Федеральное государственное автономное учреждение
высшего профессионального образования

Московский Физико-Технический Институт
КЛУБ ТЕХА ЛЕКЦИЙ

Алгоритмы.
ИВТ.

III СЕМЕСТР

Лекторы: *А. И. Гришутин. И. Д. Степанов.*

Автор: *А. Ш. Ильдаров.*
Проект на github

Осень 2020 года

Содержание

1 Паросочетания. Минимальное вершинное покрытие.

Определение 1. Двудольный граф – граф, множество вершин которого можно разбить на две части таким образом, что каждое ребро графа соединяет какую-то вершину из одной части с какой-то вершиной другой части, то есть не существует рёбер между вершинами одной и той же части.

Определение 2. Хроматическое число – минимальное число цветов, в которые можно раскрасить вершины графа так, чтобы концы любого ребра имели разные цвета.

Определение 3. Паросочетание (англ. matching) в двудольном графе — произвольное множество рёбер двудольного графа, такое что никакие два ребра не имеют общей вершины.
Мощность паросочетания – количество ребер в нем.
Максимальное паросочетание – мощность которого *наибольшая* среди всех возможных паросочетаний в данном графе.

Определение 4. Цепь – некоторый простой путь (т.е. не содержащий повторяющихся вершин или рёбер).
Чередующаяся цепь – цепь, в которой рёбра поочередно принадлежат/не принадлежат паросочетанию.
Увеличивающая цепь – чередующаяся цепь, в которой начальная и конечная вершины *НЕ* принадлежат паросочетанию.
Уменьшающая цепь – цепь, в которой начальное и конечное ребра принадлежат паросочетанию.

Определение 5. Вершины двудольного графа, инцидентные рёбрам паросочетания M, называются **насыщенными** или покрытыми.

Алгоритм Куна.

Теорема 1.1. *(Бержа) Паросочетание M в двудольном графе G – max $\iff$ в G нет увеличивающей цепи относительно M.*

Доказательство.
$\Rightarrow$:
От противного: Пусть в G с максимальным паросочетанием M существует увеличивающая цепь.

Тогда заменив в ней все рёбра, входящие в паросочетание, на невходящие и наоборот, мы получим большее паросочетание.
То есть M не являлось максимальным. Противоречие.
$\Leftarrow$:
Пусть M – не max, покажем что $\exists$ увеличивающая цепь.
Пусть M' – паросочетание: $|M'| > |M|$
Построим подграф $G' = M \bigoplus M'$, состоящий из ребер $\in$ только одному из паросочетаний.
M и M' – паросочетания $\Rightarrow$ нет вершин, которые смежны с двумя ребрами из паросочетания. То есть у каждой вершины подграфа есть не более одного ребра из M и не более одного из M'. $\Rightarrow \forall v \in G' \hookrightarrow deg(v) \leq 2$
Как известно, графы с таким свойством степеней вершин предстваляют из себя наборы цепей и циклов.
При этом длина цикла должна быть четной, ведь иначе мы будем иметь вершину, у которой два ребра, к ней смежных, принадлежат одному паросочетанию.
В циклах поровну ребер из каждого паросочетания, значит их вклад в отрыв M' от M по числу ребер – нулевой.
Значит обогнать M у M' получится только если в графе имеется цепочка нечетной длины, у которой начальное и конечное ребра лежат в M'. Вот эта вот цепочка и есть увеличивающая. □

1.1 Алгоритм Куна для поиска максимального паросочетания.

1. Фиксируем доли графа: L и R. Изначально считаем паросочетание пустым.
2. В доле L перебираем вершины в порядке увеличения номеров.
3. Если вершина насыщена, то пропускаем ее и идем дальше. В противном случае, пытаемся насытить вершину, запустив поиск увеличивающей цепи из этой вершины следующим образом:
   - 3.1. Стоя в текущей вершине v доли L, просмотрим все ребра из этой вершины.
   - 3.2. Возьмем текущее ребро (v, t_0): Если t_0 – ненасыщена, то одно ребро (v, t_0) и задает нам увеличивающую цепь, просто увеличим паросочетание с его помощью и прекратим поиск. Если же t_0 насыщена каким-то ребром (t_0, p), то пойдем вдоль этого ребра, уже в поисках увеличивающейся цепи, проходящей через ребра (v, t_0) и (t_0, p). Для этого просто перейдем в вершину p и продолжим обход из нее.

4. В конечном итоге, мы либо найдем увеличивающую цепь из вершины v и увеличим паросочетание этой цепью, тем самым насытив вершину, либо же покажем отсутствие увеличивающей цепи и невозможность насыщения вершины.

5. Возьмем следующую по порядку вершину доли L и повторим.

6. После того как мы просмотрим все вершины, пытаясь увеличить паросочетание цепью из них, мы получим максимальное паросочетание.

Разумеется, асимптотика алгоритма будет $O(n^3)$, ведь это n применений DFS.

Докажем корректность алгоритма. Она следует из теоремы Бержа и того факта, что 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'. Доказтельство. (http://gg.gg/njb4g)

Минимальное вершинное покрытие.

Определение 6. Вершинное покрытие графа G=(V,E) - такое подмножество S множества вершин графа V, что любое ребро этого графа инцидентно хотя бы одной вершине из множества S.

Определение 7. Минимальное вершинное покрытие графа – вершинное покрытие, состоящие из *наименьшего* числа вершин.

Утверждение 1.2. $|M_{max}| \leq |C_{min}|$. *Чтобы покрыть все ребра, нам нужно вершин не меньше, чем мощность наибольшего паросочетания.*

Доказательство.
Паросочетание это, как известно, набор непересекающихся рёбер. Мы хотим взять набор вершин, эти ребра покрывающий. Ясно что каждая вершина может покрыть не более одного ребра, ведь они не пересекаются по концам. Оценка снизу доказана. □

Теорема 1.3. *(Кёнига)*
В двудольном графе $|M_{max}| = |C_{min}|$.

Доказательство. Явно предъявим покрытие, размер которого равен размеру максимального паросочетания.

1. Разделим граф на доли L и R.

2. Сориентируем ребра: Из M_{max} $\longleftarrow$, остальные $\longrightarrow$

3. Обойдем граф из *ненасыщенных паросочетанием* вершин $\in$ L.

4. Получим разбиение графа на четыре множества: L^+, R^+ – вершины, лежащие в L или R соответственно, которые доступны из ненасыщенных вершин, лежащих в L. L^-, R^- – аналогично *недоступные* из ненасыщенных вершин, лежащих в L.

5. Поймем какие ребра бывают между каждыми из четырех множеств:

   5.1. Покажем, что ребер $L^+ \to R^-$ не бывает, ведь если бы такое ребро имелось, мы из достижимой вершины, лежащей в L^+ добрались бы по этому ребру до вершины, по определению R^-, недостижимой.

   5.2. Из аналогичных рассуждений понятно что не бывает ребер $R^+ \to L^-$.

   5.3. Покажем отсутствие ребер $R^- \to L^+$: От противного: Пусть (r^-, l^+) – такое ребро. Это ребро вида $\longleftarrow$, то есть принадлежащее паросочетанию.
   Вершина l^+ – насыщена, значит чтобы до нее дойти мы должны были начать обход из какой-то другой вершины l', из которой l^+ достижима (l^+ не может быть "началом"обхода, поскольку она насыщена).
   Так как l' и l^+ лежать в одной доле двудольного графа, маршрут из l' в l^+ в какой-то момент должен пойти справа налево, то есть имеется ребро вида (r', l^+), которое в силу своего направления также лежит в паросочетании. Значит из смежные ребра (r^-, l^+) и (r', l^+) лежат в одном паросочетании. Противоречие.

6. Получаем что любое ребро графа инцидентно или вершине из L^- или вершине из $R^+ \Rightarrow L^- \cup R^+$ – вершинное покрытие.

7. Покажем, что все вершины из $L^- \cup R^+$ насыщены ребрами из паросочетания:
   Это так, ведь если бы были ненасыщенные вершины в L-, то мы бы запускали из них обход, и они автоматически стали бы L+ (так как вершина достижима сама из себя, а обход мы запускаем из всех ненасыщенных левых вершин). В свою очередь, если бы в R^+ была ненасыщенная вершина, то существовала бы увеличивающая цепь, в этой вершине заканчивающаяся, что противоречит максимальности паросочетания.

8. Как было показано ранее, не существует ребер $R^+ \to L^-$, поэтому каждому ребру паросочетания инцидентна ровно одна вершина покрытия $L^- \cup R^+$. Тогда по приведенной выше оценке мы можем сделать вывод, что требуемое покрытие найдено.

□

2 Потоки

Определение 8. Сеть – ориентированный граф $G = (V, E)$, с двумя выделенными различными вершинами: s – исток и t – сток, а также с функцией $cap : E \to \mathbb{Z}_{>0}$.

Определение 9. Поток в сети – функция $f : V \times V \to \mathbb{Z}$:

1. $\forall u, v \in V \hookrightarrow f(u, v) \leq cap(u, v)$
2. $\forall v \in V \setminus \{s, t\} \hookrightarrow \sum\limits_{\substack{u \\ (u,v)\in E}} f(u, v) = \sum\limits_{\substack{w \\ (v,w)\in E}} f(v, w)$
3. $\forall u, v \in V \hookrightarrow f(u, v) = -f(v, u)$

Замечание. Свойства 2 и 3 можно объединить в следующем виде $\forall v \in V \hookrightarrow \sum\limits_{u\in V} f(v, u) = 0$

Определение 10. Остаточная сеть G_f сети G с потоком f – сеть, у которой модифицируются пропускные способности: $cap_f(u, v) = cap(u, v) - f(u, v)$, и удаляются ребра с нулевой cap_f.

Определение 11. Величина потока $|f|$– то сколько "вытекает" из s – сумма исходящих из s потоков.

Определение 12. Разрез сети G – пара подсетей (S, T): $s \in S, t \in T, S \cap T = \emptyset, S \cup T = G$
$cap(S, T) = \sum\limits_{\substack{u\in S \\ v\in T}} cap(u, v)$
$f(S, T) = \sum\limits_{\substack{u\in S \\ v\in T}} f(u, v)$

Лемма 2.1. (S, T) – *разрез* $\Rightarrow |f| = f(S, T)$

Доказательство. На семинаре. □

Лемма 2.2. (S, T) – *разрез* $\Rightarrow f(S, T) \leq cap(S, T)$

Доказательство. $\forall u, v \in V \hookrightarrow f(u, v) \leq cap(u, v) \Rightarrow$ чтд. □

Теорема 2.1. *(Форда – Фалкерсона) Следующие утверждения эквивалентны:*

1. f *– max поток в* G
2. *в* G_f *нет пути из* s *в* t
3. $|f|$ *= cap некоторого разреза.*

Доказательство.
1 ⇒ 2: От противного:
Пусть f – max, но в G_f есть путь $s \to t$.
Рассмотрим $x = min(cap_f) > 0$ на этом пути.
Значит вдоль пути можно пустить x единиц потока ⇒ в исходной сети можно было пустить на x единиц больше ⇒ f – не max.

2 ⇒ 3:
Пусть S = множество вершин, доступных из s в G_f, $t \notin S$
$T = V \setminus S$.
По определению (S, T) – разрез.
Покажем что $|f| = cap(S, T) = \sum\limits_{\substack{u \in S \\ v \in T}} cap(u, v)$.
Рассмотрим ребро (u, v): $u \in S, v \in T$
u – доступно в G_f, v – нет ⇒ оно удаляется в остаточной сети ⇒
$f(u, v) = cap(u, v) \Rightarrow \sum\limits_{\substack{u \in S \\ v \in V}} cap(u, v) = \sum\limits_{\substack{u \in S \\ v \in V}} f(u, v) \Rightarrow$
$f(S, T) = |f| = cap(S, T)$.

3 ⇒ 1:
$\exists (S, T) : \ |f| = cap(S, T)$.
При этом $\forall (S, T)$ – разрез $\hookrightarrow f(S, T) \leq cap(S, T) \Rightarrow f$ – max. □

2.1 Алгоритм Форда – Фалкерсона для поиска максимального потока.

1. Изначально поток равен 0.
2. Пока в G_f есть путь из s в t:
   2.1. $x = min(cap_f$ на этом пути $) > 0$
   2.2. Увеличим поток f на найденом пути на x и изменим остаточную сеть соответствующим образом.

$|f|$ увеливается на целое положительное число и ограничен сверху, значит алгоритм закончится. Асимтотика – $O(ans \times |E|)$

2.2 Алгоритм Эдмондса – Карпа для поиска максимального потока.

Та же самая идея, только теперь вместо произвольного пути будем выбирать кратчайший путь по ребрам. По сути, DFS меняется на BFS.

1. Изначально поток равен 0.
2. Пока в G_f есть *минимальный* путь из s в t:
   2.1. $x = min(cap_f$ на этом пути $) > 0$.
   2.2. увеличим поток f на x и изменим остаточную сеть соответствующим образом.

Алгоритм будет иметь асимптотику $O(|V| \times |E|^2)$

Докажем это.

Определение 13. $dist(u, v)$ – минимальное число ребер между вершинами.

Лемма 2.3. *Пусть поток f' получается из потока f после одной итерации алгоритма Эдмондса – Карпа.*
Пусть $dist'(u, v)$ – кратчайшее расстояние между вершинами u, v в $G_{f'}$, тогда $\forall v \in V \setminus \{s, t\} \hookrightarrow dist'(s, v) \geq dist(s, v)$.
То есть с каждой итерацией расстояние до источника не убывает ни у одной из вершин.

Доказательство. От противного:
Пусть $\exists v \in V \setminus \{s, t\} : \ dist'(s, v) < dist(s, v)$ и при этом $dist(s, v)$ – минимальное возможное среди таких v
Пусть u – предыдущая вершина перед v на кратчайшем пути из s в v в $G_{f'}$
Тогда $dist'(s, u) = dist'(s, v) - 1$ (по определению dist).
Также $dist'(s, u) \geq dist(s, u)$ ведь мы выбрали минимально удаленное v.
Рассмотрим 2 случая:
$(u, v) \in E_f$, тогда:
$dist(s, v) \leq dist(s, u) + 1 \leq dist'(s, u) + 1 \leq dist'(s, v)$
Но при этом $dist'(s, v) < dist(s, v)$
Противоречие.

$(u, v) \notin E_f$, 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, а значит вершина v – предыдущая перед u на пути из s в u
Это значит что $dist(s, v) = dist(s, u) - 1 \leq dist'(s, u) - 1 = dist'(s, v) - 2$, но ведь $dist'(s, v) < dist(s, v)$. Противоречие.

□

Определение 14. Насыщенное – ребро, вдоль которого идет поток, равный его capacity.

Лемма 2.4. *Любое ребро сети насыщается не более $O(|V|)$ раз, то есть в алгоритме $O(|V||E|)$ итераций, то есть каждая итерация насыщает хоть 1 ребро.*

Доказательство.
Рассмотрим ребро (u, v) в момент его насыщения.
Если оно насыщается, то оно лежит на кратчайшем пути от s до v (так работает наш алгоритм).
Значит $dist(s, u) + 1 = dist(s, v)$.

Теперь посмотрим, когда оно могло насытиться еще один раз:
Оно сначала должно перестать быть насыщенным, для этого нужно отменить поток вдоль (u, v), то есть пустить поток вдоль (v, u)
Пусть $dist'$ – расстояние в момент, когда пускается поток вдоль (v, u).
$dist'(s, v) + 1 = dist'(s, u)$ (Так как проталкивание происходит вдоль кратчайшего пути).
Со временем расстояния только увеличиваются, значит $dist'(s, u) = dist'(s, v) + 1 \geq dist(s, v) + 1 = dist(s, u) + 2$
Таким образом от момента насыщения, до момента повторного насыщения расстояние $dist(s, u)$ должно вырасти по меньшей мере на 2.
Однако все расстояния ограничены $O(|V|)$, значит и насыщений может быть только $O(|V|)$. □

3 Продложение потоков. Алгоритм Диница, теоремы Карзанова

3.1 Детали реализации.

```
struct Edge {
  int from = 0,
      to   = 0;

  long long cap = 0;
  long long flow = 0;
};

auto edges = vector<Edge>();
auto outbound = vector<vector<int>>();
/* v-th element -- numbers of edges from v. */
```

Листинг 1: Удобная реализация ребер

Рядом с каждым ребром всегда есть обратное, изначально с нулевой capacity. Будем присваивать обратному ребру номер исходного, увеличенный на 1. К примеру номера $2n$ и $2n + 1$. Тогда при добавлении нового ребра в граф, мы будем добавлять его и сразу же добавлять обратное, а потом сохранять номера обоих ребер в массив edges.

Теперь если мы пускаем поток по ребру с номером i, то можем удобно уменьшить поток по противоположному ребру, как имеющее номер $i \oplus 1$

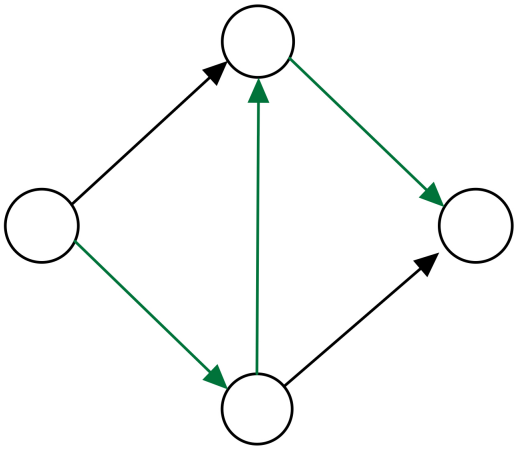
```
edges[i].flow  += delta_flow;
edges[i ^ 1].flow -= delta_flow;
```

При реализации алгоритма Форда-Фалкерсона будем просто выполнять DFS на ребрах, у которых cap $>$ flow.

3.2 Блокирующий поток.

Определение 15. Блокирующий – поток f в сети G, такой что в сети G (но вовсе не обязательно G_f) больше нет пути из s в t, вдоль которого можно протолкнуть поток.

Пример. Все ребра имеют $cap = 1$, зеленым отмечены насыщенные ребра.



Замечание. Блокирующий поток совершенно не обязательно наибольший.

3.3 Поиск блокирующего потока

За $O(nm)$

Забудем что у нас бывают обратные ребра.

Идея проста, будем пускать поток пока получается, а для оптимизации учтем тот факт, что если мы уже однажды пытались пойти вдоль ребра, и узнали что после прохода по нему дойти до t не получится, то нам не стоит идти по этому ребру снова. Это аналог метки used в DFS.

```
auto first_worthy_edge_num = vector<int>();
/*first_worthy_edge[v] -- relative number of the first edge
              which is worth considiring from v. */

long long dfs(int v, long long flow){
//If we've followed the path and we can push the flow, we do it.

  if (v == t) return flow;

//While there is at least one worthy edge.
  while (first_worthy_edge_num[v] < outbound[v].size()){
    auto cur_e = g[v][first_worthy_edge_num[v]];

  /*If there is some additional flow we can push and the edge is
           presented in initial network (is not reverse). */

  //We will code edge presence checker in Dinic's algorithm.
  if (edges[e].capacity > edges[e].flow && e is not reverse edge){
      auto next_flow = dfs(edges[e].to,
                           min(flow, edges[e].cap - edges[e].flow));


      if (next_flow > 0){
        edges[e].flow  += next_flow;
        edges[e ^ 1].flow -= next_flow;

        return next_flow;
      }
    }

    ++first_worthy_edge_num[v];
  }

  return 0;
}
```

Тогда сам поиск блокирующего потока будет иметь вид

```
while ((auto x = dfs(s, inf)) != 0){
  blocking_flow += x;
}
```

Алгоритм работает за $O(nm)$, ведь если главный вызов dfs (из main) суммарно сдвинул все номера интересных вершин на k, то dfs работал $n + k$, так как каждое ребро либо нас устроило, и мы просто увеличили номер, либо оно нас устроило, и мы пошли в новую

вершину, а сдвигать указатели долго мы не сможем, суммарно сдвигов будет не больше чем m, а значит время работы $= n*$ кол-во запусков $+m$, а каждый запуск насыщает хоть 1 ребро, при этом из-за отсутствия обратного ребра, ребра не рассыщаются. Значит имеем асимптотику $O(nm)$

3.4 Алгоритм Диница для поиска наибольшего потока

За $O(n^2m)$

Пусть G – сеть.
$dist(s, v)$ – кратчайшее расстояние между вершинами.

Определим слоистую сеть:
Вершина s.
Вершины с $dist$ 1 от s.
...
Вершины на расстоянии k от s.

При этом оставим только ребра, идущие из меньшего уровня в следующий по порядку, то есть кратчайшие пути.

Алгоритм
Вспомним что у нас бывают обратные ребра.
Пока не найден наибольший поток:

1. Построим слоистую сеть.
2. Пустим в ней произвольный блокирующий поток.

Утверждение 3.1. *Алгоритм Диница находит наибольший поток.*

Доказательство.
Теорема Форда-Фалкерсона учит нас что поток максимален $\iff$ в остаточной сети нет увеличивающего пути. Если наш алгоритм завершился, то есть не сумел пустить блокирующий поток в слоистой сети, значит в слоистой сети нет пути из s в t, а по построению слоистой сети, это значит что и в исходной сети пути из s в t. □

Утверждение 3.2. *Алгоритм Диница делает не более n итераций.*

Доказательство.
Покажем что после каждой итерации $dist'(s,t) > dist(s,t)$.
На каждом пути из s в t есть хоть одно насыщенное ребро, ведь мы пустили блокирующий поток. (По определению блокирующего потока)
После того как мы пустили блокирующий поток, у в слоистой сети исчезли некоторые ребра слева направо, а обратные ребра справа налево наоборот появились, значит расстояние в слоистой сети увеличилось, а значит увеличилось и расстояние в исходной сети. □

3.5 Теоремы Карзанова.

Зачастую асимптотика алгоритма Диница завышена, так как сеть в задаче имеет специфичный вид. Теоремы Карзанова помогают точнее оценить число итераций.

Определение 16. $C_{out}(v) = \sum\limits_{u \in V} cap(v,u)$ $C_{in}(v) = \sum\limits_{u \in V} cap(u,v)$

Замечание. Понятно что тогда через любую вершину течет не более чем $min(C_{in}(v), C_{out}(v))$

Определение 17. Общий потенциал сети $P = \sum\limits_{v \in V, v \neq s,t} min(C_{out}(v), C_{in}(v))$

Лемма 3.1. *В сети G $l = dist(s,t) \leq \frac{P}{F} + 1$, где F – максимальный поток, P – общий потенциал.*

Доказательство.
Пусть $V_i = \{v : dist(s,v) = i\}$
Получилась слоистая сеть, в которой ребра есть только из V_i в V_{i+1}
Таким образом, если разрез и пересекает какое-либо ребро, то оно идет между слоями
А сумма capacity ребер между слоями не меньше чем максимальный поток.
$cap(\cup_0^i V_k, \cup_{i+1}^l V_k) = \sum\limits_{x \in V_i, y \in V_{i+1}} cap(x,y) \geq F$
Просуммируем это неравенство по всем i:
$\sum\limits_{i=0}^{l-1} \sum\limits_{x \in V_i, y \in V_{i+1}} cap(x,y) \geq (l-1)F$
А сверху эту сумму можно оценить общим потенциалом сети, ведь сумма capacity ребер между слоями не может превышать сумму capacity ребер в одном слое, которая, в свою очередь, не превышает сумму потенциалов вершин в V_i.
$P \geq (l-1)F$ □

Лемма 3.2. *После проталкивания потока f в сети G, общий потенциал остаточной сети G_f равен общему потенциалу исходной сети.*

Доказательство.
Когда мы проталкиваем поток f вдоль ребер (u, v) и (v, w) по обратным ребрам проходит $-f$.
Таким образом уменьшение $C_{in}(v)$ за счет потока по (u, v) будет скомпенсировано увеличением $C_{in}(v)$ за счет увеличения потока по обратному ребру (w, v). □

Теорема 3.3. *Число итераций алгоритма Диница в сети G с целочисленными capacity – $O(\sqrt{P})$*

Доказательство.
Сделаем ровно $\sqrt{P}$ итераций, тогда $l \geq \sqrt{P}$, ведь, как мы уже знаем, каждая итерация увеличивает расстояние между s и t хоть на 1.
Запишем утверждение первой леммы для остаточно сети после $\sqrt{P}$ итераций:
$l \leq \frac{P}{F} + 1$, а $F = F - f$, где f – то сколько потока удалось протолкнуть за прошедшие итерации.
При это $f \geq \sqrt{P}$, ведь все capacity целочисленные, и мы не могли проталкивать меньше 1 единицы потока за итерацию.
Таким образом получаем $F \leq 2\sqrt{P}$ □

4 Строки.

4.1 Совпадение подстрок в строке.

Построим **полиномиальную** хеш-функцию, которая каждой строке в соответствие ставит число.

$$h_{p,m}(\overline{c_0 c_1 ... c_n}) = (c_0 \cdot p^{n-1} + c_1 \cdot p^{n-2} + ... + c_{n-2} \cdot p + c_{n-1}) \ \% \ m$$

При помощи схемы Горнера, такая функция считается за линейное время, ведь $h_{p,m}(\overline{c_0 c_1 ... c_n}) = (...(((c_0 \cdot p + c_1))p + c_2)p + c_3)...)p + a_{n-1}$

Так как мы имеем дело с подстроками одной большой строки, мы можем сделать предподсчет хеш-функции для всех префиксов большой строки, а дальше воспользоваться свойством хеша:

$$h(S[l, r)) = (h(S[0, r)) - h(S[0, l)) \cdot p^{r-l+1}) \ \% \ m.$$

Таким образом, после линейного предподсчета, мы можем сравнивать строки по их хешу за константное время.

4.2 Алгоритм Рабина – Карпа для поиска подстроки в строке.

Для поиска подстроки S длины m в тексте T длины n,

1. Подсчитаем $h(T[0, m))$ и $h(S[0, m))$, а также p^m.
2. В цикле по всем i от 0 до $n - m$
   2.1. Считаем $h(T[i, m + i))$ и сравниваем его с $h(S[0, m))$.
   2.2. В случае равенства, опционально, можно провести проверку наивным посимвольным сравнением или выборочным сравнением символов, для исключения малейшей вероятности коллизии.

Алгоритм работает за линейное время $O(n+m)$, однако он не устойчив перед коллизиями хеш-функции, дополнительная обработка которых увеличивает время работы.

4.3 Алгоритм Кнута – Морриса – Пратта для поиска подстроки в строке.

4.3.1 Префикс-функция

Введем для строки S префикс-функцию π

$$\pi(i) = \max_{k \in [0,i]} \{k : S[0,k) == S[i+1-k, i+1)\}$$

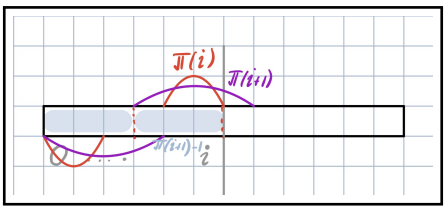
– длина наибольшего собственного суффикса подстроки S[:i], который равен префиксу такой же длины. $\pi(0) = 0$ по определению.

Пример. $S =$ "$adam$". $\pi(0) = 0$.
$\pi(1) = 0$, ведь у строки "a" есть только пустой собственный суффикс.
$\pi(2) = 0$, ведь у строки "ab" нет собственного суффикса, совпадающего с префиксом той же длины.
$\pi(3) = 1$, ведь у строки "ada" есть собственный суффикс "a" длины 1, который совпадает с префиксом "a".
$\pi(4) = 0$.

Нетрудно представить себе наивный алгоритм, подсчитывающий префикс-функцию строки за $O(n^3)$. Оптимизируем его.

Утверждение 4.1. $\pi(i+1) \leq \pi(i) + 1$

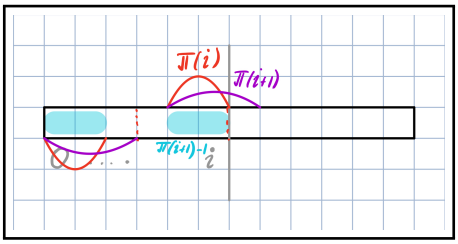
Доказательство. Рассмотрим суффикс, оканчивающийся на позиции $i+1$ и имеющий длину $\pi(i+1)$. Удалив из него последний элемент, мы получим суффикс, оканчивающийся на позиции i и имеющий длину $\pi(i+1) - 1$
Однако, мы определили $\pi(i)$ максимальную длину суфикса $\Rightarrow \pi(i) \geq \pi(i+1) - 1$.



□

Утверждение 4.2. *Если* $S[\pi(i)] == S[i+1]$, *то* $\pi(i+1) == \pi(i) + 1$.

Доказательство. Условие говорит нам, что после префикса длины $\pi(i)$ находится символ, который совпадает с символом, который находится после суффикса, оканчивающегося на позиции i. То есть наибольший суффикс, заканчивающийся на позиции $i+1$ содержит в себе все символы суффикса, заканчивающегося на позиции i и еще один символ. Это и записано в правой части.



□

Пользуясь вторым свойством, мы сможем частично оптимизировать подсчет. Если же окажется что условие равенства символов не выполняется, мы будем искать суффикс меньшей длины следующим образом: Мы знаем, что $S[1, \pi(\pi(i)))$ – суффикс подстроки $S[1, i)$

Тогда подсчет можно свести к циклу

1. $k = \pi(i)$.
   1.1. Если $S[i+1] == S[k]$, то $\pi(i+1) = k+1$.
   1.2. Если же $k == 0$, то $\pi(i+1) = 0$.
2. $k = \pi(k)$. Перейти к пункту 1.

Теперь мы будем проходиться по строке, и для каждой позиции вычислять $\pi(i)$ оптимизированно.

Утверждение 4.3. *Оптимизированный алгоритм подсчета префикс-функции имеет асимптотику* $O(n)$.

Доказательство. Из вышедоказанного утверждения мы знает что на каждой итерации по индексу строки, k увеличивается не более чем на 1. □

4.3.2 Z-функция

Введем для строки S Z-функцию z

$$z(i) = \max\{k : S[i, i+k) == S[0, k)\}$$

– длина наибольшей подстроки S, начинающейся с позиции i, которая равна префиксу S такой же длины, $z(0) = 0$ по определению.

Пример. $S =$ "$adam$". $z(0) = 0$. $z(1) = 0$. $z(2) = 1$. $z(3) = 0$.

Определение 18. Z-блок — подстрока S, начинающаяся с позиции i длины $z(i)$.

Для построения Z-функции строки S будем на каждом шаге хранить самый правый Z-блок – тот, у которого правая граница является самой правой. Назовем позиции начала и конца самого правого Z-блока l и r соответственно. Изначально $l = r = 0$.

Пусть у нас подсчитана Z-функция и Z-блок для всех позиций $< i$, тогда найдем $z(i)$.

Рассмотрим 2 случая.

1. $i \notin [l; r)$:
   Тогда нам придется считать функцию наивно, пробегая двумя указателями по строке с начала и по подстроке, начиная с позиции i. Тогда значение $z(i)$ будет определено как первая позиция j, такая что $S[j]! = S[i + j]$. После этого обновим сохраненное значение самого правого Z-блока.

2. $i \in [l, r)$:
   Заметим, что $S[l : r] == S[: (r - l)]$. Тут возможны 2 варианта:

   2.1. $z(i - l) < r - l$. То есть наибольшая подстрока, начинающаяся с позиции $i - l$, равная префиксу такой же длины, не превышает по длине самый правый Z-блок.
   Заметим, что позиция $i - l$ будет лежать в префиксе, равном по длине самый правый Z-блоку.
   Тогда $z(i) = z(i - l)$.

   2.2. $z(i - l) > r - l$. В этом случае наибольшая подстрока, начинающаяся с позиции $i - l$ выходит за пределы префикса, соответствующего самому правому Z-блоку. Тогда скажем что $z(i) = r - i$ — длина той части наибольшей подстроки, начинающейся с позиции $i - l$, которая лежит внутри префикса, соответствующего самому правому Z-блоку. **А затем будем наивно улучшать оценку, проходясь по строке начиная с позиции r.**

Утверждение 4.4. *Асимптотика вычисления Z-функции — $O(m)$, где m — длина строки.*

Доказательство. Каждая позиция в строке просматривается не более двух раз. При попадании в промежуток $[l; r)$ и наивном улучшении, или же в противном случае при непопадании в промежуток и наивном просчете. При этом, всякий раз когда мы считаем Z-функцию наивно, мы обновляем самый правый Z-блок, то есть сдвигаем его правую границу, а это может происходить не более m раз. Значит мы тратим $O(m)$ на базовые операции и $O(m)$ на суммарные наивные проверки. □

4.3.3 Алгоритм

Для поиска подстроки S длины m в тексте T:

1. Построим строки вида $S\#T$, где $\#$ — произвольный разделитель, не встречающийся в тексте.
2. По сконструированной строке построим префикс-функцию или Z-функцию.
3. Тогда если в некоторой позиции i $\pi(i) == m$, то эта позиции — конец вхождения подстроки.
   Аналогично, если $z(i) == m$, то эта позиция — начало вхождения подстроки.

Утверждение 4.5. *Алгоритм работает за* $(|S| + |T|)$.

Доказательство. Ведь Префикс-функция и Z-функция строятся за линейное время, а затем совершается только один проход всей результирующей строки. □

5 Продолжение строк. Алгоритм Ахо – Корасик для нахождения всех вхождений набора подстрок.

Мы хотим научиться искать все вхождения строк из набора в некотором тексте.

5.1 Построение бора.

Определение 19. Бор — дерево, в котором каждая вершина обозначает строку, а каждое ребро обозначает букву. Строка, соответствующая вершине (то есть заканчивающаяся в этой вершине), получается конкатенацией всех букв, соответствующих ребрам пути из корня в эту самую вершину. По определению, корню бора соответствует пустая строка.

Переход в боре можно хранить несколькими способами:

1. Массив указателей размером с алфавит — Время $O(1)$, память $O(|V|\Sigma)$.
2. Set — время $O(log(\Sigma))$, память $O(|V|)$.
3. unordered_set — время $O(1)$, память $O(|V|)$.

Определение 20. Терминальная вершина бора для набора слов — вершина, которой соответствует слово из этого набора.

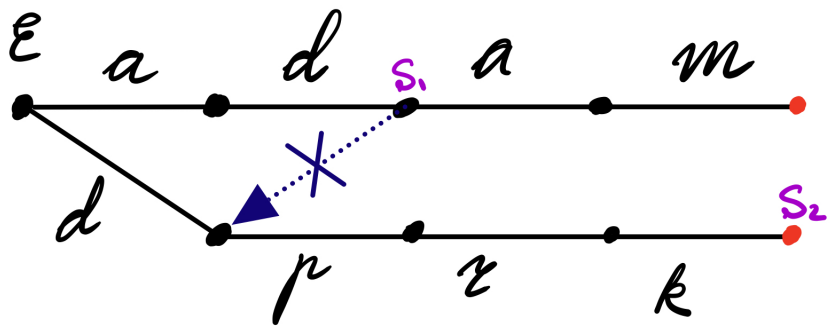
Для добавления строки в бор мы:
Прочитав очередной символ в цикле по строке, переходим по соответствующему ему ребру или создаем такое ребро если потребуется.
После завершения строки помечаем последнюю вершину как терминальную.

Таким образом, бор строится за линейное по сумме всех строк в наборе время.

5.2 Преобразование бора в автомат.

Будем понимать вершины бора и соответствующие им строки как состояния конечного детерминированного автомата. Однако мы сталкиваемся с проблемой, ребер бора не достаточно для отражения всех возможных переходов между состояниями автомата.

Пример. Ребра бора не отражают тот факт, что перейдя под воздействием символов "ad" в некоторое состояние S_1 мы все еще можем перейти в состояние S_2.



Определение 21. Суффиксная ссылка вершины v — ссылка (мнимая стрелка в боре) на вершину u, такую, что состояние u — наибольший собственный суффикс состояния v, а если такой вершины u нет, то ссылка на корень. По определению, ссылка из корня ведет в корень.

Пример. Для бора строк "$adam$" и "$dprk$" единственной суффиксной ссылкой, не ведущей в корень, будет ссылка из состояния "ad", в состояние "d", зачеркнутая на рисунке выше.

5.2.1 Нахождение суффиксных ссылок.

Чтобы найти суффиксную ссылку для вершины v:
Если v – корень, то и его суффиксная ссылка тоже корень. В противном случае:

- Пусть c — буква, преводящая из родителя в вершину v.
- Рассмотрим в качестве текущей вершины суффиксную ссылку родителя.
- Пока в рассматриваемой вершине нет ребра, соответствующего букве c, будем прыгать дальше, рассматривая ее суффиксную ссылку.
- Если мы уперлись в корень, то корень и является суффиксной ссылкой вершины v.
- Если мы нашли вершину u, с ребром c, то суффиксной ссылкой будет вершина, в которую ребро c переводит вершину u.

Концептуально, мы берем наибольший собственный суффикс родителя и откусываем от него по одной букве слева, пока не получится подстрока, которая добавлением одного символа становится собственным суффиксом нашей исходной строки.

```
1    auto get_suflink(Node_t* node){
2
```

```
        if (node == root) return root;

        auto last_c = node->c;
        auto parent = node->parent;

        while(parent != root && parent->go[last_c] == nullptr){
            parent = get_suflink(parent);
        }

        if (parent == root) return root;
        return parent->go[last_c];
    }
```

Этот алгоритм терминируется, ведь на каждом шаге мы поднимаемся выше по бору, а значит рано или поздно дойдем до корня.

Мы считаем суффиксные ссылки рекурсивно, но этот процесс можно **оптимизировать, записав суффиксную ссылку в вершину**, ведь после построения автомата она меняться не будет. Тогда мы можем ввести универсальную автоматную функцию перехода, которая не делает различия между суффиксными ссылками.

$jump(node, c) =$

son_c, если ребро бора c переводит вершину $node$ в son_c

$null$, если $node$ — корень, у которого нет ребра бора c

$jump(suflink(node), c)$, иначе

С учетом концепции универсальности перехода, переход по суффиксной ссылке вершины — переход из суффиксной ссылки родителя по ребру c, где c — ребро, которое перевело родителя в вершину.

```
    suflink(node) = jump(suflink(node->parent), node->c);
```

Тогда чтобы посчитать суффиксные ссылки мы пойдем по дереву по слоям, высчитывая suflink и jump одновременно:

1. Для корня все известно.
2. Для первого слоя suflink ведет в корень, а jump вниз или по suflink в корень и по go от него.
3. Для каждого из следующих уровней мы можем найти и suflink и jump за $O(1)$, ведь для родителя, который находится уровнем выше мы уже все посчитали.

Концептуально это BFS.

5.3 Поиск вхождений при помощи автомата.

Для поиска набора строк в тексте:

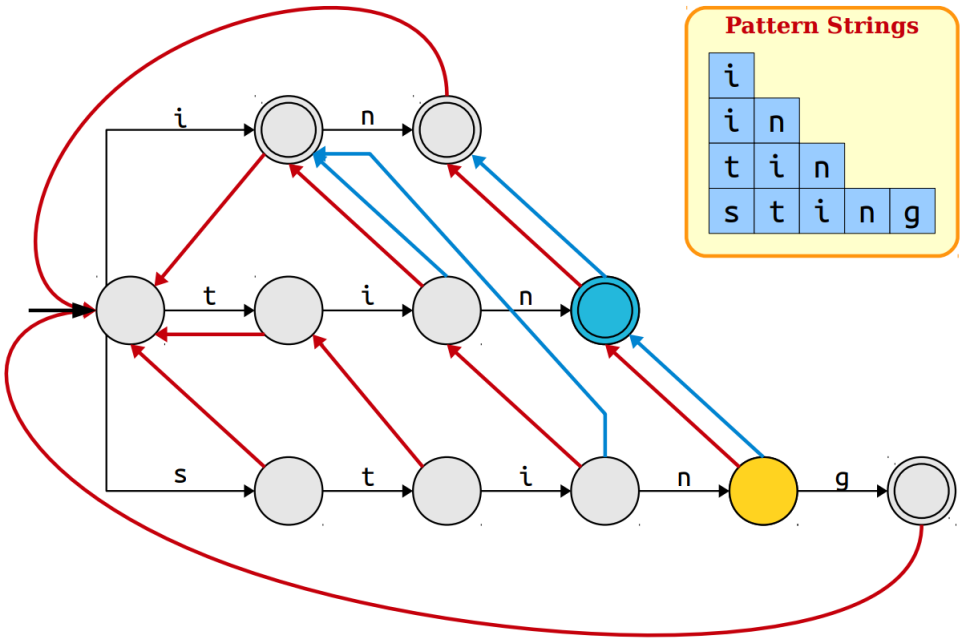
1. Построим автомат по набору строк.
2. Каждый раз будем считывать букву текста, и совершать переход по ней в автомате.

При этом, мы поймем, что нашли совпадение строки из набора, если текущая вершина терминальная, *а также если, двигаясь по суффиксным ссылкам, мы можем достигнуть терминальной вершины.*
Из-за этого отягчающего обстоятельства, определение вхождения строки в текст для каждого состояния будет занимать линейное время. Оптимизируем это движение по суффиксным ссылкам.

5.4 Нахождение сжатых терминальных ссылок.

Определение 22. Сжатая терминальная ссылка вершины v — ближайшая достижимая по суффиксным ссылкам из v вершинаw, являющаяся терминальной и не совпадающая с v.

Пример. Сжатые терминальные ссылки отмечены синим.



Концептуально, сжатые терминальные ссылки собирают в односвязный список все терминальные вершины, а значит и все подстроки, вхождение которых имело место при достижении текущего состояния. Таким образом, учитывая что сжатая терминальная ссылка ведет только наверх, мы сможем построить эти ссылки по слоям при помощи BFS.

```
if (suflink(node)->is_terminal){
    complete_link(node) = suflink(node);
}

else {
    complete_link(node) = complete_link(suflink(node));
}
```

5.5 Алгоритм с использованием сжатых терминальных ссылок.

1. Построим автомат по набору строк.
2. Построим сжатые терминальные ссылки в автомате.
3. Для каждого считанного символа c:
   - 3.1. $cur\_node = jump(root, c)$
   - 3.2. i. Если $cur\_node == null$, вхождение отсутствует, идем дальше.
     ii. В противном случае, засвидетельствуем вхождение подстроки, соответствующей текущей вершине, если она терминальная, а также всех ее сжатых терминальных ссылок.

5.6 Сложность алгоритма.

Пусть Σ — суммарная длина подстрок в наборе, T — длина текста, по которому осуществляется поиск, Ans — число найденных вхождений.

1. Построение бора требует $O(\Sigma)$
2. Построение суффиксных ссылок, по сути, BFS по дереву, требует $O(\Sigma)$
3. Построение сжатых терминальных ссылок, также BFS по дереву, требует $O(\Sigma)$
4. Цикл по каждому из символом — $O(T)$
5. Проходы по сжатым терминальным ссылкам суммарно займут $O(Ans)$

Итого, имеем $O(\Sigma + T + Ans)$

6 Суффиксный массив. LCP. Алгоритм Касаи и др.

6.1 Суффиксный массив.

6.1.1 Определение.

Пусть у нас имеется строка $T =$ "$ababbacba$".

Определение 23.
Суффиксный массив строки T — отсортированный лексикографически массив ее суффиксов.

Пример. Для нашей строки T суффиксный массив имеет вид:

1. $\emptyset$
2. "a"
3. "$ababbacba$"
4. "$abbacba$"
5. "$acba$"
6. "ba"
7. "$babbacba$"
8. "$bacba$"
9. "$bbacba$"
10. "cba"

Для удобства хранения, **мы можем отождествим суффикс с номером символа в T, с которого он начинается**.

6.1.2 Построение.

Соберем номера символов, с которых начинается каждый из суффиксов в массив.

Сортировать можно несколькими способами: Взять все суффиксы и отсортировать, сравнивая строки посимвольно. $\Theta(n^2 \log n)$ Или же, одсчитав хэши для T, бинарным поиском

находить наибольший общий префикс двух суффиксов, а затем сравним первые отличающиеся символы. $\Theta(n \log^2 n)$, делая $n \log n$ сравнений, каждое за $\log n$. Стоит помнить о том, что хэши неточны.

Однако можем делать это еще быстрее. В конец T допишем символ \$, а затем продолжим каждый суффикс, чтобы он имел вид циклического сдвига исходной строки T.

Пример.

1. "\$"
2. "*a*\$*ababbacb*"
3. "*ababbacba*\$"
4. "*abbacba*\$*ab*"
5. "*acba*\$*ababb*"
6. "*ba*\$*ababbac*"
7. "*babbacba*\$*a*"
8. "*bacba*\$*abab*"
9. "*bbacba*\$*aba*"
10. "*cba*\$*ababba*"

Будем считать что лексикографический номер символа \$ меньше чем у любого другого. Таким образом лексикографический порядок сдвига будет совпадать с порядком соответствующего ему суффикса. Мы свели задачу сортировки суффиксов к задаче сортировки циклических сдвигов строки. Для удобства сортировки, **будем считать циклический сдвиг строки бесконечным, тогда все индексы будем брать по модулю длины строки**.

Разделим сортировку на этапы, и на k-том этапе будем сортировать префиксы бесконечных циклических сдвигов строки, имеющие длину 2^k. Будем повторять их пока $2^k \leq |T|$

Пример. $\underline{k = 0}$: Префикс длины 1 — просто первый символ. Так как размах сортируемых значений ограничен воспользоваться сортировкой подсчетом.
Таким образом, мы разбили все сдвиги на блоки по первой букве. Сами блоки отсортированы, однако сдвиги внутри них имеют произвольный порядок.

Тогда каждому сдвигу поставим в соответствие номер — класс эквивалентности по префиксу некоторой длины, так чтобы лексикграфически большим префиксам соответствовал больший номер класса. Пусть $C[i]$ — класс эквивалентности сдвига, начинающегося с позиции i.

Пусть мы находимся на k-том шаге: Мы имеем разбиение сдвигов длины 2^{k-1} на классы, теперь необходимо проделать то же самое со сдвигами длины 2^k. Вспомним что сдвиг длины 2^k состоит из двух половин, классы которых мы посчитали на прошлом шаге. Тогда сравнение сдвигов длины 2^k сводится к сравнению пар $(C[i], C[i+2^{k-1}])$ и $(C[j], C[j+2^{k-1}])$.

Чтобы посчитать классы эквивалентности всех сдвигов по длине 2^k, мы должны просто взять все пары вида $(C[i], C[i+2^{k-1}])$ и стабильно отсортировать их по второму элементу. Затем пройдемся по массиву, и расставим классы эквивалентности, увеличивая номер класса при изменении рассматриваемой пары.

6.2 Наибольший общий префикс.

Хотим для двух данных суффиксов строки T найти наибольший общий префикс. Обозначим его LCP.
Для каждого элемента суффиксного массива подсчитаем его LCP со следующим элементом.

Утверждение 6.1. *Если i-ый и j-ый элементы суффиксного массива имеют общий префикс длины l, то и все элементы с индексами между i и j имеют такой же префикс длины l.*

Доказательство. Действительно, ведь суффиксный массив отсортирован лексикографически. □

Пусть $W[i]$ — позиция i-того суффикса в суффиксном массиве. Пусть $P[i]$ — суффикс, соответствующий i-тому элементу суффиксного массива.

Тогда наибольший общий префикс двух суффиксов i, j $LCP(i,j) = \min\limits_{W[i] \le m \le n \le W[j]} LCP(P[m], P[n])$, из утверждения 6.1, ведь LCP любых двух соседей точно не меньше этой величины, значит у любых суффиксов между i и j имеется такой общий префикс, а значит и у i с j он тоже имеется.
Минимум на отрезке мы умеем считать при помощи sparse table. Возникает вопрос, как оптимально посчитать LCP соседних в массиве суффиксов?

6.3 Решение с помощью промежуточных шагов.

Будем хранить разбиения на классы эквивалентности на каждом шаге, тогда сравнение строк длины 2^k сведется к сравнению $C_k[i] == C_k[j]$.
Если они равны, то можем откусить префиксы длины 2^k и сравнить дальше, иначе же уменьшим k в два раза. Для каждой пары соседних суффиксов мы перебираем k от 0 до $\log n$, значит работать это будет за $O(n \log n)$.

6.3.1 Алгоритм Касаи, Аримуры, Арикавы, Ли, Парка

Пусть $L[i] = LCP(P[i], P[i+1])$.
Заметим, что $L[W[P[i]+1]] \geq L[i]-1$, ведь у если у суффикса $P[i]$ имелся общий префикс с суффиксом $P[i+1]$ длины $L[i]$, то у суффикса $P[i]+1$, полученного из $P[i]$ отбрасыванием первого символа, точно будет общая часть длины $L[i]-1$ с суффиксом $P[i+1]+1$. $P[i]+1$ и $P[i+1]+1$ вовсе не обязательно будут соседствовать в суффиксном массиве, но из 6.1 получим необходимое неравенство. Получив оценку, точно значение будем вычислять наивно.

```
auto L = vector(n);
int current_lcp = 0;

for (size_t i = 0; i < n; i++) {

    if (W[i] == n - 1)
        continue;

    int nxt = W[P[i] + 1];

    while (max(i, nxt) + current_lcp < n &&
           val[i + current_lcp] == val[nxt + current_lcp]) {
            current_lcp++;
        }

    L[W[i]] = current_lcp;
    current_lcp = max(0, current_lcp - 1);
}
```

Наивное прибавление мы делаем не более чем n раз, значит сложность алгоритма $O(n)$

7 Суффиксное дерево. Алгоритм Укконена.

S — строка
S_i — символ на i-той позиции строки. Индексы будем считать с единицы.
S^i — суффикс строки, начинающийся с i-той позиции и включающий ее.

7.1 Суффиксное дерево.

Определение 24. Проходная — вершина с исходящей степенью равной 1.

Чтобы построить суффиксное дерево, припишем к исходной строке S символ \$, и добавим все суффиксы получившейся строчки в бор, а за тем *сожмем проходные вершины*, устранив вершину и склеив ее два ребра в одно. Заметим, что на каждом шаге добавлялась не более чем одна терминальная и не более чем одна обычная вершина, значит всего вершин $O(n)$. Для оптимизации памяти, вместо строк будем хранить индекс начала и индекс конца подстроки в S.

Бор хранит в себе все префиксы добавленных строк, значит получившееся дерево содержит все префиксы всех суффиксов, то есть *содержит все возможные подстроки S*.

Пример. Даю установку вы видите рисунок.

7.1.1 Поиск суффиксных ссылок.

Добавим в получившийся бор суффиксные ссылки. Так как бор содержит все подстроки, то суффиксная ссылка строки отличается от самой строки только отсутствием первого символа. Значит мы можем легко ее посчитать, перейдя из корня по символам кроме первого.
Тогда можем заметить, что суффиксная ссылка листа — лист, ведь каждый листо соответствует суффиксу исходной строки. Из аналогичных рассуждений, ссылка из нетерминальной вершины ведет в нетерминальную вершину.

Утверждение 7.1. *Суффиксная ссылка из вершины не может вести в середину сжатого ребра.*

Доказательство.

1. Если вершина терминальная: то ее суффиксная ссылка — терминальная вершина $\Rightarrow$ не в середине ребра.

2. Если вершина нетерминальная и не корень: то она имеет хотя бы 2 ребенка. Значит строку, соответствующую данной вершине, назовем ее α, можно в исходном тексте продлить двумя символами, c_1 и c_2. То есть в тексте есть два вхождения αc_1 и αc_2. У каждого из вхождений можем отбросить первый символ и продлить символами c_1 c_2. Значит у суффиксной ссылки данной вершины должны быть переходы по c_1 и c_2. Значит ссылка не может быть серединой ребра.

3. Если вершина — корень: то ее суффиксной ссылкой по определению является корень.

□

Однако строка может заканчиваться не в вершине, а в середине ребра. Тогда чтобы найти суффиксную ссылку строки, найдем ссылку вершины, соответствующей наибольшему префиксу данной строки и пройдем из нее по оставшимся символам строки. Это возможно, так как в силу структуры дерева, ссылку любой вершины можно продлить по тем же символам что и исходную вершину.

7.2 Алгоритм Укконена для построения суффиксного дерева.

Пусть $S =$ "$abaabab\$$"
Построим дерево вместе с суффиксными ссылками следующим образом:
Будем проходить по S слева направо и расширять строку текущим символом, достраивая дерево. В каждый момент времени мы храним дерево для уже обработанного префикса строки.

Когда мы получаем новый символ c, каждый суффикс строки увеличивается на этот самый символ.

При добавление символа c, все суффиксы в порядке убывания длины разбиваются на три категории, внутри каждой строки идут подряд.

1. Уже являющиеся листьями.

2. Нелисться, из которых еще не было перехода по c.

3. Нелисться, из которых уже был переход по c.

Тогда

- Если вершина уже лист, то при продлении к нему будет дописываться один символ, это процесс можно ускорить, дописав в строку, соответствующую этому листу исходную строку S до самого конца.
- Если перехода по c еще не было, его нужно создать, а новосозданная вершины станет листом, значит в ребре, ведущем к ней, сразу допишем всю оставшуюся часть S, а не только c.
- Если у вершины уже есть переход по c, то мы просто перейдем по c и перенесем терминальность.

Таким образом с листьями мы быстро разбираемся, дополняя их до конца.
Будем хранить указатель, на самый длинный суффикс, не являющийся листом. Создаем в нем переход по c, который станет листом. Вполне возможно что этот указатель ведет в середину ребра, и чтобы добавить вершину, нам придется это ребро расщепить. Затем перейдем по суффиксной ссылке и проделаем то же самое с ней. Продолжим так пока не окажемся в вершине, в которой уже был переход по c, перейдя из нее по c мы и окажемся в вершине, которая теперь является самым длинным суффиксом и нелистом.

```
cur = pointer to the longest non-leaf suffix.
for c in s:
    while (cur->son[c] == nullptr):
        If necessery, split current edge.
        cur->son[c] = new Node(all suffix til string end);
        cur = suflink(cur)

    cur = go(cur, b)
```

У всех создаваемых вершин, кроме терминальных будем рассчитывать суффиксные ссылки.

7.2.1 Асимптотика.

Переход по суффиксной ссылке делается $O(n)$ раз, ведь после перехода по суффиксной ссылке мы прыгаем на другую ветку, забывая о старой. Значит спускаемся вниз по символам тоже не более чем $O(n)$ раз. Как мы уже знаем, в процессе работы алгоритма создается $O(n)$ новых вершин. Таким образом алгоритм работает за $O(n)$.

8 Хештаблицы, цепочки

У нас есть некоторое множество, в которое мы хотим:

- Добавить (ключ, значение).
- Удалить ключ.
- Получить значение по ключу.

Мы уже умеем делать это при помощи деревьев поиска за $O(log(n))$. Научимся делать это за константу.
Будем считать, что все ключ лежат в некотором множестве U.

8.1 Прямая адресация.

Вспомним сортировку подсчетом, заведем таблицу размером $|U|$, занумеровав элементы.
Теперь все операции сводятся просто к работе с ячейкой массива.
Этот подход очевидно неидеален, ведь множество U может не иметь разумного ограничения сверху.

8.2 Хеш-функции

Чаще всего используется некоторое подмножество U, мы можем использовать этот факт.
Введем хеш-функцию $h : U \to Z_M$.
Теперь будем взаимодействовать с одной из M ячеек, соответствующих значениям Хеш-функции.
Однако и хеш-функция не лишена проблем, взять хотя бы коллизии. Для их разрешения существуют два метода:

- Цепочки.
- Открытая адресация. (Не путать с прямой).

8.3 Метод цепочек

Пусть у нас возникла ситуация, когда образы двух различных ключей совпадают, тогда в ячейке вместо значения, соответствующего ключу, будем хранить связный список пар $(key, value)$, которые в эту ячейку попали.

Оценим время работы Очевидным образом добавление работает за $O(1)$, ведь это просто переход в ячейку и добавление в начало списка.
С удалением и поиском сложнее, ведь они, помимо перехода, должны в худшем случае просмотреть всю цепочку, поэтому их можно оценить как $O(|C^k|)$ — длина соответствующей цепочки.

Замечание. Стоит заметить, что добавлению может потребовать пройтись по цепочке чтобы не добавлять одно и то же значение два раза, тогда время работы будет оцениваться как $O(1 + |C^k|)$

8.3.1 Простое равномерное хеширование

Simple Uniform Hashing. На длины цепочек влияет хеш-функция, поэтому нам хотелось бы подобрать ее оптимальнее.
Заведем $\mathbb{H}$ — пространство всех хеш функций вида $h : U \to Z_M$.

Каждая хеш-функция задается своими значениями на всех возможных ключах, то есть вектором из $|U|$ координат.

Чтобы сконструировать SUH-функцию, будем набирать этот вектор независимо по каждой координате, каждое значение которой имеет вероятность появления $\frac{1}{M}$.

Мат. ожидание времени операции.

Теорема 8.1. *В предположении Simple Uniform Hashing мат. ожидание длины цепочки* $— O(\frac{N}{M})$*, где* N *— число добавленных ключей.*

Доказательство.

$$|C^j| = \sum_{i=1}^{N} I(h(k_i) = j) \Rightarrow E|C^j| = \sum_{i=1}^{N} P(h(k_i) = j)$$

Так как значение функции равномерно выбиралось из всего множества ключей $P(h(k_i) = j) = \frac{1}{M} \Rightarrow E|C^j| = \frac{N}{M}$ □

Определение 25. Load factor — $\alpha = \frac{N}{M}$ — показатель загруженности таблицы.

Следствие 8.1.1. Время работы *неуспешного* поиска значения по ключу или же удаления для всех Хеш-функций имеют мат. ожидание времени работы $O(1 + \frac{N}{M})$

Теорема 8.2. *В предположении SUH среднее по всем ключам мат. ожидание успешного поиска или удаления* $O(1 + \frac{N}{M})$.

Замечание. Это более сильное утверждение чем в предыдущей теореме, которая ничего не говорит о ключах.
По сути, теорема говорит, что если мы выбрали случайную хеш-функцию, а за тем случайный ключ, то для него все будет работать недолго.

Доказательство.
Пусть наши ключи в порядке добавления были $\{k_1, ..., k_N\}$.
Тогда время поиска значения по ключу k_i: $T(k_i) = |\{j > i : h(k_j) = h(k_i)\}|$ – число добавленных после этого совпадающих ключей, ведь как мы помним, новые пары добавляются в начало цепочки.
На языке индикаторов: $T(k_i) = \sum\limits_{j=i}^{N} I(h(k_j) = h(k_i)$. Тогда среднее мат. ожидание по всем ключам это $\frac{1}{N} \sum\limits_{i=1}^{N} E[\sum\limits_{j=i}^{N} I_{i,j}]$
Так как $P(h(k_i) = h(k_j)) = \frac{1}{M}$, то среднее мат ожидание равняется $\frac{1}{N} \sum\limits_{i=1}^{N} [1 + \frac{N-i}{M}] = O(1 + \frac{N}{M})$ □

Rehash Однако с ростом N время работы операций так же растет. Добьемся того чтобы амортизированная оценка была $O(1)$:
Пусть в какой-то момент N превысила M, тогда возьмем новую функцию $h : U \to Z_{2M}$, то есть увеличим число ячеек в таблице, тогда старые ключи придется пересчитать.
Теперь все оценки станут амортизированными.

Затраты памяти Стоит учесть, что для каждого из возможных значений ключа мы должны хранить значение хеш-функции на нем, таким образом требуется $O(|U|)$ памяти. Давайте сделаем лучше, сохранив краеугольный камень — равновероятность каждого значения.

8.3.2 Универсальное хеширование

Universal Hashing

Зафиксируем H — семейство Хеш-функций $h : U \to Z_M$.

Определение 26. Универсальное — семейство хеш-функций, для которого выполняется свойство: $\forall x, y \in U \hookrightarrow P_h(h(x) = h(y)) = O(\frac{1}{M})$.

Нас интересует мат. ожидание длины цепочки для фиксированного ключа:

$$E|C^k| = \sum_{i=1}^{N} P(h(k_i) = h(k)) \leq 1 + O(\frac{N}{M}).$$

Единица появляется так как при совпадении ключей вероятность равенства хешей — единица.

Построим универсальное семейство, параметризованное двумя числами:
Будем хешировать число: $U = \{0, ..., p-1\}$, где p — достаточно большое простое число.

$$h_{a,b}(x) = ((ax + b)\%p)\%m$$

Не трудно заметить, что не имеет смысла брать a и b большие чем $p \Rightarrow a \in 1, ..., p-1, b \in 0, ..., p-1$.

Утверждение 8.3. *Такое семейство — универсально.*

Доказательство.
В силу простоты p $\forall x, y \hookrightarrow ax + b \equiv ay + b(\mod p) \iff x \equiv y(\mod p)$
Таким образом колизия может возникнуть только после того как мы возьмем значение по модулю m.

Проанализируем когда при взятии по модулю бывают коллизии:
Пара $(ax+b, ay+b)$ при фиксированных x и y и варьирующихся a и b пробегает все точки квадрата без диагонали. Даю установку, вы видите рисунок. Значит все точки рано или поздно будут достигнуты.
При этом $P(h(x) = h(y)) = P((ax + b)\%m = (ay + b)\%m) = P(u\%m = v\%m)$

Кроме того, $u, v \in Z_p : u \neq v$, всего таких пар $p(p-1)$.
Тогда разница между u и склеивающимися с ним значениями — ровно m.
Тогда склеивающиеся с u можно количественно оценить сверху как $(\lceil \frac{p}{M} \rceil - 1)$.
Тогда общее число склеивающихся пар оценивается как $p \cdot (\lceil \frac{p}{M} \rceil - 1)$.
Значит вероятность совпадения можно оценить как $\frac{p \cdot (\lceil \frac{p}{M} \rceil - 1)}{p(p-1)} \leq \frac{\frac{p+M-1}{M} - 1}{p-1} = \frac{1}{M}$. □

8.4 Задача Fixed Set, Static Perfect Hashing

По заданному набору из N ключей нам нужно построить Fixed Set — структуру, позволяющую по ключу получать значение и делать это в худшем случае за $O(1)$ по времени, $O(n)$ по памяти и $O(n)$ по времени построения.

Заведем таблицу размера M.
Выберем произвольную хеш-функцию и разложим ключи по ячейкам, руководствуясь этой функцией.
Пусть в i-той ячейке хранятся n_i значений, построим еще одну таблицу, в которой n_i^2 ячеек, и у которой *нет коллизий*, сохраним указатель на эту новую таблицу в ячейке исходной таблицы. Таким образом получение значение по ключу будет происходить за $O(1)$, ведь это просто просто обращение к индексу сначала по глобальной хеш-функции, а затем по локальной.
Таким образом, из неравенства Маркова получаем $P(\#\text{коллизий на } n_i \text{ объектов в } n_i^2 \text{ ячейках} \geq 1) \leq \frac{1}{2}$

Имеем вероятность того что есть коллизия не более чем $\frac{1}{2}$.
Тогда мы можем просто перебирать хеш-функции из семейства и достаточно рано мы получим ту, которая не дает коллизий.

Мы выполнили ограничение по времени, но возможно ситуация, когда глобальная хеш-функцию кладет все элементы в одну ячейку, тогда нарушается условие по памяти.
Чтобы этого избежать будем подбирать глобальную хеш-функцию так, чтобы $\sum n_i^2$ была $O(N)$.
$E \sum |C^i|^2 = E \sum [\frac{|C^i| * |C^i - 1}{2} * 2 + |C^i|]$
Мат. ожидание суммы длин цепочек это в точности N, $\frac{|C^i| * |C^i - 1}{2}$ — количество пар в цепочке, мат. ожидание их суммы это количество коллизий, тогда E $\sum |C^i|^2 = N + 2 * E$ $\#$числа коллизий $= N + 2 \cdot \sum\limits_{i<j} P(h(k_i) == h(k_j)) = N + 2 * \frac{N(N-1)}{2M}$.
В предоположении что $N = M$ получаем $O(N)$. Из неравенства Маркова имеем $P(\sum n_i^2 > 4N) \leq \frac{2N}{4N} = \frac{1}{2}$.

Лемма 8.1. *Пусть $\xi_1, \xi_2, \ldots$ — независимые случайные величины, которые принимают*

$\{0,1\}$ *и* $P(\xi_i = 0) < p \in (0,1)$.
Пусть $\eta = \min\limits_k \xi_k = 1$.
Тогда $E\eta \leq \frac{1}{(1-p)^2}$

Доказательство.

$$P(\eta = \infty) = P(\xi_1 = 0) \cdot P(\xi_2 = 0) \cdot ... \leq \lim_{k \to \infty} p^k = 0$$

$$E\eta = \sum_{k=0}^{\infty} k \cdot P(\xi_1 = \xi_2 = ... = \xi_{k-1} = 0, \xi_k = 1) \leq \sum_{k=0}^{\infty} k \cdot p^{k-1} = \frac{1}{(1-p)^2}$$

□

Тогда мы можем сделать вывод, что мат. ожидание времени перебора каждой из хеш-функций это константа. Всех этих функций $O(N)$, значит и время построение $O(N)$.
Такое хеширование и называется Perfect Hashing.

9 Хеш-таблицы с открытым ключем. Фильтры. Кукушкино хеширование

9.1 Открытый ключ

У нас все еще есть некоторый универсум U, из которого берутся N ключей, а также есть хеш-функция, с помощью которой мы хотим хранить набор ключей, осуществляя добавление, удаление и поиск элементов.

Заведем M ячеек, и при добавлении элемента будем считать его хеш по модулю M, за тем :

- Если ячейка свободна, просто запишем элемент.
- Если же слот занят, то просто пойдем направо, пока не найдем свободную ячейку.

Такой подход называется **линейное пробирование**.

9.1.1 Линейное пробирование

Введем обозначение $run(x, i) = (h(x) + i)\%M$, где x — ключ, а i — номер попытки. Если в нашей таблице есть свободные ячейки, то рано или поздно мы их найдем, однако рано или поздно таблица заполнится, в этом случае мы просто увеличиваем массив в два раза, выбираю новую хеш-функцию и заново вставляя все элементы.

Чтобы произвести поиск элемента x нужно рассмотреть ячейку $h(x)$:

- Если ячейка свободна, то x нет в таблице
- В противном случае, проверяем этот элемент на равенство с x, если он не совпадает, то x нужно искать в следующих ячейках по линейному run. Таким образом идем до первой пустой ячейки или пока не найдем x.

Чтобы удалить элемент x, кажется разумным просто идти по линейному run пока не попадем в пустую ячейки или найдем x и очистить его ячейку, однако в случае если имеется элемент $y : h(y) = h(x)$, очистив ячейку, в которой лежал x, мы сломаем алгоритм поиска y.
Чтобы этого избежать будем не очищать ячейку, а ставить в ней *tombstone*, говорящий о том, что значение существовало, но было удалено.

Тогда очевидным образом придется модифицировать функцию поиска, теперь она должна будет встретив *tombstone* идти дальше по *run*. Функция добавления теперь будет вставлять элемент не только в случае если она нашла пустое место, но и в случае попадания на *tombstone*.

Не смотря на то, что такой подход хорошо ложится на кэш, он имеет очевидную проблему: *run* может длиться долго, например если рядом будет лежать большое количество значений с одинаковым хешем. Однако удачный выбор хеш-функции позволяет этой проблемы избежать.

Определение 27. K-independent Hashing — хеш-функция обладает свойством $\forall x_1, ..., x_k \hookrightarrow h(x_1), h(x_2), ..., h(x_k)$ — независимые в совокупности случайные величины.

Северокорейские ученые доказали, что если для линейного пробирования взять 2-independent хеш, то мат. ожидание времени работы по перебору ячеек $O(\sqrt{n})$, 3-independent — $O(\log n)$, 5-independent — $O(1)$
Простых примеров 5-independent хеш-функций пока не найдено.

Теорема 9.1. *Если функция 2-independent, а во входных данных достаточно энтропии, то для времени работы достигается оценка* $O(1)$.

То есть в реальной жизни, гда данные хорошо распределены, достаточно 2-independent функции.

9.1.2 Квадратичное пробирование

$run(x, i) = (h(x) + (-1)^i i^2$

9.1.3 Двойное хеширование

Зафиксируем две хеш-функции h_1 и h_2, тогда $run(x, i) = (h_1(x) + h_2(x) \cdot i)\%M$
То есть длину прыжка мы параметризуем входными данными.
В таком случай северокорейские ученые говорят, что если h_1, h_2 — 2-independent, то мат. ожидание времени работы $O(1)$

9.1.4 Кукушкино хеширование

Заведем две хеш-функции h_1 и h_2, а также две Хеш-таблицы с одинаковым количеством ячеек.

Теперь, чтобы вставить элемент x:

1. Положим элемент в первую таблицу.
2. Если если ячейка была занята, вытесним старый элемент и положим его во вторую таблицу.
3. Если при добавлении элемента во вторую таблицу, какой-то ключ был вытеснен, положим его в первую таблицу.
4. Будем повторять эти действия до тех пор пока не уложим все элементы или не попадем в бесконечный цикл.
5. Если мы попали в цикл произведем rehash.

Теорема 9.2. *Мат. ожидание времени одной вставки вставки составляет* $O(1)$.
При этом на m *вставок время работы составляет* $O(1)$ *с вероятностью* $1 - O(\frac{1}{m})$

9.2 Фильтры

Мы хотим решать задачу проверки принадлежности объекта множеству, мы уже оптимизировали время работы, однако хочется уметь отвечать на все те же вопросы не тратя много памяти.

9.2.1 Фильтр Блума

Для увеличения эффективности по памяти пожертвуем чуть-чуть корректностью нашей структуры, то есть разрешим ложно-положительные ответы с некоторой вероятностью ε, но однозначно запретим ложно-отрицательные.

Заведем k равномерных хеш-функций и битовый массив размера M.

Теперь, чтобы добавить элемент x, вычислим значения всех k хеш-функций, и для каждого из значений выставим в массиве бит, порядковый номер которого равен этому значению. Если бит уже был выставлен, то его не нужно менять.

Тогда чтобы проверить наличие элемента, нужно посчитать значения хеш-функций и проверит выставленность битов.

Вероятность ошибки Чтобы ошибка произошла нужно, чтобы у отсутствующего в таблице ключа были выставлены все биты.

Пусть у нас вставляется один ключ, посчитаем вероятность с которой в заданный бит будет выставлена единица. Из не совсем верного предположения независимости битов можем сделать грубую оценку:

После вычисления h_1 имеем вероятность того, что бит не был выставлен $1-\frac{1}{M}$, ведь функция равномерна.

Тогда вероятность того, что бит был выставлен после добавления n ключей — $1-\left(1-\frac{1}{M}\right)^{k\cdot n}$

Значит вероятность того, что конкретные k бит были выставлены составляет $\left(1-\left(1-\frac{1}{M}\right)^{k\cdot n}\right)^k$

Оценка без предположения независимости на самом деле $\frac{1}{M^{k(n+1)}}\cdot\sum\limits_{i=1}^{M} i^k\cdot C_m^i\cdot F(kn,i)$, где $F(kn,i)$ — число сюръекций из множества размера kn в множество размера i.

В силу естественных причин мы воспользуемся грубой оценкой.

Оптимальное количество хеш-функций Анализ показывает что оптимум при фиксированных N и M составляет $\frac{M}{N}\cdot\log 2$.

При фиксированных N и ε — $M=\frac{n\log\varepsilon}{(\log 2)^2}\Rightarrow M\approx 1.44\log_2\frac{1}{\varepsilon}$ и $k=\log_2\frac{1}{\varepsilon}$

Применение фильтра Блума часто имеет место в базах данных, которые сначала пропускают запрос через быстрый фильтр, отклоняющий неинтересные запросы.

Существенный недостаток фильтра Блума заключается в том, что нельзя просто взять и удалить элемент.

9.2.2 Кукушкин фильтр

Так же быстр, но позволяет удалять элемент. Заведем две таблицы, в которых будем хранить *отпечатки ключей* — битовые строки $f(k)$.

Для того чтобы удобно переносить значения из одной таблицы в другую, нам нужно уметь быстро считать h_1 и h_2 по отпечатку ключа.

Зафиксируем $h_1(x)=hash(x)$. Тогда примем $h_2(x)=h_1(x)\oplus hash(f(x))$.

Тогда имея одну из хеш-функций и f мы легко можем посчитать другую.

Не смотря на то, что пространство $f(x)$ сильно уже пространства x, это не влияет на производительность.

На один объект в кукушкином фильтре используется $\frac{\log_2(1/\varepsilon)+3}{\alpha}$, где α это показатель загруженности.

10 Сложность вычислений.

Всю свою жизнь вы занимались Fine-grained complexity, то есть имея алгоритм за $O(n^2)$ сделать алгоритм за $O(n \log n)$.

В сложности вычислений мы отходим от тонких материй, переставая различать $O(n)$ и $O(n^2)$ и будем считать эффективными – задачи класса P.

Определение 28. Класс задач P — класс задач, которые решаются за полиномиальное время.

Кроме того, мы поймем какие задачи на сегодняшний день не представляется возможным решать за $O(poly(n))$.

10.1 Машина Тьюринга.

Определение 29. Машина Тьюринга — набор объектов $(\Sigma, \Gamma, Q, \delta, q_s, q_a, q_r)$, где :

- Σ — входной алфавит, символы, которые могут быть переданы на вход.
- Γ — ленточный алфавит, то что может быть напечатано на ленту. $\Sigma \subset \Gamma$, $\# \in \Gamma$ — пробельный символ, Γ — конечно.
- Q — множество состояний, Q — конечно.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, N, R\}$ — функция переходов, получив состояние и символ, она вернет состояние, в которое нужно перейти, символ, который нужно вывести, а также направление, по которому нужно сдвинуться по ленте.
- q_s — стартовое состояние.
- q_a — терминальное состояние, свидетельствующее о том, что полученное слово — хорошее.
- q_r — терминальное состояние, свидетельствующее о том, что полученное слово — плохое.

То есть наша машина будет рассматривать слово, определяя, лежит ли оно в языке, и давать бинарный ответ — да или нет.

Не будем формально вводить вычисление, интуитивно говоря, мы имеем полоску бумаги, разделенную на клетки, изначально заполненные пробелами. На каком-то куске ленты

написано входное слово $x \in \Sigma^*$, которе мы будем определять на принадлежность рассматриваемому языку.

Пример. Правда ли что x — правильная скобочная последовательность.

В начальной конфигурации головка указывает на начало слова x и машина находится в состоянии q_s. На каждый такт работы машины, мы применяем функцию δ. Сама функция как бы зашита в управляющее устройство машины.

Применение δ продолжается пока мы не придем в терминальное состояние q_a или q_r.

Определение 30. Будем говорить, что **машина распознает язык** $L \subset \Sigma^*$ **за время** T, если при любом входе из языка L она приходит в верное терминальное состояние не больше чем за T.

10.1.1 Многоленточная машина.

Определение 31. Многоленточная машина Тьюринга — набор объектов $(\Sigma, \Gamma, Q, \delta, k, q_s, q_a, q_r)$, где:

- k — количество лент.
- $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, N, R\}^k$.

То есть на каждой ленте есть своя головка, которая по ней перемещается, независимо от других головок.

Теорема 10.1. *Если язык L распознается на k-ленточной машине за $T(n)$, то существует одноленточная машина, распознающая L за $O(k \cdot T^2(n)$*

Замечание. Именно поэтому не имеет смысла различать по эффективности полиномиальное время.

Определение 32. $DTIME(T(n))$ — класс языков, которые распознаются за T на многоленточной машине.

Определение 33. Класс языков P — $\bigcup\limits_{c=1}^{\infty} DTIME(n^c)$.

Определение 34. Класс языков EXP — $\bigcup\limits_{c=1}^{\infty} DTIME(2^{n^c})$.

10.1.2 Недетерминированная машина.

Определение 35. Недетерминированная машина Тьюринга — набор объектов $(\Sigma, \Gamma, Q, \delta, k, q_s, q_a, q_r)$, где:

- $\delta : Q \times \Gamma^k \rightrightarrows Q \times \Gamma^k \times \{L, N, R\}^k$. — многозначная функция, то есть для одного входного значения есть несколько переходов, из них мы выбираем произвольно.

Определение 36. Недетерминированная машина **принимает слово** x если хотябы в одной из ветвей вычислений достигается q_a.

Определение 37. Говорим что M отвергает x, если она его не принимает. Время работы — максимальная длина ветви.

Определение 38. $NTIME(T(n))$ — класс языков, которые распознаются на недетерминированной машине Тьюринга за время $T(n)$.

Определение 39. $NP - \bigcup\limits_{c=1}^{\infty} NTIME(n^c)$.

Замечание. Здесь NP — **Nondeterministic-polynomial**, а вовсе не Non-polynomial.

Теорема 10.2. $P \subset NP \subset EXP$

Доказательство.
Перво вложение верно очевидным образом.
Второе вложение верно так как в каждой точке имеется константное количество ветвлений переходов, а значи их все можно смоделировать за экспоненциальное время. □

Определение 40. Язык A **сводится полиномиально к** языку B: $A \leq_p B$ — существует полиномиально-вычислимая функция $f : \forall x \hookrightarrow x \in A \iff f(x) \in B$.

Замечание. То есть мы можем однозначно определить лежит ли x в A, определив принадлежность $f(x)$ к B.

Определение 41. NP-трудный — язык, такой что $\forall A \in NP \hookrightarrow A \leq_p B$

Определение 42. NP-полный — NP-трудный язык, который и сам лежит в NP.

Замечание. То есть это значит самый сложный в классе NP.

Пример. Примеры NP-полных языков.

- $SAT = \{\phi : phi$ пропозициональная выполнимая формула. $\}$

- $CLIQUE = \{(G, k) : \omega(G) \geq k\}$.
- $COL = \{(G, k) : \chi(G) \leq k\}$.
- $DHAMPATH = \{G :$ в нем есть гамильтонов путь. $\}$

Теорема 10.3. *$A \in NP \iff \exists$ детерм. машина $V(x, s)$, работающая за poly(x) :*
$\forall x \hookrightarrow x \in A \iff \exists s : V(x, s) = 1$

Замечание. Машина V называется верификатором. Аргумент s называется сертификатом. Концептуально условие означает, что какой бы мы сертификат не предложили машине, она не назовет слово не из A хорошим.
То есть принадлежность языка классу NP означает, что для каждого из его слов есть какой-то сертификат, убеждающий машину в том, что это слово хорошее. Таким образом принадлежность к $A \in NP$ легко проверить.

Пример. При помощи теоремы проверим, что языки действительно принадлежать NP.

- SAT — действительно, в качестве сертификата можем передать тот самый набор, который выполняет ϕ.
- $CLIQUE$ — в качестве сертификата передадим те самые k вершин, образующие клику.
- COL — сертификат это раскраска в k цветов.
- $DHAMPATH$ — гамильтонов путь.

Теорема 10.4. *(Кука – Левина)*
SAT — NP-полный язык.

Доказательство.
Построив сертификат мы уже доказали что язык лежит в классе NP, теперь покажем что он NP-трудный.
Для этого покажем, что любой язык из NP можно к нему свести:
Пусть $L \in NP$ то есть имеется детерм. машина $V(x, s)$, работающая за poly($|x|$), такая что $\forall x \hookrightarrow x \in L \iff \exists s : V(x, s) = 1$.

Тогда для заданного s запишем всю цепочку вычислений машины до того как она придет в q_a.
На каждом такте головка перемещается не более чем на одну клетку, а значит если машина

работает за полиномиальное время, то и перемещение головки также будет полиномиальным. То есть нашу рассматриваемую область можно ограничить на poly(x) до начала входных данных и на poly(x) после начала входа.
Будем рассматривать таблицу, каждая строка котрой — рассматриваемая часть ленты на некотором шаге. Ее высота также poly(x)
Заведем булеву функцию ψ, которая будет кодировать корректность вычисления, а именно то, что:

- Стартовая конфигурация именно та, которая нужна: (q_start, x, s)
- Каждая следующая строка получается из предыдущей с помощью перехода по δ.
- В последней строке есть q_a.

Каждая из ячеек таблицы будет переменной ψ. При этом можно утверждать, что вся таблица кроме некоторой фиксированой части, которая строится по входным данным s, фиксирована, тогда:
$x \in L \iff \exists s : \psi(s) = 1$. То есть x лежит в языке если вычисление можно сделать корректным, подставив некоторый сертификат.
Теперь будем кодировать все символы из исходного алфавита Σ последовательностями битов фиксированной длины, например 10. Теперь ячейка машины Тьюринга будет соответствовать 10 булевым переменным.
<...> Я вижу тут долгое рассуждение доказывающее корректность, а вы?
Таким образом, SAT — NP-трудный язык, а значит и NP-полный.

□