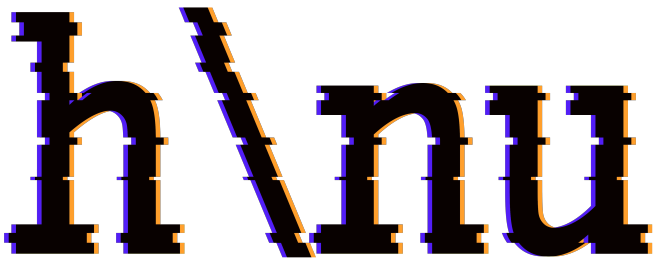
Федеральное государственное автономное учреждение
высшего профессионального образования

Московский Физико-Технический Институт
КЛУБ ТЕХА ЛЕКЦИЙ

Алгоритмы и структуры данных

III СЕМЕСТР
Физтех Школа: *ФПМИ*
Направление: *ПМИ/КТ*
Лектор: *Мацкевич Степан Евгеньевич*



Автор: *Евсюков Никита*
Проект на GitHub

2020 года

Содержание

Важно!

Прошу обратить внимание, что это не конспект лекций, а конспект курса. Большая часть информации взята с Кормена, emaxx, neerc и других источников.

1 Лекция 1: Поиск строк

1.1 Основные понятия.

Рассмотрим задачу поиска подстрок. Формализуем её следующим образом.

Утверждение. Пусть текст задан в виде массива $T[1 \dots n]$, а образец (шаблон) — в виде массива $P[1 \dots m]$, где $m \leq n$. Причём элементы массивов — символы из конечного алфавита.

Определение 1.1. Символьные массивы P и T называют строками символов.

Пусть Σ — конечный алфавит, а Σ^* — множество всех строк конечной длины, образованных с помощью этого алфавита, тогда введём некоторые формальные определения.

Определение 1.2. ω — префикс строки x (обозначается как $\omega \sqsubset x$), если $x = \omega y$ для некоторого $y \in \Sigma^*$

Определение 1.3. ω — суффикс строки x (обозначается как $\omega \sqsupset x$), если $x = y\omega$ для некоторого $y \in \Sigma^*$

Утверждение. Префикс (суффикс) может быть собственным, это означает, что он не совпадает с самой строкой.

1.2 Постановки задачи поиска. Тривиальный алгоритм поиска подстроки в строке.

Для начала дадим несколько вспомогательных определений.

Определение 1.4. P встречается в тексте T со сдвигом s, если $0 \leq s \leq n - m$ и $T[s + j] = P[j]$ для $1 \leq j \leq m$

Определение 1.5. s — допустимый сдвиг, если P встречается в T со сдвигом s.

Утверждение. Задача поиска подстроки таким образом представляет собой задачу поиска всех допустимых сдвигов

Рассмотрим тривиалный алгоритм поиска подстроки в строке

```
1  n = T.size();
```

```
  m = P.size();
  for (size_t s = 0; s < n - m; ++s) {
    if (T.substr(s + 1, s + m) == P) {
      std::cout << s << std::endl;
    }
  }
```

Утверждение. Очевидно, что время его работы равно $O((n - m + 1)m)$, что в худшем случае ($m = \frac{n}{2}$) равно $O(n^2)$.

1.3 Префикс-функция. Тривиальный алгоритм нахождения.

Определение 1.6. Префикс-функция — массив чисел $\pi[0 \dots n-1]$, где $\pi[i]$ — наибольшая длина наибольшего собственного суффикса подстроки $s[0 \dots i]$, совпадающего с её префиксом. Или же

$$\pi[i] = \max\{k : (k < i) \wedge (s[0 \dots k-1] = s[i-k+1 \dots i]\}$$

$\pi[0]$ положим равной нулю.

Например, у подстроки abca строки abcabcd префикс длины 1 совпадает с суффиксом, т.е. $\pi[3] = 1$

Исходя из определения можно написать простейший алгоритм

```
std::vector<int> prefix(std::string& s) {
  std::vector<int> pi(s.length());

  for (size_t i = 0; i < s.length(); ++i) {
    for (size_t k = 0; k <= i; ++k) {
      if (s.substr(0, k) == s.substr(i - k + 1, k)) {
        pi[i] = k;
      }
    }
  }
}
```

Утверждение. Очевидно, что его асимптотика $O(n^3)$.

1.4 Линейный алгоритм нахождения. Доказательство времени работы.

Оптимизируем наш простейший алгоритм. Для начала заметим, что префикс функция увеличивается не более, чем на единицу, т.е. $\pi[i+1] - \pi[i] \leq 1$. Т.е. могло произойти не более n увеличений функции (каждый раз увеличение не более чем на 1) и, соответственно, не более n уменьшений.

Утверждение. Алгоритм может иметь асимптотику $O(n^2)$

Действительно, достаточно заметить, что нам нужно произвести $O(n)$ сравнений строк (сравнение необходимо только при увеличении/уменьшении префикс-функции).

Далее нужно как-то избавиться от тяжёлой операции сравнения подстрок. **Но как это сделать?** Будем использовать то, что мы уже посчитали.

Утверждение. Если $s[i+1] = s[\pi[i]]$, то $\pi[i+1] = \pi[i] + 1$

Доказательство легко видно на рисунке

$$\overbrace{\overbrace{s_1 s_2}^{\pi[i]}\ \overbrace{s_3}^{s_3 = s_{i+1}}}^{\pi[i+1]} \ldots \overbrace{s_{i-1} s_i}^{\pi[i]}\, s_{i+1}$$

А что делать, если $s[i+1] \neq s[\pi[i]]$? Тогда попробуем рассмотреть суффикс поменьше длиной k и проверить, существует ли равный ему префикс, в таком случае, если мы найдем максимальное такое k, то нам останется проверить равенство $s[i+1] = s[k]$.

Покажем на примере

$$\underbrace{\overbrace{s_0 s_1}^{\pi[i]}\, s_2 s_3}_{k} \ldots \underbrace{\overbrace{s_{i-3} s_{i-2}}^{\pi[i]}\, s_{i-1} s_i}_{k}$$

Утверждение. $k = \pi[\pi[i] - 1]$ (вычитание единицы из-за нумерации строк с 0)

Это легко вытекает из предыдущего рисунка, действительно, если мы рассмотрим суффикс длины $\pi[i]$ и найдём в нём ещё один суффикс, отвечающий нашему условию, то мы получим требуемое.

Таким образом, получим итоговый алгоритм:

1. Считаем $\pi[i]$ от $i = 1$ до $i = n - 1$
2. Тестируем образец длины j по описанной выше схеме
3. Останавливаем перебор при $j = 0$

```
1  std::vector<int> prefix(std::string &s) {
2    std::vector<int> pi(s.length());
3
4    for (size_t i = 1; i < s.length(); ++i) {
5      size_t j = pi[i - 1];
6      while (j > 0 && s[i] != s[j]) {
7        j = pi[j - 1];
8      }
9      if (s[i] == s[j]) ++j;
10     pi[i] = j;
11   }
12   return pi;
13 }
```

Утверждение. Представленный алгоритм работает за $O(n)$

1.5 Подсчёт префикс-функции для строки q$t. Алгоритм Кнута-Морриса-Пратта.

Пусть $|q| = n, |t| = m$. Рассмотрим значение префикс-функции в таком случае.

Утверждение. Значения $\pi[i]$ при $i > n$ равны 0, а равенство $\pi[i] = n$ означает окончание вхождения искомого паттерна.

Таким образом получаем алгоритм Кнута-Морриса-Пратта. Т.к. значение префикс-функции не может превысить n мы можем хранить только искомую строку (следует из самого алгоритма описанного выше). Таким образом получаем требуемую асимптотику.

Утверждение. Алгоритм Кнута-Морриса-Пратта работает за $O(n + m)$ и $O(n)$ памяти.

1.6 Z-функция. Тривиальный алгоритм нахождения.

Определение 1.7. Z-функция строки s — массив длины n ($|s| = n$), где $z[i]$ — длина наибольшего общего префикса строки s и её i-ого суффикса.

Рассмотрим тривиальный алгоритм, который перебирает ответ для каждого i

```
1  std::vector<int> zFunction(std::string& s) {
2    std::vector<int> z(s.length());
3
4    for (size_t i = 1; i < s.length(); ++i) {
5      while ((i + z[i] < n) && (s[z[i]] == s[i + z[i]])) {
6        ++z[i];
7      }
8    }
9    return z;
10 }
```

Утверждение. Асимптотика такого алгоритма, очевидно, $O(n^2)$

1.7 Линейный поиск Z-функции. Доказательство времени работы.

Для оптимизации алгоритма воспользуемся тем же, т.е. будет использовать вычисленные значения.

Определение 1.8. Отрезок совпадения — подстрока, совпадающая с префиксом строки s

Например, для строки abab, ab — отрезок совпадения.

Будем вычислять значения z-функции по очереди от $i = 1$ до $n-1$ и хранить значения $[l; r]$ самого правого отрезка совпадения, т.е. r указывает нам на правую границу, до которой просканировал алглоритм.

Далее на некотором i-ом шаге возможно два случая:

- $i > r$, тогда нам ничего не известно про следующие символы, т.к. они не были просканированы алгоритмом, так что запустим тривиальный алгоритм от этого i, после чего (если $z[i] > 0$) обновим значения $[l; r]$
- $i \leq r$, тогда мы можем инициализировать значение $z[i]$ чем-то большим 0. Рассмотрим на примере.

Утверждение. В случае $i \leq r$ можно проинициализировать $z[i]$ таким образом, а далее аналогичным образом запустить тривиальный алгоритм поиска.

$$z[i] = \min(r - i + 1, z[i - l])$$

Доказательство этого утверждения увидим на рисунке

$$a_1 a_2 a_3 \underbrace{\overbrace{a_4}^{i-l} a_5}_{z[i-l]} \ldots a_l a_{l+1} a_{l+2} \underbrace{a_i a_r}_{z[i-l]}$$

Т.к. мы знаем, что $[l; r]$ — отрезок совпадения, а $z[i - l]$ мы уже посчитали, то для начального инициализации можно будет использовать посчитанное значение. Ограничение в $r - i + 1$ нужно для того, чтобы повторяющийся кусок не вышел за границы r, т.к. мы ничего не знаем о символах после r.

Приведём реализацию:

```
1  std::vector<int> zFunction (std::string& s) {
2    int l = 0, r = 0;
3    std::vector<int> z(s.length());
4
5    for (int i = 0; i < s.length(); ++i) {
6      if (i <= r) {
7        z[i] = std::min(r - i + 1, z[i - l]);
8      }
9
10     while ((i + z[i] < n) && (s[z[i]] == s[i + z[i]])) {
11       ++z[i];
12     }
13
14     if (i + z[i] - 1 > r) {
15       l = i;
16       r = i + z[i] - 1;
17     }
18   }
19   return z;
20 }
```

Утверждение. Асимптотика данного алгоритма $O(n)$

Доказательство. Достаточно рассмотреть цикл *while*, т.к. остальные операции выполняются за константу. Заметим что при $i > r$ каждая итерация цикла продвигает r вправо (кроме случаев, когда $s[0] \neq s[i]$, тогда вовсе не будет итераций цикла *while*. Если же $i \leq r$,

итерация цикла либо продвинет r, либо не случится вовсе. В таком случае, итераций цикла *while* будет не более n штук. ■

1.8 Применение для поиска подстроки в строке.

Применение и оптимизация памяти аналогично префикс-функции.

2 Лекция 2: Бор. Алгоритм Ахо-Корасик.

2.1 Структура данных бор.

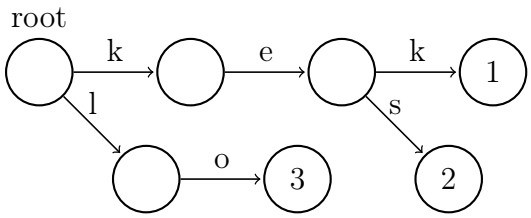
Рассмотрим дерево, в котором каждому ребру соответствует некоторый символ. Тогда мы можем интерпретировать символы на ребрах на пути из корня в вершину как символы некоторой строки. Мы получили бор, или же, более формально:

Определение 2.1. Бор — структура данных для хранения строк, которая представляет из себя подвешенное дерево с символами на рёбрах и удовлетворяет некоторым свойствам.

Основные свойства бора:

- Нет ребёр с одинаковыми символами, выходящих из одной вершины.
- Вершины, которые являются последними в некоторой строке, помечаются как терминальные.

Пример бора.



В примере выше массив строк следующий: *kek, kes, lo*.

Теперь давайте разберёмся, как строить бор. Будем хранить в каждой вершине информацию о том, является ли она листом (т.е. существует ли строка, которая оканчивается на вершине) и массив указателей на детей вершин `next`, где `next[i]` — указатель на вершину, в которую ведёт ребро с индексом `i`.

Теперь для построения бора встанем в корень и посмотрим, есть ли из неё переход по ребру `s[i]` (`s` — строка, которую мы вставляем в бор), так проходим по всем символам строки и ставим метку на вершине, в которую дошли.

Утверждение. Вставка в **бор** происходит за $O(n)$, а потребляемая память — $O(nk)$, где k — размер алфавита.

2.2 Алгоритм Ахо-Корасик.

Пусть у нас есть набор строк-образцов `pattern[i]` и некоторая строка `s`, необходимо эффективно искать всех вхождения образцов в текст. Для этой задачи воспользуемся алгоритмом **Ахо-Корасик.**

1. Строим **бор** из входных строк за $O(m)$, где m—длина строк.
2. Создание суффиксных ссылок.

   Это who? Для их осознания представим ситуацию, что мы, как обычно, гуляли по бору, но тут вдруг оказалось, что идти нам некуда (вот блин). Тогда, с одной стороны, мы можем пойти в корень и отправиться гулять дальше, но вдруг какой-то паттерн начинается также, как и суффикс того, по чему мы прошли. Например, у нас набор строк состоит из $abcd$ и bca, и вот мы гуляем по строке $abca$. Мы пришли по рёбрам a, b, c бора и оказывается, что есть только один путь (не путю) по ребру d, а следующий символ в нашей строке a, что вместе с пройденными ранее bc образуют один из наших паттернов. Т.е. нам нужно научиться как-то быстро переходить к наибольшему собственному суффиксу пройденного пути в случае неудачи. На помощь приходят суффиксные ссылки! Теперь более формально.

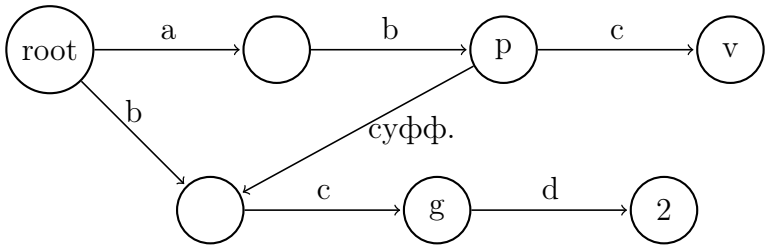
   Определение 2.2. Суффиксная ссылка — это вершина, в которой оканчивается наидлиннейший собственный суффикс. Суффиксная ссылка корня для удобства будет ссылкой на корень.

   Теперь пока в текущей вершине нет перехода по соответствующей букве, мы выполняем переход по суффиксной ссылке. Таким образом мы вводим важное понятие функции перехода из вершины v по символу c, которая определяется следующим образом:

   - корень бора, если текущая вершина — корень.
   - `v.next[c]`, где `next` — массив переходов из текущей вершины, если переход `c` существует.
   - `v.child[c]`, где `child` — массив детей из текущей вершины, если ребёнок по ребру `c` существует.
   - функция перехода по символу c от вершины по суффиксной ссылке.

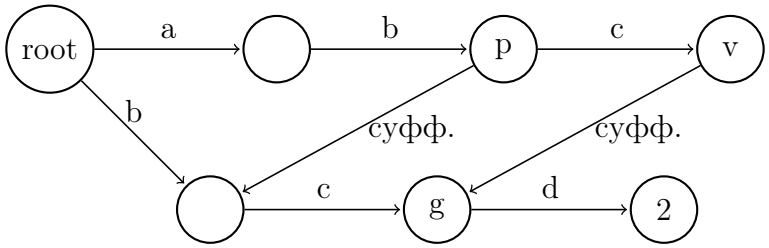
   Теперь научимся искать суффиксные ссылки. Заметим, что нам достаточно пройти в родителя вершины (пусть мы сделали это по ребру c), пройти по его суффиксной ссылке и выполнить переход (т.е. функцию перехода) по ребру c. Интересно, что функцию перехода мы определили с использованием суффиксных ссылок, а суффиксные ссылки — с помощью функции перехода. Мы получили двойную рекурсию. Рассмотрим подробнее на примере.

Пример нахождения суффиксной ссылки



Пусть у нас есть такой бор и уже есть суффиксная ссылка для вершины p. Теперь найдём суффиксную ссылку для вершины v. Пойдём в родителя p и перейдём по его суффиксной ссылке. Мы получили возможный максимальный собственный суффикс для нашей вершины v, осталось выполнить переход по символу c, в данном случае мы попадём в следующую вершину g, но могли перейти по следующей суффиксной ссылке и ещё уменьшить наш суффикс. Итого:

Пример нахождения суффиксной ссылки



Таким образом суффиксные ссылки и функцию перехода можно вычислять ленивой динамикой. Достаточно сохранять уже посчитанные значения функции перехода и суффиксных ссылок.

3. Создание сжатых суффиксных ссылок.

   Возможна ситуация, когда суффиксных ссылок становится слишком много. Например, если наш бор выродился в «бамбук» без ветвления. Тогда нам дорого хранить массив всех шаблонов, которые оканчиваются в вершине. Решением большого количества переходов по суффиксным ссылкам и излишнего хранения могут стать сжатые суффиксные ссылки.

   Определение 2.3. Сжатая суффиксная ссылка — такая ссылка, которая образована несколькими переходами по обычным суффиксным ссылкам, и ведёт сразу в терминальную вершину.

   Тогда решением проблемы хранение будет обход по сжатым суффиксным ссылкам. Также с помощью сжатых суффиксных ссылок решается задача о поиске всех строк из заданного набора в тексте.

2.3 Оценка времени работы алгоритма Ахо-Корасик.

Пусть n — количество вершин, m — суммарная длина паттернов, k — количество ответов.

Тогда обработка одного символа текста складывается из:

- Построение бора по паттернам за $O(m)$.
- Вызов функции перехода. Ленивое построение для каждого узла и символа вычисляет функцию не более раза. Итого: $O(n+m)$.
- Построение суффиксных ссылок по одному вызову на каждую вершину бора. Каждый переход по сжатой ссылке либо находит ответ, либо приходит в корень.

Итого получаем: $O(n+m+k)$.

3 Лекция 3: Суффиксный массив.

3.1 Суффиксный массив. Построение за $O(n^2 \log n)$.

Пусть нам дана строка s длины n.

Определение 3.1. i-ый суффикс строки — подстрока $s[i \dots n-1]$.

Определение 3.2. Суффиксный массив — перестановка индексов суффиксов в лексикографическом порядке, т.е. сортированный массив собственных суффиксов строки.

Попробуем наивно построить наш массив. Для начала заметим, что сортировка суффиксов эквивалентна сортировке циклических сдвигов строки с добавленным символом сентиленом, символом, который заведомо не встречается в строке. Действительно, любой суффикс можно получить из строки циклическим сдвигом при условии, что мы знаем, где конец у строчки (символ сентилен).

Тогда сравнение строк происходит за $O(n)$, используя сортировку за $O(n \log n)$, получаем итоговую асимптотику $O(n^2 \log n)$.

3.2 Поиск подстроки в тексте с использованием суффиксного массива.

С помощью суффиксного массива можно искать все вхождения строки в другую. Для начала рассмотрим наивное решение.

1. Строим суффиксный массив.
2. Наши суффиксны отсортированы, так что бинарным поискам по массиву можно найти диапазон, в котором исследуемая строка имеет тот же первый символ, что и суффиксы в диапазоне. Отбрасывая первый символ, мы не теряем отсортированность, так что мы можем аналогично применить бинарный поиск.

Утверждение. Асимптотика такого алгоритма $O(n \log m)$, где n — длина образца, а m — длина текста.

Для доказательства заметим, что сравнение суффикса с образцом не превышает длину образца.

Теперь рассмотрим более быстрый алгоритм. Во время бинарного поиска паттерн и все суффиксы из рассматриваемого диапазона имеют общий префикс. Тогда идея в том, чтобы не сравнивать все символы из префикса каждый раз.

Для начала введём новое понятие:

Определение 3.3. LCP(x,y) — длина наибольшего общего префикса строк x и y.

Общая идея алгоритма в том, что мы ищем границы ответа с помощью бинарного поиска по суффиксному массиву. Опишем поиск левой границы ответа, правая ищется аналогично.

1. Если образец больше последнего суффикса и меньше первого, то поиск можно останавливать, т.к. образец не встречается в строке.
2. Введём некоторые обозначения:
   - L — левая граница поиска в суффиксном массиве (т.е. индекс элемента, левее которого мы не ищем).
   - R — правая граница поиска.
   - $M = \frac{L+R}{2}$.
   - p — наш паттерн.
   - $suffix[]$ — суффиксный массив.
   - $l = LCP(suffix[L], p)$ — наименьший общий префикс элемента с левой границы поиска и нашего паттерна.
   - $r = LCP(suffix[R], p)$.
   - $m_l = LCP(suffix[L], suffix[M])$.
   - $m_r = LCP(suffix[R], suffix[M])$.
3. Положим изначально $L = 0$, $R = |S|$, где S — изначальная строка.
4. На каждом шаге будем сравнивать l и r (изначально их можно найти с помощью алгоритма Касаи (о нём пойдёт речь далее).

   Тогда, если $l \geq r$:

   - $m_l > l$. Тогда мы получили, что элементы из $[L, M]$ имеют общий префикс больше, чем общий префикс левого суффикса и паттерна. Тогда мы должны пойти в отрезок $[M, R]$, ведь тогда на всём отрезке $[L, M]$ будет общий префикс такой, что он имеет различные символы с префиксом p той же длины.
   - $m_l = l$. Тогда нам остаётся проверить следующие символы $suffix[M]$ и p и найти первый различный: если первый различный у p больше, чем у $suffix[M]$, то идём в отрезок $[M, R]$, иначе — в $[L, M]$.
   - $m_l < l$. Аналогично первому случаю идём в $[L, M]$.

   Если $l < r$ всё аналогично. Запишем в сжатой форме:

- $m_r > r \Rightarrow R = M$.
- $m_r = r \Rightarrow$ считаем $LCP(suffix[M], p)$, действуем в зависимости от различного символа.
- $m_r < r \Rightarrow L = M$.

Тогда асимптотика полученного алгоритма $O(p+log(S))$, если мы предпосчитали LCP (делаем не более $O(p)$ сравнений символов и $O(log(S))$ занимает бинарный поиск по массиву суффиксов). Иначе $O(p \log S)$ в худшем случае.

3.3 Построение суффиксного массива за $O(n \log n)$.

Для начала отметим, что сортировать мы будем не суффиксы, а циклические сдвиги строки с сентиленом (что эквивалентно). Соответственно введём понятие циклической подстроки $s[i \dots j]$ при $j < i$: $s[i \dots n-1] + s[0 \dots j]$.

Наш алгоритм состоит из $\lceil \log n \rceil$. На i-ой фазе будут сортироваться циклические подстроки длины 2^i. На каждой фазе мы поддерживаем массив перестановки циклических подстрок p и массив классов эквивалентности для подстрок.

Определение 3.4. Массив классов эквивалентности — массив, который удовлетворяет следующим свойствам:

- Если один суффикс меньше другого, то и номер класса он получает меньший.
- Если суффиксы равны, то и их классы эквивалентности равны.
- Номер класса $c[i]$ соответствует подстроке, начинающейся с индекса i.

Рассмотрим на примере первые два шага алгоритма для строки *aba*:

$$\begin{aligned} 0: &\quad p = \{0, 2, 1\} \quad c = \{0, 1, 0\} \\ 1: &\quad p = \{1, 2, 0\} \quad c = \{1, 2, 0\} \end{aligned}$$

Теперь перейдём непосредственно к алгоритму. Рассмотрим нулевую фазу: на ней нам нужно отсортировать циклические подстроки размером 1, т.е. отдельные символы:

```
1  std::vector<int> count(alphabetSize, 0);
2
3  for (int i = 0; i < s.length(); ++i) {
4          ++count[s[i]];
5  }
6
7  for (int i = 1; i < alphabetSize; ++i) {
8          count[i] += count[i - 1];
9  }
10
11 for (int i = 0; i < s.length(); ++i, --count[s[i]]) {
12         p[count[s[i]]] = i;
13 }
```

Теперь нам нужно построить массив классов проходом по массиву p:

```
classes[p[0]] = 0;

for (int i = 1, classIndex = 1; i < s.length(); ++i) {
        if (s[p[i]] != s[p[i - 1]]) {
                ++classIndex;
        }

        classes[p[i]] = classIndex - 1;
}
```

Далее пусть мы находимся на фазе k, выполнив при этом $k-1$ фаз. Заметим, что циклические подстроки размером 2^k состоят из двух подстрок длины 2^{k-1}, которые были обсчитаны на предыдущем шаге, мы можем это использовать. Взглянем на массив classes с предыдущего шага, в нём содержиться информация о подстроках длины 2^{k-1}, необходимая для их сравнения. А именно пары вида (classes[i], classes[i + (1u << (k − 1))]), т.е. элементы по индексу i и $i + 2^{k-1}$. Осталось эти пары отсортировать. Заметим, что подстроки длины 2^{k-1} уже отсортированы на предыдущем шаге в массиве p. Тогда мы можем получить сортировку половинок подстрок длины 2^k с помощью массива p. Достаточко вычесть из каждого элемента массива 2^{k-1}, ведь при переходе к строкам вдвое большей длины эти подстроки становятся их вторыми половинками. После этого останется стабильной цифровой сортировкой отсортировать по первому элементу пары и пересчитать массив classes.

Утверждение. Асимптотика алгоритма — $O(n \log n)$.

Действительно, нам нужно $O(n)$ на каждый шаг, а всего шагов — $O(\log n)$.

3.4 Алгоритм Касаи.

Важно!

В доказательстве корректности этого алгоритма довольно тяжело разобраться, если придерживаться математической строгости. Так что опустим это всё и разберёмся по сути.

Этот алгоритм позволяет вычислять LCP соседних суффиксов в суффиксном массиве за чудесные $O(n)$.

1. Для начала введём нужные обозначения, они будут очень похожи на обозначения с neerc'a, чтобы было проще разобраться потом.
   - S — наша строка.
   - S_i — суффикс строки с i-ого символа.
   - Suf — суффиксный массив. Сразу обговорим, что кроме индекса в элементе массива будет суффикс, соответствующий ему (далее будет показано на примере).

- Suf^{-1} — массив, обратный суффиксному. По сути, это просто массив суффиксов, отсортированный по длине суффикса, сразу покажем на примере:

Пример массивов для строки *abacaba*.

i	$Suf[i]$	$S_{Suf[i]}$
0	6	*a*
1	4	*aba*
2	0	*abacaba*
3	2	*acaba*
4	5	*ba*
5	1	*bacaba*
6	3	*caba*

i	$Suf^{-1}[i]$	S_i
0	2	*abacaba*
1	5	*bacaba*
2	3	*acaba*
3	6	*caba*
4	1	*aba*
5	4	*ba*
6	0	*a*

  На примере показано соответствие между массивами. Рассмотрим его ещё раз. Пусть мы зафиксировали элемент с $i = 1$ из массива Suf. Тогда $Suf[1] = 4$ и $S_{Suf[1]} = S_4 = aba$, эта строка соответствует элементу Suf^{-1} с индексом $j = 4$. Заметим, что $Suf^{-1}[4] = 1$, что равно изначальному взятому $i = 1$. Получается, $Suf^{-1}[i]$ показывает, на каком месте в массиве Suf находится i-ый суффикс.

- $LCP[x, z]$ — длина наибольшего общего префикса строк S_x, S_z. По сути, здесь индексы обозначают индексы в правой табличке из примера сверху. Например, $LCP[0, 2] = LCP[abacaba, acaba] = 1$.

- $lcp[x, z]$ — длина наибольшего общего префикса строк $S_{Suf[x]}, S_{Suf[z]}$. Здесь же индексы аналогично обозначают индексы в левой табличке из примера сверху. Например, $lcp[0, 2] = LCP[a, abacaba] = 1$.

2. Теперь рассмотрим три вспомогательных леммы. Все примеры далее будут использовать массивы для строки *abacaba*, описанные выше. Рекомендуется при разборе лемм держать эти таблички на виду, они пригодятся.

Лемма 3.1. *lcp двух суффиксов — минимум среди всех lcp пар суффиксов между ними в массиве* Suf*. Или же:*

$$lcp[y - 1, y] \geq lcp[x, z], x < y \leq z.$$

Доказательство. Для начала приведём пример. Рассмотрим суффиксы из Suf с $i = 1$ и $i = 3$ их $lcp[1, 3] = lcp[aba, acaba] = 1$. Теперь рассмотрим все пары суффиксов между ними: это пара с индексами $(1, 2)$ и $(2, 3)$, теперь посчитаем их lcp: $lcp[1, 2] = lcp[aba, abacaba] = 2$, $lcp[2, 3] = lcp[abacaba, acaba] = 1$. Их минимум равен 1, что совпадает с $lcp[1, 3]$. Действительно, если lcp каких-либо суффиксов i, j из массива Suf больше lcp пары суффиксов между ними, то существует различающийся символ на позиции первых $lcp[i, j]$ символов, но из-за лексикографического порядка суффиксов это невозможно. ■

Лемма 3.2. *Рассмотрим суффиксы на позициях $x-1$ и x в массиве Suf. Пусть $lcp[x-1,1] > 1$, тогда суффиксы, полученные удалением первого символа, будут в том же лексикографическом порядке, что и изначальные. Простыми словами: если у соседних суффиксов $lcp > 1$, то удаление первого символа не изменит их порядок. Или же:*

$$lcp[x-1,x] > 1 \Rightarrow Suf^{-1}[Suf[x-1]+1] < Suf^{-1}[Suf[x]+1].$$

Доказательство. Для начала приведём пример. Рассмотрим суффиксы из Suf с $i = 1$ и $j = 2$. $lcp[1,2] = lcp[aba, abacaba] = 2 > 1$, так что условие леммы выполняется. Теперь удалим первый символ у суффиксов, получим суффиксы ba и $bacaba$ с индексами 4 и 5 из массива Suf, т.е. порядок сохранился. Доказательство следует из того, что, если суффиксы отсортированы в лексиграфическом порядке при двух совпадающих символах в начале, то и при одном совпадающем символе они останутся отсортированы в лексиграфическом порядке. Осталось разобраться, почему формула в формулировке соответствует тому, что написано в доказательстве. Действительно, значения $Suf[x-1]$ и $Suf[x]$ равны индексам соответствующих суффиксов в массиве Suf^{-1}, прибавление 1 к ним означает «спуск» по массиву Suf^{-1} вниз, что равносильно отсечению символу сначала (смотрите таблички), а обратное отображение Suf^{-1} вернёт позиции урезанных на один символ суффиксов в массиве Suf. ■

Лемма 3.3. *У двух соседних суффиксов в массиве Suf, у которых $lcp > 1$, при отсечении первого символа lcp уменьшается на 1. Или же:*

$$lcp[x-1,x] > 1 \Rightarrow LCP[Suf[x-1]+1, Suf[x]+1] = lcp[x-1,x] - 1.$$

(обращаю внимание, что $LCP(x,y)$ — длина наибольшего общего префикса строк S_x, S_y, все обозначения есть выше).

Доказательство. Думаю, само утверждение не вызывает сомнений, осталось разобраться, почему это выражается такой формулой. Действительно, значения $Suf[x-1]$ и $Suf[x]$ равны индексам соответствующих суффиксов в массиве Suf^{-1}, прибавление 1 к ним означает «спуск» по массиву Suf^{-1} вниз, что равносильно отсечению символу сначала (смотрите таблички), а $LCP[Suf[x-1]+1, Suf[x]+1]$ — формальная запись того, что мы рассматриваем суффиксы по данным индексам из нашего массива Suf^{-1} (взгляните на таблички сверху). ■

3. Докажем ещё одну вспомогательную лемму (на neerc'e она соответствует лемме из пункта вспомогательные утверждения).

   Для начала введём новые обозначения (дублируют обозначения с neerc'a):

   - i — индекс рассматриваемого суффикса в массиве Suf^{-1} (назовём этот суффикс первым).
   - $i-1$ — индекс соседнего с ним суффикса в массиве Suf^{-1} (назовём этот суффикс вторым).
   - $q = Suf^{-1}[i]$ — индекс первого суффикса в массиве Suf.

- $p = Suf^{-1}[i-1]$ — индекс второго суффикса в массиве Suf.
- $j - 1 = Suf[p-1]$ — индекс в массиве Suf^{-1} соседнего с вторым суффиксом в массиве Suf. Действительно, p — индекс второго суффикса в массиве Suf, тогда $p-1$ — индекс соседнего с ним в Suf, а $Suf[p-1]$ — его индекс уже в массиве Suf^{-1}.
- $k = Suf[q-1]$ — аналогично индекс в массиве Suf^{-1} соседнего с первым суффиксом в массиве Suf.

Лемма 3.4. *LCP суффикса и предыдущего суффикса из $Suf \geq LCP$ предшественника этого суффикса из Suf^{-1} и предыдущего суффикса (относительно предшественника) в Suf с удалёнными первыми элементами, если их изначальный $LCP > 1$. Или же:*

$$LSP(j-1, i-1) > 1 \Rightarrow LCP(k, i) \geq LCP(j, i).$$

Доказательство. Будем проводить доказательство паралелльно с примером, чтобы было нагляднее. Пусть $i = 2$, что соответствует суффиксу $acaba$ из массива Suf^{-1}, или же $S_i = acaba$ (смотрите табличку выше). Тогда $i - 1 = 1$, что соответствует $S_{i-1} = bacaba$. $j - 1 = Suf[Suf^{-1}[i-1] - 1] = Suf[5-1] = 5$, что соответствует суффиксу ba, который, как и говорилось, соседний с исходным $bacaba$. Аналогично находим $k = 0$ и соответствующий суффикс $abacaba$. Таким образом нашу лемму для конкретного случая можно записать так:

$$LSP(\underbrace{j-1}_{=ba}, \underbrace{i-1}_{=bacaba}) > 1 \Rightarrow LCP(\underbrace{k}_{=abacaba}, \underbrace{i}_{=acaba}) \geq LCP(\underbrace{j}_{=a}, \underbrace{i}_{=acaba}).$$

Для начала заметим, что из 3.2 и того, что в массиве Suf суффиксы S_{j-1} и S_{i-1} соседние (это следует из определения $j-1$)

$$LCP(\underbrace{S_{j-1}}_{=ba}, \underbrace{S_{i-1}}_{=bacaba}) > 1 \Rightarrow \underbrace{Suf^{-1}[j-1+1]}_{=a} < \underbrace{Suf^{-1}[i-1+1]}_{=acaba}.$$

Теперь заметим, что

$$\underbrace{Suf^{-1}[k]}_{=abacaba} = \underbrace{Suf^{-1}[i]}_{=acaba} - 1.$$

Т.е. мы берём предыдущий суффикс в Suf перед $acaba$. Тогда получаем

$$\underbrace{Suf^{-1}[j]}_{=a} \leq \underbrace{Suf^{-1}[k]}_{=abacaba} < \underbrace{Suf^{-1}[i]}_{=acaba}.$$

И наконец из 3.1

$$LCP(\underbrace{j}_{=a}, \underbrace{i}_{=acaba}) \leq LCP(\underbrace{k}_{=abacaba}, \underbrace{i}_{=acaba}).$$

■

4. Теперь осталось доказать теорему, которая является комбинацией 3.4 и 3.3.

Теорема 3.5. *Формулировка очень похожа на 3.4*

$$LCP[(j-1,i-1)>1 \Rightarrow LCP(k,i) \geq LCP(j-1,i-1)-1.$$

Доказательство. Сначала из 3.2 и условия нашей теоремы

$$LCP(\underbrace{j}_{=a},\underbrace{i}_{=acaba}) = LCP(\underbrace{j-1}_{ba},\underbrace{i-1}_{bacaba})-1.$$

Совмещая с 3.4

$$LCP(\underbrace{k}_{=abacaba},\underbrace{i}_{=acaba}) \geq LCP(\underbrace{j}_{=a},\underbrace{i}_{=acaba}) = LCP(\underbrace{j-1}_{ba},\underbrace{i-1}_{bacaba})-1.$$

■

Теперь мы наконец-то можем перейти к практической реализации. Из теоремы 3.5 мы получаем ограничение снизу на значение LCP от предыдущих вычисленных LCP. Тогда мы можем последовательно рассматривать все суффиксы по порядку от S_1 до S_k, используя при этом LCA, посчитанное ранее. Ниже приведена реализация, основанная на этом неравенстве.

```
1  std::string s;
2  int length = s.length;
3  std::vector<int> sufArr; // суффиксный массив(считаем, что мы его заполнили)
4  std::vector<int> lcp; // то, что мы заполняем
5
6  void countLCP() { //Kasai
7    std::vector<int> pos(length);
8    for (int i = 0; i < length; ++i) {
9      pos[sufArr[i]] = i;
10   }
11   int k = 0;
12   for (int i = 0; i < length; ++i) {
13     if(k > 0) {
14       --k;
15     }
16     if(pos[i] == length - 1){
17       lcp[pos[i]] = -1;
18       k = 0;
19     } else {
20       int j = sufArr[pos[i] + 1];
21       while (std::max(i + k, j + k) < length && s[i + k] == s[j + k])
22         ++k;
23       lcp[pos[i]] = k;
24     }
25   }
26 }
```

Утверждение. Асимптотика данного алгоритма $O(n)$.

Это следует из того, что на каждой итерации значение LCP может уменьшиться не более, чем на единицу (следует из 3.5), тогда всего значение LCP увеличиться не более, чем на $O(2n)$, а каждое увеличение происходит за $O(1)$.

4 Лекция 4: Суффиксное дерево. Алгоритм Укконена.

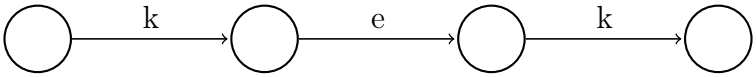
4.1 Суффиксное дерево

Определение 4.1. Суффиксное дерево строки — бор всех её суффиксов.

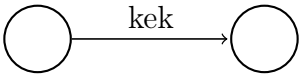
Утверждение. В таком дереве $O(n^2)$ вершин.

Такое количество вершин нас не устраивает, попробуем его уменьшить. Заменим все рёбра без ветвления на «длинные» рёбра, которые будут представлять собой последовательности символов, из которых раньше состояли «несжатые» рёбра. Таким образом, мы получили новые вершины («явные»), а также «неявные» вершины, которые были у нас в несжатом дереве, запомним, что они существуют. Хранить последовательности символов дорого, поэтому будем хранить их как координаты подстроки в изначальной строке.

До «сжатия»



После



Определение 4.2. Сжатое суффиксное дерево — сжатое по описанному выше принципу дерево суффиксов, в каждом ребре которого храним координаты подстроки.

Определение 4.3. Путевая метка вершины — координаты подстроки, соответствующей ребру, которое ведёт в вершину.

Утверждение. В сжатом суффиксном дереве $O(n)$ вершин.

Для доказательства достаточно рассмотреть худший случай, когда строка состоит из разных символов.

Утверждение. Пусть в сжатом дереве n листьев, тогда промежуточных вершин $\leq n - 1$.

Доказательство. Доказательство по индукции. База очевидна, пусть для всех $k \leq n$ выполнено, докажем для $k = n + 1$.

Рассмотрим корень дерева, пусть его степень равна 2 (меньше быть не может, т.к. это сжатое суффиксное дерево). Пусть в левом поддереве l_1 промежуточных вершин, а в правом l_2, тогда промежуточных вершин из предположения индукции будет $\leq (l_1 - 1) + (l_2 - 1) + 1 = l_1 + l_2 - 1 = l - 1$ (добавляем единицу в конце, т.к. сам корень будет промежуточной вершиной. ■

Некоторые свойства сжатого суффиксного дерева:

- Каждая внутренняя вершина имеет не меньше двух детей.
- Каждое ребро помечено непустой подстрокой строки s.
- Никакие два ребра, выходящие из одной вершины, не могут быть помечены строками, начинающимися с одного и того же символа.
- Дерево содержит все суффиксы строки, причём каждый суффикс заканчивается точно в листе.

Рассмотрим подробнее последний пункт. На самом деле, сейчас возможна ситуация, когда суффикс заканчивается не в листе, чтобы избежать такого, изначально добавим к строке сентилен.

Суффиксные ссылки проводим аналогично, прямо как в боре.

Лемма 4.1. *Для любой внутренней вершины существует суффиксная ссылка, ведущая в некоторую внутреннюю вершину.*

Доказательство. Рассмотрим некоторую внутреннюю вершину, пусть её путевая метка (4.3) равна $s[j \dots i]$, вершина внутренняя, значит в ней есть ветвление, но и вершина, соответствующая путевой метке $s[j+1 \dots i]$, также будет внутренней (ведь ничего не изменится от удаления первого символа ветвление не изменится). ■

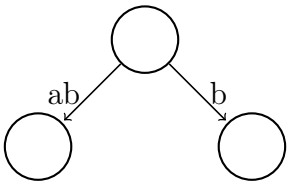
Теперь попробуем построить сжатое суффиксное дерево. Рассмотрим наивный алгоритм построения за $O(n^2)$: для каждого символа строки будем добавлять суффикс, начинающийся с него за $O(n)$.

Теперь подробнее рассмотрим процесс добавления отдельного символа в конец строки.

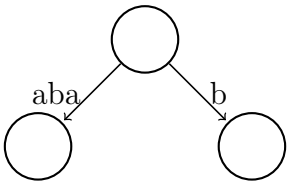
Возможно несколько случаев:

1. Продление листа: пусть у нас суффикс $s[k \dots i-1]$ заканчивается в листе, тогда при добавлении нового символа $s[i]$ достаточно просто продлить ребро, соответствующее листу. В итоге мы получим, что лист останется листом — это называется инвариантностью листа.

До добавления



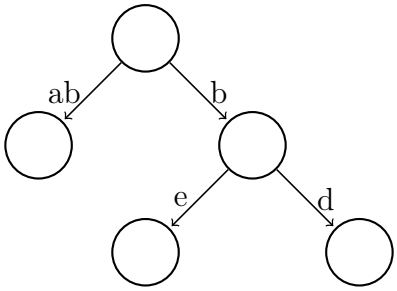
После



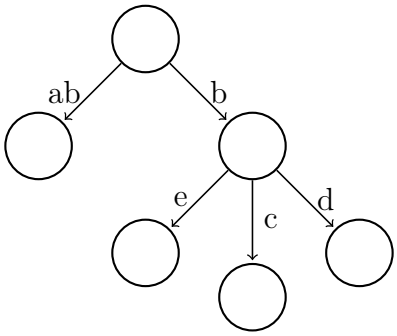
2. Ответвление. Пусть при добавлении нового символа c мы пришли в вершину (явную или неявную), но не лист.

   - Тогда если мы пришли в явную вершину, то нам достаточно добавить лист и соединить этот лист с нашей явной вершиной ребром, соответствующим символу c.

До добавления

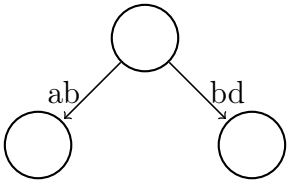


После

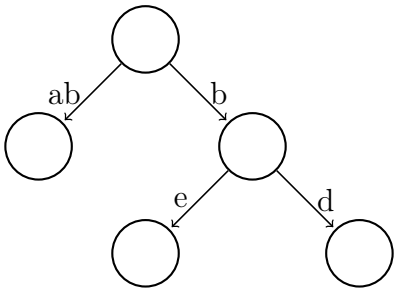


   - Если же мы пришли в неявную вершину, то необходимо сделать эту вершину явной (т.к. от неё появляется ветвление), добавить новый лист, в который ведёт ребро символа c, и соединить с этой вершиной.

До добавления



После



3. Ничего не делать. Такая ситуация возникает, если из суффикса, по которому мы идём, если путь по нашему символу.

Запомним эти переходы и их номера.

4.2 Алгоритм Укконена.

Теперь перейдём ближе к алгоритму. Как говорилось ранее, шаг алгоритма подразумевает добавление символа к каждому суффиксу. Мы уже говорили выше, какие типы добавления бывают. Заметим, что с помощью суффиксных ссылок мы можем легко перейти к суффиксу с длиной меньше на 1. Таким образом, нам достаточно переходить по суффиксным ссылкам до корня и постепенно добавлять к суффиксам наш новый символ.

Заметим, что на очередном шаге типы позиций (3 типа, которые описаны выше) могут изменяться определённым образом, а именно:

- $1 \to 1$: , т.е. лист остаётся листом (про это мы уже разговаривали)
- $2 \to 1$, на картинках видно, что мы всегда добавляем лист
- $3 \to 3, 2$, т.к. переход приведёт к позиции, из которой либо есть переход по следующему символу, либо нет, но там точно не лист

Теперь докажем важную лемму, которая позволит окончательно сформулировать алгоритм.

Лемма 4.2. *Пусть мы находимся на фазе добавления нового символа в некоторый j-ый суффикс. Тогда если мы встретили тип добавления 3, все следующие добавления к меньшим суффиксам будет этого же типа. Т.е. мы можем остановить вставку, как только мы встретили тип вставки 3.*

Доказательство. Если путь с меткой $s[j \dots i-1]$ с типом вставки 3 продолжается символом i, то и путь $s[j+1 \dots i-1]$ будет продолжаться этим же символом i. ■

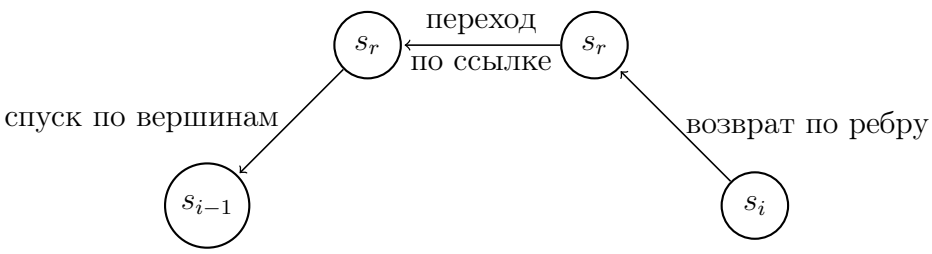
Осталось избавиться от обработки типа 1, действительно, при вставке элемента тип не поменяется, тогда «продлим» листья изначально до бесконечности, что позволит не продлевать на 1 на каждом шаге.

Итоговый алгоритм:

- Обрабатываем один символ
- Вершины типа 1 изначально все продлили до бесконечности
- Обрабатываем вершины типа 2
- Как только обработали вершину типа 3, заканчиваем шаг, запоминаем индекс
- На следующем шаге начинаем обработку с этого индекса, т.к. все вершины до него стали типа 1 или уже были этого типа

Осталось только научиться считать суффиксные ссылки, будем это делать аналогично Ахо-Корасику: поднимемся до ближайшей явной вершины, перейдём по её суффиксной ссылке (она существует по 4.1) и продлим её нашим исходным символом. (спуск после перехода по суффиксной ссылке можно делать не сравнивая все символы ребра, а только первый символ и длину подстрок, это возможно, т.к. суффиксная ссылка приведёт к суффиксу изначальной ветки, а значит там точно будет путь по этим символам).

Пример



В примере нам нужно посчитать суффиксную ссылку для s_i, мы переходим в ближайшую явную вершину s_r, переходим по её суффиксной ссылке, а дальше спускаемся по тем же вершинам (какие содержаться в ребре, по которому мы поднялись).

4.3 Доказательство асимптотики.

Для начала рассмотрим вспомогательные понятия:

Определение 4.4. Глубина вершины — число рёбер на пути от корня до вершины.

Лемма 4.3. *При переходе по суффиксной ссылке глубина уменьшается не более чем на* 1.

Доказательство. Прямо следует из определения суффиксной ссылки. Пусть путь из корня в исходную вершину $A = s[j+1 \dots i]$, а путь в вершину, куда мы приходим по суффиксной ссылке $B = s[j \dots i]$. Они будут различаться на 1 только в случае, когда в первую вершину на пути B ведёт ребро из одного символа. ■

Лемма 4.4. *Число переходов по рёбрам внутри фазы (вставка текущего символа) номер i равно $O(i)$.*

Доказательство. Найдём количество переходов по рёбрам при поиске конца суффикса. Переход во внутреннюю вершину уменьшает глубину на 1, переход по суффиксной ссылке уменьшает глубину не более чем на 1. (4.3) Таким образом, мы уменьшаем глубину не более чем на 2. Так как высота не может увеличиться больше глубины дерева, то суммарно высота не может увеличиться больше, чем на $2i$. Получаем число переходов по рёбрам за одну фазу $O(i)$. ■

Теорема 4.5 (Асимптотика алгоритма Укконена)**.** *Алгоритм Укконена работает за* $O(n)$.

Доказательство. В течение работы алгоритма создаётся не более $O(n)$ вершин (действительно, если у нас у каждой вершины хотя бы два ребёнка, а число листов по определению n, то из индукции по n легко следует это утверждение, основная идея в том, чтобы отрезать листья и сводить к предположению индукции). Все суффиксы, который заканчиваются в листах (тип 1) мы продлеваем за $O(1)$, случаи типа 3 мы вовсе не обрабатываем, а за обработку случаев 2 суммарно будет создано не более $O(n)$ вершин (так как их всего $O(n)$), т.е. амортизационно на каждой фазе будет создано $O(1)$ вершин. Каждую фазу мы начинаем добавление суффикса не с корня, а с индекса, на котором в предыдущей фазе было применено правило 3, то суммарное количество переходов (из 4.4) будет $O(n)$ или $O(1)$ амортизационно. ■

5 Лекция 5: Хеширование строк. Алгоритм Рабина-Карпа.

5.1 Хеширование строк. Вычисление полиномиального хеша методом Горнера.

Научимся хешировать строки. Пусть нам дана строка $s = s_0, s_1, \ldots, s_{n-1}$. Можно использовать следующие хеш-фунции:

- $h_1(s) = (s_0 + s_1 a + s_2 a^2 + \ldots + s_{n-1} a^{n-1}) \bmod M$
- $h_2(s) = (s_0 a^{n-1} + s_1 a^{n-2} + s_2 a^{n-3} + \ldots + s_{n-1}) \bmod M$

Теперь выберем константы a и M. В качестве M удобно брать степени двойки (например, 2^{32}), чтобы взятие остатка было равносильно беззнаковому переполнению.

Наша хеш-функция должна изменяться при изменении одного символа в строке, т.е. чтобы все значения $s \cdot a^u, 0 \le s < M$ были различны. Этого можно добиться, если a и M будут взаимно простыми.

Теорема 5.1. *1. Если a и M не взаимно простые, то*

$$\{s \cdot a \bmod M, 0 \le s < M\} \neq \{0, \ldots, M-1\}.$$

2. Если a и M взаимно простые, то

$$\{s \cdot a \bmod M, 0 \le s < M\} = \{0, \ldots, M-1\}.$$

Доказательство. 1. a и M не являются взаимно простыми, тогда у них есть какой-то общий делитель $d > 1$: $a = d \cdot x, M = d \cdot y$. Тогда:

$$s \cdot a = M \cdot k + r$$
$$r = s \cdot a - M \cdot k$$
$$r = s \cdot d \cdot x - d \cdot y \cdot k$$
$$r = d(s \cdot x - y \cdot k)$$

Получаем требуемое.

2. Доказательство от противного. Пусть это множество $\{s \cdot a \bmod M, 0 \le s < M\}$ имеет меньше M различных элементов. Тогда $\exists i < j : ia \equiv ja (\bmod M)$. Следовательно, $(j-i)a = M \cdot u$, т.е. $j - i$ делится на M (a и M взаимно простые). Но $0 < j - i < M$, получаем противоречие.

■

Метод Горнера для вычисление хеш-функции:

$$h_2(s) = (((s_0 a + s_1)\, a + s_2)\, a + \ldots + s_{n-2})\, a + s_{n-1}.$$

5.2 Алгоритм Рабина-Карпа.

Пусть нам нужно найти подстроку p длины m в тексте t длины n.

Алгоритм очень простой:

1. Считаем хеш шаблона p.
2. Считаем хеш всех окон размера m в тексте t, сдвигаем окно. Этот хеш считает на основе предыдущего.
3. Если получили совпадение хешей, сравниваем строки.

В лучшем случае (хеш шаблона не найден) $O(n + m)$.

Найдено k совпадений хеш-функции, тогда асимптотик $O(n + m + km)$.

5.3 Быстрое вычисление хеша подстроки на базе хеш-значений префиксов.

Пусть нам известны значения хешей для всех префиксов. Т.е. $h(s[0 \ldots r-1]) = s_0 x^{r-1} + s_1 x^{r-2} + \ldots + s_{r-1}$. Научимся быстро считать хеш подстроки, ответ простой:

$$h(s[l \ldots r-1]) = h(s[0 \ldots r-1]) - x^{r-l} \cdot h(s[0 \ldots l-1]) = s_l x^{r-l-1} + \ldots + s_{r-1} \bmod M.$$

6 Лекция 6: Вычислительная геометрия. Выпуклая оболочка 2D.

6.1 Геометрические понятия.

Тут ничего нового, думаю, все и так всё знают. Обратим внимание читателя на две идеи:

- Сравнивать вещественные числа нужно с использованием константы «эпсилон»
- Проверку знака угла поворота необходимо осуществлять без всяких тригонометрических функций для уменьшения погрешности. Можно воспользоваться векторным произведением.

6.2 Выпуклая оболочка 2D.

Определение 6.1. Выпуклое множество — множество M, в котором выполняется следующее свойство:

$$\forall P, Q \in M : PQ \in M.$$

Определение 6.2. Выпуклая оболочка множества — пересечение всех выпуклых множеств, содержащих все заданные точки.

Утверждение. В каждой граничной точке выпуклого множества существует касательная.

Определение 6.3. Изохрона — множество точек, достижимых из данной за фиксированное время.

6.3 Алгоритм Джарвиса.

Или же «заворачивание подарка» — ищем выпуклую оболочку последовательно, против часовой стрелки, начиная с определённой точки.

Приведём описание алгоритма:

1. Возьмём самую нижнюю левую точку p_0 нашего множества.
2. На каждом следующем шаге для последней добавленной точки p_i ищем p_{i+1} среди всех недобавленных точек и p_0 с максимальным полярным углом (при равенстве сравниваем по расстоянию), добавляем её. Если добавили p_0, заканчиваем алгоритм.

Утверждение. Корректность следует из построения, т.к. на каждом шаге алгоритма мы получаем прямую, относительно которой все точки множества лежат слева.

Утверждение. Асимптотика алгоритм $O(n \cdot h)$, где h — размер выпуклой оболочки. Таким образом, в худшем случае $O(n^2)$.

6.4 Алгоритм Грэхема.

Ищем точки последовательно, используя стек.

Приведём описание алгоритма:

1. Возьмём самую нижнюю левую точку p_0 нашего множества.
2. Сортирует остальные точки можества по полярному углу.
3. Добавляем в ответ (в стек) p_1 — самую первую из отсортированных точек.

4. Берём следующую точку по счёту. Пока новая точка образует неправый поворот с предыдущими двумя, удаляем последнюю точку из стека.
5. Добавляем эту следующую точку в ответ (в стек) и повторяем с пункта 3, пока не дойдём до начальной.

Рассмотрим доказательство корректности алгоритма:

Доказательство. Докажем по индукции. База из трёх точек очевидна.

Пусть для $i-1$ точек истинная оболочка совпадает с построенной. Теперь рассмотрим истинную оболочку $ch(S \cup i) = ch(S) \cup i \setminus P$, где P — множество всех точек, видимых из i (это означает, что мы можем провести из i отрезок до точки так, чтобы он не пересекал уже построенную оболочку). Мы строим оболочку против часовой стрелки, а i-ая точка точно лежит в оболочке (мы так предположили в начале доказательства перехода), значит P состоит из последних подряд идущих точках, которые мы добавили на предыдущих шагах, а именно их мы удаляем на текущем шаге. Переход доказан. ■

Утверждение. Асимптотика алгоритма $O(n \log n)$. Обход за $O(n)$ (каждая точка добавляется в ответ не более одного раза). Сортировка занимает $O(n \log n)$.

6.5 Разделяй и властвуй.

1. Делим изначальное множество на примерно равные множества.
2. Строим для каждого множества оболочку рекурсивно (или можем запустит Грехема).
3. Дальше сливаем полученные оболочки.

Утверждение. Если мы умеем сливать оболочки за $O(n)$, то асимптотика алгоритма $O(n \log n)$ по мастер-теореме из

$$T(n) \leq C \cdot n + 2T\left(\frac{n}{2}\right).$$

Приведём пример алгоритма для слияния двух оболочек за $O(n)$.

1. Берём у одной оболочки точку центра масс за опорную.
2. Если центр масс оказался внутри второй оболочки, задача сводится к слиянию двух сортированных списков
3. Иначе проводим касательные к второй оболочке, выкидываем точки, которые попали между касательных, сливаем оставшиеся точки.

7 Лекция 7: Выпуклая оболочка 3D.

7.1 Заворачивание подарка.

Некоторые предварительные сведения, которые будут относиться ко всем способам построения оболочки:

1. Никакие три точки не лежат на одной прямой.
2. Никакие четыре точки не лежат в одной плоскости.
3. Точек как минимум 4.

Также по теореме Эйлера имеем:

$$V - E + F = 2,$$

где V— число вершин, E— рёбер, F— граней.

К тому же $3F \leq 2E \Rightarrow E + 6 \leq 3V$. Получили, что в выпуклой оболочке число граней и рёбер равно $O(V)$. Это в дальшейшем пригодится в доказательствах.

Теперь опишем непосредственно алгоритм (пусть на вход подаются координаты точек (x, y, z):

1. Выбираем начальную точку P с минимальной координатой z (если таких несколько, можно взять любую) и добавляем в оболочку.
2. Выбираем следующую точку Q так, чтобы угол PQ с плоскостью Oxy был минимальным, берём её в оболочку. Ребро PQ принадлежит оболочке.
3. Выбираем следующую точку R так, чтобы угол между PQR и Oxy был минимальным, берём её в оболочку. Грань PQR принадлежит оболочке. Произошло заворачивание подарка.
4. Каждое ребро полученной грани добавляем в очередь на обработку. Для каждого ребра из этой очережи нужно найти смежную грань выпуклой оболочки (аналогично ищем третью вершину грани так, чтобы угол между гранью, в которой изначально находится это ребро, и гранью, образованной ребром и новой точкой, был минимальный). Новая грань может добавить два ребра и закрывает хотя бы одно.
5. Таким образом закрываем все рёбра из очереди гранями оболочки. Завершаем, когда закрыто последнее ребро.

Утверждение. Данный алгоритм работает за $O(n^2)$. Каждый шаг, включая первый, отрабатывает за $O(n)$, всего шагов $O(n)$ (размер оболочки).

7.2 Видимые грани

Описание алгоритма:

1. Собираем тетраэдр из любых четырёх точек исходного множества.
2. Для каждой нерассмотренной точки перебираем все построенные грани выпуклой оболочки. Оставляем невидимые из точки грани. Удаляем видимые грани. (про видимость поговорим потом)
3. Добавляем в оболочку новые грани, которые содержат рассматриваемую точку и рёбра на пересечении видимой и невидимой части граней.

Для проверки видимости точки каждой грани храним в фиксированном порядке обхода относительно нормали к грани. Тогда видимость означает сонаправленность нормали к

грани и вектора от рассматриваемой точки к грани (т.е. скалярное произведенеие больше 0). Чтобы лучше понять эту часть, можно представить себе виток с током, тогда направление тока задаёт обход точек грани, а вектор магнитной индукции — нормаль к плоскости.

Утверждение. Асимптотика алгоритма $O(n^2)$. Каждый шаг за $O(n)$, всего шагов $O(n)$.

7.3 Разделяй и властвуй

Описание алгоритма:

1. Разделяем точки плоскостью на примерно равное число точек.
2. Рекурсивно строим каждую половинку. Если количество точек меньше 7, строим за $O(n^2)$.
3. Объединяем выпуклые оболочки в одну.

Процесс объединение за $O(n)$.

1. Проецируем на плоскость оболочки, находим между ними мост. Для этого можно идти по трём вершинам одной оболочки и смотреть на поведение, например, псевдовекторного произведения этих трёх точек и нижней точки второй оболочки.
2. Сливаем две оболочки от моста, найденного на предыдущем шаге, не учитывая вершины, выкинутые на предыдущем шаге.

Утверждение. Сложность алгоритма $O(n \log n)$. Но объединение не гарантированно за $O(n)$ (из лекции есть сложные случаи).

8 Лекция 8: Сумма Минковского. Scan line.

8.1 Сумма Минковского.

Определение 8.1. Суммой Минковского двух множеств $S_1 \subset R^2, S_2 \subset R^2$ называется множество $S_1 \oplus S_2 \colon \{p + q \colon p \in S_1, q \in S_2\}$.

Определение 8.2. Отрицанием множества $S \subset R^2$ называется множество $-S \colon \{-p \colon p \in S\}$.

Теорема 8.1. *Сумма Минковского двух выпуклых многоугольников есть выпуклый многоугольник, количество вершин которого не превосходит сумму вершин двух исходных многоугольников.*

Доказательство. Любая крайняя точка в направлении какого-то вектора (это значит что мы проектируем нашу фигуру на вектор и выбираем самую крайнюю точку) складывается из крайних точек каждой фигуры в направлении этого же вектора. Так что каждое ребро из суммы Минковского мы можем сопоставить одному ребру исходной фигуры. Как же это сделать? Выберем произвольное ребро α из суммы Минковского. Построим к нему

нормаль и заметим, что это ребро, очевидно, крайнее относительно этой нормали, а значит, оно образовано двумя крайними точками исходных множеств. Это означает, что хотя бы у одной из фигур должно быть ребро, которое является крайним в направлении этой нормали. Сопоставим это ребро рассматриваемому ребру из суммы Минковского. ■

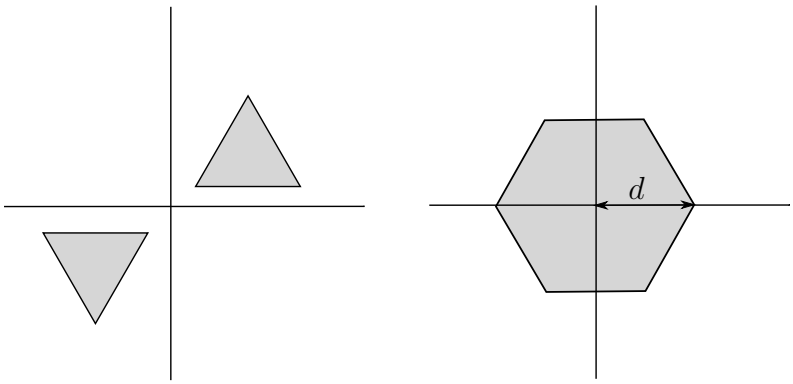
Описание алгоритма:

1. Сортируем грани каждого множества по полярному углу
2. Начинаем обход с крайней левой точки в одну сторону, например
3. Сливаем два массива сортированных массива рёбер

Утверждение. Время работы алгоритма $O(n+m)$, где n и m — число граней в исходных множествах.

Использование суммы Минковского:

- Поиск диаметра множества
  1. Ищем выпуклую оболочку S
  2. Считаем сумму Минковского $S \oplus -S$
  3. Находим самую дальнюю точку от нуля в получившейся сумме



Поиск диаметра

- Проверка пересечения выпуклых многоугольников
  1. Находим $S = P \oplus -Q$
  2. Многоугольники будут пересекаться, если $(0,0) \in S$

8.2 Проверка принадлежности точки многоугольнику.

Если мы пустим из точки луч, то чётность числа пересечений этого луча покажет, лежит точка внутри многоугольника или нет. Если число пересечений нечётно, точка лежит внутри многоугольника.

Осталось обработать некоторые случаи, пусть мы проверяем принадлежность точки q, тогда итоговый алгоритм:

1. Проходим по каждому ребру ab многоугольника
2. Если $q \in ab$, то сразу возвращаем $true$
3. Если $a_y = b_y$, то пропускаем этот отрезок, он не влияет на чётность пересечений
4. Если $q_y = max(a_y, b_y)$ и $q_x < min(a_x, b_x)$, то увеличиваем счётчик пересечений
5. Если $q_y = min(a_y, b_y)$, пропускаем это ребро
6. Если q_y лежит между a_y и b_y и поворот точек a, b, q левый, то увеличиваем счётчик пересечений

По сути, с помощью этого алгоритма мы учитываем только верхнюю точку какого-то ребра (чтобы избежать повторного подсчёта какого-то ребра).

8.3 Сканирующая прямая.

Описание алгоритма: Будем двигать вертикальную сканирующую прямую слева направо с $-\infty$ до $+\infty$. Для каждого отрезка в какой-то момент времени его точка появится на прямой, затем с движением может сдвинуться, а потом и вовсе пропасть. Для пересекающихся отрезков заметим, что их точки на сканирующей прямой в какой-то момент совпадут. Будем рассматривать отрезки, отсортированные по y.

Тогда справедливы следующие утверждения:

- Для поиска пересекающейся пары достаточно рассматривать при каждом положении сканирующей прямой только соседние отрезки, ведь два непересекающихся отрезка никогда не меняют своего относительного положения и два пересекающихся отрезка в момент точки своего пересечения окажутся соседями друг друга в списке.
- Достаточно рассматривать сканирующую прямую только в позициях, когда появляются или удаляются отрезки, ведь относительный порядок отрезка при вставке в наш список не меняется (вспомним про то, что непересекающиеся отрезки не меняют своего относительного порядка).
- При появлении нового отрезка достаточно вставить его на нужное место в списке и проверить его на пересечение с соседними отрезками с списке (это также следует из того, что, если отрезки пересекаются, то они будут соседями в списке).
- При исчезновении отрезка нужно также проверить на пересечение только сосдение отрезки (из предыдущего утверждения).
- Чтобы не пропустить пересечение отрезков по вершине (и корректно обработать вертикальные отрезки), необходимо сначала производить все вставки в список при появлении отрезка, а потом уже удаления.

Утверждение. Асимптотика алгоритма $O(n \log n)$. Всего за алгоритм будет $O(n)$ операций со списком (для которого мы используем `std::set`), тем самым в итоге получаем $O(n \log n)$.

9 Лекция 9: Триангуляция Делоне. ЕМОД.

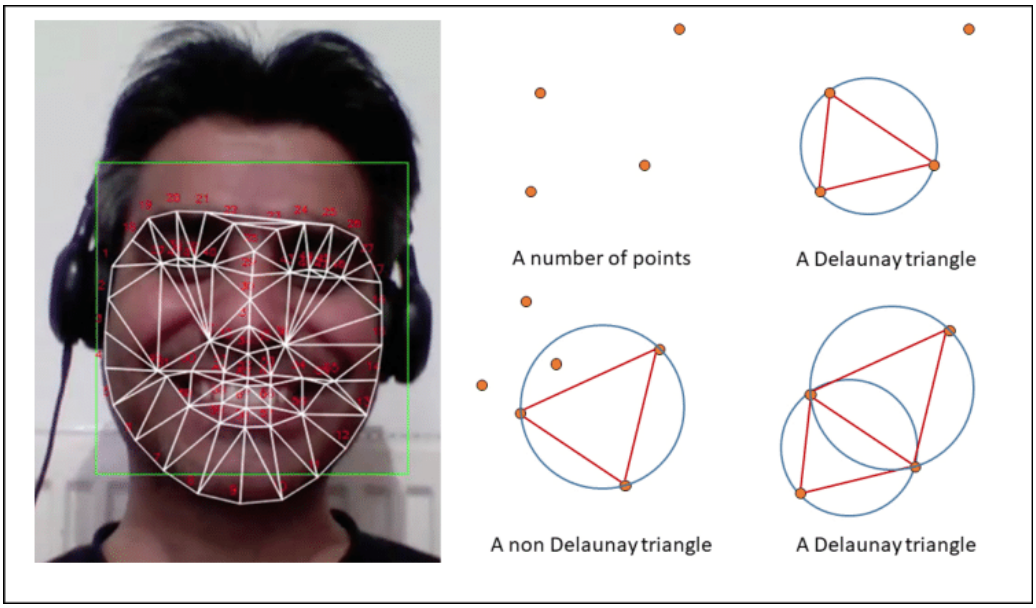
9.1 Триангуляция Делоне.

Определение 9.1. Триангуляция — планарный граф, все внутренние области которого являются треугольниками.

Определение 9.2. Выпуклая триангуляция — триангуляция, для которой минимальный многоугольник, охватывающий все треугольники, будет выпуклым.

Определение 9.3. Триангуляция удовлетворяет условию Делоне, если внутри окружности, описанной вокруг любого построенного треугольника, не попадает ни одна из заданных точек триангуляции.

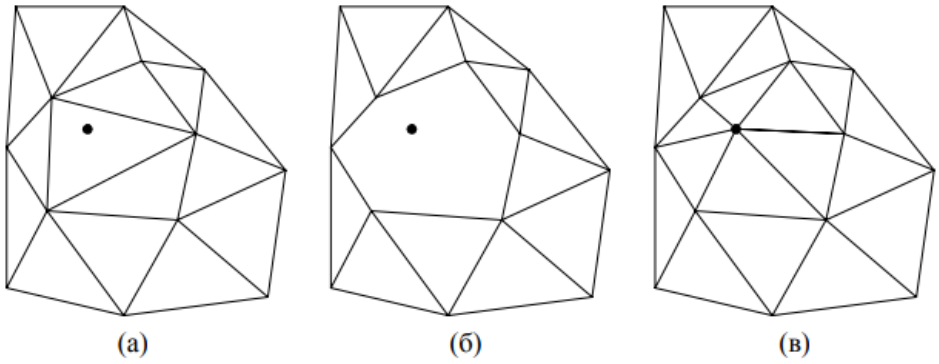
Определение 9.4. Триангуляция Делоне — выпуклая триангуляция, которая удовлетворяет условию Делоне.



Пример триангуляций

Существует множество алгоритмов построения триангуляции Делоне за $O(n \log n)$ и $O(n^2)$. Приведем пример алгоритма «Удаляй и строй». Построение происходит итеративно, опишем процесс вставки очередной вершины:

1. Находим треугольники, в описанные окружности которых входит новая точка (это можно делать, например, задачей локализации точки, а потом проверкой смежных треугольников)
2. Удаляем эти треугольники, образуется многоугольник с точкой внутри (случай б на рисунке)
3. Соединяем новую точку с вершинами образованного многоугольника



«Удаляй и строй»

Утверждение. Асимптотика такого алгоритма $O(n^2)$.

Доказательство. Первый пункт алгоритма (поиск треугольника) можно делать обходом всех треугольников за $O(n)$, нахождение треугольников для удаление также за $O(n)$, итого получаем $O(n^2)$. ■

9.2 Связь триангуляции с выпуклой оболочкой.

Спроектируем точки плоскости на параболоид так, что

$$(x, y) \to (X, Y, Z), X = x, Y = y, Z = x^2 + y^2.$$

Заметим, что в таком случае проекция триангуляции Делоне на параболоид является её нижняя выпуклая оболочка. Проведём плоскость через три точки триангуляции Делоне на параболоиде. Плоскость сечёт параболоид по эллипсу, который в проекции на плоскость даст окружность. По свойству триангуляции Делоне внутри этой окружности нет других точек множества. Следовательно на параболоиде все точки множества лежат по одну сторону от плоскости трёх спроектированных точек. Поясним последний переход подробнее.

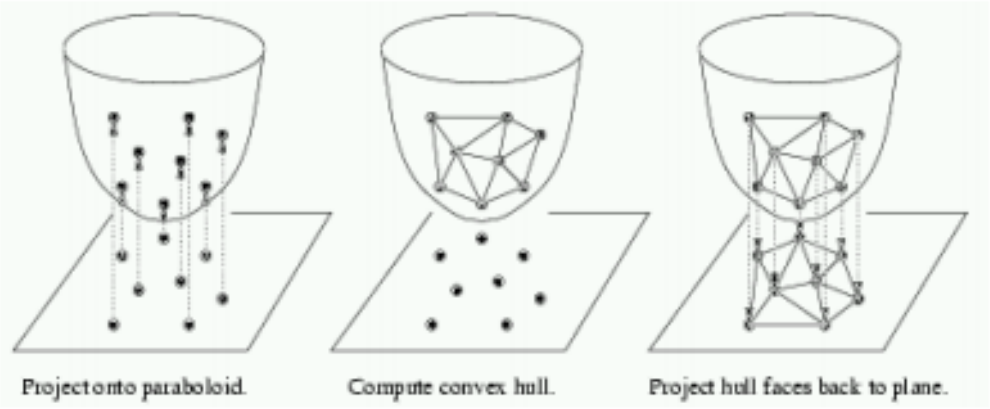
Пусть рассматриваемая окружность имеет уравнение:

$$x^2 + y^2 + ax + by + c = 0.$$

При переходе к параболоиду с заменой $X = x, Y = y, Z = x^2 + y^2$ получаем:

$$aX + bY + Z + c = 0.$$

Что соответствует уравнению плоскости, которое как раз определяет относительное положение других точек и плоскости.



Связь триангуляции Делоне и выпуклой оболочки

9.3 ЕМОД.

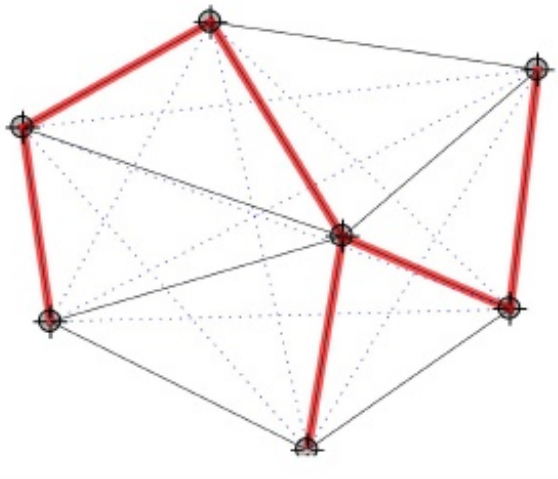
Пусть на плоскости задано множество из n точек.

Определение 9.5. Евклидово минимальное остовное дерево (ЕМОД) — минимальное остовное дерево в полном взвешенном графе на данном множестве точек, где вес ребра — евклидово расстояние между точками.

Наивное построение с помощью обычных алгоритмов построения MST (Прим, Крускал, Борувка) за $O(n^2 \log n)$, т.к. граф полный ($E = O(n^2)$).

Построим ЕМОД с помощью триангуляции Делоне:

1. Строим триангуляцию Делоне за $O(n \log n)$
2. Находим MST на построенной триангуляции обычным алгоритмом (Прим, Крускал, Борувка) за $O(n \log n)$ — получаем ЕМОД за чудесные $O(n \log n)$.



Пример нахождения EMOД с помощью триангуляции

Осталось доказать, что любое EMOД содержит только рёбра триангуляции и никакие другие. (очень мутная теорема)

Теорема 9.1. *Любое ребро, не входящее в триангуляцию Делоне, не содержится ни в каком EMST.*

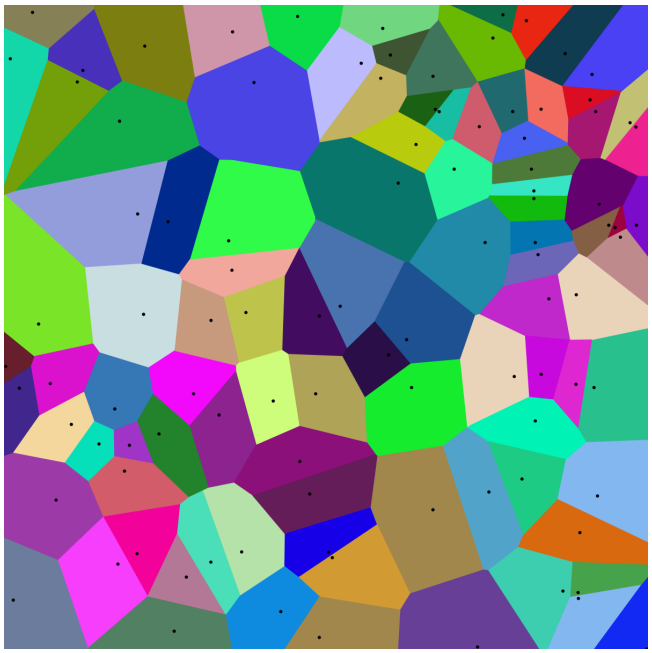
Доказательство. Рассмотрим ребро e между точками p и q, которое не является ребром триангуляции Делоне. Верно следующее утверждение: если существует цикл с двумя входными точками на границе, который не содержит других входных точек, отрезок между этими точками является ребром любой триангуляции Делоне. Из этого утверждения вытекает, что цикл C с e в качестве диаметра должен содержать некоторую другую точку r внутри. Но тогда r ближе к p и q, чем они по отношению друг к другу. Значит pq является самым длинным ребром в цикле $p \to q \to r \to p$, а значит, что не принадлежит никакому $EMST$. ■

10 Лекция 10: Диаграмма Вороного.

10.1 Диграмма Вороного.

Пусть на плоскости задано множество S, содержащее N точек $p_1, p_2, \ldots, p_n$.

Определение 10.1. Диаграмма Вороного — разбиение $\mathbb{R}^2$ на геометрические места точек $V_1, V_2, \ldots, V_n$, в каждом из которых p_i — ближайшая точка.

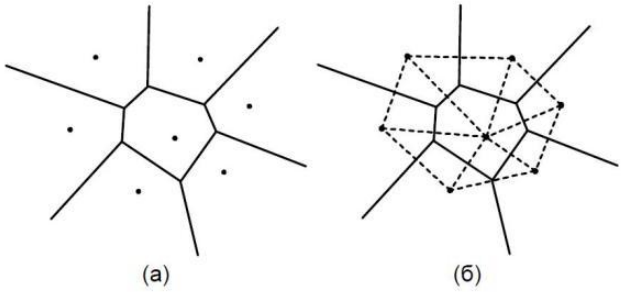


Пример диаграммы Вороного

Определение 10.2. Многоугольник Вороного — многоугольник, образованный пересечением полуплоскостей точек, более близких к p_i, нежели чем к $p_j \in A$, где A — множество всех точек. Или же клеточка диаграммы Вороного.

10.2 Связь с триангуляцией Делоне.

Теорема 10.1. *Ребро x_ix_j принадлежит некоторому треугольнику триангуляции Делоне $\Leftrightarrow$ некоторый отрезок серединного перпендикуляра к x_ix_j является ребром диаграммы Вороного.*



Связь диаграммы Вороного и триангуляции Делоне

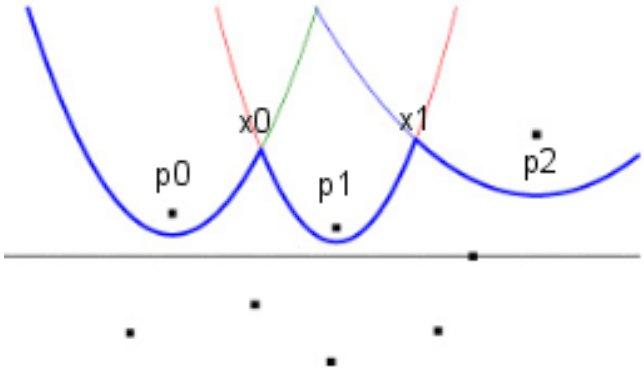
10.3 Построение диаграммы Вороного через оболочку.

1. Проектируем точки на параболоид
2. Строим 3D выпуклую оболочку, получаем триангуляцию Делоне

3. Из триангуляции получаем диаграмму Вороного

10.4 Алгоритм Форчуна.

Данный алгоритм поддерживает заметающую прямую (по аналогии со ScanLine), а также береговую линию, которая состоит из различных парабол и кусков парабол. Пусть все точки слева от заметающей прямой обработаны, тогда береговая линия отделяен порцию плоскости, внутри которой диаграмма Вороного может быть известна, независимо от других точек справа.



Пример для понимания береговой линии (только здесь она горизонтальная)

Для каждой точки слева от заметающей прямой можно определить параболу, которая равноудалена как от этой точки, так и от заметающей прямой. Тогда заметающая прямая будет директрисой для параболы, а точка — фокус. По мере движения прямой, вершины береговой линии в которых две параболы пересекаются, вычерчивают рёбра диграммы Вороного.

При движении прямой мы заносим в очередь с приоритетом события, которые могли бы изменить структуру береговой линии (например, добавление новой параболы, т.е. появление новой точки, а также пересечение серединных перпендикуляров — исчезновение одной параболы), а также храним структуру береговой линни в дереве. Алгоритм состоит из последовательного удаления события из очереди с приоритетом, нахождения изменений событий в береговой линии и обновления структуры данных.

Утверждение. Асимптотика алгоритма $O(n \log n)$. Всего у нас $O(n)$ событий для обработки (каждое будет ассоциировано с некоторым свойством диаграммы Вороного) и на каждое событие $O(\log n)$ времени для обработки.

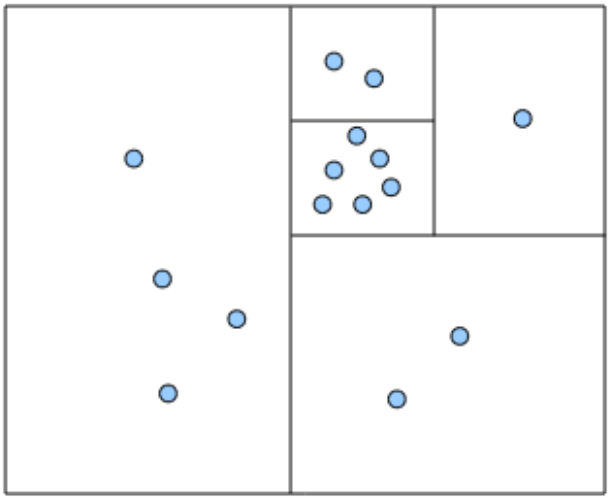
11 Лекция 11: Индексирование гео.

11.1 Задачи поиска геометрии.

- Множество точек. Поиск точек, находящихся в прямоугольнике.
- Множество точек. Поиск ближайшей точки множества.

- Множество полигонов. Поиск полигонов множества, содержащих заданную точку.
- Множество полигонов. Поиск ближайшего полигона множества.

11.2 kd-дерево

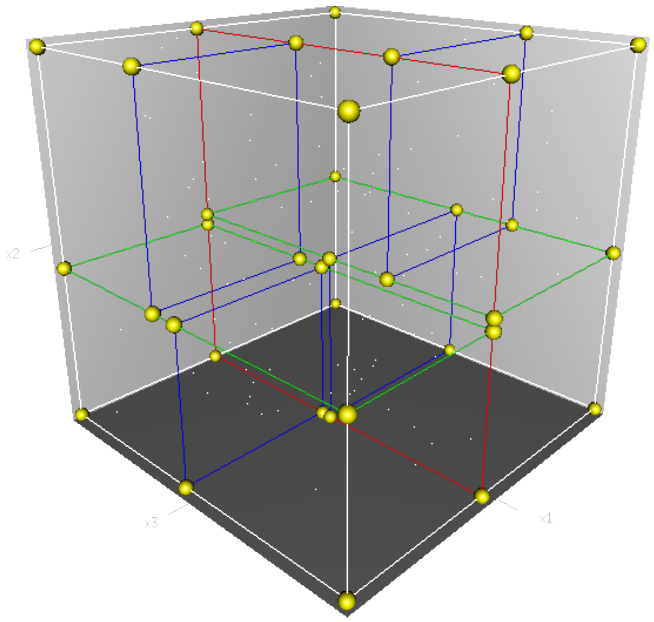


Пример разбиения плоскости (каждое множество можно рассматривал как поддерево $k-d$ дерева)

Определение 11.1. K-d дерево — статическая структура данных для хранения точек в k-мерном пространстве. Позволяет отвечать на запросы, какие точки лежат в данном прямоугольнике. Каждое поддерево содержит некоторое множество точек. В поддереве точки разбиты три множества: точки левее, точки правее и медиана (сам узел поддерева).

Рассмотрим алгоритм построения:

1. Разбиваем все точки вертикальной прямой на две примерно равных группы и получим правого и левого ребёнка
2. Далее строим поддеревья для левого и правого ребёнка, но теперь уже разбиваем горизонтальной прямой
3. На следующем уровне разбиваем снова вертикальными прямыми (или же, в случае больших размерностей разбиваем прямой, параллельной следующей оси координат)
4. Продолжаем построение рекурсивно (окончание рекурсии, когда в поддереве оказывается одна вершина)



Пример трёхмерного $k-d$ дерева

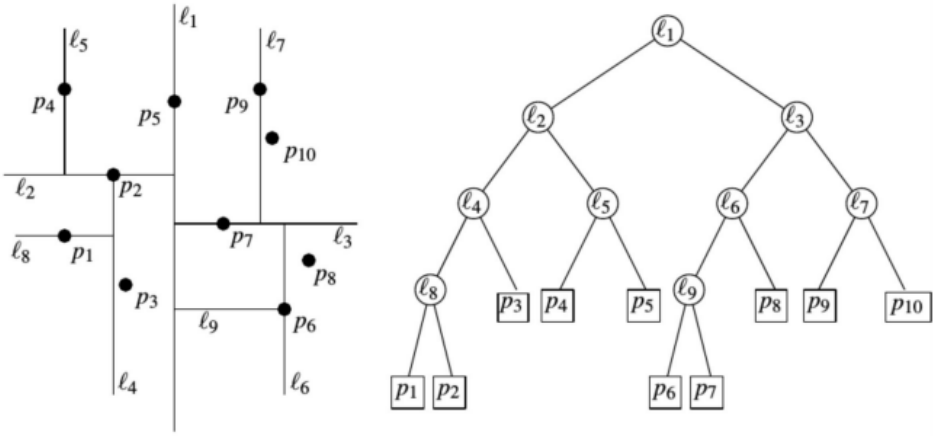
Возможная проблема такого разбиения: большое количество точек с одной координатой, в таком случае можно сравнивать лексиграфически (по нескольким координатам).

Лемма 11.1. *Данный алгоритм построения работает за* $O(n \log n)$.

Доказательство. Напишем уравнение для рекурсии:

$$T(n) = O(1), n = 1$$
$$T(n) = O(n) + 2 \cdot T\left(\frac{n}{2}\right)$$

По мастер-теореме о рекурсии получаем асимптотику $O(n \log n)$. ■



Ещё один пример $k-d$ дерева

Пусть `region(V)` — функция, которая возвращает по вершине область, за которую отвечает (её границы могут быть на бесконечности). Её можно считать при построении $k-d$ дерева или же вычислять при поиске.

Опишем рекурсивный процесс поиска точек в прямоугольнике R из вершины V с детьми $V.left$ и $V.right$ соответственно (изначально $V = root$):

1. Если V — лист, останавливаем процесс, возвращем все вершины из V
2. Если `region(V.left)` $\subset R$, возвращаем $V.left$, если же `region(V.left)` пересекает R, рекурсивно запускаем поиск от $V.left$
3. Аналогично для правого ребёнка

Теорема 11.2 (О времени на запрос)**.** *Перечисление точек в прямоугольнике выполняется за $O(\sqrt{n} + ans)$, где ans — размер ответа.*

Доказательство. Нам необходимо оценить число рекурсивных вызовов (добавка ans возникает, когда мы заканчиваем рекурсию). Заметим, что рекурсивные вызовы выполняются только для тех вершин, регионы которых пересекают R, но не содержатся в нём. Такие регионы обязательно будут пересекать одну сторону хотя бы одну сторону R. Таким образом сведём задачу к оценки количества регионов, которые могут пересекаться, например, произвольной вертикальной прямой. Обозначим как $Q(n)$ максимально возможное количество регионов, пересекаемых какой-либо вертикальной прямой в дереве.

Рассмотрим произвольную вертикальную прямую, она будет пересекать регион корня и какого-то одного из его детей (пусть левого), но не будет пересекать регион другого ребёнка. Левый ребёнок разбит горизонтальной прямой на две части, в каждой примерно по $\frac{n}{4}$ вершин, причём эти части будут делиться также вертикальной прямой. Получаем рекурсивное соотношение:

$$Q(n) = 2 + 2 \cdot Q\left(\frac{n}{4}\right)$$
$$Q(n) = O(1), n = 1$$

Глубина дерева же:

$$log_4 n = \frac{1}{2}\log_2 n.$$

Из мастер-теоремы получаем:

$$Q(n) = O(2^{\frac{1}{2}\log_2 n}) = O(\sqrt{n}).$$

■

Кроме точек такие $k-d$ деревья можно строить для многоугольников.

11.3 Гео-хеш.

Определение 11.2. Гео-хеш — функция, которая преобразует точку на карте в строку.

Алгоритм разбиения Земли:

1. Вся Земля разбивается на 32 прямоугольника. 8 по долготе, 4 по широте.
2. Каждый прямоугольник разбивается на 32. 4 по долготе, 8 по широте.
3. Каждый прямоугольник разбивается на 32 прямоугольника. 8 по долготе, 4 по широте.
4. ...
5. Каждый из полученных прямоугольников обозначается символом из определённого алфавита.

Утверждение. Если представить каждый символ гео-хеша побитово, то чётные биты — широта, нечётные — долгота.

Вычисление гео-хеша по точке:

1. Координаты преобразуются из диапазонов $[-180, 180)$ и $[-90, 90)$ в целые числа с нужной битностью (её можно вычислить по нужной длине гео-хеша)
2. Сливаем биты широты и долготы по очереди
3. Конвертируем в нужный алфавит (например, Base-32)

Утверждение. На основе гео-хешей можно строить различные хеш-таблицы для хранения геометрических объектов.

12 Лекция 12: Длинная арифметика.

12.1 Основные понятия.

Определение 12.1. Длинная арифметика — набор алгоритмов для поразрядной работы с числами произвольной длины.

Утверждение. Для хранения больших чисел можно использовать массив цифр по по некоторому основанию c. Знак числа можно хранить отдельным флагом.

$$A = a_0 + a_1 \cdot c + a_2 \cdot c + \ldots + a_{n-1} \cdot c^{n-1}.$$

Также для удобства можно считать, что у нас нет лидирующих нулей и перед всеми алгоритмами делать нормировку наших чисел (откидывать лидирующие нули, если они появляются).

Утверждение. Пусть длина двух больших чисел m и n, соответственно. «Школьное» вычитание и сложение будет работать за $O(\max(n, m))$. Умножение же будет работать за $O(nm)$.

12.2 Алгоритм Карацубы.

Пусть у нас есть два n-значных числа A и B в десятичной системе счисления,

$$A = ax + b, B = cx + d,$$

где $x = 10^{\frac{n}{2}}$ (примем, что n — чётное).

Получается, в a мы имеем $\frac{n}{2}$ старших разрядов, в b — младших. Распишем произведение:

$$AB = (ax + b) \cdot (cx + d) = acx^2 + (ad + bc) \cdot x + bd.$$

Заменим сумму произведений:

$$(ad + bc) = (a + b) \cdot (c + d) - ac - bd.$$

Получим:

$$AB = acx^2 + ((a + b) \cdot (c + d) - ac - bd) \cdot x + bd.$$

Итого мы свели наше умножение в задачу умножения ac, bd, $(a + b) \cdot (c + d)$, а также сложения, которое выполняется за $\Theta(n)$. Это большие числа, но их разрядность в два раза меньше и их можно посчитать рекурсивно. Размер задачи уменьшился в два раза, количество подзадач равно трём, значит по мастер-теореме имеем следующую асимптотику:

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n) = \Theta\left(n^{\log_2^3}\right).$$

12.3 Деление.

Самый простой вариант: бинарный поиск по ответу, но асимптотика тут оставляет желать лучшего.

Более эффективным вариантом является «школьное» деление в столбик двух многочленов. Асимптотика тут уже лучше: $O(n^2)$. Далее приведена формализация этого деления из Кнута.

Пусть у нас есть два многочлена $u(x)$ и $v(x)$. Мы можем представить их в таком виде:

$$u(x) = q(x) \cdot v(x) + r(x),$$

где $deg(r) < deg(v)$.

Утверждение. Вышеизложенное разложение единственно.

Доказательство. Пусть это не так и уравнению удовлетворяют две разные пары многочленов $(q_1(x), r_1(x)), (q_2(x), r_2(x))$. Тогда справедливы следующие равенства:

$$q_1(x) \cdot v(x) + r_1(x) = q_2(x) \cdot v(x) + r_2(x)$$
$$(q_1(x) - q_2(x)) \cdot v(x) = r_2(x) - r_1(x)$$

Из предположения $q_1(x) \neq q_2(x) \Rightarrow q_1(x) - q_2(x) \neq 0 \Rightarrow$

$$deg((q_1 - q_2) \cdot v) = deg(q_1 - q_2) + deg(v) \geq deg(v) > deg(r_2 - r_1)$$

Пришли к противоречию, т. к. $deg((q_1 - q_2) \cdot v)$ не может быть больше $deg(r_2 - r_1)$. ■

Пусть нам даны два многочлена вида:

$$u(x) = u_m x^m + \ldots + u_1 x + u_0$$
$$v(x) = v_n x^n + \ldots + v_1 x + v_0$$

Результатом алгоритма будут многочлены из соотношения выше:

$$q(x) = q_{m-n} x^{m-n} + \ldots + q_0$$
$$r(x) = r_{n-1} x^{n-1} + \ldots + r_0$$

Далее буквально изложен алгоритм деления многочленов в столбик, он итеративный по k, для простоты понимания напишем, за что отвечают индексы:

- k — позиция текущего разряда в частном, пробегает значения от $m - n$ до 0.
- j — позиция разрядов многочлена u, из которых необходимо вычесть на текущем разряде, чтобы получить остаток, пробегает значения от $n + k - 1$ до k.

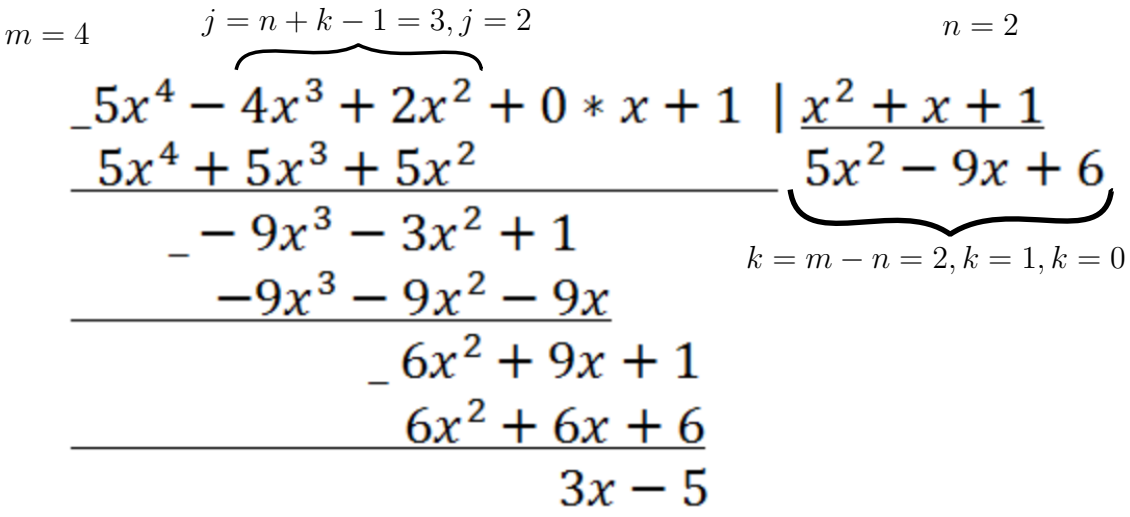


Иллюстрация коэффициентов

Далее сам алгоритм:

1. Итерируемся по $k = m - n, m - n - 1, \ldots, 0$.
2. На каждом шаге вычисляем $q_k = \frac{u_{n+k}}{v_n}$. И пересчитываем остаток: $u_j = u_j - q_k \cdot v_{j-k}$ для $j = n + k - 1, n + k - 2, \ldots, k$.

13 Лекция 13: Преобразование Фурье. FFT.

13.1 Дискретное преобразование Фурье.

Пусть у нас есть некоторый набор коэффициентов (значений) $x_i, i = 0 \ldots N - 1$

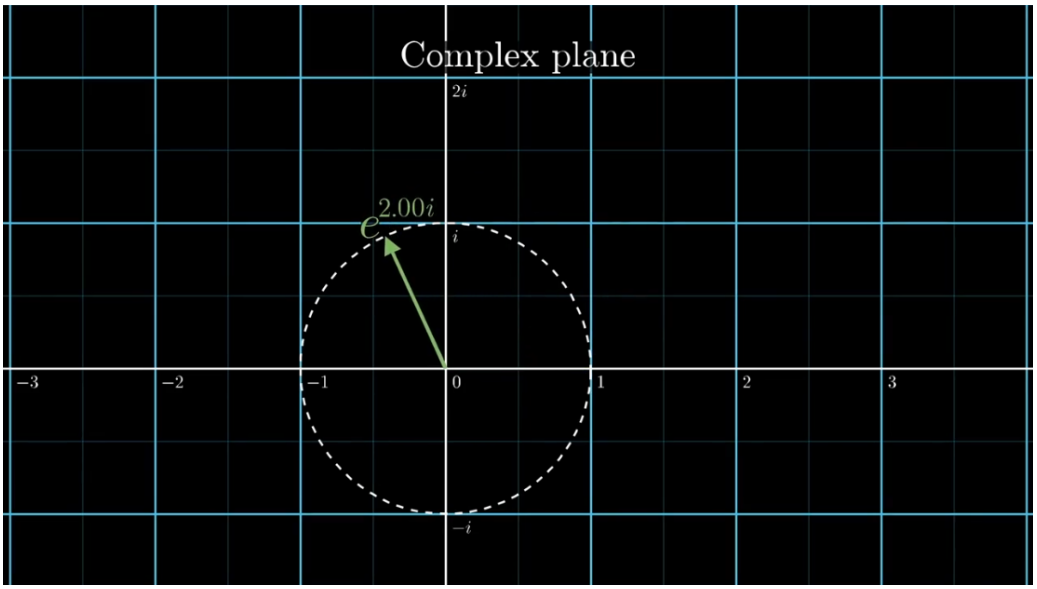
По своей сути, дискретное преобразование Фурье — преобразование одного набора коэффициентов (значений) в другой набор по формуле:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} kn}.$$

Давайте разберёмся, в чём смысл этой формулы (следующую часть с картинками можно пропустить, если вы торопитесь и не хотите особо вникать, все картинки взяты с замечательного видео).

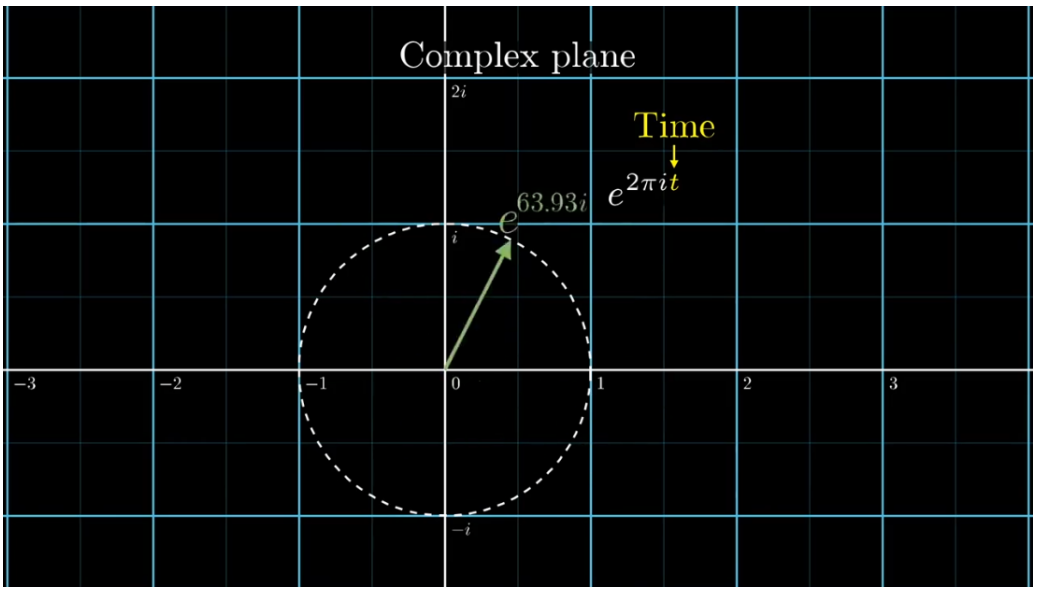
Сделаем оговорку, что дискретное преобразование Фурье обычно применяют к функциям от некоторого параметра t (можно воспринимать его как время, например, при анализе сигналов), эти функции мы и дискретизируем с некоторой точностью и получаем исходный набор значений. К нему и применяется преобразование.

Для начала рассмотрим комплексную плоскость и вектор e^{ik}:



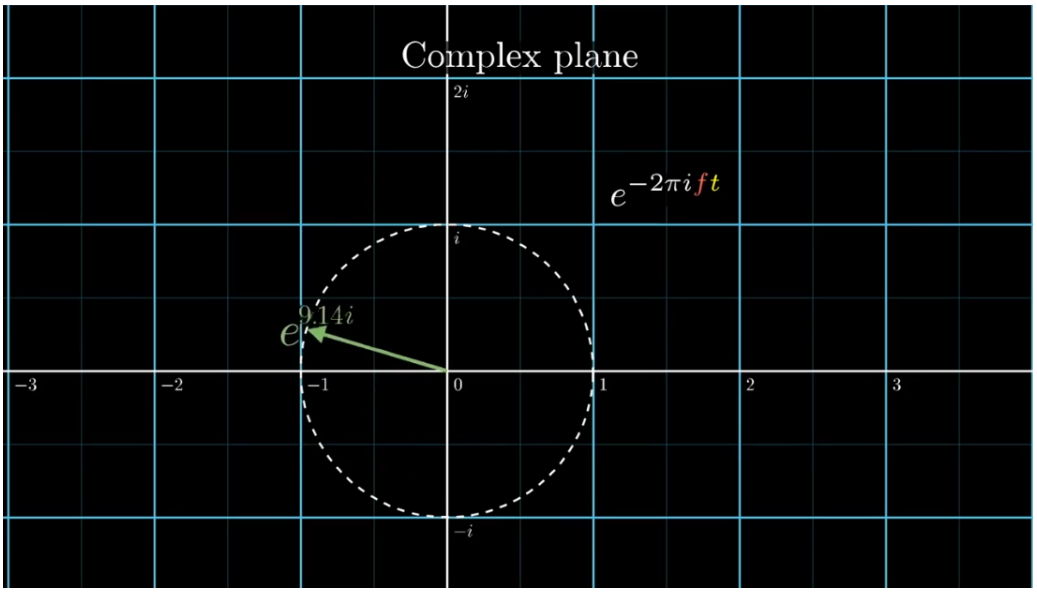
Вектор e^{ik} на комплексной плоскости.

Как известно из формулы Эйлера, этот вектор будет описывать окружность при повороте (изменении k). Далее перепишем для удобства $k = 2\pi t$, где t можно воспринимать как время (про применение дискретного преобразования поговорим позже, там оно понадобится).



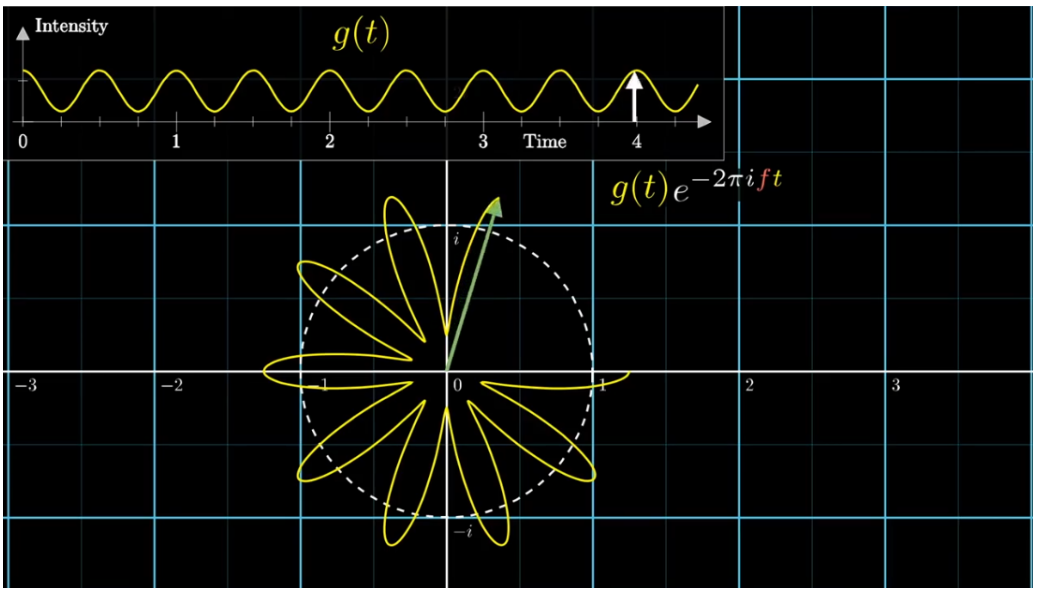
Сделаем замену $k = 2\pi t$.

Теперь добавим множитель f в степень, чтобы регулировать «скорость поворота» вектора. Также добавим знак $-$ в степень по соглашению.



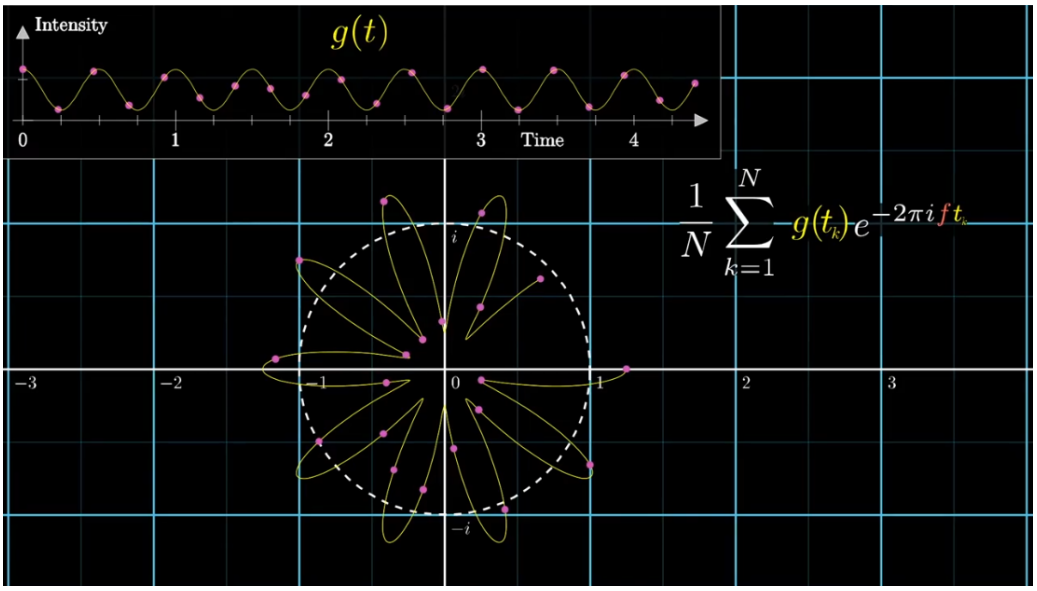
Добавим множитель f.

Пусть теперь модуль нашего вектора не 1 а некоторая функция $g(t)$ (для неё мы и делаем преобразование Фурье).



Добавим множитель $g(t)$.

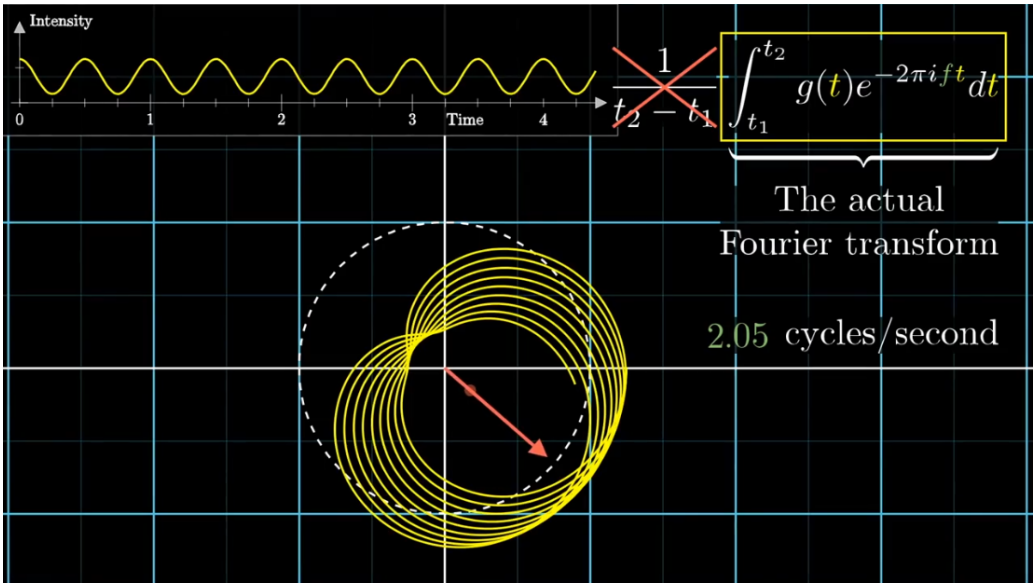
Как мы видим из рисунка, наша функция принимает различные значения, но нам нужно как-то охарактеризовать это множество значений в зависимости от параметра f. Для этого мы возьмём некоторый дискретный набор значений функции и получим среднее значение.



Усредним наши значения.

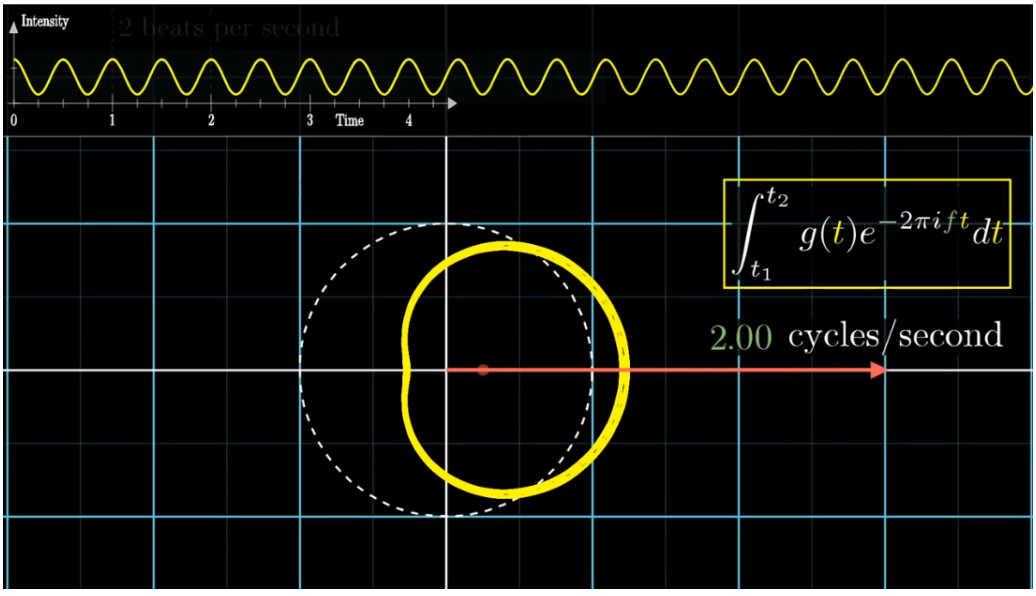
Геометрический смысл такой формулы — центр масс системы точек (розовые точки на предыдущем рисунке). Получается, наша полученная функция теперь описывает положение центра масс системы в зависимости от параметра f.

При $N \to \infty$ мы получаем интеграл. Осталась одна важная деталь: прямое преобразование Фурье — это наша построенная функция, но без множителя $\frac{1}{N}$.



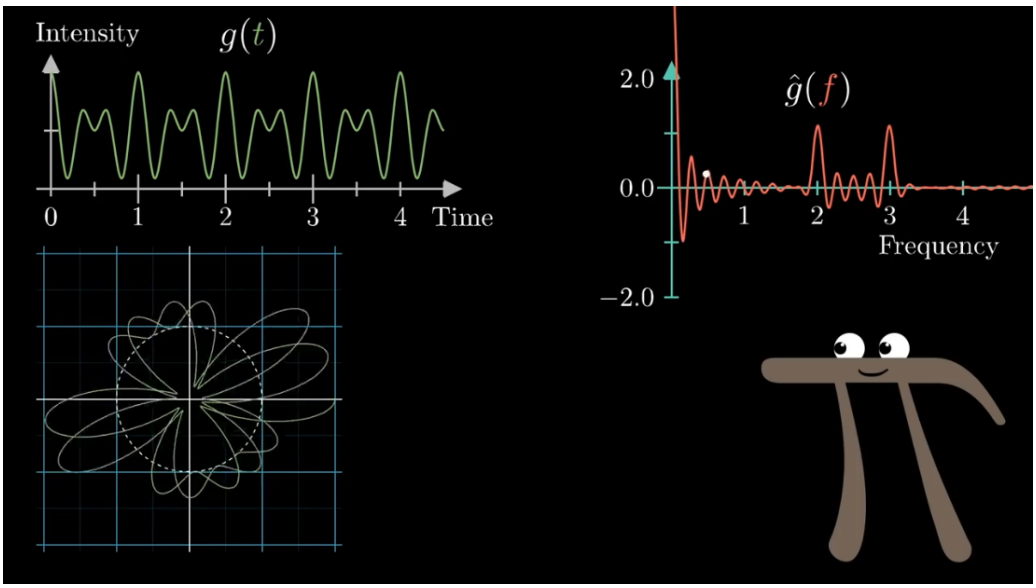
Уберём множитель $\frac{1}{N}$.

Что теперь мы получили? На самом деле, теперь, чем дольше в нашей функции присутствует функция на определённой частоте, тем больше у нас будет значение после преобразования Фурье.



Частота нашей функции совпала с f, так что красный вектор (наше значение после Фурье) остаётся на месте, но растёт.

Итого мы по нашей исходной функции зависимости от t построили функцию зависимости от частоты f. В формуле дискретного преобразования Фурье частота f указана неявно в виде $\frac{k}{N}$, а времени t соответствует n.



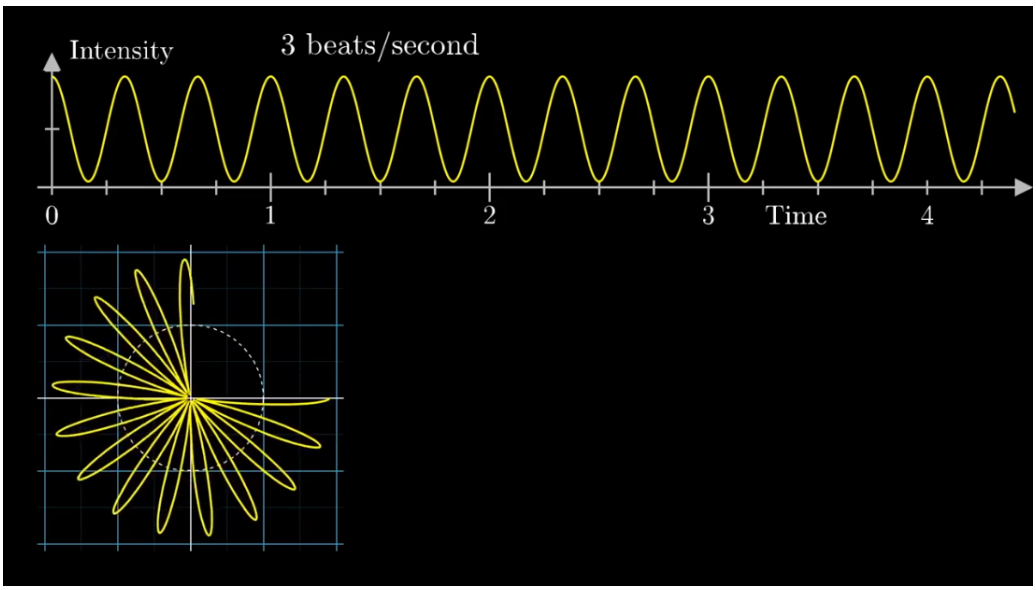
Итого

Часть с картинками закончилась, дальше формулы.

Аналогично можно ввести обратное преобразование Фурье:

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N}kn}.$$

В процессе построения преобразования Фурье мы рассматривали некоторое множество значений нашей функции, так вот это множество, на самом деле, является множеством значений на комплексных корнях из 1 n-ой степени. Это легко понять, обратившись к части с картинками. Действительно, мы при построении домножили наш вектор $e^{-2\pi it}$ на значение нашей функции $g(t)$. Т. е. фактически натянули нашу функцию на окружность, который описывает единичный вектор в комплексной плоскости.



«Натягиваем» нашу функцию на окружность, которую описывает единичный вектор в комплексной плоскости.

Утверждение. Сумма всех корней вида $w_n^k = e^{\frac{2\pi ik}{n}}, k = 0 \dots n-1$ равна нулю.

Доказательство. Это напрямую следует из того, как мы выбираем наши точки на комплексной окружности. Фактически, мы делим её этими точками на n дуг равной длины. В итоге эти точки образуют равносторонний многоугольник. Если просуммируем его вершины как вектора, получим в итоге 0. ■

13.2 Быстрое преобразование Фурье.

Пусть у нас есть многочлен степени n, где n — степень двойки.

$$A(x) = a_0x^0 + a_1x^1 + \ldots + a_{n-1}x^{n-1}.$$

Разделяем его на два многочлена:

$$A_0(x) = a_0x^0 + a_2x^1 + \ldots + a_{n-2}x^{\frac{n}{2}-1}$$
$$A_1(x) = a_1x^0 + a_3x^1 + \ldots + a_{n-1}x^{\frac{n}{2}-1}$$

Для которых выполняется:

$$A(x) = A_0(x^2) + xA_1(x^2).$$

Мы уже выяснили ранее, что дискретное преобразование Фурье для многочлена — его значение на комплексных корнях единицы, тогда с учётом $w_n^k = e^{\frac{2\pi k}{n}} = (w_n^1)^k$ получаем:

$$DFT(a_0, a_1, \ldots, a_{n-1}) = (A(w_n^0), A(w_n^1), \ldots, A(w_n^{n-1})).$$

Докажем вспомогательное утверждение:

Утверждение.

$$DFT(A \cdot B) = DFT(A) \cdot DFT(B).$$

Доказательство. Следует из того, что в каждой точке при умножении многочленов их значения перемножаются, т. е.

$$(A \cdot B)(x) = A(x) \cdot B(x).$$

■

Перейдём к саму алгоритму. Из неравенства для $A(x)$ видно, что если мы научимся за линию считать $A(x)$ из посчитанных на предыдущем шаге $A_0(x^2)$ и $A_1(x^2)$, то асимптотика алгоритма будет $O(n \log n)$.

Пусть $\{y_k^0\}_{k=0}^{\frac{n}{2}-1} = DFT(A_0), \{y_k^1\}_{k=0}^{\frac{n}{2}-1} = DFT(A_1)$.

Найдём выражение для $DFT(A) = \{y_k\}_{k=0}^{n-1}$: Для первой половины коэффициентов по определению:

$$y_k = A(w_n^k) = A_0(w_n^{2k}) + w_n^k \cdot A_1(w_n^{2k}) = y_k^0 + w_n^k y_k^1, \text{ при } k = 0 \dots \frac{n}{2} - 1.$$

Учитывая $w_n^n = 1, w_n^{\frac{n}{2}} = -1$:

$$\begin{aligned} y_{k+\frac{n}{2}} = A\left(w_n^{k+\frac{n}{2}}\right) &= A_0(w_n^{2k+n}) + w_n^{k+\frac{n}{2}} A_1(w_n^{2k+n}) \\ &= A_0(w_n^{2k} w_n^n) + w_n^k w_n^{\frac{n}{2}} A_1(w_n^{2k} w_n^n) = A_0(w_n^{2k}) - w_n^k A_1(w_n^{2k}) = y_k^0 - w_n^k y_k^1 \end{aligned}$$

Полученные преобразования называют «преобразованием бабочки».

Таким образом, задача DFT сводится к двум подзадачам и пересчёту индексов по формулам выше. Обратное DFT ничем не отличается от прямого, кроме формулы пересчёта коэффициентов. Её можно найти, посчитав обратную матрицу к матрице перехода при прямом DFT.

Некоторые оптимизации:

1. Для начала заметим, что на первом уровне рекурсии к первому массиву относятся элементы на чётных позициях, т. е. те, у которых младшие биты позиции равны нулю, а ко второму массиву — те, у которых младшие биты позиции равны единице. На следующем шаге аналогично со второй позицией и т. д. Таким образом, мы можем переупорядочить наш набор элементов так, чтобы элементы занимали свои позиции после окончания рекурсии. Например, для $n = 8$ вид следующий:

   $$a = \{[(a_0, a_4), (a_2, a_6)], [(a_1, a_5), (a_3, a_7)]\}.$$

   Пусть теперь после выполнения алгоритма к каждой половине у нас такой вид массива на последнем шаге:

   $$a = \left\{\left[y_0^0, y_1^0, y_2^0, y_3^0,\right], \left[y_0^1, y_1^1, y_2^1, y_3^1,\right]\right\}.$$

   Теперь нам осталось объединить оба массива. Элементы стоят так, что мы можем сделать это прямо на месте без доп. памяти с помощью преобразования, полученного ранее:

   $$a = \left\{\left[y_0^0 + w_n^0 y_0^1, y_1^0, y_2^0, y_3^0,\right], \left[y_0^0 - w_n^0 y_0^1, y_1^1, y_2^1, y_3^1,\right]\right\}.$$

   Аналогичным образом для других элементов массива. Таким образом, мы избавились от рекурсии вовсе: сделали поразрядную обратную сортировку, а потом применили преобразование бабочки для всех уровней рекурсии.

2. Можно предпосчитать обратную сортировку для битов, если функция вызывается часто.
3. Предпосчитать все нужные степени w_n^k.

13.3 Умножение многочленов.

Утверждение. Для произведения двух многочленов справедлива формула:

$$A \cdot B = DFT^{-1}\left(DFT(A) \cdot DFT(B)\right).$$

Доказательство. Следует из утверждения, что $DFT(A \cdot B) = DFT(A) \cdot DFT(B)$. Здесь и в формуле произведения подразумевается почленное умножение для DFT. ■

Утверждение. Асимптотика такого умножения — $O(n \log n)$.

Доказательство. Каждое DFT выполняется за $O(n \log n)$, почленное умножение многочленов за $O(n)$. ■

14 Лекция 14: Комбинаторные игры.

14.1 Игра с камнями.

Условие Есть N камней, игрок может брать от 1 до K камней. Побеждает игрок, взявший последний камень.

Разбор в зависимости от числа N:

- $N = 0$ — проигрыш
- $N = 1, \ldots, k$ — выигрыш
- $N = k + 1$ — проигрыш
- $N = k + 2, \ldots, 2k + 1$ — выигрыш (берём от 1 до k камней соответственно)
- $N = 2(k + 1)$ — проигрыш
- ...

14.2 Модификация игры с камнями.

Условие Есть N камней, игрок может брать от 1 до K камней. Проигрывает игрок, взявший последний камень.

Разбор в зависимости от числа N:

- $N = 0$ — выигрыш
- $N = 1$ — проигрыш
- $N = 2, \ldots, k + 1$ — выигрыш (можно взять от 1 до k камня соответственно и свести для оппонента ситуацию к $N = 1$)

- $N = k + 2$ — проигрыш
- $N = k + 3, \ldots, 2k + 2$ — выигрыш
- $2(k + 1) + 1$ — проигрыш
- ...

14.3 Метод симметричной стратегии.

Условие Игра в монеты за круглым столом. Игроки по очереди кладут круглые монеты на круглый стол так, чтобы они не пересекались. Игрок, который не может сделать ход, проигрывает.

У первого игрока есть выигрышная стратегия, вот она:

- Первым ходом кладём монету в центр
- Далее повторяем каждый ход оппонента симметрично относительно центра

Это выигрышная стратегия, т. к. после хода первого игрока у нас картинка расположения монет всегда симметрична относительно центра. А это значит, что если второй игрок нашёл, куда поставить, то и симметричное место будет свободно.

Некоторые классификации игр:

Определение 14.1. Комбинаторные игры — игры с полной информацией, которые ведут два игрока, делая ходы по очереди.

Определение 14.2. Справедливая игра — игра, в которой возможные ходы каждого игрока совпадают.

Определение 14.3. Нормальная игра — игра, в которой игрок, который не может сделать ход, проигрывает.

14.4 Игры на графе.

Пусть некоторая игра ведётся на некотором графе G. Т. е. текущее состояние игры — некоторая вершина графа, и из каждой вершины рёбра идут в те вершины, в которые можно пойти следующим ходом.

Определение 14.4. Терминальные вершины — вершины, из которых нельзя сделать ход.

Определение 14.5. Выигрышная игра — вне зависимости от действий второго игрока, первый выигрывает.

Определение 14.6. Проигрышная игра — вне зависимости от действий первого игрока, второй игрок может выиграть.

Определение 14.7. Ничейная игра — игра, которая не является ни выигрышной, ни проигрышной.

Утверждение. Ничейные игры продолжаются бесконечно, граф ничейной игры должен содержать циклы.

Утверждение. Если из некоторой вершины есть ребро в проигрышную вершину, то он выигрышная.

Утверждение. Если из некоторой вершины все рёбра исходят в выигрышные вершины, то это вершина проигрышная.

Утверждение. Если граф не содержит циклы, то все его вершины можно разбить на два непересекающихся множества выигрышных и проигрышных вершин.

Алгоритм построения такого разбиения:

- Граф без циклов, значит можно запустить топологическую сортиовку.
- Обходим граф в порядке, обратном топологической сортировки и определяем, какая вершина перед нами.

14.5 Стратегия

Определение 14.8. Стратегия — отображение из множества нетерминальных вершин в множество вершин графа.

Определение 14.9. Выигрышная стратегия для первого игрока — стратегия, следуя которой, первый игрок выигрывает, вне зависимости от действий второго игрока.

Утверждение. В любой игре на ациклическом графе существует стратегия s, такая что для игр, начинающихся в выигрышных вершинах, она является выигрышной.

Доказательство. Достаточно переходить по графу по первоначальному правилу (из проигрышной в любую выигрышную, из выигрышной — в проигрышную. ■

14.6 Ретроспективный анализ.

Пусть W— множество выигрышных вершин, L — проигрышных.

Утверждение. В графе с циклами нельзя использовать алгоритм поиска W, L с помощью топологической сортировки.

Таким образом, приходим к алгоритму ретро-анализа:

1. Начинаем обход с терминальных вершин с помощью dfs или bfs.

2. Для каждого ребра (u, v) исходя из определённых ранее правил получаем:

$$v \in L \Rightarrow u \in W$$
$$v \in W \Rightarrow u \in L, \text{ если } (u, v) \text{ — последнее среди нерасмотренных ребер.}$$

3. Дополнительно нужно для каждой вершины хранить количество необработанных исходящих ребер (нужно для второго условия в пункте 2).

Утверждение. Время работы такого алгоритма $O(|E|)$.

Данный алгоритм можно модифицировать так, чтобы он выяснял не только тип вершины, но и минимальное число ходов до выигрыша (максимальное число ходов до проигрыша). Для этого вместо dfs воспользуемся bfs и будем для каждого уровня вершин хранить число до победы (до проигрыша). Получается, мы разделим все наши вершины на классы вида (напрямую определение выигрышных и проигрышных позиций):

1. $L_0 = \{u \mid \text{из } u \text{ не выходит ни одной дуги}\}$
2. $W_1 = \{u \mid \exists v \in L_0 \colon uv \in E\}$
3. $L_2 = \{u \mid \forall v \colon uv \in E \to v \in W_1\}$
4. ...
5. $L_{2i} = \{u \mid \forall v \colon uv \in E \to v \in W_{2i-1}\}$
6. $W_{2i+1} = \{u \mid \exists v \in L_{2i} \colon uv \in E\}$

Утверждение. $W = \bigcup L_i, L = \bigcup L_i$.

Утверждение. Множество ничейных позиций $D = V \setminus (P \cup N)$.

15 Лекция 15: Минимакс. Альфа-бета отсечение. Теория Шпрага-Гранди.

15.1 Алгоритм минимакс.

Утверждение. Для игры можно построить дерево игры — граф игры, расположенный по слоям возможных ходов.

Такое дерево ходов можно обойти с помощью ретроспективного анализа, но часто:

- сложно определить исход игры
- сложно оценить минимальное число ходов до выигрыша (максимальное число ходов до проигрыша)

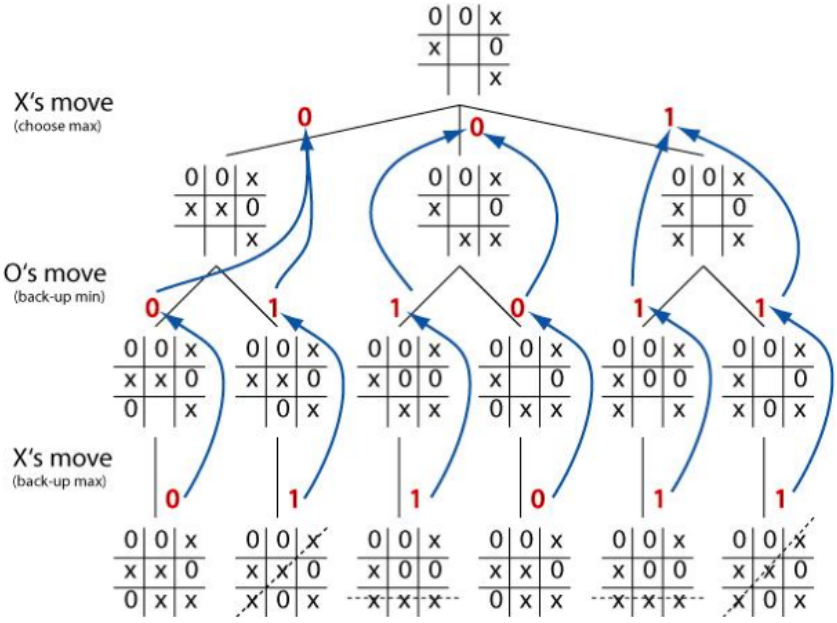
Поэтому вводят дополнительную функцию оценки вероятности исхода (приблизительного числа ходов до исхода).

Определение 15.1. Эвристика — числовая характеристика игры, зависящая только от состояния игры. Равна $+\infty$ для выигрышных, $-\infty$ для проигрышных.

Примером эвристики для шахмат, например, может послужить разница между суммы весов своих фигур и суммы оппонента.

Рассмотрим теперь алгоритм минимакс, который позволяет получить оценку для всей игры с помощью эвристик.

1. Просматриваем дерево, начиная с терминальных вершин, находим их эвристику.
2. Эвристику следующего уровня смотрим по минимуму (максимуму) всех детей для каждой вершины в дереве игры. Минимум или максимум выбирает в зависимости от того, чем сейчас ход. Минимум — ход оппонента (он старается нам навредить) и максимум для нашего хода.



Пример дерева игры для крестиков-ноликов.

Утверждение. Время работы алгоритма минимакса приблизительно равно коэффициенту ветвления, возведенному в степень числа слоёв.

15.2 Альфа-бета отсечение.

Число просмотренных поддеревьев можно уменьшить, если сразу отметать ходы неэффективные для нас. Пусть мы рассмотрели уже один свой ход 1 и все возможные ответы соперника и начали рассматривать свой второй ход 2. Как только среди ответов соперника появится такой ход, который ставит нас в ситуацию хуже, чем самая плохая ситуация, в которую может нас поставить соперник, если мы сходим ходов 1, то можно прекращать перебор возможных ответов соперника на ход 2 и переходить к рассмотрению следующего нашего хода.

Значение в узле обозначим за $f(v)$, тогда введём две функции:

- α — текущее максимальное значение оценки узла дял уровня максимизации.

$$\alpha = \max(\alpha, f(v)).$$

Изначально $\alpha = -\infty$.

- β — текущее минимальное значение оценки узла дял уровня минимизации.

$$\beta = \min(\beta, f(v)).$$

Изначально $\beta = +\infty$.

Утверждение. Из определение понятно, что ситуация $\alpha \geq \beta$ эквивалентна описанной выше.

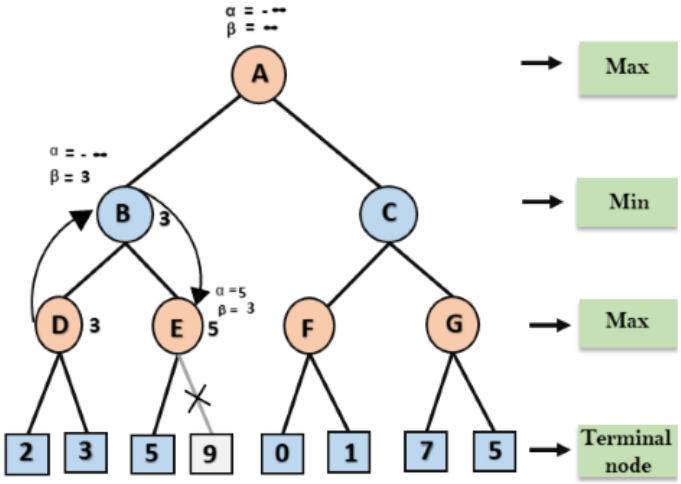


Иллюстрация к альфа-бета отсечению.

15.3 Теория Шпрага-Гранди.

Утверждение. Теория Шпрага-Гранди описывает равноправные игры, т. е. игры, в которых ращрешённые ходы зависят только от состояния игры. Игроки полность равноправны. Также предполгается что игра конечна. Иными словами, такую игру можно полностью описать ориентированным ациклическим графом.

Определение 15.2. Игра «Ним» — игра, при которой два игрока по очереди вытаскивают из какой-то кучки камней ненулевое число камней. Проигрывает тот, кто не может сделать ход (все кучки пусты).

Определение 15.3. Прямая сумма игр $A + B$ — игра, составленная из двух, при которой игрок в начале хода выбирает одну из игр, а потом делает в ней ход (нельзя выбрать игру, которая находится в терминальном состоянии).

Утверждение. Прямая сумма в случае Нима эквивалентна двум кучкам камней (каждую кучку можно рассматривать как отдельную игру).

Утверждение. Прямая сумма коммутативна и ассоциативна, нейтральным элементом относительно неё является игра в терминальной вершине, т. е. $A + A_0 = A$.

Утверждение. Если A — игра на ациклическом графе, то $A + A$ — проигрышная игра.

Доказательство. Оппоненту достаточно применить симметричную стратегию. ■

Определение 15.4. Исход игры — функция $v(A) = +1$, если A — выигрышная, $v(A) = -1$, если A — проигрышная. $v(A) = 0$, если A — ничейная.

Определение 15.5. Две игры A и B эквивалентны по Гранди ($A \sim B$), если $\forall$ игры C верно:

$$v(A + C) = v(B + C).$$

Утверждение. Это отношение является отношением эквивалентности.

Утверждение.

$$(A_1 \sim A_2) \wedge (B_1 \sim B_2) \Rightarrow (A_1 + B_1) \sim (A_2 + B_2).$$

Доказательство.

$$v((A_1+B_1)+C) = v(A_1+(B_1+C)) = v(A_2+(B_1+C)) = v((A_2+C)+B_1) = v((A_2+B_2)+C).$$

■

15.4 Эквивалентность проигрышных игр.

Теорема 15.1 (Эквивалентность проигрышных)**.** *Для любой проигрышной игры L и любой игры C выполнено равенство $v(L + C) = v(C)$.*

Доказательство. Необходимо рассмотреть три случая — C выигрышная, проигрышная или ничейная.

Если C выигрышная или проигрышная, то выигрывающий в C игрок (первый или второй, соответственно) играет в C в соответствии со своей выигрышной стратегией, а если противник делает ходы в L, то отвечает в ней в соответствии с выигрышной стратегией в L для второго игрока.

Пусть теперь C — ничейная. Проведем индукцию по числу ходов до проигрыша в L.

База очевидна. Если L проигрышная за 0 ходов, то $L+C$ — ничейная (в L нельзя сделать ход изначально).

Пусть теперь L проигрышная, и для всех игр, в которых первый игрок проигрывает быстрее чем в L, утверждение доказано. Теперь рассмотрим игру $L + C$. Можно ли в ней выиграть?

Делать в таком случае ход в L бессмысленно, т. к. противник точно сможет на него ответить, и задача будет сведена к той же, но проигрыш в L будет за меньшее число шагов. Ходить в C в выигрышную позицию также не имеет смысла, т. к. в таком случае противник получил пару из выигрышной и проигрышной игр, в которой выиграет (по доказанному в самом начале). Значит нужно ходить в ничейную позицию, и $L + C$ — ничейная. ■

Утверждение. Если A и B — проигрышные игры, то $A \sim B \sim A_0$.

Утверждение. Для любой игры A на ациклическом графе выполняется $A + A \sim A_0$.

Утверждение. Существует бесконечно много классов эквивалентности выигрышных игр.

Обозначим за A_i игру Ним с одной кучкой в i камней. Очевидно, что A_0 — проигрышная, а A_i — выигрышная (можно подло взять все камни за раз).

Лемма 15.2. *Игра A_i не эквивалентна A_j для любых $i \neq j$.*

Доказательство. Пусть $i < j$. Возьмём в качестве C игру A_i. Тогда игра $A_i + A_i$ проигрышная (можно использовать симметричную стратегию. С другой стороны, игра $A_j + A_i$ — выигрышная. Достаточно взять из кучки с j камнями $j - i$ камней, тогда игра сведется к предыдущему случаю. ■

15.5 Теория Шпрага-Гранди для игр на ациклических графах.

Лемма 15.3. *(Об эквивалентности ниму)*

$$A \sim A_i \Leftrightarrow A + A_i \sim A_0.$$

Доказательство. $\Rightarrow$

$$A \sim A_i \Rightarrow A + A_i \sim A_i + A_i \sim A_0.$$

$\Leftarrow$

$$A + A_i \sim A_0 \Rightarrow A \sim A + A_0 \sim A + (A_i + A_i) \sim (A + A_i) + A_i \sim A_0 + A_i \sim A_i.$$

■

Теорема 15.4 (Шпраг, Гранди)**.** *Если A — игра на ациклическом графе, то $\exists i \colon A \sim A_i, i = g(A)$. Если из игры A существуют ходы в игры $A_1, A_2, \ldots, A_n$, то*

$$g(A) = mex(g(A_1), g(A_2), \ldots, g(A_n)),$$

где функция тех от множества чисел возвращает наименьшее неотрицательное числа, не встречающееся в этом множестве.

Доказательство. Доказательство будем вести по индукции по макимальному пути в графе игры A от начальной вершины до терминальной. Если эта длина равна 0, то начальная вершина A является терминальной, значит является проигрышной и эквивалентна A_0.

Пусть для всех игр с длиной максимального пути меньше, чем в A, утверждение доказано. Пусть из игры A существуют ходы в игры $A_1, A_2, \ldots, A_n$. Максимальная длина пути из начальной вершины до терминальной в каждой из этих игр меньше чем в игре A, значит по предположению индукции каждая из них экзвивалентна игре ним. Выберем $i = mex(g(A_1), g(A_2), \ldots, g(A_n))$.

По предыдущей лемме $A \sim A_i \sim A + A_i \sim A_0$. Т. е. необходимо доказать, что $A + A_i$ — проигрышная.

Пусть первый игрок сделал первый ход в игре A. Теперь второй игрок будет ходить в сумме $A_j + A_i$ для некоторого j. $A_j \sim A_{g(A_j}$ (это следует из предположения индукции), $i \neq g(A_j)$ (по определению i) $\Rightarrow A_j + A_i$ — выигрышная (достаточно вспомнить стратегию игры при двух кучках камней).

Значит первый игрок должен сделать свой ход в игре A_i. Такой ход возможен в случае $i > 0$. После этого хода второй игрок должен делать ход в сумме $A+A_k, k < i$. По определению i — минимальное число, которое не встречается среди $g(A_1), \ldots, g(A_n)$, то $\exists j\colon g(A_j) = k$. Тогда второй игрок может сделать в A ход в игру A_j, и первый игрок получит игру A_j+A_k, которая является проигрышной, т. к. $A_j \sim A_k$ (потому что $g(A_j) = k$).

Значит $A + A_i$ проигрышная, значит $A \sim A_i$, что и требовалось доказать. ■

Теорема 15.5 (Число Гранди суммы игр)**.**

$$A_i + A_j \sim A_{i \wedge j}.$$

Доказательство. Докажем по индукции по сумме $i + j$.

База: $i + j = 0 \Rightarrow (i = 0) \wedge (j = 0) \Rightarrow A_i + A_j = A_0 + A_0 \sim A_0 = A_{i \wedge j}$ Переход: учитывая предыдущую теорему, для доказательства достаточно показать, что

$$i \wedge j = mexF(i, j),$$

где $F(i, j)$ — множество чисел Гранди всех игр, в которые возможен ход их игры $A_i + A_j$:

$$F(i, j) = \{i \wedge 0, i \wedge 1, \ldots, i \wedge (j - 1), 0 \wedge j, 1 \wedge j, \ldots, (i - 1) \wedge j\}.$$

Очевидно, что $i \wedge j$ не лежит в $F(i, j)$. Осталось показать, что все меньшие числа есть в этом множестве.

Рассмотрим произвольное число $r < i \wedge j$. Пусть t — номер самого старшего разряда, в котором r отличается от $i \wedge j$. Ясно, что r имеет в этом разряде ноль, а $i \wedge j$ — единицу (т. к. $r < i \wedge j$). Значит t-е разряды i и j различны, пусть, без ограничения общности, t-й разряд числа i равен единице.

Тогда построим число i' таким образом: разряды старше t-го в i' сделаем равным соответствующим разрядам i, t-й разряд обнулим, а более младшие выберем так, чтобы выполнялось равенство $i' \wedge j = r$ (мы можем это сделать, потому что меняем в i только t-й бит и младше него).

По построению $i' < i$ (мы обнулили t-й бит), значит по построению $F(i, j)$ число $i' \wedge j = r$ встречается в $F(i, j)$.

Значит $i \wedge j$ является минимальным числом, которое не встречается в множестве $F(i, j)$, а значит $A_i + A_j \sim A_{i \wedge j}$, что и требовалось доказать. ■