как ранее $\forall u, v \in V : \rho(u, v) \leqslant d_{uv}$) $\geqslant \rho(u, i) + \rho(i, v) \geq$ (по неравенству треугольника) $\geqslant \rho(u, v)$. Итак $d_{uv} \geqslant \rho(u, v)$ — противоречие.

Тогда можно избавиться от одной размерности в массиве $d$, т.е. использовать двумерный массив $d_{uv}$. В процессе работы алгоритма поддерживается инвариант $\rho(u, v) \leqslant d_{uv} \leqslant d_{uv}^{(i)}$, а, поскольку, после

выполнения работы алгоритма $\rho(u, v) = d_{uv}^{(i)}$, то тогда будет выполняться и $\rho(u, v) = d_{uv}$.

```
d = ω
    for i ∈ V
        for u ∈ V
            for v ∈ V
                d[u][v] = min(d[u][v], d[u][i] + d[i][v])
```

Данная реализация работает за время $\Theta(n^3)$, но требует уже $\Theta(n^2)$ памяти. В целом, алгоритм Флойда очень прост, и, поскольку в нем используются только простые операции, константа, скрытая в определении $\Theta$ весьма мала.

**Note.** Алгоритм Флойда легко модифицировать таким образом, чтобы он возвращал не только длину кратчайшего пути, но и сам путь. Для этого достаточно завести дополнительный массив next , в котором будет храниться номер вершины, в которую надо пойти следующей, чтобы дойти из u в v по кратчайшему пути.

```
d = ω
    for i ∈ V
        for u ∈ V
            for v ∈ V
                d[u][v] = min(d[u][v], d[u][i] + d[i][v])
                next[u][v] = next[u][i]

func getShortestPath(u, v):
        if d[u][v] == ∞
                print "No path found"
        c = u
        while c != v
                print c
                c = next[c][v]
        print v
```

**Утверждение.** При наличии цикла отрицательного веса в матрице D появятся отрицательные числа на главной диагонали.

**Доказательство:**

Так как алгоритм Флойда последовательно релаксирует расстояния между всеми парами вершин $(i, j)$, в том числе и теми, у которых $i = j$, а начальное расстояние между парой вершин $(i, i)$ равно нулю, то релаксация может произойти только при наличии вершины $k$ такой, что $d[i][k] + d[k][i] < 0$, что эквивалентно наличию отрицательного цикла, проходящего через вершину $i$.

## Алгоритм Джонсона

Алгоритм Джонсона находит кратчайшие пути между всеми парами вершин во взвешенном ориентированном графе с любыми весами ребер, но не имеющем отрицательных циклов.

В этом алгоритме используется метод изменения веса. Суть его заключается в том, что для заданного графа $G$ строится новая весовая функция $\omega_\varphi$, неотрицательная для всех ребер графа $G$ и сохраняющая кратчайшие пути. Такая весовая функция строится с помощью так называемой потенциальной функции.

Пусть $\varphi : V \to \mathbb{R}$ — произвольное отображение из множества вершин в вещественные числа. Тогда новой весовой функцией будет $\omega_\varphi(u, v) = \omega(u, v) + \varphi(u) - \varphi(v)$.

Такая потенциальная функция строится добавлем фиктивной вершины $s$ в $G$, из которой проведены ориентированные ребра нулевого веса во все остальные вершины графа, и запуском алгоритма Форда-Беллмана из нее ($\varphi(v)$ будет равно длине кратчайшего пути из $s$ в $v$). На этом же этапе мы сможем обнаружить наличие отрицательного цикла в графе.

**Лемма.** Пусть $P,\ Q$ — два пути $a \rightsquigarrow b$ и $\omega(P) < \omega(Q)$. Тогда $\forall\varphi\ \omega_\varphi(P) < \omega_\varphi(Q)$

**Доказательство:**

Рассмотрим путь $P\ u_0 \to u_1 \to u_2 \to \ldots \to u_k$

Его вес с новой весовой функцией равен $\omega_\varphi(P) = \omega_\varphi(u_0u_1) + \omega_\varphi(u_1u_2) + \ldots + \omega_\varphi(u_{k-1}u_k)$.

Вставим определение функции $\omega_\varphi\omega_\varphi(P) = \varphi(u_0)+\omega(u_0u_1)-\varphi(u_1)+\ldots+\varphi(u_{k-1})+\omega(u_{k-1}u_k)-\varphi(u_k)$

Заметим, что потенциалы все промежуточных вершин в пути сократятся. $\omega_\varphi(P) = \varphi(u_0)+\omega(P)-\varphi(u_k)$

По изначальному предположению $\omega(P) < \omega(Q)$. С новой весовой функцией веса соответствующих путей будут

$\omega_\varphi(P) = \varphi(a) + \omega(P) - \varphi(b)$

$\omega_\varphi(Q) = \varphi(a) + \omega(Q) - \varphi(b)$

Отсюда, $\omega_\varphi(P) < \omega_\varphi(Q)$

**Th.** В графе $G$ нет отрицательных циклов $\Leftrightarrow$ существует потенциальная функция $\phi\ \forall uv \in E\ \omega_\varphi(uv) \geqslant 0$

**Доказательство:**

$\Leftarrow$ Рассмотрим произвольный $C$ — цикл в графе $G$

По лемме, его вес равен $\omega(C) = \omega_\varphi(C) + \varphi(u_0) - \varphi(u_0) = \omega_\varphi(C) \geqslant 0$

$\Rightarrow$ Добавим фиктивную вершину $s$ в граф, а также ребра $s \to u$ весом 0 для всех $u$.

Обозначим $\delta(u, v)$ как минимальное расстояние между вершинами $u,\ v$, введем потенциальную

функцию $\phi$

$\phi(u) = \delta(s, u)$

Рассмотрим вес произвольного ребра $uv \in E$ $\omega_\phi(uv) = \phi(u)+\omega(uv)-\phi(v) = \delta(s,u)+\omega(uv)-\delta(s,v)$.

Поскольку $\delta(s,u)+\omega(uv)$ — вес какого-то пути $s \rightsquigarrow v$, а $\delta(s,v)$ — вес кратчайшего пути $s \rightsquigarrow v$, то $\delta(s,u)+\omega(uv) \geqslant \delta(s,v) \Rightarrow \delta(s,u)+\omega(uv)-\delta(s,v) = \omega_\varphi(uv) \geqslant 0$.

**Псевдокод:**

Предварительно построим граф $G' = (V',\ E')$, где $V' = V \cup \{s\}$, $s \notin V$, а $E' = E \cup \{(s,\ v) : \omega(s,v) = 0,\ v \in V\}$

```
function Johnson(G): int[][]
    if BellmanFord(G', ω, s) == ''false''
        print (this graph has negative cycle)
        return ∅
    else for v ∈ V'
    φ(v) = δ(s, v) // δ(s, v) is counted by Ford–Bellman algo
    for (u, v) ∈ E'
        ω_φ(u, v) = ω(u, v) + φ(u) − φ(v)
    for u ∈ V
        Dijkstra(G, ω_φ, u)
        for v ∈ V
            d_uv ← δ_φ(u, v) + φ(v) − φ(u)
    return d
```

Итого, в начале алгоритм Форда-Беллмана либо строит потенциальную функцию такую, что после перевзвешивания все веса ребер будут неотрицательны, либо выдает сообщение о том, что в графе присутствует отрицательный цикл.

Затем из каждой вершины запускается алгоритм Дейкстры для составления искомой матрицы. Так как все веса ребер теперь неотрицательны, алгоритм Дейкстры будет работать корректно. А поскольку перевзвешивание таково, что кратчайшие пути относительно обеих весовых функций совпадают, алгоритм Джонсона в итоге корректно найдет все кратчайшие пути между всеми парами вершин.

Алгоритм Джонсона работает за $O(VE + VD)$, где $O(D)$ — время работы алгоритма Дейкстры. Если в алгоритме Дейкстры неубывающая очередь с приоритетами реализована в виде фибоначчиевой кучи, то время работы алгоритма Джонсона есть $O(V^2 \log V + VE)$. В случае реализации очереди с приоритетами в виде двоичной кучи время работы равно $O(VE \log V)$.

# Миностовы и СНМ

## Остовное дерево. Построение с помощью обхода в глубину и в ширину.

**Def.** $T \subset G$ — **остовное дерево**, если $T$ содержит все вершины $G$ и является деревом

### Построение с помощью обхода в глубину/ширину

**Утверждение.** Любое дерево обхода будет остовным деревом.

**Algorithm.** Запустим бфс/дфс и посмотрим дерево обхода. Получим остовное дерево

## Определение минимального остовного дерева.

**Def.** Минимальным остовным деревом во взвешенном неориентированном графе называется остовное дерево минимального веса.

## Теорема о разрезе. Доказательство.

**Def.** $(S, T)$ — разрез, если $S \cup T = V, S \cap T = \varnothing$

**Def.** $(u, v)$ пересекает разрез $(S, T)$, если $u$ и $v$ — в разных частях разреза.

**Def.** Ребро $(u, v) \notin G'$ называется безопасным, если при добавлении его в $G'$, $G' \cup \{(u, v)\}$ также является подграфом некоторого минимального остовного дерева графа $G$.

**Th.** Рассмотрим связный неориентированный взвешенный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$. Пусть $G' = (V, E')$ — подграф некоторого минимального остовного 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:**

Достроим $E'$ до некоторого минимального остовного дерева, обозначим его $T_{min}$. Если ребро $e \in T_{min}$, то лемма доказана, поэтому рассмотрим случай, когда ребро $e \notin T_{min}$. Рассмотрим путь в $T_{min}$ от вершины $u$ до вершины $v$. Так как эти вершины принадлежат разным долям разреза, то хотя бы одно ребро пути пересекает разрез, назовем его $e'$. По условию леммы $w(e) \leqslant w(e')$. Заменим ребро $e'$ в $T_{min}$ на ребро $e$. Полученное дерево также является минимальным остовным деревом графа $G$, поскольку все вершины $G$ по-прежнему связаны и вес дерева не увеличился. Следовательно $E' \cup \{e\}$ можно дополнить до минимального остовного дерева в графе $G$, то есть ребро $e$ — безопасное. ■

## Алгоритм Прима. Аналогия с алгоритмом Дейкстры.

**Algorithm.** Алгоритм Прима – алгоритм поиска минимального остовного дерева.

- Выбираем произвольную стартовую вершину – начало строящегося дерева. Строим очередь с приоритетом вершин, до которых есть ребро от стартовой. Приоритет = длина ребра до стартовой.
- Итеративно добавляем к дереву вершину из очереди с ребром додерева, имеющим вес = приоритет вершины. Добавляем новыевершины, доступные из добавленной вершины по ребрам графа ссоответствующими приоритетами. Либо обновляем приоритетвершины в очереди, если от добавленной вершины к ней ведет болеелегкое ребро.

## Доказательство с помощью теоремы о разрезе. Оценка времени работы для различных реализаций очереди с приоритетом: бинарная куча, Фибоначчиева куча (последнее без доказательства)

**Th.** Алгоритм Прима корректен

**Доказательство:**

Пусть граф G был связным, т.е. ответ существует. Обозначим через T остов, найденный алгоритмом Прима, а через S — минимальный остов. Очевидно, что T действительно является остовом (т.е. поддеревом графа G). Покажем, что веса S и T совпадают.

Рассмотрим первый момент времени, когда в T происходило добавление ребра, не входящего в оптимальный остов S. Обозначим это ребро через e, концы его — через a и b, а множество входящих на тот момент в остов вершин — через V (согласно алгоритму, a принадлежит V, b не принадлежит V, либо наоборот). В оптимальном остове S вершины a и b соединяются каким-то путём P; найдём в этом пути любое ребро g, один конец которого лежит в V, а другой — нет. Поскольку алгоритм Прима выбрал ребро e вместо ребра g, то это значит, что вес ребра g больше либо равен весу ребра e.

Удалим теперь из S ребро g, и добавим ребро e. По только что сказанному, вес остова в результате не мог увеличиться (уменьшиться он тоже не мог, поскольку S было оптимальным). Кроме того, S не перестало быть остовом (в том, что связность не нарушилась, нетрудно убедиться: мы замкнули путь P в цикл, и потом удалили из этого цикла одно ребро).

Итак, мы показали, что можно выбрать оптимальный остов S таким образом, что он будет включать ребро e. Повторяя эту процедуру необходимое число раз, мы получаем, что можно выбрать оптимальный остов S так, чтобы он совпадал с T. Следовательно, вес построенного алгоритмом Прима T минимален. ■

**Утверждение.** Алгоритм Прима работает за $O(ElogV)$ с очередью на куче и за $O(VlogV + E)$ при очереди на Фибонначиевой куче.

**Доказательство:**

- Извлечение вершины из очереди с приоритетом – $O(logV)$
- Вставка в кучу O(logV)

Пункт 1) выполнится V раз, пункт 2) E раз. ■

## Алгоритм Крускала. Доказательство. Оценка времени работы.

**Algorithm.** Алгоритм Крускала – еще один алгоритм поиска минимального остовного дерева.

- Сортируем все ребра графа по весу.
- Инициализируем лес деревьев. Изначально каждая вершина — дерево.
- Последовательно рассматриваем ребра графа в порядке возрастания веса. Если очередное ребро соединяет два разных дерева из леса, то объединяем эти два дерева этим ребром в одно дерево. Если очередное ребро соединяет две вершины одного дерева из леса, то пропускаем такое ребро.
- Повторяем 3), пока в лесу не останется одно дерево.

**Th.** Алгоритм Крускала корректен

**Доказательство:** Рассмотрим шаг алгоритма. Пусть текущее ребро при добавлении не создает цикл. Тогда оно соединяет два разных поддерева в MST, то есть соединяет разрез. У него минимальный вес, значит оно безопасно.

**Утверждение.** Алгоритм Крускала работает за O(Elog(E))

**Доказательство:**

Сортировка E займет O(ElogE). Работа с CHM займет $O(E\alpha(V))$, где $\alpha$ — обратная функция Аккермана, которая не превосходит 4 во всех практических приложениях и которую можно принять за константу. Алгоритм работает за $O(E(logE + \alpha(V))) = O(ElogE)$. ■

## Система непересекающихся множеств. Эвристика ранга с доказательством оценки времени работы

**Def.** DSU — система непересекающихся множеств

- Create(u) — создать множество с одним элементом u.
- Find(u) — найти множество по элементу u, чтобы можно было сравнить их (Find(u) == Find(v)).

- Union(u, v) — объединить два множества, одно из которых содержит элемент u, а другое — элемент v.

**Def.** В операции Union будем присоединять дерево с меньшим рангом к дереву с большим рангом. Это называется ранговая эвристика. Есть два варианта ранговой эвристики: в одном варианте рангом дерева называется количество вершин в нём, в другом — глубина дерева (точнее, верхняя граница на глубину дерева, поскольку при совместном применении эвристики сжатия путей реальная глубина дерева может уменьшаться).

**Утверждение.** Ранговая эвристика дает O(log(n)) на каждый запрос.

Рассмотрим ранговую эвристику по глубине дерева. Покажем, что если ранг дерева равен k, то это дерево содержит как минимум $2^k$ вершин (отсюда будет автоматически следовать, что ранг, а, значит, и глубина дерева, есть величина $O(\log n)$). Доказывать будем по индукции: для $k = 0$ это очевидно. При сжатии путей глубина может только уменьшиться. Ранг дерева увеличивается с $k-1$ до $k$, когда к нему присоединяется дерево ранга $k-1$; применяя к этим двум деревьям размера $k-1$ предположение индукции, получаем, что новое дерево ранга $k$ действительно будет иметь как минимум $2^k$ вершин. ■

## Эвристика сжатия пути без доказательства.

**Def.** Эвристика сжатия пути заключается в следующем: когда после вызова find_set(v) мы найдём искомого лидера p множества, то запомним, что у вершины v и всех пройденных по пути вершин — именно этот лидер p. Проще всего это сделать, перенаправив их parent[] на эту вершину p.

## Алгоритм Борувки. Доказательство. Оценка времени работы.

**Algorithm.**

- В лесу каждое дерево – одна вершина. Каждая вершина — отдельное дерево.
- Для каждого поддерева $T_i$ выбираем самое легкое ребро, соединяющее $T_i$ с остальным графом. Добавляем найденные ребра в лес, объединяя связанные им деревья.

**Th.** Алгоритм Борувки корректнен.

**Доказательство:**

- Будет построено дерево. Цикл не добавится, если выбрать один «внешний» способ упорядочения.
- Будет построено дерево минимального веса. От противного. По аналогии с теоремой о разрезе рассмотреть первое добавленное ребро, не принадлежащее мин.остову.

**Утверждение.** Алгоритм Борувки работает за $O(ElogV)$

**Доказательство:** На каждом шаге алгоритма количество деревьев в лесу уменьшается как минимум в два раза. Следовательно, количество итераций не больше log V. Один шаг выполняется за O(E). Общее время работы — $O(ElogV)$. ■

## Приближение решения задачи коммивояжера с помощью минимального остовного дерева.

Задача коммивояжера ставится следующим образом: Дан взвешенный полный граф с неотрицательными весами, необходимо найти минималный гамильтонов цикл (все же знают, что такое гамильтонов цикл, да?))

Данная задача относится к классу $NP$-полных задач, то есть не существует полиномиального алгоритма, решающего данную задачу абсолютно точно.

### Приближенное решение с точностью 2

1. Найти миностов в графе
2. Удвоить каждое ребро (заметим тогда, что удвоенное дерево будет эйлеровым подграфом исходного графа)
3. Найти эйлеров обход этого подграфа
4. Ответом будет гамильтонов цикл, посещая его вершины в порядке полученного эйлерового подхода

**Доказательство:** Заметим, что ответ на задачу не превосходит веса миностова, а вес построенного гамильтонового цикла не превосходит удвоенного веса миностова, чей вес является нижней оценкой на ответ данной задачи.

**Algorithm.** Строим MST, делаем обход по MST сокращая пути. Получаем приблеженное с точностью до 2.

# Потоки

## Определение сети. Определение потока.

**Def.** Сеть $G = (V, E)$ представляет собой, в котором каждое $(u, v) \in E$ имеет положительную пропускную способность $c(u, v) > 0$. Если $(u, v) \notin E$, предполагается что $c(u, v) = 0$.

**Def.** Потоком $f$ в $G$ является действительная функция $f : V \times V \to R$, удоволетворяющая условиям: 1) $f(u, v) = -f(v, u)$ (антисимметричность);

2) $|f(u, v)| \leqslant c(u, v)$ (ограничение пропускной способности), если ребра нет, то $f(u, v) = 0$;

3) $\sum\limits_{v} f(u, v) = 0$ для всех вершин $u$, кроме $s$ и $t$ (закон сохранения потока). Величина потока $f$ определяется как $|f| = \sum\limits_{v \in V} f(s, v)$.

## Физический смысл. Аналогия с законами Кирхгофа.

Поток это ток. Алгебраическая сумма токов равна 0. Я не учу физику отбебитесь.

А теперь немного о потоках расскажет человек, сдававший законы Кирхгофа на экзамене по общесосу. Для цепей есть два закона Кирхгофа:

1. Сумма втекающих токов равна сумме вытекающих токов, то есть тут подразумевалось автором вопроса в программе то, что все, что втекло в вершину обязано из нее выйти (если вершина на сток и не исток)

2. Для любого замкнутого контура сумма падений напряжений равна алгебраической сумме источников ЭДС на данном контуре (тут я не могу придумать нормальной аналогии с потоками)

## Определение разреза. Понятия потока через разрез.

**Def.** $(s, t)$ — разрезом $\langle S, T\rangle$ в сети $G$ называется пара множеств $S, T$, удоволетворяющих условиям:
1)$s \in S, t \in T$
2)$S = V \setminus T$

**Def.** Поток в разрезе $\langle S, T\rangle$ обозначается $f(S, T)$ и вычисляется по формуле: $f(S, T) = \sum\limits_{u \in S} \sum\limits_{v \in T} f(u, v)$.

## Доказательство факта, что поток через любой разрез одинаковый.

**Утверждение.** Пусть $\langle S, T\rangle$ — разрез в $G$. Тогда $f(S, T) = |f|$

**Доказательство:**

$f(S,T) = f(S,V) - f(S,S) = f(S,V) = f(S \setminus s, V) + f(s,V) = f(s,V) = |f|$

1-е равенство выполняется, так как суммы не пересекаются: $f(S,V) = f(S,S) + f(S,T)$

2-е равенство выполняется из-за антисимметричности: $f(S,S) = -f(S,S) = 0$

3-е равенство выполняется, как и 1-е, из-за непересекающихся сумм

4-е равенство выполняется из-за сохранения потока ■

## Понятие остаточной сети. Понятие дополняющего пути.

**Def.** Для заданной транспортной сети $G = (V, E)$ и потока $f$, остаточной сетью, в $G$, порожденной потоком $f$, является сеть $G_f = (V, E_f)$, где $E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}$. $c_f(u,v)$ — то есть данная величина является пропускной способностью ребра в остаточной сети.

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v), & \text{если } (u,v) \in E; \\ f(v,u), & \text{если } (v,u) \in E; \\ 0, & \text{иначе} \end{cases}$$

**Def.** Для заданной транспортной сети $G = (V, E)$ и потока $f$ дополняющим путем $p$ является простой путь из истока в сток в остаточной сети $G_f = (V, E_f)$.

### Необходимость отсутствия дополняющего пути для максимальности потока.

**Утверждение.** Общую величину потока можно увеличить на значение потока по дополняющему пути.

## Теорема Форда-Фалкерсона.

**Th.** Следующие утверждения эквивалентны

1. Поток f максимален
2. Остаточная сеть $G_r$ не содержит дополняющих путей
3. Для некоторого разреза $(S,T)$ выполнено равенство $|f| = c(S,T)$

**Доказательство:**

$1 \Rightarrow 2$. В остаточной сети максимального потока нет дополняющих путей, иначе поток можно было бы увеличить.

$2 \Rightarrow 3$. Построим явно такой разрез, для которого $|f| = c(S,T)$. В остаточной сети $_f$ нет пути из s в t. В качестве S возьмем множество вершин, достижимых из s. В качестве T - дополнение S. T не пусто и содержит t. Для всех ребер (u,v), пересекающих разрез, $c_f(u,v) = 0$. Следовательньо, $|f| = c(S,T)$.

$3 \Rightarrow 1$. Для любого потока $|g| = g(S,T) \leqslant c(S,T)$. Значит, f — максимальный поток. ■

## Алгоритм Форда-Фалкерсона. Поиск минимального разреза

**Algorithm.** Алгоритм Форда-Фалкерсона

1. Строим остаточную сеть $G_f = G$
2. Пока $\exists$ дополняющий путь $p$ в $G_r$
   - Вычисляем $c_p$ — пропускную способность пути p
   - $f+ = f_p$, где $f_p$ - поток, пропущенный по p. // То есть тут складываются две матрицы "функции"потоков.
   - $|f|+ = c_p$ // К максимальной величине потока прибавляем величину потока добавочного пути

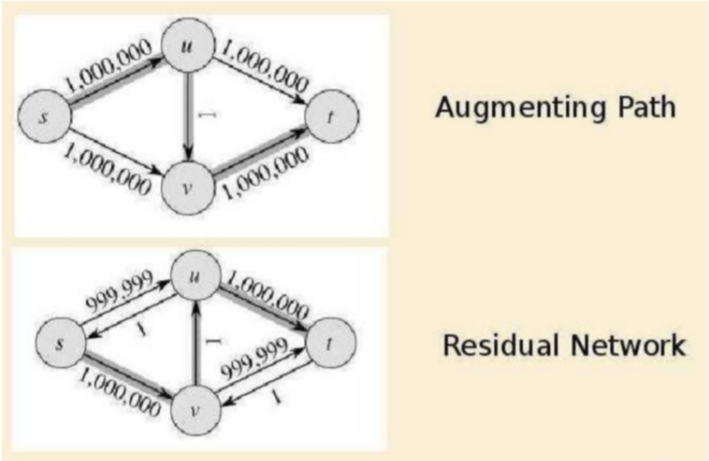
**Def.** Минимальным разрезом называется разрез с минимально возможной пропускной способностью

**Algorithm.** Поиск наименьшего разреза. Ищем максимальный поток, разделяем на S - s + все вершины достижимые из S, T — дополнение. По следующему утверждению это будет минимальный разрез.

**Утверждение.** Если $f(S,T) = c(S,T)$, то поток $f$ — максимален, а разрез $\langle S,T\rangle$ — минимален.

**Доказательство:**

Заметим, что $f(S_1,T_1) \leqslant c(S_2,T_2)$ для любых двух разрезов $\langle S_1,T_1\rangle$ и $\langle S_2,T_2\rangle$ в сети $G$, так как $f(S_1,T_1) = |f| = f(S_2,T_2) \leqslant c(S_2,T_2)$. Значит, если расположить все величины потоков и разрезов на оси OX, то у потоков с разрезами может быть максимум 1 точка пересечения. Очевидно, что эта точка определяет максимальный поток среди всех потоков и минимальный разрез среди всех разрезов сети $G$.




## Пример целочисленной сети, в котором алгоритм работает долго.

## Алгоритм Эдмондса-Карпа.

**Algorithm.** Алгоритм Эдмондса-Карпа — оптимизация алгоритма Форда-Фалкерсона, в которой увеличивающий путь = кратчайший путь между s и t по количеству ребер. $T = O(VE^2)$ Одна итерация:

1. BFS в остаточной сети
2. Добавление найденного пути в поток
3. Обновление остаточной сети

**Th.** Алгоритм Эдмондса-Карпа корректен

**Доказательство:**

На каждой итерации цикла while поток в графе $G$ увеличивается вдоль одного из кратчайших путей в $G_f$ из истока $s$ в сток $t$. Этот процесс повторяется до тех пор пока существует кратчайший $s \rightsquigarrow t$ путь в $G_f$. Если в $G_f$ не существует кратчайшего пути из $s$ в $t$, значит, не существует вообще никакого $s \rightsquigarrow t$ пути в $G_f$ следовательно по теореме Форда-Фалкерсона найденный поток $f$ максимальный. ■

## Доказательство, что кратчайшее расстояние в остаточной сети не уменьшается.

**Лемма.** Если в сети $G(V, E)$ с истоком $s$ и стоком $t$ увеличение потока производится вдоль кратчайших $s \rightsquigarrow t$ путей в $G_f$, то для всех вершин $v \in V \backslash \{s, t\}$ длина кратчайшего пути $\delta_f(s, v)$ в остаточной сети $G_f$ не убывает после каждого увеличения потока.

**Доказательство:**

Предположим противное, пусть существует вершина $v \in V\backslash\{s,t\}$ такая, что после какого-то увеличения потока длина кратчайшего пути из $s$ в $v$ уменьшилась. Обозначим потоки до и после увеличения соответственно за $f$ и $f'$. Пусть $v$ — вершина, расстояние $\delta'_f(s,v)$ от $s$ до которой минимально и уменьшилось с увеличением потока. Имеем $\delta'_f(s,v) < \delta_f(s,v)$. Рассмотрим путь $p = s \rightsquigarrow u \to v$, являющийся кратчайшим от $s$ к $v$ в $G'_f$. Тогда верно, что $\delta'_f(s,u) = \delta'_f(s,v) - 1$. По выбору $v$ и из предыдущего утверждения получаем, что $\delta'_f(s,u) \geqslant \delta_f(s,u)$.
Предположим $(u,v) \in E_f$, тогда $\delta_f(s,v) \leqslant \delta_f(s,u) + 1 \leqslant \delta'_f(s,u) + 1 = \delta'_f(s,v)$. Это противоречит $\delta'_f(s,v) < \delta_f(s,v)$.
Пусть $(u,v) \notin E_f$, но известно, что $(u,v) \in E'_f$. Появление ребра $(u,v)$ после увеличения потока означает увеличение потока по обратному ребру $(v,u)$. Увеличение потока производится вдоль кратчайшего пути, поэтому кратчайший путь из $s$ в $u$ вдоль которого происходило увеличения выглядит как $s \rightsquigarrow v \to u$, из чего получаем $\delta_f(s,v) = \delta_f(s,u) - 1 \leqslant \delta'_f(s,u) - 1 = \delta'(s,v) - 2$. Данное утверждение противоречит $\delta'_f(s,v) < \delta_f(s,v)$. Итак мы пришли к противоречию с существованием вершины $v$, кратчайшее расстояние до которой уменьшилось с увеличением потока. ■

## Общая оценка времени работы алгоритма Эдмондса-Карпа.

**Утверждение.** Количество итераций алгоритма Эдмондса-Карпа не превосходит $VE/2$

**Доказательство:**

На увеличивающем пути p есть ребро минимальной пропускной способности. Это ребро насыщается, исчезает из остаточной сети. Такое ребро будем называть критическим.
Ребро $(u,v)$ может быть критическим в следующий раз, если между этими моментами ребро $(v,u)$ лежало на пути p.

$$p'(s,u) = p'(s,v) + 1 \geqslant p(s,v) + 1 = p(s,u) + 2$$

То есть между двумя моментами "критичности"одного ребра расстояние до вершины эьтого ребра в ребрах увеличивается не менее чем на 2. Тогда ребро не может быть критическим более чем V/2 раз. На каждой итерации минимум одно ребро критическое ⇒ итераций не более $VE/2$
■

**Утверждение.** Алгоритм Эдмондса-Карпа работает за $O(VE^2)$

## Слоистая сеть. Алгоритм Диница. Оценка времени работы без доказательства.

**Def.** Слоистая сеть $G_L$ - подмножество сети G, включающее только те ребра $(u,v)$, для которых $p(s,v) = p(s,u) + 1$

**Def.** Блокирующий поток - такой поток в слоистой сети, что в слоистой сети не существует пути

из s в t по ненасыщенным ребрам.

**Algorithm.** Алгоритм Диница

1. Для каждого ребра $(u, v)$ данной сети $G$ зададим $f(u, v) = 0$.
2. Построим слоистую сеть $G_L$ из дополняющей сети $G_f$ данного графа $G$. Если $d[t] = \infty$, остановиться и вывести $f$.
3. Найдём блокирующий поток $f'$ в $G_L$. (просто dfs, но можно еще удалять ребра, по которым не прошли)
4. Дополним поток $f$ найденным потоком $f'$ и перейдём к шагу 2.

**Th.** Алгоритм Диница корректен

**Доказательство:**

Предположим, что в какой-то момент во вспомогательной сети, построенной для остаточной сети, не удалось найти блокирующий поток. Это означает, что сток вообще не достижим во вспомогательной сети из истока. Но поскольку она содержит в себе все кратчайшие пути из истока в остаточной сети, это в свою очередь означает, что в остаточной сети нет пути из истока в сток. Следовательно, применяя теорему Форда-Фалкерсона, получаем, что текущий поток в самом деле максимален. ■

**Утверждение.** Весь алгоритм Диница может выполняться за $O(VE^2)$ или за $O(V^2E)$. Также возможно достичь асимптотики $O(VE \log V)$, если использовать динамические деревья Слетора и Тарьяна.

# Паросочетания

**Def.** Паросочетание в графе - множество попарно несмежных ребер

**Def.** Паросочетание M в двудольном графе - произвольное множество рёбер двудольного графа, такое что никакие два ребра не имеют общей вершины.

**Def.** Вершины двудольного графа, инцедентные ребрам паросочетания M, называются покрытыми, а неинцидентные - свободными

**Def.** Рёберное покрытие графа — множество рёбер, в котором каждая вершина графа инцидентна по меньшей мере одному ребру покрытия.

**Def.** Числом реберного покрытия называется размер минимального реберного покрытия графа $G$ и обозначается через $\rho(G)$

**Def.** Число рёбер в наибольшем паросочетании графа $G$ называется числом паросочетания

**Def.** *Наибольшее* паросочетание — это такое паросочетание $M$ графа $G$, которое не содержится ни в каком другом паросочетании этого графа, то есть к нему невозможно добавить ни одно ребро, которое бы являлось несмежным ко всем рёбрам паросочетания

**Def.** *Максимальное* паросочетание — это паросочетание в графе $G$, содержащее максимальное количество ребер

**Def.** Паросочетание называется совершенным(полным), если оно покрывает все вершины графа

**Def.** Чередующаяся цепь - путь в двудольном графе, для любых двух соседних рёбер которого верно, что одно из них принадлежит паросочетанию, а другое нет

**Def.** Дополняющая цепь - чередующаяся цепь, у которой оба конца свободны.

**Def.** Уменьшающая цепь - чередующаяся цепь, у которой оба конца покрыты.

**Def.** Сбалансированная цепь - чередующаяся цепь, у которой один конец свободен, а другой покрыт.

## Теорема Берджа

**Th.** (Бержа) Паросочетание M в двудольном графе G является максимальным тогда и только тогда, когда в G нет дополняющей цепи.

**Доказательство:**

$\Rightarrow$

Пусть в двудольном графе $G$ с максимальным паросочетанием $M$ существует дополняющая цепь. Тогда пройдя по ней и заменив вдоль неё все рёбра, входящие в паросочетание, на невходящие и наоборот, мы получим большее паросочетание. То есть $M$ не являлось максимальным. Противоречие.

$\Leftarrow$

От противного. Пусть $M$ – не наибольшее. Пусть $M'$ – другое паросочетание, $|M'| > |M|$. Рассмотрим их симметрическую разницу. В таком графе все вершины имеют степень не больше 2. Найдется компонента, в которой ребер $M'$ больше. Это будет увеличивающий путь для $M$.

## Сведение задачи поиска максимального паросочетания в двудольном графе к задаче поиска максимального потока

**Algorithm.** Сведение задачи поиска максимального паросочетания в двудольном графе к задаче поиска максимлаьного потока

Сделаем исток $s$ и сток $t$. Проведем ребра из $s$ в первую долю и из второй доли в $t$. У всех ребер поставим пропускную способность 1. Найдем максимальный поток.

**Утверждение.** Ребра максимального потока - ребра максимального паросочетания

**Доказательство:**

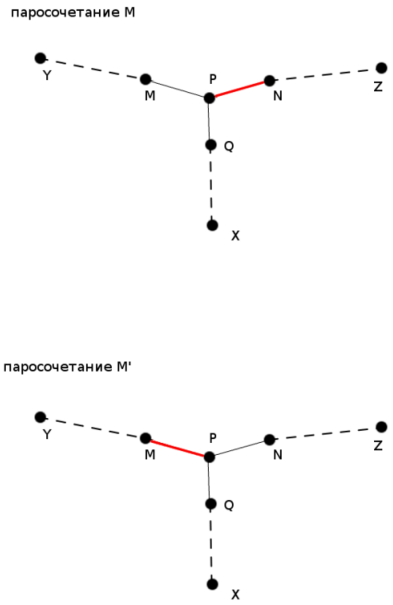
Покажем, что эти ребра образуют паросочетание. Пусть не так, тогда есть ребра, которые пересекаются по вершине x. Но тогда либо в x входит два ребра либо выходит два ребра. Оба случая невозможны в силу построения и пропускной способности в 1

Почему это паросочетание максимально? Пусть есть дополняющий путь. Тогда есть вершина в первой доли, в которую не приходит поток ($x_1$), и вершина во второй, из которой не исходит поток ($x_2$). Построим поток $s \to x_1 \to x_2 \to t$. Получится поток больше максимального ?!.

## Алгоритм Куна

**Th.** Если из вершины $x$ не существует дополняющей цепи относительно паросочетания $M$ и паросочетание $M'$ получается из $M$ изменением вдоль дополняющей цепи, тогда из $x$ не существует дополняющей цепи в $M'$.

**Доказательство:**




Доказательство от противного.

Допустим в паросочетание внесли изменения вдоль дополняющей цепи ($y \rightsquigarrow z$) и из вершины $x$ появилась дополняющая цепь. Заметим, что эта дополняющая цепь должна вершинно пересекаться с той цепью, вдоль которой вносились изменения, иначе такая же дополняющая цепь из $x$ существовала и в исходном паросочетании.

Пусть $p$ – ближайшая к $x$ вершина, которая принадлежит и новой дополняющей цепи и цепи ($y \rightsquigarrow z$). Тогда $MP$ – последнее ребро на отрезке ($y \rightsquigarrow p$) цепи ($y \rightsquigarrow z$), $NP$ – последнее ребро на

отрезке $(z \rightsquigarrow p)$ цепи $(y \rightsquigarrow z)$, $QP$ - последнее ребро лежащее на отрезке $(x \rightsquigarrow p)$ новой дополняющей цепи.

Допустим $MP$ принадлежит паросочетанию $M'$, тогда $NP$ ему не принадлежит.<br> (Случай, когда $NP$ принадлежит паросочетанию $M'$ полностью симметричен.)

Поскольку паросочетание $M'$ получается из $M$ изменением вдоль дополняющей цепи $(y \rightsquigarrow z)$, в паросочетание $M$ входило ребро $NP$, а ребро $MP$ нет. Кроме того, ребро $QP$ не лежит ни в исходном паросочетании $M$, ни в паросочетании $M'$, в противном случае оказалось бы, что вершина $p$ инцидентна нескольким рёбрам из паросочетания, что противоречит определению паросочетания.

Тогда заметим, что цепь $(x \rightsquigarrow z)$, полученная объединением цепей $(x \rightsquigarrow p)$ и $(p \rightsquigarrow z)$, по определению будет дополняющей в паросочетании $M$, что приводит к противоречию, поскольку в паросочетании $M$ из вершины $x$ не существует дополняющей цепи.

**Algorithm.**

Задан граф $G\langle V, E\rangle$, про который известно, что он двудольный, но разбиение не задано явно. Требуется найти наибольшее паросочетание в нём

Алгоритм можно описать так: сначала возьмём пустое паросочетание, а потом — пока в графе удаётся найти увеличивающую цепь, — будем выполнять чередование паросочетания вдоль этой цепи, и повторять процесс поиска увеличивающей цепи. Как только такую цепь найти не удалось — процесс останавливаем, — текущее паросочетание и есть максимальное.

В массиве `matching` хранятся паросочетания $(v, \texttt{matching}[v])$ (Если паросочетания с вершиной $v$ не существует, то $\texttt{matching}[v] = -1$). А *used* — обычный массив "посещённостей"вершин в обходе в глубину (он нужен, чтобы обход в глубину не заходил в одну вершину дважды). Функция dfs возвращает *true*, если ей удалось найти увеличивающую цепь из вершины $v$, при этом считается, что эта функция уже произвела чередование паросочетания вдоль найденной цепи.

Внутри функции просматриваются все рёбра, исходящие из вершины $v$, и затем проверяется: если это ребро ведёт в ненасыщенную вершину *to*, либо если эта вершина *to* насыщена, но удаётся найти увеличивающую цепь рекурсивным запуском из $\texttt{matching}[to]$, то мы говорим, что мы нашли увеличивающую цепь, и перед возвратом из функции с результатом *true* производим чередование в текущем ребре: перенаправляем ребро, смежное с *to*, в вершину $v$.

В основной программе сначала указывается, что текущее паросочетание — пустое (массив `matching` заполняется числами $-1$). Затем перебирается вершина $v$, и из неё запускается обход в глубину dfs, предварительно обнулив массив *used*.

```
bool dfs(v: int):
        if (used[v])
                return ''false''
        used[v] = ''true''
        for to in g[v]
```

```
        if (matching[to] == -1 or dfs(matching[to])):
                matching[to] = v
                return ''true''
        return ''false''

function main():
        fill(matching, -1)
        for i = 1..n
        fill(used, ''false'')
        dfs(i)
        for i = 1..n
        if (matching[i] != -1)
                print(i, " ", matching[i])
```

Итак, алгоритм Куна можно представить как серию из $n$ запусков обхода в глубину на всём графе. Следовательно, всего этот алгоритм исполняется за время $O(nm)$, где $m$ — количество рёбер, что в худшем случае есть $O(n^3)$ Если явно задано разбиение графа на две доли размером $n_1$ и $n_2$, то можно запускать `dfs` только из вершин первой доли, поэтому весь алгоритм исполняется за время $O(n_1 m)$. В худшем случае это составляет $O(n_1^2 n_2)$.

# Структуры данных

Следующие структуры данных будут предложены для решения задач *RSQ* (нахождение суммы на подотрезке)) и *RMQ* (нахождение минимума на подотрезке). Основными характеристиками этих задач являются:

- Можно ли изменять исходные элементы? Если да, то это *dynamic* версия, иначе — *static*
- Известны ли заранее все вопросы. Если да, то это $offline$ версия, иначе — *online*.

Решим задачу *online static RMQ* с помощью структуры данных *sparse table*.

## Разреженная таблица

Разреженная таблица — двумерная структура данных $ST[i][j]$, построенная на бинарной операции $F$, для которой выполнено следующее:

$ST[i][j] = F(A[i], A[i+1], \ldots, A[i+2^j-1]), \quad j \in [0 \ldots \log N]$.

Иначе говоря, в этой таблице хранятся результаты функции $F$ на всех отрезках, длины которых равны степеням двойки. Объём памяти, занимаемый таблицей, равен $O(N \log N)$, и заполненными

являются только те элементы, для которых $i + 2^j \leqslant N$.

Простой метод построения таблицы заключён в следующем рекуррентном соотношении:

$$ST[i][j] = \begin{cases} F(ST[i][j-1], ST[i+2^{j-1}][j-1]), & \text{если } j > 0; \\ A[i], & \text{если } j = 0; \end{cases}$$

Заметим, что для корректности данного соотношения необходимо, чтобы операция $F$ удовлетворяла следующим свойствам:

- Коммутативность $F(a, b) = F(b, a)$
- Ассоциативность $F(a, F(b, c)) = F(F(a, b), c)$
- Идемпотентность $F(a, a) = a$

Заметим, что задачу $RSQ$ нельзя решать с помощью этой структуры, так как сумма неидемпотентна.

Выполним сначала предподсчет, суть которого в вычислении массива $fl\_log[j] = \lfloor \log_2 j \rfloor$. Таким образом, построение и предподсчет занимают $O(N \log N)$ времени.

Теперь заметим, что для отрека $[l, r]$ верно, что

$F(A[l], A[l+1], \ldots, A[r]) = F(ST[l][j], ST[r-2^j+1][j]$, где $j = fl\_log[r-l+1]$

Таким образом, ответ на запрос дается за $O(1)$.

В принципе, данным алгоритмом можно решать и оффлайн-версию данной задачи.

## Дерево отрезков

Дерево отрезков способно за $O(\log N)$ получать на подотрезке результат любой операции, которая ассоциативна, коммутативна и имеет нейтральный элемент. В частности, задача $RSQ$, как правило, решается с использованием этой структуры данных (о дереве Фенвика мы умолчим, так как в программе его нет, ну и в курсе оно не рассматривалось).

### Построение

Опишем нерекурсивное построение дерева на массиве длины $N$. Для удобства будем считать, что $\log N \in \mathbb{N}$, иначе дозаполним до степени двойки нейтральными элементами. Теперь заведем массив длины $2N - 1$ и будем его заполнять таким образом, что последние $N$ элементов будут элементами исходного массива, а первые $N-1$ элементов заполним следующим образом: $t[i] = F(t[2i+1], t[2i+2])$ (заполнение от элемента с номером $N - 2$ и до 0 в цикле)

Заполним массив дерева отрезов для операции минимум и массива:

| a[ ] | — | — | — | — | — | 7 | 1 | 8 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| t[ ] | 1 | 2 | 1 | 2 | 3 | 7 | 1 | 8 | 2 | 3 | 4 |

**Обработка операции снизу**

1. Если элемент, попавший на левую границу, является правым сыном, то запишем в результат значение, полученное после выполнения нашей операции над предыдущим результатом и значением этого элемента, а левую границу перемещаем на один элемент вправо. Аналогично с правой границей (является ли она левым сыном). Таким образом мы учтем вклад нужной нам вершины и избавимся от вклада ненужного нам поддерева.

2. Устанавливаем границы отрезка на родительские элементы текущих границ. Это позволит узнать, входит ли полученный отрезок в искомый или нет. Повторяем этап 1, пока границы не пересекутся.

3. Если после завершения цикла границы совпадут, значит полученный отрезок входит в искомый, и надо пересчитать результат.

Для большей понимабельности процесса приведем псевдокод алгоритма (переменные $left, right$ указывают на индексы **листьев**, $f$ — исходная операция, на которой строилось дерево):

```
int query(left: int, right: int):
    leftRes = neutral
    rightRes = neutral
    while (left < right)
        if (left % 2 == 0)
            leftRes = f(leftRes, t[left])
        left \= 2
        if (right % 2 == 1)
            rightRes = f(t[right], rightRes)
        right = right \ 2 - 1
    if (left == right)
        leftRes = f(leftRes, t[left])
    return f(leftRes, rightRes)
```

**Обработка операции сверху**

Внимание, тут будет дерево отрезков храниться не как массив результатов, а как настоящее дерево из узлов с указателями на детей. Приведем сразу псевдокод, а потом поясним его:

```
int query(int node, int a, int b)
```

```
l = tree[node].left
r = tree[node].right
if (intersection([l, r),  [a, b)) is empty)
    return neutral
if ([l, r) is subset [a, b))
    return tree[node].res
return f(query(node * 2 + 1, a, b), query(node * 2 + 2, a, b))
```

Теперь поясним, что тут происходит. Возможны три ситуации:

- Полуинтервал, за который отвечает вершина, не пересекается с искомым, значит вернем нейтральный элемент
- Полуинтервал, за который отвечает вершина, входит целиком в искомый, значит вернем значение в этой вершине
- Иначе вернем результат операции на детях

Заметим, что по сути мы разбиваем исходный интервал на дизъюнктное объединение интервалов таких, что глубина вершин, отвечающих за каждый «подынтервал» минимальна. То есть спускаемся от корня вниз, сливаем результаты в поддеревьях, пока не поднимемся до корня.

### Обновление элемента

Рассмотрим на примере, какие индексы надо изменить:

| i[] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| a[] | — | — | — | — | — | 7 | 1 | 8 | 2 | 3 | 4 |
| t[] | 1 | 2 | 1 | 2 | 3 | 7 | 1 | 8 | 2 | 3 | 4 |
| $a'$[] | — | — | — | — | — | 7 | 1 | 8 | **6** | 3 | 4 |
| $t'$[] | **1** | **1** | 1 | **6** | 3 | 7 | 1 | 8 | **6** | 3 | 4 |

Тогда зная индекс листа $i$, который надо изменить, заметим, что индекс элемента, который придется заменить: $\lfloor \frac{i-1}{2} \rfloor$. Тогда можно просто написать функцию, аналогичную подъему по дереву во время построения снизу, только она будет обновлять значения от листьев до корней, пересчитывая, какой элемент надо изменить, а потом высчитывать его новое значение (заметим, что все значения глубины ниже уже известны).

### Массовое (групповое) обновление

Договоримся, что будут две функции: *op* — функция, по которой строили дерево отрезков, и функция *ch*, по ней будут обновлять **отрезки** элементов исходного массива. Также договоримся, что помимо

ограничений на операцию *op* (ассоциативность, коммутативность и существование нейтрального), на операцию *ch* также наложим ограничения:

1. Существование нейтрального элемента
2. $ch(a, ch(b, c)) = ch(ch(a, b), c)$
3. $ch(op(a, b), c) = op(ch(a, c), ch(b, c))$

В каждой вершине, помимо непосредственно результата выполнения операции *ch*, будем хранить несогласованность — величину, с которой нужно выполнить операцию *op* для всех элементов текущего отрезка. Тем самым мы сможем обрабатывать запрос массового обновления на любом подотрезке эффективно, вместо того чтобы изменять все $O(N)$ значений. Как известно из определения несогласованных поддеревьев, в текущий момент времени не в каждой вершине дерева хранится истинное значение, однако когда мы обращаемся к текущему элементу мы работаем с верными данными. Это обеспечивается «проталкиванием» несогласованности детям (процедура *push*) при каждом обращений к текущей вершине. При этом после обращения к вершине необходимо пересчитать значение по операции *ch*, так как значение в детях могло измениться.

Опишем процедуры для осуществления задуманного:

**«Проталкивание» несогласованности** Рассмотрим проталкивание, цель которого сделать несогласованность текущей вершины нейтральной

```
void push(int node) {
        tree[2 * node + 1].d = ch(tree[2 * node + 1].d, tree[node].d);
        tree[2 * node + 2].d = ch(tree[2 * node + 2].d, tree[node].d;
        tree[node].d = ch_neutral
}
```

**Обновление** Процедура обновления на отрезке. Данная процедура выполняет разбиение текущего отрезка на подотрезки и обновление в них несогласованности. Очень важно выполнить push как только идет рекурсивный вызов от детей, чтобы избежать некорректной обработки в детях. И так как значение в детях могло измениться, то необходимо выполнить обновление ответа по операции *op* на текущем отрезке.

```
void update(int node, int a, int b, T val) {
        l = tree[node].left;
        r = tree[node].right;
        if ([l, r) intersect [a, b) is empty)
            return;
```

```
        if ([l, r) is subset [a, b)
            tree[node].d = ch(tree[node].d, val);
            return;
        (1)
        push(node);
        (2)
        update(2 * node + 1, a, b, val);
        update(2 * node + 2, a, b, val);
        (3)
        tree[node].ans = op(ch(tree[2 * node + 1].ans, tree[2 * node + 1].d),
                            ch(tree[2 * node + 2].ans, tree[2 * node + 2].d));
}
```

Прокомментируем данный код:

1. Разбили искомый отрезок на дизъюнктное объединение интервалов
2. Протолкнули несогласованность в детей
3. Заметим, что на данный момент в узле несогласованность нейтральна(!), а значит нам нужно получить истинный ответ в этом узле, который получается путем ответов на запрос для детей (заметим, что тут мы гарантируем тот факт, что при проталкивании мы протолкнули определенность так, чтобы измененные несогласованности давали верный ответ (верно из свойств операции $ch$)

**Получение запроса** Осталось рассмотреть последнюю (и самую нужную операцию) — получение ответа на запрос. Заметим, что в ней логика та же, что и в обновлении, только возвращаются другие значения (в данном случае возвращаются ответы из поддеревьев), а в обновлении выполняли проталкивание несогласованности так, чтобы она появилась как можно выше в поддеревьях, в которых был запрос на изменение.

```
int query(int node, int a, int b) {
        l = tree[node].left;
        r = tree[node].right;
        if  ([l, r) intersect [a, b) is empty)
            return op_neutral;
        if ([l, r) is subset [a, b)
            return ch(tree[node].d, tree[node].ans);
        push(node);
        ans = op(query(node * 2 + 1, a, b), query(node * 2 + 2, a, b));
        tree[node].ans = op(ch(tree[2 * node + 1].ans, tree[2 * node + 1].d),
```

```
                                ch(tree[2 * node + 2].ans, tree[2 * node + 2].d));
        return ans;
}
```

## LCA или RMQ, вот в чем вопрос

### Задача LCA

Пусть дано дерево $T$, подвешенное за вершину $r$. Тогда назовем наименьшим общим предком двух вершин $u, v$ $LCA(u, v)$ такую вершину $X$, что она лежит на путях $u \to r$ и $v \to r$, при этом такая вершина глубже всех подходящих.

**Algorithm.** Наивное решение.

Решим задачу максимально просто, а именно для каждой вершины за линейное время найдем ее глубину, поднимемся до уровня той, что выше, потом одновременный подъем, пока не добрались до общего предка. Сложность линейная на запрос.

**Algorithm.** Метод двоичных подъемов

Сделаем предподсчет двумерной матрицы $dp[v][i]$ — номер вершины, в которую мы придём если пройдём из вершины $v$ вверх по подвешенному дереву $2^i$ шагов, причём если мы пришли в корень, то мы там и останемся. Для этого сначала обойдем дерево в глубину, и для каждой вершины запишем номер её родителя $p[v]$ и глубину вершины в подвешенном дереве $d[v]$. Если $v$ — корень, то $p[v] = v$. Тогда для функции $dp$ есть рекуррентная формула:

$$dp[v][i] = \begin{cases} p[v] & i = 0, \\ dp[dp[v][i-1]][i-1] & i > 0. \end{cases}$$

Заметим, что в силу определения $i \leq \log_2 n$, так как иначе мы остаемся в корне. Таким образом, матрица $dp[][]$ занимает $O(N \log N)$ памяти, ее построение занимает $O(N \log N)$ времени.

Теперь рассмотрим, как будет происходить ответ на запрос. Пусть $c = LCA(v, u)$, тогда по определению $c \leq min(d[v], d[u])$. Пусть $d[u] < d[v]$, тогда нам надо подняться вверх на $d[v] - d[u]$ шагов. Как это быстро сделать? Разложим разность глубин по степеням двойки и заметим, что в силу определения $dp$, проходя последовательно вверх на эти степени, мы поднимемся на тот же уровень, что и вершина $u$. Этот шаг занимает $O(\log N)$ времени.

Теперь будем считать, что $d[u] = d[v], u \neq v$. Заведем счетчик $k = \log_2 N$ и будем уменьшать его, поднимаясь обеими вершинами, пока они не сравняются. Заметим, что данный метод уравняет высоты вершин (то есть найдет самого глубокого предка) за $O(\log N)$ времени.

Таким образом, данный метод имеет предподсчет за $O(N \log N)$ и ответ на запрос за $O(\log N)$ времени. При этом памяти тратится $O(N \log N)$.

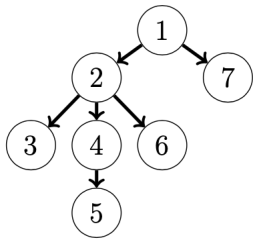
### Сведение LCA к RMQ

**Algorithm.** Сведение $LCA$ к $RMQ$.

Пускай задано дерево $T$, тогда запустим $DFS$ из корня, по мере которого заполним следующие массивы:

1. Массив глубин $h$, в котором для каждой вершины в порядке обхода посчитаем высоту
2. Список посещений узлов $Order$, в который вершина добавляется каждый раз в порядке обхода $DFS$ (массив обход ао втором обходе на картинке)
3. Массив $First$, который заполняется индексами в массиве $Order$, когда мы встречаем вершину с номером $i$

На картинке ниже нас интересует информация о втором и третьем обходах:




| | |
|---|---|
| Эйлеров обход 1-го типа (обычный) | (1,2) (2,3) (3,2) (2,4) (4,5) (5,4) (4,2) (2,6) (6,2) (2,1) (1,7) (7,1) |
| Эйлеров обход второго типа (±1) | Обход: 1 2 3 2 4 5 4 2 6 2 1 7 1<br>Высоты: 0 1 2 1 2 3 2 1 2 1 0 1 0 |
| Обход третьего типа | Обход: 1 2 3 4 5 6 7 |

Массив $H = [0, 1, 2, 2, 3, 2, 1]$. Массив $First = [1, 2, 3, 5, 6, 9, 12]$.

Заметим, что ответом на запрос $LCA(u, v) = rmq(Order, First[u], First[v])$. Например (пример по картинке выше в 1-индексации, спасибо Копелиовичу), рассчитаем:

$LCA(4, 5) = rmq(Order, First[4], First[5]) = rmq(Order, 5, 6) = 4$.

$LCA(2, 7) = rmq(Order, First[2], First[7]) = rmq(Order, 2, 12) = 1$.

**Доказательство:** $DFS$ по пути из $u$ в $v$ пройдёт через $LCA$. Это будет вершина минимальной высоты на пути, так как, чтобы попасть в ещё меньшие, $DFS$ должен сперва выйти из $LCA$.

Таким образом, данный метод имеет предподсчет за $O(N)$ и ответ на запрос за $O(\log N)$ времени. При этом памяти тратится $O(N)$.

**Note.** Можно воспользоваться алгоритмом Фараха-Колтона-Бендера для решения данной задачи (это решение задачи $RMQ$, где модуль разности соседних элементов равен единице), так как элементы в массиве глубин различаются как раз не более, чем на 1. Данный алгоритм позволяет с предподсчетом за $O(N)$ отвечать на запрос за $O(1)$.

### Сведение RMQ к LCA

**Algorithm.** Сведение задачи $RMQ$ к $LCA$

Алгоритм очень простой: построим декартово дерево по неявному ключу на данном массиве, обозначим $A$ как массив вершин (ниже поясним, что это такое). Тогда по определению такого дерева получим, что $RMQ(i, j) = LCA(A[i], A[j])$.

Учитывая, что существует построение за линейное время (предподсчет за $O(N)$ времени) и используя решение задачи $LCA$, получим, что отвечать на запрос можно также за $O(\log N)$ времени ($O(1)$) при использовании и готовности рассказывать Фараха-Колтона-Бендера).

## Декартово дерево по неявному ключу

**Def.** Декартово дерево по неявному ключу — структура данных, способная выполнять функционал расширенного вектора, то есть способная делать вставку не только в конец, но и в произвольное место, аналогично с удалением. То есть такое дерево можно считать быстрым динамическим массивом.

Авторы надеются, что читатели помнят, что такое обычное декартово дерево (правда же?), поэтому перейдем сразу к делу.

Введем вспомогательную величину $C$ для каждого узла, смысл которой в том, что в поддереве данной вершины (включая ее саму) находится $C$ вершин. Тогда мы спокойно можем знать размеры правого и левого поддеревьев, что очень полезно будет для нас.

Вспомним, что для поддержания структуры декартового дерева использовались две операции: $split$ — разделить дерево на два и $merge$ — слить два дерева. Заметим, что операция $merge$ не требует модификаций, тогда как операцию разделения придется модифицировать.

**Algorithm.** Разделение дерева на два

$split(root, k)$ — операция разделения дерева с корнем в вершине $root$ таким образом, чтобы отрезаны от дерева были ровно $k$ вершин.

Обозначим за $l$ число вершин в левом поддереве для данного корня, тогда возможны две ситуации:

- $l \geq k$, тогда запустим рекурсивно процедуру разделения так, чтобы новым корнем в процедуре стал корень левого поддерева, а число необходимых для отрезания вершин оставим неизменным (новым левым сыном бывшего корня станет правая часть ответа рекурсивного вызова)
- $l < k$, тогда сделаем все то же самое, только симметрично (ну и число вершин в процедуре должно быть $k - l - 1$

Заметим, что для каждой вершины, по которой прошлось разрезание, необходимо обновить ее параметр $C$, который считается по определению.

Для ответа на данный вопрос программы необходимо также научиться делать вставку и удаление, с чем вы прекрасно сами справитесь, разделив в нужных местах и слив в нужном порядке.