Курс

"Алгоритмы и структуры данных"

3 семестр ФПМИ МФТИ

Даниил Гагаринов

Артур Кулапин

github.com/yaishenka

github.com/kulartv

Осень 2019

Содержание

РК 1.

Префикс функция

Def. Префикс функция от строки S и индекса i - длина наибольшего (не равного всей подстроке) префикса подстроки $S[1..i]$, который одновременно является суффиксом этой подстроки. $(S[1..k] = S\,[(i-k+1)..i])$

Пример: $\pi(\texttt{abcdabscabcdabia}) = [0,0,0,0,1,2,0,0,1,2,3,4,5,6,0,1]$

Algorithm. Наивный алгоритм

```
int[] prefixFunction(string s):
int[] p = int[s.length]
fill(p, 0)
for i = 0 to s.length - 1
    for k = 0 to i - 1
        if s[0..k] == s[i - k..i]
            p[i] = k
return p
```

Algorithm. Эффективный алгоритм

Note. Заметим, что $p[i+1] \leqslant p[i]+1$. Чтобы показать это, рассмотрим суффикс,оканчивающийся на позиции $i+1$ и имеющий длину $p[i+1]$, удалив из него последний символ, мы получим суффикс, оканчивающийся на позиции i и имеющий длину $p[i+1]-1$, следовательно неравенство $p[i+1] > p[i]+1$ неверно.

Note. Избавимся от явных сравнений строк. Пусть мы вычислили $p[i]$, тогда, если $s[i+1] = s[p[i]]$, то $p[i+1] = p[i]+1$. Если окажется, что $s[i+1] \neq s[p[i]]$, то нужно попытаться попробовать подстроку меньшей длины. Хотелось бы сразу перейти к такому наибольшей длины, для этого подберем такое k, что $k = p[i]-1$. Делаем это следующим образом. За исходное k необходимо взять $p[i-1]$, что следует из первого пункта. В случае, когда символы $s[k]$ и $s[i]$ не совпадают, $p[k-1]$ следующее потенциальное наибольшее значение k, что видно из рисунка. Последнее утверждение верно, пока $k > 0$, что позволит всегда найти его следующее значение. Если $k = 0$, то $p[i] = 1$ при $s[i] = s[1]$, иначе $p[i] = 0$.

```
int[] prefixFunction(string s):
p[0] = 0
for i = 1 to s.length - 1
        k = p[i - 1]
```

```
        while k > 0 and s[i] != s[k]
                k = p[k - 1]
        if s[i] == s[k]
                k++
        p[i] = k
return p
```

Утверждение. Время работы алгоритма - O(n)

Доказательство: Для доказательства этого нужно заметить, что итоговое количество итераций цикла while определяет асимптотику алгоритма. Теперь стоит отметить, что k увеличивается на каждом шаге не более чем на единицу, значит максимально возможное значение $k = n - 1$. Поскольку внутри цикла while значение k лишь уменьшается, получается, что k не может суммарно уменьшиться больше, чем $n - 1$ раз. Значит цикл while в итоге выполнится не более n раз, что дает итоговую оценку времени алгоритма $O(n)$.

Алгоритм Кнута-Морриса-Пратта

Задача. Дана цепочка T и образец P. Требуется найти все позиции, начиная с которых P входит в T.

Algorithm. Построим строку $S = P\#T$, где $\#$ — любой символ, не входящий в алфавит P и T. Посчитаем на ней значение префикс-функции p. Благодаря разделительному символу $\#$, выполняется $\forall i : p[i] \leqslant |P|$. Заметим, что по определению префикс-функции при $i > |P|$ и $p[i] = |P|$ подстроки длины P, начинающиеся с позиций 0 и $i - |P| + 1$, совпадают. Соберем все такие позиции $i - |P| + 1$ строки S, вычтем из каждой позиции $|P| + 1$, это и будет ответ. Другими словами, если в какой-то позиции i выполняется условие $p[i] = |P|$, то в этой позиции начинается очередное вхождение образца в цепочку.

Algorithm. Эффективный алгоритм КМП. Найдем префикс функцию от паттерна = $p[]$. Посмотрим на строку $S = P\#T$. Любой префикс начиная с позиции $|P| + 1$ - префикс строки P. То есть для построения префикс функции такой строки нам нужны только значения префикс функции для паттерна. Считаем следующий символ строки. Посчитаем для него префикс функцию, используя значения префикс функции для паттерна. Если значение = $|P|$, то запишем эту позицию.

```
std::vector<int> GetAllEntries(std::string pattern, size_t text_size) {
  std::vector<int> entries;
  auto pattern_pi = GetPrefixfunction(pattern);

  size_t last_prefix_function_value = 0;
```

```
  for (size_t i = 0; i < text_size; ++i) {
    size_t current_value = last_prefix_function_value;

    char next_symbol;
    std::cin >> next_symbol;

    while (pattern_pi[current_value] > 0 &&
        next_symbol != pattern[current_value]) {
      current_value =
          pattern_pi[current_value - 1];
    }

    if (next_symbol == pattern_pi[current_value]) {
      last_prefix_function_value = current_value + 1;
    } else {
      last_prefix_function_value = 0;
    }

    if (last_prefix_function_value == pattern.size()) {
      entries.push_back(i - pattern.size() + 1);
    }
  }
}
```

Z-функция

Def. Z-функция от строки s и индекса i - длина наиболее длинного префикса начинающегося с позиции i суффикса строки S, который одновременно является и префиксом всей строки S. Обычно считается, что $Z(s, 0) = 0$

Пример: $Z(\texttt{abcdabscabcdabia}) = [0, 0, 0, 0, 2, 0, 0, 0, 6, 0, 0, 0, 2, 0, 0, 1]$

Algorithm. Наивный алгоритм

```
int[] zFunction(s : string):
int[] zf = int[n]
for i = 1 to n - 1
  while i + zf[i] < n and s[zf[i]] == s[i + zf[i]]
    zf[i]++
return zf
```

Note. Нетрудно заметить, что наивный алгоритм работает за квадратичное время.

Algorithm. Линейный алгоритм

Z-блоком назовем подстроку с началом в позиции i и длиной $Z[i]$. Для работы алгоритма заведём две переменные: $left$ и $right$ — начало и конец Z-блока строки S с максимальной позицией конца $right$ (среди всех таких Z-блоков, если их несколько, выбирается наибольший). Изначально $left = 0$ и $right = 0$. Пусть нам известны значения Z-функции от 0 до i-1. Найдём $Z[i]$. Рассмотрим два случая:

1. Пусть $i > right$, тогда просто пробегаемся по строке S и сравниваем символы на позициях $S[i + j]$ и $S[j]$. Пусть j первая позиция в строке S для которой не выполняется равенство $S[i+j] = S[j]$, тогда j это и Z-функция для позиции i. Тогда $left = i$, $right = i+j-1$. В данном случае будет определено корректное значение $Z[i]$ в силу того, что оно определяется наивно, путем сравнения с начальными символами строки.

2. Пусть $i \leqslant right$, тогда сравним $Z[i-left]+i$ и $right$. Если $right$ меньше, то надо просто наивно пробежаться по строке начиная с позиции $right$ и вычислить значение $Z[i]$. Корректность в таком случае также гарантирована. Иначе мы уже знаем верное значение $Z[i]$, так как оно равно значению $Z[i - left]$.

Заметим, что данный алгоритм обращается к каждому символу строки не более двух раз, поэтому асимптотика алгоритма не выше $O(|S|)$. Далее будет приведен псевдокод алгоритма выше:

```
int[] zFunction(s : string):
int[] zf = int[n]
int left = 0, right = 0
```

```
for i = 1 to n - 1
  zf[i] = max(0, min(right - i, zf[i - left]))
  while i + zf[i] < n and s[zf[i]] == s[i + zf[i]]
    zf[i]++
  if i + zf[i] > right
    left = i
    right = i + zf[i]
return zf
```

Algorithm. Алгоритм Кнута-Морриса-Прата, вариант 2

1. Построим Z-функцию по строке (pattern + '#' + text).
2. Тогда по определению, если значение Z-функции для данного индекса равно длине паттерна, то мы нашли начало вхождения паттерна в текст.

Algorithm. Построение строки по Z-функции.

Пусть в массиве z хранятся значения Z-функции, в s будет записан ответ. Пойдем по массиву z слева направо. Для правильной работы алгоритма будем считать, что $z[0] = 0$.

Нужно узнать значение $s[i]$. Для этого посмотрим на значение $z[i]$: если $z[i] = 0$, тогда в $s[i]$ запишем ещё не использованный символ или последний использованный символ алфавита, если мы уже использовали все символы. Если $z[i] \neq 0$, то нам нужно записать префикс длины $z[i]$ строки s. Но если при посимвольном записывании этого префикса в конец строки s мы нашли такой j (индекс последнего символа строки), что $z[j]$ больше, чем длина оставшейся незаписанной части префикса, то мы перестаём писать этот префикс и пишем префикс длиной $z[j]$ строки s.

Докажем этот алгоритм:

1. Если выяснится, что $z[i] = k$ $(k > 0)$ и $\forall l : l \in (0, k)$ $z[i + l] = 0$, то получится, что префикс с позиции i никак не влияет на префикс, который может начинаться с позиции $i+k$. Соответствует случаю, когда два соседних префикса ненулевой длины не пересекаются.
2. Если же один из префиксов целиком вложен в больший префикс, то восстановив большой префикс мы автоматически восстановим малый.
3. Если префиксы пересекаются, то нам нужно переписать часть префикса, который начинается раньше, и начать писать другой префикс (начало этого префикса запишет конец префикса, начинающегося раньше). Это будет корректным решением, так как мы делаем аналогичные обратные действия, что и при построении Z-функции линейным алгоритмом (см. определение Z-блока).

Algorithm. Построение Префикс-функции по Z-функции.

Пусть Z-функция хранится в массиве $z[0 \dots n-1]$. Префикс-функцию будем записывать в массив $p[0 \dots n-1]$. Заметим, что если $z[i] > 0$, то для всех элементов с индексом $i+j$, где $0 \leqslant j < z[i]$, значение $p[i+j]$ будет не меньше, чем длина подстроки с i по $i+j$, что равно $j+1$ (как изображено на рисунке).

Также заметим, что если мы уже установили в какую-то позицию значение j с позиции i, а потом пытаемся установить значение j' с позиции i', причём $i < i'$ и $i + j = i' + j'$, то изменение с позиции i' только уменьшит значение $p[i+j]$. Действительно, значение после первого присвоения $p[i+j] = j > j' = p[i'+j']$. В итоге получаем алгоритм: идем слева направо по массиву z и, находясь на позиции i, пытаемся записать в p от позиции $i + z[i] - 1$ до i$j + 1$, где j пробегает все значения $0 \dots z[i] - 1$, пока не наткнемся на уже инициализированный элемент. Слева от него все значения тоже нет смысла обновлять, поэтому прерываем эту итерацию.

Убедимся, что алгоритм работает за линейное время. Каждый элемент устанавливается ровно один раз. Дальше на нем может случиться только break. Поэтому в итоге внутренний цикл суммарно отработает за количество установленных значений и количество break. Количество установленных значений — n. А число break тоже будет не больше n, так как каждый break переводит внешний цикл на следующую итерацию, откуда получаем итоговую асимптотику $O(n)$.

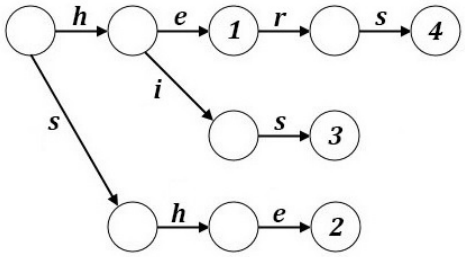
Псевдокод данного алгоритма:

```
int[] buildPrefixFunctionFromZFunction(int[] z):
int[] p = int[z.length]
fill(p, 0)
for i = 1 to z.length - 1
  for j = z[i] - 1 downto 0
    if p[i + j] > 0
      break
    else
      p[i + j] = j + 1
return p
```

Бор

Def. Бор - структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной. Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца.

Пример: Бор для набора образцов he,she,his,hers



Утверждение. Построение $O(|P|)$, где $|P| = \sum\limits_{w \in dict} |w|$

Утверждение. Память $O(|P| * |\sum|)$

Алгоритм Ахо-Корасик.

Суффиксная ссылка. Построение бора. Построение суффиксной ссылки. Оценка времени работы.

Def. Пусть $[u]$ - слово, которое составляет путь от корня до u в боре

Def. Тогда $\pi(u) = \{v | [v] - \text{суффикс } [u], |[v]| \text{ - максимальная}\}$ - суффиксная ссылка

Note. $\pi(u) = \delta(\pi(u \to parent), c)$, где c - символ, по которому был переход в u.

Построим суффиксные ссылки обходом в ширину. O(n)

Сжатые суффиксные ссылки.

$$up(u) = \begin{cases} \pi(u), & \text{if } \pi(u) \text{ is terminal;} \\ \varnothing, & \text{if } \pi(u) \text{ is root;} \\ up(\pi(u)), & \text{else.} \end{cases}$$

Algorithm. Алгоритм Ахо-Корасик

По очереди просматриваем символы текста. Для очередного символа c переходим из текущего состояния u в состояние, которое вернёт функция $\delta(u, c)$. Оказавшись в новом состоянии, отмечаем по сжатым суффиксным ссылкам строки, которые нам встретились и их позицию (если требуется). Если новое состояние является терминалом, то соответствующие ему строки тоже отмечаем.

$$\text{Где } \delta(u,c) = \begin{cases} v, & \text{if } v \text{ is son by symbol } c \text{ in trie;} \\ root, & \text{if } u \text{ is root and } u \text{ has no child by symbol } c \text{ in trie} \\ \delta(\pi(u),c), & \text{else.} \end{cases}$$

Утверждение. Ассимптотика на массивах O(|T| + |P||$\sum$| + t), где t - максимальное число вхождений

Будем хранить не массив, а map. Тогда общая память $|P|$.

Утверждение. тогда время будет O((|T| + |P|)log|$\sum$| + t)

Существует несколько оптимизаций данного алгоритма, направленных на случаи, когда нас интересует только первое вхождение образца в текст:

1. Сброс сжатых суффиксных ссылок для посещённых вершин. Существенно ускорить работу алгоритма могут пометки о посещённости узла, то есть если узел посещён, то не переходить по сжатым суффиксным ссылкам. Вместо хранения пометок можно просто сбрасывать сжатую суффиксную ссылку.

2. Сброс пометки терминальной вершины. В изначальном множестве образцов могут быть дублирующиеся строки. Мы можем хотеть их различать, если с одинаковыми строками связана разная мета-информация. Тогда при попадании в терминальную вершину можно осуществлять сброс пометки этой терминальной вершины, что сэкономит время на обновлении информации о вхождении образцов в текст. Тривиальным примером, в котором возникает ситуация долгой обработки, служит огромное множество образцов из одинаковых символов и текст только из этих символов.

РК 2.

Суффиксный массив

Def. Суффиксным массивом строки $s[1 \dots n]$ называется массив $Suff$целых чисел от 1 до n, такой, что суффикс $s[Suff[i] \dots n]$ — i-й в лексикографическом порядке среди всех непустых суффиксов строки s.

Пример: $Suff(abaab) = (2, 3, 0, 4, 1)$. Рассмотрим суффиксы строки $S = abaab$ в отсортированном порядке: $aab, ab, abaab, b, baab$, тогда нетрудно убедиться, что суффикс aab начинается со второй позиции (или $S[2 \dots N-1] = aab$)

Algorithm. Наивный алгоритм за $O(n^2 logn)$.

Просто отсортировать все суффиксы строки. Сортировка требует $O(nlogn)$ сравнений, каждое из них $O(n)$ времени.

Algorithm. Эффективный алгоритм за $O(nlogn)$.

Здесь стоит отметить, что далее будет проводиться сортировка **циклических сдвигов строки** S, чтобы далее суметь получить по данной сортировке сортировку именно суффиксов, необходимо дописать к строке S в конец символ, который меньше всех символов из алфавита (обычно этим символом является \$ в реализации). Далее будем считать, что строка S уже имеет \$ в конце.

Сразу заметим, что поскольку мы сортируем циклические сдвиги, то и подстроки мы будем рассматривать циклические: под подстрокой $S[i \dots j]$, когда $i > j$, понимается подстрока $s[i \dots n-1]$ + $s[0 \dots j]$ (конкатенация двух подстрок, сначала то есть берется суффикс, а потом дописывают префикс, чтобы длина исходной строки не изменилась, ибо сдвиг не меняет длины).

Данный алгоритм имеет в себе примерно $logn$ фаз. Далее номер фазы обозначим за $k \in \{0, \dots, \lceil logn \rceil\}$. Сортируются циклические подстроки длины 2^k. На последней фазе будут сортироваться подстроки длины $2^{\lceil logn \rceil} \geqslant n$, что эквивалентно сортировке циклических сдвигов (чего мы и хотим достичь в итоге).

На каждой фазе алгоритм помимо перестановки p[0 ... n - 1] индексов циклических подстрок будет поддерживать для каждой циклической подстроки, начинающейся в позиции i с длиной 2^k, **номер** $c[i]$ **класса эквивалентности**, которому эта подстрока принадлежит. В самом деле, среди подстрок могут быть одинаковые, и алгоритму понадобится информация об этом. Кроме того, номера c[i] классов эквивалентности будем давать таким образом, чтобы они сохраняли и информацию о порядке: если один суффикс меньше другого, то и номер класса он должен получить меньший. Классы будем для удобства нумеровать с нуля.

На нулевой фазе мы должны отсортировать циклические подстроки длины 1 и разделить их на классы эквивалентности (просто одинаковые символы должны быть отнесены к одному классу эквивалентности). Это можно сделать тривиально, например, сортировкой подсчётом. Для каждого символа посчитаем, сколько раз он встретился. Потом по этой информации восстановим массив $p[]$. После этого, проходом по массиву $p[]$ и сравнением символов, строится массив $c[]$.

Далее, пусть мы выполнили фазу $k-1$ (т.е. вычислили значения массивов $p[]$ и $c[]$ для неё), теперь научимся за $O(n)$ выполнять следующую, фазу с номером k. Для этого заметим, что циклическая подстрока длины 2^k состоит из двух подстрок длины 2^{k-1}, которые мы можем сравнивать между собой за $O(1)$, используя информацию с предыдущей фазы — номера $c[]$ классов эквивалентности. Таким образом, для подстроки длины 2^k, начинающейся в позиции i, вся необходимая информация содержится в паре чисел $(c[i], c[i+2^{k-1}])$ (используется массив $c[]$ с предыдущей фазы, все индексы берутся по модулю длины строки).

Тогда надо отсортировать подстроки длины 2^k просто по этим парам чисел, это и даст нам требуемый порядок, т.е. массив $p[]$. Поскольку элементы пар не превосходят n, то можно выполнить сортировку подсчётом (или же с помощью LSD), то есть линейное время на переход между фазами, а таких переходов $O(logn)$, то есть итого $O(nlogn)$ времени на построение.

Пример: Построим массивы $p[]$ и $c[]$ для строки $S = aaba$ (в данном случае доллар при анализе игнорируется).

$$k = 0: \ p = (0, 1, 3, 2); \ c = (0, 0, 1, 0);$$

$$k = 1: \ p = (0, 3, 1, 2); \ c = (0, 1, 2, 0);$$

$$k = 2: \ p = (3, 0, 1, 2); \ c = (1, 2, 3, 0);$$

Разберемся на примере перехода с фазы $k = 1$ на фазу $k = 2$. Рассмотрим пары из пар $(c[i], c[i + 2^{k-1}])$ и i: (02, 0), (10, 1), (20, 2), (01, 3). Посортируем пары по первым элементам (которые тоже пары), получим следующий набор: (01, 3), (02, 0), (10, 1), (20, 2). Тогда сразу заметим, что массив $p[]$ уже получен (в силу определения массива c[] на шаге $k-1$) — это просто вторые компоненты пар $p_k[] = (3, 0, 1, 2)$.

Теперь расставим в соответствие парам их новые классы эквивалентности: пойдем по порядку отсортированного массива из пар и будем писать им в соответствие новые классы эквивалентности (в случае равенства пар они будут повторяться, но не в этом случае) $(01, 0)$, $(02, 1)$, $(10, 2)$, $(20, 3)$. Чтобы восстановить $c_k[]$, второй элемент пары из массива выше задаст значение, которое лежать в $c_k[]$ по индексу, который равен второму элементу пары из массива пар в абзаце выше по соответствующему первому элементу, то есть $c_k[] = (1, 2, 3, 0)$.

Поиск подстроки в строке с помощью суффмасса

Algorithm. Поиск подстроки в строке с помощью суффмасса

Простейший способ узнать, встречается ли образец в тексте, используя суффиксный массив, — взять первый символ образца и бинарным поиском по суффиксному массиву найти диапазон с суффиксами, начинающимися на такую же букву. Так как все элементы в полученном диапазоне отсортированы, а первые символы одинаковые, то оставшиеся после отбрасывания первого символа суффиксы тоже отсортированы. А значит, можно повторять процедуру сужения диапазона поиска уже по второму, затем третьему и так далее символу образца до получения либо пустого диапазона, либо успешного нахождения всех символов образца.

Бинарный поиск работает за время равное $O(\log |s|)$, а сравнение суффикса с образцом не может превышать длины образца.

Таким образом время работы алгоритмы $O(|p| \log |s|)$, где s — текст, p — образец.

Алгоритм Касаи и ко. (пяти азиатов)

Def. Алгоритм Касаи, Аримуры, Арикавы, Ли, Парка — алгоритм, позволяющий за линейное время вычислить длину наибольших общих префиксов (англ. longest common prefix, LCP) для всех соседних суффиксов строки, отсортированных в лексикографическом порядке.

Для пояснения алгоритма введем следующие обозначения:

S_i — суффикс строки S, начинающийся в позиции i.

$Suff[]$ — суффиксный массив.

$Suff^{-1}[]$ — обратный суффиксный массив (если $Suff[i] = k$, то $Suff^{-1}[k] = i$)

$LCP(S_i, S_j)$ — длина наибольшего общего префикса S_i и S_j (обозначения см. выше)

$lcp[i] = LCP(S_{Suff[i-1]}, S_{Suff[i]})$ — LCP для соседних строк в порядке суфмассива.

Algorithm. Алгоритм Касаи

Для начала теорема, которая необходима для доказательства линейной асимптотики:

Если $lcp[Suff^{-1}[i-1]] = LCP(S_{Suff[Suff^{-1}[i-1]-1]}, S_{i-1}) > 1$, то
$lcp[Suff^{-1}[i]] = LCP(S_{Suff[Suff^{-1}[i]-1]}, S_{Suff[i]}) \geqslant lcp[Suff^{-1}[i-1]] - 1$.

А теперь разберемся и осознаем, что нам проще заполнять LCP в порядке $Suff^{-1}$, тогда заменим $Suff^{-1}[i]$ на k и $Suff^{-1}[i-1]$ на $k-1$. После такой замены перепишем теорему:

Если $lcp[k-1] > 1$, то $lcp[k] \geqslant lcp[k-1] - 1$.

Будем вычислять массив LCP, зная уже $Suff$. Для начала за линейное время построим $Suff^{-1}$. Теперь в силу теоремы выше заметим, что, чтобы вычислить очередное значение $lcp[i]$, нам необязательно сравнивать суффиксы $S_{Suff^{-1}[i-1]}$ и $S_{Suff^{-1}[i]}$ с самого начала, достаточно сравнивать суффиксы, начиная со значения $lcp[Suff^{-1}[i-1]] - 1$. Здесь предлагается аналогичная замена обозначений, но в реализации алгоритма это так и будет выглядеть.

Здесь и далее считается, что итерация идет по k, а $i = Suff^{-1}[k], i-1 = Suff^{-1}[k-1]$. Реализация алгоритма заключается в том, что если $lcp[i-1] \leqslant 1$, то просто идем по двум суффиксам с самого начала, пока они равны — получили $lcp[i]$, иначе пользуясь теоремой выше идем по двум суффиксам, только начиная не с начала, а с $lcp[i-1] - 1$.

На каждой итерации текущее значение LCP может быть не более чем на единицу меньше предыдущего. Таким образом, значения LCP в сумме могут увеличиться не более, чем на $2n$ (с точностью до константы). Следовательно, алгоритм построит LCP за линейное время.

Суффиксное дерево

Def. Суффиксное дерево для строки S — бор, содержащий все суффиксы строки S (и только их). Так как суффиксное дерево может требовать памяти порядка квадрата от длины строки, оно не очень полезно.

Def. Сжатое суффиксное дерево — структура данных, оптимизированная относительно обычного суффиксного дерева, достоинством которой является линейные требования по памяти.

Def. Сжатое суффиксное дерево T для строки S (где $|S| = n$) — дерево с n листьями, обладающее

следующими свойствами:

1. каждая внутренняя вершина дерева имеет не меньше двух детей;
2. каждое ребро помечено непустой подстрокой строки s;
3. никакие два ребра, выходящие из одной вершины, не могут иметь пометок, начинающихся с одного и того же символа;
4. дерево должно содержать все суффиксы строки s, причем каждый суффикс заканчивается точно в листе и нигде кроме него.

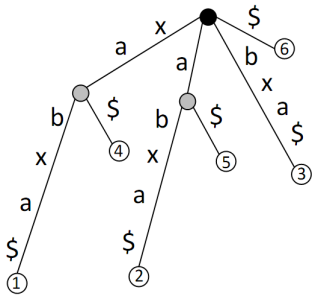
Для устранения в боре неоднозначности, где заканчивается данный суффикс, используют символ доллара (см. раздел про Суффиксный массив).

Нетрудно заметить, что так как суффиксов n, то и листьев в дереве тоже n.

Утверждение. Число внутренних вершин дерева, каждая из которых имеет не менее двух детей, меньше числа листьев (индукция по длине строки). Так как это дерево, то и число ребер тоже линейно.

Как следствие, всего вершин порядка $O(n)$. Теперь опишем, как будем хранить дерево. Хранят его обычно в двумерном массиве T размера $|V| * |\Sigma|$, где V — множество вершин. Каждая ячейка $T[i][j]$ данной матрицы хранит в себе следующую информацию: ребро, которое ведет из вершины i по символу j, а также два индекса $(left, right)$, чтобы знать, какая подстрока лежит на ребре. Таким образом, память линейна, так что **далее сжатое суффиксное дерево будем называть обычным деревом.**

Пример: Сжатое суффиксное дерево для строки $xabxa$:



Algorithm. Тривиальный алгоритм построения.

Заметим, что тривиальное построение бора линейно по суммарной длине слов, содержащихся в нем. Нетрудно заметить, что в суффиксном боре длина слов, на которых он построен тривиальным образом, порядка n^2, а значит классический алгоритм построит такое дерево за квадратичное время (но не отчаивайтесь! Укконен всех Уккоконет, при этом за линейное время)

Algorithm. Нетривиальный неэффективный алгоритм построения.

Казалось бы, как так вышло, что нам нужен кубический по сложности алгоритм построения, если есть выше квадратичный, но есть надежда, что нам улучшат кубическую сложность до линейной. Так вот, будем строить суффиксное дерево (алгоритмом выше) для каждого из префиксов. Тогда в итоге сложность $O(n^3)$, так как n раз вызываем квадратичное построение.

Def. Неявное суффиксное дерево по строке S$ — это дерево, полученное из суффиксного дерева путем удаления символа $ отовсюду, потом удаляются ребра без символов и удаляются вершины, имеющие меньше двух детей.

Для примера выше неявное суффиксное дерево — это три ребра, торчащие из корня, на которых написано bxa, abxa, xabxa.

Def. T_i — неявное суффиксное дерево строки $S[1 \dots i]$.

Note. Далее индексация будет вестись с единицы, так как автор книги так захотел.

Алгоритм Укконена

Общая идея

Алгоритм Укконена делится на n фаз. На фазе $i+1$ строится дерево T_{i+1} из T_i. При этом каждая фаза делится на $i+1$ продолжение, по одному на каждый суффикс $S[1 \dots i+1]$. В продолжении j фазы $i+1$ алгоритм сначала находит конец пути из корня, помеченного подстрокой $S[j \dots i]$, затем он продолжает данную ветку, добавляя в ее конея символ $S[i+1]$, если таковое ответвление еще не существовало. Таким образом, в продолжениях 1, 2... помещаются в дерево строки $S[1 \dots i+1], S[2 \dots i+1] \dots$ Продолжение с номером $i+1$ вставляет символ $S[i+1]$, если таковой в дереве еще не существовал. T_1 — дерево из одного ребра, помеченного символом $S[1]$.

Опишем правила продолжения суффикса. В этом разделе считаем, что на фазе $i+1$ на продолжении j $S[j \dots i] = Suff$ — суффикс $S[1 \dots i]$:

1. В текущем дереве путь $Suff$ заканчивается в листе, значит необходимо добавить к этому ребру букву $S[i+1]$ в конец метки.

2. Ни один путь из конца $Suff$ не начинается на букву $S[i+1]$, но хотя бы один путь из этой вершины уже есть, тогда сделаем новое ответвление из конца $Suff$, при этом пометим это ребро символом $S[i+1]$. Если выяснилось, что $Suff$ заканчивался внутри ребра, то делаем новую вершину и разбиваем метку ребра на две (см. пример).

3. Если в конце $Suff$ уже есть путь или в том же ребре имеется символ $S[i+1]$, то ничего делать не нужно.

Пример: Неявные деревья для строк $axabx$ и $axabxb$.

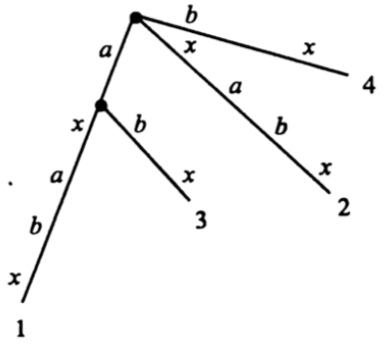


рис. 1

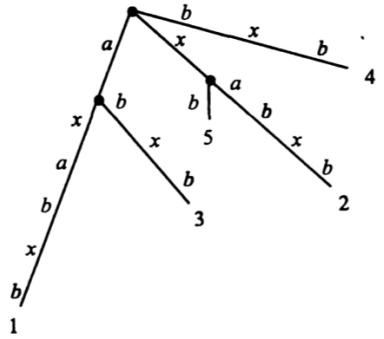


рис. 2

На данном примере первые четыре префикса оканчиваются в листьях, пятый — в дуге, а в шестом префиксе первые четыре суффикса получают продолжение по правилу 1, пятый — по правилу 2 (суффикс xb), а шестой — по номеру 3 (просто b).

Эвристики

Далее будут рассмотрены эвристики, уменьшающие сложность по времени с кубической до линейной:

1. **Суффиксная ссылка** для вершины u — вершина $s(u)$ такая, что если до вершины u путь идет по строке $x\alpha$, где α — строка произвольной длины (может и пустая), то до вершины $s(u)$ идет путь по строке α.

   Можно доказать, что для каждой внутренней вершины суффиксная ссылка уже определена на момент завершения фазы.

   Рассмотрим для произвольной $i+1$ фазы ее первое продолжение: это по сути дописывание символа $S[i+1]$ к подстроке $S[i]$, а на нее можно хранить указатель, как на самую длинную строку в дереве, то есть время константное. Рассмотрим второе продолжение: пусть γ — текст (далее метка) на ребре (v, 1), где 1 — лист, содержащий всю строку $S[1 \dots i]$. Тогда пройдем по γ вверх по ребру (1, v), потом пройдем по суффиксной ссылке в $s(v)$ (она нам известна, так как v — внутренняя вершина или корень). А далее спускаемся по γ — возможно в силу построения дерева.

   Пример: На рис. 2 при поиске, куда вставить суффикс xabxb заметим, что вершина v — вершина между корнем и листом 1, а $s(v)$ — вершина, от которой идет ответвление на суффикс xb. Тогда заметим, что продолжение $\gamma = abxb$ можно найти просто спустившись до листа.

   Рассмотрим более поздние продолжения: на них аналогично второму действуем, то есть на одну верину вверх, потом по суффиксной ссылке, а дальше спускаемся вниз по продолжению. Единственное отличие, что если при порождении нового суффикса была создана новая внутренняя вершина (см. пример), то необходимо построить для нее сразу суффиксную ссылку.

   К сожалению, в худшем случае асимптотика еще не улучшена:(

2. **Скачок по счетчику.** Пусть $g = |\gamma|$, тогда зная первую букву γ можно сразу прыгнуть к вершине вниз по ребру, начинающемуся с данной буквы из вершины $s(v)$. Зная длину этого ребра (g'), полагаем счетчик $g = g - g'$. Делаем так, пока не пришли к ситуации, что остаток букв на ребре короче суффикса строки. Тогда просто переходим к символу с номером оставшегося счетчика.

   Суть в том, что теперь мы умеем спускаться не пропорционально длине суффикса, а пропорционально числу вершин в пути.

   Утверждение. При использовании скачков любая фаза алгоритма за $O(m)$.

   Note. Заметим, что асимптотика по памяти до сих пор линейна, это успех!

3. **Правило правила 3.** Заметим, что если на фазе $i + 1$ суффикс добавлялся по правилу 3 на продолжении j, то данный суффикс уже есть целиком, а значит можно прекращать фазу.

   Note. Данная эвристика еще не помогает в худшем случае сделать асимптотику линейной, но в среднем сильно ускоряет алгоритм.

4. **Правило листа.** Заметим, что если алгоритм породил лист, то заданная вершина навсегда останется листом, то есть на ребро можно накидывать буквы, но новых разветвлений в листе быть не может. То есть если имеется лист с пометкой k, то на всех последующих фазах для продолжения k будет применяться правило 1.

   Тогда заметим, что для каждой следующей фазы k не убывает. Тогда можно построить нестрого возрастающую последовательность из k_i такую, что на фазе $i + 1$ все продолжения по правилу 1 можно заменить на изменение в листе с номером k_i индекса подстроки, которая в нем оканчивается до $i + 1$.

Итоговый алгоритм

1. По правилу листа скипаем все продолжения на фазе $i + 1$ до продолжения j_i.
2. С помощью скачков по счетчику и суффиксным ссылкам пропрыгаем, пока не добрались до правила 3 для добавления суффикса (положим, что это произошло на продолжении j').
3. По правилу правила 3 скипнем все остальное, при этом нетрудно осознать (если твоя фамилия Укконен), что $j_{i+1} = j' - 1$.
4. Повторить первые три шага по числу продолжений, то есть m раз.

Утверждение. Данный алгоритм работает за линейное время, так как суммарное число явных продолжений (стадия 2 алгоритма) не больше $2m$, а такое продолжение требует константного времени (скачок по счетчику и усредняем на фазу).

Заметим, что мы построили неявное суффиксное дерево! Теперь перейдем к явному. Для этого достаточно дописать к исходной строке знак $ и сделать еще один шаг алгоритма, тогда все суффиксы станут листами и закроются долларами, так как не будет суффиксов, которые префиксы других суффиксов.

Утверждение. Алгоритм Укконена строит сжатое суффиксное дерево за линию!

Note. Внимание, далее детали реализации, не повторяйте этот трюк в домашних условиях!

Вы думали, что все? А нет! Еще осталось для всех листьев поменять их глобальную константу, которая появляется в реализации правила листа) Это делается обходом по дереву за линейное время.

РК 3.

Выпуклая оболочка

Def. Выпуклая оболочка для набора точек — многогранник минимального объема, внутри которого (или на его границе) лежат все точки из набора.

Выпуклая оболочка в 2D

Def. Выпуклая оболочка для набора точек на плоскости — многоугольник минимальной площади, внутри которого (или на его границе) лежат все точки из набора.

Алгоритм Джарвиса

Пусть дано множество точек $P = \{P_1, P_2, \ldots, P_n\}$.

1. В качестве начальной берётся самая левая нижняя точка P_1 (её можно найти за $O(n)$ обычным проходом по всем точкам), она точно является вершиной выпуклой оболочки.
2. Следующей точкой P_2 берём такую точку, которая имеет наименьший положительный полярный угол относительно точки P_1 как начала координат. После этого для каждой точки P_i ($2 < i <= |P|$) против часовой стрелки ищется такая точка P_{i+1}, путём нахождения за $O(n)$ среди оставшихся точек (+ самая левая нижняя), в которой будет образовываться наибольший угол между прямыми P_{i-1}, P_i и P_i, P_{i+1}. Она и будет следующей вершиной выпуклой оболочки.
3. Нахождение вершин выпуклой оболочки продолжается до тех пор, пока $P_{i+1} \neq P_1$.

В тот момент, когда следующая точка в выпуклой оболочке совпала с первой, алгоритм останавливается — выпуклая оболочка построена.

Алгоритм Грэхэма

Пусть дано множество точек $P = \{P_1, P_2, \ldots, P_n\}$.

1. Найдем самую нижнюю (затем самую левую) точку из набора P_0, она гарантированно входит в выпуклую оболочку.
2. Проводим сортировку точек по углу между векторами $\overrightarrow{P_0P_i}, \overrightarrow{OX}$. Получим набор $P_1, P_2, \ldots P_n$
3. Заведем стек точек. Добавим в него точки P_0, P_1.
4. Будем строить оболочку против часовой стрелки. Тогда будем добавлять в стек точки, пока поворот от вектора $\overrightarrow{P_{s-1}P_s}$ до $\overrightarrow{P_sP_k}$ будет левым. Тут P_{s-1} и P_s — точки под вершиной стека и на ней соответственно, а P_k — точка из исходного набора, которая ранее не рассматривалась.
5. Если получилось, что поворот от $\overrightarrow{P_{s-1}P_s}$ до $\overrightarrow{P_sP_k}$ правый, то удалим из стека P_s. Далее будем рассматривать поворот от $\overrightarrow{P_{s-2}P_{s-1}}$ до $\overrightarrow{P_{s-1}P_k}$. В зависимости от ситуации будем делать либо шаг 4, либо шаг 5.
6. Тогда пройдясь таким образом по всем точкам из набора, в стеке останутся точки, образующие выпуклую оболочку.

Выпуклость следует из того, что все повороты полученного многоугольника левые, а тот факт, что мы получили именно оболочку, напрямую вытекает из построения.

Асимптотическая сложность этого алгоритма складывается из сортировки и прохода стеком. Проход стеком линеен, так как каждая точка может быть либо добавлена туда, либо выкинута оттуда и не добавлена вновь.

Значит итоговая сложность алгоритма: $O(NlogN)$ — сложность сортировки.

Разделяй и властвуй

Пусть P — исходный набор точек.

1. Если $|P| < k$, где k — некоторая константа, то построим алгоритмом Грэхэма выпуклую оболочку напрямую.
2. Если $|P| \geq k$, то разобьем P на два непересекающихся набора P_1, P_2. К каждому рекурсивно применим алгоритм.
3. Слить две полученные оболочки.

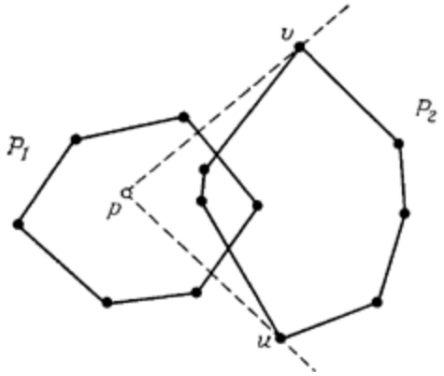
Опишем слияние двух выпуклых многоугольников, то есть получение выпуклой оболочки для объединения двух выпуклых многоугольников P_1, P_2.

1. Выберем точку S так, чтобы она лежала внутри первого многоугольника (например, точка пересечения медиан треугольника на любых трех вершинах из многоугольника P_1).

2. Если S — внутренняя для P_2, то перейдем к шагу 3.

3. Заметим, что если S внутренняя для обоих многоугольников, то вершины обоих многоугольников упорядочены по полярному углу относительно S. Тогда сольем оба отсортированных массива в один отсортированный за линейное время. Перейдем к шагу 5.

4. Если S лежит вне P_2, то построим касательные из S к P_2, то есть найдем такие вершины $U, V \in P_2$, что из S виден весь P_2. Это будет равносильно тому, что пройдем по всему P_2 и найдем вершины U, V так, что угол между $\overrightarrow{OX}, \overrightarrow{SU}$ минимален и угол между $\overrightarrow{OX}, \overrightarrow{SV}$ максимален.



Тогда исходный многоугольник представляет из себя объединение двух цепочек, в которых полярный угол изменяется монотонно. Ту, что ближе к S, можно отбросить, так как она будет лежать внутри объединенной оболочки, а вторую цепочку сливаем с P_1 как в пункте 3.

5. Теперь воспользуемся линейным по времени проходом стека как в алгоритме Грэхема, что позволяет получить выпуклую оболочку на уже отсортированном массиве.

Асимптотика алгоритма: $T(N) = 2T(N/2) + O(N)$, то есть $T(N) = O(NlogN)$.

Выпуклая оболочка в 3D

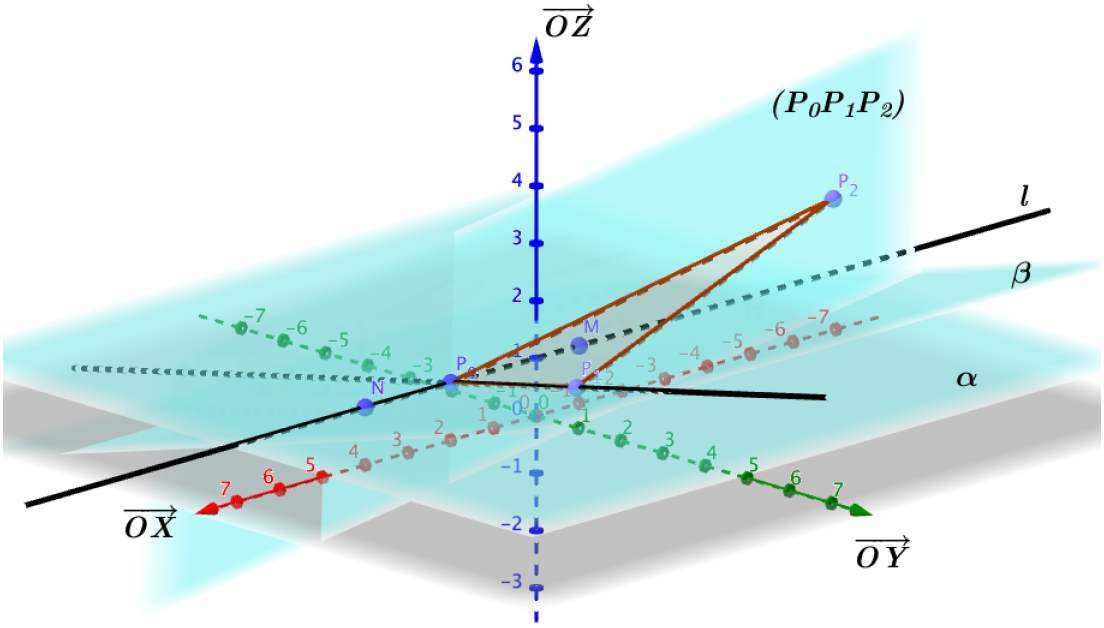
Def. Выпуклая оболочка для набора точек в трехмерном пространстве — многогранник минимального объема, внутри которого (или на его границе) лежат все точки из набора.

Алгоритм Джарвиса (заворачивание подарка)

1. Для начала нам необходимо три точки, которые гарантированно лежат на грани, принадлежащей выпуклой оболочке. Найдем первую точку P_0 таким образом, что ее координата по z меньше всех, при равенстве ищем ту, у которой меньше всех абсцисса, ну и при равенстве ищем наименьшую по ординате.

Далее построим плоскость $\alpha : \alpha \parallel OXY, P_0 \in \alpha$. Выберем две точки $M, N \in \alpha, P_0 \in MN$.Тогда будем искать P_1 таким образом, что $\angle(\alpha, (MP_1N))$ максимален. Заметим, что как бы мы не выбирали l, P_1 всегда будет вершиной выпуклой оболочки. Здесь MN — *опорное ребро*

Тогда пусть плоскость $\beta = (MNP_1)$. Будем искать P_2 так, чтобы $\angle(\beta, (P_0P_1P_2))$ оказался максимальным. Заметим, что общим у этих двух плоскостей будет *опорное ребро* P_0P_1.



На рисунке отмечены все элементы построения, коричневым выделена первая грань.

2. Дальше создадим стек Q из ребер, в него положим первые три ребра: P_0P_1, P_1P_2, P_0P_2. Теперь будем извлекать по одному ребру из стека и искать для него вторую грань выпуклой оболочки, которая содержит его. Если данная грань еще не обрабатывалась (например, поддерживать в сете грани как три точки/ребра), то добавлять в стек два оставшихся ребра новой грани.

3. Если стек пуст, то мы победили, то есть мы замкнули нашу оболочку, подарок завернут.

Нетрудно заметить, что всего итераций $O(N)$, на каждую итерацию требуется $O(N)$ времени. Итоговая сложность алгоритма: $O(N^2)$.

Разделяй и властвуй

1. Отсортируем все точки по абсциссе (при равенстве абсцисс можно рассматривать неравенства на дргуие координаты), тогда сделаем разбиение: $C_1, C_2 : P = C_1 \bigsqcup C_2, |C_1| = |C_2| \pm 1, C_1 = \{K \in P \mid x_K \leq x_0\}$. То есть все точки из C_1 лежат левее некоторой полоскости, параллельной плоскости OYZ.

2. Находим рекурсивно для C_1, C_2 выпуклые оболочки (например, заворачиванием подарка при достаточно малых поднаборах, например, когда их строго меньше восьми) или же, когда размер равен четырем, то мы вроде как победили, так как это тетраэдр, то есть выпуклая фигура.

3. Сливаем оболочки из шага 2.

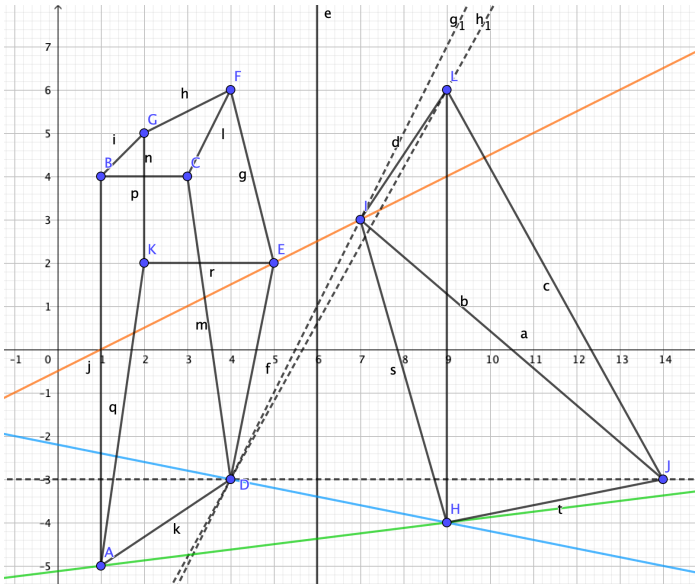
Опишем подробнее слияние из шага 3 (не для слабонервных).

Для того, чтобы вообще все делать красиво и эффективно, воспользуемся такой структурой данных, как реберный список с двойными связями (РСДС) для хранения проекций.

Статья с алгоритмом и РСДС.

1. Спроецируем обе фигуры на плоскость, перепендикулярную разбивающей плоскости из шага 1 прошлого алгоритма (та плоскость параллельна OYZ), тогда будем проектировать на плоскость OXY, то есть делаем апликаты всех точек нулевыми. Проекции будем называть K_1, K_2.

2. *Опорное ребро* для многоугольника T — прямая, проходящая через некоторую точку A, лежащую вне многоугольника и через некоторую вершину T_i так, что весь T лежит в какой-либо полуплоскости относительно ее (далее будем считать, что T должен лежать сверху). Заметим, что для выпуклого многоугольника достаточно перебрать все прямые AT_i и проверить, лежат ли ребра $T_{i-1}T_i,\ T_iT_{i+1}$ в верхней полуплоскости относительно AT_i.



Найдем *опорное ребро* для двух проекций. Для этого возьмем по одной точке, расстояния которых до прямой $x = x_0$ минимальны (то есть минимальна разность их абсциссы и x_0). Проведем ребро Q_1Q_2 через эти точки, где Q_1 из первого многоугольника, а Q_2 — из второго. Далее если это ребро не опорное для какого-либо из многоугольников, то будем сдвигать соответствующую этому многоугольнику вершину Q_i, пока полученное ребро не станет опорным для этого многоугольника. Если при этом полученное ребро оказалось *не опорным* для второго, то двигаем сторую точку по его контуру.

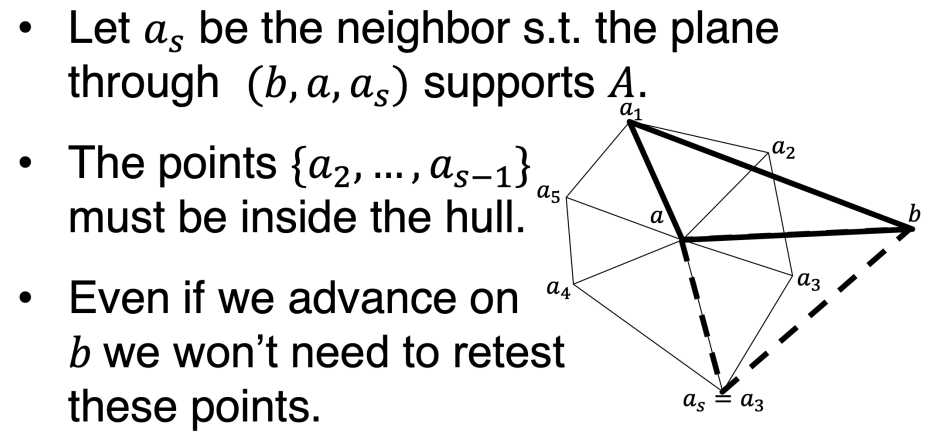
На рисунке ниже приведены шаги построения *опорного* ребра: оранжевое — первоначальное ребро, потом обнаружили, что для первого многоугольника данная прямая не опорная, поэтому двигаем Q_1 по часовой стрелке вдоль контура (в итоге пришли в точку D) — множество

пунктирных прямых, последняя из которых голубая — прямая DH, которая опорная для второго, но не для первого. Теперь необходимо дальше двигать точку по часовой стрелке по первому многоугольнику — пришли к зеленой прямой, опорной для обоих многоугольников.

Тогда при обратном переходе в пространство данное ребро $e = AB$ будет ребром объединенной выпуклой оболочки.

3. Теперь рассмотрим плоскость $\gamma \,||\, OZ, e \in \gamma$. Пусть A_i — вершины, смежные A, а B_i — вершины, смежные B. Тогда семейство плоскостей α_i состоит из плоскостей вида ABA_i, сответственно β_i состоит из плоскостей ABB_i. Обозначим за α_{max} плоскость, образующую максимальный угол с γ, аналогично определим β_{max}. Тогда новой построенной гранью будет та из $\alpha_{max}, \beta_{max}$, что образует наибольший угол с γ.

4. Если хранить смежные точки в порядке обхода по часовой стрелке, то можно заметить, что если ABA_s стала новой гранью, то вершины $A_i, i < s$ лежат внутри выпуклой оболочки, а значит их можно просто удалить. То есть каждая новая грань позволяет удалить некоторое множество вершин, которые гарантированно лежат внутри выпуклой оболочки, что позволяет выполнить слияние за $O(|C_1| + |C_2|)$.



Алгоритм Чана

Будем обозначать за $\hat{P}$ проекцию точки $P(x, y, z)$ на плоскость OXY, но эта проекция будет меняться со временем по следующему закону: $\hat{P}(x, z - ty)$, где t — «время».

Тогда можно заметить, что если строить проекцию фигуры на плоскость в зависимости от времени, то получим нечто вроде «развертки» фигуры на плоскость.

Далее будем рассматривать не исходный набор, а набор из $\hat{P}$.

1. Сортируем набор $\hat{P}$ по абсциссе.

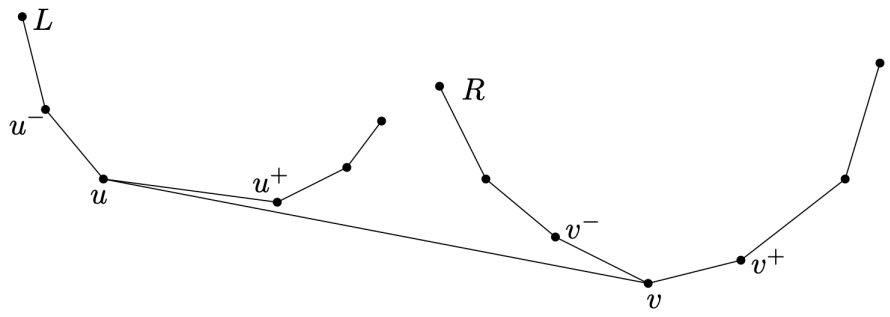
2. Разбиваем на два равных набора $\hat{L}, \hat{R}$. Ищем для них оболочки рекурсивно.

3. Сливаем оболочки из шага 2.

Теперь рассмотрим слияние:

1. Будем создавать «кадры» для обеих оболочек, состоящие из выпуклых многоугольников.

2. Найдем *опорное* ребро для них за линейное время. Для этого сначала возьмем две точки, чьи абсциссы ближе всего к «разделяющей» абсциссе, обзовем их u, v соответственно. Также будем называть u^+, u^- вершины, которые на одну правее и на одну левее по абсциссе соответственно (в отсортированном массиве это ее соседи), аналогично определим v^+, v^-. Далее сдвигаем вершину u на одну влево, проверяем, стали ли повороты u^-uv, uvv^+ оба против часовой стрелке, если нет, то сдвигаем вершину v на одну вправо и снова проверяем условие выше. Тогда мы найдем опорное ребро за линейное время.



3. Запускаем «фильм-развертку» для оболочки. Будем поддерживать вставку и удаление для обеих выпуклых оболочек, то есть если добавляется новая точка в «кадр», то добавлять ее в выпуклую оболочку или нет, зависит от ее местоположения относительно u, v. А именно, мы *всегда* добавляем точку в рассмотрение для плоских выпуклых оболочек, а в пространственную добавляем только если она лежит левее u или правее v. При этом uv остается *опорным*, пока повороты u^-uv, uvv^+ оба против часовой стрелке, а uu^+v, uv^-v — по часовой. Каждое событие вставки/удаления приводит к тому, что одна из вершин u^-, u^+, v^-, v^+ могут измениться. Далее рассмотрим варианты, как менять *опорное* ребро на ходу.

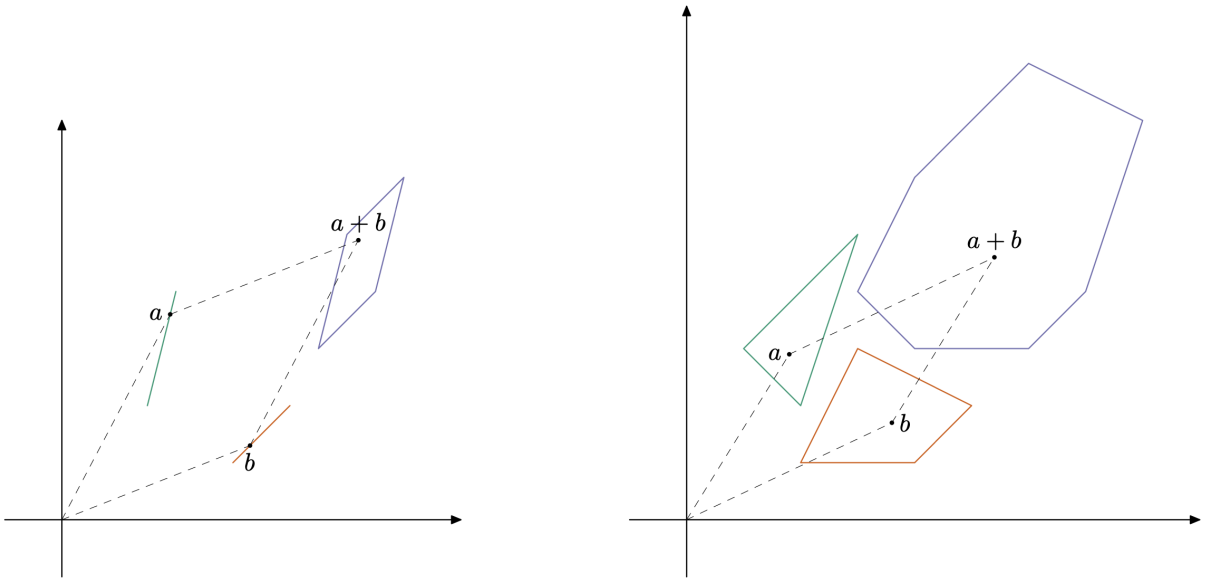
   - Если u^-uv стал повернут по часовой, то u^-v — новое ребро, а из пространственной оболочки удаляем u.
   - Если uu^+v стал повернут против часовой, то u^+v — новое ребро, а в пространственную оболочку добавляем u^+ между u, v.
   - Если uvu^+ стал повернут по часовой, то uv^+ — новое ребро, а из пространственной оболочки удаляем v.
   - Если uu^+v стал повернут против часовой, то uv^- — новое ребро, а в пространственную оболочку добавляем v^- между u, v.

4. Чтобы дойти до следующего обновления, для каждого из шести событий выше (2 варианта добавления новой точки или 4 варианта изменения ребра) рассчитаем время до их появления, выберем минимальное и обработаем. Если так вышло, что новых событий не настанет (а всего их линейное количество), то оболочка получена (точнее получен набор точек, упорядоченный по абсциссе).

Note. Для быстрой работы будем использовать для хранения вершин двусвязный список, также будем хранить указатели на нужные нам вершины.

Сумма Минковского

Пусть имеются два множества точек A, B, тогда суммой Минковского $A \oplus B$ называют множество всех точек, чьи радиус-векторы являются суммами радиус-векторов точек из A и из B.



(a) Сумма Минковского двух отрезков

(b) Сумма двух двух выпуклых многоугольников

Заметим, что сумма Минковского для двух выпуклых многоугольников размеров M, N также выпукла, а число сторон в сумме не превосходит $M + N$.

Построение

Algorithm. Построение суммы Минковского двух выпуклых фигур.

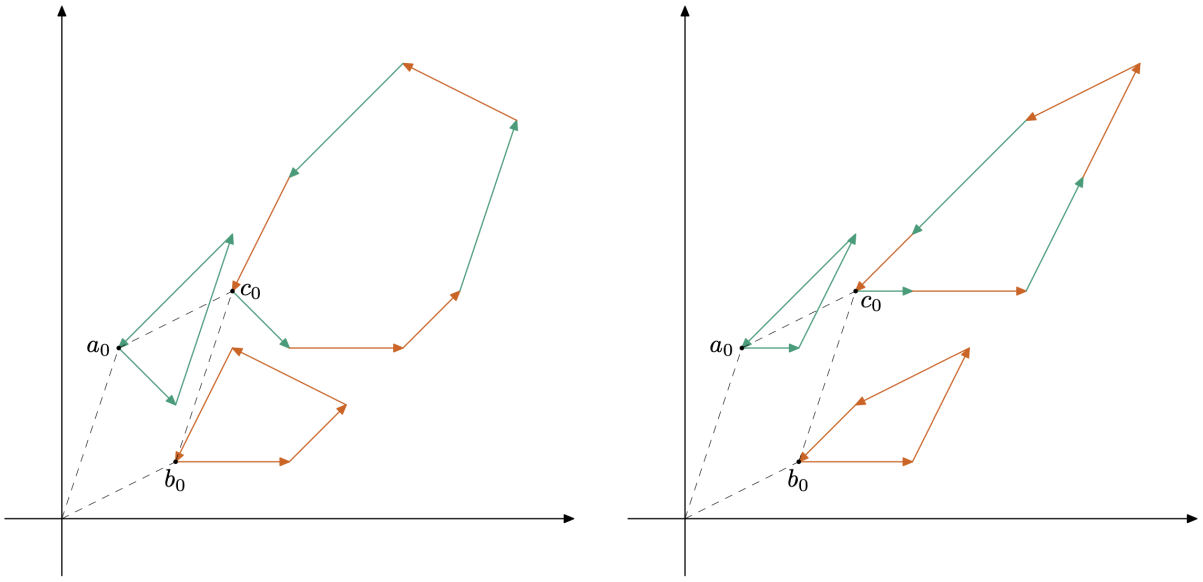
1. Выберем направление, относительно которого будем искать крайние точки, на практике обычно берут направление $v = (-\infty, -1)$. Далее выберем крайние точки A, B в обоих многоугольниках, тогда крайней точкой в сумме Минковского будет вершина с координатами $(x_A + x_B, y_A + y_B)$.

2. Теперь возьмем обход против часовой стрелки, положим вектора в массивы в порядке этого обхода. Заметим, что в таком порядке они уже будут отсортированы по полярному углу в порядке возрастания.

3. Сливаем два отсортированных массива векторов.

4. Будем последовательно откладывать вектора из массива, полученного на шаге 3, от крайней точки, полученной в шаге 1.

Все шаги выполняются за $O(N + M)$, что и является итоговой асимптотикой алгоритма.



(a) Сумма двух выпуклых многоугольников

(b) Пример с колинеарными векторами

Применения

1. Пересекаются ли два выпуклых многоугольника? Пусть B' — многоугольник, полученный отражением B относительно начала координат. Тогда $A \cap B \neq \varnothing \Leftrightarrow (0,0) \in A \oplus B'$.

2. Расстояние между выпуклыми многоугольниками:

$$dist(A, B) = \min_{a \in A, b \in B} |a - b| = \min_{a \in A, b \in B'} |a + b| = \min_{c \in A \oplus B'} |c|.$$

Триангуляция Делоне

Def. Триангуляция набора точек — разбиение выпуклой оболочки данного набора на непересекающиеся треугольники.

Def. Свойство Делоне заключается в том, что для любого треугольника из триангуляции не найдется точки из данного набора, лежащей внутри (но не на границе) окружности, описанной около этого треугольника.

Def. Триангуляция является трангуляцией Делоне, если она удовлетворяет свойству Делоне.

Построение методом 3D оболочки

1. Все точки нам даны на плоскости OXY. Рассмотрим их проекцию на параболоид $x^2 + y^2 = z$, то есть точка вида (x, y) перейдет в $(x, y, x^2 + y^2)$. Сложность $O(N)$
2. Самый простой шаг: построим $3D$-выпуклую оболочку на полученном наборе точек. Сложность $O(NlogN)$
3. Заметим, что проекции полученных граней на плоскость OXY дадут необходимое разбиение на треугольники. Сложность $O(N)$.

Итоговая сложность алгоритма: $O(NlogN)$.

Note. Стоит заметить, что трангуляция *всегда* существует; при этом она *единственна*, если никакие четыре точки не лежат ни на одной окружности.

Факт выше читателю предлагается проверить самостоятельно, ну и подсказка: если точки лежали на одной окружности, то они будут лежать на одной плоскости на параболоиде.

Построение с помощью сканирующей прямой от Павла Косицына

Евклидово минимальное остовное дерево

Евклидово минимальное остовное дерево (EMST) — это минимальное остовное дерево набора из n точек на плоскости, где вес ребра между любой парой точек является евклидовым расстоянием между двумя точками. Простыми терминами, оно связывает набор точек с помощью отрезков так, что общая длина всех отрезков минимальна и любая точка может быть достигнута из другой точки по этим отрезкам.

Th. В триангуляции Делоне содержатся все ребра EMST.

Если поверить прошлой теореме, то для каждого ребра из триангуляции найдем его вес — длину и запустим алгоритм поиска обычного MST. Итоговая сложность: $O(NlogN)$.

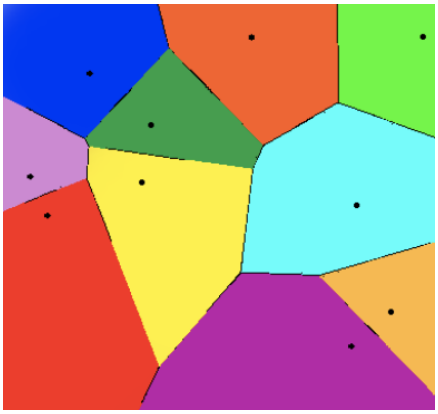
Диаграмма Вороного

Для набора точек назовем локусом данной точки часть плоскости, точки из которой ближе к данной точке, чем ко всем остальным.

Пусть A_i — набор точек, тогда $L(A_k) = \{P \in \mathbb{R}^2 \mid \rho(A_k, P) \leq \rho(A_i, P)\}$ — локус точки A_k.

Наивный алгоритм

Для начала построим для данной точки B ее локус. Для этого построим отрезки $b_i = A_iB$, теперь построим прямые h_i, являющиеся серединными перпендикулярами к отрезкам b_i. Тогда локусом будет пересечение *всех* полуплоскостей, порожденных семейством h_i, при этом выбираем из каждой ту, где лежит точка B. Тогда построим локусы для всех точек, тогда их объединение и будет искомой диаграммой. Локусы также называют ячейками диаграммы.



Так как полуплоскости можно пересечь за $O(NlogN)$, итоговая асимптотика алгоритма $O(N^2logN)$.

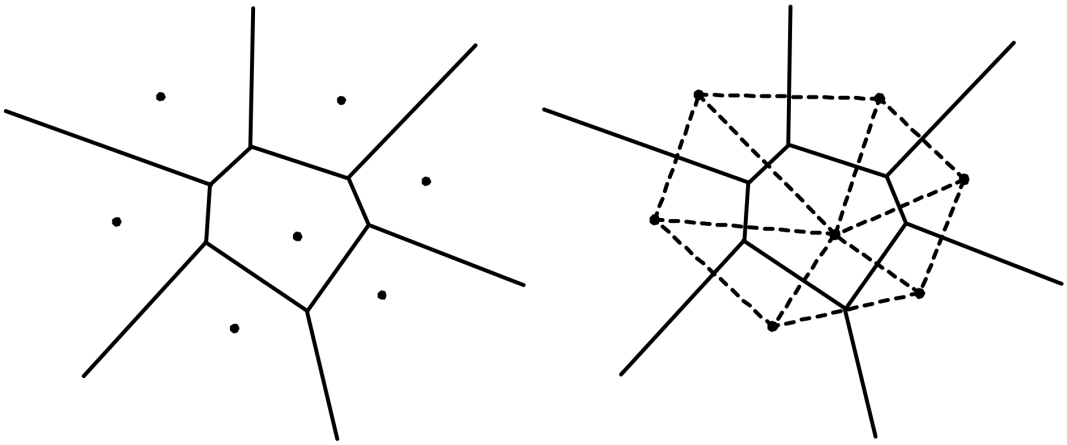
Метод трангуляции Делоне

Триангуляцию Делоне можно получить из диаграммы Вороного следующим образом: соединим ребром две вершины диаграммы, если у их ячеек есть смежное ребро.

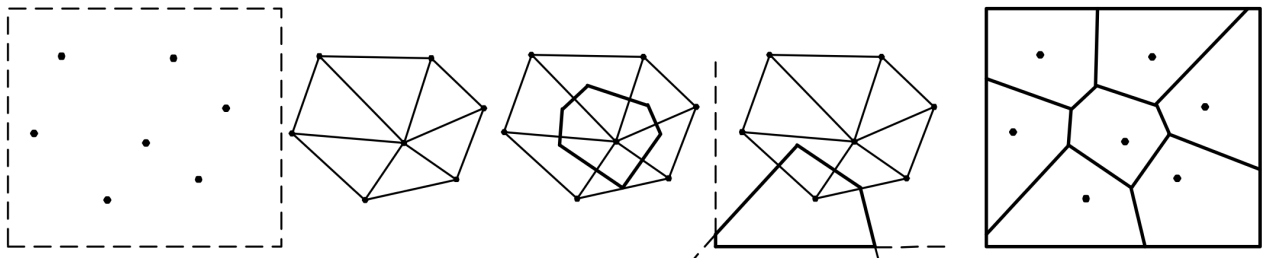
На рисунке слева диаграмма Вороного, а справа — триангуляция Делоне.

Algorithm. Построение диаграммы Вороного по триангуляции Делоне

1. Строим триангуляцию Делоне (любимая $3D$ оболочка) за $O(NlogN)$
2. Для каждого треугольника триангуляции найдем центр описанной окружности за $O(N)$.
3. Для каждого треугольника вычислим центр ячейки Вороного. Для этого обходим вокруг текущего узла по смежным треугольникам и собираем центры их описанных окружностей. Если



треугольник находится не на границе триангуляции, то таким образом мы соберем координаты соответствующего многоугольника Вороного этого узла. Если этот узел находится на границе, значит, многоугольник Вороного является бесконечной фигурой, поэтому необходимо в этом случае выполнить отсечение двух его бесконечных сторон.



Данный шаг просматривает каждое ребро триангуляции не более двух раз, а значит работает за $O(E) = O(N)$.

Итоговая сложность алгоритма: $O(NlogN)$.

Алгоритм Форчуна

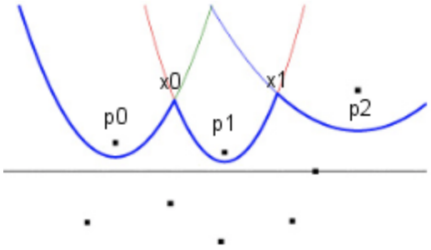
Алгоритм использует заметающую прямую, которая (для определенности) будет горизонтальной и идти сверху вниз.

1. Отсортируем точки по ординате за $O(NlogN)$.
2. Далее у нас будут события двух типов: событие точки и событие круга.

   Событие точки — событие, когда наша сканирующая прямая приходит к очередной точке. При этом у нас появляется новая парабола, чьей фокус константен (это добавленная точка), а ее директрисой является наша сканирующая прямая, то есть парабола расширяется по мере удаления прямой.

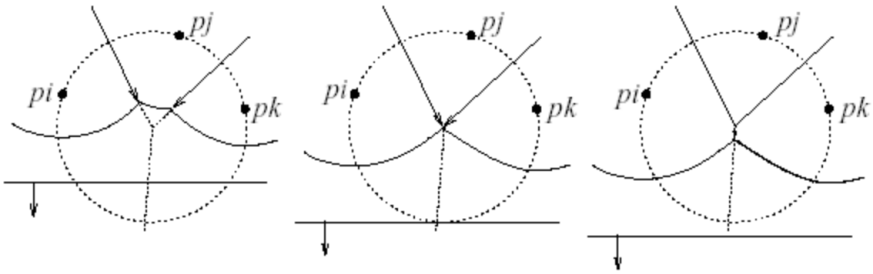
Событие круга — событие, когда две параболы, полученные из событий точки пересекаются. Множество точек внутри параболы будет задавать очередную ячейку Вороного. Далее прямая будет идти по мере обработки событий.

По мере продвижения прямой будет получен фронт из парабол, который может накладываться друг на друга. В момент наложения и будет происходить событие круга.



3. В момент события точки будем создавать две крайние точки — пересечение новой параболы с уже существующим фронтом.

4. В момент события круга происходит пересечение *трех* парабол, при этом крайние «поглощают» среднюю параболу, далее общая точка крайних парабол движется по прямой, при этом точка «схлопывания» будет очередной вершиной диаграммы (поверьте в это, и будет вам счастье).



Algorithm. Сам Форчун.

1. Создаем очередь событий с приоритетами. Получается она по мере продвижения сканирующей прямой, при этом событиям точки соответствуют добавления новой точки (проход сканирующей прямой по ним, а событиям окружности соответствуют ординаты нижних точек окружности, построенной на трех соседних вершинах.

   Понять, какие две вершины являются соседними для только что добавленной вершины можно, найдя параболы, в которые придут ветви параболы, соответствующие только что добавленной точке.

   Например, на рисунке с точками p_0, p_1, p_2 мы добавляем новую точку, и соответствующая ей парабола попадет в параболы для точек p_1, p_2. То есть для новой точки будут найдены ближайшие: p_1, p_2! Теперь по этим трем точкам построим окружность, нижняя точка которой и породит событие круга!!

2. Пока очередь не пуста, обрабатываем по порядку события в зависимости от их типов.

3. Как только события закончились, надо обработать границы.

ПОБЕДА!!!!

РК 4.

Длинная арифметика

Алгоритм Карацубы

Пусть $A = \overline{a_{n-1}, \ldots, a_1, a_0}$ и $B = \overline{b_{n-1}, \ldots, b_1, b_0}$ в системе счисления с основанием $base$. Заметим, что тогда каждое из них можно разбить на два слагаемых, а именно ($m = \frac{n}{2}$):

$$A = \overline{a_{n-1}, \ldots, a_{m+1}, a_m} * base^m + \overline{a_{m-1}, \ldots, a_1, a_0},\ B = \overline{b_{n-1}, \ldots, b_{m+1}, b_m} * base^m + \overline{b_{m-1}, \ldots, b_1, b_0}.$$

Далее будем считать, что $A = A_0 + A_1 * base^m,\ B = B_0 + B_1 * base^m$. Тогда распишем умножение:

$$A * B = A_0 * B_0 + (A_1 * B_0 + A_0 * B_1) * base + A_1 * B_1 * base^{2m}.$$

Теперь заметим, что:

$$(A_1 + B_0) * (A_0 + B_1) = (A_0 + A_1) * (B_0 + B_1) - A_0 * B_0 - A_1 * B_1.$$

Таким образом, итоговая формула для умножения:

$$A * B = A_0 * B_0 + ((A_0 + A_1) * (B_0 + B_1) - A_0 * B_0 - A_1 * B_1) * base + A_1 * B_1 * base^{2m}.$$

Теперь нужно делать **три** умножения вместо четырех (как в умножении столбиком). Заметим, что возведение $base$ в квадрат — просто сдвиг числа с дописыванием нулей что требует линейной асимптотики. Итоговое время работы алгоритма: $O(n^{log_2 3})$

То есть, рекурсивно вызывая такое перемножение для длинных чисел, полчим значительный прирост в быстродействии. При этом стоит отметить, что на числах длины порядка несколька десятков метод столбика работает эффективнее, так что алгоритм можно представить достаточно просто:

1. Для каждого из трех перемножений вызовем метод Карацубы, пока длина чисел велика.

2. Если длина достаточно мала, то перемножаем столбиком.

Деление

Делим классическим методом, то есть столбиком, при этом очередную цифру ответа ищем бинарным поиском. Если использовалось умножение Карацубы, то утверждается, что деление работает за $O(n^2)$, где n — длина числа.

Преобразования Фурье

Дискретное преобразование Фурье

Рассмотрим многочлен степени n: $A(x) = a_0x^0 + a_1x^1 + \ldots, a_{n-1}x^{n-1}$ (считаем далее, что n — степень двойки, иначе можно дополнить нулевыми коэффициентами).

Заметим, что у уравнения $u^n = 1$ существует n корней вида $u_{n,k} = e^{\frac{2i\pi k}{n}}$. Тогда *дисретным* преобразованием Фурье от многочлена называют следующий набор:
$DFT(a_0, \ldots, a_{n-1}) = (A(u_{n,0}), A(u_{n,1}), \ldots, A(u_{n,n-1})) = (A(u^0), A(u^1), \ldots A(u^{n-1}))$.

Обратное DFT — процедура получения набора исходных коэффициентов по набору коэффициентов Фурье.

Быстрое преобразование Фурье

Def. Быстрое преобразование Фурье (FFT) — процедура вычисления DFT для многочлена степени n за $O(nlogn)$ времени.

Algorithm. Вычисление FFT

Пусть $A_0(x) = a_0x^0 + a_2x^1 + \ldots + a_{n-2}x^{\frac{n}{2}-1}$, $A_1(x) = a_1x^0 + a_3x^1 + \ldots + a_{n-1}x^{\frac{n}{2}-1}$, тогда получаем: $A(x) = A_0(x^2) + x * A_1(x^2)$.

1. По формуле выше разобьем на набор на два поднабора длины вдвое меньшей, вычислим для них рекурсивно DFT.
2. Сольем два поднабора в один, вычислим для него DFT.

Вооружившись знаниями матана, получаем, что если мы обозначим наборы $DFT(A_0)$ и $DFT(A_1)$ за $y_0^0, \ldots, y_{\frac{n}{2}-1}^0$ и за $y_0^1, \ldots, y_{\frac{n}{2}-1}^1$ соответственно, то получим результат:

$$y_k = y_k^0 + u_n^k * y_k^1,\ y_{k+\frac{n}{2}} = y_k^0 - u_n^k * y_k^1,\ \text{где } n \in (0, \frac{n}{2} - 1).$$

Таким образом, типичный *Разделяй и властвуй* за $O(nlogn)$ времени.

Обратное преобразование Фурье

Прямое преобразование Фурье можно записать в матричном виде, тогда вычисление обратного преобразования заключается в решении однородной СЛУ, что сводится к расчету обратной матрицы коэффициентов, состоящей из элементов вида u_n^{ij}.

Тогда можно получить формулу вида:

$$a_k = \sum_{j=0}^{n-1} y_j u_n^{-kj}.$$

То есть мы теперь умеем считать прямое и обратное DFT за $O(nlogn)$ времени.

Заметим, что с помощью преобразований Фурье можно перемножать длинные числа, так как $A = \overline{a_{n-1}, \ldots, a_1, a_0} = a_{n-1} * base^0 + \ldots + a_1 * base^{n-2} + a_0 * base^{n-1}$, то есть длинное число — результат подстановки в многочлен системы основания счисления.

$A * B = DFT^{-1}(DFT(A) * DFT(B))$, где $DFT(A) * DFT(B) = (y_0^A * y_0^B, \ldots, y_{n-1}^A * y_{n-1}^B)$.

Игры

Игра Ним

Правила игры просты: есть p кучек с $N_1, \dots N_p$ камнями, при этом за ход игроку разрешается взять из любой кучки (только одной) любое положительное количество камней. При этом проигрывает тот, кто не может взять ход.

Теорема Шпрага-Гранди

Пусть $G(x)$ — множество позиций, в которые можно перейти из позиции x.

Тогда определим функцию Гранди как $F(x) = min\{n \geq 0 | \ \forall y \in G(x): \ n \neq F(y)\}$. Для конечных позиций $F(x) = 0$.

Th. Шпрага-Гранди

Обозначим за сумму игр $P_1, \dots, P_k$ объединение игр на нескольких полях, где за ход можно поменять состояние только одного поля. Тогда верно равенство:

$F(P_1, \dots, P_k) = F(P_1) \oplus \dots \oplus F(P_k)$, где $\oplus$ обозначает побитовое «исключающее или», то есть «XOR»

Th. Эквивалентность игр

Любая игра, чей процесс не зависит от того, кто ходит на данный момент (то есть зависит только от позиции на поле) эквивалентна игре Ним.

Th. Результат игры

Любая позиция, чья функция Гранди равна нулю, проигрышная, то есть если игрок попал или начал с позиции с нулевой функцией Гранди, то он заведомо проиграет.

Игры на графе

Def. Игра называется справедливой, если каждый игрок из одной позиции может делать такие же ходы, как и противник. Пример несправедливой — шахматы, справедливой — игра Ним.

Def. Игра детерминированная, если в правилах, возможности или очерёдности ходов, определении момента завершения игры или результата не участвует элемент случайности. Пример детерминированной — шахматы, недетерминированной игры — нарды *(шутка про армян)*.

Для любой детерминированной игры можно построить ориентированный ациклический граф состояний, если известна вся информация о ее состояниях и переходах (игры с полной информацией).

Def. Игра называется нормальной, если листья ее графа проигрышные.

Минимакс

Пусть для игры построен ее граф. Данный граф будет деревом с (возможно) повторющимися состояниями, тогда определим оценку позиции следущим образом:

Пусть x — текущая позиция, а $G(x)$ — множество позиций, куда можно перейти, тогда

$$M(x) = \begin{cases} max\{M(y) : y \in G(x)\}, & \text{если ходит первый игрок и } G(x) \neq \varnothing \\ min\{M(y) : y \in G(x)\}, & \text{если ходит второй игрок и } G(x) \neq \varnothing \\ 1, & \text{если в вершине выигрывает первый и } G(x) = \varnothing \\ -1, & \text{если в вершине выигрывает второй и } G(x) = \varnothing \end{cases}$$

Note. Для определения функции оценки необязательно графу быть деревом, что позволяет объединять вершины с одинаковым состоянием в одну вершину.

Узнав оценку корня, можем узнать, кто выиграет и, соответственно, выигрышную стратегию.

В данном алгоритме можно определить еще надбавку к оценке, оценивающую состояние на поле в целом, например в шахматах можно ее ввести за позиционное преимущество. Это позволяет при применении *альфа-бета отсечении* строить дерево игры заметно быстрее.

Альфа-бета отсечение

Этот алгоритм является оптимизацией (или эвристикой) Минимакса. Его суть заключается в исключении из рассмотрения поддеревьев на основе двух параметров: $\alpha = max(\alpha, f(V_i))$ и $\beta = min(\beta, f(V_i))$

Рассмотрим его на примере дерева некоторой игры:

- V_i — узлы дерева
- $V_i[\alpha, \beta]$ — узлы дерева решений с указанными параметрами
- MIN, MAX — уровни, где играет игрок, выбирающий ход на основе значений **на уровне ниже** по принципу на его уровне (MAX или MIN соответственно)
- C_i — i-е отсечение

Тогда при расчете функции оценки для каждой верины будут также обновляться параметры α, β для рассматриваемой вершины, и если $f(V_i) < \alpha$, то нет даже смысла идти в поддерево вершины V_i, так как мы знаем, что есть путь лучше. Аналогичная ситуация, если $f(V_i) > \beta$.

Теперь рассмотрим отсечение C_1: после обработки правого поддерева вершины V_4 возникла ситуация, что $\alpha > \beta$, что говорит о том, что произошел конфликт в оценочной функции и этого поддерева больше не существует в рассмотрении. Остальные отсечения рассматриваются абсолотно аналогично.

Таким образом, после того, как было построено дерево игры, запускают это алгоритм, позволяющий сильно уменьшить дерево в размерах и, как следствие, поиски стратегий в нем.