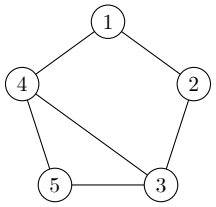
Содержание

1 Введение в графы.

Граф — это очень простая штука: кружочки, соединённые палочками. Разделяются два вида графов: ориентированные и неориентированные. Некоторым алгоритмам пофиг, где они работают, но большинству — нет.
Оба объекта полезны в хозяйстве. Неориентированные обычно показывают некоторое симметричное отношение. А ориентированные — какие-то состояния и переходы между ними.



Хранение графа. Первый способ — матрица смежности. Это просто табличка, где показаны рёбра.

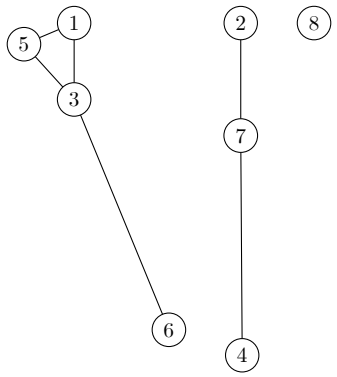
	1	2	3	4	5
1		1		1	
2	1		1		
3		1		1	1
4	1		1		1
5	1		1	1	

Когда вершин много, а рёбер — мало, табличка получается большой и слабоинформативной.
Вообще как мы в графах что-то оцениваем? У нас есть n — количество вершин и m — количество рёбер. И мы оцениваем алгоритмы относительно двух параметров. Но эти два параметра связаны. Если граф полон, в нём $m \approx n^2$, а если разреженный, то, вообще говоря, может быть и 0, но такие графы не очень осмысленны, поэтому мы будем рассматривать случай, когда $m = \mathrm{O}(n)$.
Хорошо, второй способ: списки смежности.

$$
\begin{aligned}
1 &: [2, 4] \\
2 &: [1, 3] \\
3 &: [2, 4, 5] \\
4 &: [1, 3, 5] \\
5 &: [3, 4]
\end{aligned}
$$

Это существенно удобнее, потому что занимает $\mathrm{O}(m)$ памяти и позволяет быстро перебрать рёбра, исходящие из данной вершины. И чаще всего мы будем использовать этот способ.

2 Depth first search (поиск в глубину).

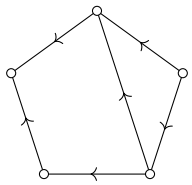


DFS работает так. Давайте возьмём вершину, пометим её. Переберём соседей: для каждого рекурсивно запустимся (то есть пометим их и запустимся от всех соседей). При этом запускаемся мы только от тех, кого ещё не пометили.

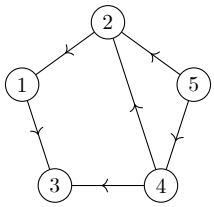
```
def dfs(v):
    mark[v] = True
    for u in graph[v]:
        if not mark[u]:
            dfs(u)
```

Утверждение: если запустить этот алгоритм от вершины v, он пометит все вершины её компоненты связности и только их. То что только их, тривиально, если мы пришли в вершину, мы в неё пришли из начальной. Почему все? Возьмём вершину u. Существует путь от v до v. При этом первая не помечена, а вторая — помечена. Значит где-то на пути есть две соседних вершины с разной помеченностью. А это противоречие, потому что непонятно, как почему мы из помеченной не пошли в непомеченную.

Немного про ориентированные графы. Орграфы бывают разные: либо циклы в них есть, либо нет. И в ациклических жить обычно сильно проще. Например, ациклический граф можно топологически отсортировать:



Вершины этого графа можно отсортировать так, чтобы рёбра шли от большего номера к меньшему:



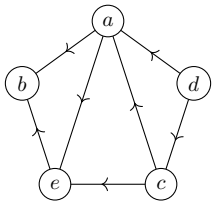
Искать такую сортировку очень легко: ищем вершину, в которую никто не входит, помечаем её максимальным индексом, удаляем из графа, повторяем.
Очень легко можно доказать, что вершина, из которой никто не выходит, в ациклическом графе есть, а

также тривиально, что этот алгоритм делает именно то, что нам нужно.
Теперь как это чудо быстро написать?

```
# zero --- множество/стек/список/... вершин с нулевой степенью.
for i in range(n):
    v = zero.pop()
    topsort.append(v)
    for u in graph[v]:
        deg[u] -= 1
        if deg[u] == 0:
            zero.append(u)
```

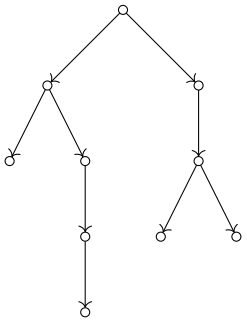
Сложность этого $O(m)$.

Топологическая сортировка с помощью DFS. Рассмотрим вот такой граф:

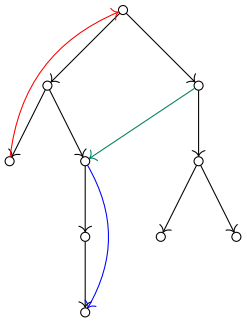


Начнём из вершины a. В DFS в тот момент, когда возвращаемся, запихиваем вершину в массив. Например тут мы идём из a в b, добавляем b в массив. Потом из a идём в e, в b уже не идём, добавляем в массив e. Добавляем саму вершину a. Дальше запускаем DFS, например, из d, идём по рёбрам в c, добавляем её, потом d. Получаем правильный ответ.
Как доказать, что это правильно? Да тривиально, мы кладём вершину в массив тогда, когда положили всех, откуда из неё ведёт ребро. А как мы пользуемся отсутствием циклов? А тут мы по сути можем придти вершину, в которую уже входили. То есть уже помеченную, но ещё не добавленную в массив. А это плохо, тут доказательство и ломается.
А как проверить наличие цикла в графе? Можно запустить DFS и детектить вершины, в которые мы уже зашли, но ещё не вышли обратно. Можно запустить первый алгоритм по поиску топологической сортировки, и нам рано или поздно скажут, что `zero` пуст.

Ориентированные графы с циклами. А давайте посмотрим, как выглядит DFS в графе с циклами? А точнее, просто посмотрим на DFS и узнаем, по каким рёбрам мы не ходили. Вот пусть мы ходили так:

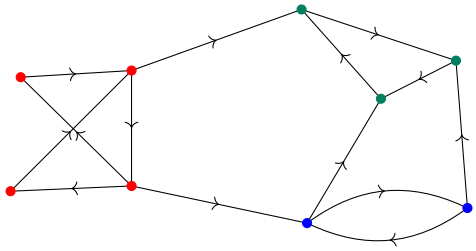


Тогда у нас могут быть рёбра «вниз» (синие), «вверх» (красные) и «влево» (зелёные). «Вправо» не может быть, иначе мы бы по ним прошли.

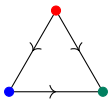


И циклы нам, как несложно заметить, дают только «вверх».

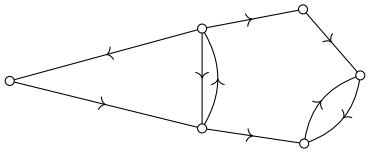
Сильная связность. Вершины u и v сильно связны, если есть путь от первой во вторую и есть путь от во второй в первую. Несложно доказать, что это отношение эквивалентности, а значит существуют компоненты сильной связности.



По этому чуду можно построить «конденсацию» — слияние каждой компоненты сильной связности в отдельную вершинку.



Она всегда ациклична, и обычно на ней строят топологическую сортировку, которая помогает решать задачу.
Как искать компоненты сильной связности? Есть алгоритм Косарайю. Выглядит алгоритм так. Давайте возьмём наш знакомый алгоритм, который в конце DFS'а добавляет вершину в список. Как он будет работать на таком графе?



Запустим DFS из, например, первой вершины. Получим массив $[1, 6, 5, 2, 3, 4]$. А теперь смотрите, что делаем. Давайте идти по этому массиву и запускать обычный DFS по **обратным рёбрам**. То есть вот что. Рассмотрим 1. Из неё по обратным рёбрам дойдём до 2 и 5. Это его компонента сильной связности, вау. Следующая наша вершина — 6. Из неё дойдём только до 1, там мы уже были, значит выделим только саму 6. И опять мы правильно нашли компоненту сильной связности 6. 5 и 2 уже были, пропускаем. А запустив DFS из 3, пометим ещё и 4.
Теперь докажем, что это работает. Понятно, что при втором DFS'е компоненту сильной связности мы пометим точно (она явно вся достижима по обратным рёбрам). Вопрос лишь в том, почему не будет ничего лишнего. Ну, давайте посмотрим на первый DFS. И посмотрим на случайную компоненту сильной связности (A). Мы впервые пришли в неё в какой-то вершине v. Понятно, что мы выйдем из этой

вершины только тогда, когда обойдём всю компоненту. А ещё, выйдя из вершины v, мы закончили ещё и обработку всех компонент сильной связности B, в которые из A ведут рёбра. А теперь как это согласуется с нашим массивом? Да понятно, у нас первая вершина компоненты связности A идёт раньше, чем первая вершина B. В частности, обратный DFS из первой вершины пометит нам компоненту связности, в которую ничего не входит. Следующий обратный DFS пометит нам компоненту связности, в которую входят рёбра только из первой, а в неё мы не пойдём. И так далее.
Их этого доказательства, кстати, также следует, что алгоритм Косарайю даст нам не просто конденсацию, он даст её топологически отсортированной.
Понятно, что сложность этого — $O(m)$, потому что это два DFS'а.

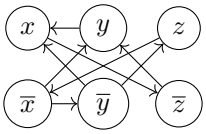
2-SAT Пусть у нас есть булева формула вида

$$(x \vee y) \wedge (\overline{x} \vee \overline{z}) \wedge (x \vee \overline{y}) \wedge (y \vee z)$$

Вам нужно найти переменных, когда это истина. Или доказать, что такого нет. При это 2-SAT решается быстро и легко вот так. Сначала сделаем вот так:

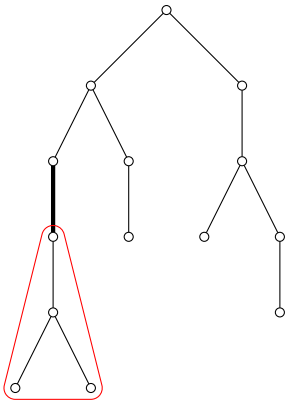
$$x \vee y \iff \overline{x} \to y \wedge \overline{y} \to x$$

И преобразуем всё, что у нас есть, в стрелочки. И эти стрелочки будут у нас рёбрами в графе. То есть у нас мы заведём граф



Давайте теперь заметим, что когда мы по рёбрам мы можем дойти $x \rightsquigarrow \overline{x} \rightsquigarrow x$, то решений точно нет. То есть если x и $\overline{x}$ находятся в одной компоненте сильной связности. Теперь давайте докажем, что если такого нет, то задача разрешима. Как докажем? А вот как. Для начала заметим, что в одной компоненте связности все вершины имеют одну и ту же истинность. Теперь заметим, что у нас кососимметричный граф (то есть такой, что у нас вершины бьются по парам, и если из x идёт ребро в y, то для их парных ребро идёт обратно). А также заметим, что конденсация кососимметричного графа кососимметрична. Почему? Ну, сначала нам нужно доказать, что если две вершины в одной К.С.С., то симметричные им — тоже. Это, вообще говоря, тривиально. И из этого, честно говоря, тривиально, что конденсация кососимметрична. Но также не содержит циклов, что не может радовать. Это значит, что её можно топологически посортить, а потом начать слева направо раздавать ещё не проинициализированным переменным значение `False`. Почему это будет корректно? Ну, рассмотрим последнюю вершину. Её мы поставили `True`. Из неё ничего не выходит, а в её двойственную (`False`) ничего не входит. Значит мы могли так всё присвоить, вне зависимости от других вершин. Уберём нашу пару, вырежем её, везде в остальных местах повторим доказательства.

Двусвязность. Две вершины рёберно двусвязны, если есть два рёберно непересекающихся пути между ними. Это отношение эквивалентности, компоненты вершинной двусвязности соединены мостами, мосты — ребро, при удалении которого граф разваливается.
Два ребра вершинно двусвязны, если между их концами есть два непересекающихся пути, это тоже отношение эквивалентности, вершины, которые инциденты двум рёбрам из разных компонент — точки сочленения, и при их удалении граф также разваливается.
Это всё было на дискретке, читайте. А нам интересны алгоритмы. Например, как искать мосты. Заметим следующий факт: мост всегда находится в дереве обхода в глубину (нам же надо как-то перейти из одной компоненты вершинной двусвязности в другую). О'кей, рассмотрим дерево обхода

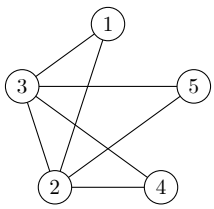


Нам надо как-то проверить, что из поддерева не бывает рёбер вверх. Так вот давайте хранить такую величину: `up[v]` — самая высокая вершина, в которую ведёт ребро откуда-то из поддерева `v`. Хранится либо номер (тогда у вершины выше должен быть номер меньше) вершины, либо время входа, что, по сути, одно и то же. Тогда как это считать? Да тривиально, мы берём все значения `up[u]` для `u` – ребёнок `v`, берём все рёбра из `v` вверх (кроме того, откуда мы пришли) и берём `t_in[v]`, и берём из всего этого минимум. Как тогда проверять на мост? Ну, смотрите, если `v` — ребёнок `p` и `up[v] == t_in[v]`, то мост. Иначе — не мост.
Как выделять сами компоненты связности? Ну, можно удалить мосты и запустить новый DFS. Или можно все вершины пихать в стек при входе, а потом при нахождении моста всё резко снимать.
Как искать точки сочленения? Ну, очень похоже. `v` — точка сочленения, если для какого-то её ребёнка `u` верно `up[u] < t_in[v]`. И всё это правда для всех вершин, кроме корня. С корнем всё просто, если у корня 2 ребёнка, он точка сочленения. Как искать компоненты? Точно также: пихаем в стек рёбро `(v, u)` при заходе в вершину `u`. Когда мы выходим из `u` и понимаем, что всё поддерево `u` отвалится, мы достаём из стека `(v, u)` и всё после, говорим, что компонента.

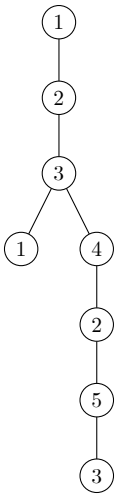
Эйлеров путь/цикл. Эйлеров путь/цикл — маршрут/замкнутый маршрут, проходящий по всем рёбрам единожды. Задача интересная что в ориентированных, что в неориентированных графах. Цикл есть тогда, когда граф связен и степени всех вершин чётны (входящая и исходящая степени всех вершин равны). Путь — когда это верно для всех вершин кроме двух (а в оставшихся двух входящая степень отличается от исходящей на 1, в одной на один больше, в другой — меньше).
Как это чудо искать? Давайте запустим пародию на DFS следующего толка. Мы метим не вершины, а рёбра. И происходит так:

```
def dfs(v):
    for u in graph[v]:
        if not marked[(v, u)]:
            marked[(v, u)] = True
            dfs(u)
```

При это наш эйлеров цикл состоит из рёбер, по которым мы возвращаемся. Представим следующий граф:



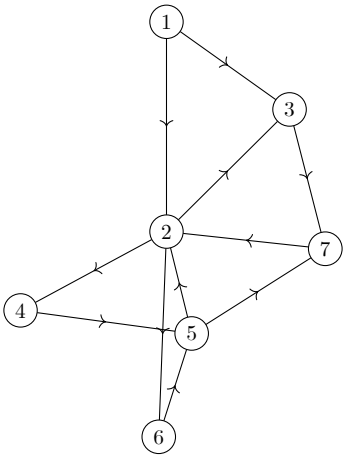
Когда мы запускаем эту пародию на DFS, мы получаем, например, такое дерево:



Хорошо, почему это даёт нам то, что нам нужно? Давайте докажем, что мы в любой момент имеем маршрут из добавленных рёбер. Докажем по индукции. База очевидна. Пусть мы имеем фрагмент пути. И мы как-то из последнего ребра оттуда куда-то пошли. Если мы пришли не в него, то по соображениям чётности можем пойти дальше. Если мы пришли в него, и нам некуда пойти, возвращаемся из рекурсии, добавляя ребро, которое, какая удача, соединено с `v`, то есть корректно продолжает наш путь. И, тривиально, мы пройдём все рёбра, найдя тем самым путь.

3 Дерево доминаторов.

Точки сочленения и мосты — это своего рода узкие места в графе. А в ориентированных графах всё со связностью хуже, но тоже хочется найти узкие места. То есть хочется найти такие вершины, что есть её удалить, то пропадёт путь.
Пусть у нас выделена вершина s. Пусть из неё достижимы все вершины. Давайте скажем, что одна вершина u доминирует другую v, если на любом пути из $s \rightsquigarrow v$ встречается u (то есть, если убрать вершину u, из s в v не будет пути). При этом сама вершина своим доминатором не является.



Есть путь $1 \to 3 \to 7$, то есть 7 если кто-то и доминирует, то 1 или 3. 1 доминирует 7? Да. А 3? Нет, ведь можно обойти. А вот 5 доминируется 2, например.
Давайте что-нибудь докажем.

Утверждение. *Если u доминирует v, а v доминирует w, то u доминирует w.*

Это тривиально. А что менее тривиально,

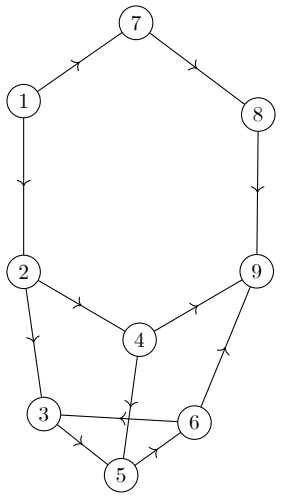
Вершина	Непосредственный доминатор
2	1
3	2
4	2
5	2
6	5
7	1
8	7
9	1

Утверждение. *Если x и y доминируют v, то одна из x и y доминирует другую.*

Доказательство. Возьмём путь $s \rightsquigarrow v$. На нём встречаются и x и y. Пусть сначала x, потом y. Докажем, что x доминирует y. Если это не так, что есть путь $s \rightsquigarrow y$, не проходящий через x. Ну, тогда можно пройти по нему до y, и от y в v и получить, что x не доминирует v. □

Следствие 0.1. *В частности это значит, что x и y имеют строго определённый порядок.*

Что в итоге происходит? Рассмотрим вершину. У неё есть какие-то доминаторы. Нас интересует самый нижний из них (*непосредственный доминатор*), ведь если мы знаем его, мы можем узнать все доминаторы каждой вершины.

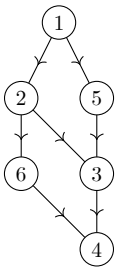


Теперь у кого какой непосредственный доминатор? Теперь мы хотим построить дерево доминаторов. Это бывает нужно, чтобы понять, какие вершины пори удалении приведут к чему-то плачевному. Там, garbage collector большой кусок съест или что ещё.
Алгоритм. Сначала рассмотрим ацикличный граф. Что с ним делаем? Ну, топсорт, объективно. Теперь мы построили какой-то кусок дерева доминатора. Нам надо взять новую вершину и понять, что делать. Ну, в неё идут какие-то рёбра из вершин раньше. Нам надо уметь искать наименьшего общего предка дерева доминаторов (для заданного набора вершин). И делаем мы это двоичными подъёмами.
Циклы всё усложняют. Рассмотрим v. Запустим DFS из s. Утверждается, что одна из вершин дерева DFS — непосредственный доминатор. Осталось лишь узнать, какая. Давайте снизу вверх пойдём и начнём доказывать, что данный предок — не доминатор. Когда найдём ту, для которой нельзя доказать, — это будет непосредственный доминатор.
Давайте занумеруем вершины в порядке времени входа. Это для простоты записи теорем дальше.

Теорема 1. *Пусть $u \rightsquigarrow v$ и $u < v$. Тогда на пути $u \rightsquigarrow v$ встречается предок v.*

Доказательство. Пусть мы стоим в вершине. Если она предок — доказали. Иначе мы находимся в одном из поддеревьев. У нас бывают рёбра только вверх, вниз и влево. Вниз нам ничего не дают, влево — попадаем в более левое поддерево, а вверх рано или поздно приведёт нас к предку. □

Итак, схема алгоритма. Давайте для каждой вершины построим полудоминатор, а потом, используя их, построим доминаторы. Что такое полудоминатор? u является полудоминатором v, если существует путь $u \rightsquigarrow v$, в котором все промежуточные вершины $> v$. Нас интересует только полудоминаторы, которые предки v. На картине это выглядит как то, что от u мы идём куда-то вправо, а потом по перекрёстным рёбрам идём в v.
Что нам дают полудоминаторы? А то, что все предки ниже их — явно не доминаторы. А значит нас интересует самый высокий полудоминатор. Тривиально, это один из предков. Также тривиально, что минимальный полудоминатор не выше непосредственного доминатора. Вот пример, когда это разные вершины:



Тут непосредственный доминатор 4 — это 1, а наименьший полудоминатор — 2.
Итак, давайте строить полудоминаторы. Пусть u — полудоминатор v. Тогда путь $u \rightsquigarrow v$ либо пуст, либо нет. Если пусть $u \rightsquigarrow v$ пуст, то все такие вершины подходят. Переберём их все, будет ок. Теперь переберём все такие w, что $w > v$ и $w \to v$. К сожалению, u не обязано быть полудоминатором w. Зато есть следующее утверждение:

Утверждение. *Если $u \rightsquigarrow w \to v$ — полудоминирующий путь, то и полудоминирует вершину x, такую что она лежит на этом пути и является предком w.*

Доказательство. Пусть y — наименьший общий предок w. $y < v$, значит на пути её нет. Давайте теперь в пути $u \rightsquigarrow v$ возьмём самую раннюю вершину из $y \rightsquigarrow w \setminus \{y\}$. Это будет x, хочется нам сказать. Надо доказать, что на $u \rightsquigarrow x$ все вершины $> x$. Если там вершина $< x$, то воспользуемся теоремой. То есть на пути от этой вершины до x есть предок x. А его там быть не может, потому что если это предок из пути $y \rightsquigarrow x$, то мы должны были взять его как x, а если предок из $u \rightsquigarrow y$, то у них номера меньше v. □

Утверждение. *Обратное тоже верно (если u является полудоминатором какой-то вершины из $u \rightsquigarrow w$, то является полудоминатором u).*

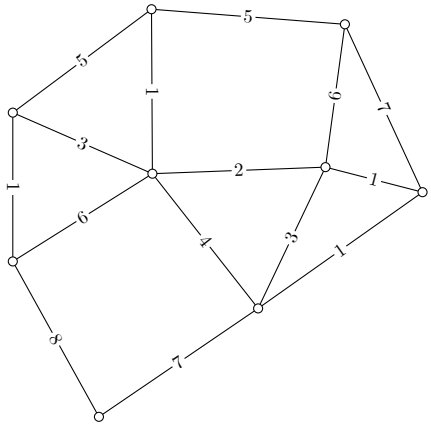
Доказательство. Очевидно. На пути $u \rightsquigarrow x \rightsquigarrow w \to v$ все вершины $> v$. □

Из чего тогда состоит алгоритм? Мы берём все w, берём y — наименьший общий предок v и w. Перебрав все вершины на пути $y \rightsquigarrow w$ и взяв по всем ним наименьшего полудоминатора, получим наименьший полудоминатор x.
Теперь как это считать? Ну, нужно считать вершины по убыванию времени входа, очевидно. А чтобы считать минимум, можно использовать двоичные подъёмы, правда считать двоичные подъёмы нужно лениво.
Теперь давайте из полудоминаторов сделаем доминаторы. Как? Возьмём вершину v и её минимального полудоминатора $\text{sdom}[v]$. Посмотрим внимательно на то, что строго ниже полудоминатора. И посмотрим на минимальные полудоминаторы всех вершин там. Есть два варианта: либо всё это ниже либо равно $\text{sdom}[v]$, то $\text{sdom}[v]$ — непосредственный доминатор v. Почему? Предположим его можно обойти. Пусть мы обошли, и на пути $\text{sdom}[v] \rightsquigarrow v$ пришли в u. Тогда тот момент, когда мы сошли с $s \rightsquigarrow \text{sdom}[v]$, мы нашли вершину, которая является полудоминатором u. Это, на самом деле, понятно от противного. Пусть какая-то из вершин $\text{sdom}[v] \rightsquigarrow v \setminus \{\text{sdom}[v]\}$ имеет своим полудоминатором вершину выше $\text{sdom}[v]$. Найдём вершину u с самым высоким полудоминатором. Если таких несколько, возьмём самую высокую. Тогда непосредственный доминатор v такой же, как и u. Ну, все вершины, не являющиеся

доминаторами u, не являются и доминаторами v. То есть надо доказать, что непосредственный доминатор u — доминатор v. Пусть нет. Тогда обходной путь должен входить в $u \rightsquigarrow v$, а значит у неё полудоминатор выше, чем у u.

4 Минимальное остовное дерево (MST).

Есть неориентированный взвешенный граф.



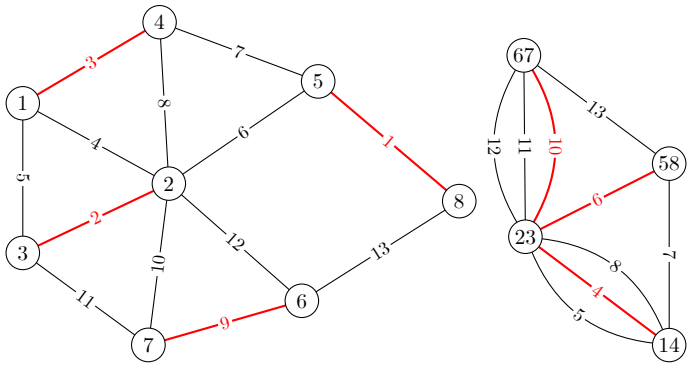
Лемма 1 (Лемма о разрезе)**.** *Пусть вершины нашего графа разбиты на A и B. Пусть они соединены какими-то рёбрами, и ребро минимального веса из них — uv. Тогда uv (или ребро с тем же весом) принадлежит MST.*

Доказательство. У нас в A есть кусок дерева, и в B — тоже. При этом через все вершины и A, и B дерево проходит. Если у нас в дереве не uv, то вырежем то ребро, которое соединяет A и B и добавим вместо него uv. □

Алгоритм Краскала . Возьмём минимальное ребро графа. Его можно взять в MST, так как оно — минимальный разрез одной из его вершин и всего остального. Потом возьмём ребро следующего веса. Оно отделяет первые 2 вершины от остального. И так далее: добавляем следующее по весу ребро в MST, если это ребро не соединяет уже связанные вершины. Делаем мы это при помощи CHM. Асимптотика — $O(m \log m)$, потому что сортировка.

Алгоритм Прима. Возьмём любую вершину, скажем, что она «ключевая», и будем растить дерево от неё. Возьмём вершину и всё остальное. Это разрез, берём минимальное ребро. Теперь у нас есть кусок дерева и всё остальное, теперь тоже берём ребро. Как это реализовывать? Ну, добавим рёбра в кучу. Что будет при добавлении ребра? Ну, во-первых, у нас некоторые рёбра будут некорректны, да ещё и рёбра добавятся. В тупой реализации имеем $m \log m$. Давайте как-нибудь оптимизируем. Давайте для каждой вершины, которая не в A хранить только минимальное ребро в неё. Что тогда делать на этапе добавления? Если есть ребро в новую вершину, добавить. Если есть ребро в уже существующую, сравнить веса, подменить. При этом если граф достаточно разреженный, вам поможет Фибоначчиева куча, а если почти полный — то лучше массив.

Алгоритм Борувки. Возьмём каждую вершину нашего графа. Минимальное ребро от каждой вершины точно лежит в MST. Возьмём все эти рёбра, добавим в MST и сожмём компоненты связности. Повторить до посинения.



Каждый раз мы уменьшаем количество вершин вдвое, на каждом этапе мы смотрим на m рёбер. Итого $m \log n$. Правда, у алгоритма есть следующая проблема: надо, чтобы все веса были разными. Иначе алгоритм Борувки может найти вам цикл.

Рассуждения по теме. Лучшая асимптотика у нас получилась $O(n \log n + m)$, в алгоритме Прима. Люди вообще умеют делать за $O(m\alpha(m; n))$, используя «слабые кучи». Люди умеют $O(m)$ рандомизированно. Можно ли за $O(m)$ честно, неизвестно. Но если можно, у нас уже есть алгоритм. ЧТО? А вот суть в том, что у нас уже есть алгоритм, мы знаем, что он оптимален, но никто не знает, какая у него асимптотика. Работает этот алгоритм ориентировочно так:

- Разобьём наш граф на кусочки размера $\log \log \log n$. Их не очень много типов.
- В каждом куске мы не можем сразу посчитать MST, но можем для каждой структуры взять дерево решения.
- Дальше применить вовне алгоритм на $O(m\alpha(m; n))$, который сократится до $O(m)$.

Алгоритм Каргера — Тарьяна — Клейна. Это тот, который рандомизированно за $O(m)$. Для начала применим к нашему графу G Борувку 3 раза. Там будет $\frac{n}{8}$ вершин и m рёбер. Возьмём полученный граф G' и удалим случайную половину рёбер. Построим в этом графе H минимальной остовный лес T рекурсивно. Наш лес T также является остовным лесом G. Давайте научимся говорить о рёбрах, что они не минимальны. Если ребро имеет больший вес, чем максимальное ребро пути между двумя вершинами в нашем лесу, то это ребро явно не в MST. Так, удалением рёбер из G', получим G''. Утверждается, что в нём в среднем будет мало рёбер. У нас есть рёбра T (их $\frac{n}{8}$) и остальные. Утверждается, что остальных будет примерно $\frac{n}{8}$.

Доказательство. Во-первых, мы точно удалим все рёбра $H \setminus T$. Теперь посмотрим на рёбра G' в порядке возрастания веса. Мы будем добавлять их в пустой граф, если они в H. Рассмотрим ребро, которые мы не смогли удалить в G''. Тогда, объективно, оно соединяет (в H) две компоненты связности. В каждой такой ситуации с вероятностью $\frac{1}{2}$ добавляем ребро в H (тем самым уменьшая количество компонент связности), а с вероятностью $\frac{1}{2}$ — удаляем его, и оно входит в те самые «остальные», что мы хотим посчитать. Уменьшить количество компонент связности можно $\frac{n}{8}$ раз. Утверждается, что в среднем мы насчитаем $\frac{n}{8}$. □

В среднем у нас $\frac{n}{4}$ рёбер. Давайте теперь возьмём G'', и сделаем с ним то же самое, что мы делали с G.
Все операции,что мы делали, линейны (кроме удаления плохих рёбер). Это, утверждается, тоже делается за линейное время, но это мы не будем обсуждать. Тогда докажем следующее:

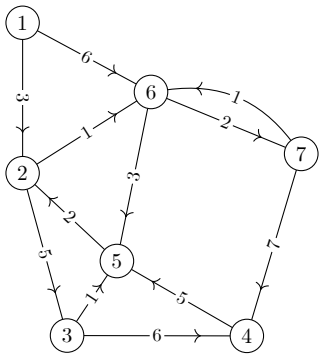
$$T(n, m) \leqslant an + bm$$

Ну,

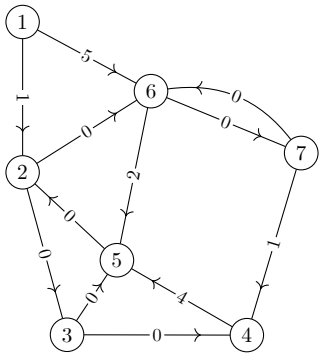
$$T(n, m) = \underbrace{n + m}_{\text{всё линейное}} + \left(a\frac{n}{8} + b\frac{m}{2}\right) + \left(a\frac{n}{8} + b\frac{n}{4}\right)$$

Взяв $b = 2$, $a = 2$, получим всё хорошо.

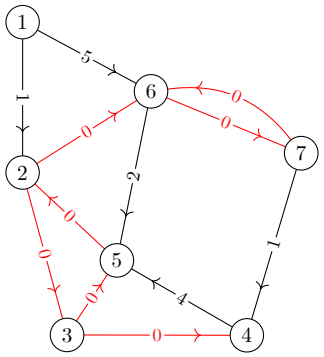
Minimal spanning arboresence. Есть у нас ориентированный граф.



Там есть выделенная вершина, и у каждого ребра есть вес. Надо построить ориентированное дерево с корнем в s и минимального веса. Ориентированное дерево — это когда мы можем добраться от s докуда угодно.
Тут не работает никакая хорошая лемма о разрезе, поэтому решения существенно сложнее. И мы знаем будем рассматривать алгоритм Эндмундса — Чу — Лю. Идея такая. Возьмём каку-то некорневую вершину. В неё входят какие-то рёбра. В ответе входящее ребро будет только одно. Давайте возьмём все эти рёбра и вычтем из их весов константу. Например, константу, равную минимуму из рёбер. Очевидно, что это н изменит минимальное дерево.

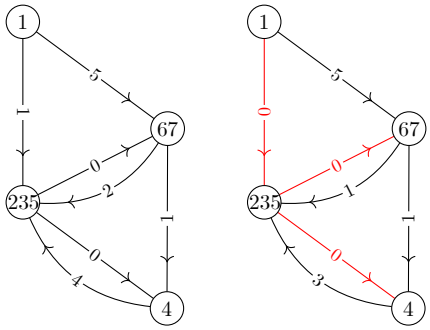


Теперь у каждой вершины есть ребро веса 0. Если их несколько, давайте оставим только одно произвольное.

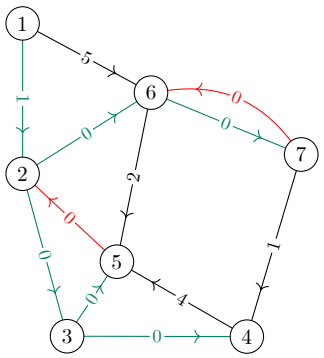


Дальше есть два случая: либо наши красные рёбра образуют дерево, либо нет. В первом случае мы, очевидно, победили. В нашем случае нет ребра из s, то есть это, объективно, не дерево.
В противном случае у нас есть цикл из красных рёбер. Почему? Ну, возьмём любую вершину. У каждой вершины есть входящее красное ребро. Пойдём по нему. Если мы придём в s (из любой вершины), мы

победили. Если не придём в s, то зациклимся. Что делаем с красным циклом? Сжимаем его целиком в одну вершину и сделаем то же самое для полученного графа.



Когда мы получим дерево, разожмём вершины обратно и ответом будет вот какое дерево: мы берём наше дерево в сжатом графе, переносим его рёбра на оригинальный граф, и из красных рёбер оставляем те, которые не противоречат.



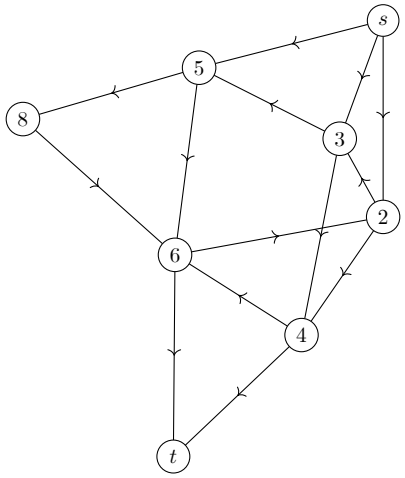
Что мы в итоге делаем? строим красные рёбра, проверяем на дерево, ищем цикл и сжимаем его. Всё это делается за $O(m)$. Проблема в том, что мы можем сжимать вершины n раз. Это долго.
Возможно делать это за $O(n \log n + m)$. Это совсем ужас, а нам хочется научится за $O(m \log n)$.
Давайте мы не будем перестраивать всё совсем с нуля при сжатии цикла. Возьмём вершину v и все входящие в неё рёбра. Вычтем минимальное из всех, получим нулевое ребро. Пойдёмте к вершине, из которой это нулевое ребро идёт. Сделаем с ней то же самое. Делаем это пока либо мы не придём в s (тогда мы нашли кусок финального дерева), либо не зациклились. Если дошли до s, выбираем новую вершину и делаем с ней то же самое (только нам уже не обязательно будет доходить до s, а можно будет до уже выделенной части итогового дерева). Проблемы будут у нас тогда, когда мы зациклились в процессе хождения. Ну так давайте не отходя от кассы сожмём наш цикл. И из полученной вершины пойдём дальше искать минимальное ребро.
Теперь как это чудо реализовывать? Ну, для сжатия очень легко использовать СНМ. Правда у нас во все места в асимптотике влезет $\alpha(m; n)$. Но это не единственное, что нам надо уметь: нам надо уметь брать рёбра, искать минимум и уменьшать всё. Вообще нам для этого хватит и кучи, просто рядом с кучей хранить вычитаемое. Но нам надо уметь сливать две кучи в одну. Это можно делать в биномиальной, левосторонней или Фибоначчевой кучей. Да можно хоть в обычной, асимптотика будет побольше, но пофиг. Итак, в каждой вершине храним все входящие рёбра в Фибоначчиевой куче. Сжимать кучи мы тоже будем не более n раз, итого получим $O(n)$ времени на сжатие. Но кучи нам не достаточно, на сложно будет прибавлять в куче что-то. Поэтому давайте возьмём ещё и ещё один СНМ, и рёбра хранить ещё и в нём. И в нём же хранить в каждом узле константу Δ, и тогда в куче мы при объединении в качестве сравнения элементов берём не сами веса, а доставая их из СНМ.
Можно вместо СНМ и куч брать декартово дерево, и тогда всё будет существенно проще, но асимптотика получится не той. Есть правда существенная проблема в петлях. Можно их убить моментально, а можно убивать лениво при получении ребра.

Чтобы увеличить асимптотику, придётся класть не все рёбра в кучу, а только активные. Активное ребро — только то, которое ведёт из одной вершины в самую максимально нижнюю вершину нашего пути. Более того, давайте будем ещё и для каждой вершины хранить отсортированный по активности список рёбер. Тогда петель будет не больше, чем сжатых вершин. Проблема в том, что активные рёбра меняются. Когда мы добавляем новую вершину в путь, из некоторой вершины в неё может идти ребро. Тогда прошлое активное ребро из той вершины перестанет быть активным, а новое становится. И мы в начало списка вставляем вершину. Когда же у нас сжимается цикл, из каждой его вершины может идти активное ребро, все они обратятся в петли, все их можно удалить. А из каждой не-его вершины может идти несколько рёбер внутрь цикла, и активным может быть не то, потому что нас интересует только наименьшее по весу ребро из них, а не самое низкое. Хочется взять конец нашего отсортированного списка и его сжать в минимальное ребро.

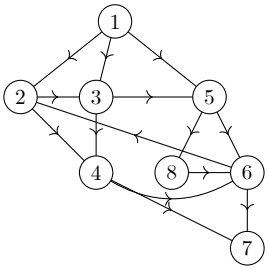
5 Кратчайшие пути в графах.

У нас есть граф и две вершины s и t, между которыми мы хотим найти кратчайший путь. «Кратчайший» — это либо про количество рёбер, либо про сумму весов на пути. Причём во втором случае есть принципиальная разница между графами, в которых есть отрицательные веса, и графами, когда таких нет.

Кратчайшие пути от выделенной вершины в невзвешеном графе. Это решается очень простым алгоритмом, называющемся BFS.



Давайте будем сначала искать вершины, расстояние до которых 1, потом те, до которых расстояние 2 и так далее. Как это делать? Ну, расстояние 1 — то, откуда можно придти из s (в нашем случае — 2, 3 и 5). Потом перебираем вершины с расстоянием 1, пытаемся из них куда-то пойти. Если мы там ещё не были, мы знаем, что там 2. И так далее.

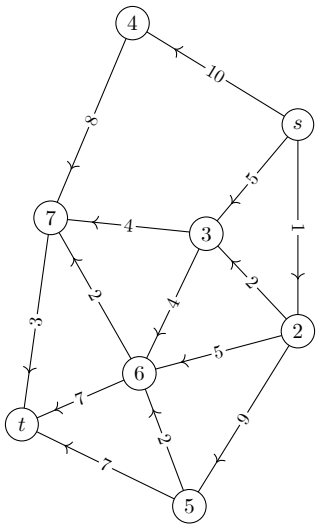


На самом деле, мы в этом алгоритме нашли расстояние от заданной вершины до всех. Это, на самом деле, свойство почти всех алгоритмов, которые ищут пути. Хотя, на самом деле, в случае BFS мы

могли и не искать растояние до всех вершин, а остановиться, когда дойдём до вершины t. Но в худшем случае это нам не очень поможет. Работать это будет за $O(n+m)$.
Как это реализовывать, кроме как втупую? Ну, смотрите, мы каждый раз по одному берём вершины с расстоянием k и добавляем вершины с расстоянием $k+1$. Так давайте вместо того, чтобы брать и добавлять вершины в разные места, будем делать всё это в одном.

```
d = [-1 for v in range(n)]
d[s] = 0
q = [s]
while len(q) > 0:
    v = q[0]
    del q[0]
    for u in graph[v]:
        if d[u] == -1:
            d[u] = d[v] + 1
            q.append(u)
```

Внимание, вопрос: а как узнать не только длину пути, но и сам путь? Да, ну, тривиально, просто в отдельном массиве сохранять, из какой вершины было последнее ребро кратчайшего пути до данной вершины. То, что мы получим, называется деревом кратчайших путей. Что ещё можно? Можно считать, что мы приходит в вершину не из одной предыдущей, а из всех из предыдущего слоя сразу. Тогда мы получим уже не дерево, в просто граф кратчайших путей. Также называется слоистой сетью. Устроен он так: у нас все вершины разбиваются на слои, и взятые нами рёбра идут с k-того уровня на $k+1$-й. А все не взятые рёбра идёт либо вверх, либо между вершинами одного уровня.
Всё это очень хорошо, но можно ли быстрее $O(n+m)$? Ну, интуитивно в общем случае нельзя. Но если у нас s и t близко, когда мы остановимся при нахождении s, мы не обойдём большой кусок графа, что радует. Но ещё можно идти одновременно от s и t в разные стороны. Тогда, в среднем, будет получаться быстрее, потому что ходя только из t мы обойдём максимум k^d вершин, а идя из двух мест — $k^{d/2}$, где d — расстояние, а k — средняя степень вершины.



Кратчайшие пути от выделенной вершины во взвешеном графе без отрицательных весов. Ну, мы уже знаем расстояние от s до s. Оно ноль. Для какой вершины мы ещё знаем? Ну, до той, откуда из s идёт минимальное ребро. В нашем случае 2. До неё расстояние 1. Хм-м, ну так что идейно делаем? Пусть мы уже знаем расстояния до каких-то вершин (множество A). И у нас есть рёбра, соединяющие эти вершины с остальными. Утверждается, что для ребра uv, где $d[u]+w_{uv}$ минимально расстоянием является именно что $d[u]+w_{uv}$. Почему? Ну, очевидно. Рассмотрим вершину v и её кратчайший путь

до s. Если он не содержит uv, значит он пересекает наши два множества в другом месте. То есть у нас есть путь $s \rightsquigarrow x \to y \rightsquigarrow v$, где $s \dots x \in A$, тогда путь $s \rightsquigarrow x \to y$ уже больше, чем $s \rightsquigarrow u \to v$, противоречие.
Как искать минимальное ребро? Давайте вот что делать. Для каждой вершины из A хранить $d[u]$ — расстояние. А для остальных вершин хранить текущий минимальный путь. В том же массиве будем хранить.

```
A = {s}
d = [weight[s][v] if v in graph[s] else infinity for v in range(n)]
while len(A) != n:
    v = # такое v вне A, что d[v] минимально
    A.add(v)
    for u in graph[v]:
        d[u] = min(d[u], d[v] + weight[u][v])
```

Как хорошо искать v? Ну, тут уже точно кучей. Мы n раз ищем и удаляем минимум, m раз уменьшаем вес. На двоичной куче это будет работать за $O(m \log n)$, на Фибоначчиевой куче — за $O(n \log n + m)$. На очень насыщенном графе можно применить вместо кучи использовать массив, и получится $O(n^2)$. Кстати, тут можно использовать ту же оптимизацию, когда мы идём с разных концов одновременно.
Но есть ещё одно соображение. Пусть мы знаем что-то про наш граф, и имеем оценку на расстояние со следующими свойствами:

$$\operatorname{dist}(v; t) \geqslant \varphi(v) \qquad \varphi(v) \leqslant w_{uv} + \varphi(u)$$

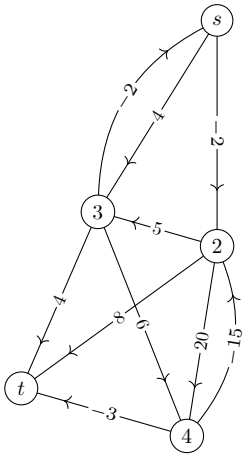
Например, вместо вершин есть точки на плоскости, и каждое ребро — его длина (между какими-то точками нет рёбер). Тогда в качестве φ можно взять расстояние между вершинами.
Давайте доставать вершины не в порядке возрастания d, а в порядке возрастания $d[v] + \varphi(v)$. Тогда мы по сути перебираем не весь граф, а только участок, недалеко уходящий от пути $s \rightsquigarrow v$.
Почему это вообще работает? Ну, вообще у нас просто получился граф с другими весами:

$$w'_{uv} = w_{uv} - \varphi(u) + \varphi(v)$$

А отсюда, у нас просто меняются веса всех путей таким образом:

$$\operatorname{dist}'(s, t) = \operatorname{dist}(s, t) - \varphi(s) + \varphi(t)$$

То есть длина всех путей меняется на константу, а значит кратчайший путь таким остаётся. Кстати, теперь понятно, зачем нам условие на $\varphi(v) \leqslant w_{uv} + \varphi(u)$, чтобы новые веса были неотрицательны, а значит чтобы работал алгоритм Дейкстры. Для особых ценителей это можно делать с двух сторон.



Кратчайшие пути от выделенной вершины во взвешеном графе без циклов отрицательного веса. Тут есть существенная проблема — цикл отрицательного веса $2 \to 3 \to 4 \to 2$. Тогда по нему мы можем ходить сколько угодно раз. А если нам надо найти простой путь, то к этой задаче можно свести поиск гамильтонова пути (создадим граф со всеми рёбрами веса -1, минимальный простой путь — гамильтонов путь).
Поэтому давайте считать, что циклов отрицательного веса нет. Что это значит? Это, на самом деле, значит, что существует простой кратчайший путь. его и будем искать. А искать мы будем алгоритмом Форда — Беллмана. Это, по сути, динамическое программирование. Пусть `d[v][k]` — кратчайший путь из s в v из k рёбер.
База динамики: `d[s][0] = 0`, `d[v][0] = infinity`.
Переход: `d[v][k] = min([d[u][k - 1] + weight[u][v] for u in reverse_edges[v]])`.
Ответ: `min(d[t])`.
Асимптотика этого? Мы делаем n итераций, и на каждой смотрим на все рёбра: $O(nm)$. Что тут можно оптимизировать? Во-первых, память. Зачем хранить квадратный массив, если нам в каждый момент нужна только предыдущая строка. Разве что в конце нам надо взять минимум по всем путям из s в t. Это можно пофиксить тупым способом: просто завести переменную и на каждой итерации перехода обновлять её, — а можно чуть более сложным: изменить нашу динамику до того, чтобы хранить кратчайший путь из s в v из $\leqslant k$ рёбер. Это, кстати, позволит нам проверять, изменился у нас массив или нет, потому что если нет, можно прекратить алгоритм. Но на самом деле иметь две строки нашей динамики — многовато, можно одну:

```
d = [infinity for v in range(n)]
d[s] = 0
for k in range(n - 2):
    for u, v in edges:
        d[v] = min(d[v], d[u] + weight[u][v])
```

Почему это работает? Потому что на k-той итерации `d[v]` — длина какого-то пути $s \to v$, который не хуже, чем путь из $\leqslant k$ рёбер.

Детекция отрицательных циклов. Очевидно, отрицательный цикл в графе есть тогда и только тогда, когда `d[*][n] != d[*][n - 1]`. Почему? Ну, если он есть, мы не можем остановить релаксацию. Если релаксация произошла на итерации после последней, значит путь из n рёбер короче, чем путь из $n-1$, а в пути из n вершин есть цикл. Вероятно, отрицательный, раз уж путь меньше всего уже рассмотренного.
Чтобы найти сам цикл, а не узнать его наличие, придётся хранить, откуда произошла релаксация, и при последней релаксации взять вершину и пойти по обратным ссылкам.

Кратчайшие пути между любой парой вершин. Пусть у нас есть `d[i][j]` и номер итерации k. Тогда `d[i][j]` — кратчайший путь $i \rightsquigarrow j$ среди всех путей, все внутренние вершины которых имеют номера не больше k. Тогда что при $k = 0$? Ну,

$$d[i][j] = \begin{cases} w_{ij} & ij \in E \\ 0 & i = j \\ +\infty & \text{otherwise} \end{cases}$$

Тогда что у нас с переходом? Ну, мы можем либо идти через k, либо нет. Если мы идём — мы сначала идём $i \rightsquigarrow k$, а потом — $k \rightsquigarrow j$. Оба этих пути у нас уже посчитаны.

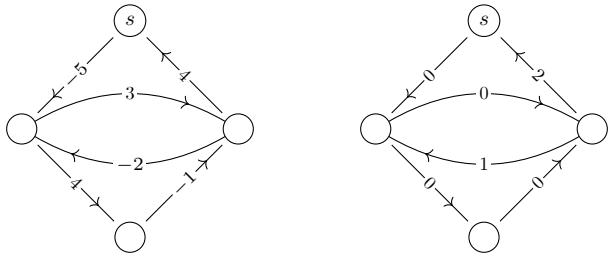
```
for k in range(n):
    for i in range(n):
        for j in range(n):
            d[i][j] = min(d[i][j], d[i][k] + d[k][j])
```

Детекция отрицательных циклов. Работает точно также, как в алгоритме Форда — Беллмана, разве что тут отрицательные расстояния очень быстро растут по модулю, а значит очень скоро переполняются, так что за ними надо следить.

Кратчайшие пути между любой парой вершин для разреженных графов. Пусть у нас есть потенциалы $\varphi(v)$ и мы перевзвешиваем граф так:

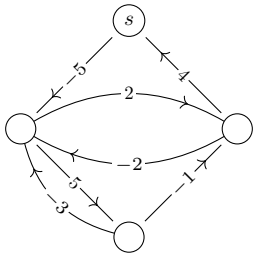
$$w'_{uv} = w_{uv} + \varphi(u) - \varphi(v)$$

Мы уже обсуждали, что после такой операции кратчайшие пути не изменятся. А значит, если мы подберём такие потенциалы, что все веса станут положительными, то можно будет просто запустить Дейкстру! И, если у нас нет отрицательных циклов, то такое существуют. И равны они, барабанная дробь, $\varphi(v) = \text{dist}(s; v)$.



Резонный вопрос: какой в этом смысл, если мы уже должны знать расстояния? А на самом деле смысл есть: сначала запустим Форда — Беллмана, а потом n раз Дейкстру. Тогда мы узнаем все пути за $O(nm + n^2 \log n)$. Это для разреженных графов лучше, чем $O(n^3)$. Это алгоритм Джонсона, знакомьтесь.

Кратчайшие пути от выделенной вершины во взвешеном графе с целыми весами. Пусть у нас веса всех рёбер целые из диапазона $[-N : N]$. Тогда существует алгоритм, который ищет все пути от данной вершины за $O(\sqrt{n}m \log N)$. На самом деле $\log N$ — это не то, что мы хотим вставлять в асимптотику, потому что это в каком-то смысле размер ввода. Если бы у нас было N в асимптотике, всё было бы совсем грустно, это даже не полином от размера ввода получился бы. А $\log N$ — как раз полином, посколько это количество битов или символов во всех числах.



Давайте делать вот какой алгоритм. Начнём с некоторого графа, у которого все $w_{uv} \in \{0; 1\}$, а потом будем делать с ним следующие операции:

1. Все w_{uv} удвоить.
2. Некоторые w_{uv} уменьшить на 1

Как мы тогда получим 5, например? Ну, вот так:

$$1 \to 2 \to 4 \to 3 \to 6 \to 5$$

И вообще операций у нас будет $\log N$ каждой операции, что легко заметить из двоичного разложения чисел в N бит. Мы по сути умножаем на 2 $\log N$ раз и иногда после этого вычитаем единицу. А это значит, что мы со всеми рёбрами можем такое делать за $\log N$.
А теперь давайте с нашими операциями вот что делать: поддерживать потенциалы $\varphi(v)$, такие что веса всех рёбер с их учётом неотрицательны. Какие у нас в начале потенциалы? Ну, $\varphi(v) = 0$ хватит. Что мы делаем в операции умножения на 2? Ну, удвоим потенциалы. А вот что делать при уменьшении

весов на 1, уже интересно.
У нас какие-то w_{uv} стали равны -1. Мы хотим пересчитать потенциалы так, чтобы w'_{uv} были неотрицательны. Рассмотрим граф G', состоящий из рёбер $w'_{uv} \leqslant 0$. Что если в нём есть цикл? Это значит, что у нас есть нулевой или отрицательный цикл. Заметим, что вес цикла от потенциалов не меняется, а значит если у нас есть отрицательный цикл в w'_{uv}, то и в w_{uv} он тоже есть. А от него мы не избавимся нашими двумя операциями, то есть в исходном графе был цикл отрицательного веса. Как проверить его наличие? Возьмём G', построим в нём компоненты сильной связности. Если ребро с $w'_{uv} = 1$ соединяет вершины одной компоненты, получается цикл отрицательного веса. В противном случае, раз мы уже сделали ККС, решим задачу на конденсации. У нас рёбра веса -1 соединяют компоненты. Теперь осталось у компонент сильной связности G' поменять потенциалы так, чтобы всё было хорошо. Что мы можем сделать? Мы можем взять какой-то набор компонент, из которого ничего не выходит, и уменьшить ему потенциал на 1. Тогда у нас все отрицательные рёбра, входившие в это множество, станут хорошими, а плохих рёбер не появится. Проблема в том, как выбрать это множество. Ну, мы можем взять вершину, в которую всходит отрицательное ребро, посмотреть, кто из неё достижим, и уменьшить потенциал везде там. Правда, после этого придётся перестраивать G', снова делать ККС, чтобы искать новые отрицательные рёбра. И работает это за $O(mn)$, потому что наши операции конденсации и уменьшения потенциала работают за $O(m)$, а делаем мы это максимум для n вершин. Поэтому давайте убивать такие вершины массово. Ну, если у нас есть несколько вершин, в которые входят отрицательные рёбра, и эти самые вершины не связаны путём (никакая пара), то возьмём всех их. Но такого набор вершин может не быть. Если для любой пары вершин они соединены путём в одну из сторон, то все наши отрицательные рёбра лежат на цепочке. И тут начинается самое интересное. Давайте возьмём конец цепочки, и всем, что из неё достижимо, уменьшим потенциал. У нас добавятся какие-то нулевые рёбра, добавим их в G'. Потом возьмём следующее отрицательное ребро, сделаем то же самое. На самом деле это можно (но совершенно не просто) делать за линейное от длины цепочки время.
А теперь финальный штрих. Утверждается, что в графе есть либо $\sqrt{k}$ не связанных вершин, в которые входят отрицательные рёбра, либо цепочка длины $\sqrt{k}$, где k — количество вершин, в которые входят отрицательные рёбра. Отсюда имеем

$$T(k) = T(k - \sqrt{k}) + 1$$

И это на самом деле $T(k) = O(\sqrt{k})$.

6 Игры на графах.

Самая простая игра на графах выглядит так: пусть есть ориентированный граф, и игроки по очереди двигают фишку в направлении одного из рёбер. Проигрывает тот, кто не может сделать ход.
Как это делать в простом случае: для ациклического графа? Ну, сделать топологическую сортировку и посчитать на ней динамику. Вообще игры на ациклических графах довольно просты обычно. Можно, например, назначит каждому ребру цену, которую надо заплатить своему противнику для прохода по этому ребру, и когда ходить некуда, игра кончается. Побеждает самый богатый. Ну так тут тоже можно сделать динамику на топологической сортировке.
Есть, правда, игры с проблемами. Если два игрока пытаются оптимизировать разные параметры, то очень сложно с этим что-то делать. Например, при проходе по ребру игроку дают определённую сумму денег и каждый максимизирует свою.
Ещё есть игра в поддавки: то же самое, но тот, кто пришёл в ранее проигрышную позицию, теперь выигрывает. И тут, на самом деле, мы делаем абсолютно то же самое, что и оригинальной игре, но проигрышные состояния делаем выигрышными. Если мы играли в то, что тот, кто не может сделать ход, проигрывает, можно даже буквально свести поддавки к оригиналу, добавив новую тупиковую вершину и добавить рёбра в неё из каждого прошлого тупика.
А ещё бывают ассиметричные игры. Например, в графе есть два типа рёбер и игроки ходят по разным типам. И снова тот, кто не может походить, проиграет. Тогда всё, что нам надо — завести две динамики под обоих игроков.
А вот когда у нас есть циклы... Начинается всё совсем плохое. Потому что у нас на циклах бывают

позиции с ничьей. А бывает такое, что на одном цикле бывают и выигрышные, и проигрышные, и ничейные позиции. Поэтому тут хочется что-то делать с этим. В частности, хочется что-то делать с определением победы. Мы должны победить за конечное количество шагов. Кстати, можно ходы явно ограничить количеством вершин. Если мы в честной игре прошли по всем вершинам, значит мы зациклились и пройдём дальше по тому же циклу. Поэтому давайте считать, что тот, кто выигрывает, хочет выиграть как можно быстрее, а тот, кто проигрывает, хочет поиграть подольше (как в ЕГЭ). И тут уже всё становится более приятно. Алгоритм вот такой

1. Все тупики — проигрышные.
2. Если из вершины есть ребро в проигрышную, эта выигрышная.
3. Если из вершины все рёбра в выигрышную, эта проигрышная.
4. Если мы больше не можем пометить ничего, все оставшиеся вершины — ничейные. Почему? По индукции, на самом деле.

Как это делать? Ну, BFS по обратным рёбрам.
Понятно, что игра никак не меняется от поддавков или рёбер разных цветов.

Функция Гранди. А пусть у нас есть две игры (в обеих проигрывает тот, кто не может ходить)! Тогда суммой игра называется такая игра, в которой вы можете за один ход походить в одной из этих игр. Проигрывает тот, кто не может ходить.
Тогда наша игра — это тоже игра на некотором графе. Множество вершин на ней — декартово произведение вершин графов изначальных вершин. А рёбра понятно, как строить.
Проблема в том, что считать это за время $O(nm)$ мы не хотим, это долго. Поэтому придётся думать. Давайте вот о чём подумаем. Что будет, если в обеих играх мы проигрываем? Что у нас происходит в сумме игр? Ну, мы тоже проиграем, потому что наш противник может ходить в той же игре, что и мы. Если в одной игре мы проигрываем, а в другой — выигрываем, то что? То в сумме мы выигрываем, потому что мы ходим там, где выигрываем. А когда обе игры выигрышные, всё довольно грустно и непонятно. Но вот на самом деле теория игр говорит нам, что сумма двух выигрышных игр проигрышная только если они примерно одинаковы (вообще это определение эквивалентности, но пофиг). И тут появляется товарищ Гранди:
Пусть

$$G(v) = \underset{uv \in E}{\operatorname{mex}}(G(u))$$

Где MEX множества — минимальное неотрицательное целое число, которого нет в множестве.
Для начала по индукции можно доказать, что $G(v) = 0$ тогда и только тогда, когда v — проигрыш. Так вот, на самом деле то, насколько $G(v)$ больше нуля, задаёт эквивалентность выигрышных игр. Так вот давайте докажем, что

$$G(A + B) = G(A) \oplus G(B)$$

где $\oplus$ — побитовый хог.
Почему это так? Ну, у нас из позиции $(A; B)$ есть переходы в игры типа $(A'; B)$ и в игры вида $(A; B')$. Пусть $x = G(A) \oplus G(B)$; тогда докажем следующее:

$$\forall y < x \; \exists G(A' + B) = y \vee \exists G(A + B') = y$$

$$\nexists G(A' + B) = x \wedge \nexists G(A + B') = x$$

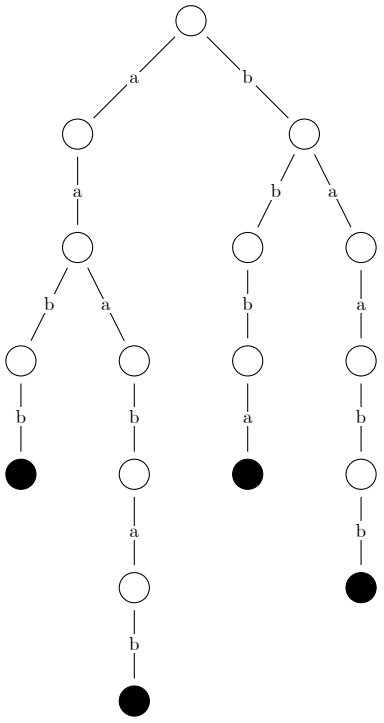
Сначала докажем второе от противного. Пусть, не умаляя общности, $\exists G(A' + B) = x$. По индукции мы уже доказали, что $G(A' + B) = G(A') \oplus G(B)$. То есть $G(A') \oplus G(B) = G(A) \oplus G(B)$. Отсюда $G(A') = G(A)$, что не может быть верным.
Первое будет посложнее. Рассмотрим $G(A)$ и $G(B)$. Они имеют какие-то битовые записи. И есть x и y с тоже какими-то битовыми записями. Что значит, что $y < x$? Это значит, что у них есть общий префикс, а потом в x идёт единица, а в y — ноль. Эта самая единица значит, что (не умаляя общности) в $G(A)$ на этом месте единица, а в $G(B)$ — ноль. Посмотрим теперь на $y \oplus G(B)$. Получим число, у которого на

соответствующем месте ноль, а до него — $G(A)$. Тогда $y \oplus G(B) < G(A)$, а значит $\exists A'\ G(A') = y \oplus G(B)$ (по определению функции Гранди).
Ну так и всё, теперь, если нам надо посчитать, кто побеждает в сумме (возможно, нескольких) игр, считаем функцию Гранди каждой и берём xor всех. Ну, это хорошо, а играть как? Да элементарно. $a \operatorname{xor} b = 0 \Leftrightarrow a = b$. А отсюда, если мы видим, что в одной игре функция Гранди равна a, а в другой — $b < a$, пойдём $a \mapsto b$, получим сумму игр с нулевой функцией Гранди. То есть надо идти туда, где xor равен нулю.

Массовый поиск подстрок. Пусть у нас есть сразу несколько строк. И мы хотим в одном тексте все их поискать.

1. $s_1 = \text{aabb}$.
2. $s_2 = \text{aaabab}$.
3. $s_3 = \text{baabb}$.
4. $s_4 = \text{bbba}$.

Вывести все вхождения — это долго, их много, мы это в любом случае это число выводить будем за $O(n^2)$. Поэтому хотеть вывести все вхождения нельзя. Можно хотеть вывести любое вхождение любой строки или посчитать что-то и вывести количество. Например, рассмотрим задачу про цензуру. Стример говорит что-то на twitch'е, надо проверить, есть ли в его речи ниггеры или пидоры.
За сколько мы умеем решать, если у нас n запрещённых строк с суммарной длиной L, а говорит текст имеет длину m. За сколько мы уже имеем решать эту задачу? Ну, за $O(L + mn)$. L — это хорошо, а вот nm — не очень. Давайте пойдём по тексту и будем все строки сразу искать.
Чтобы аккуратно хранить множество строк, есть такая штука как «бор» («trie»). Это дерево букв, в котором каждый путь до терминальной вершины соотвествует строке.



Чёрным помечены терминальные состояния (т.е. те строки, что у нас есть в множестве). В эту штуку можно быстро добавить новую строку за её длину. Можно за долину строки проверить, есть ли она в множестве. Можно использовать бор в качестве множества, но лучше для этой цели использовать хэш-таблицу. Бор используется для других целей.
Какие вершины есть в этом дереве? Ну, любая вершина — префикс некоторой строки множества (и наоборот, любой префикс есть в множестве).
План такой: идёт по тексту слева направо и будем искать наши строки. Как будем искать? Давайте будем идти по тексту слева направо. Нам не нужно помнить все буквы текста, у нас самая длинная строка имеет длину 6, можно последние 6 букв помнить. Но это не всё, что можно отбросить. Пусть у нас текст такой:

abbaababbaba

А дальше мы пока недочитали. Интересует ли нас последняя буква a? Да, с неё может начинаться. А предпоследняя? Тоже, ведь есть строка, начинающаяся с ab. А дальше нет, ни один другой суффикс уже прочитанного текста не является префиксом строки. Давайте вот хранить самый длинный префикс и, тривиально, при переходе к следующему элементу нам не понадобятся более длинные префиксы. Кстати, что с переходом? Если нам приходит новый символ, который подходит под префикс, то и оставим как есть. Если не подходит, новый самый длинный префикс надо найти. Как? Ну, это похоже на КМП. Код ещё напишем, не беспокойтесь.
Что происходит? Давайте хранить вершину бора, которая соответствует наидлиннейшему префиксу. Смотрим на следующий символ. Если из вершины есть путь по этому символу, то хорошо. А если нет, надо что-то делать. Для решения подобных проблем придумали *суффиксные ссылки*. Пусть α — префикс какой-то s_i. $s[\alpha]$ — самый длинный суффикс α, также являющийся суффиксом некоторой s_j. Правда ли, что это то, что мы хотим? Ну, да, но нет. Мы хотели не самый длинный суффикс, мы хотели такой, к которому можно приделать символ. Ну, возьмём $s[\alpha]$, если не подходит, возьмём $s[s[\alpha]]$. И так далее. Что будет, когда мы дойдём до корня? За'`if`'аем.

```
cur = root
for x in T:
    while cur != root and go[cur][x] == None:
        cur = suf_link[cur]
    if go[cur][x] != None:
        cur = go[cur][x]
```

Две детали: пока непонятно, как считать суффиксные ссылки. Во-вторых, самый длинный префикс — это хорошо, но если у нас есть строка aabbababab и ab, то есть текст начинается на ab, мы не увидим, что придём в терминальное состояние.
Начнём со второго. Давайте в начале пометим как терминальные не только истинно терминальные, но ещё и такие, у которых суффикс терминальный. Чтобы посмотреть на суффиксы, надо просто по суффиксным ссылкам пойти до конца.
Теперь как искать суффиксные ссылки? Пусть у нас есть строка α. И пусть у α последний символ — x. И $s[\alpha]$ — строка, которая если не пуста, то кончается на x. Хм-м. Давайте возьмём родителя α (всё, α, кроме x). У него уже посчитаны суффиксные ссылки. Давайте от родителя α переберём все суффиксы, пока не найдём того, у которого есть переход по x.
В каком порядке строить суффиксные ссылки? Да пофиг, рекурсия с запоминанием, циклов не будет т.е. суффиксные ссылки короче, чем сама вершина.
Утверждение: время работы нашего алгоритма равно $O(L+T)$. Почему мы строим суффиксные ссылки за линейное время? Рассмотрим путь от корня до какой-то строки s_i. Всё хорошо, кроме того, что мы можем несколько раз брать суффиксную ссылку. Но каждый раз, когда мы это делаем, мы уменьшаем длину строки, а поскольку увеличиваем её мы n раз, уменьшаем тоже не больше n. Как в КМП, короче.
Давайте теперь захотим для **каждой** строки понять, встречается ли она в тексте. Тогда когда мы храним суффикс текста, у нас в нём встречается s_i, когда в него можно придти по суффиксным ссылкам. Так давайте тогда просто помечать все состояния дерева, где мы были, а в конце в каждом суффиксном поддереве каждой s_i искать, были ли мы в какой-то вершине там. Дальше отсюда легко найти количество вхождений, самое левое/правое и прочее хорошее.
Lifehack: не хочется `if`'ать корень никогда. Давайте сделаем корню суффиксную ссылку в строку длины -1. К ней можно приписать всё, что угодно, получив пустую строку.
Вся эта штука называется алгоритмом Ахо — Корасика.

Суффиксные структуры.

Суффиксный массив. Отвечает на всякие запросы на подстроках. Например, у вас есть строка S и нам приходят запросы вида «входит ли P в заданную подстроку S».
Рассмотрим все суффиксы строки. Любая подстрока S является префиксом какого-то её суффикса. Что такое суффиксный массив? Рассмотрим строку

$$S = \text{abaabb}$$

Выпишем все её суффиксы в отсортированном (лексикографически) порядке.

ε
aabb
abaabb
abb
b
baabb
bb

Если у нас есть такая структура, то как мы будем искать паттерн в строке? Ну, двоичным поиском найдём, какой суффикс нам нужен, и всё.
Как это хранить? Мы же не хотим тратить память на строки, давайте задавать суффикс индексом первого его элемента. Т.е. в случае выше получим

$$[6, 2, 0, 3, 5, 1, 4]$$

Задача на сегодня: построить этот массив (обозначим его p) быстро.
Способ номер один: втупую взять и посортировать. Это будет в худшем случае работать за $n^2 \log n$. Но если строка рандомная, то компаратор работает в среднем за константу. Поэтому на практике так можно, и это в целом работает. Что в этом хорошего? А того, что для quick sort'а тратит мало дополнительной памяти. Поэтому на практике так даже лучше.
Но мы с вами теоретики, и хотим сделать всё за $n \log n$. Можно вообще сделать это за линейное время (если алфавит маленький), но не сейчас. Пункт номер 1: сортировать строки неприятно, потому что они разной длины. Поэтому добавим в нашу строку в конец символ, который меньше всех букв. Обозначим его как \$. И вместо суффиксов возьмём и сделаем циклические сдвиги полученной строки:

\$abaabb
aabb\$ab
abaabb\$
abb\$aba
b\$abaab
baabb\$a
bb\$abaa

Дальше заполним строку по циклу собой же, пока её длина не станет степенью двойки

\$abaabb\$
aabb\$aba
abaabb\$a
abb\$abaa
b\$abaabb
baabb\$ab
bb\$abaab

Теперь хотим делать вот какую вещь: на итерации номер k p состоит из отсортированных подстрок $S[i; i + 2^k)$ (если S циклическая).
Начнём с итерации 0. Нам нужно сортировать строки длины 1. Это делается либо быстрой сортировкой, либо сортировкой подсчётом за размер алфавита. Нам не важно, у нас и так цель — $O(n \log n)$.
Как мы переходим $k \mapsto k + 1$? Пусть у нас есть массив строк длины 2^k. Мы хотим получить массив строк длины 2^{k+1}. Как это будем делать? Возьмём строки длины 2^{k+1}. Распилим её пополам. Получим

	p	c
\$a	6	1
aa	2	2
ab	0	3
ab	3	3
b\$	5	4
ba	1	5
bb	4	6

две строки, каждая из которых есть в каком-то месте массива p. Поэтому как сравнить пару из строк длины 2^k? Пусть $A = A_1A_2$, $B = B_1B_2$. Тогда

$$A < B \Leftrightarrow A_1 < B_1 \text{ or} (A_1 = B_1 \text{ and } A_2 < B_2)$$

Как сравнивать строки длины 2^k за $O(1)$?
Решим более абстрактную задачу: у нас есть массив отсортированных объектов, и мы хотим уметь сравнивать объекты в нём. Ну, всё очень просто, давайте инвертируем массив: построим другой массив, где по объекту можно получить его номер. Тогда, если индекс A меньше индекса B, то $A \leqslant B$. Но нам надо $A < B$. Что делать? Давайте пойдём по массиву и гомоморфно сопоставим объектам числа. То есть если объекты равны, то их числа равны.
То есть теперь у нас на каждой итерации есть не только массив p, но и массив классов эквивалентности (c). Что мы тогда делаем? Собираем из строк пары. Для каждой половины строки мы можем понять её класс эквивалентности, и всё хорошо. А как пересчитать классы? Ну, если у нас есть две строки, у которых у обеих половин попарно совпадают классы, то это один класс. Если разные, то классы разные.
Пример. Что мы с этим делаем? Мы хотим упорядочить строки, которые представляются парами индексов i и $i + 2^k$. Это логарифм итераций, каждый раз сортировка. Но давайте считать, что мы сортируем не абы что, а пары маленьких целых чисел. И их можно сортировать цифровой сортировкой. Получится как раз $n \log n$.
На строках очень много каких полезных вещей сводятся к нахождению наибольшего общего префикса. Давайте научимся искать наибольший общий префикс двух суффиксов. Для этого давайте предподсчитаем длину наибольшего общего префикса каждой пары соседних строк. В нашем примере:

\$
aabb\$
abaabb\$
abb\$
b\$
baabb\$
bb\$

...это будет

$$l = [0, 1, 2, 0, 1, 1]$$

Поскольку суффиксы отсортированы, если у двух строк есть общий суффикс, у всех строк между ними есть этот же самый общий суффикс. То есть всё, что нам надо, найти минимум на отрезке посчитанного массива, а это мы уже умеем.
Как строить это массив? Алгоритмом Касаи — Аримуры — Арикавы — Ли — Парка. Возьмём суффиксы в порядке уменьшения их длины. Для каждого втупую посчитаем LCP с его предыдущим. Пусть наш суффикс сейчас имеет номер j. И суффикс до него имеет номер i. Посмотрим на суффиксы $j + 1$ и $i + 1$. Они находятся где-то в массиве. Так вот, если у i и j совпадают k первых символов, то у $j + 1$ и $i + 1$ есть точно $k - 1$ общий символ. А значит и у всего между ними тоже есть этот префикс, а значит $k - 1$ символов в следующем сравнении можно пропустить.

Итого: сравниваем строки. Если получили, что их LCP равно $k \neq 0$, то на следующем шаге при сравнении пропускаем $k-1$ символ в сравнении (они точно равны).
Дальше следует алгоритм Каркkайнена — Сандерса, который строит суффиксный массив за линейное время, но жрёт много памяти, поэтому на практике неприменим, ну и впизду его писать. Почитайте Викиконспекты, если хотите.

Суффиксное дерево. Возьмём какую-нибудь строку и построим для неё бор. И положим в бор все суффиксы. Получим суффиксное дерево. В этом дереве, как несложно заметить, содержатся все префиксы всех суффиксов. То есть все подстроки.
Есть проблема — дерево может быть большим, размера n^2. Чтобы это исправить, если у нас есть нетерминальная вершина с ровно одним ребёнком, можно убрать её и её предка и потомка соежинить реброс с двумя буквами. Тогда у нас будет $O(n)$ вершин.
Как хранить рёбра, чтобы они не занимали кучу памяти? Ну, каждое ребро — подстрока исходной строки, можно хранить два индекса.
Как ходить по такому дереву? Да легко, у нас просто состоянием является не только вершина, но ещё и место на ребре, если там больше одной буквы.
Теперь как на этом делать суффиксные ссылки? Ну, если бы сжатия не было, то тривиально, что суффиксная ссылка вершины — она же без первой буквы. Теперь более сложное утверждение, если α не сжалась, то её суффиксная ссылка — тоже. Ну, если α не сжалась, значит либо у неё 2 ребёнка, либо она терминальная. Во втором случае она — суффикс s, а значит и её суффиксная ссылка — терминальная. Если у α есть два рёбенка, значит в строке s встречается αx и αy. Ну, а значит и с суффиксной ссылкой то же самое.
Да и вообще при прохождении по суффиксным ссылкам переходы только появляются.

7 Числовые алгоритмы.

У нас всё же бывают ситуации, когда нам надо делать что-то не с произвольными объектами со структурой, а именно с числами. Например, объекты нельзя сортировать быстрее $\Omega(n \log n)$. А для небольших чисел есть цифровая сортировка.
И второй момент: у нас есть RAM-модель. Там у нас есть адреса, и мы к ним обращаемся. Адрес у нас, объективно, число. И если мы хотим индексировать память размера n, то у нас адреса должны быть w-битными числами, где $w \geqslant \log_2 n$. В RAM-модели мы как-то считали, что мы умеем обращаться к элементу массива. А это сложение двух чисел и разименование. Сложение двух w-битных чисел за сколько работает... За $O(1)$ было у нас в RAM-модели, что странно, но в целом компьютеры так работают. А ещё у нас есть умножение, и если сумматор константной высоты построить можно, то вот умножить за $O(1)$ сложно. Поэтому умножение использовать хочется пореже. И ещё пореже хочется использовать деление, потому что оно ещё более мутное. А есть ещё и битовые операции, они уж точно наши друзья и работают быстро. Но так-то тоже не вполне очевидно, как для w-битного числа сделать сдвиг.
Мы будем считать, что все эти операции работают за $O(1)$, и что все они есть.

Дерево ван Эмде Боаса. Это чудо отлично работает за $O(\log w)$. Есть ещё Fusuion-дерево, оно работает за $O(\log_w n)$. Поэтому мы ещё захотим их склеить, чтобы if'ать используемую структуру в зависимости от w и n. Полученная структура будет работать за $O(\sqrt{\log n})$, и что хорошо, оно не зависит от w. Но это всё потом.
Итак, к дереву. Сделать мы хотим `insert`, `erase` и `lower_bound`. Это такое дерево, которое очень сильно ветвится. Давайте сделаем массив на 2^w элементов. В каждом будет 0 или 1. Потом решим проблему ненужности такого здорового массива, а сейчас попилим это массив на куски размера $2^{w/2}$. В каждом куске делаем то же самое (то есть каждый такой кусок пилим на $2^{w/4}$ и так далее) и останавличаемся, когда куски станут размера 2. Как делать `insert` и `erase`, понятно, а как делать `lower_bound`? Ну, рекурсивно. Берём тот блок, где должен быть наш элемент, запускаемся рекурсивно от него. Если он нашёл нам `lower_bound`, мы победили. Но там же может не быть ни одного элемента.

Придётся идти в следующий блок, да ещё и несколько раз, что очень долго, не хочется так.
Поэтому есть решение. Смотрите: у нас есть блоки размера $2^{w/2}$. Давайте создадим ещё одно дерево ван Эмде Боаса, в котором мы будем хранить, пустые у нас блоки или нет. Несложно заметить, что если мы найдём `lower_bound` в нём, то мы как раз найдём правильный блок.

```
template <size_t w>
struct vEB
{
    std::array<vEB<1 << (w / 2)>, 1 << (w / 2)> children;
    vEB<1 << (w / 2)> blocks;
};

std::optional<size_t> lower_bound(size_t x)
{
    size_t b = x >> (w / 2);
    size_t x1 = x & ((1 << (w / 2)) - 1);
    std::optional<size_t> ans1 = children[b].lower_bound(x1);
    if (ans1)
        return (b << (w / 2)) + *ans1;
    else
    {
        std::optional<size_t> b1 = blocks.lower_bound(b + 1);
        if (!b1)
            return std::nullopt;
        else
            return (b1 << (w / 2)) + *(children[b1].lower_bound(0));
    }
}

void insert(size_t x)
{
    size_t b = x >> (w / 2);
    size_t x = x & ((1 << (w / 2)) - 1)
    children[b].insert(x1);
    blocks.insert(b);
}
```

Добились ли мы этим асимптотики $O(\log w)$? Нет, конечно. Потому что у нас рекурсия теперь образует дерево. особенно в случае с `insert`. Но давайте вот на что посмотрим: если наш блок и так непустой, добавлять его в `blocks` незачем, он и так там. А если он пустой, нам, наверное, что-то попроще можно делать.
А что с `lower_bound`'ом? А давайте хранить в каждом блоке минимум и максимумом. Тогда если `x` меньше максимума, мы сделаем только первый вызов, иначе первый можно не делать. А последний вызов можно не делать вообще никогда, надо просто минимум вернуть.
Итак, что нам надо? Из простого — надо сохранить минимум и максимум. Из сложного — надо научиться быстро добавлять в пустое множество.
План такой: в текущей структуре храним минимум и максимум, а на следующем уровне храним не всё, что должны, а всё, кроме минимума и максимума. Это решение помогает нам жить с добавлением в пустое множество (у нас меняется только минимум и максимум, но не массивы).

```
template <size_t w>
struct vEB
{
    std::optional<size_t> min, max;
    std::array<vEB<1 << (w / 2)>, 1 << (w / 2)> children;
```

```
    vEB<1 << (w / 2)> blocks;
};

void add(size_t x)
{
    if (min == std::nullopt)
    {
        min = max = x;
        return;
    }
    if (x > *max)
        std::swap(*max, x);
    if (x < *min)
        std::swap(*min, x);
    size_t b = x >> (w / 2);
    size_t x1 = x & ((1 << (w / 2)) - 1);
    if (!children[b].min)
        blocks.add(b);
    children[b].add(x1);
}

std::optional<size_t> lower_bound(size_t x)
{
    if (max == std::nullopt || x > *max)
        return std::nullopt;
    if (x <= *min)
        return min;
    size_t b = x >> (w / 2);
    size_t x1 = x & ((1 << (w / 2)) - 1);
    std::optional<size_t> ans1 = children[b].lower_bound(x1);
    if (ans1)
        return (b << (w / 2)) + *ans1;
    std::optional<size_t> b1 = blocks.lower_bound(b1);
    if (!b1)
        return max;
    else
        return (b1 << (w / 1)) + children[*b1].min;
}
```

Как делать удаление? Ну, теперь совершенно понятно

Немного битовой магии. За счёт чего вообще Fusion-дерево может работать тем быстрее, чем больше w? За счёт того, что операции с чистами работают за $O(1)$. А значит мы можем себе позволить за одну операцию сделать несколько операций с числами поменьше. Как, например, сравнить два маленьких числа? Ну, очень просто: перед ними приписать бит (перед первым единицу, перед вторым — ноль) и вычесть. После чего посмотреть на первый бит. Так можно сделать для двух массивов чисел, если у нас есть возможность раздвинуть эти массивы, запихнув по биту между элементами.
А если у нас есть длинное число (из нескольких маленьких) и узнать количество чисел меньше x? Тут сложнее. Ну, во-первых, считаем, что мы можем раздвинуть этот массив. Во-вторых, надо размножить x. Делается это умножением его на константу типа `0b1000010000010000100001` (тут считается, что в маленьких числах 5 знаков). Дальше из результата вычитания надо взять только правильные биты (это битовая маска), а потом узнать их количество. Как узнавать количество? Есть вот такой план. Если мы возьмём сам наш результат, у него в старшем бите нужный бит. Если сдвинем результат на размер блока B, там на том же месте будет следующий интересный бит. Если на $2B$ — третий. Ну так

возьмём сумму $1 + 2^B + 2^{2B} + \cdots$ и умножим наш результат на него. В его середине будет то, что мы хотим там увидеть.