Алгоритмы и структуры данных (основной поток).
3 семестр.
Конспект лекций (2021).
Лектор: Степанов Илья Даниилович

Алексей Горбулев

2021

Содержание

Disclaimer

Внимание! Некоторые конспекты лекций, возможно, требуют переработки. Также по историческим причинам (этот конспект начал создаваться до вступления в КТЛ) отличается стиль. Посмотреть проект этого конспекта можно здесь. За перевод в стиль КТЛ буду благодарен, поскольку это оказывается не таким простым занятием.

Лекция 1. Базовые строковые алгоритмы

Базовые строковые алгоритмы

Ничего нового не будет, глобальные темы: строки, геометрия, вероятностные алгоритмы. Строки — самые полезные структуры данных среди тех, которые мы учим 3 семестра.

Def. Строка — последовательность символов $S_0S_1 \dots S_{n-1}$, где $S_i \in \{a, b, \dots, z\}$.

Вхождение шаблона в текст

Задача. Пусть *pattern* — шаблон, а *text* — текст. Узнать все вхождения (количество вхождений) шаблона в текст.

Будут рассмотрены решения с хешами (Рабин-Карп), префикс-функцией, Z-функцией.

Полиномиальное хеширование

Мы хотим хранить хеши — сжатую запись объектов, при этом могут быть коллизии, но считаем, что это происходит довольно редко. Пусть у нас строка $S_0S_1 \dots S_{n-1} = S$, тогда $h(S) = (S_0 \cdot p^{n-1} + S_1 \cdot p^{n-2} + \dots + S_{n-1})\%m$, где p — простое небольшое число, m — простое большое число. S_i — это либо номер в ASCII (int(s[i])), либо просто номер в алфавите (больше подходит, если алфавит уже, чем ASCII), это будет s[i] - 'a' + 1.

Если мы работаем в $\Sigma = \{a, b, \dots, z\}$, и берём $p = 31, p > 26 = |\Sigma|$, и их хеш будет число в p-ичной системе счисления. Если мы бы не брали по модулю, получали бы идеальное хеширование. Но мы берём по модулю, возникают коллизии.

Утверждение. (без доказательства) Пусть m — случайное число из $[10^9, 2 \cdot 10^9]$. Тогда $(s_0 \cdot p^{n-1} + \dots + s_{n-1})\%m$ близко к равномерному распределению на $[0, m-1]$, и вероятность коллизии минимальна.

Если $h(s) \neq h(t) \Rightarrow s \neq t$. $h(s) = h(t) \Rightarrow s = t$ с большой вероятностью, но не 100%, приблизительно $1 - \frac{1}{m}$. Так и работают все алгоритмы на хешах — с большой вероятностью ответ будет правильным.

Algorithm. (Рабин-Карп) Пусть у нас есть шаблон *pat* и текст t. Возьмём все подстроки текста

t длины $|pat|$, и если $h(substr)$ совпадёт с $h(pat)$, то считаем, что найдено вхождение. Здесь h — полиномиальный хеш. Асимптотика: $O(|pat| + |t|)$.

Пусть у нас дан $text = t_0 t_1 \dots t_{n-1}$, $pref[i] = h(t_0, \dots, t_i)$. Тогда:

```
pref[0] = t[0];
pref[i+1] = (pref[i] * p + t[i+1]) % m;
h(t[l], ..., t[r]) = (pref[r] - pref[l - 1] * pow(p, r - l + 1)) % m; // l >= 1
```

Префикс-функция

Def. Пусть есть строка $S = S_0 S_1 \dots S_{n-1}$, тогда $\pi_0, \pi_1, \dots, \pi_{n-1}$ — префикс-функция для S, где π_i — длина максимального собственного суффикса подстроки $S_0 S_1 \dots S_i$, который совпадает с префиксом той же длины.

Пример: $s = ababbabbaaba$, $\pi : [0, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3]$.

Пусть есть алгоритм, находящий π за $O(|s|)$. Вспомним задачу про вхождение шаблона в текст. Рассмотрим строку $pattern\#text$, где $\#$ — разделитель, отличный от всех остальных символов. Тогда значение префикс-функции не превосходит $|pattern|$, и если значение $\pi = |pattern|$, то мы обнаружили вхождение шаблона в текст.

Note. Из-за $\#$ значение π не сможет превзойти $|pattern|$.

Algorithm. (Нахождение префикс-функции) Пусть $\pi_0, \dots, \pi_{i-1}$ известны. Ищем π_i. Заметим, что:

1) $\pi_i \leqslant \pi_{i-1} + 1$, $\pi_{i-1} \geqslant \pi_i - 1$ (рассматриваем новый символ, либо префикс удлиняем на 1, либо начинаем с нуля)

2) супрефикс для $s_0 \dots s_i$ без последнего символа — это супрефикс для $s_0 \dots s_{i-1}$

3) второй по длине супрефикс для $s_0 \dots s_{i-1}$ имеет длину $\pi_{\pi_{i-1}-1}$

Но почему верен пункт 3? Например, $s = abacabadabacaba$, пусть предыдущий символ: $i-1$, непустые собственные префиксы по длине убывают так: $abacaba$, aba, a; $\pi_{i-1} = 7$.

Теперь реализуем:

```
vector<int> p_function(const string& s) {
  int n = s.size();
  vector<int> p(n);
  for (int i = 1; i < n; ++i) {
    int j = p[i-1];
    while (j > 0 && s[j] != s[i]) {
      j = p[j - 1];
    }
    if (s[j] == s[i]) {
      ++j;
    }
    p[i] = j;
```

```
  }
  return p;
}
```

Пример: $s = abacabaab$, $\pi = [0, 0, 1, 0, 1, 2, 3, ?]$. Хотим найти значение ?. $j = 3$ не подходит, символы не подходят. Возьмём $j = p[2] = 1$, но и тут тоже проблема, и так заканчиваем в $j = p[0] = 0$. Последняя надежда на равенство первого и текущего символов, и это удачно. $? = 1$.

Посмотрим на асимптотику. Чем больше итераций while, тем меньше значение префикс-функции. Суммарно цикл while работает $O(n)$ времени, так как значение префикс-функции уменьшается хотя бы на 1, а расти может не более, чем на 1. Значит, и весь алгоритм работает за $O(n)$ времени, а также $O(n)$ памяти.

Плюс префикс-функции: никакой вероятности, всё точно, детерминировано.

Z-функция

Есть строка $s_0 s_1 \dots s_{n-1}$, значения функции: $z_0, z_1, \dots, z_{n-1}$, каждое значение z_i — наибольшая длина префикса строки $s_i, \dots, s_{n-1}$, который совпадает с префиксом $s_0, \dots, s_{n-1}$ той же длины (Z-блока).

Наивный алгоритм работает за $O(n^2)$, передвигаем два указателя до первого различия.

Def. Z-блок — префикс строки $s_i, \dots, s_{n-1}$, совпадающий с префиксом строки $s_0, \dots, s_{n-1}$ той же длины.

Algorithm. (Z-функция) Построим Z-функцию для строки $pattern\#text$, если с позиции i начинается вхождение шаблона, то $z[i] = |pattern|$, $z[i] \leqslant |pattern|$ в силу того, что # не входит ни в $pattern$, ни в $text$.

Найдём Z-функцию так. Храним Z-блок с самой правой границей. Пусть $s_l, \dots, s_r$ — тот самый Z-блок. Пусть $z_0, \dots, z_{i-1}$ известны, $i \in [l, r]$. Заметим, что если мы сдвинем блок на l, то рассмотрим ту же строку, но $z[i-l]$ уже посчитан. $z[i] \geqslant z[i-l]$, если $i + z[i-l] - 1 \leqslant r - l$. Иначе $z[i] \geqslant (r-l) - (i-l) + 1 = r - i + 1$.

```
vector<int> z_function(const string& s) {
  int n = s.size();
  vector<int> z(n); //z[0] = n by defition, but nobody cares
  int l = - 1, r  = -1;
  for (int i = 1; i < n; ++i) {
    if (l <= i && i <= r) {
      z[i] = min(z[i-l], r - i + 1);
    }
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
      ++z[i];
    }
    if (i + z[i] - 1 > r) {
      l = i;
```

```
      r = i + z[i] - 1;
    }
  }
  return z;
}
```

Анализ асимптотики будет на следующей неделе.

Лекция 2. Бор, алгоритм Ахо-Корасик

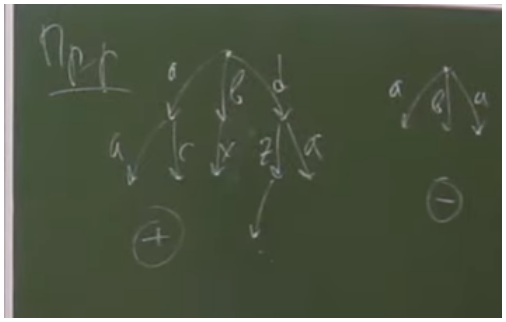
Анализ времени работы алгоритма нахождения Z-функции

Время работы: $O(n)$: при увеличении $z[i]$ сдвигает правую границу хотя бы на 1, количество итераций *while* не превысит количество возможных сдвигов вправо, а таковых максимум $O(n)$.

Бор

Def. Бор — корневое дерево, на рёбрах которого написаны символы алфавита, причём если из вершины исходит несколько рёбер, то обязательно они соответствуют разным символам ($\forall v$ из неё не может исходить двух рёбер с одинаковыми символами).

Пример:



Реализация бора

Построим бор по набору слов $s_1, \ldots, s_n$. Для удобства заведём структуру, которая хранит вершину бора.

```
struct node {
  map<char, int> to;
  bool term; // is node terminal? is there the final letter?
};
```

Далее заведём сам бор:

```
vector<node> true;
```

Реализация процедуры добаления слова в бор:

```
void add(const string &s) {
  int v = 0;
  for (int i = 0; i < s.size(); ++i) {
    if (!trie[v].to.count(s[i])) {
      trie.push_back(node());
      trie[v].to[s[i]] = int(trie.size()) - 1;
    }
    v = trie[v].to[s[i]];
  }
  trie[v].term = true;
}
```

Note. Вместо *map* подойдут и другие структуры. Например, можно использовать массив длины $|\Sigma|$, где несуществующие значения заполнены -1. Подойдёт так же и хеш-таблица.

Сравним их по памяти. Массив: $|\Sigma|$, *map* и хеш-таблица: k (число исходящих из вершины рёбер).

Сколько нужно ждать выполнения запроса? Массив: $O(1)$. *map*: $O(logk)$. Хеш-таблица: $O(1)$ в среднем.

Илья Степанов не особо часто использует хеш-таблицу: при маленьких значениях алфавита хеш-таблица может работать не так быстро.

Пример задачи, которую можно решить с помощью бора

Пример: Нам требуется поддерживать множество чисел S и поддерживать запросы:

1) добавить x в S;

2) узнать, есть ли y в S.

Можно это сделать с помощью бора. Представим числа как битовые строки (то есть в двоичной записи). Добавление числа в S — сродни добавления строки. Если мы дошли до терминальной вершины после всех спусков — число уже есть. Иначе его нет, и в случае запроса 1 есть возможность добавить данное число. Если у нас *int*, то у нас не более 30-31 спусков, и каждый запрос выполнится за $O(log(10^9))$ в случае использования *map*.

Задача. Реализовать запрос 3: удалить число x из S.

Алгоритм Ахо-Корасик

Научимся решать задачу. Пусть $s_1, s_2, \ldots, s_n$ — словарь (набор слов), $s_1, \ldots, s_n$ — словарные слова. Ограничений на пересечение словарных слов нет. Нужно найти общее число вхождений словарных

слов в текст t.

Ещё одна задача: вывести все вхождения словарных слов в текст t.

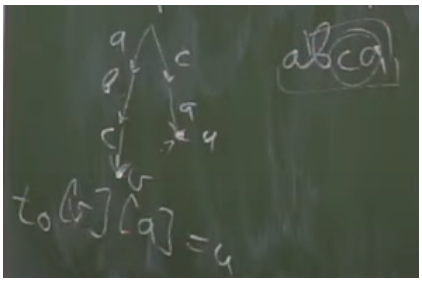
Первым делом строим бор по словарю $s_1, \ldots, s_n$. Пусть V — множество вершин бора, Σ — алфавит.

Note. Вершина v однозначно определяется соответствующим словом, то есть строкой, ведущей из корня.

Алгоритм для любой вершины $v \in V$ находит $link[v]$ — суффиксную ссылку — самый длинный собственный суффикс v, который можно прочитать от корня (есть в боре). При этом считаем, что у любого слова есть пустой суффикс, и если нет собственного суффикса ненулевой длины, то суффиксная ссылка будет вести в корень. При этом формально $link[root]$ не определён — с технической точки зрения, обращаться к суффиксной ссылке корня не будет никакого смысла.

Далее для любой вершины и любой буквы считаем $to[v][c]$ — самый длинный суффикс строки $v + c$, который есть в боре (v — вершина, c — буква), при этом не обязательно являющийся собственным.

Пример:



Реализация алгоритма Ахо-Корасик

Как у нас будут выглядеть верщины автомата? По мнению Ильи Степанова, вот так:

```
struct node {
  int to[alphabet.size()];
  bool term;
  int link;
  node() {
    memset(to, -1, sizeof(to));
    term = false;
    link = -1;
  }
};
```

Как мы будем строить суффиксные ссылки и другое? Конечно же, поиском в ширину! Будем хранить $queue < int > q$ — очередь вершин, для которых мы уже знаем суффиксную ссылку и величины to.

Храним вершины так:

```
vector <node> t;
```

Теперь пишем код!

```
void Aho-Corasick() {
  t[0].link = 0;
  for (int c = 0; c < alphabet.size(); ++c) {
    if(t[0].to[c] != -1) {
      continue;
    }
    t[0].to[c] = 0;
  }
  queue<int> q;
  q.push(0);
  while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int c = 0; c < alphabet.size(); ++c) {
      int u = t[v].to[c];
      if (t[u].link != -1) {
        continue;
      }
      t[u].link = (v == 0 ? 0 : t[t[v].link].to[c]);
      for (int d = 0; d < alphabet.size(); ++d) {
        if (t[u].to[d] != -1) {
          continue;
        }
        t[u].to[d] = t[t[u].link].to[d];
      }
      q.push(u);
    }
  }
}
```

Note. Раскрытие вершины — рассмотрение детей вершины!

Корректность алгоритма следует из корректности поиска в ширину. Асимптотика: $O(|V|\cdot|\Sigma|)$. Лучше нельзя в силу того, что нужно посчитать значения $to[v][c]$.

Решение задачи про суммарное число вхождений

Решение: Строим бор на $s_1, \ldots, s_n$, затем применяем алгоритм Ахо-Корасик. Читаем текст t. $v_0 = 0, to[v_0][t_0] = v_1, to[v_1][t_1] = v_2$ и так далее. Текущая вершина v_j соответствует самому длинному суффиксу строки $t_0 \ldots t_j$, который есть в боре. Каждое вхождение соответствует пути до терминальной вершины по суффиксным ссылкам. Если мы хотим посчитать число новых вхождений, то нужно предпосчитать, сколько встретится терминальных вершин на пути к суффиксным ссылкам, что делается за линейное время.

Лекция 3. Суффиксный массив

Решение задачи о суммарном числе вхождений слов в текст

Продолжаем. Пусть строка t состоит из k символов. Тогда $v_0 = root$, $v_{j+1} = to[v_j][t_{j+1}]$.

Утверждение. Для любого j v_j соответствует самому длинному суффиксу строки $t_1 \dots t_j$, который есть в боре.

Доказательство: Индукция по j. База: $j = 0$, тогда строка пуста, так как v_0 соответствует корню. Переход: пусть верно для v_j, докажем для v_{j+1}. По предположению индукции известен самый длинный суффикс строки $t_1, \dots, t_j$, который есть в боре, соответствующий вершине v_j. Приписываем новый символ t_{j+1}. Если внезапно суффикс увеличится, то при отбрасывании t_{j+1} получится более длинная строка, которая уже есть в боре, и может быть длиннее, чем v_j, но такого не может быть по предположению индукции. Левая граница не сможет уйти за границы самого длинного суффикса при рассмотрении v_j, иначе придём к противоречию, и новый самый длинный суффикс при рассмотрении $v_j + t_{j+1}$ равен $u = to[v_j][t_{j+1}]$, что и требовалось доказать. ■

Вхождения словарных слов, появляющихся в момент времени j ($t_1 \dots t_j$), целиком содержатся в v_j, и интересуют нас только символы из v_j, знаем, что все потенциальные суффиксы возникают с помощью суффиксных ссылок. Если нас интересует количество словарных слов, то нас интересует количество терминальных вершин, встречающихся у нас на пути по суффиксным ссылкам. Так и делаем: к ответу в момент j прибавляем количество терминальных вершин, встречающихся у нас на пути по суффиксным ссылкам из v_j. Чтобы улучшить асимптотику и обрабатывать каждую вершину за $O(1)$, предпосчитаем количество терминальных вершин, встречающихся на пути по суффиксным ссылкам, через поиск в ширину. Если текущая вершина не является терминальной, то такое количество в точности равно количеству терминальных вершин, встречающихся на пути суффиксным ссылкам из вершины, куда можно пойти из текущей по суффиксной ссылке. Если текущая вершина является терминальной, то к аналогичному количеству прибавляем 1.

Суффиксный массив

Def. Пусть есть строка $s = s_0 \dots s_{n-1}$. Обозначим $s^i = s_i s_{i+1} \dots s_{n-1}$.[1] Суффиксный массив строки s — это такая перестановка чисел от 0 до $n-1$, задающаяся набором чисел p_0, p_1, …, p_{n-1}, такая что $s^{p_0} < s^{p_1} < \dots < s^{p_{n-1}}$.[2]

Так как длины всех суффиксов различны, все неравенства являются строгими.

Def. $LCP\,(s,t)$[3] — длина наибольшего общего префикса s и t.

[1] Как признал Илья Степанов, данное обозначение не является общепринятым.
[2] Суффиксы рассматриваются в прямом лексикографическом порядке.
[3] LCP (англ.: Longest Common Prefix) — наибольший общий префикс.

Введём массив lcp: $lcp_0, lcp_1, \ldots, lcp_n$, где $lcp_i = LCP\left(s^{p_i}, s^{p_{i+1}}\right)$.

Note. $lcp_i \neq LCP\left(s^i, s^{i+1}\right)$, $lcp_i = LCP\left(s^{p_i}, s^{p_{i+1}}\right)$.

Утверждение. Пусть $p_l = i$, $p_r = j$, $l < r$. Тогда $LCP\left(s^i, s^j\right) = \min\limits_{k \in [l, r-1]} lcp_k$.

Доказательство: Если $p_l = i$, $p_r = j$, $l < r$, то это означает, что два суффикса s^i и s^j после сортировки в прямом лексикографическом порядке располагаются на местах l и r соответственно.

Рассмотрим суффиксы, располагающиеся после сортировки на местах между l и r. Считаем значения LCP двух суффиксов, являющихся соседними после сортировки, уже известными. Пусть $\min\limits_{k \in [l, r-1]} lcp_k = x$. Сначала поймём, что у всех суффиксов, начиная с s^i, заканчивая s^j, первые x символов одинаковы. Это следует из того, что $LCP\left(s^p, s^q\right)$, $p, q \in [l, r]$, $p \neq q$ не меньше x.

Если $LCP\left(s^i, s^j\right)$ равен хотя бы $x + 1$, то существует следующий за x символами некоторый другой символ, обозначим его за c. Но тогда для любого суффикса, начиная с s^i, заканчивая s^j, символ c должен точно так же следовать за предыдущими x символами. Пусть в некотором другом суффиксе между s^i и s^j за x символами стоит некоторый другой символ d, тогда был бы нарушен прямой лексикографический порядок, аналогично в случае с символом b, а ещё в случае, когда этот суффикс состоит ровно из x символов, и приходим к противоречию. ■

Чтобы за $O(1)$ ответить на запрос вида $LCP(i, j)$, пригодится структура данных, которая позволит это сделать за $O(n \log n)$ предпосчёта — Sparse Table.

Применение суффиксного массива

Суффиксный массив можно использовать в задаче, когда поступает много запросов на равенство подстрок. Можно попробовать использовать хеши, ответ будет почти всегда верен, но вероятность ошибки положительна. Поэтому лучше делать детерминировано: с помощью суффиксного массива. Заметим, что если $r_i - l_i \neq r_2 - l_2$, то строки не равны. Иначе найдём $k = LCP\left(s^{l_1}, s^{l_2}\right)$ за $O(1)$, и если $k \geqslant r_1 - l_1 + 1$, то подстроки равны, иначе они снова не равны.

Алгоритм построения суффиксного массива

Преобразуем строку s в $s\#$, где $\#$ меньше всех символов c лексикографически. Зациклим $s\#$, то есть если при рассмотрении суффикса попали на $\#$, то можем продолжить читать слово сначала, и можно рассматривать подстроки циклического слова длины всего исходного слова. Далее будем сортировать циклические суффиксы. Это будет тем же самым, что и сортировка суффиксов исходного слова, что и позволяет символ $\#$, код которого меньше кода всех остальных символов. Поймём, что это так, рассмотрев наибольший общий префикс двух подстрок. Если первый символ после наибольшего общего префикса в первой строке лексикографически меньше первого символа после наибольшего общего префикса во второй строке, то первая строка лексикографически меньше второй строки, и дальнейшие символы уже не так важны. Если наибольший общий префикс таков, что в одной из

строк следующим символом является символ #, то эта строка однозначно меньше другой строки.

Пример:

Номер первого символа	abacaba	abacaba#
—	—	#abacaba
6	a	a#abacab
4	aba	aba#abac
0	abacaba	abacaba#
2	acaba	acaba#ab
5	ba	ba#abaca
1	bacaba	bacaba#a
3	caba	caba#aba

На k-м шаге получим сортировку всех циклических подстрок длины 2^k, а также их разбиение по классам эквивалентности.

Пример: Рассмотрим ту же строку $s\# = abacaba\#$. Пусть $k = 0$. Тогда:

Номер первого символа	Символ	Класс эквивалентности
7	#	0
0	a	1
2	a	1
4	a	1
6	a	1
1	b	2
5	b	2
3	c	3

Пусть p_i — начальный индекс подстроки длины 2^k, идущей i-й в порядке сортировки, c_i — номер класса эквивалентности строки $s_i s_{i+1} \dots s_{i+2^k-1}$. Здесь можем применить сортировку подсчётом.

```
cnt[256];  // zeroes
for i = 0...n-1
  ++cnt[s[i]];
for i=1...255
  cnt[i]+=cnt[i-1];
for i=n-1...0
  p[--cnt[s[i]]] = i;
```

Далее строим массив c. Пусть известны массивы p и c на k-й итерации.

Note. Каждая подстрока длины 2^{k+1} распадается на подстроки длины 2^k, конкатенация которых составляет ту же самую подстроку длины 2^{k+1}, в каждой из которых есть свой класс эквивалентности

x и y соответственно. Равенство двух подстрок длины 2^{k+1} означает равенство классов эквивалентности их двух непересекающихся подстрок длины 2^k.

Чтобы отсортировать подстроки, достаточно отсортировать пары (c_0, c_{2^k}), (c_1, c_{1+2^k}), (c_2, c_{2+2^k}) и так далее. Такую сортировку тоже можно сделать за $O(n)$ с помощью поразрядной сортировки: сначала по второму элементу пары, а затем стабильно, то есть с сохранением исходного порядка, по первому элементу пары. Кроме того, каждое значение c_i принадлежит отрезку $[0, n-1]$. Действительно, наибольшее количество классов эквивалентности может быть достигнуто в случае, когда каждый символ строки является уникальным.

За p' обозначим полученную сортировку всех подстрок длины 2^{k+1}. Сортировка тоже будет идти по классам эквивалентности.

Асимптотика алгоритма

Асимптотика: $O(n \log n)$. Sparse Table строится за $O(n \log n)$, в сортировке пар классов эквивалентности $\log n$ шагов, и сортировка пар происходит с помощью поразрядной сортировки, которая выполняется за $O(n)$.

Построение массива lcp (алгоритм Касаи)

Обозначим за pos_i такое j, что $p_j = i$, то есть j — такой индекс в суффиксном массиве, которому соответствует суффикс s^i. Таким образом, получится обратная перестановка к p. Найдём наивным алгоритмом lcp_{pos_0}. Заметим, что $pos_{j+1} > pos_1$ при $l \geqslant 1$.

```
l = 0;
for w = 0..n-1 {
  i = pos[w];
  if (i == n-1) {
    l = 0;
    continue;
  }
  j = p[i + 1];
  l = max(l - 1, 0);
  while (w + l < n && j + l < n && s[w + l] == s[j + l]) {
    ++l;
  }
  lcp[i] = l;
}
```

Нужно показать, почему величина l уменьшается максимум на 1. Если l равно 1, то и доказывать нечего, так как $\max(0, 0) = 0$. Предположим, что $l \geqslant 2$, то есть наименьший общий префикс двух предыдущих строк был длины хотя бы 2. Обозначим эти строки за s^{w-1} и s^u. Строки, полученные из них путём удаления первого символа, обозначим за s^w и s^{u+1} соответственно. Тогда s^w лексико-

графически меньше s^{u+1}, как и s^{w-1} относительно s^u, так как то же отношение выполняется для символа строки s^{w-1}, следующего за наибольшим общим префиксом, относительно символа строки s^u, расположенного аналогичным образом. Однако между строками s^w и s^{u+1} могут расположиться ещё строки. Но нам нужна именно длина наибольшего общего префикса между s^w и следующей за ней строкой, и она хотя бы $l-1$ по утверждению об $LCP\,(s^i, s^j)$.

Если на самом деле длина наибольшего общего префикса больше, то увеличим его, пока символы совпадают.

Асимптотика: $O\,(n)$. Это следует из того, что каждая итерация $while$ увеличивает значение l хотя бы на 1, а каждая итерация внешнего цикла уменьшает значение l максимум на 1. Но тогда суммарно изменений не больше, чем $O\,(n)$.

Лекция 4. Суффиксное дерево

Постановка задачи. Сжатый бор

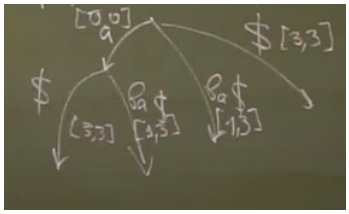
Пусть s — строка, оканчивающаяся на \$. Пусть $s^0, s^1, \ldots, s^{n-1}$ — все её суффиксы. Хотим построить сжатый бор на этих строках.

Def. «Проходная» вершина — вершина бора с исходящей степенью 1 и входящей степенью 1.

Def. Сжатый бор — бор, где пути только по проходным вершинам объединены в переход по соответствующему подслову (подстроке) s, и из каждой вершины не могут выходить два и более рёбер по подсловам (подстрокам), начинающиеся на одну и ту же букву.

Как хранить такие подстроки? Их можно хранить как множество из двух индексов $[l, r]$.

Пример: Пусть $s = aba\$$.



Закономерность роста числа вершин и рёбер

Утверждение. В суффиксном дереве строки длины n всего $O(n)$ вершин и $O(n)$ рёбер.

Доказательство: Построим простейший алгоритм для построения суффиксного дерева за $O(n^2)$. Каждый новый суффикс «идёт» по рёбрам сжатого бора, если требуется новое ребро, то расщепляется текущее ребро, создаётся новая вершина, откуда появляется ребро по оставшемуся суффиксу

суффикса в ещё одну новую вершину. Будет на каждом шаге добавлено не более 2 рёбер и 2 вершин, соответственно, с каждой итерацией их число растёт линейно. ■

Заметим, что количество рёбер и вершин растёт линейно. Значит, есть надежда, что возможен алгоритм, строящий суффиксное дерево за $O(n)$.

Позиции в дереве. Суффиксные ссылки

Посмотрим, какие есть позиции в дереве:

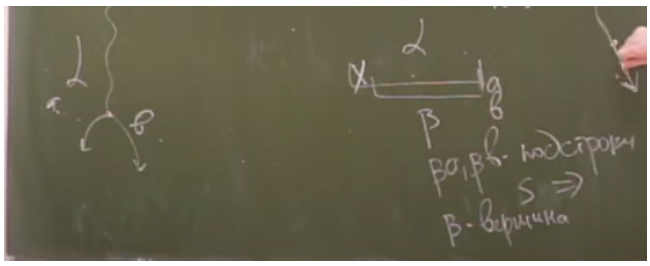
1. Вершины;
2. Родительская вершина ребра;
3. Первая буква ребра;
4. Количество прочитанных символов

Суффиксные ссылки такие же, как в алгоритме Ахо-Корасик: они так же указывают на самый длинный собственный суффикс, который есть в боре. Однако применительно к суффиксному дереву есть одна особенность: они являются указателями на позицию в дереве, полученную отбрасыванием первого символа.

Утверждение. Суффиксная ссылка вершины — вершина.

Доказательство: Вершина может либо быть листом, либо не быть. Так как суффиксы заканчиваются на фиктивный символ \$, обозначающий конец слова, то вершины, соответствующие суффиксам, являются листьями. Так как суффиксные ссылки получаются «отбрасыванием» первого символа строки, соответствующей пути из корня до вершины, то суффиксная ссылка вершины, соответствующей суффиксу, соответствует суффиксу, а каждый суффикс соответствует листу, если он непуст, или корню, если он пуст.

Нелистовые вершины являются так или иначе ветвлениями, то есть у них есть «продолжения» в другие вершины, которых хотя бы две по построению. Суффиксная ссылка для листа — либо лист, либо корень.



Сверху иллюстрация того, как выглядят суффиксные ссылки для нелистовых вершин. Пусть нелистовая вершина такова, что пути из корня до неё соответствует строка α, тогда есть хотя бы два

способа продолжить α: αa и αb. Пусть β — строка, полученная отбрасыванием первого символа из α. Так как αa и αb являются подстроками исходной строки, то βa и βb тоже её подстроки. Но тогда если прочитали строку β, то её тоже можно продолжить до βa или βb, значит, существует вершина, соответствующая строке β, так как происходит расщепление при построении. ■

Для нахождения суффиксной ссылки для произвольной позиции в дереве создадим функцию $GetLink$. Если позиция соответствует вершине, то суффиксная ссылка известна. Иначе позиция задаётся родительской вершиной ребра p, первой буквой ребра a и количеством прочитанных символов k. Возможна неприятность, что придётся пройти не одно, а много рёбер. Рассмотрим вершину p. Возможны два случая:

1. Пусть p не является корнем. Тогда перейдём по суффиксной ссылке из p, и из найденной вершины спустимся по пути $s_l, s_{l+1}, \ldots, s_{l+k-1}$, возможно, придётся менять рёбра.
2. Пусть p является корнем, и есть путь из корня по $s_l, s_{l+1}, \ldots, s_{l+k-1}$. Тогда отбросим s_l и прочтём подстроку $s_{l+1}, \ldots, s_{l+k-1}$ из корня.

Алгоритм Укконена

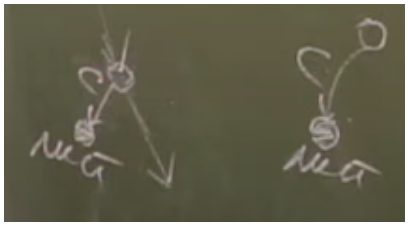
Пусть c — новый символ. Суффиксы, начиная с наибольшего и заканчивая наименьшим, будут делиться на 3 типа:

1. Блок листьев (I);
2. Блок вершин, не являющихся листьями, из которых нет перехода по c (II);
3. Блок вершин, не являющихся листьями, из которых есть переход по c (III).

Рассмотрим наименьший суффикс, являющимся листом. Тогда и все большие суффиксы — листы. Если какой-то больший суффикс — не лист, то из него есть хотя бы ещё два ребра, но тогда это должно выполняться и для наименьшего суффикса, который является листом.

Вершины I-й категории — листья на всех последующих итерациях. Поэтому дописываем все символы до \$, и больше не смотрим на эти вершины.

С вершинами II-й категории возможны две ситуации. В первой ситуации расщепляется ребро, в второй ситуации просто создаётся новая вершина. Каждая из новых вершин станет листом.



Как будем их обрабатывать? Создадим переход по c. Возможно, придётся для этого расщепить ребро. Допишем всё до \$, и больше их не рассматриваем.

Как перебирать суффиксы II-й категории? Для этого достаточно хранить самый длинный суффикс II-й категории $curr$ и брать $GetLink$ до тех пор, пока не придём в вершину, из которой есть переход по c.

Реализация перебора суффиксов

```
for c in s {
  while (!HasTrans(c)) {   // there is no transition from c
    CreateTrans(curr, c, $);
    curr = GetLink(curr);
  }
  curr = go(curr, c); // longest non-leaf suffix
}
```

В конце проставляем терминальность: вся s и все её суффиксные ссылки.

Note. (про сохранение суффиксных ссылок). Это работает, если для каждой вершины известна суффиксная ссылка. Но при создании новых вершин тоже находим их суффиксные ссылки.

Суммарно итераций цикла $while$ будет $O(n)$, так как каждая итерация переносит вершину в I-ю категорию.

Поймём, почему это так. Пусть $depth(v)$ — это количество рёбер от корня до v. Посмотрим, как $GetLink(curr)$ влияет на $depth(curr)$. Сначала $depth$ уменьшается на 1 при (curr $\to$ p). Затем изменяем p на $link(p)$; тогда глубина не может измениться меньше, чем на -1. В конце глубина увеличится на 1. Тогда изменений глубины $O(n)$.

Почему второй пункт верен? Дело в том, что суффиксная ссылка для p получается отбрасыванием первого символа. Посмотрим на соответствующий несжатый бор. Если на пути до p хотя бы x промежуточных вершин, то на пути до $link(p)$ их хотя бы $x-1$, и между ними есть соответствие. Значит, $depth(link(p)) \geqslant depth(p) - 1$.

Note. Если алфавит $|\Sigma|$ не является константным, то потребуется $O(n|\Sigma|)$ памяти и $O(nlog|\Sigma|)$ времени при использовании массива и красно-чёрного дерева.

Лекция 5. Суффиксный автомат

Обсудим, что было ранее. Для строки s мы строили суффиксное дерево, и мы могли быстро ответить на запрос, является ли t подстрокой s, так как любая подстрока по сути является префиксом суффиксов. Читаем t с корня. Если смогли прочитать все буквы t_i, то t — это строка. Когда у нас много запросов, именно суффиксно дерево предпочтительнее алгоритма KMP.

Введение в конечные автоматы

Работаем с детерминированными конечными автоматами (ДКА), которые, проще говоря, являются графами, ориентированные рёбра которых соответствуют буквам, по которым возможен переход, и из одной вершины не могут быть хотя бы 2 ребра, соответствующие одной и той же букве перехода. Кроме того, среди вершин таких графов выделяют стартовое состояние q_{start} и несколько терминальных состояний. С их помощью довольно просто обрабатывать строку: со стартового состояния читаем по порядку каждый её символ. Если с помощью автомата возможно обработать строку, то эта строка принимается автоматом, иначе не принимается.

Def. Пусть A — автомат. Тогда $L(A)$ — множество всех принимаемых автоматом A слов.

Нашей целью будет построить по строке s построить минимальный ДКА A, такой что $L(A)$ — множество всех суффиксов строки s, включая пустое слово ε.

Def. Пусть L — язык, $x,\ y$ — слова. Тогда $R_L(X) = \{z \mid xz \in L\}$ — правый контекст x относительно языка L.

Def. $x \sim_L y$, если $R_L(x) = R_L(y)$.

Def. $[x]_L$ — класс эквивалентности слова x.

Теорема Майхилла-Нероуда

Th. (Майхилла-Нероуда) Пусть L — язык, причём число множество классов эквивалентности всех возможных слов $\{[x]_L\}_x\ \forall x \in L$ конечно. Тогда:

1. Любой автомат A, принимающий язык L, содержит хотя бы k вершин (состояний), где k — мощность множества классов эквивалентности всех возможных слов;
2. Существует автомат A, принимающий язык L и содержащий ровно k вершин (состояний).

Суффиксный автомат

Начиная с этого момента, будем рассматривать только строки s, а также L будет языком на всех суффиксах s.

Утверждение. Пусть $R_L(u) = R_L(v)$. Тогда либо u — суффикс v, либо v — суффикс u.

Доказательство: Существует хотя бы одно слово w такое, что либо $uw \in L$ и является суффиксом s, либо $vw \in L$ и является суффиксом s, то тогда u и v имеют общую позицию последнего вхождения их символа, и одно слово является суффиксом другого. ■

Утверждение. Пусть C — класс эквивалентности. Тогда C является некоторой строкой, и несколько её самых длинных суффиксов.

Доказательство: Пусть u — самая длинная строка в C. Тогда остальные строки из C — её суффиксы. Почему в C лежит отрезок суффиксов? Пусть $v \in C$. Тогда $R_L(v) = R_L(u)$. Пусть w — суффикс, причём $|w| > |v|$. Тогда $R_L(w) \subset R_L(v)$, так как v — суффикс w. Аналогичным образом $R_L(u) \subset R_L(w)$. Значит, $R_L(u) \subset R_L(w) \subset R_L(v)$, откуда $R_L(w) = R_L(u)$, и $w \in C$. ■

Введём обозначения: $[x]_L$, $R_L \leftrightarrow [x]_S$, R_S, $longest(C)$ — самая длинная строка в C, $len(C) = |longest(C)|$, $link(C)$ — суффиксная ссылка, устроенная следующим образом: $link(C) = [x]$, где x — самый длинный суффикс $longest(c)$, который не лежит в c.

Note. $|C| = len(C) - len(link(C))$.

Утверждение. (Критерий того, что строка является *longest*). Пусть u — подстрока s. Тогда $u = longest([u]_s)$ тогда и только тогда, когда либо u — префикс s, либо существуют различные a и b, что au, bu — подстроки s.

Доказательство:

$\Longrightarrow$ Пусть $u \neq longest([u])$. Но тогда $u \sim cu$, $R(u) = R(cu)$, откуда u и cu имеют одинаковое множество вхождений в s.

$\Longleftarrow$ Пусть $u = longest([u])$, u — не префикс s. Рассмотрим все символы перед вхождениями u. Нужно доказать, что среди них есть хотя бы 2 раза. Если это неверно, то они равны d, тогда $u \sim du$, и $|du| > |longest([du])|$. Противоречие. ■

Алгоритм построения суффиксного автомата

Будем соблюдать несколько принципов:

1. Инкрементальный подход;
2. Не следим за терминальностью. Проставим её в самом конце.

Пусть для строки S построен суффиксный автомат. К S приписывается символ c, и автомат нужно перестроить. Классами эквивалентности у нас будут представители *longest*. Как меняется множество самых длинных строк в своих классах? Если u было *longest*, то есть $u = longest([u]_S)$, то u останется *longest*. Значит, могут только появиться новые самые длинные строки, и таковой будет обязательно Sc, $Sc = longest([Sc]_{Sc})$. $[Sc]_{Sc}$ — множество всех суффиксов Sc, не являющихся подстроками S.

Пусть T — ещё одна новая самая длинная строка *longest*. Тогда T — подстрока S, так как все появившиеся слова, не являющимися подстроками S, уже были рассмотрены, они принадлежат $[Sc]_{Sc}$. Раз T стало *longest*, то T — суффикс Sc, перед которым стоит $y + x$. Тогда yT — не подстрока s. Отсюда вывод: единственный кандидат на роль T — это самый длинный суффикс Sc, который является подстрокой S, и новых состояний будет не больше двух.

Обозначим за S_0 наибольший суффикс S_c, который является подстрокой S. $link([Sc]_S c) = Sc_S c$.

Утверждение. Пусть v — вершина (состояние) автомата. Рассмотрим все вершины (состояния) $u_1, \dots, u_k$, по которым можем перейти в вершину (состояние) v по одному переходу. Тогда:

1. Каждый из переходов из u_i в v соответствует одной и той же букве;
2. $link\,(u_i) = u_{i+1}\ \forall i$

Note. $[\varepsilon] = \{\varepsilon\}$ соответствует корневой вершине.

Доказательство: Рассмотрим u_k, из которой возможно перейти в v по букве c, но единственный способ получить одно из слов, которым соответствует состояние v — сделать конкатенацию u_k и c. То же и для других классов эквивалентности. ■

Лекция 6

Продолжение про суффиксный автомат

Считаем, что для строки S уже построен суффиксный автомат. Посмотрим для Sc. Возникает класс эквивалентности $[Sc]$, и поняли ранее, что S_0 — наибольший суффикс Sc, который является подстрокой S. Также для некоторой вершины v верно, что на всех рёбрах, ведущих в неё, написана одна и та же буква, кроме того, они части одного и того же суффиксного пути. А теперь поймём, как будут меняться рёбра при добавлении буквы c.

Утверждение. После добавления c большинство рёбер в старом автомате, соответствующем S, сохранятся. Потенциальные изменения:

1. Рёбра в $[Sc]$ (из этого состояния рёбер точно не будет, иначе была бы более длинная строка);
2. Если S_0 — новый «longest», то состояние q расщепилось, могут измениться в рёбра в состояние q (они могут вести в другое состояние), а также нужно понять, откуда теперь ведут рёбра из q.

Больше ничего не изменится, так как неучтёнными останутся рёбра между вершинами (состояниями), которые не изменились.

Теперь поймём, какие рёбра нужно провести в $[Sc]$. На них должна быть написана одна и та же буква c. Скажем тогда, что мы должны провести ребро из строки класса $[u]$ в класс $[Sc]$, на котором буква c, если uc — суффикс Sc, и uc при этом не подстрока S. Пусть мы хотим пойти из $[S]$, перемещаемся по суффиксным ссылкам, и так дошли до некоторой вершины p. Самый простой случай: дошли до корня.

Код:

```
while (p != -1 && p.trans[c] == nullptr) {
    to[p][c] = Sc;
    p = link[p];
}
```

Разбор случаев:

1. Если из корневой (стартовой) вершины нет перехода по c ($p == 1$). Тогда из S переходим по суффиксным ссылкам по корня, $S_0 = \varepsilon$ и c не встречался в S. Действительно, это может быть только в том случае, если c — символ, не встречавшийся ранее и в строке, и в автомате. Кроме того, ранее мы заметили, что $link([Sc]) = [S_0]$, и суффиксная ссылка новой вершины будет вести в корень;

2. Если мы по суффиксным ссылкам дошли до вершины, из которой есть переход по c. Пусть p — первая найденная такая вершина. Тогда утверждаем, что из p по c можно перейти в q, которому принадлежит S_0. Тогда $S_0 = longest(p) + c$. Если S_0 не было $longest$, то он станет им, и есть шанс его расщепить. Не нужно расщеплять q, если и только если S_0 была $longest(q)$, и $len(q) = len(p) + 1$. Чтобы память не росла квадратично, храним лишь длины $longest$. Тогда состояния не меняются. Обрабатывать будем просто: скажем, что $link(Sc) = q$, и на этом закончим. Вспоминаем, что нужно проводить и как: из u проведём по c ребро в Sc, если uc — не подстрока S, и u — суффикс S.

3. Теперь разберём случай, когда из p есть переход по c в вершину q, но $len(q) > len(p) + 1$. Тогда состояние q придётся расщепить. Тогда можно представить строку, соответствующую q, и её несколько длинных суффиксов по убыванию длины. S_0 не является самым длинным из них. Что нужно сделать? S_0 будет новым $longest$, и первое новое состояние q будет соответствовать суффиксам, длиннее S_0, второе новое $clone$ — остальным суффиксам. Рёбра могли перенаправиться: либо в оставшееся q, либо в $clone$. Рассмотрим суффиксный путь. Для вершин, соответствующим более длинным суффиксам строки S, чем p, ничего не изменится, исходящие рёбра, ведущие в q, сохранятся. А вот для остальных ситуация изменится вот так:

   ```
   while (p != -1 && to[p][c] == q) {
       to[p][c] = clone;
       p = link[p];
   }
   ```

   Что делать с рёбрами, исходящими из q и $clone$? Прежде всего заметим, что $R_{Sc}(clone) = R_{Sc}(q) \sqcup \{\varepsilon\}$. Нужно понять, как отличаются множества вхождений строк, соответствующим классам $[q]$ и $[clone]$. $longest(q)$ не будет суффиксом Sc, и множество вхождений $longest(q)$ в Sc совпадает с множеством вхождений в S. А вот у S_0 появилось одно новое вхождение. Для q Sc — не терминальная строка, для $clone$ Sc — терминальная строка. Этим и отличается q от $clone$ в плане исходящих рёбер. В итоге $to[clone] = to[q]$. Теперь проставляем ссылки. $len(clone) = len(p) + 1$, $link(clone) = link(q)$, $link(q) = clone$, $link([Sc]) = clone$.

Асимптотика

Th. (без доказательства) В суффиксном автомате, построенном по строке длины n, не более чем $(2n-1)$ вершин и не более $(3n-4)$ рёбер начиная с некоторого n_0.

Асимптотика: если считать алфавит константным, то $O(n)$, если алфавит считать непостоянным, то $O(n \log |\Sigma|)$ при использовании красно-чёрного дерева, перенаправлений рёбер: $O(n)$. На каждом шаге алгоритма добавляется либо одна, либо две вершины, а также добавляются рёбра в новую вершину. Суммарно добавлений рёбер будет $(3n-4)$. Разберём несколько случаев:

1. В случае, когда по суффиксным ссылкам дошли до корня, и из корневой вершины не было перехода по букве c, алгоритм дальше ничего не делает. Рёбра учтены.

2. В случае, когда нашлась такая вершина p, что из неё есть переход в вершину q по букве c, и $len\,(q) = len\,(p) + 1$, алгоритм дальше тоже ничего не делает. Рёбра учтены.

3. Самое сложное начинается в случае, когда из p есть переход по c в q, но $len\,(q) > len\,(p) + 1$, и q приходится расщеплять. Тогда перенаправление старых рёбер уже не учитывается, и этот момент нужно анализировать более тонко. Проанализируем путь по суффиксным ссылкам из вершины S, а также аналогичный путь из вершины S_c, который проходит через вершину *clone*. Чем больше перенаправили рёбер, тем меньше вершин на пути по суффиксным ссылкам. Путь по суффиксным ссылкам из S до корня разбивается на блоки в зависимости от того, в какую вершину на пути из S_c до корня ведёт перенаправленное ребро. Перенаправление рёбер из q до *clone* работает пропорционально размеру блока. Но тогда чем больше вершин в блоке, тем меньше вершин на пути из S_c до корня через *clone*. Пусть в блоке вершин, перенаправленные рёбра которых ведут в *clone*, k вершин, и на пути из S до корня по суффиксным ссылкам m вершин. Тогда количество вершин на пути из S_c до корня через *clone* не превосходит $m+2-k$. Так как новых вершин не более двух, но k хотя бы 1, то количество итераций такого цикла $O\,(n)$. Значит, суммарно перенаправлений тоже будет $O\,(n)$

Реализация

Каждая вершина обладает полями *len*, *link*, *to*, *last* (отвечающее длиннейшему суффиксу, построенному на текущий момент).

```
vector<node> t;

void add(char c) {
    t.push_back(node());
    curr = t.size() - 1;
    p = last;
    while (p != -1 && t[p].to[c] == -1) {
        t[p].to[c] = curr;
```

```
        p = t[p].link;
    }
    if (p == -1) {
        t[curr].link = 0;
        last = curr;
        return;
    }
    q = t[p].to[c];
    if (t[q].len == t[p].len + 1) {
        t[curr].link = q;
        last = curr;
        return;
    }
    t.push_back(node());
    clone = t.size() - 1;
    while (p != -1 && t[p].to[c] == q) {
        t[p].to[c] = clone;
        p = t[p].link;
    }
    t[clone].to = t[q].to;
    t[clone].len = t[p].len + 1;
    t[clone].link = t[q].link;
    t[q].link = clone;
    t[curr].link = clone;
    last = curr;
}

main() {
    t.push_back(node());
    return 0;
}
```

Задача

Есть шаблон p и текст t, позволяется не более k ошибок, нужно найти все неточные вхождения с не более чем k ошибками, и они могут идти, где угодно.

Для этого нужно FFT — быстрое преобразование Фурье.

Теория чисел

Алгоритм Евклида

Задача. Есть два числа a, b. Нужно найти их наибольший общий делитель.

Решение: Заметим, что $(a, b) = (a, b - a)$, если $b \geqslant a$, это доказать вполне несложно, тем более множество делителей совпадает. Будем брать по модулю до тех пор, пока не придём к более очевидному. Обобщим утверждение: $(a, b) = (a, b\%a)$, если $b \geqslant a$.

```
int gcd(int a, int b) {
    if (b < a) {
        swap(a, b);
    }
    if (a == 0) {
        return b;
    }
    return gcd(a, b % a);
}
```

Время работы: $O(\log \max(a, b))$. Оценим, как быстро убывают числа. Пусть $(a, b) \mapsto (b\%a, a)$. Разберём два случая:

1. Если $b \geqslant 2a$, то $b\%a < a \leqslant \frac{b}{2}$.
2. Иначе $b \in [a, 2a - 1]$, тогда $b\%a = b - a$, $a \geqslant \frac{b}{2}$, откуда $b\%a \leqslant b - \frac{b}{2} = \frac{b}{2}$.

На каждой итерации результат уменьшается хотя бы вдвое, отсюда асимптотика $O(\log \max(a, b))$.

Расширенный алгоритм Евклида

Задача. Пусть $(a, b) = d$. Нужно найти такие x, y, что $ax + by = d$ (то есть ищем линейное представление НОД).

Решение: Пусть найдены $\hat{x}$ и $\hat{y}$, такие что $(b\%a) \cdot \hat{x} + a \cdot \hat{y} = d$. Распишем подробнее: $(b\%a) = b - a \cdot floor\left(\frac{b}{a}\right)$, $b \cdot \hat{x} + a \cdot \hat{y} - a \cdot floor\left(\frac{b}{a}\right) \cdot \hat{x} = d$, $a\left(\hat{y} - floor\left(\frac{b}{a}\right) \cdot \hat{x}\right) + b \cdot \hat{x} = d$.

Упражнение. Покажите, что (x, y) — одно из решений $(x + kb, y - ka)$, $k \in \mathbb{Z}$.

Лекция 7. Теория чисел

Th. (без доказательства) Пусть $\pi(n)$ — количество простых чисел среди натуральных $(1, \ldots, n)$. Тогда $\pi(n) = \frac{n}{ln\ n} + O\left(\frac{n}{ln^2\ n}\right)$.

Th. (без доказательства) Сумма по простым числам p, не превосходящих n, чисел вида $\frac{1}{p}$ равна $ln\ ln\ n + M + O\left(\frac{1}{n}\right)$, где M приблизительно равно $0,261$.

Th. (без доказательства) $\sum\limits_{k=1}^{n} \frac{1}{k} = ln\ n + \gamma + O\left(\frac{1}{n}\right)$, где γ (константа Эйлера) приблизительно равна $0,577$.

Решето Эратосфена

Задача. Найти все простые среди чисел $1, 2, \ldots, n$.

У нас есть простой алгоритм проверки x на простоту за $O(\sqrt{n})$: перебираем возможные делители от 1 до $\sqrt{n}$, если x не является простым, то найдётся среди них хотя бы один его делитель, иначе x является простым.

Можно сделать лучше, чем за $O(n \cdot \sqrt{n})$. Сначала считаем, что все числа, кроме 1, простые, далее перебираем их в порядке возрастания, если p является простым, то $2p$, $3p$, $\ldots$ не являются простыми.

```
prime[2...n] <- {true};
for p = 2...n {
    if (!prime[p]) {
        continue;
    }
    for j = 2 ... n/p
        prime[j * p] = false;
}
```

Корректность

Корректность следует из того, что если число на самом деле простое, то оно и останется простым, и его не пометим как составное, все составные тоже найдём.

Асимптотика

Для оценки найдём $\sum\limits_{p \leqslant n, p \in \mathbb{P}} \frac{n}{p} = \Theta(n \cdot ln\ ln\ n)$, отсюда и асимптотика $O(n \cdot ln\ ln\ n)$.

Решето Эратосфена за $O(n)$

Оно полезно тем, что для каждого натурального числа будет найден минимальный простой делитель. Пусть $d_{min}(x)$ — минимальный простой делитель числа $x \in [1, \ldots, n]$. Критерий простоты: $d_{min}(x) = x$.

```
vector<int> primes;
mind[2...n] = {2, 3, 4, ..., n};
for k = 2...n {
    if (mind[k] == k) {
        primes.push_back(k);
    }
    for (p : primes && p * k <= n && p <= mind[k]) {
        mind[p * k] = p;
    }
}
```

Далее установим несколько фактов:

1. Если $mind[x]$ для простых x найдётся корректно, то в $primes$ точно лежат все простые.

   Действительно, если x не является простым, $d_{min}(x) = q$, $q < x$. Когда $k = \frac{x}{q}$ и $p = q$, то вспоминим, что $mind[x] = q$, среди простых точно будет q, $p \cdot k \leqslant n$ тоже будет корректным, как и $p \leqslant mind[k]$. Довольно просто установить, что $d_{min}(x) \geqslant q$.

2. $mind[x]$ когда-то найдётся верно, и $mind[x]$ обновится всего лишь однажды. Изменений в массиве $mind$ $O(n)$, асимптотика: $O(n)$.

Факторизация числа

Кроме того, это решето полезно для факторизации: разложения некоторого числа x на простые множители. Разделим x на его минимальный простой делитель, делаем то же самое с полученным числом, и так до тех пор, пока не останется простое число. Это работает за $O(log\ n)$, так как делений максимум $O(log\ n)$. Но вот при разложении произвольного числа, который не был пройден решетом, такой хорошей асимптотики добиться не получится, в лучшем случае будет $O(\sqrt{n})$, что экспоненциально долго: $2^{\frac{1}{2}\log_2 x}$.

Обращение по модулю

Утверждение. Пусть $(a, m) = 1$. Тогда существует такой x, что $ax \equiv 1\,(mod\ m)$. Этот x будет являться обратным к a по модулю m.

Доказательство: НОД можно представить линейно: с помощью расширенного алгоритма Евклида за $O(log\ a + log\ m)$ возможно найти такие x и y, что $ax + my = 1$.

Криптографические протоколы

Ситуация такая: Алиса и Боб хотят обмениваться информацией, между ними есть открытый канал без шума: каким было отправлено сообщение, таким оно и отправится. Однако есть подслушница (earsdropper) Ева, которая пытается перехватить сообщение, используя всего лишь открытый канал.

Существуют несколько криптографических протоколов:

Гаммирование

Алиса и Боб тайно договариваются о секретном ключе $x \in \{0,1\}^n$. Самое простое, что они могут сделать: зайти в аудиторию, сгенерировать случайный ключ, и договориться, что таким ключ и будет. Алиса хочет передать битовое слово $y \in \{0,1\}^n$ Бобу, по каналу передаётся $x \oplus y$. Боб сможет

легко получить y, так как $(x \oplus y) \oplus x = y$, и лишь только он с Алисой знает x, и больше никто, поэтому x трудно восстановить.

Но есть проблемы. Договариваться нужно очно, кодирование в некотором смысле одноразовое, в следующий раз тоже придётся договориться секретно.

Алгоритм Диффи-Хеллмана

Его можно считать некоторым усовершенствованием предыдущего. Можно договариваться на расстоянии. Алиса будет генерировать простое p и первообразный корень по модулю p, равный g. Потребуется знание, что $\{g^0, g^1, \ldots, g^{p-2}\} \underset{mod\ p}{=} \{1, 2, \ldots, p-1\}$. Затем Алиса публикует найденные g и p в открытом доступе. Затем Алиса генерирует случайное a, а Боб генерирует случайное b. Алиса передаёт Бобу g^a, а Боб передаёт Алисе g^b. Алиса знает a и g^b, Боб знает g^a и b, и они оба могут найти g^{ab}, который можно использовать как секретный ключ.

Ева знает g, p, g^a, g^b. Еве хотелось бы дискретно прологарифмировать g^a, решив уравнение вида $g^x \equiv c\,(mod\ p)$, чтобы получить a и возвести g^b в степень a, тем самым ключ мог бы быть найден. Однако задача дискретного логарифмирования довольно сложная, и её никто не умеет решать за полиномиальное время.

Протокол RSA

Представим, что государственный орган публикует открытый ключ, используя который, граждане могут отправлять ему свои заявления, на которые, возможно, нет ответа. А теперь вернёмся к Алисе, Бобу и Еве. Боб генерирует различные простые p и q, затем он публикует в открытый доступ число $N = pq$, а также число e, взаимно простое с $(p-1)(q-1)$. Как работает общение? Если Алиса хочет послать число x Бобу, то она по открытому каналу посылает ему число $x^e\,(mod\ N)$. Для декодирования сообщения Бобу нужно найти d — число, обратное к e по модулю $(p-1)(q-1)$, и вычисляет $(x^e)^d$, что будет равно x по модулю N.

Для того, чтобы показать, что этот алгоритм работает, потребуется теорема Эйлера: если $(a, m) = 1$, то $a^{\varphi(m)} \equiv 1$ по модулю m, откуда $a^{\varphi(m)+1} \equiv a$ по тому же модулю. Здесь $\varphi(m)$ — функция Эйлера от числа m, кроме того, $\varphi(pq) = (p-1)(q-1)$. Поскольку $ed \equiv 1$ по модулю $(p-1)(q-1) = \varphi(N)$, то Боб сможет декодировать сообщение. Он сможет найти функцию Эйлера от N достаточно быстро. А вот сторонний наблюдатель не сможет декодировать сообщение, хотя бы относительно быстро.

Алгоритм Штрассена

Пусть есть есть две матрицы A и B размера $2^k \times 2^k$. Цель: найти $A \cdot B = C$. Наивный алгоритм умножения матриц работает за $\Theta(n^3)$, где $n = 2^k$. Однако можно лучше. Представим A в виде

$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$, где A_{11}, A_{12}, A_{21}, A_{22} — квадратные матрицы равного порядка $2^{k-1} \times 2^{k-1}$. Аналогично представим B и C. А теперь введём 7 следующих матриц:

1. $P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
2. $P_2 = (A_{21} + A_{22}) B_{11}$
3. $P_3 = A_{11}(B_{12} - B_{22})$
4. $P_4 = A_{22}(B_{21} - B_{11})$
5. $P_5 = (A_{11} + A_{12}) B_{22}$
6. $P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
7. $P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

По мастер-теореме $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2) \Longrightarrow T(n) = \Theta\left(n^{\log_2 7}\right)$, что уже лучше наивного алгоритма.

Матрицу C вычислим так:

1. $C_{11} = P_1 + P_4 - P_5 + P_7$
2. $C_{12} = P_3 + P_5$
3. $C_{21} = P_2 + P_4$
4. $C_{22} = P_1 - P_2 + P_3 + P_6$

А теперь покажем, почему $C_{22} = A_{21}B_{12} + A_{22} + B_{22}$. Как сказал Илья Степанов, придётся раскрыть скобки и умереть: $C_{22} = A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} - A_{21}B_{11} - A_{22}B_{11} + A_{11}B_{12} - A_{11}B_{22} + A_{21}B_{11} + A_{21}B_{12} - A_{11}B_{11} - A_{11}B_{12}$. Далее сокращаем, получаем необходимый результат.

Числа Стирлинга первого и второго рода

Рассмотрим многочлены $x(x-1)(x-2)\dots(x-n+1) = \sum\limits_{k=0}^{n} s(n,k)\, x^k$. Числа вида $s(n,k)$ — числа Стирлинга I-го рода.

Пример: $k^2 = k(k-1) + k$, $\sum\limits_{k=0}^{n} k^2 C_n^k = \sum\limits_{k=0}^{n} k(k-1) C_n^k + \sum\limits_{k=0}^{n} k C_n^k$. Подобное возникает при распределении Пуассона, когда случайная величина $\xi \sim Pois(\lambda)$[4], и нужно посчитать $\mathbb{E}\xi^3$.

[4]На самом деле, здесь можно обойтись и без чисел Стирлинга. Если распределение было бы биномиальным, то числа Стирлинга здесь точно бы пригодились.

Утверждение. $|s(n,k)|$ — количество перестановок на n элементах, которые в разложении на циклы имеют ровно k циклов.

Пример: $n=3$, есть перестановки (123), (23), id, (13), (12), (132). При $k=2$ подойдут 3 перестановки: (12), (13), (23).

Для доказательства утверждения нужно доказать рекурренту $s(n+1,k) = -n \cdot s(n,k) + s(n,k-1)$, $k \geqslant 1$; $s(0,0)=1$, $s(n,0)=0$ при $n \geqslant 1$.

Числа Стирлинга второго рода решают обратную задачу, когда нужно выразить x^n как $\sum\limits_{k=0}^{n} x(x-1)\dots(x-k+1) \cdot S(n,k)$.

Лекция 8

Немного об алгоритме Карацубы

В прошлый раз мы обсудили алгоритм Штрассена, который позволяет быстрее наивного алгоритма перемножать матрицы. Аналогичным образом позволяет перемножать многочлены алгоритм Карацубы, асимптотика которого выходит из рекурренты:

$$T(n) = 3T\left(\tfrac{n}{2}\right) + \Theta(n)$$
$$T(n) = \Theta\left(n^{\log_2 3}\right) = O\left(n^{1.59}\right)$$

Мы не будем это подробно разбирать, так как будет разобран алгоритм, позвволяющий перемножать многочлены за $O(n \log n)$.

Быстрое преобразование Фурье

Пусть есть многочлен $P(x) = a_0 + a_1 x + \dots + a_{n-1}x^{n-1}$. Тогда P однозначно восстанавливается по своим значениям в произвольных n попарно различных точках. По сути, это сводится к системе линейных уравнений:

$$\begin{pmatrix} x_1^0 & x_1^1 & \dots & x_1^{n-1} \\ x_2^0 & x_2^1 & \dots & x_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_n^0 & x_n^1 & \dots & x_n^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ \vdots \\ y_n \end{pmatrix}$$

Идея: перемножить P и Q, $R = P \cdot Q$, используя следующие шаги:

1. Найдём $P(x_1), \dots, P(x_n)$ $Q(x_1), \dots, Q(x_n)$ в некоторых точках $x_1, \dots, x_n$;

2. $R(x_i) = P(x_i) \cdot Q(x_i)$

3. Восстановим R по его значениям в $x_1, \ldots, x_n$.

Кроме того, должны выполняться следующие условия:

$$n > deg\ R = deg\ P + deg\ Q$$

$$n \geqslant deg\ P + deg\ Q + 1$$

Здесь $x_1, \ldots, x_n$ — комплексные корни из 1 степени n. Нам поможет представление комплексных чисел на плоскости. Введём $\omega = e^{i \cdot \frac{2\pi}{n}}$, $\omega^n = 1$.

Полезное замечание. Для работы с комплексными числами в языке C++ используйте std::complex.

Первый шаг

Вычислим $P(\omega^0), \ldots, P(\omega^{n-1})$. Кроме того, рассмотрим многочлены:

$$P(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

$$P_0(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3 + \ldots$$

$$P_!(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3 + \ldots$$

Выполняется следующее равенство: $P(x) = P_0(x^2) + x \cdot P_1(x^2)$. Считаем, что n является чётным. Тогда достаточно найти значения P_0 и P_1 в точках $(\omega^0,\ \omega^2,\ \omega^4,\ \ldots,\ \omega^{n-2})$.

Найдём асимптотику: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \Longrightarrow T(n) = \Theta(n \log n)$.

Второй шаг

Пусть уже найдены $P(\omega^0),\ \ldots,\ P(\omega^{n-1})$ и Q в тех же точках. Найдём R по правилу: $R(\omega^j) = P(\omega^j) \cdot Q(\omega^j)$. Асимптотика: $\Theta(n)$.

Третий шаг

Здесь нам пригодится определитель Вандермонда. Введём обозначение:

$$W = \begin{pmatrix} \omega^{0\cdot 0} & \omega^{0\cdot 1} & \ldots & \omega^{0\cdot(n-1)} \\ \omega^{1\cdot 0} & \omega^{1\cdot 1} & \ldots & \omega^{1\cdot(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^{(n-1)\cdot 0} & \omega^{(n-1)\cdot 1} & \ldots & \omega^{(n-1)\cdot(n-1)} \end{pmatrix}$$

Тогда столбец из $P(\omega^0), \ldots, P(\omega^{n-1})$ можно найти, умножив матрицу W на столбец из $a_0, \ldots, a_{n-1}$.

Пусть $R(x) = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$. Тогда столбец из $c_0, \ldots, c_{n-1}$ представим в виде произведения матрицы W^{-1} на столбец из $R(\omega^0), \ldots, R(\omega^{n-1})$.

Пусть $W_{ij} = \omega^{i \cdot j}$, $V_{ij} = (\omega^{-1})^{i \cdot j}$. Тогда $V \cdot W = n \cdot E_n$, где E_n — единичная матрица порядка n. Отсюда $W^{-1} = \frac{1}{n} V$.

Теперь найдём сумму: $\sum\limits_{j=0}^{m-1} (\omega^{-1})^{ij} \cdot \omega^{jk} = \sum\limits_{j=0}^{m-1} \omega^{j(k-i)} = \sum\limits_{j=0}^{m-1} \omega^{(k-i)j}$. Эта сумма тождественно равна нулю, что можно понять из суммы бесконечно убывающей геометрической прогрессии, положив знаменатель q равным ω^{k-i}. Эта арифметическая выкладка нужна была для того, чтобы понять, как находить W^{-1}.1

Основная алгоритмическая часть. Запускаем первый шаг, положив вместо ω ω^{-1}. Считаем результат, делим все числа на n. Это будет обратным преобразованием Фурье. Асимптотика: $\Theta(n \log n)$.

В итоге задача решена за $\Theta(n \log n)$, где $n = \Theta(\max(deg\ P,\ deg\ Q))$.

Возможные проблемы

Основная проблема связана с точностью чисел с плавающей точкой, возможны ошибки, которые при накоплении могут вызвать ошибки. Однако на практике алгоритм работает точно, так как если перемножаем целочисленные многочлены, то погрешность невелика, если коэффициенты R не превосходят 10^{11} или 10^{12}.

Разворачивание рекурсии

Так как довольно много непростых операций с комплексными числами, то если алгоритм реализовать рекурсивно, то он будет работать долго и потреблять довольно много памяти. Поэтому требуется его реализовать нерекурсивно.

Посмотрим, как алгоритм будет работать на массиве длины 8: $[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$. В левую половину перемещаем коэффициенты с чётным индексом, в правую половину перемещаем коэффициенты с нечётным индексом, и так до тех пор, пока не будет разбиения на отдельные коэффициенты. Всего получится k новых уровней, не включая исходного, если $n = 2^k$.

Рассмотрим $a_{rev(0)}, \ldots, a_{rev(n-1)}$. Здесь $rev(i)$ — представление i как числа в двоичной системе счисления с k знаками, где запись развернули справа налево. Здесь считаем, что n представим в виде 2^k. Например, $n = 8 = 2^3,\ k = 3,\ rev(3) = 110_2 = 6$.

Попытаемся реализовать пересчёт за $O(N)$:

```
rev[0] = 0;
oldest = -1;
for (int mast = 1; mask < (1 << k); ++mask) {
```

```
  if (!(mask & (mask - 1))) {
    ++oldest;
  }
  rev[mask] = rev[mask ^ (1 << oldest)] | (1 << (k - oldest - 1));
}
```

Пусть мы знаем состояние массива на j-м снизу уровне. Рассмотрим многочлен $T(x) = a_i + a_{i+1}x + \cdots + a_{i+2^{j^{i+1}}-1} \cdot x^{2^{j^{i+1}}-1}$. Его можно найти, зная его разбиение на T_0 и T_1: $T(x) = T_0(x^2) + x \cdot T_1(x^2)$. Пусть ω — корень из 1-й степени. Найдём $T(\omega^0), \ldots, T(\omega^{j-1})$. Считаем $s < \frac{j}{2}$, тогда:

$$T(\omega^s) = T_0(\omega^{2s}) + \omega^s \cdot T_1(\omega^{2s})$$
$$T\left(\omega^{\frac{j}{2}+s}\right) = T_0(\omega^{2s}) - \omega^s \cdot T_1(\omega^{2s}).$$

Реализация (здесь $pow(a, b)$ обозначает a^b, со степенями 2 лучше применять именно битовые сдвиги; кроме того, здесь $root(x, y) = \sqrt[y]{x}$):

```
for (int j = 0; j < k; ++j) {
  omega = root(1, pow(2, j + 1));
  for (int i = 0; i < n; i += pow(2, j + 1)) {
    for (int s = 0; s < pow(2, j); ++s) {
      x = arr[i + s];
      y = arr[i + s + pow(2, j)];
      arr[i + s] = x + pow(omega, s) * y;
      arr[i + s + pow(2, j)] = x - pow(omega, s) * y;
    }
  }
}
```

Отказ от $\mathbb{C}$, переход в $\mathbb{Z}_p$

Если $p = 2^k \cdot r + 1$, где r — нечётное, то существует такой элемент $\omega \in \mathbb{Z}_p$, такое что числа ω^0, ω^1, $\ldots$, ω^{2^k-1} попарно различны. При достаточно большом p можно перенести вычисления в $\mathbb{Z}_p$, и тем самым удастся избежать проблемы с точностью.

Пример: Например, часто в контестах на Codeforces применяется число $p = 998244353 = 119 \cdot 2^{23} + 1$, $\omega = 31$. Значение ω можно найти простым перебором.

Кроме того, это может быть нужно в ситуации, когда коэффициенты многочленов могут превысить 10^{12}, можно рассматривать коэффициенты по модулям p_1 и p_2, и далее можно применить китайскую теорему об остатках, чтобы найти коэффициенты единственным образом.

Лекция 9. Элементарная геометрия. Триангуляции

Элементарная геометрия

Точки и векторы в $\mathbb{R}^2$

В $\mathbb{R}^2$ можно задать декартовую систему координат, обладающей осями абсцисс и ординат. Каждую точку можно задать двумя координатами (a, b), обозначающие проекцию на каждую из осей. Каждой точке можно сопоставить радиус вектор из $(0, 0)$ в (a, b). Для точки заведём структуру point:

```
struct point {
  int x, y;
};
```

Пусть у нас есть вектор $\vec{v}$ из точки p в точку q. Его можно задать как $(q.x - p.x, q.y - p.y)$. Для структуры point можно написать оператор «минус»:

```
point operator - (const point& other) const {
  return {x - other.x, y - other.y}
}
```

Можно также реализовать оператор «плюс», соответствующее сложению векторов, а также оператор «звёздочка» умножения на скаляр.

Прямая

Далее нам пригодится примитив «прямая», и мы хотим, чтобы она однозначно задавалась. Удобно её представлять в виде $y = kx + b$, но есть проблема: вертикальные прямые так не задаются. Можно было бы повернуть на случайный угол α. Но с целыми числами возможна проблема. Поэтому будем представлять прямую в виде $ax + by + c = 0$:

```
struct line {
  int a, b, c;
};
```

Но есть проблема: прямая может задаваться неоднозначно. Чтобы убедиться в этом, достаточно умножить обе части на λ. Удобно строить прямую по двум точкам:

```
line (const point& p, const point& q) {
  if (p == q) {
    exception;
  }
  a = p.y - q.y;
  b = q.x - p.x;
  c = p.x * q.y - p.y * q.x;
}
```

Действительно, если есть две точки $p\,(x_1, y_1)$, $q\,(x_2, y_2)$, то $a = y_1 - y_2$, $b = x_2 - x_1$, $c = x_1y_2 - x_2y_1$. Давайте это докажем:

Доказательство: Подстановка для p: $(y_1 - y_2) \cdot x_1 + (x_2 - x_1) \cdot y_1 + (x_1 y_2 - x_2 y_1) = 0$. То же самое будет для q. Заметим, что много что сокращается. ■

Большой плюс заключается в том, что целых чисел будет достаточно для представления прямой.

С прямой $ax + by + c = 0$ связаны два вектора: вектор нормали $\vec{n}(a, b)$ и направляющий вектор $\vec{l}(b, -a)$, можно взять как $(-b, a)$, ничего не поменяется. Понять, что направляющий вектор задан корректно, можно подстановкой, с помощью которой можно прийти к $a(x + b) + b(y - a) + c = ax + by + c = 0$. Понять, что $\vec{l}$ перпендикулярен $\vec{n}$ с помощью скалярного произведения, который равен нулю: $\left(\vec{l}, \vec{n}\right) = b \cdot a - a \cdot b = 0$.

Расстояние от точки до прямой

Пусть дана прямая $ax + by + c = 0$ и точка $p(p.x, p.y)$. Обозначим $p.x$ за x_0, $p.y$ — за y_0.

Утверждение. Рассстояние от точки p до прямой $ax + by + c$ будет равно $\frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$.

Доказательство: Опустим перпендикуляр из точки p на прямую $ax + by + c = 0$, основанием будет служить точка $q = p + \vec{n} \cdot \lambda$, при этом q лежит на прямой. Точка q является проекцией точки p на прямую $ax + by + c = 0$. Запишем в матричном виде:

$$q = \begin{pmatrix} x_0 + a \cdot \lambda \\ y_0 + b \cdot \lambda \end{pmatrix}$$

Далее подставляем и решаем:

$$a(x_0 + a \cdot \lambda) + b(y_0 + b \cdot \lambda) + c = 0$$

$$\lambda(a^2 + b^2) = -ax_0 - by_0 - c$$

$$\lambda = -\frac{ax_0 + by_0 + c}{a^2 + b^2}$$

Расстоянием будет $|\lambda \cdot \vec{n}| = |\lambda| \cdot |\vec{n}| = \frac{|ax_0 + by_0 + c|}{a^2 + b^2} \cdot \sqrt{a^2 + b^2}$. ■

Заметим, что по найденному значению λ можно найти и координаты проекции точки на прямую.

Однако от целых чисел придётся перейти к числам с плавающей точкой, возможно, придётся завести класс *Rational*, с помощью которого можно поддерживать рациональные числа.

Пересечение прямых

Пригодится метод Крамера. Пусть у нас есть две прямые $a_1 x + b_1 y + c_1 = 0$ и $a_2 x + b_2 y + c_2 = 0$. По сути мы решаем систему линейных уравнений:

$$\begin{cases} a_1 x + b_1 y = -c_1; \\ a_2 x + b_2 y = -c_2. \end{cases}$$

Считаем определитель матрицы, составленной из a_1, a_2, b_1, b_2 как $\Delta = a_1 b_2 - a_2 b_1$, посчитаем ещё $\Delta_1 = -c_1 b_2 + c_2 b_1$, $\Delta_2 = -a_1 c_2 + a_2 c_1$. Если $\Delta = 0$, то прямые либо параллельны, либо совпадают, так как направляющие векторы либо параллельны, либо совпадают. Если $\Delta \neq 0$, то единственным решением системы будет $\left(\frac{\Delta_1}{\Delta}, \frac{\Delta_2}{\Delta}\right)$.

Как понять, что прямые совпадают? Прямые совпадают тогда и только тогда, когда существует такое число k, что $a_1 = k a_2$, $b_1 = k b_2$, $c_1 = k c_2$. Если $a_2 \neq 0$, то подойдёт $k = \frac{a_1}{a_2}$.

Пересечение окружности и прямой

Как хранить окружность? Можно хранить точку p, отвечающую за центр окружности, и её радиус r. Пересечение окружности и прямой l есть тогда и только тогда, когда расстояние между точкой p и прямой l не превосходит r. Наивно можно полагать, что ищем решение системы уравнений:

$$\begin{cases} ax + by + c = 0; \\ (x - x_0)^2 + (y - y_0)^2 = r^2 \end{cases}$$

Можно опустить перпендикуляр из центра окружности на прямую. Далее можно отступить от проекции q на некоторое расстояние по обе стороны. Найдём направляющий вектор прямой l, нормируем его. Найдём $k = \sqrt{r^2 - dist^2(p, q)}$, где $dist(p, q) = \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2}$. Находим $q + k\vec{v}$, $q - k\vec{v}$, где $\vec{v}$ — направляющий вектор l единичной длины. Если всего лишь одна точка пересечения, то вернём проекцию q.

Пересечение двух окружностей

Пусть заданы две окружности: одна с центром в точке (x_1, y_1) и радиусом r_1; другая с центром в точке (x_2, y_2) и радиусом r_2. Казалось бы, можно было бы рассматривать много разных случаев, но это довольно громоздко. Проще рассмотреть систему двух квадратных уравнений, вычесть из второго уравнения первое, сократив все квадраты, получим линейное уравнение относительно x и y, которое задаст прямую, которую нужно пересечь с первой окружностью, и именно это прямая пересекает точки пересечения двух окружностей.

$$\begin{cases} (x - x_1)^2 + (y - y_1)^2 = r_1^2; \\ (x - x_2)^2 + (y - y_2)^2 = r_2^2. \end{cases}$$

И здесь тоже не обойтись без чисел с плавающей точкой.

Скалярное и псевдовекторное произведение

Скалярное произведение (dot product) двух вектор (x_1, y_1) и (x_2, y_2) можно вычислить как $x_1 x_2 + y_1 y_2$. Кроме того, зная векторы $\vec{u}$ и $\vec{v}$ и угол φ между ними, то скалярное произведение можно вычислить

как $|\vec{u}| \cdot |\vec{v}| \cdot \cos\varphi$. По косинусу можно узнать тип угла.

Пусть $(\vec{u}, \vec{v})$ — скалярное произведение, $\cos\varphi = \frac{(\vec{u},\vec{v})}{|\vec{u}|\cdot|\vec{v}|}$, и $\varphi = arccos\left(\frac{(\vec{u},\vec{v})}{|\vec{u}|\cdot|\vec{v}|}\right)$, $\varphi \in [0;\pi]$.

Триангуляции многоугольников

Есть некоторый многоугольник — замкнутая несамопересекающаяся ломаная. Нужно найти его триангуляцию — разбиение многоугольника на треугольники с помощью диагоналей такое, что все эти треугольники лежат в многоугольники. Диагонали здесь являются отрезками между несоседними вершинами, полностью лежащие во внутренности многоугольника.

Она полезна во многих прикладных задачах. Например, есть задача о многоугольнике и вахтёршах. Для её решения как раз нужно построить триангуляцию, и в каждый треугольник помесить вахтёршу.

Упражнение. Показать, что треугольников всегда ровно $n-2$, где n — число вершин в многоугольнике.

Упражнение. Достаточно, что хватит $n\ div\ 3$ вахтёрш, чтобы обозреть весь многоугольник.

Упражнение. Бывают многоугольники, для которых необходимо $\frac{n}{3} - const$ вахтёрш.

Построение триангуляции

Если многоугольник выпуклый, то построить его триангуляцию возможно за $O(n)$, рассмотрев диагонали из фиксированной точки.

Если многоугольник не является выпуклым, то задача становится намного интереснее.

Def. Ухом в многоугольнике называется треугольник $v_{i-1}v_iv_{i+1}$, который целиком лежит в многоугольнике и при этом внутри этого треугольника нет других вершин.

Утверждение. В любом простом (без самопересечений) многоугольнике с $n \geqslant 4$ есть хотя бы два не пересекающихся по внутренности уха.

Доказательство: Индукция по n.

База: $n = 4$. Каждый простой четырёхугольник можно разбить на два уха. В выпуклом многоугольнике (например, квадрате), всего 2 возможных разбиения, в невыпуклом — всего лишь одно.

Переход: рассмотрим произвольную выпуклую вершину v_i такую, что угол при ней меньше π. Рассмотрим две соседние вершины v_{i+1} и v_{i-1}, проводим разрез, в оставшемся многоугольнике по предположению индукции найдутся хотя бы два уха.

Но что будет в случае, когда найдётся внутри предполагаемого уха некоторая точка q, которая является самой близкой к v_i среди точек внутри или на границе треугольника. Тогда v_iq — диагональ, можно провести разрез по ней, и в каждой из двух оставшихся частей будет не менее, чем три

вершины, и тем самым в каждой части будет хотя бы одно ухо. ■

Алгоритм построения триангуляции

Поддерживаем вершины многоугольника в двусвязном списке. Далее напишем процедуру $FindEar$, которая проводит отрезок между v_{i+1} и v_{i-1} и смотрит, попали ли ещё точки внутри. Если нет, то это ухо, иначе найдём точку внутри треугольника, и проведём диагональ. Точек найдётся не более, чем $\frac{n}{2}+1$, учитывая, что два многоугольника пересекаются по двум вершинам. Далее запустимся рекурсивно от каждой из частей. Найдём асимптотику выполнения операции FindEar. $T(n) = \Theta(n) + T\left(\frac{n}{2}\right) \Longrightarrow T(n) = \Theta(n)$.

Алгоритм весь будет работать за $\Theta(n^2)$. Существует алгоритм за $O(n \log n)$, который, скорее всего, рассмотрен не будет.

Лекция 10. Выпуклая оболочка

Выпуклая оболочка

Def. Пусть $\vec{u} = (x_1, y_1)$, $\vec{v} = (x_2, y_2)$. «Векторным» произведением $\vec{u} \times \vec{v} = [\vec{u}, \vec{v}] = cross(\vec{u}, \vec{v})$ называется следующая величина:

$$x_1 y_2 - x_2 y_1 = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix}$$

Свойства «векторного произведения»:

1. Его величина численно равна ориентированной площади параллелограмма, натянутого на векторы $\vec{u}$ и $\vec{v}$;
2. $\vec{u} \times \vec{v} = |u| \cdot |v| \cdot \sin\alpha$;
3. Знак векторного произведения говорит нам о направлении кратчайшего поворота между двумя векторами: если знак положительный, то против часовой стрелки, если знак отрицательный, то по часовой стрелке, если значение равно нулю, то векторы либо сонаправлены, либо направлены противоположно.

Def. Пусть S — множество точек в $\mathbb{R}^2$, $|S| = n$. Тогда выпуклой оболочкой S называется минимальное по включению пересечение выпуклых надмножеств S: $conv(S) = \bigcap\limits_{F \supset S} F$, где F — выпуклое множество.

Note. $conv(S)$ — минимальное по включению выпуклое множество, содержащее S.

Note. $conv(S)$ — выпуклый многоугольник с вершинами в некоторых точках S.

Наивный алгоритм построения выпуклой оболочки

Наивный алгоритм за $O(n^3)$: перебираем пары точек за $O(n^2)$, проведём прямую через текущую пару точек, за $O(n)$ проверяем, действительно ли все точки лежат либо по одну сторону от этой прямой[5], что будет означать, что они лежат в выпуклой оболочке. Если пара точек действительно образует сторону выпуклой оболочки, то проведём её.

Заворачивание подарка (алгоритм Джарвиса)

Этот алгоритм работает за $O(n \cdot h)$, где h — число вершин в выпуклой оболочке. Рассмотрим точку из S с минимальной ординатой. Если таких несколько, то выберем из них точку с минимальной абсциссой. Выбираем минимальную из оставшихся по ординате точку. Образуем сторону P_1P_2. Посмотрим, какую сторону выбрать следующей. Угол между сторонами должен быть как можно большим. Для текущей вершины перебираем $O(n)$ вершин, текущую из них обозначим за Q, найдём из них такую точку Q^*, что их «векторное» произведение между P_2Q и P_2Q^* меньше нуля для любого Q, так как угол поворота в этом случае будет отрицательным.

```
Q* - any point
for Q in S
  if cross(P2Q, P2Q*) > 0:
    Q* = Q
```

При нулевом векторном произведении, что бывает в случае, когда несколько точек лежат на одной прямой, нужно предпосчитать вершину, наиболее удалённую от P_2.

Сортировка по координатам (алгоритм Эндрю)

Этот алгоритм работает за $O(n \log n)$. Отсортируем S по парам (x, y) сначала по возрастанию x, затем по возрастанию y. Строим нижнюю и верхнюю части выпуклой оболочки. Определим крайние левую и правую точку текущей оболочки. В отдельных векторах будем хранить нижнюю и верхнюю часть. При рассмотрении новой вершины проводим касательные к оболочке через неё, всё, что попало в угол между касательными, попадает в текущую оболочку. Если все ближайшие точки принадлежат нижней оболочке, удаляем точки, которые попали в угол между касательными. Если аналогичное с верхней оболочкой, то и поступаем аналогично.

Пусть new — новая добавленная точка, Q_1, P_1 — точки из нижней оболочки. Если $cross(Q_1P_1, P_1new) < 0$, то P_1 нужно удалить.

Пока в нижней оболочке хотя бы две точки, находим последние две точки P и Q, и если

[5] Допустимо, что некоторые точки могут лежать на прямой между текущей парой точек, но недопустимо, что некоторые точки лежат на продолжениях лучей, так как в этом случае текущая пара точек не будет образовывать сторону выпуклой оболочки.

$cross\,(QP, Pnew) < 0$, то удаляем последний элемент вектора, соответствующего нижней оболочке.

С верхней оболочкой делаем аналогичное, но нужно будет проверять векторные произведения не на то, что они отрицательные, а на то, что они положительные: нижнюю оболочку обходим против часовой стрелки, а верхнюю оболочку обходим по часовой стрелке.

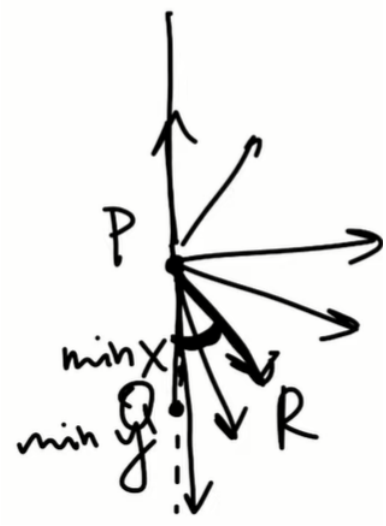
Почему это работает? Мы идём с конца нашего вектора, считаем, что точки уже расположены слева направо, посмотрим на отрезок, соответствующий касательной, и на остальные отрезки, проведённые из new в точки между касательной и самой правой точкой. Тогда точки, которые нужно будет удалить, лежат во внутренности угла между касательными к оболочке, проведёнными из new, следовательно, соответствующие векторные произведения будут отрицательными.

Асимптотика $O\,(n \log n)$ достигается за счёт сортировки по координатам за $O\,(n \log n)$ и $O\,(n)$ удалений, так как каждая точка удалится не более одного раза.

Сортировка по углу (алгоритм Грэхема)

Алгоритм работает за $O\,(n \log n)$, он во многом аналогичен предыдущему. Возьмём сначала точку с минимальной абсциссой, а если таковых несколько, то с минимальной ординатой среди них. Ни одна точка не находится ниже, ни одна точка не находится левее. Значит, все векторы из выбранной точки будут проведены в правой полуплоскости, и их можно отсортировать по полярному углу из выбранной точки.

Пусть P — самая нижняя среди самых левых точек, PQ образует вектор, направленный строго вниз, тогда векторы вида PR можно отсортировать по углу $\angle QPR$, где R — произвольная точка из оставшихся. Сортировать будем по возрастанию. Явно находить значение угла не требуется, достаточно векторного произведения.



Пусть R_1 и R_2 — произвольные две точки из оставшихся. Если $\angle R_1 P R_2 > 0$, то точка R_1 встретится раньше точки R_2. Если R_1 и R_2 находятся на одной прямой, то есть соответствующее векторное

произведение равно нулю, то если R_1 находится ближе к P, чем R_2, то R_1 раньше R_2.

Пусть уже была построена выпуклая оболочка на некоторых точках. Пусть новая точка находится левее и выше, чем текущая построенная выпуклая оболочка. Тогда просто добавим новую точку в вектор. Иначе поступаем аналогичным образом, как в алгоритме построения выпуклой оболочки сортировкой по координатам: удалим вершины выпуклой оболочки, находящиеся внутри угла между касательными к выпуклой оболочке, проведёнными через новую точку.

Пусть new — новая вершина, A и B — последние две вершины. Условие на удаление такое: пока в векторе хотя бы две точки, если $cross\,(BA, Anew) < 0$, то нужно удалить две последние точки из вектора, иначе нужно выйти из цикла.

Динамическая выпуклая оболочка

Задача. Пусть S — изменяющееся множество точек. Нужно построить для него выпуклую оболочку, при добавлении новой точки нужно её перестроить.

Решение: Храним нижнюю и верхнюю выпуклые оболочки в виде $set < Point >$, где точки отсортированы по абсциссе. Заметим, что новая точка необязательно лежит правее выпуклой оболочки: она может лежать где угодно. Удаления работать не будет, но будет работать идея с касательными.

Поймём, как обновляется верхняя выпуклая оболочка. Пусть new — новая точка. Разберём самый сложный случай, когда нужно провести касательные и удалить некоторые точки. Проведём вертикальную прямую, разделив на левую и правую части. Если и потребуется удалить точки, то это будет несколько самых левых точек правой части и несколько самых правых точек левой части. Если мы сможем найти разрез, то тогда сможем и удалить точки обеих частей, которые нарушают выпуклость, что можно понять по векторному произведению. Разрез можно найти с помощью бинарного поиска, например, с помощью $lower\_bound$ из стандартной библиотеки. Это позволит найти и самую левую точку правой части, и самую правую точку левой части. Передвигаться по точкам можно, увеличивая или уменьшая соответствующий итератор на 1.

Что будет, если точка находится ниже верхней выпуклой оболочки? Тогда делаем бинарный поиск, проводим разрез на две части, смотрим на отрезок между самой правой точкой левой части и самой левой точкой правой части, и если точка лежит под отрезком, то ничего делать не нужно.

Что будет, если точка находится левее верхней выпуклой оболочки? Тогда смотрим на самую левую вершину выпуклой оболочки, и, двигаясь вправо, убираем точки, которые нарушают выпуклость. В случае, когда точка находится правее верхней выпуклой оболочки, поступаем аналогично.

Операции с нижней выпуклой оболочки выполняются аналогично.

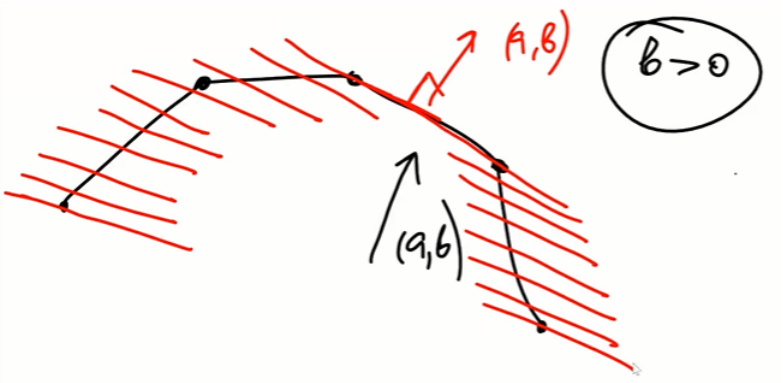
Асимптотика операции вставки: $O^*\,(\log n)$, так как вставку и удаление можно выполнять за $O\,(\log n)$, но удалений суммарно может быть не более, чем всего точек. Если будет q запросов, то суммарно запросы выполнятся за $O\,(q \log q)$.

Note. Реализовывать на двусвязном списке невыгодно, так как не будет возможности проводить бинарный поиск, и суммарно запросы выполнятся за $O(q^2)$ при q запросах.

Задача о максимальном скалярном произведении

Задача. Дано фиксированное множество S из n точек. Поступает q запросов вида (a, b). Нужно найти $\max\limits_{(x,y)\in S} (x, y) \cdot (a, b)$.

Решение: Заметим, что максимум скалярного произведения достигается тогда, когда (x, y) лежит на границе выпуклой оболочки множества S. Из более ранних курсов известно, что для любой константы γ геометрическое место точек таких, что $(x, y) \cdot (a, b) = \gamma$ представляет собой прямую, ортогональную (a, b). Разобьём $conv(S)$ на верхнюю и нижнюю части. Разберём случай, когда $b > 0$. Представим себе прямые, проходящие через границу верхней выпуклой оболочки, ортогональные (a, b):



Заметим, что если мы будем двигаться по границе верхней выпуклой оболочки, то сначала значение скалярного произведения будет возрастать, а затем будет убывать. Сделаем бинарный поиск по производной. Если в точке p_{m+1} значение больше, чем в p_m, то левую границу передвигаем на позицию, соответствующую p_m, если в точке p_{m+1} значение меньше, чем в p_m, то правую границу передвигаем на позицию, соответствующую p_m. Если значения совпали, то выведем ответ.

```
while (r - l > 1):
  m = (r + l) >> 1;
  x = dot(p[m], vector(a, b));
  y = dot(p[m + 1], vector(a, b));
  if (x < y):
    l = m;
  if (x > y):
    r = m;
  if (x == y):
    return x;
```

Разберём случай, когда $b < 0$. Тогда рассмотрим нижнюю выпуклую оболочку, и проводим аналогичные операции.

Асимптотика: $O(\log n)$, так как бинарный поиск выполняется за $O(\log n)$.

Упражнение. Пусть S — изменяющееся множество точек. Поступает q запросов вида (a, b). Нужно найти $\max\limits_{(x,y)\in S}(x, y) \cdot (a, b)$ за $O(\log n)$[6].

Лекция 11

Задача о максимальном расстоянии между двумя точками множества

Задача. Пусть S — множество точек. Нужно найти две самые далёкие точки данного множества, то есть нужно найти $\max\limits_{P,Q\in S} dist(P, Q)$.

Решение: Заметим, что максимум достигается на вершинах выпуклой оболочки множества S. Если отрезок находится внутри выпуклой оболочки, то его можно продлить до пересечения со стороной выпуклой оболочки в обе стороны. За $O(n \log n)$ строим выпуклую оболочку множества S. Докажем лемму:

Лемма. Пусть PQ — диаметр S, то есть верно следующее:

$$dist(P, Q) = \max_{P^*,Q^*\in S} dist(P^*, Q^*)$$

Проведём прямые l_1 и l_2, перпендикулярные PQ, проходящие через P и Q соответственно. Тогда S целиком лежит в полосе между l_1 и l_2.

Доказательство: Пусть это не так. Тогда рассмотрим некоторую точку выпуклой оболочки, расположенную за пределами полосы между l_1 и l_2. Тогда диаметр на самом деле должен был быть больше. Противоречие. ■

Обозначим за l_1 крайнюю левую вертикальную прямую, за l_2 — крайнюю правую вертикальную прямую. Далее начинаем синхронно вращать эти две прямые против часовой стрелки. Изменения начнутся тогда, когда одна прямая начнёт содержать одну из сторон выпуклой оболочки, тогда обновим одну из двух опорных точек. Всего таких переключений происходит $O(n)$ раз. В каждый момент переключения одной опорной точки измеряем расстояние между текущими опорными, обновляем ответ.

Теперь о технических деталях. Чтобы узнать, где раньше произойдёт переключение, сравним векторные произведения. Пусть $\vec{u}$ и $\vec{v}$ — векторы, соответствующие двум сторонам. Переключение l_1 произойдёт раньше, если и только если $cross(\vec{u}, \vec{v}) < 0$.

Утверждение. Пусть $ABCD$ — трапеция с основаниями AB и CD. Тогда $\max(AC, BD) \geqslant \max(BC, AD)$.

[6] Для этого пригодится неявное дерево поиска: обычного дерева будет недостаточно.

Доказательство: Пусть не так, и $BC = \max(BC, AD) > AC, BD$. Если $BC = \max(BC, AD) > AC$, то если рассмотреть перпендикуляр из точки B на прямую CD, то D лежит ближе к основанию перпендикуляра, чем C. Аналогично можно рассмотреть иные ситуации. ■

Сумма Минковского

Пусть M_1 и M_2 — два многоугольника (со внутренностью).

Def. Сумма Минковского $M_1 + M_2$ — множество $\{a + b \mid a \in M_1, b \in M_2\}$.

Утверждение. Если M_1 и M_2 — выпуклые многоугольники, то их сумма $M_1 + M_2$ тоже является выпуклым многоугольником.

Доказательство: Докажем выпуклость. Покажем, что если $P, Q \in M_1 + M_2$, то $[P, Q] \subset M_1 + M_2$. Если $P, Q \in M_1 + M_2$, то $P = P_1 + P_2$, $Q = Q_1 + Q_2$, нижний индекс i соответствует множеству M_i. Из выпуклости M_1 и M_2 следует, что $[P_1, Q_1] \subset M_1$, $[P_2, Q_2] \subset M_2$. Покажем, что $[P, Q] \subset M$. Пусть $[P_1, Q_1] = P_1 + t(Q_1 - P_1)$, $[P_2, Q_2] = P_2 + t(Q_2 - P_2)$. Сложим эти два отрезка, получим требуемое.

Покажем, что получится многоугольник. Докажем следующее:

$$M_1 + M_2 = conv(\{P + Q \mid P \in vertices(M_1), Q \in vertices(M_2)\})$$

Из выпуклости следует следующий факт:

$$M_1 + M_2 \supset conv(\{P + Q \mid P \in vertices(M_1), Q \in vertices(M_2)\})$$

Покажем вложение в обратную сторону. Рассмотрим точку $X+Y \in M_1+M_2$, то есть $X \in M_1, Y \in M_2$. Рассмотрим триангуляции M_1 и M_2, так как они существуют, то точки найдутся в хотя бы одном из треугольников из соответствующих триангуляций. Рассмотрим получившиеся треугольники, в одном из которых лежит X (вершины — A_1, A_2, A_3), в другом — Y (вершины — B_1, B_2, B_3). X представим в виде линейной комбинации $\lambda_1 A_1 + \lambda_2 A_2 + \lambda_3 A_3$, где $\lambda_i \geqslant 0$, $\sum\limits_i \lambda_i = 1$. Аналогично с Y, коэффициенты в соответствующей линейной комбинации — μ_i. Это можно показать, проведя чевианы треугольников.

$$X + Y = \lambda_1 A_1 + \lambda_2 A_2 + \lambda_3 A_3 + \mu_1 B_1 + \mu_2 B_2 + \mu_3 B_3$$

$$X + Y \in conv(\{A_1 + B_1, A_1 + B_2, A_1 + B_3, \ldots, A_3 + B_3\})$$

Соответствующие коэффициенты: $\lambda_1\mu_1$, $\lambda_1\mu_2$, $\lambda_1\mu_3$, $\ldots$, $\lambda_3\mu_3$.

Утверждение. (без доказательства) Точка X лежит в $conv(\{P_1, P_2, \ldots, P_n\}) \iff \exists \lambda_1, \ldots, \lambda_n \geqslant 0$, $\lambda_1 + \cdots + \lambda_n = 1 : X = \lambda_1 P_1 + \cdots + \lambda_n P_n$ (выпуклая комбинация)

Вывод: $M_1 + M_2$ — выпуклый многоугольник. ■

Алгоритм нахождения суммы Минковского

Найдём крайние левые нижние точки выпуклых многоугольников M_1 и M_2 (сначала рассматриваем по x, потом по y). Обозначим их за S_1 и S_2. $S_1 + S_2$ будет вершиной $S_1 + S_2$, так как по свойству точек S_1 и S_2 точка $S_1 + S_2$ будет крайней левой нижней точкой многоугольника $M_1 + M_2$. Далее рассмотрим векторы, соответствующие сторонам каждого из многоугольников. Обходим векторы против часовой стрелки, начиная с левой нижней точки каждого из многоугольника. Записываем в качестве новой стороны самую «крутую» из двух сторон, то есть если $cross\,(u, v) > 0$, то u нужно расположить раньше v. Переключаем соответствующий вектор одного из многоугольников на следующий в порядке обхода, невыбранный вектор останется тем же. Выбранный вектор откладываем от конца последнего на данный момент вектора строящейся суммы $M_1 + M_2$. В стартовом случае откладываем выбранный вектор от точки $S_1 + S_2$. Получим выпуклый многоугольник, так как сумма векторов, соответствующих сторонам M_1, равна нулю и сумма векторов, соответствующих сторонам M_2, равна нулю.

Утверждение. Алгоритм нахождения суммы Минковского корректен.

Доказательство: Построим настоящую сумму Минковского $M_1 + M_2$. Точка $S_1 + S_2$ действительно лежит в настоящей сумме Минковского по свойству выбранных точек S_1 и S_2. Пусть несколько первых сторон совпали. Посмотрим, какой может быть следующая сторона. Построенный многоугольник вложен в настоящую сумму Минковского по построению. Покажем, что настоящая сумма Минковского вложена в построенный многоугольник. Пусть P — вершина M_1, Q — вершина M_2. Соответствующая вершина построенного многоугольника — $P + Q$. Пусть $R_1 + R_2$ — следующая в порядке обхода вершина настоящей суммы Минковского. Тогда $R_1 \in M_1$, $R_2 \in M_2$. Пусть A — следующая в порядке обхода вершина M_1, B — следующая в порядке обхода вершина M_2. Пусть при построении выбран вектор $\vec{u}$, соответствующий либо стороне PA многоугольника M_1, либо стороне QB многоугольника M_2, выбор зависит от векторного произведения. Тогда следует следующее:

$$\begin{cases} cross\left(\overrightarrow{u}, \overrightarrow{PR_1}\right) \geqslant 0 \\ cross\left(\overrightarrow{u}, \overrightarrow{QR_2}\right) \geqslant 0 \end{cases} \Longrightarrow cross\left(\overrightarrow{u}, \overrightarrow{PR_1} + \overrightarrow{QR_2}\right) \geqslant 0 \Longrightarrow cross\left(\overrightarrow{u}, \overrightarrow{(P+Q)(R_1+R_2)}\right) \geqslant 0$$

Отсюда следует, что точка предполагаемая точка $R_1 + R_2$ не лежит вне построенного многоугольника $M_1 + M_2$, и $R_1 + R_2$ должна совпасть с концом вектора $\vec{u}$, отложенного от точки $P + Q$, если $R_1 + R_2$ — следующая в порядке обхода вершина настоящей суммы Минковского. Таким образом, настоящая сумма Минковского совпадает с построенным многоугольником $M_1 + M_2$. ■

Алгоритм строит сумму Минковского за $O\,(n + m)$.

Применение суммы Минковского

С помощью суммы Минковского можно проверить, пересекаются ли два выпуклых многоугольника M_1 и M_2, то есть существует ли такая точка P, что $P \in M_1$, $P \in M_2$. Рассмотрим разность Мин-

ковского $M_1 - M_2$. Пересекаемость равносильна тому, что $(0,0) \in (M_1 - M_2)$. Построить разность можно как сумму M_1 и $-M_2$.

Ещё можно найти расстояние между двумя выпуклыми многоугольниками:

$$dist(P,Q) = ||P-Q||_2$$[7]
$$P \in M_1, Q \in M_2$$

Найдём в разности Минковского $M_1 - M_2$ точку с минимальной евклидовой нормой, то есть с минимальным расстоянием до $(0,0)$. По сути, расстояние равно расстоянию от $(0,0)$ до ближайшей стороны многоугольника $M_1 - M_2$.

Упражнение. Пусть M_1 и M_2 — невыпуклые многоугольники, требуется найти $M_1 + M_2$. Можно триангулировать два многоугольника, построим все попарные суммы Минковского треугольников, и в конце их объединим в один многоугольник. Требуется найти способ объединения, а также асимптотику.

Ещё можно проверить принадлежность точки многоугольнику.

Упражнение с семинара. Если многоугольник выпуклый, то проверить можно за $O(\log n)$.

Решение: Разобьём многоугольник на треугольники по диагоналям, проведённым из левой нижней точки. Тогда и угол при левой нижней вершине разобьётся на k углов, где k — количество получившихся треугольников. Бинарным поиском определим угол, внутри которого лежит точка. Далее за $O(1)$ определим, лежит ли точка внутри треугольника.

Note. Если многоугольник является невыпуклым, то его можно триангулировать за $O(n^2)$, далее проверим, лежит ли эта точка в одном из треугольников. Однако есть решение за $O(n)$. Рассмотрим некоторый угол $\angle APB$. Величина каждого такого угла известна. Сумма углов с вершиной в точке P будет равна $\pm 2\pi$ и не равна нулю, если P принадлежит многоугольнику, иначе нет.

Проблемы. Тригонометрические операции работают довольно долго, точность не является абсолютной.

Лекция 12

Вновь о задаче проверки принадлежности точки многоугольнику

Решение: Рассмотрим многоугольник, необязательно выпуклый. Рассмотрим в нём некоторую точку, запустим из неё луч вправо. Если количество пересечений с границей многоугольника нечётно, то точка принадлежит многоугольнику, иначе нет. Каждая точка пересечения отвечает за измене-

[7] Здесь норма считается евклидовой.

ние области, через которую проходит луч. Этого решения будет достаточно, если возможные точки пересечения являются внутренними точками сторон многоугольника.

Проблема: вершины многоугольника могут попасть на луч, при прохождении луча через вершину многоугольника область не меняется, и о принадлежности точки ничего сказать нельзя.

Решение 1. Можно запускать вместо горизонтального луча случайный луч, но для этого нужно использовать числа с плавающей точкой. Недопустимых лучей, то есть таких лучей, которые могут пройти через вершину многоугольника, не больше, чем n, где n — количество вершин многоугольника. Если считать, что луч задаётся начальной точки и углом поворота $\alpha \in [0, 2\pi]$, то мера множества недопустимых углов равна нулю. Если вдруг луч проходит через вершину, то сгенерируем луч заново.

Решение 2. Пусть все точки имеют целочисленные координаты в $[0, A] \times [0, A]$. Проверяем на принадлежность многоугольнику точку p. Из точки p запустим вектор $(A, A + 1)$. Тогда на луче, запущенном из точки p, проходящем через вектор $(A, A + 1)$, не встретится ни одной вершины многоугольника. Действительно, если бы хотя бы одна вершина встретилась, то это была бы целочисленная точка, отличная от начала и конца вектора. Пусть эта точка имеет координаты (x, y). Тогда вектор $(x - p.x, y - p.y)$ коллинеарен вектору $(A, A + 1)$, и он представим в виде $\lambda \cdot (A, A + 1)$. Однако координаты вектора $(A, A + 1)$ являются взаимно простыми числами. В силу ограничения на значения координат, должно выполняться условие $\lambda \in (0, 1)$, что приведёт к тому, что координаты точки пересечения могут быть не целыми числами. Так как $A\lambda$ и $(A + 1)\lambda$ являются целыми числами, то и λ тоже будет целым числом, но тогда конец вектора будет за пределами многоугольника.

Однако решение имеет недостаток: нужно знать значение A.

Решение 3. Запустим горизонтальный луч, разберёмся с вершинами на луче. Пусть (a, b) — сторона многоугольника. При необходимости переставим точки a и b так, чтобы выполнялось $b.y \geqslant a.y$.

```
if (b.y <= p.y || a.y > p.y) {
  continue;
}
if (cross(p - a, b - a) < 0) {
  ++cnt;
}
```

Здесь cnt — количество пересечений. Работоспособность показывается через перебор случаев.

1. Пусть отрезок $[a, b]$ находится левее горизонтального луча, проведённого из точки p, и этот луч не пересекает отрезок. Тогда $cross\,(p - a, b - a) > 0$, так как угол в этом случае образуется положительный.

2. Пусть отрезок $[a, b]$ пересекает горизонтальный луч, проведённый из точки p. Тогда $cross\,(p - a, b - a) < 0$, так как образуется угол, направленный по часовой стрелке.

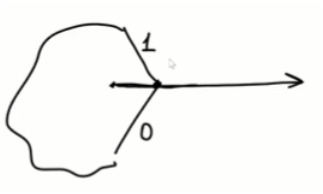
Далее проверим число пересечений на чётность.

```
if (cnt % 2 == 1) {
  cout << "p is in our figure" << endl;
} else {
  cout << "p is not in our figure" << endl;
}
```

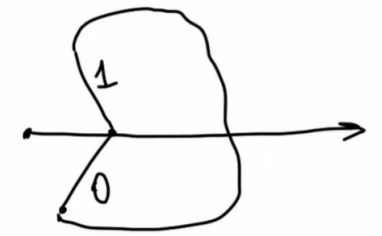
Note. Нужно отдельно проверить, не лежит ли точка p на границе многоугольника.

Асимптотика: $O(n)$, где n — количество вершин многоугольника.

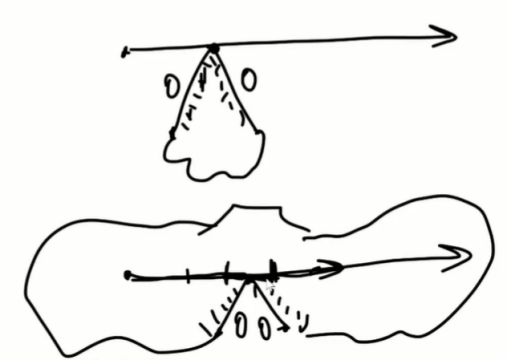
Чтобы показать корректность, нужно рассмотреть случаи, когда хотя бы один из концов отрезка находится на луче. Один из возможных случаев:



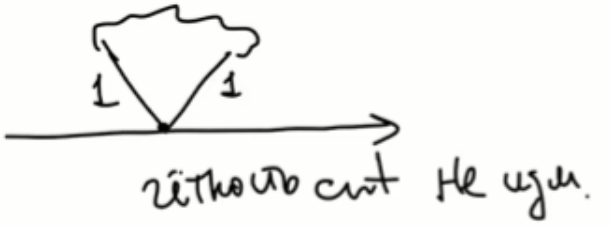
В этом случае *cnt* изменяется на $1 + 0 = 1$, и изменение области действительно происходит.



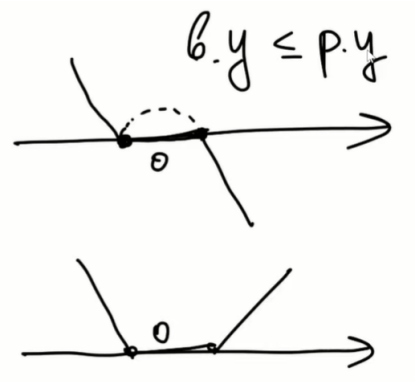
Этот случай разбирается аналогично. Посмотрим на случаи, когда область не изменяется:



В случаях, когда оба отрезка находятся не ниже, *cnt* прибавляется на $1 + 1 = 2$, тем самым чётность не изменяется.



В случае, когда отрезок лежит на луче, *cnt* не изменяется, не изменяется и чётность.



Пересечение полуплоскостей

Каждая полуплоскость задаётся прямой и направлением.

Задача. Даны n полуплоскостей. Нужно найти их пересечение.

Note. Считаем, что в множестве полуплоскостей x, y не превосходят некоторого числа $inf = 10^9$ по модулю. Этот приём называется bounding box, и широкое применение он получил в разработке компьютерных игр.

В большинстве случаев получается выпуклый многоугольник. Выпуклость следует из того, что эта фигура — пересечение выпуклых фигур. Однако может получиться и неограниченная выпуклая фигура.

Представление полуплоскостей

Прямую храним в виде чисел a, b, c таких, что $ax + by + c = 0$. Полуплоскость можно хранить в виде таких же чисел, что $ax + by + c \geqslant 0$. Альтернативно можно хранить нормаль к прямой. Векторы нормалей нормированы, то есть $a^2 + b^2 = 1$.

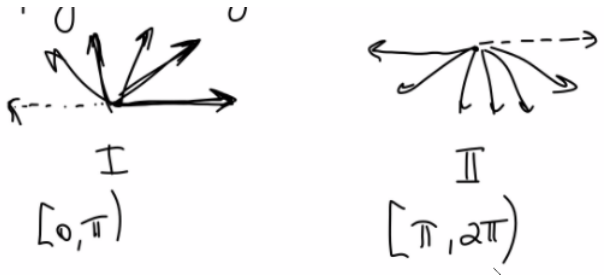
Алгоритм за $O(n^2)$

Изначально посмотрим на bounding box, затем пройдём по всем полуплоскостям в произвольном порядке, и «отрезаем» часть, которую не принадлежит полуплоскости. Храним выпуклый многоугольник, который является временным ответом, и для любого i пересекаем этот многоугольник с i-й полуплоскостью. В многоугольнике $O(n)$ вершин, пересечение полуплоскостью происходит за $O(n)$, и итоговая асимптотика выходит $O(n^2)$.

Если для любой вершины v многоугольника $a \cdot v.x + b \cdot v.y + c \geqslant 0$, то менять ничего не нужно. Если же для любой вершины v выполняется, что $a \cdot v.x + b \cdot v.y + c < 0$, то ответом будет пустое множество.

Алгоритм за $O(n \log n)$

Разобьём полуплоскости на 2 класса по направлению вектора нормали:



Если пересечь все полуплоскости первого класса, то получится в некотором смысле нижняя огибающая нашего ответа. Если аналогично пересечь все полуплоскости второго класса, то получится верхняя огибающая нашего ответа.

Внутри каждого класса отсортируем все полуплоскости по углу нормали, используя векторное произведение. Сортировка происходит за $O(n \log n)$. Далее объединим отсортированные списки полуплоскостей в один.

Note. Если сортировать исходный список, то предикат с векторным произведением не сработал бы.

Отсортированный список полуплоскостей храним в деке (deque, double ended queue). В цикле по i от 0 до $n-1$ делаем следующее: пока в деке есть хотя бы две полуплоскости и точка пересечения двух последних не лежит в h_i, то делаем $deque.pop\_back()$;, далее аналогично делаем $deque.pop\_front()$;, пока в деке есть хотя бы две полуплоскости и точка пересечения двух первых не лежит в h_i, затем делаем $deque.push\_back(h_i)$. После делаем ещё два цикла *while*, но на этот раз пока в деке хотя бы три полуплоскости, и точка пересечения последних двух не лежит в первой, то извлекаем с конца, пока в деке хотя бы три полуплоскости, и точка пересечения первых двух не лежит в последней, то извлекаем с начала. Если в деке осталось не более двух полуплоскостей, то ответ пуст. Иначе в деке лежат стороны искомого многоугольника.

Что делать с прямыми, у которых нормали параллельны? Среди полуплоскостей с одним и тем же вектором нормали оставляем только полуплоскость с наименьшим c.

Лекция 13

Диаграмма Вороного

Задача о почтовых отделениях (Post office problem)

Для того, чтобы понять, где можно применить диаграмму Вороного и триангуляцию Делоне, рассматривают задачу о почтовых отделениях.

Задача. (Post office problem) В некоторой стране, заданной на плоскости, уже открыты n почтовых отделений. Пусть известно, где находится некоторый человек. Найти для него самое близкое почтовое отделение, чтобы он смог быстрее отправить письмо.

Решение: Заметим, что страна разобьётся на n «регионов», каждый регион будет соответствовать самому близкому почтовому отделению к всем точкам региона. Осталось найти способ, как можно эффективно разбить страну на регионы.

Основные определения

«Почтовые отделения» в общем случае называются сайтами (от английского слова site) — это точки, по которым строится разбиение плоскости на несколько частей таких, что каждая из частей соответствует ближайшей к одной из заданных точек.

Def. Вершина диаграммы Вороного — точка, равноудалённая от нескольких сайтов.

Def. Пусть $P = \{p_1, \ldots, p_n\}$ — набор сайтов. Пусть $V(p_i)$ — ячейка диаграммы Вороного, которая определяется следующим образом:

$$V(p_i) = \{q \in \mathbb{R}^2 \mid dist(q, p_i) \leqslant dist(q, p_j) \ \ \forall j\}$$

Диаграммой Вороного $Vor(P)$ называется совокупность всех ячеек диаграммы Вороного $V(p_i)$.

Наивный алгоритм построения

Вытекает напрямую из определения диаграммы Вороного. Рассмотрим условие при фиксированных i, j:

$$dist(q, p_i) \leqslant dist(q, p_j) \ \ \forall j$$

Тогда множество точек q, для которых неравенство выполняется, является полуплоскостью, лежащей по одну из двух сторон серединного перпендикуляра к отрезку между p_i и p_j, и при том в стороне, где расположена точка p_i.

Рассмотрим для каждого сайта и каждой точки из диаграммы Вороного построим полуплоскости, пересечём их, получим выпуклые многоугольники, которые и станут ячейками диаграммы Вороного. Пересечение полуплоскостей возможно за $O(n \log n)$, для каждого сайта неравенств будет $n - 1$, и полуплоскостей будет $n - 1$. Асимптотика: $O(n^2 \log n)$.

Следствие Каждая ячейка диаграммы Вороного — выпуклая фигура, обобщённый выпуклый многоугольник.

Доказательство: Это следует из того, что пересечение выпуклых фигур выпукло. ■

Свойства диаграммы Вороного

Утверждение. Пусть P — множество сайтов, $Vor\,(P)$ — диаграмма Вороного, построенная по множеству сайтов P. Тогда верны следующие утверждения:

1. Если все сайты из P лежат на одной прямой, то $Vor\,(P)$ состоит из $(n-1)$ параллельной прямой;
2. В противном случае рёбрами диаграммы Вороного $Vor\,(P)$ могут выступать только отрезки или лучи, причём $Vor\,(P)$ связана, если её рассматривать как граф.

Доказательство: Докажем первое утверждение. Рассмотрим точки на горизонтальной прямой. Тогда по построению границами ячеек диаграммы Вороного будут серединные перпендикуляры между парами соседних точек, и их будет ровно $(n-1)$.

Докажем второе утверждение. Пусть не все сайты лежат на одной прямой. Покажем, что бесконечные прямые не могут быть рёбрами диаграммы Вороного $Vor\,(P)$. Пусть бесконечная прямая проведена через точки p_i и p_j, а точка p_k, которая располагается по той же стороне от серединного перпендикуляра, что p_j, не лежит на этой прямой. Рассмотрим серединный перпендикуляр к отрезку между точками p_j и p_k. Так как $p_k \notin p_ip_j$, то и серединный перпендикуляр к отрезку p_jp_k не будет параллелен серединному перпендикуляру к отрезку p_ip_j, и эти два перпендикуляра пересекутся. Тогда найдутся точки на серединном перпендикуляре к p_ip_j, которые ближе к p_k, чем к p_i и p_j, и они не могут находиться на границе ячейки $V\,(p_j)$. Значит, что рёбрами могут быть только отрезки или лучи, и диаграмма Вороного $Vor\,(P)$ связна. Диаграмма Вороного $Vor\,(P)$ может быть несвязна, только если одна из ячеек ограничена с двух сторон параллельными прямыми, что может быть только в случае, когда все сайты лежат на одной прямой. ■

Th. Если $n \geqslant 3$ и не все сайты лежат на одной прямой, то $Vor\,(p)$ содержит не более $2n-5$ вершин и не более $3n-6$ рёбер.

Доказательство: Так как $Vor\,(p)$ здесь будет связным графом, то для $Vor\,(p)$ применима формула Эйлера: $V-E+F=2$. Добавим фиктивную вершину v_∞, которая отвечает за лучи, и если эта точка существует, то она соединена хотя бы с тремя бесконечными лучами. Тогда $(V+1)-E+n=2$. Так как степень каждой вершины хотя бы 3, то количество рёбер можно оценить снизу через неравенство $2E \geqslant 3\,(V+1)$. Отсюда:

$$V-E+n=1$$

$$E=V+n-1 \geqslant \tfrac{3}{2}V+\tfrac{3}{2}$$

$$\tfrac{1}{2}V \leqslant n-\tfrac{5}{2}$$

$$V \leqslant 2n-5$$

$$E=V+n-1 \leqslant 3n-6$$

■

Def. Пусть P — набор сайтов, точка q лежит на плоскости. Тогда $C_q(P)$ — круг максимального радиуса с центром в точке q, который во внутренности не содержит ни одного сайта.

Th. Верны следующие утверждения:

1. Точка q является вершиной диаграммы Вороного $Vor(P)$ тогда и только тогда, когда $C_q(P)$ содержит на своей границе хотя бы три сайта;
2. В диаграмме Вороного $Vor(P)$ участвует серединный перпендикуляр к отрезку p_ip_j тогда и только тогда, когда существует точка q, лежащая на серединном перпендикуляре к отрезку p_ip_j, такая что $C_q(P)$ содержит на границе p_i и p_j, и больше никакие точки не содержит ни внутри, ни на границе.

Доказательство: Докажем первое утверждение.

$\Longrightarrow$ Для любой вершины q диаграммы Вороного выполнено, что её степень хотя бы 3, значит, q лежит на границе хотя бы трёх ячеек диаграммы Вороного. Значит, точка q равноудалена от хотя бы трёх сайтов, которые порождают эти ячейки. Пусть существует более близкий сайт, но тогда точка q не могла лежать в трёх исходных ячейках.

$\Longleftarrow$ Рассмотрим точку q и такой круг $C_Q(P)$, что на его границе хотя бы три сайта. Тогда точка q лежит на пересечении серединных перпендикуляров, и к q есть хотя бы три самых близких сайта, но тогда q лежит хотя бы на трёх ячейках, и q — вершина всех этих ячеек по построению.

Докажем второе утверждение.

$\Longrightarrow$ Рассмотрим серединный перпендикуляр к отрезку p_ip_j, выберем в качестве точки q любую внутреннюю точку той части перпендикуляра, которая участвует в диаграмме Вороного. Рассмотрим круг с центром в точке q и радиусом qp_j. Круг также содержит p_i. Круг пуст, иначе к q есть более близкий сайт. На границе круга больше нет других сайтов, так как иначе q по первому утверждению должна быть вершиной диаграммы Вороного $Vor(p)$.

$\Longleftarrow$ Пусть существует точка q, лежащая на серединном перпендикуляре к отрезку p_ip_j, такая что $C_q(P)$ содержит на границе p_i и p_j, и больше никакие точки не содержит ни внутри, ни на границе. Тогда самые близкие точки к q — это точки p_i и p_j, остальные находятся строго дальше. Тогда окрестность точки q является общей частью границы ячеек $V(p_i)$ и $V(p_j)$. ■

Алгоритм Форчуна

Алгоритм основан на принципе сканирующей прямой. Сканирующая прямая будет идти сверху вниз. Идея таковая: пусть l — текущая прямая, рассмотрим сайты выше неё и ниже неё, тогда диаграмма Вороного корректно построена для множества таких точек, что расстояние от них до какого-либо

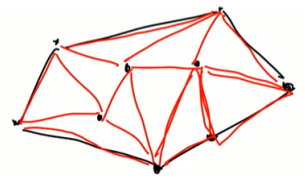
из сайтов меньше, чем расстояние до прямой, тогда сайт p_i будет фокусом параболы, а l — её директрисой, и все эти параболы в совокупности образуют береговую линию (beach line). Рассмотрим момент, когда обрабатывается новый сайт (site event, событие-точка). Появляется новая парабола, изначально вырождающаяся в луч, и она вставляется в береговую линию. Есть ещё один момент, связанный с удалением из береговой линии — так называемый circle event. Одна из дуг параболы вырождается в точку, которая равноудалена от трёх фокусов и директрисы. Смотрим на тройки все тройки подряд идущих дуг парабол, вычисляем точку, равноудалённую от трёх фокусов (строим центр описанной окружности соответствующего треугольника). Всего $O(n)$ событий, обработка событий идёт с помощью дерева поиска, и вся обработка занимает $O(n \log n)$ времени.

Note. Илья Степанов предпочёл рассказать общие слова об алгоритме Форчуна, не приводя деталей.

Триангуляция Делоне

Def. Пусть $P = \{p_1, \ldots, p_n\}$ — набор сайтов. Триангуляцией называется набор попарно непересекающихся по внутренности треугольников с вершинами в сайтах, объединение которых равно выпуклой оболочке для множества сайтов P.

Пример:



Применение

Задача. Даны n точек. Для каждой точки p_i известна высота над уровнем моря h_1. Восстановить ландшафт по этим точкам.

Решение: Первый способ: построить диаграмму Вороного на уровне моря, затем каждую ячейку, соответствующую p_i, поднять на высоту h_i, получится ступенчатая диаграмма, в случае, когда у двух близко расположенных точек сильно разные высоты, могут возникнуть проблемы. Второй способ: построить триангуляцию на множестве сайтов на уровне моря, затем поднять каждую точку на соответствующую высоту h_i, проблем уже меньше, но могут возникнуть проблемы с маленькими углами.

Задача. Найти триангуляцию, в которой минимальный угол максимален.

Линейность размера триангуляции

Утверждение. Пусть есть триангуляция множества из n сайтов, причём на границе выпуклой оболочки $conv\,(P)$ лежат k сайтов, и не все сайты лежат на одной прямой. Тогда в этой триангуляции ровно $2n-2-k$ треугольника и $3n-3-k$ рёбер.

Доказательство: Так как триангуляция в объединении даёт $conv\,(P)$, то триангуляция связна. Воспользуемся формулой Эйлера: $V-E+F=2$. Пусть в триангуляции m треугольников. Тогда:

$$n-E+(m+1)=2$$

$$E=\frac{3m+k}{2}$$

$$n-\frac{3m+k}{2}+m+1=2 \Longrightarrow m=2n-2-k$$

$$E=\frac{3(2n-2-k)+k}{2}=3n-3-k$$

■

Граф Делоне

Def. Пусть P — набор сайтов, $Vor\,(P)$ — диаграмма Вороного. Тогда граф Делоне — граф, обладающий свойствами:

1. Вершины представляют собой сайты;
2. Рёбра соединяют сайты, ячейки которых имеют общую сторону в виде ребра или луча.

Th. Граф Делоне является плоским.[8]

Def. Триангуляция Делоне — любая триангуляция, полученная из графа Делоне подразбиением всех граней на треугольники.

Лекция 14

О графе Делоне

Напомним определения с прошлой лекции, а также дополним их. Будем говорить, что две ячейки $V\,(p_i)$ и $V\,(p_j)$ являются соседними в диаграмме Вороного, если они пересекаются по части границы положительной длины.

Def. Граф Делоне — граф, обладающий свойствами:

[8] Доказательство будет на следующей лекции.

1. $V = \{p_1, \ldots, p_n\}$
2. $E = \{\{p_i, p_j\} : V(p_i), V(p_j) —$ соседние $\}$

Th. Граф Делоне — плоский граф.

Доказательство: Пусть не так, пусть p_ip_j и p_kp_l — пересекающиеся рёбра графа Делоне. Так как p_ip_j — ребро, то существует круг с центром в точке x_{ij}, такой что на границе лежат p_i, p_j и только они, и внутри нет точек. Интервал $x_{ij}p_i$ является подмножеством $V(p_i)$. Ни одна вершина треугольника T_{kl} не лежит внутри T_{ij}. Аналогично ни одна вершина треугольника T_{ij} не лежит внутри T_{kl}. Тогда один из отрезков p_ix_{ij} и p_jx_{ij} пересекает один из отрезков p_kx_{kl} и p_lx_{kl}. Но это означает, что две ячейки диаграммы Вороного пересекаются по внутренности. Противоречие. ■

Th. Верны следующие утверждения:

1. Сайты p_i, p_j, p_k являются вершинами одной (ограниченной) грани графа Делоне тогда и только тогда, когда существует точка q, такая что $C_q(P)$ содержит на границе точки p_i, p_j, p_k;
2. Отрезок p_ip_j является ребром графа Делоне тогда и только тогда, когда существует точка q на серединном перпендикуляре к отрезку p_ip_j, такая что $C_q(P)$ содержит на границе точки p_i и p_j и больше никого.

Следствие Любая (ограниченная) грань графа Делоне — вписанный в окружность выпуклый многоугольник.

Def. Любая триангуляция графа Делоне — триангуляция Делоне.

Note. Вне зависимости от триангуляции минимальный угол один и тот же.

Критерии триангуляции Делоне

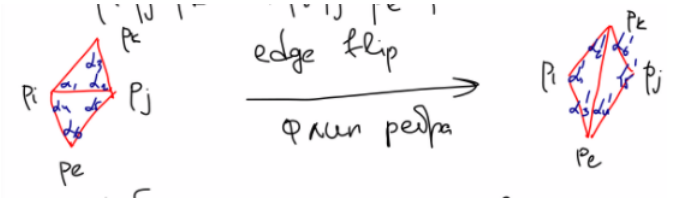
Th. (первый критерий, без доказательства) Триангуляция T является триангуляцией Делоне тогда и только тогда, когда круг, описанный вокруг любого треугольника из T не содержит внутри себя других вершин триангуляции.

Дальше потребуется определение:

Def. Пусть в триангуляции T есть два треугольника $\triangle p_ip_jp_k$ и $\triangle p_ip_jp_l$, и их объединение — выпуклый четырёхугольник. Тогда операция *edge flip* преобразует разбиение выпуклого четырёхугольника из двух треугольников $\triangle p_ip_jp_k$ и $\triangle p_ip_jp_l$ в $\triangle p_ip_kp_l$ и $\triangle p_jp_kp_l$.

Def. Ребро p_ip_j называется нелегальным, если после операции *edge flip* над ребром p_ip_j минимальный угол увеличился:

$$\min_t \alpha'_t > \min_t \alpha_t$$



Утверждение. (критерий легальности) Пусть в триангуляции T есть два треугольника $\triangle p_i p_j p_k$ и $\triangle p_i p_j p_l$. Тогда ребро $p_i p_j$ нелегально тогда и только тогда, когда круг, описанный вокруг $p_i p_j p_k$ содержит p_l строго внутри.

Доказательство: Используется теорема Фалеса. ■

Утверждение. (без доказательства) Пусть p, q, r — вершины треугольника, заданные при обходе по часовой стрелке. Тогда точка s лежит внутри круга, описанного вокруг треугольника, тогда и только тогда, когда следующий определитель положителен:

$$\begin{vmatrix} p_x & p_y & p_x^2 + p_y^2 & 1 \\ q_x & q_y & q_x^2 + q_y^2 & 1 \\ r_x & r_y & r_x^2 + r_y^2 & 1 \\ s_x & s_y & s_x^2 + s_y^2 & 1 \end{vmatrix}$$

Def. Триангуляция T называется легальной, если в ней нет нелегальных рёбер.

Утверждение. Из любой триангуляции можно построить легальную за конечное число операций *edge flip* над рёбрами.

Доказательство: Рассмотрим вектор всех углов. Пусть в триангуляции t треугольников, тогда $(\alpha_1, \alpha_2, \ldots, \alpha_{3t})$ — список углов в порядке возрастания. Каждое выполнение операции *edge flip* увеличивает этот список лексикографически. ■

Th. (второй критерий, без доказательства) Триангуляция T легальна, если и только если T — триангуляция Делоне.

Th. Триангуляция Делоне максимизирует минимальный угол.

Доказательство: Пусть T — триангуляция Делоне с минимальным углом α, T' — другая триангуляция с минимальным углом $\beta > \alpha$. Легализуем T', получим T'' с минимальным углом $\gamma \geqslant \beta > \alpha$. T'' — тоже триангуляция Делоне. Но минимальные углы триангуляций Делоне получились разными, а должны быть одинаковыми. Противоречие. ■

Алгоритм построения триангуляции Делоне

Алгоритм будет итеративным рандомизированным (вероятностным). Возьмём самую высокую точку p_0 в множестве сайтов P, среди нескольких точек с наибольшим y возьмём точку с наибольшим x. Возьмём фиктивныые точки p_{-1} и p_{-2} так, чтобы внутри треугольника $\triangle p_0 p_{-1} p_{-2}$ оказались все

оставшиеся сайты. Далее перемешаем сайты в случайном порядке. Поддержим легальную триангуляцию для множества сайтов $\{p_{-2}, p_{-1}, \ldots, p_{r-1}\}$ Если точка попала в треугольник, то разобьём этот треугольник на ещё несколько треугольников. Если ранее легальное ребро оказалось нелегальным, то это связано только с этой попавшей точкой. Выполним операцию $edge\ flip$. Проверим далее легальность других рёбер, и так до тех пор, пока не восстановится легальность.

Note. Этот алгоритм конечный. Легализаций происходит n раз.

Th. (без доказательства) Математическое ожидание времени работы алгоритма есть $O(n \log n)$, при этом количество операций $edge\ flip$ в среднем $O(n)$.

Основная сложность — локализация, то есть определение, в каком треугольнике лежит p_r.

```
random_shuffle(p[1], p[2], ..., p[n - 1]);
for r = 1, ..., n - 1:
  if (IsInside(p[r], Triangle(p[i], p[j], p[k]))) {
    DeleteTriangle(p[i], p[j], p[k]);
    AddTriangle(p[i], p[r], p[k]);
    AddTriangle(p[i], p[j], p[r]);
    AddTriangle(p[j], p[k], p[r]);
    LegalizeEdge(p[r], p[i], p[j]);
    LegalizeEdge(p[r], p[j], p[k]);
    LegalizeEdge(p[r], p[k], p[i]);
  }
  if (IsBorderPoint(p[r], Triangle(p[i], p[j], p[k]), Triangle(p[j], p[i], p[l]))) {
    delete 2 old triangles;
    add 4 new triangles;
    legalize 4 edges;
  }

LegalizeEdge(Point pr, Point pi, Point pj):
  Point pk; // vertex of the second triangle;
  if (IsIllegal(pi, pj)) {
    delete 2 old triangles;
    add 2 new triangles;
    LegalizeEdge(pr, pi, pk);
    LegalizeEdge(pr, pk, pj);
  }
```

Утверждение. После завершения всех легализаций рёбер получим легальную триангуляцию.

Доказательство: Это следует из того, что добавляемые рёбра одной из вершин содержат p_r. Такие рёбра обязательно лежат в графе Делоне. Остаётся доказать, почему утверждение о рёбрах выполняется.[9] ■

Идея доказательства: Рассмотрим точку p_r внутри круга C, на границе которого есть точки p_i, p_j, p_k. Сузим круг C до круга C', такого что C' содержит на границе точки p_i и p_r на границе и

[9] На экзамене этого не потребуется, однако на лекции рассказана общая идея.

больше никого. Аналогично можно было сузить до одного из двух других кругов в зависимости от точки.

Задача локализации

Задача. Узнать, в каком треугольнике (в каких двух треугольниках) лежит точка p_r.

Решение: Храним историю триангуляции в виде ориентированного ациклического графа. Каждая вершина такого графа соответствует треугольнику, который когда-то существовал. Треугольник может разбиться на три треугольника. После переворачивания ребра из вершин, соответствующих двум старым треугольникам, проведём рёбра в одни и те же две вершины, соответствующим новым треугольникам.

Процесс локализации происходит в несколько шагов:

1. Встать в корень;
2. Спуститься в такой дочерний треугольник, где лежит p_r;
3. Завершиться в листе, соответствующем треугольнику текущей триангуляции.

За счёт случайного порядка $p_1, \ldots, p_{n-1}$ граф будет иметь в среднем логарифмическую глубину.

Обработка фиктивных вершин

Будем говорить, что точка p находится выше точки q, если либо $p.y > q.y$, либо при $p.y = q.y$ $p.x < q.x$.

Формально, p_{-1} — точка ниже всех точек P, такая что из неё все точки в порядке обхода по часовой стрелке видны в порядке высоты. p_{-1} расположен настолько далеко, что не лежит ни в одном круге, построенным по трём точкам из P.

Формально говоря, p_{-2} — точка выше всех точек P, что из неё в порядке обхода против часовой стрелки все точки из $P \cup \{p_{-1}\}$ видны в порядке возрастания высоты. p_{-2} расположена достаточно далеко, что p_{-2} находится вне любого круга, построенного по трём точкам из $P \cup p_{-1}$.

Далее проверим легальность.

Возможно несколько случаев:

1. Если p_ip_j — ребро треугольника $p_{-2}p_0p_{-1}$, то p_ip_j легально;
2. Если i, j, k, $l \geqslant 0$, то делаем обычную проверку через определитель;
3. p_ip_j легально тогда и только тогда, когда $\min(k, l) < \min(i, j)$, среди i, j — не больше одного отрицательного, k, l — хотя бы одна — p_r — не больше одного отрицательного.