SPb HSE, 2 курс, осень 2022/23
Конспект лекций по алгоритмам

Собрано 9 декабря 2022 г. в 11:32

Содержание

Лекция #1: Суффиксный массив

7 ноября 2022

Def 1.0.1. *Суффиксный массив s – отсортированный массив суффиксов s.*

Суффиксы сортируем в лексикографическом порядке. Каждый суффикс однозначно задается позицией начала в $s \Rightarrow$ на выходе мы хотим получить перестановку чисел от 0 до $n-1$.

• **Тривиальное решение:** `std::sort` отработает за $\mathcal{O}(n \log n)$ операций '<' $\Rightarrow$ за $\mathcal{O}(n^2 \log n)$.

1.1. Построение за $\mathcal{O}(n \log^2 n)$ хешами

Мы уже умеем сравнивать хешами строки на равенство, научимся сравнивать их на "`>/<`".
Бинпоиском за $\mathcal{O}(\log(min(|s|, |t|)))$ проверок на равенство найдём $x = lcp(s, t)$.
Теперь $less(s, t) = (s[x] < t[x])$. Кстати, в `C/C++` после строки всегда идёт символ с кодом 0.

Получили оператор меньше, работающий за $\mathcal{O}(\log n)$ и требующий $\mathcal{O}(n)$ предподсчёта.
Итого: суффмассив за $\mathcal{O}(n + (n \log n) \cdot \log n) = \mathcal{O}(n \log^2 n)$.

При написании сортировки нам нужно теперь минимизировать в первую очередь именно число сравнений $\Rightarrow$ с точки зрения `C++::STL` быстрее будет работать `stable_sort` (MergeSort внутри).

Замечание 1.1.1. Заодно научились за $\mathcal{O}(\log n)$ сравнивать на больше/меньше любые подстроки.

1.2. Применение суффиксного массива: поиск строки в тексте

Задача: дана строка t, приходят строки-запросы s_i: "является ли s_i подстрокой t".

Предподсчёт: построим суффиксный массив p строки t.
В суффиксом массиве сначала лежат все суффиксы $< s_i$, затем $\geqslant s_i \Rightarrow$ бинпоиском можно найти $\min k\colon\ t[p_k:] \geqslant s_i$. Осталось заметить, что (s_i – префикс $t[p_k:]$) $\Leftrightarrow$ (s_i – подстрока t).

Внутри бинпоиска можно сравнивать строки за линию, получим время $\mathcal{O}(|s_i| \log |t|)$ на запрос.
Можно за $\mathcal{O}(\log |t|)$ с помощью хешей, для этого нужно один раз предподсчитать хеши для t, а при ответе на запрос насчитать хеши s_i. Получили время $\mathcal{O}(|s_i| + \log |t| \cdot \log |s_i|)$ на запрос.

В разд. 1.5 мы улучшим время обработки запроса до $\mathcal{O}(|s_i| + \log |t|)$.

1.3. Построение за $\mathcal{O}(n^2)$ и $\mathcal{O}(n \log n)$ цифровой сортировкой

Заменим строку `s` на строку `s#`, где `#` – символ, лексикографически меньший всех в `s`.
Будем сортировать циклические сдвиги `s#`, порядок совпадёт с порядком суффиксом.
Длину `s#` обозначим n.

Решение за $\mathcal{O}(n^2)$: цифровая сортировка.
Сперва подсчётом по последнему символу, затем по предпоследнему и т.д.
Всего n фаз сортировок подсчётом. В предположении $|\Sigma| \leqslant n$ получаем время $\mathcal{O}(n^2)$.
Суффмассив, как и раньше задаётся перестановкой начал... теперь циклических сдвигов.

Решение за $\mathcal{O}(n \log n)$: цифровая сортировка с удвоением длины.
Пусть у нас уже отсортированы все подстроки длины k циклической строки `s#`.
Научимся за $\mathcal{O}(n)$ переходить к подстрокам длины $2k$.

Давайте требовать не только отсортированности но и знания “равны ли соседние в отсортированном порядке”. Тогда линейным проходом можно для каждого i насчитать тип (цвет) циклического сдвига $c[i]$: $(0 \leqslant c[i] < n) \land (\texttt{s[i:i+k)} < \texttt{s[j:j+k)} \Leftrightarrow c[i] \leqslant c[j])$.

Любая подстрока длины $2k$ состоит из двух половин длины $k \Rightarrow$
переход $k \to 2k$ – цифровая сортировка пар $\langle c[i], c[i{+}k] \rangle$.

Прекратим удвоение k, когда $k \geqslant n$. Порядки подстрок длины k и n совпадут.

Замечание 1.3.1. В обоих решениях в случае $|\Sigma| > n$ нужно первым шагом отсортировать и перенумеровать символы строки. Это можно сделать за $\mathcal{O}(n \log n)$ или за $\mathcal{O}(n + |\Sigma|)$ подсчётом.

Реализация решения за $\mathcal{O}(n \log n)$.
`p[i]` – перестановка, задающая порядок подстрок длины `s[i:i+k)` циклической строки `s#`.
`c[i]` – тип подстроки `s[i:i+k)`.
За базу возьмём $k = 1$

```
bool sless( int i, int j ) { return s[i] < s[j]; }
sort(p, p + n, sless);
cc = 0; // текущий тип подстроки
for (i = 0; i < n; i++) // тот самый линейный проход, насчитываем типы строк длины 1
  cc += (i && s[p[i]] != s[p[i-1]]), c[p[i]] = cc;
```

Переход: (у нас уже отсортированы строки длины k) $\Rightarrow$ (уже отсортированы строки длины $2k$ по второй половине) $\Rightarrow$ (осталось сделать сортировку подсчётом по первой половине).

```
// pos - массив из n нулей
for (i = 0; i < n; i++)
  pos[c[i] + 1]++; // обойдёмся без лишнего массива cnt
for (i = 1; i < n; i++)
  pos[i] += pos[i - 1];
for (i = 0; i < n; i++): // p[i] - позиция начала второй половины
  int j = (p[i] - k) mod n; // j - позиция начала первой половины
  p2[pos[c[j]]++] = j; // поставили подстроку s[j,j+2k) на правильное место в p2
cc = 0; // текущий тип подстроки
for (i = 0; i < n; i++) // линейным проходом насчитываем типы строк длины 2k
  cc += (i && pair_of_c(p2[i]) != pair_of_c(p2[i-1])), c2[p2[i]] = cc;
c2.swap(c), p2.swap(p); // не забудем перейти к новой паре (p,c)
```

Здесь `pair_of_c(i)` – пара $\langle$`c[i]`, `c[(i + k) mod n]`$\rangle$ (мы сортировали как раз эти пары!).

Замечание 1.3.2. При написании суффмассива в контесте рекомендуется, прочтя конспект, написать код самостоятельно, без подглядывания в конспект.

1.4. LCP за $\mathcal{O}(n)$: алгоритм Касаи

Алгоритм Касаи считает LCP соседних суффиксов в суффиксном массиве. Обозначения:

- $p[i]$ – элемент суффмассива,
- $p^{-1}[i]$ – позиция суффикса `s[i:]` в суффмассиве,
- $next_i = p[p^{-1}[i] + 1]$, $lcp_i = \text{LCP}(i, next_i)$. Наша задача – насчитать массив lcp_i.

Утверждение 1.4.1. Если у i-го и j-го по порядку суффикса в суффмассиве совпадают первые k символов, то на всём отрезке $[i, j]$ суффмассива совпадают первые k символов.

Lm 1.4.2. Основная идея алгоритма Касаи: $lcp_i > 0 \Rightarrow lcp_{i+1} \geqslant lcp_i - 1$.

Доказательство. Отрежем у `s[i:]` и `s[next`$_i$`:]` по первому символу.
Получили суффиксы `s[i+1:]` и какой-нибудь `r`.
(`s[i:]` $\neq$ `s[next`$_i$`:]`) $\wedge$ (первый символ у них совпадал) $\Rightarrow$
(`r` в суффмассиве идёт после `s[i+1:]`) $\wedge$ (у них совпадает первых $lcp_i{-}1$ символов) $\overset{1.4.1}{\Rightarrow}$
у `s[i+1:]` и `s[next`$_{i+1}$`]` совпадает хотя бы $lcp_i{-}1$ символ $\Rightarrow lcp_{i+1} \geqslant lcp_i - 1$. ■

Собственно алгоритм заключается в переборе $i \nearrow$ и подсчёте lcp_i начиная с $\max(0, lcp_{i+1}{-}1)$.

Задача: уметь выдавать за $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$ LCP любых двух суффиксов строки s.

Решение: используем Касаи для соседних, а для подсчёта LCP любых других считаем RMQ. RMQ мы решили в прошлом семестре. Например, Фарах-Колтоном-Бендером за $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$.

1.5. Быстрый поиск строки в тексте

Представим себе простой бинпоиск за $\mathcal{O}(|s|\log(|text|))$. Будем стараться максимально переиспользовать информацию, полученную из уже сделанных сравнений.

Для краткости $\forall k$ обозначим k-й суффикс (`text[p`$_k$`:]`) как просто k.
Инвариант: бинпоиск в состоянии $[l, r]$ уже знает $lcp(s, l)$ и $lcp(s, r)$.
Сейчас мы хотим найти $lcp(s, m)$ и перейти к $[l, m]$ или $[m, r]$.
Заметим, $lcp(s, m) \geq max\{min\{lcp(s, l), lcp(l, m)\}, min\{lcp(s, r), lcp(r, m)\}\} = x$.
Мы умеем искать $lcp(l, m)$ и $lcp(r, m)$ за $\mathcal{O}(1) \Rightarrow$ `for(` $lcp(s, m) = x$`; можем;` $lcp(s, m)$`++)`.

Кстати, $lcp(l, m)$ и $lcp(r, m)$ не обязательно считать Фарах-Колтоном-Бендером, так как, аргументы lcp – не произвольный отрезок, а вершина дерева отрезков (состояние бинпоиска). Предподсчитаем lcp для всех $\leqslant 2|text|$ вершин и по ходу бинпоиска будем спускаться по Д.О.

Теорема 1.5.1. Суммарное число увеличений на один $lcp(s, ?)$ не более $|x|$

Доказательство. Сейчас бинпоиск в состоянии l_i, m_i, r_i. Следующее состояние: l_{i+1}, r_{i+1}.
Предположим, $lcp(s, l_i) \geqslant lcp(s, r_i)$. Будем следить за величиной $z_i = \max\{lcp(s, l_i), lcp(s, r_i)\}$.
Пусть $lcp(s, m_i) < z_i \Rightarrow lcp(s, m) = x \wedge l_{i+1} = l_i \Rightarrow z_{i+1} = z_i$. Иначе $x = z_i \wedge z_{i+1} = lcp(s, m_i)$. ■

1.6. (*) Построение за $\mathcal{O}(n)$: алгоритм Каркайнена-Сандерса

На вход получаем строку s длины n, при этом $0 \leqslant s_i \leqslant \frac{3}{2}n$.
Выход – суффиксный массив. Сортируем именно суффиксы, а не циклические сдвиги.

Допишем к строке 3 нулевых символа. Теперь сделаем новый алфавит: $w_i = (s_i, s_{i+1}, s_{i+2})$.
Отсортируем w_i цифровой сортировкой за $\mathcal{O}(n)$, перенумеруем их от 0 до $n{-}1$.
Запишем все суффиксы строки s над новым алфавитом:

$t_0 = w_0 w_3 w_6 \dots$
$t_1 = w_1 w_4 w_7 \dots$
$t_2 = w_2 w_5 w_8 \dots$
$\dots$
$t_{n-1} = w_{n-1}$

Про суффиксы t_{3k+i}, где $i \in \{0, 1, 2\}$, будем говорить "суффикс i-типа".

Запустимся рекурсивно от строки $t_0 t_1$. Длина $t_0 t_1$ не более $2\lceil\frac{n}{3}\rceil$.

Теперь мы умеем сравнивать между собой все суффиксы 0-типа и 1-типа.

Суффикс 2-типа = один символ + суффикс 0-типа $\Rightarrow$
их можно рассматривать как пары и отсортировать за $\mathcal{O}(n)$ цифровой сортировкой.

Осталось сделать merge двух суффиксных массивов.
Операция merge работает за линию, если есть "`operator <`", работающий за $\mathcal{O}(1)$.
Нужно научиться сравнивать суффиксы 2-типа с остальными за $\mathcal{O}(1)$.

$\forall i, j\colon t_{3i+2} = s_{3i+2}t_{3i+3},\ t_{3j} = s_{3j}t_{3j+1} \Rightarrow$ чтобы сравнить суффиксы 2-типа и 0-типа, достаточно уметь сравнивать суффиксы 0-типа и 1-типа. Умеем.

$\forall i, j\colon t_{3i+2} = s_{3i+2}t_{3i+3},\ t_{3j+1} = s_{3j+1}t_{3j+2} \Rightarrow$ чтобы сравнить суффиксы 2-типа и 1-типа, достаточно уметь сравнивать суффиксы 0-типа и 2-типа. Только что научились.

• **Псевдокод.**

Пусть у нас уже есть `radixSort(a)`, возращающий перестановку.

```
def getIndex(a): # новая нумерация, O(|a| + max_i a[i])
  p = radixSort(a)
  cc = 0
  ind = [0] * n
  for i in range(n):
    cc += (i > 0 and a[p[i]] != a[p[i-1]])
    ind[p[i]] = cc
  return ind

def sufArray(s): # 0 ⩽ s_i ⩽ 3/2 n
  n = len(s)
  if n < 3: return slowSlowSort(s)
  s += [0, 0, 0]
  w = getIndex( [(s[i], s[i+1], s[i+2]) for i in range(n)] )
  index01 = range(0, n, 3) + range(1, n, 3) # с шагом 3
  p01 = sufArray( [w[i] for i in index01] )
  pos = [0] * n
  for i in range(len(p01)): pos[index01[p01[i]]] = i # позиция 01-суффикса в p01
  index2 = range(2, n, 3)
  p2 = getIndex( [(w[i], pos[i+1]) for i in index2] )
  def less(i, j): # i mod 3 = 0/1, j mod 3 = 2
    if i mod 3 == 1: return (s[i],pos[i+1]) < (s[j],pos[j+1])
    else: return (s[i],s[i+1],pos[i+2]) < (s[j],s[j+1],pos[j+2])
  return merge(p01 ∘ index01, p2 ∘ index2, less)
  # ∘ - композиция: index01[p01[i]], ...
```

Для $n \geqslant 3$ рекурсивный вызов делается от строго меньшей строки:
$3 \to 1{+}1,\ 4 \to 2{+}1,\ 5 \to 2{+}2,\ \dots$.

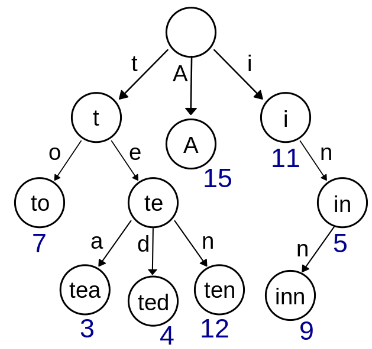
Неравенством $s_i \leqslant \frac{3}{2}n$ мы в явном виде в коде нигде не пользуемся.
Оно нужно, чтобы гарантировать, что `radixSort` работает за $\mathcal{O}(n)$.

Есть и другие идеи построения суффиксного массива за линию.
Из более быстрых и современных стоит отметить [Nong, Zhang & Chan (2009)].
Реализации более быстрого SA-IS: [google-implementation], [SK-implementation].

Лекция #1: Бор

7 ноября 2022

1.7. Собственно бор



Бор – корневое дерево. Рёбра направлены от корня и подписаны буквами. Некоторые вершины бора подписаны, как конечные.

Базовое применение бора – хранение словаря `map<string, T>`.

Пример из wiki бора, содержащего словарь $\{\text{A}:15,\ \text{to}:7,\ \text{tea}:3,\ \text{ted}:4,\ \text{ten}:12,\ \text{i}:11,\ \text{in}:5,\ \text{inn}:9\}$.

Для строки `s` операции `add(s)`, `delete(s)`, `getValue(s)` работают, как спуск вниз от корня.

Самый простой способ хранить бор: `vector<Vertex> t;`, где `struct Vertex { int id[|Σ|]; };` Сейчас рёбра из вершины `t` хранятся в массиве `t.id[]`. Есть другие структуры данных:

Способ хранения	**Время спуска по строке**	**Память на ребро**
`array`	$\mathcal{O}(\lvert s\rvert)$	$\mathcal{O}(\lvert\Sigma\rvert)$
`list`	$\mathcal{O}(\lvert s\rvert\cdot\lvert\Sigma\rvert)$	$\mathcal{O}(1)$
`map (TreeMap)`	$\mathcal{O}(\lvert s\rvert\cdot\log\lvert\Sigma\rvert)$	$\mathcal{O}(1)$
`HashMap`	$\mathcal{O}(\lvert s\rvert)$ с большой const	$\mathcal{O}(1)$
`SplayMap`	$\mathcal{O}(\lvert s\rvert+\log S$	$\mathcal{O}(1)$

Иногда для краткости мы будем хранить бор массивом `int next[N][|Σ|];`. `next[v][c] == 0` $\Leftrightarrow$ ребра нет.

1.8. Сортировка строк

Если мы храним рёбра в структуре, способной перебирать рёбра в лексикографическом порядке (не хеш-таблица, не список), можно легко отсортировать массив строк:
(1) добавить их все в бор, (2) обойти бор слева направо.
Для `SplayMap` и n и строк суммарной длины S, получаем время $\mathcal{O}(S+n\log S)$.
Для `TreeMap` получаем $\mathcal{O}(S\log|\Sigma|)$.

Замечание 1.8.1. Если бы мы научились сортировать строки над произвольным алфавитом за $\mathcal{O}(|S|)$, то для $\Sigma=\mathbb{Z}$, получилась бы сортировка целых чисел за $\mathcal{O}(|S|)$.

Часто размер алфавита считают $\mathcal{O}(1)$.
Например строчные латинские буквы – 26, или любимый для биологов $|\{A,C,G,T\}|=4$.

Лекция #2: Ахо-Корасик и Укконен

21 ноября 2022

2.1. Алгоритм Ахо-Корасика

Даны текст t и словарь $s_1, s_2, \ldots, s_m$, нужно научиться искать словарные слова в тексте. Простейший алгоритм, отлично работающий для коротких слов, – сложить словарные слова в бор и от каждой позиции текста i попытаться пройти вперёд, откладывая суффикс t_i вниз по бору, и отмечая все концы слов, которые мы проходим. Время работы – $\mathcal{O}(|t| \cdot \max |s_i|)$.

Ту же асимптотику можно получить, сложив все хеши всех словарных слов в хеш-таблицу, и проверив, есть ли в хеш-таблице какие-нибудь подстроки t длины не более $\max |s_i|$.

Давайте теперь соптимизируем первое решение также, как префикс-функция, позволяет простейший алгоритм поиска подстроки в строке улучшить до линейного времени. Обобщение префикс-функции на бор – суффиксные ссылки:

Def 2.1.1. $\forall$ *вершины бора* v*:*
$str[v]$ *– строка, написанная на пути от корня бора до* v*.*
$suf[v]$ *– вершина бора, соответствующая самому длинному суффиксу* $str[v]$ *в боре.*

$\forall$ позиции текста i насчитаем вершину бора v_i: $str[v_i]$ – суффикс $t[0{:}i)$, $|str[v_i]| \to \max$.
Пересчёт v_i**:**

```
v[0] = root, p = root
for (i = 0; i < |t|; i++)
  while (next[p][t[i]] == 0) // нет ребра
    p = suf[p]
  v[i+1] = p = next[p][t[i]]
```

Чтобы цикл `while` всегда останавливался введём фиктивную вершину `f` и
сделаем `suf[root] = f`, $\forall c$ `next[f][c] = root`.

Поиск словарных слов. Пометим все вершины бора, посещённые в процессе: `used[`v_i`] = 1`. В конце алгоритма поднимем пометки вверх по суффиксным ссылкам: `used[v]` $\Rightarrow$ `used[suf[v]]`. Для i-го словарного слова при добавлении мы запомнили вершину `end[i]`, тогда наличии этого слова в тексте лежит в `used[end[i]]`. Также можно насчитывать число вхождений.

Суффссылки. Чтобы всё это счастье работало осталось насчитать суффссылки.

Способ #1: полный автомат.

```
suf[v] = go[suf[parent[v]]][parent_char[v]];
go[v][c] = (next[v][c] ? next[v][c] : next[suf[v]][c]);
```

Мы хотим, чтобы от `parent[v]` и `suf[v]` всё было уже посчитано $\Rightarrow$ нужно или перебирать вершины в порядке bfs от корня **(1a)**, или считать эту динамику рекурсивно-лениво **(1b)**.

Способ #2: пишем bfs от корня и пытаемся продолжить какой-нибудь суффикс отца.

```
q <-- root
while q --> v:
  z = suf[parent[v]]
  while next[z][parent_char[v]] == 0:
```

```
    z = suf[z]
  suf[v] = next[z][parent_char[v]]
  for (auto[c,vertex] : next[v]) q <-- c
```

Этот способ экономнее по памяти, если `next` – не массив, а, например, `map<int,int>` (2).

Теорема 2.1.2. Время построения линейно от длины суммарной строк, но не от размера бора.

Доказательство. Линейность от размера бора ломается на примере «бамбук длины n из букв a, из листа которого торчат рёбра по n разным символам». Линейность от суммарной длины строк следует из того, что если рассмотреть путь, соответствующий $\forall$ словарному слову s_i, то при вычислении суффссылок от вершин именно этого пути, указатель z в `while` всё время поднимался, а затем опускался не более чем на 1 $\Rightarrow$ сделал не более $2|s_i|$ шагов. ■

Сравнение способов.

Пусть размер алфавита равен k, число вершин бора V, сумма длин строк в словаре S. Заметим $V \leqslant S$, но может быть сильно меньше, если у строк длинный общий префикс.

(1a) Ровно $\Theta(k \cdot V + S)$ времени, $\Theta(k \cdot V)$ памяти.
(1b) В худшем случае $k \cdot V$, но на практике за счёт ленивости быстрее.
(2) $\Theta(S)$ времени, $\Theta(V)$ памяти (линия и там, и там). Времени именно S, не V.

2.2. Суффиксное дерево, связь с массивом

Def 2.2.1. *Сжатый бор: разрешим на ребре писать не только букву, но и строку. При этом из каждой вершины по каждой букве выходит всё ещё не более одного ребра.*

Def 2.2.2. *Суффиксное дерево – сжатый бор построенный из суффиксов строки.*

Lm 2.2.3. Сжатое суффиксное дерево содержит не более $2n$ вершин.

Доказательство. Индукция: база один суффикс, 2 вершины, добавляем суффиксы по одному, каждый порождает максимум +1 развилку и +1 лист. ■

• Построение суффдерева из суффмассива+LCP

Пусть мы уже построили дерево из первых i суффиксов в порядке суффмассива. Храним путь от корня до конца i-го. Чтобы добавить $(i+1)$-й, поднимаемся до высоты $LCP(i, i{+}1)$ и делаем новую развилку, новый лист. Это несколько `pop`-ов и не более одного `push`-а. Итого $\mathcal{O}(n)$.

• Построение суффмассива+LCP из суффдерева

Считаем, что дерево построено от строки $s\$$ $\Rightarrow$ (листья = суффиксы).
Обходим дерево слева направо. Если в вершине используется неупорядоченный `map` для хранения рёбер, сперва отсортируем их. При обходе выписываем номера листьев-суффиксов.
$LCP(i, i+1)$ – максимально высокая вершина, из пройденных по пути из i в $i+1$.
Время работы $\mathcal{O}(n)$ или $\mathcal{O}(n \log |\Sigma|)$.

2.3. Суффиксное дерево, решение задач

• Число различных подстрок.

Это ровно суммарная длина всех рёбер. Так как любая подстрока есть префикс суффикса ⇒ откладывается от корня дерева вниз до «середины» ребра.

• Поиск подстрок в тексте.

Строим суффдерево от текста. ∀ строку s можно за $\mathcal{O}(|s|)$ искать в тексте спуском по дереву.

• Общая подстрока k строк.

Построим дерево от $s_1\#_1 s_2\#_2 \ldots s_k\#_k$, найдём самую глубокую вершину, в поддереве которой содержатся суффиксы k различных типов. Время работы $\mathcal{O}(\sum|s_i|)$, оптимально по асимптотике. Константу времени работы можно улучшать за счёт уменьшения памяти – строить суффдерево не от конкатенации, а лишь от одной из строк.

2.4. Алгоритм Укконена

Обозначение: $ST(s)$ – суффиксное дерево строки s.
Алгоритм Укконена – онлайн алгоритм построения суффиксного дерево. Нам поступают по одной буквы c_i, мы хотим за амортизированное $\mathcal{O}(1)$ из $ST(s)$ получать $ST(sc_i)$.

За квадрат это делать просто: храним позиции концов всех суффиксов, каждый из них продлеваем вниз на c_i, если нужно, создаём при этом новые рёбра/вершины.

Ускорение #1: суффиксы, ставшие листьями, растут весьма однообразно – рассмотрим ребро $[l, r)$, за которое подвешен лист, тогда всегда происходит `r++`. Давайте сразу присвоим $[l, \infty)$.

Теперь опишем жизненный цикл любого суффикса:
рождается в корне, ползёт вниз по дереву, разветвляется, становится саморастущим листом. Нам интересно обработать только момент разветвления.

<u>Lm</u> 2.4.1. ⌉ Суффикс длины k не разветвился ⇒ все более короткие тоже не разветвились.

Доказательство. Суффикс длины k не разветвился ⇒ он встречался в s как подстрока.
Все более короткие являются его суффиксами ⇒ тоже встречаются в s ⇒ не разветвятся. ■

Ускорение #2: давайте хранить только позицию самого длинного неразветвившегося суффикса. Пока он спускается по дереву, ничего не нужно делать. Как только он разветвится, нужно научиться быстро переходить к следующему по длине (отрезать первую букву).

Ускорение #3: отрезать первую букву = перейти по суффссылке, давайте от всех вершин поддерживать суффссылки. Если мы были в вершине, когда не смогли пойти вниз, теперь всё просто, перейдём по её суффссылке. Если же мы стояли посередине ребра и создали новую вершину v, от неё следует посчитать суффссылку. Для этого возьмём суффссылку её отца $p[v]$ и из $suf[p[v]]$ спустимся вниз на строку, соединяющую $p[v]$ и v.

```
void build(char *s):
  int N = strlen(s), VN = 2 * Ns;
  int vn = 2, v = 1, pos; // идём по ребру из p[v] в v, сейчас стоим в pos
  int suf[VN], l[VN], r[VN], p[VN]; // «ребро p[v] → v» = s[l[v]:r[v])
  map<char,int> t[VN]; // собственно рёбра нашего бора
  for (int i = 0; i < |Σ|; i++) t[0][i] = 1; // 0 = фиктивная, 1 = корень
  l[1] = -1, r[1] = 0, suf[1] = 0;
```

```
  for (int n = 0; n < N; n++):
    char c = s[n];
    auto new_leaf = [&]( int v ) {
      p[vn] = v, l[vn] = n, r[vn] = ∞, t[v][c] = vn++;
    };
    go:;
    if (r[v] <= pos) { // дошли до вершины, конца ребра
      if (!t[v].count(c)) { // по символу c нет ребра вперёд, создаём
        new_leaf(v), v = suf[v], pos = r[v];
        goto go;
      }
      v = t[v][c], pos = l[v] + 1; // начинаем идти по новому ребру
    } else if (c == s[pos]) {
      pos++; // спускаемся по ребру
    } else {
      int x = vn++; // создаём развилку
      l[x] = l[v], r[x] = pos, l[v] = pos;
      p[x] = p[v], p[v] = x;
      t[p[x]][s[l[x]]] = x, t[x][s[pos]] = v;
      new_leaf(x);
      v = suf[p[x]], pos = l[x]; // вычисляем позицию следующего суффикса
      while (pos < r[x])
        v = t[v][s[pos]], pos += r[v] - l[v];
      suf[x] = (pos == r[x] ? v : vn);
      pos = r[v] - (pos - r[x]);
      goto go;
    }
```

Теорема 2.4.2. Суммарное время работы n первых шагов равно $\mathcal{O}(n)$.

Доказательство. Понаблюдаем за величиной z «число вершин на пути от корня до нас». Пока мы идём вниз, z растёт, когда переходим по суффссылке, z уменьшается максимум на 1 $\Rightarrow$ возьмём потенциал $\varphi = -z$, суммарное число шагов вниз не больше n. ∎

2.5. LZSS

Решим ещё одну задачу – сжатие текста алгоритмом LZSS.
В отличии от использования массива, дерево даёт чисто линейную асимптотику и простейшую реализацию – насчитаем для каждой вершины $l[v]$ = самый левый суффикс в поддереве и при попытки найти $j < i$: $LCP(j, i) = \max$ будем спускаться из корня, пока $l[v] < i$.

Лекция #2: Хеширование

21 ноября 2022

2.6. Универсальное семейство хеш функций

[wiki] [итмо-конспект] [Carter,Wegman'1977]

Def 2.6.1. *Хеш-функция. Сжимающее отображение* $h : U \to M$, $|U| > |M|$, $|M| = m$.

Def 2.6.2. *Универсальная система хеш-функций (1-я версия определения).*
Множество хеш-функций $\mathcal{H}$ *– универсальная система, если*

$$\forall x, y\ x \neq y\colon \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \tfrac{1}{m}$$

Def 2.6.3. *Универсальная система хеш-функций (2-я версия определения* $\equiv$ *1-й).*

$$\forall x, y\ x \neq y\colon \sum_{h \in \mathcal{H}} [h(x) = h(y)] \leq \tfrac{|\mathcal{H}|}{m}$$

Теорема 2.6.4. $\mathcal{H}_{p,m} = \{(a, b) : x \to ((ax + b) \mod p) \mod m\}$, $a \in [1, p), b \in [0, p)$.

Утверждение: $\mathcal{H}_{p,m}$ универсально для $|U| = [0, p)$ и $M = [0, m)$ $(m \leqslant p)$.

Доказательство. Зафиксируем пару x, y. Проверим для неё определение универсальности.

$f(x) = ax + b \bmod p$, $f(y) = ay + b \bmod p$

$\big[f(x) = f(y) \Leftrightarrow a(x-y) \equiv 0 \bmod p\big] \Rightarrow (a \neq 0, x \neq y)$ по модулю p коллизий нет и
у нас есть биекция $\langle a, b\colon a \neq 0\rangle \leftrightarrow \langle f(x), f(y)\colon f(x) \neq f(y)\rangle$.

Осталось понять, для скольки хеш-функций $f(x) \equiv f(y) \bmod m$?

$$\sum_{(a,b), a \neq 0} [f(x) \equiv f(y) \bmod m] \overset{\text{биекция}}{=} \sum_{f(x) \neq f(y)} [f(x) \equiv f(y) \bmod m] \overset{(*)}{\leqslant} \tfrac{p(p-1)}{m} = \tfrac{|\mathcal{H}|}{m}$$

(*) Зафиксируем $f(x)$, будем перебирать $f(y) = f(x){+}1, f(x){+}2, \ldots$ В каждом блоке из m вариантов, только последний даёт $f(x) \equiv f(y) \bmod m \Rightarrow$ таких $f(y)$ ровно $\lfloor \frac{p-1}{m} \rfloor \leqslant \frac{p-1}{m}$. ■

2.7. Оценки для хеш-таблицы с закрытой адресацией

[wiki] [wiki-probability] [2-choice-hashing]

Пусть у нас есть универсальное семейство хеш-функций $\mathcal{H}$.
Хеш-таблица с закрытой адресацией (на списках) – вероятностный алгоритм.
Вся вероятностная часть заключается в том, что мы выбираем случайную $h \in \mathcal{H}$.

Операции `Find(x)`, `Del(x)` работают за длину списка.
Оценим для них матожидание времени работы.

$E[time(find(z))] = E[\text{длины списка}] = \sum\limits_{x \in table} Pr[h(x) = h(z)] = 1 + (n-1) \cdot \frac{1}{m} = \mathcal{O}(1 + \frac{n}{m})$, где

n – количество элементов в таблице, а m – размер таблицы (и диапазон хеш-функции).

Замечание 2.7.1. Мы оценили именно матожидание времени работы. Матожидание средней длины списка всегда $\frac{m}{n}$, даже если все элементы всегда класть в один список.

Теперь несколько утверждений без доказательства.

Утверждение 2.7.2. E[максимальной длины списка] $= \mathcal{O}(\log n)$

Утверждение 2.7.3. 2-choice hashing. Модифицируем хеш-таблицу: будем использовать две хеш-функции h_1, h_2 и при добавлении элемента будем выбирать из списков $a[h_1(x)], a[h_2(x)]$ список меньшей длины. Тогда E[максимальной длины списка] $= \Theta(\log\log n)$.

2.8. Оценки других функций для хеш-таблиц

Популярна функция $x \to x \bmod n$, где n размер таблицы (число списков, длина массива открытой адресации). Поскольку все x-ы могут иметь одинаковый остаток по модулю n, $\forall n \exists$ контртест. Новая идея, давайте брать случайное $n \in \P\sup[N, 2N)$, и хеш функцию из семейства $\mathcal{H} = size \in \P\sup[n, 2n), x \to x \bmod size$.

Lm 2.8.1. $\forall x \neq y\ Pr_{h\in\mathcal{H}}[h(x) = h(y)] \leqslant \frac{\log_n m}{n/\ln n}$.

Доказательство. Коллизия $(x - y) \equiv 0 \Rightarrow$ оцениваем вероятность попасть в простой делитель $(x - y)$. У числа до m не более $\log_n m$ простых делителей $\geqslant n$, простых на $[n, 2n) \approx \frac{n}{\ln n}$. ■

Для хеш-таблицы на списках нам этого хватает. У хеш-таблицы с открытой адресацией есть ещё проблема с тестом: $[1, 2, 3, \dots, \frac{n}{4}] + [1, 2, 3, \dots, \frac{n}{4}]$, на нём работает за квадрат.
TODO

2.9. (-) Фильтр Блюма

[wiki] [итмо-конспект]

Прелюдия.
Мы хотим вероятностную структуру данных, которая умеет делать всего две операции – добавлять x, и проверять, добавлен ли уже в структуру x.
Главная фича нашей структуры по сравнению с более простыми аналогами (массив, хеш-таблица) — очень мало памяти, $\mathcal{O}(n)$ бит.

Собственно структура.
Хотим хранить не более n x-ов. Для этого у нас есть m бит и k хеш-функций $h_1, \dots, h_k$.

```
bitset<m> a; // m нулей
Add(x): a[h1] = a[h2] = ... = a[hk] = 1;
Find(x): return a[h1] & a[h2] & ... & a[hk];
```

Собственно алгоритм окончен, осталось разобраться, с какой вероятностью это работает.
Во-первых, если `Find` вернул 0, элемент точно ещё не был добавлен.
Оценим ошибку `Find`-а, который вернул 1. При оценке для простоты предположим, что все nk значений, которые вернули k хеш-функций данных n элементов, равномерно распределены.

Pr[в i-й ячейке 0] $= (1-1/m)^{kn} \approx exp(\frac{-kn}{m}) \Rightarrow Pr$[ложного срабатывания `Find`] $= \left(1 - exp\left(\frac{-kn}{m}\right)\right)^k$

Посчитаем, какое при фиксированных $\langle n, m\rangle$ оптимально выбрать k?
Дифференцируем по k, решаем $Pr'(k) = 0$, получаем $k = (\ln 2)\cdot\frac{m}{n}$ и $Pr[error] = 2^{-k} = 0.6185^{m/n}$.
Например, если нам дали лишь $5n$ бит на всё про всё, мы достигли вероятности ошибки ≈9%.

2.10. (-) Совершенное хеширование

[wiki] [итмо-конспект] [cs.cmu.edu] [practice:bbhash] [Fredman'1984]

Вспомним, как мы ищем младший бит 64-битного целого x.

```
uint64_t i2(uint64_t x) { return x & (~x + 1); }
```

Таким образом мы получили младший бит i в форме 2^i.
Осталось сделать последнее действие, $2^i \to i$.

```
int table[67];
void init() { forn(i, 64) table[2^i % 67] = i; }
int get(uint64_t i2) { return table[i2 % 67]; }
```

Здесь все остатки по модулю 67 различны $\Rightarrow$
мы получили детерминированный алгоритм поиска номера младшего бита за $\mathcal{O}(1)$.

$x \to x \bmod 67$ – пример совершенной хеш-функции для множества фиксированных ключей $\{x_0 = 2^0, x_1 = 2^1, \dots, x_{63} = 2^{63}\}$.

Def 2.10.1. *Совершенная хеш-функция – любая инъективная функция.*

То есть, хеш-функция, у которой по определению не возникает коллизий.
Чтобы построить такую хеш-функцию, нам, конечно, нужно заранее знать набор ключей $x_1, x_2 \dots, x_k$ (см. пример с младшим битом).

Задача построения совершенной хеш-функции.
Дан набор ключей $x_1, x_2, \dots, x_n$, найти такую функцию
$h\colon X = \{x_1, x_2, \dots, x_n\} \to [0, m) \cap \mathbb{Z}$, что все $h(x_i)$ различны и функция вычислима за $\mathcal{O}(1)$.
m будем называть размером хеш-функции (m в итоге станет размером массива хеш-таблицы).

2.10.1. (-) Одноуровневая схема

Возьмём $m = n^2$ и любое универсальное семейство хеш-функций $\mathcal{H}\colon X \to [0, m)$.
Например, $\mathcal{H} = \{h_{a,b}(x) = ((ax + b) \bmod p) \bmod m\}$, $m \leqslant p$, p простое.

E(количества коллизий) $= \frac{1}{2}n(n-1) \cdot \frac{1}{m} < \frac{1}{2} \Rightarrow$ с вероятностью хотя бы $\frac{1}{2}$ коллизий нет $\Rightarrow$

Алгоритм: берём случайную $h \in \mathcal{H}$, пока не повезёт. E(времени работы) $= \mathcal{O}(n)$, размер n^2.

2.10.2. (-) Двухуровневая схема

Возьмём $m = n$ и любое универсальное семейство хеш-функций $\mathcal{H}\colon X \to [0, m)$.
Выберем случайную $h \in \mathcal{H}$ и посмотрим на списки $A_y = \{x \mid h(x) = y\}$.
Для каждого A_y построим одноуровневую совершенную хеш-функцию f_y.
Оценим суммарный диапазон всех внутренних уровней:

$$E(\sum_y |A_y|^2) = \sum_{ij} Pr[h(x_i) = h(x_j)] = n \cdot 1 + n(n-1) \cdot \tfrac{1}{n} = 2n-1$$

Осталось внутренние уровни выписать в один массив.
Для этого берём префиксные суммы: `pref_sum[y+1] = pref_sum[y] +` $|A_y|^2$.

Итого, наша хеш-функция: `int hash(x) { y = h(x); return pref_sum[y] + f[y](x); }`

E(времени работы) $= \mathcal{O}(n)$, E(размера) $= 2n-1$.

2.10.3. (*) Графовый подход

[[RandomGraphs|book]]
Начнём с забавного факта

Утверждение 2.10.2. Рассмотрим случайны неорграф из $3n$ вершин и n рёбер.
Pr[отсутствия циклов] $\geqslant \frac{1}{2}$.

Строим совершенную хеш-функцию от $x_1, \dots, x_n$.
Возьмём $m = 3n$ и любое универсальное семейство хеш-функций $\mathcal{H}\colon X \to [0, m)$.
Выберем $h_1, h_2 \in H$. Каждому x_i сопоставим ребро $\langle h_1(x_i), h_2(x_i)\rangle$.
Если граф не ацикличен, выберем другие хеш-функции, повторим. Пусть ацикличен.

Тогда запишем значения f в вершинах и определим $hash(x_i) = (f[h_1(x_i)] + f[h_2(x_i)]) \bmod n$.

Как записать значения f, чтобы было верно $hash(x_i) = i$? Граф – лес. В каждом дереве в корне пишем ноль (любое число), остальные числа расставит `dfs` по дереву.

2.11. (*) Хеширование кукушки

[wiki] [Pagh,Rodler'2001]

За сколько работает хеш-таблица на списках?
`Add` за $\mathcal{O}(1)$ в худшем, `Find` и `Del` за $\mathcal{O}(1)$ в среднем.

Сейчас мы придумаем, как сделать наоборот:
`Add` за $\mathcal{O}(1)$ в среднем, `Find` и `Del` за $\mathcal{O}(1)$ в худшем.

Заведём массив a размера $m = 3n$ и две хеш-функции: $h_1, h_2 \in \mathcal{H}_m$.
$\forall$ элемент x всегда живёт в одной из двух ячеек — $h_1(x)$ или $h_2(x)$.
`Find` и `Add`, очевидно, обращаются не более чем к двум ячейкам — $h_1(x)$ и $h_2(x)$.
`Add` сперва ищет свободную среди $h_1(x), h_2(x)$. Если заняты обе, начнём выталкивать элементы:

```
AddIfBothAreOccupied(x):
  i = h1(x)
  while (a[i] != -1):
    y = a[h1(x)], a[h1(x)] = x, x = y
    i = h1[y] + h2[y] - i // вторая из двух ячеек, где может жить y
```

Почему и за сколько работает `Add`?
Представим себе граф с рёбрами между $h_1(x)$ и $h_2(x)$. Как мы уже знаем, он с большой вероятностью ацикличен (значит `Add` хотя бы не циклится). Теперь осталось дождаться, пока голос Белы Боллобаша (автор книги Random Graphs) не нашепчет нам недостающую мудрость — глубина случайного дерева $\mathcal{O}(\log n)$.

Лекция #3: Теория чисел

пропушенная лекция

3.1. (-) Решето Эратосфена

Задача: найти все простые от 1 до n.
Решето Эратосфена предлагает вычёркивать числа, кратные уже найденным простым:

```
vector<bool> is_prime(n + 1, 1); // Хорошо по памяти даже при n ≈ 10^9!
is_prime[0] = is_prime[1] = 0;
for (int i = 2; i <= n; i++)
  if (is_prime[i]) // Нашли новое простое!
    for (int j = i + i; j <= n; j += i)
      is_prime[j] = 0; // cnt++, чтобы определить константу
```

Замечание 3.1.1. Сейчас для каждого числа мы находим лишь один бит. Код легко модифицировать, чтобы для каждого числа находить наименьший простой делитель.

Данную версию кода можно соптимизировать в константу раз, пользуясь тем, что у любого не простого числа есть делитель не более корня.

```
for (int i = 2; i * i <= n; i++) // cnt++, чтобы определить константу
  if (is_prime[i])
    for (int j = i * i; j <= n; j += i)
      is_prime[j] = 0; // cnt++, чтобы определить константу
```

Можно ещё соптимизить: мы ищем только нечётные простые $\Rightarrow$
внешний цикл можно вести только по нечётным i, а во внутреннем прибавлять $2i$.
Теперь у нас три версии решета, отличающиеся не большими оптимизациями.
Эмпирический запуск при $n = 10^6$ даёт значения `cnt`: $3.7752\,n, 2.1230\,n, 0.8116\,n$ соответственно.

Теорема 3.1.2. Обе версии работают за $\Theta(n \log\log n)$.

Доказательство. При достаточно больших k верно $0.5\,k\log k \leqslant p_k \leqslant 2\,k\log k$ (без док-ва).
Так как $\lceil\frac{n}{p_k}\rceil = \mathcal{O}(1) + \frac{n}{k\log k}$, время работы 1-й версии равно

$$n + \mathcal{O}(n) + \sum_{k=\mathcal{O}(1)}^{p_k \leqslant n} \lceil\tfrac{n}{p_k}\rceil = \Theta(n + \sum_{k=\mathcal{O}(1)}^{n/\log n} \tfrac{n}{k\log k}) =$$

$$\Theta(n + n\int\limits_{\mathcal{O}(1)}^{n} \tfrac{1}{x\log x}\,dx) = \Theta(n + n\log\log n - \mathcal{O}(1)) = \Theta(n\log\log n)$$

■

• Более быстрое решение.

Чтобы найти все простые от 1 до n за $\mathcal{O}(n)$, достаточно модифицировать алгоритм так, чтобы каждое составное x помечать лишь один раз, например, наименьшим простым делителем x.

Пусть $d[x]$ – номер наименьшего простого делителя x ($primes[d[x]]$ – собственно делитель).
Пусть $x = primes[d[x]] \cdot y \Rightarrow (d[y] \geqslant d[x] \vee y = 1)$
Алгоритм: перебирать y, а для него потенциальные $d[x]$ (простые не большие $d[y]$).

```
vector<int> primes, d(n + 1, -1);
for (int y = 2; y <= n; y++)
  if (d[y] == -1)
    d[y] = primes.size(), primes.push_back(\red{y});
  for (int i = 0; i <= d[y] && y * primes[i] <= n; i++)
    d[y * primes[i]] = i; // x=y*primes[i], i=d[x]
```

3.2. (-) Решето и корень памяти (на практике)

Пусть нам нужно найти все простые на промежутке $(n-\sqrt{n}..n] = (l..r]$.
Это можно сделать за $\mathcal{O}(\sqrt{n}\log\log n)$ времени и $\mathcal{O}(\sqrt{n})$ памяти.
1. У не простого числа до n есть простой делитель $\leqslant \sqrt{n} \Rightarrow$ посчитаем все простые до $\sqrt{n}$.
2. Будем этими простыми «просеивать» нужный нам интервал... Для простого p, сперва нужно пометить $l-(l \bmod p)+p$, затем с шагом p до r, как в обычном решете.

3.3. (-) Вычисление мультипликативных функций функций на $[1, n]$

Def 3.3.1. *Функция мультипликативна* $\Leftrightarrow \forall a, b\colon (a,b)=1\ f(ab)=f(a)f(b)$.

Т.е. имея разложение числа на простые $x=\prod_i p_i^{\alpha_i}$, имеем $f(x)=f(x/p_1^{\alpha_1})f(p_1^{\alpha_1})$

• Примеры:

Простой делитель n		`p[x] = primes[d[x]]`
Удобное обозначение		`y[x] = x / p[x]`
Степень этого делителя		`deg[x] = (d[y[x]] == d[x] ? deg[y[x]] + 1 : 1)`
Результат отщепления $p_1^{\alpha_1}$		`rest[x] = (d[y[x]] == d[x] ? rest[y[x]] : y[x])`
Собственно $p_1^{\alpha_1}$		`term[x] = x / rest[x]`
Число взаимнопростых	$\varphi(n)=n\prod_i \frac{p_i-1}{p_i}$	`phi[x] = phi[rest[x]] * (term[x]/p[x])* (p[x] - 1)`
Число делителей	$\sigma_0(n)=\prod_i(\alpha_i+1)$	`s0[x] = s0[rest[x]] * (deg[x] + 1)`

Чуть сложнее посчитать сумму делителей: $\sigma_1(n)=\prod_i(p_i^0+p_i^1+\dots+p_i^{\alpha_i})=\prod_i \frac{p_i^{\alpha_i+1}-1}{p_i-1}$.
Итого: `s1[x] = s1[rest[x]] * (term[x] * p[x] - 1)/ (p[x] - 1)`.

3.4. (*) Число простых на $[1, n]$ за $n^{2/3}$

Минимальный простой делитель x обозначаем $d[x]$. Массив d на $[1, m]$ мы умеем насчитывать решетом Эратосфена за $\mathcal{O}(m)$. $d[1] := +\infty$. Заодно мы нашли за $\mathcal{O}(m)$ все простые на $[1, m]$.

Def 3.4.1. $\pi(n)$ – *количество простых чисел от* 1 *до* n

Def 3.4.2. $f(n,k)$ – $|\{x\in[1,n] \mid d[x]\geqslant p_k\}|$, *где* p_k – k*-е простое.*

<u>Lm</u> 3.4.3. $\pi(n)=\pi(\sqrt{n})+f(n,k(\sqrt{n}))$, где $k(x)$ – номер первого простого, большего x.

Теперь $\pi(\sqrt{n})$ и $k(\sqrt{n})$ найдём за линию решетом, а $f(n,k(\sqrt{n}))$ рекурсивно по рекурренте:

$$f(n,k)=f(n,k-1)-f(\lfloor n/p_{k-1}\rfloor, k-1)$$

Поясним формулу: $\big(f(n,k-1)-f(n,k)\big)$ – количество чисел вида $p_{k-1}\cdot x$, где $d[x]\geqslant p_{k-1}$.
Количество таких x на $[1,n]$ есть $f(\lfloor n/p_{k-1}\rfloor, k-1)$.
База: $f(n,0)=n$, $f(0,k)=0$.

- **Самое важное отсечение.**

$f(n,k)$ есть количество пар $\langle i, d[i]\rangle : i \leqslant n \wedge d[i] \geqslant k$.

Зафиксируем некое m, предподсчитаем $d[1..m]$, теперь $\forall n \leqslant m$ $f(n,k)$ – запрос на плоскости. Более того, мы можем вычислять f процедурой вида:

```
int result = 0;
void calc(int n, int k, int sign):
  if (k == 0)
    result += sign * n;
  else if (n <= m)
    queries[n].push_back({k, sign});
  else
    calc(n, k - 1, sign), calc(n / p[k - 1], k - 1, -sign);
```

Тогда в итоге мы получим пачку из q запросов на плоскости, уже отсортированных по n. Обработаем их одним проходом сканирующей прямой с деревом Фенвика за $\mathcal{O}((m+q)\log m)$.

- **Оценка времени работы, выбор m.**

Если мы считаем $\pi(n)$, пришли рекурсией в состояние (x,k), то $x = \lfloor n/y \rfloor$.
Посчитаем число состояний рекурсии (x,k), что в $f(x,k)$ отсечение $x \leqslant m$ ещё не сработало, а в $f(x/p_{k-1}, k-1)$ уже сработало.

1. Есть не более $\sqrt{n}$ таких состояний с $x = n$.
2. Если же $x \neq n, x = \lfloor n/y \rfloor \Rightarrow p_{k-1} \leqslant y \leqslant n/m$, т.к. простые мы перебираем по убыванию.

Осталось посчитать «число пар $\langle x,k\rangle$» = «число пар $\langle y, p_{k-1}\rangle$», их $\mathcal{O}((n/m)^2)$.
Вспомнив, что простых до t всего $\Theta(t/\log t)$, можно дать более точную оценку: $\mathcal{O}((n/m)^2/\log\frac{n}{m})$.
Каждая пара даст 1 запрос, увеличит время scanline на $\log m$.

В предположении $m = \Theta(n^\alpha)$, $\log m = \Theta(\log\frac{n}{m}) \Rightarrow$ общее время работы $\mathcal{O}(m\log m + (n/m)^2)$.
Асимптотический минимум достигается, как обычно при $m\log m = (n/m)^2 \Rightarrow m\log^{1/3} m = n^{2/3}$.

Упражнение 3.4.4. Итоговое время работы – $m\log m = \Theta((n\log n)^{2/3})$.

- **Другие оптимизации.**

Предлагают предподсчёт для малых k: $k \leqslant 8$ на практике, $k = \Theta(\log n/\log\log n)$ в теории.
Для $n < p_{k-1}$ можно отвечать не за log, а за $\mathcal{O}(1)$.

3.5. Определения

Def 3.5.1. $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p$ *поле остатков по модулю* p.

Def 3.5.2. $(\mathbb{Z}/m\mathbb{Z})^*$ *Группа по умножению* $\{a\colon (a,m) = 1 \wedge 1 \le a < m\}$.

Def 3.5.3. *Линейное диофантово уравнение* (a, b, c *даны, нужно найти* x, y).

$$ax + by + c = 0;\ \ x, y \in \mathbb{Z}$$

Деление: $\frac{a}{b} = a \cdot b^{-1}$

• Алгоритм Евклида

Используется для подсчёта gcd (greatest common divisor):
$gcd(a,b) = gcd(a-b, b)$, повторяя вычитание много раз, получаем $gcd(a,b) = gcd(a \bmod b, b)$

Итого: `int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }`
Замечание: мы одинаково действуем и при $a \geqslant b$, и при $a < b$. Что сделает код в этом случае?

Время работы: 1 шаг, чтобы получить $a \geqslant b$, далее заметим $\min(b, a \bmod b) \leqslant \frac{a}{2} \Rightarrow$ за каждые 2 шага a будет уменьшаться минимум вдвоё.

3.6. Расширенный алгоритм Евклида

Задача: найти x и y: $ax + by = gcd(a,b)$
Повторим шаг обычного Евклида: найдём x_1 и y_1 для b и $a \bmod b$: $bx_1 + (a \bmod b)y_1 = gcd(a,b)$.
Заметим $a \bmod b = a - \lfloor\frac{a}{b}\rfloor b \Rightarrow x = y_1, y = x_1 - \lfloor\frac{a}{b}\rfloor y_1$.

```
def euclid(a, b): # returns (x,y): ax + by = gcd(a, b)
  if b == 0: return 1, 0
  x, y = euclid(b, a % b);
  return y, x - (a // b) * y # целочисленное деление
```

Нерекурсивная реализация: база, $a \cdot 1 + b \cdot 0 = a$
$a \cdot 0 + b \cdot 1 = b$
Мы можем добавить переход из двух строк $a \cdot x_i + b \cdot y_i = r_i$
$a \cdot x_{i+1} + b \cdot y_{i+1} = r_{i+1}$
в новую $r_{i+2} = r_{i+1} \bmod r_i = r_{i+1} - kr_i$ (k – частное)
$x_{i+2} = x_{i+1} - kx_i$
$y_{i+2} = y_{i+1} - ky_i$

Алгоритм: `while` $r_{i+1} \neq 0$ `do` получить новую строку, `i++`.
В конце алгоритма ответ содержится в x_i, y_i, r_i.

• Решение диофантового уравнения:

Если $c \bmod gcd(a,b) \neq 0$, то решений нет.
Иначе найдём x, y: $ax + by = gcd(a,b)$ и домножим уравнение на $c/gcd(a,b)$.

3.7. (-) Свойства расширенного алгоритма Евклида

Следующие утверждения обсуждались и доказывались на практике:

(a) $\forall i$ в строке $ax_i + by_i = r_i$ верно, что $(x_i, y_i) = 1$

(b) $\max|x_i| \leqslant |b|$ и $\max|y_i| \leqslant |a| \Rightarrow$
$\forall$ типа `T`, если исходные данные помещаются в тип `T`, то и все промежуточные тоже.

Также на практике были решены следующие задачи:

(c) Найдите класс решений уравнения $ax \equiv b (\text{mod } m)$

(d) Найдите x, y: $ax + by = c$, $|x| + |y| \to \min$

3.8. Обратные в $(\mathbb{Z}/m\mathbb{Z})^*$ и $\mathbb{Z}/p\mathbb{Z}$

Задача: a и m даны, хотим найти x: $\quad a \cdot x \equiv 1 \bmod m$
Первый способ – решить диофантово уравнение $ax + my = 1 = gcd(a, m)$
Другой способ – воспользоваться малой теоремой Ферма или теоремой Эйлера:

$$a^{p-1} = 1 \bmod p \Rightarrow x = a^{p-2} \text{ (для простого)}$$
$$a^{\varphi(m)} = 1 \bmod m \Rightarrow x = a^{\varphi(m)-1} \text{ (для произвольного)}$$

Замечание 3.8.1. Функцию Эйлера считать долго!

$$\text{Пусть } n = \prod p_i^{\alpha_i} \Rightarrow \varphi(n) = n \prod \frac{p_i - 1}{p_i}, \text{ для вычисления нужна факторизация } n$$

Замечание 3.8.2. Способ с расширенным Евклидом лучше: работает $\forall m$, все промежуточные значения по модулю $\leqslant a, b$ (для $a, b \leqslant 2^{63}$ хватает `int64_t`, для Ферма нет).

3.9. (-) Возведение в степень за $\mathcal{O}(\log n)$

Сводим к $\mathcal{O}(\log n)$ умножениям. Считаем, что одно умножение работает за $\mathcal{O}(1)$.

```
def pow(x, n):
  if n == 0: return 1
  return pow(x**2, n // 2)**2 * (x if n % 2 == 1 else 1)
```

Замечание 3.9.1. При возведении в степень, если исходные данные в типе `T`, то при умножении по модулю мы можем столкнуться с переполнением `T`...
Есть два решения: или умножение за $\mathcal{O}(1)$ превратить в $\mathcal{O}(\log n)$ сложений тем же алгоритмом, или использовать вещественные числа:

```
int64 mul(int64 a, int64 b, int64 m): // 0 ⩽ a,b < m
  int64 k = (long double)a * b / m; // посчитано с погрешностью!
  int64 r = a * b - m * k; // в знаковом типе формально это UB =(
  while (r < 0) r += m;
  while (r >= m) r -= m;
  return r;
```

3.10. (-) Обратные в $\mathbb{Z}/p\mathbb{Z}$ для чисел от 1 до k за $\mathcal{O}(k)$

Сейчас одно обращение работает за $\mathcal{O}(\log p) \Rightarrow$ задачу мы умеем решать только за $\mathcal{O}(k \log p)$.
Время улучшать! Используем динамику: зная, обратные к $1..i{-}1$ найдём i^{-1}.

Теорема 3.10.1. $i^{-1} = -\lfloor \frac{m}{i} \rfloor \cdot (m \bmod i)^{-1}$

Доказательство. $0 \equiv m = (m \bmod i) + i \cdot \lfloor \frac{m}{i} \rfloor$. Домножим на $(m \bmod i)^{-1}$:
$0 \equiv 1 + (m \bmod i)^{-1} \cdot i \cdot \lfloor \frac{m}{i} \rfloor \Rightarrow i^{-1} \equiv -(m \bmod i)^{-1} \lfloor \frac{m}{i} \rfloor$ ∎

3.11. (-) Первообразный корень

Пусть p – простое. Работаем в $(\mathbb{Z}/p\mathbb{Z})^*$ (группа по умножению по модулю p).
На алгебре вы доказывали (и хорошо бы помнить, как именно!), что

$$\exists g \colon \{1, 2, \dots, p{-}1\} = \{g^0, g^1, \dots, g^{p-2}\}$$

Такое g называется *первообрАзным корнем.*

• Задача проверки.

Дан g, проверить, является ли он первообразным корнем.
Если не является, то $ord\, g < p{-}1$, при этом $ord\, g \,|\, p{-}1$. То есть, достаточно перебрать все d делители $p{-}1$, и для каждого возведением в степень проверить, что $g^d \neq 1$.
Возведение в степень работает за $\mathcal{O}(\log p)$, если p помещается в машинное слово.
На длинных числах умножение и деление с остатком по модулю работают $\mathcal{O}(\log p \log\log p)$, итого возведение в степень за $\mathcal{O}(\log^2 p \log\log p)$.
Делители перебирать нужно не все, а только вида $\frac{p-1}{\alpha}$, где α – простой делитель $p{-}1$.
Обозначим p_k – k-е простое. Далее будем без доказательства пользоваться тем, что $p_k \geqslant k \log k$.

<u>Lm</u> 3.11.1. Пусть $f(x)$ – число различных простых делителей у $x \Rightarrow \forall x\ f(x) = \mathcal{O}(\frac{\log x}{\log\log x})$

Доказательство. Худший случай: x – произведение минимальных простых.
$x = \prod_{i=1..k} p_i \geqslant \prod_{i=1..k} i \Rightarrow \log x \geqslant \sum_{i=1..k} \log i = \Theta(k \log k) \Rightarrow k = \mathcal{O}(\frac{\log x}{\log\log x})$. ∎

<u>Теорема</u> 3.11.2. Мы научились проверять кандидат g за $\mathcal{O}(\mathtt{FACT} + \log^3 p)$.

Факторизация нужна, как раз чтобы найти простые делители $p{-}1$.

• Задача поиска.

И сразу решение: ткнём в случайное число, с хорошей вероятностью оно подойдёт.
Из детерминированных решений популярным является перебор в порядке $1, 2, 3, \dots$
Shoup'92 доказал, что в предположении обобщённой Гипотезы Римана, нужно $\mathcal{O}(log^6 p)$ шагов.

Обозначим как $G(p)$ множество всех первообразных корней для p.

<u>Теорема</u> 3.11.3. a – случайное от 1 до $p{-}1 \Rightarrow Pr[a \in G(p)] = \Omega(\frac{1}{\log\log p})$ (т.е. хотя бы столько).

Доказательство. Пусть g – $\forall$ первообразный, тогда $G(p) = \{g^i \mid (i, p{-}1) = 1\} \Rightarrow \frac{G(p)}{p-1} = \frac{\varphi(p-1)}{p-1}$.
Осталось научиться оценивать $\frac{\varphi(n)}{n} = \prod_{q|n} \frac{q-1}{q} \geqslant \prod \frac{i \log i - 1}{i \log i}$, где q – простые делители n.
Логарифмируем: $\log \frac{\varphi(n)}{n} \geqslant \sum \log(i \log i{-}1) - \sum \log(i \log i) = \Theta(-\sum_{i=1}^{k} \frac{1}{i \log i})) = \Theta(-\log\log k)$
Здесь k – число простых делителей. Получаем $\frac{\varphi(n)}{n} \geqslant \Theta(\frac{1}{\log k}) = \Theta(\frac{1}{\log\log n})$. ∎

Следствие 3.11.4. Получили ZPP-алгоритм поиска за $\mathcal{O}(\mathtt{FACT} + \log^3 p\, \log\log p)$.

Замечание 3.11.5. Пусть есть алгоритм T, который работает корректно, если дать ему правильный первообразный корень. Пусть мы ещё умеем проверять корректность результата T. Тогда проще всего вместо того, чтобы искать первообразный корень, $\approx \log\log p$ раз запустить алгоритм, подсовывая ему случайные числа.

3.12. Криптография. RSA.

Два типа шифрования:

Симметричная криптография. Один и тот же ключ позволяет и зашифровать, и расшифровать сообщение. Примеры шифрования: xor с ключом; циклический сдвиг алфавита.

Криптография с открытым ключем. Боб хочет послать сообщение Алисе и шифрует его *открытым ключем* Алисы (e), ключ (e) знают все. Для расшифровки Алисе понадобится ее закрытый ключ (d), который знает только она. Сами функции для шифровки и расшифровки открыты, их знают все.

• **RSA.** (Rivest, Shamir, Adleman, 1977)

Выберем два больших простых числа p, q. Посчитаем $n = pq, \varphi(n) = (p-1)(q-1)$.
Выберем случайное $1 \leqslant e < \varphi(n)$, посчитаем d: $ed \equiv 1 \bmod \varphi(n)$.
Итого: генерим случайно p, q, e; вычисляем n, $\varphi(n)$, d.
Тогда открытым ключем будет пара $\langle e, n\rangle$, а закрытым – $\langle d, n\rangle$.
Действия Боба для шифрования: $m \to \mu = m^e \bmod n$
Действия Алисы для дешифровки: $\mu \to m = \mu^d \bmod n$
Проверим корректность: $(m^e)^d = m^{ed} = m^{\varphi(n)\cdot k+1} \equiv 1^k \cdot m^1 = m$.

Алгоритм надежен настолько, насколько сложна задача факторизации чисел.
Числа умеют факторизовать так (более полный список на wiki):

(a) $\mathcal{O}(n^{1/2})$ – тривиальный перебор всех делителей до корня.
(b) $\mathcal{O}(n^{1/4} \cdot gcd)$ – Эвристика Полларда, была в главе про вероятностные алгоритмы.
(c) $L_n(1/2, 2\sqrt{2})$ – алгоритм Диксона-Крайчика (есть на 3-м курсе)
(d) $L_n(1/2, 2)$ – метод эллиптических кривых (алгоритм Ленстры)
(e) $L_n(1/3, (32/9)^3)$ – SNFS

Здесь $L_n(\alpha, c) = \mathcal{O}(e^{(c+o(1))(\log n)^\alpha})$.
При $0 < \alpha < 1$ получаем L_n между полиномом и экспонентой.
При $\alpha = 1$ получаем L_n – ровно полином $n^{c+o(1)}$.

Обычно в RSA используют ключ длины $k = 2048$.
При шифровке/расшифровке используют $\mathcal{O}(k)$ операций деления по модулю,
её мы скоро научимся реализовывать за $\mathcal{O}(k^2/w^2)$ и $\mathcal{O}(k \log^2 k)$, оптимальное время – $\mathcal{O}(k \log k)$.
Итого: простейшая реализация RSA даёт время $\mathcal{O}(k^3)$, оптимальная – $\mathcal{O}(k^2 \log k)$ и $\mathcal{O}(k^3/w^2)$.

• **Взлом RSA.**

На практике разобраны два случая:

(a) Вариант взлома при $e = 3$.
(b) Вариант взлома через Оракул, который ломает случайные сообщения с вероятностью 1%.

3.13. Протокол Диффи-Хеллмана

Есть Алиса и Боб. Им и вообще всем людям на Земле известны числа g и p.
Алиса и Боб хотят создать неизвестный более никому секретный ключ.
Для этого они пользуются следующим алгоритмом (протоколом):

1. Алиса генерируют большое случайное число a, Боб b.
2. Алиса передаёт Бобу по открытому каналу (любой может его слушать) $g^a \bmod p$
3. Боб передаёт Алисе по открытому каналу $g^b \bmod p$
4. Алиса знает $\langle a, g^b \rangle \Rightarrow$ может вычислить $g^{ab} \bmod p$, аналогично Боб. Ключ готов.

Предполагается, что злоумышленник не может вмешаться в процесс передачи данных, но может все данные перехватить. Злоумышленник в итоге знает g, g^a, g^b, p, но не знает a и b.

Оказывается, что задача «получить g^{ab} по этим данным» не проще дискретного логарифмирования, а она не проще факторизации.

3.14. (-) Дискретное логарифмирование

Задача схожа по записи с обычным логарифмированием: $a^x = b \Rightarrow x = \log_a b$.
Собственно её нам и предстоит решить, только все вычисления по модулю m.
Заметим, что x имеет смысл искать только в диапазоне $[0, \varphi(m))$ и $\varphi(m) < m$.

Общая идея решения: корневая по x. Любую степень корнем поделим на две части...
Возьмём $k = \lceil\sqrt{m}\rceil$, построим множество пар $B = \{\langle a^0, 0\rangle, \langle a^k, k\rangle, \langle a^{2k}, 2k\rangle, \dots, \langle a^{(k-1)k}, (k-1)k\rangle\}$.
Пусть x существует, поделим его с остатком на k: $x = ik + j \Rightarrow a^x = a^{ki}a^j \wedge \langle a^{ki}, ki\rangle \in B$.
Заметим, что $a^{ki}a^j = b \Leftrightarrow a^{ki} = ba^{-j}$. Осталось перебрать j:

```
a1 = inverse(a) # один раз за O(log p)
for j in range(k):
  if b in B: # с точки зрения реализации B - словарь
    x = j + B[b] # словарь ;)
  b = mul(b, a1) # по модулю! в итоге на j-й итерации у нас под рукой ba^{-j}
```

Если `B` – хеш-таблица, то суммарное время работы $\mathcal{O}(\sqrt{p})$.

Замечание 3.14.1. Если для каждого b в `B[b]` хранить весь список индексов, код легко модифицировать так, чтобы он находил все решения.

3.15. (-) Корень k-й степени по модулю

Решаем уравнение вида $x^k \equiv a \bmod p$. Даны k, b, $p \in \mathbb{P}$.
Дискретно прологарифмируем по основанию первообразного кореня: $a = g^b$, x ищем в виде g^y.
Получаем $g^{ky} \equiv g^b \bmod p \Leftrightarrow ky \equiv b \bmod p{-}1 \Leftrightarrow y \equiv \frac{b}{k} \bmod p{-}1, x = g^y$.
Время работы – логарифмирование + деление, т.е. $\sqrt{p} + \log p$.

3.16. (-) КТО

Китайская Теорема об Остатках (К.Т.О.) была у вас на алгебре в простейшем виде:

Теорема 3.16.1. $\begin{cases} x \equiv a_i(m_i) \\ \forall i \neq j \ (m_i, m_j) = 1 \end{cases} \Rightarrow \exists! a \in [0, M) \colon x \equiv a(M)$, где $M = \prod m_i$.

Также показывалось, что $a = \sum a_i e_i$, где e_i подбиралось так, что $e_i \equiv 1(m_i), \forall j \neq i \ e_i \equiv 0(m_j)$.
Собственно, если обозначить $t = \prod_{j\neq i} m_j$, то $e_i = (t \cdot t^{-1}(m_i)) \bmod M$.

Сейчас мы пойдём чуть дальше и рассмотрим случай произвольных модулей m_i.
Первым делом $\forall i$ факторизуем $m_i = \prod p_{ij}^{\alpha_{ij}}$. И заменим сравнения на $\forall j \ x \equiv a_i \bmod p_{ij}^{\alpha_{ij}}$.
Для каждого простого p оставим только главное сравнение: p^α с максимальным α.
Нужно проверить, что любые сравнения по модулям вида p^β главному не противоречат.
Итого за $\mathcal{O}(\mathtt{FACT})$ мы свели задачу к КТО, или нашли противоречие.

3.16.1. (-) Использование КТО в длинной арифметике.

Пусть нам нужно посчитать X – значение арифметического выражения. Во время вычислений, возможно, появляются очень длинные числа, но мы уверены, что $X \leqslant L$ (L дано).
Возьмём несколько случайных 32-битных простых p_i, чтобы их произведение было больше L.
Понадобится $k \approx \log L$ чисел. Теперь k раз найдём $X \bmod p_i$, оперируя только с короткими числами, а затем по формулам из КТО соберём $X \bmod \prod p_i$.
Поскольку $X \leqslant L < \prod p_i$, мы получили верный X.

Лекция #4: Линейные системы уравнений

28 ноября 2022

СЛАУ – Система линейных алгебраических уравнений.

Решением СЛАУ мы сейчас как раз и займёмся. Заодно научимся считать определитель матрицы, обращать матрицу, находить координаты вектора в базисе.

- **Постановка задачи.**

Дана система уравнений $\begin{cases} a_{00}x_0 + a_{01}x_1 + \dots + a_{0\,n-1}x_{n-1} = b_0 \\ a_{01}x_0 + a_{11}x_1 + \dots + a_{0\,n-1}x_{n-1} = b_1 \\ \dots \\ a_{m-1\,0}x_0 + a_{m-1\,1}x_1 + \dots + a_{m-1\,n-1}x_{n-1} = b_{m-1} \end{cases}$

Для краткости, будем записывать $Ax = b$, где A – матрица, b – вектор-столбец.
Мы – программисты, поэтому нумерация везде с нуля.
Задача – найти какой-нибудь x, а лучше всё множество x-ов.

Все a_{ij}, b_i – элементы поля (например, $\mathbb{R}, \mathbb{C}, \mathbb{Z}/p\mathbb{Z}$), т.е. нам доступны операции +, -, *, \.

4.1. Гаусс для квадратных невырожденных матриц.

В данной части мы увидим классического Гаусса. Такого, как его обычно описывают.
Наша цель – превратить матрицу A в треугольную или диагональную.
Наш инструмент – можно менять уравнения местами, вычитать уравнения друг из друга.
Для удобства реализации сразу начнём хранить b_i в ячейке a_{in}
План: для каждого i в ячейку a_{ii} поставить ненулевой элемент, и, пользуясь им и вычитанием строк, занулить все другие ячейки в i-м столбце.

```
1  // этот код работает только для m = n, det A ≠ 0
2  // тем не менее, чтобы не путаться в размерностях, мы в разных местах пишем и m, и n
3  vector<vector<F>> a(m, vector<T>(n + 1)); // b хранится последним столбцом a
4  for (int i = 0; i < n; i++):
5    int j = i;
6    while (isEqual(a[j][i], 0)) // isEqual для ℝ обязана использовать ε
7      j++; // ненулевой элемент точно найдётся из невырожденности
8    swap(a[i], a[j]); // меняем строки местами, кстати, за O(1)
9    for (int j = 0; j < n; j++) // перебираем все строки
10     if (j != i) // если хотим получить диагональную
11     if (j > i) // если хотим получить треугольную
12       if (!isEqual(a[j][i], 0)): // хотим в [i,j] поставить 0, вычтем i-ю строку
13         F coef = a[j][i] / a[i][i];
14         for (int k = i; k <= n; k++) // самая долгая часть
15           a[j][k] -= a[i][k] * coef;
```

Строка 14 – единственная часть, работающая за $\mathcal{O}(n^3)$, поэтому для производительности важно, что цикл начинается с i, а не с 0 (так можно, так как слева от i точно нули).

Если результат Гаусса – диагональная матрица, то $x_i = b_i/a_{ii}$.
Из треугольной матрицы x-ы нужно восстанавливать в порядке $i = n{-}1 \dots 0$:
$x_i = (b_i - \sum_{j=i+1..n-1} x_j \cdot a_{ij})/a_{ii}$. Время восстановления $\mathcal{O}(n^2)$.

Замечание 4.1.1. Вычисление определителя. При `swap` строк $\det A$ меняет знак, при вычитании строк $\det A$ не меняется $\Rightarrow$ за то же время мы умеем вычислять $\det A$.

Пример 4.1.2. Работа Гаусса.

$$\left[\begin{array}{ccc|c} \boxed{1} & 1 & 2 & 5 \\ 3 & 3 & 8 & 5 \\ 2 & 5 & 1 & 5 \end{array}\right] \xrightarrow{\texttt{i=0}} \left[\begin{array}{ccc|c} 1 & 1 & 2 & 5 \\ 0 & 0 & 2 & -10 \\ 0 & 3 & -3 & -5 \end{array}\right] \xrightarrow{\texttt{swap}} \left[\begin{array}{ccc|c} 1 & 1 & 2 & 5 \\ 0 & \boxed{3} & -3 & -5 \\ 0 & 0 & 2 & -10 \end{array}\right] \xrightarrow{\texttt{i=1}} \left[\begin{array}{ccc|c} 1 & 0 & 3 & 6\frac{2}{3} \\ 0 & 3 & -3 & -5 \\ 0 & 0 & \boxed{2} & -10 \end{array}\right] \xrightarrow{\texttt{i=2}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 21\frac{2}{3} \\ 0 & 3 & 0 & -20 \\ 0 & 0 & 2 & -10 \end{array}\right]$$

Итого: $x = [21\frac{2}{3}, -6\frac{2}{3}, -5]$

Оценим время работы в худшем случае (всегда заходим в `if` в `12`-й строке):

Для превращения в треугольную $\sum_i i^2 \approx n^3/3$
Для превращения в диагональную $\sum_i ni \approx n^3/2$

$\Rightarrow$ если важна скорость, приводите к $\triangle$-ой. Когда на первом месте удобство, к диагональной.

4.2. Гаусс в общем случае

Если мы подойдём в вопросу чисто математически, придётся ввести трапецевидные и ступенчатые матрицы. Возможно, нам захочется менять столбцы (переменные) местами.

Мы хотим сразу удобный универсальный код. Поэтому задачу сформулируем так:
По одному добавляются пары $\langle a_i, b_i \rangle \in \langle \mathbb{F}^n, \mathbb{F} \rangle$, нам нужно поддерживать базис пространства $linear\{a_1, a_2, \ldots, a_m\}$ и поддерживать множество решений системы $Ax = b$.

```
vector<vector<F>> A; // текущий базис и прикреплённые b-шки
vector<int> col; // для каждого базисного вектора храним номер столбца, который он обнуляет
bool add(vector<F> a) { // a[0],...,a[n-1],b
  for (size_t i = 0; i < A.size(); i++)
    if (!isEqual(a[col[i]], 0)):
      F coef = a[col[i]] / A[i][col[i]];
      for (size_t k = 0; k < a.size(); k++)
        a[k] -= A[i][k] * coef;

  size_t i = 0;
  while (i < a.size() && isEqual(a[i], 0))
    i++;
  if (i == a.size()) return 1; // уравнение - линейная комбинация предыдущих
  if (i == a.size() - 1) return 0; // выразили из данных уравнений «0+..+0 = 1»
  A.push_back(a), col.push_back(i); // добавили в базис новый вектор
  return 1; // система всё ещё разрешима
}
```

Время работы добавления m векторов, если $dim\, linear\{a_1, \ldots, a_m\} = k$, работает за $\mathcal{O}(mnk)$.
Восстановим решение. Свободные переменные – ровно те, что не вошли в `col`.

```
vector<F> getX(): // O(n + k^2), что для маленьких k гораздо быстрее обычного O(n^2)
  vector<F> x(n, 0); // пусть свободные переменные равны нулю
  for (int i = A.size() - 1; i >= 0; i--):
    x[col[i]] = A[i][n]; // A[i].size() == n + 1
    for (size_t j = i + 1; j < A.size(); j++)
      x[col[i]] -= A[i][col[j]] * x[col[j]];
  return x; // нашли какое-то одно решение
```

Теперь запомним начальное `s = getX()` и переберём те столбцы j, которыя являются свободными переменными. Для каждого j начнём восстановление ответа с `x[j] = 1`, и после строки

4 учтём слагаемое `A[i][j]*x[j]`. Результат для j-й переменной обозначим v_j.
Теперь у нас есть всё пространство решений: $s + linear\{v_1 - s, v_2 - s, \dots, v_{n-k} - s\}$.
Время: $\mathcal{O}((n-k)(n+k^2))$. Где $(n-k)$ – размерность пространства решений.

4.3. Гаусс над $\mathbb{F}_2$

Самая долгая часть Гаусса – вычесть из одной строки другую, умноженную на число.
В $\mathbb{F}_2$ вычетание – $\oplus$, а умножение &. Любимый нами `bitset` обе операции сделает за $\mathcal{O}(n/w)$.
Полученное ускорение применимо к обоим версиям Гаусса, описанным выше.

4.4. Погрешность

При вычислениях в $\mathbb{Z}/p\mathbb{Z}$, естественно, отсутствует. При вычислениях в $\mathbb{R}$ она зашкаливает.
Рассмотрим *матрицу Гильберта* G: $g_{ij} = \frac{1}{i+j}; i, j \in [1, n]$.
Попробуем решить руками уравнение $Gx = 0$: $\det G \neq 0 \Rightarrow \exists!\, x = \{0, \dots, 0\}$.
Теперь применим Гаусса, реализованного в типе `double` при $\varepsilon = 10^{-12}$.
Уже при $n = 11$ в процессе выбора ненулевого элемента мы не сможем отличить 0 от не нуля.
Окей! `double` (8 байт) $\to$ `long double` (10 байт), и $\varepsilon = 10^{-15}$. Та же проблема при $n = 17$.

• **Решения проблемы:**

Обычно, чтобы хоть чуть-чуть уменьшить погрешность выбирают, не любой ненулевой элемент в столбце i, а max по модулю элемент в подматрице $[i, n] \times [i, n]$ (*эвристика* max *элемента*).
В инкрементальном способе (добавлять строки по одной) эту оптимизацию не применить.

Во многих языках реализованы длинные вещественные числа, например, Java: `BigDecimal`.
Но всё равно возникает вопрос, какую точность выбрать? Содержательный ответ можно будет извлечь из курса «вычислительные методы», а простой звучит так:

1. Запустите Гаусса два раза с k и $2k$ значащими знаками.
2. Если ответы недопустимо сильно отличаются, k слишком мало, его нужно увеличить.

«Детский» способ. Пусть известно ограничение по времени (например, ровно 1) секунда, выберем max возможную точность, чтобы уложиться в ограничение.

На некотором классе матриц меньшую погрешность даёт *метод простой итерации*.

4.5. Метод итераций

Пусть нам нужно решить систему $x = Ax + b$. При этом $||A|| < 1$ (вы ведь помните про нормы?).

Решение: начнём с $x_0 =$ `random`, будем пересчитывать $x_{i+1} = Ax_i + b$.

Сделаем сколько-то шагов, последний выдадим, как ответ. Сложность $\mathcal{O}(n^2 t)$, t – число шагов.
На самом деле даже меньше: $\mathcal{O}(Et)$ шагов, где E – число ненулевых ячеек матрицы.

Если бы система имела более простой вид $x = Ax$, мы могли бы вычислять быстрее:
$A \to A^2 \to A^4 \to \dots \to A^{2^k} \to (A^{2^k})x$, итого сложность $\mathcal{O}(n^3 \log t)$.
Попробуем такой же фокус провернуть с исходной системой $x = Ax + b$.

$x_0 \to Ax_0 + b \to A^2x_0 + Ab + b \to \dots \to A^k x_0 + \underbrace{A^{k-1}b + \dots + Ab + b}_{\text{Обозначим } S_i, k=2^i}$.

Заметим: $S_k = A^{k/2}S_{k-1} + S_{k-1} = (A^{k/2} + E)S_{k-1}$.

База: $T_0 = A, S_0 = b$. Переход: $\begin{cases} S_{i+1} = (T_i + E)S_i \\ T_{i+1} = T_i^2 \end{cases}$

$||A|| = \sup_{|x|=1} |Ax|$, $||A^{2^k}|| \leqslant ||A||^{2^k} \xrightarrow{k\to\infty} 0 \Rightarrow$ пренебрежём слагаемым $A^{2^k}x_0$ (или возьмём $x_0 = 0$).

Замечание 4.5.1. Есть два способа запустить t итераций: линейная итерация $x \to Ax$, работает за $\mathcal{O}(Et)$, и итерация с удвоением $A \to A^2$, работает за $\mathcal{O}(n^3 \log t)$. Если в задаче имеется быстрая сходимость и матрица A разреженная, линейная итерация можеть работать лучше.

4.6. Вычисление обратной матрицы

Задача: дана A над полем, найти $X\colon AX = E$.

Из $\det A \cdot \det X = 1$ следует, что A невырождена $\Rightarrow$ решение единственно.
Каждый столбец матрицы X задаёт систему уравнений $\Rightarrow$ нахождение X за $\mathcal{O}(n^4)$ очевидно.
Чтобы получить время $\mathcal{O}(n^3)$, заметим, что у систем матрица A общая, различны лишь столбцы $b \Rightarrow$ системы можно решать одновременно.
Также, как мы записывали b_i в a_{in}, если есть сразу $b_{i0}, b_{i1}, \dots, b_{ik}$, то $\forall j$ запишем b_{ij} в $a_{i\,n+j}$.
Далее будем оперировать со строками длины $n+k$. Время работы $\mathcal{O}((n+k)n^2) = \mathcal{O}(n^3)$.

Короткое изложение: записали AE как матрицу из $2n$ столбцов, привели Гауссом A к диагональному, а затем даже единичному виду $\Rightarrow$ на месте E у нас как раз A^{-1}.

Следствие 4.6.1. Над $\mathbb{F}_2$ обратную матрицу мы научились искать за $\mathcal{O}(n^3/w)$.

4.7. Гаусс для евклидова кольца

Напомним, *евклидово кольцо* – область целостности с делением с остатком
(есть +, -, * и / с остатком). Например, $\mathbb{Z}$. Или $\mathbb{R}[x]$ – многочлены, $\mathbb{Z}[i]$ – Гауссовы числа.

Сейчас у нас получится чуть изменить обычного Гаусса, приводящего матрицу A к треугольной. А вот к диагональному виду, увы, привести не получится.

Основная операция в Гауссе – имея столбец i, строки i и j при $a_{ii} \neq 0$ занулить a_{ji}.
Раньше мы вычитали из строки a_j строку a_i, умноженную на $\frac{a_{ji}}{a_{ii}}$. Теперь у нас нет деления...
Зато у нас есть деление с остатком $\Rightarrow$ есть алгоритм Евклида.

Давайте на элементах a_{ji} и a_{ii} запустим Евклида. Только по ходу Евклида вычитать будем строки целиком. Результат: $\forall k < i$ все a_{ik}, a_{jk} как были нулями, так и остались, а один из a_{ji} и a_{ii} занулился. *Пример:*

$$\left[\begin{array}{cccc|c} 2 & 8 & 2 & 0 & 1 \\ 0 & \boxed{7} & 5 & 4 & 1 \\ 0 & \boxed{3} & 6 & 3 & 1 \end{array}\right] \xrightarrow{7-3\cdot 2} \left[\begin{array}{cccc|c} 2 & 8 & 2 & 0 & 1 \\ 0 & \boxed{1} & -7 & -2 & -1 \\ 0 & \boxed{3} & 6 & 3 & 1 \end{array}\right] \xrightarrow{3-1\cdot 3} \left[\begin{array}{cccc|c} 2 & 8 & 2 & 0 & 1 \\ 0 & 1 & -7 & -2 & -1 \\ 0 & 0 & 27 & 9 & 4 \end{array}\right]$$

Как этим можно пользоваться?

1. Для подсчёта определителя квадратной матрицы.
2. Для решения системы $Ax = b$, если нет свободных переменных (например, $\det A \neq 0$)

Если есть свободные переменные, то во время восстановления ответа по треугольной матрице в формуле $x_i = (b_i - \sum_{j=i+1..n-1} x_j \cdot a_{ij})/a_{ii}$ у нас может «не поделиться».
В случае $\det A \neq 0$ это значило бы «нет решений», а тут это значит «мы неправильно задали значения свободным переменным».

Замечание 4.7.1. Если стоит задача «проверки невырожденности матрицы над $\mathbb{Z}$», то разумно, чтобы избежать длинных чисел, вычисления проводить не над $\mathbb{Z}$, а по большому простому модулю. При подсчёте определителя матрицы над $\mathbb{Z}$ для борьбы с длинными числами можно использовать приём из разд. 3.16.1.

4.8. Разложение вектора в базисе

Вернёмся в мир полей и безопасного деления.
Если нам дают базис пространства $\{v_1, \dots, v_n\}$ и вектор p, просят разложить p в базисе v, проще всего решить систему уравнений $Ax = p$, где v_i – столбцы матрицы A. Время $\mathcal{O}(n^3)$.

Если нам дают сразу много векторов $p_1, p_2, \dots p_k$, то мы дописываем их к A: $\left[A \,|\, p_1 p_2 \dots p_k\right]$, как делали в разд. 4.6, и получаем итоговое время $\mathcal{O}(n^2(n{+}k) + nk) = \mathcal{O}(n^3 + n^2 k)$.

Теперь будем решать online задачу – нам нужно сделать некий предподсчёт от базиса, а вектора p_i будут выдавать по одному. Раскладывать каждый p_i хочется за $\mathcal{O}(n^2)$.

Заметим, что в Гауссе разд. 4.2 мы как раз по сути разложили вектор... только не на исходные вектора, а на текущие строки матрицы. Ок, давайте для каждой строки матрицы хранить коэффициенты c_{ij}: $a_i = \sum_j c_{ij} v_j$, где a_i – строки матрицы, а v_j – исходные вектора. Тогда для нового вектора $a_k = \sum_{i=0..k-1} \alpha_i a_i = \sum_j v_j (\sum_i \alpha_i c_{ij})$. Новые коэффициенты $c_{kj} = \sum_i \alpha_i c_{ij}$. Итого мы за $\mathcal{O}(n^2)$ нашли коэффициенты строки, которую собираемся добавить в базис.

4.8.1. Ортогонализация Грама-Шмидта

Другой способ сделать базис удобным для пользования – привести его к ортонормированному виду. Далее следует описание Ортогонализации Грама-Шмидта, знакомое вам с алгебры:

```
for i=0..k-1
  for j=0..i-1
    v[i] -= v[j] * scalarProduct(v[i], v[j])
  v[i] /= sqrt(scalarProduct(v[i], v[i]))
```

Получить координаты вектора p в новом милом базисе проще простого: $x_i = \langle p, v_i \rangle$.
Время разложения вектора в базисе – $\mathcal{O}(n^2)$ сложений и умножений.

Опять же, если мы хотим координаты в исходном базисе, то для каждого v_i нам нужно будет таскать вектор коэффициентов: $v_i = \sum_j \alpha_j v_j^*$ (выражение через исходный базис), и вычитая v_i, вычитать и вектора коэффициентов. Время разложения p в исходном базисе – тоже $\mathcal{O}(n^2)$.

4.9. Вероятностные задачи

Рассмотрим орграф, на рёбрах которого написаны вероятности.
Для каждой вершины верно, что сумма исходящих вероятностей равна 1.
Если бы мы хотели с некоторой вероятностью оставаться в вершине, добавили бы петлю.

Что нас может интересовать?

1. Начав в вершине v, в какой вершине с какой вероятностей мы находимся при $t = \infty$?
2. Какова вероятность, что, начав в v мы дойдём до вершины A раньше, чем до вершины B (A – спасти принцессу, B – свалиться в болото)?
3. Какое матожидание числа шагов в пути из вершины v в вершину A?
4. Какое условное матожидание числа шагов в пути из вершины v в вершину A,
   если попадание в B – смерть?

• Орграф ацикличен $\Rightarrow$ динамика

Если исходный орграф ацикличен + в нём могут быть петли, все задачи решаются динамикой.
Пример для матожидания без петли: $E[v] = 1 + \sum p[v \to x] \cdot E[x]$
Пример для матожидания с петлёй вероятности q:
$E[v] = 1 + (1-q)(\sum p[v \to x] \cdot E[x]) + q \cdot E[v] \Rightarrow E[v] = \frac{1}{1-q} + \sum p[v \to x] \cdot E[x]$.

• Произвольный орграф $\Rightarrow$ итерации или Гаусс

Рассмотрим вторую задачу. Тогда $p[A] = 1, p[B] = 0$, а на каждую другую вершину есть линейное уравнение $p[v] = \sum p[v \to x] \cdot p[x]$. Давайте решать!

Гаусс работает за $\mathcal{O}(V^3)$, обладает скверной погрешностью.
Метод итераций в данном случае будет работать так:

1. Изначально все кроме $p[A]$ нули.
2. Затем мы все $p[v]$ пересчитываем по формуле: $p[v] = \sum_x p[v \to x] \cdot p[x]$.

Можно пересчитывать возведением матрицы в степень:

$$\boxed{k \text{ фаз за } \mathcal{O}(V^3 \log k)}, \text{ а можно } \boxed{\text{в лоб за } \mathcal{O}(E \cdot k)}$$

Замечание 4.9.1. Полезно ознакомиться с практиками, домашними заданиями, разборами.

Замечание 4.9.2. Пусть получившуюся систему уравнений задаёт матрица A.
Если бы было $||A|| < 1$, мы бы уже сейчас сказали про сходимость. Но $||A|| \leqslant 1$, поэтому анализ сходимости метода итераций ждёт вас в курсе *численных методов*.

4.10. (*) СЛАУ над $\mathbb{Z}$ и $\mathbb{Z}/m\mathbb{Z}$

4.10.1. (*) СЛАУ над $\mathbb{Z}$

Решение #1. Оценим величину ответ: пусть $|ans| < x$. Решим ту же систему по простым модулям $p_1, p_2, \ldots p_k$: $\prod p_i \geqslant x$. Через КТО восстановим ответ. Время работы $\mathcal{O}(n^3 k)$. Минусы: нужно оценить ответ. Модификация: если известно, что ответ $\exists$, можно находить k итеративным удвоением: $k = 1 \to 2 \to 4 \to \ldots$.

Решение #2. Пытаемся всё сделать одним Гауссом. Запустить Гаусса проблем нет. Проблема — выбрать свободные переменные так, чтобы всё поделилось: $\forall i\ b_i - \sum_{j>i} a_{ij}x_j \equiv 0 \bmod a_{ii}$.

Как теперь $x_i = \frac{b_i - \sum_{j>i} a_{ij}x_j}{a_{ii}}$ подставлять в следующие линейные уравнения и остаться в целых числах? Можно вместо деления наоборот все уравнения умножить на a_{ii}. Такой процесс вызовет большой, но предсказуемый рост коэффициентов: первое уравнение умножится на $a_{22}a_{33}\ldots a_{nn}$.

4.10.2. (*) СЛАУ по модулю

Сперва представим себе полный ад: много линейных уравнений и каждое по своему модулю m_i. КТО даёт нам возможность для упрощений: факторизуем все m_i, получим уравнения $\bmod p^k$. Сгруппируем уравнения по p. Если есть два уравнения $\langle a_1, x\rangle = b_1 \bmod p^i$ и $\langle a_2, x\rangle = b_2 \bmod p^j$, при $i > j$ второе можно домножить на p^{i-j}: $\langle a_2 p^{i-j}, x\rangle = b_2 p^{i-j} \bmod p^i \Rightarrow \forall p$ все степени i равны.

Решаем для каждого p свою систему. В конце КТО даёт ответ для исходной системы.

4.10.3. (*) СЛАУ над $\mathbb{Z}/p^k\mathbb{Z}$

Можно действовать похоже на второе решение для $\mathbb{Z}$.
Разница в том, что все новые уравнения «чтобы поделились» будут уже $\bmod p^{k-1} \Rightarrow$ процесс нужно повторить $\leqslant k$ раз. Время работы $\mathcal{O}(n^3 k)$.

Лекция #5: Быстрое преобразование Фурье

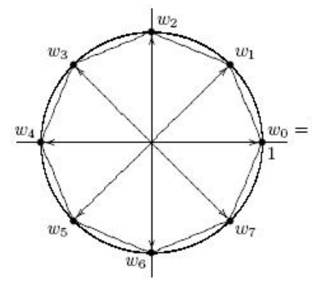
5 декабря 2022

Перед тем, как начать говорить «Фурье» то, «Фурье» сё, нужно сразу заметить:
Есть непрерывное преобразование Фурье. С ним вы должны столкнуться на теорвере.
Есть тригонометрический ряд Фурье. И есть общий ряд Фурье в гильбертовом пространстве, который появляется в начале курса функционального анализа.
Мы же с вами будем заниматься исключительно дискретным преобразованием Фурье.
Коротко DFT (Discrete Fourier transform). FFT – по сути то же, первая буква означает «fast».

Задача: даны два многочлена A, B суммарной длины $\leqslant n$, переменожить их за $\mathcal{O}(n \log n)$.

Длина многочлена – $\gamma(A) = (\deg A) + 1$. Вводим её, чтобы везде не писать «-1».
Если даны n точек (x_i, y_i), все x_i различны, $\exists!$ интерполяционный многочлен длины n, построенный по этим точкам (из алгебры). Ещё заметим: $\gamma(AB) = \gamma(A) + \gamma(B) - 1$. Наш план:

1. Подобрать удачные k и точки $w_0, w_1, \dots, w_{k-1}$: $k \geqslant \gamma(A) + \gamma(B) - 1 = n$.
2. Посчитать значения A и B в w_i.
3. $AB(x_i) = A(x_i)B(x_i)$. Эта часть самая простая, работает за $\mathcal{O}(n)$.
4. Интерполировать AB длины k по полученным парам $\langle w_i, AB(w_i)\rangle$.



Вспомним комплексные числа:

$e^{i\alpha}e^{i\beta} = e^{i(\alpha+\beta)}$ $e^{i\varphi} = (\cos\varphi, \sin\varphi)$, $\overline{(a,b)} = (a, -b) \Rightarrow \overline{e^{i\varphi}} = e^{i(-\varphi)}$
Извлечение корня k-й степени: $\sqrt[k]{z} = \sqrt[k]{e^{i\varphi}} = e^{i\varphi/k}$
Если взять все корни из 1 степени 2^t, возвести в квадрат,
получатся ровно все корни степени 2^{t-1}. Корни из 1 степени k: $e^{ij/k}$.

5.1. Прелюдия к FFT

Возьмём $\min N = 2^t \geqslant n$ и $w_j = e^{ij/N}$. Тут мы предполагаем, что $A, B \in \mathbb{C}[x]$ или $A, B \in \mathbb{R}[x]$.
Пусть есть многочлены $A(x) = \sum a_i x^i$ и $B(x) = \sum b_i x^i$. Ищем $C(x) = A(x)B(x)$.
Обозначим их значения в точках $w_0, w_1, \dots, w_{k-1}$: $A(w_i) = fa_i, B(w_i) = fb_i, C(w_i) = fc_i$.
Схема быстрого умножения многочленов:

$$a_i, b_i \xrightarrow{\mathcal{O}(n\log n)} fa_i, fb_i \xrightarrow{\mathcal{O}(n)} fc_i = fa_i fb_i \xrightarrow{\mathcal{O}(n\log n)} c_i$$

5.2. Собственно идея FFT

$A(x) = \sum a_i x^i = (a_0 + x^2 a_2 + x^4 a_4 + \dots) + x(a_1 x + a_3 x^3 + a_5 x^5 + \dots) = P(x^2) + xQ(x^2)$
Т.е. обозначили все чётные коэффициенты A многочленом P, а нечётные соответственно Q.

$\gamma(A) = n$, все $w_j^2 = w_{n/2+j}^2 \Rightarrow$ многочлены P и Q нужно считать не в n, а в $\frac{n}{2}$ точках.

```
def FFT(a):
  n = len(a)
  if n == 1: return a # посчитать значение A(x) = a[0] в точке 1
  a ---> p, q # разбили коэффициенты на чётные и нечётные
  p, q = FFT(p), FFT(q)
  w = exp(2pi*i/n) # корень из единицы n-й степени
  for i=0..n-1: a[i] = p[i%(n/2)] + w^i*q[i%(n/2)]
  return a
```

Время работы $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$.

5.3. Крутая реализация FFT

Чтобы преобразование работало быстро, нужно заранее предподсчитать все $w_j = e^{2\pi ij/N}$.
Заметим, что p и q можно хранить прямо в массиве a.
Тогда получается, что на прямом ходу рекурсии мы просто переставляем местами элементы a, и только на обратном делаем какие-то полезные действия.
Число a_i перейдёт на позицию $a_{rev(i)}$, где $rev(i)$ – перевёрнутая битовая запись i.
Кстати, $rev(i)$ мы уже умеем считать динамикой для всех i.
При реализации на `C++` можно использовать комплексные числа из STL: `complex<double>`.

```
const int K = 20, N = 1 << K;
complex<double> root[N];
int rev[N];
void init():
  for (int j = 0; j < N; j++):
    root[j] = exp(2π·i·j/N); // cos(2πj/N), sin(2πj/N)
    rev[j] = (rev[j >> 1] >> 1) + ((j & 1) << (K - 1));
```

Теперь, корни из единицы степени k хранятся в `root[j*N/k]`, $j \in [0, k)$. Две проблемы:
1. Доступ к памяти при этом не последовательный, проблемы с кешом.
2. Мы $2N$ раз вычисляли тригонометрические функции.
$\Rightarrow$ можно лучше, вычисления корней #2:

```
for (int k = 1; k < N; k *= 2):
  num tmp = exp(π/k);
  root[k] = {1, 0}; // в root[k..2k) хранятся первые k корней степени 2k
  for (int i = 1; i < k; i++)
    root[k + i] = (i & 1) ? root[(k + i) >> 1] * tmp : root[(k + i) >> 1];
```

Теперь код собственно преобразования Фурье может выглядеть так (используем `root` #2):

Алгоритм 5.3.1. *Эффективная реализация FFT*

```
void FFT(a, fa): // a -> fa
  for (int i = 0; i < N; i++)
    fa[rev[i]] = a[i]; // можно иначе, но давайте считать массив «a» readonly
  for (int k = 1; k < N; k *= 2) // уже посчитаны FFT от кусков длины k, база: k=1
    for (int i = 0; i < N; i += 2 * k) // [i..i+k) [i+k..i+2k) --> [i..i+2k)
      for (int j = 0; j < k; j++): // оптимально написанный стандартный цикл FFT
        num tmp = root[k + j] * fa[i + j + k]; // вторая версия root[]
        fa[i + j + k] = fa[i + j] - tmp; // exp(2πi(j+n/2)/n) = -exp(2πij/n)
        fa[i + j] = fa[i + j] + tmp;
```

5.4. Обратное преобразование

Теперь имея при $w = e^{2\pi i/n}$:
$fa_0 = a_0 + a_1 + a_2 + a_3 + \dots$
$fa_1 = a_0 + a_1 w + a_2 w^2 + a_3 w^3 + \dots$
$fa_2 = a_0 + a_1 w^2 + a_2 w^4 + a_3 w^3 + \dots$
$\dots$
Нам нужно научиться восстанавливать коэффициенты $a_0, a_1, a_2, \dots$, имея только fa_i.

Заметим, что $\forall j \neq 0 \sum\limits_{k=0}^{n-1} w^{jk} = 0$ (геометрическая прогрессия). А при $j = 0$ получаем $\sum\limits_{k=0}^{n-1} w^{jk} = n$.
$\Rightarrow fa_0 + fa_1 + fa_2 + \dots = a_0 n + a_1 \sum_k w^k + a_2 \sum_k w^{2k} + \dots = a_0 n$
Аналогично $fa_0 + fa_1 w^{-1} + fa_2 w^{-2} + \dots = \sum_k a_0 w^{-k} + a_1 n + a_2 \sum_k w^k + \dots = a_1 n$
И в общем случае $\sum_k fa_k w^{-jk} = a_j n$.
Заметим, что это ровно значение многочлена с коэффициентами fa_k в точке w^{-j}.
Осталось заметить, что множества чисел $\{w_j \mid j = 0..n{-}1\}$ и $\{w_{-j} \mid j = 0..n{-}1\}$ совпадают $\Rightarrow$

```
void FFT_inverse(fa, a): // fa → a
  FFT(fa, a)
  reverse(a + 1, a + N) // w^j ↔ w^{-j}
  for (int i = 0; i < N; i++) a[i] /= N;
```

Другой способ. Возьмём код 5.3.1, заметим, что строки 7-8-9 обратимы:

```
tmp = (fa[i+j] + fa[i+j+k]) / 2
fa[i+j] -= tmp, fa[i+j+k] = tmp / root[k+j];
```

$\Rightarrow$ запустим циклы 4-5-6 в обратном порядке, обращая каждый шаг прямого FFT.

5.5. Два в одном

Часто коэффициенты многочленов – вещественные числа.
Если у нас есть многочлены $A(x), B(x) \in \mathbb{R}[x]$, возьмём числа $c_j = a_j + ib_j$ и посчитаем $fc = FFT(c)$. Тогда по fc за $\mathcal{O}(n)$ можно легко восстановить fa и fb.

Для этого вспомним про сопряжения комплексных чисел:
$\overline{x+iy} = x - iy, \overline{a \cdot b} = \overline{a} \cdot \overline{b}, w^{n-j} = w^{-j} = \overline{w^j} \Rightarrow \overline{fc_{n-j}} = \overline{C(w^{n-j})} = \overline{C}(w^j) \Rightarrow$
$fc_j + \overline{fc_{n-j}} = 2 \cdot A(w^j) = 2 \cdot fa_j$. Аналогично $fc_j - \overline{fc_{n-j}} = 2i \cdot B(w^j) = 2i \cdot fb_j$.

Теперь, например, для умножения двух $\mathbb{R}[x]$ можно использовать не 3 вызова FFT, а 2.

5.6. Умножение чисел, оценка погрешности

Общая схема умножения чисел:
цифра – коэффициент многочлена ($x = 10$); умножим многочлены; сделаем переносы.

Число длины n в системе счисления 10 можно за $\mathcal{O}(n)$ перевести в систему счисления 10^k. Тогда многочлены будут длины n/k, умножение многочленов работать за $\frac{n}{k} \log \frac{n}{k}$ (убывает от k).
Возникает вопрос, какое максимальное k можно использовать?
Коэффициенты многочлена-произведения будут целыми числами до $(10^k)^2 \frac{n}{k}$.
Чтобы в типе `double` целое число хранилось с погрешностью меньше 0.5 (тогда его можно правильно округлить к целому), оно должно быть не более 10^{15}.
Получаем при $n \leqslant 10^6$, что $(10^k)^2 10^6/k \leqslant 10^{15} \Rightarrow k \leqslant 4$.
Аналогично для типа `long double` имеем $(10^k)^2 10^6/k \leqslant 10^{18} \Rightarrow k \leqslant 6$.
Это оценка сверху, предполагающая, что само FFT погрешность не накапливает... на самом деле эта оценка очень близка к точной.

5.7. Применение. Циклические сдвиги.

Часто вылезает не «умножение многочленов», а подсчёт «скалярных произведений массива a и сдвигов массива b». Это ровно коэффициенты $A(x) \cdot B(x)$, где $A(x) = \sum a_i x^i$, $B(x) = b_i x^{n-i}$.

Лекция #6: Длинная арифметика

5 декабря 2022

6.1. Простейшие операции

Самое главное — научиться операциям над целыми беззнаковыми числами.
Целые со знаком — то же + дополнительно хранить знак.
Вещественные — то же + хранить экспоненту: $12.345 = 12345e{-}3$, мы храним 12345 и -3.

Удобно хранить число в «массиве цифр», младшие цифры в младших ячейках.
Во примерах ниже мы выбираем систему счисления $\mathtt{BASE} = 10^k$, $k \to \max$: нет переполнений.
Пусть есть длинное число a. При оценки времени работы будем использовать обозначения: $|a| = n$ – битовая длина числа и $\frac{n}{k}$ – длина числа, записанного в системе 10^k. Помните, $\max k \approx 9$.

Если мы ленивы и уверены, что в процессе вычислений не появятся числа длиннее N, наш выбор – `int[N];`, иначе обычно используют `vector<int>` и следят за длиной числа.

Примеры простейших операций:

```
1  const int N = 100, BASE = 1e9, BASE_LEN = 9;
2  vhoid add(int *a, int *b) { // сложение за O(n/k)
3    for (int i = 0; i + 1 < N; i++) // +1, чтобы точно не было range check error
4      if ((a[i] += b[i]) >= BASE)
5        a[i] -= BASE, a[i + 1]++;
6  }
7  void subtract(int *a, int *b) { // вычитание за O(n/k), a ⩾ b
8    for (int i = 0; i + 1 < N; i++) // +1, чтобы точно не было range check error
9      if ((a[i] -= b[i]) < 0)
10       a[i] += BASE, a[i + 1]--;
11 }
12 int divide(int *a, int k) { // деление на короткое за O(n/k), делим со старших разрядов
13   long long carry = 0; // перенос с более старшего разряда, он же остаток
14   for (int i = N - 1; i >= 0; i--):
15     carry = carry * BASE + a[i]; // максимальное значение carry < BASE^2
16     a[i] = carry / k, carry %= k;
17   return carry; // как раз остаток
18 }
19 int mul_slow(int *a, int *b, int *c) { // умножение за (n/k)^2
20   fill(c, c + N, 0);
21   for (int i = 0; i < N; i++)
22     for (int j = 0; i + j < N; j++)
23       c[i + j] += a[i] * b[j]; // здесь почти наверняка произойдёт переполнение
24   for (int i = 0; i + 1 < N; i++) // сначала умножаем, затем делаем переносы
25     c[i + 1] += c[i] / BASE, c[i] %= BASE;
26 }
27 void out(int *a) { // вывод числа за O(n/k)
28   int i = N - 1;
29   while (i && !a[i]) i--;
30   printf("%d", a[i--]);
31   while (i >= 0) printf("%0*d", BASE_LEN, a[i--]); // воспользовались таки BASE_LEN!
32 }
```

Чтобы в строке `19` не было переполнения, нужно выбирать `BASE` так, что $\mathtt{BASE}^2 \cdot \mathtt{N}$ помещалось в тип данных. Например, хорошо сочетаются $\mathtt{BASE} = 10^8, \mathtt{N} = 10^3$, тип — `uint64_t`.

6.2. (-) Бинарная арифметика

Пусть у нас реализованы простейшие процедуры: «+, -, mul2, div2, less, equal, isZero». Давайте выразим через них «*, \, gcd». Обозначим $|a| = n, |b| = m$.

Умножение будет полностью изоморфно бинарному возведению в степень.

```
num mul(num a, num b):
  if (isZero(b)) return 0; // если храним число, как vector, то isZero за O(1)
  num res = mul(mul(mul2(a), div2(b));
  if (mod2(b) == 1) add(res, a); // функция mod2 всегда за O(1)
  return res;
```

Глубина рекурсии равна m. В процессе появляются числа не более $(n+m)$ бит длины $\Rightarrow$ каждая операция выполняется за $\mathcal{O}(\frac{n+m}{k}) \Rightarrow$ суммарное время работы $\mathcal{O}((n+m)\frac{m}{k})$. Если большее умножать на меньшее, то $\mathcal{O}(\max(n,m)\min(n,m)/k)$.

Деление в чём-то похоже... деля a на b, мы будем пытаться вычесть из a числа $b, 2b, 4b, \ldots$.

```
pair<num, num> div(num a, num b): // найдём для удобства и частное, и остаток
  num c = 1, res = 0;
  while (b < a) // (n-m) раз
    mul2(b), mul2(c);
  while (!isZero(c)): // Этот цикл сделает ≈ n-m итераций
    if (a >= b) // O(n/k), так как длины a и b убывают от n до 1
      sub(a, b), add(res, c); // O(n/k)
    div2(b), div2(c); // O(n/k)
  return {res, a};
```

Шагов главного цикла $n-m$. Все операции за $\mathcal{O}(\frac{n}{k}) \Rightarrow$ суммарное время $\mathcal{O}((n-m)\frac{n}{k})$.

Наибольший общий делитель сделаем самым простым Евклидом «с вычитанием». Добавим только одну оптимизацию: если числа чётные, надо сразу их делить на два...

```
num gcd(num a, num b):
  int pow2 = 0;
  while (mod2(a) == 0 && mod2(b) == 0)
    div2(a), div2(b), pow2++;
  while (!isZero(a) && !isZero(b)):
    while (mod2(a) == 0) div2(a);
    while (mod2(b) == 0) div2(b);
    if (a < b) swap(a, b);
    a = sub(a, b); // одно из чисел станет чётным
  if (isZero(a)) swap(a, b);
  while (pow2--) mul2(a);
  return a;
```

Шагов главного цикла не больше $n+m$. Все операции выполняются за $\max(n,m)/k$. Отсюда суммарное время работы: $\mathcal{O}(\max(n,m)^2/k)$.

6.3. Деление многочленов за $\mathcal{O}(n\log^2 n)$

Коэффициенты многочлена $A(x)$: $A[0]$ – младший, $A[\deg A]$ – старший. $\gamma(A) = \deg A - 1$.

Задача: даны $A(x), B(x) \in \mathbb{R}[x]$, найти $Q(x), R(x)$: $\deg R < \deg B \wedge A(x) = B(x)Q(x) + R(x)$.

Сперва простейшее решение за $\mathcal{O}(\deg A \cdot \deg B)$, призванное побороть страх перед делением:

```
pair<F*,F*> divide(int n, F *a, int m, F *b): // deg A = n, deg B = m, 𝔽 - поле
  F q[n-m+1];
  for (int i = n - m; i >= 0; i--): // коэффициенты в порядке убывания x^i
    q[i] = a[i+m] / b[m]; // m - степень ⇒ b[m] ≠ 0
    for (int j = 0; j <= m; j--) // вычитать имеет смысл, только если q[i] ≠ 0
      a[i+j] -= b[j] * q[i]; // можно соптимизить, перебирать только b[j] ≠ 0
  return {q, a}; // в массиве a[] как раз остался остаток
```

Теперь перейдём к решению за $\mathcal{O}(n\log^2 n)$.
Зная Q, мы легко найдём R, как $A(x) - B(x)Q(x)$ за $\mathcal{O}(n\log n)$. Сосредоточимся на поиске Q.
Пусть $\deg A = \deg B = n$, тогда $Q(x) = \frac{a_n}{b_n}$. То есть, $Q(x)$ можно найти за $\mathcal{O}(1)$.
Из этого мы делаем вывод, что Q зависит не обязательно от всех коэффициентов A и B.

Lm 6.3.1. $\deg A = m, \deg B = n \Rightarrow \deg Q = m - n$, и Q зависит только от $m{-}n{+}1$ старших коэффициентов A и $m{-}n{+}1$ коэффициентов B.

Доказательство. Рассмотрим деление в столбик, шаг которого: A `-=` $\alpha x^i B$. $\alpha = \frac{A[i+\deg B]}{B[\deg B]}$.
Поскольку $i + \deg B \geqslant \deg B = n$, младшие n коэффициентов A не будут использованы. ■

Теперь будем решать задачу:
Даны $A, B \in \mathbb{R}[x]\colon \gamma(A) = \gamma(B) = n$, найти $C \in \mathbb{R}[x]\colon \gamma(C) = n$,
что у A и BC совпадает n старших коэффициентов.

```
int* Div(int n, int *A, int *B) // n - степень двойки (для удобства)
  C = Div(n/2, A + n/2, B + n/2) // нашли старших n/2 коэффициентов ответа
  A' = Subtract(n, A, n+n/2-1, Multiply(C, B))
  D = Div(n/2, A', B + n/2) // сейчас A' состоит из n/2 не нулей и n/2 нулей
  return concatenate(D, C) // склеили массивы коэффициентов; вернули массив длины ровно n
```

Здесь `Subtract` – хитрая функция. Она знает длины многочленов, которые ей передали, и сдвигает вычитаемый многочлен так, чтобы старшие коэффициенты совместились.

Время работы: $T(n) = 2T(n/2) + \mathcal{O}(n\log n) = \mathcal{O}(n\log^2 n)$. Здесь $\mathcal{O}(n\log n)$ – время умножения.

6.4. (-) Деление чисел

Оптимально использовать метод Ньютона, внутри которого все умножения – FFT.
Тогда мы получим асимптотику $\mathcal{O}(n\log n)$. Об этом можно будет узнать на третьем курсе.

Разделяй и властвуй для многочленов также можно применить для чисел. Только аккуратно: мы вычислим не $\frac{n}{2}$ старших цифр, а лишь $\frac{n}{2} - \mathcal{O}(1)$.

Подробно мы сегодня изучим только метод за $\mathcal{O}((n/k)^2)$.

Простейшие методы (оценка времени деление числа битовой длины $2n$ на число длины n).

1. Бинпоиск по ответу: $\mathcal{O}(n^3/k^2)$ при простейшем умножении, $\mathcal{O}(n^2\log n)$ при Фурье внутри.
2. Бинарной арифметикой (+, -, `mul2`, `div2`): $\mathcal{O}(n^2/k)$ времени.
3. Деление в столбик: $\mathcal{O}(n^2/k)$, $\mathcal{O}(n^2/k^2)$ времени. На нём остановимся подробнее.

6.5. (-) Деление чисел за $\mathcal{O}((n/k)^2)$

Делить будем в столбик. У нас уже было деление многочленов за квадрат. Действуем также. Нужно научиться быстро искать старшую цифру частного.

- **«Бинпоиск»:** $\mathcal{O}(n^2/k)$

Обозначим систему счисления $\mathtt{S}$, $\log \mathtt{S} = k$.
Ищем старшую цифру частного бинарным поиском за $\mathcal{O}(n)$: k итераций бинпоиска, на каждой проверяем за $\mathcal{O}(n/k)$. Итоговое время деления $\mathcal{O}((n/k) \times k \times (n/k)) = \mathcal{O}(n^2/k)$.

- **«Деление старших цифр»:** $\mathcal{O}(n^2/k^2)$

Старшую цифру можно почти точно посчитать за $\mathcal{O}(1)$.
Если старшие цифры чисел a и b – a_n и b_m соответственно, то хочется взять $\approx \frac{a_n}{b_m}$.
Такая формула не работает. Пример: делим 99 на 19, получится $\frac{9}{1} = 9$, должно получиться 4. Тогда возьмём не одну, а две старшие цифры $z_0 = \frac{a_n a_{n-1}}{b_m b_{m-1}}$. Другой вариант $z_1 = \frac{a_n a_{n-1}}{b_m (b_{m-1}+1)}$. Ниже представлен код деления в столбик, в строке 3 вычисляется старшая цифра, как z_1.

```
1 Div(an, a, bn, b) // [0..an] [0..bn] a[i]·10^i
2   for (j = an - bn; j >= 0; j--) // считаем частное со старших разрядов
3     c[j] = (a[an-j]*S+a[an-1-j])/(b[bn]*S+b[bn-1]+1) // b[bn] != 0
4     a -= b*c[j]*S^i
5     while (a >= b*S^i) c[j]++, a -= b*S^i; // взяли цифру меньше, чем нужно
6     a[an-j-1] += a[an-j]*S, a[an-j] = 0 // перенос
```

Обозначим за x реальное значение старшей цифры.
Для обеих формул (z_0, z_1) можно показать, $|z_i - x| \leqslant 2$. Докажем для $z_1 = \frac{a_n a_{n-1}}{b_m (b_{m-1}+1)}$.

Lm 6.5.1. $z_1 \leqslant x$

Доказательство. При вычислении z_1 числитель x округлили вниз, знаменатель x вверх. ■

Lm 6.5.2. $a_n < \mathtt{S}^2$

Доказательство. Обычно цифры в системе счисления $\mathtt{S}$ меньше $\mathtt{S}$, но мы могли ещё $\mathtt{S}(\mathtt{S}-1)$ перенести из предыдущего разряда $\Rightarrow \max a_n = \mathtt{S}(\mathtt{S}-1) + (\mathtt{S}-1) < \mathtt{S}^2$. ■

Lm 6.5.3. $x \leqslant z_1 + 2$

Доказательство. $x \leqslant y = \frac{a_n(a_{n-1}+1)}{b_m b_{m-1}}$. Оценим разность $y - z_1 =$
$\frac{1}{b_m b_{m-1}} + \frac{a_n a_{n-1}}{b_m b_{m-1} \cdot b_m (b_{m-1}+1)} \leqslant \frac{1}{\mathtt{S}} + \frac{z_1}{b_m b_{m-1}} \leqslant \frac{1}{\mathtt{S}} + \frac{x}{S} \leqslant 1 + 1 = 2$ ■

Лекция #7: Умножение матриц и 4 русских

декабрь 2022

[Охотин '2021]. Конспект Александра Охотина по нашей теме.

[Арлазаров,Диниц,Кронрод,Фараджев '1970]. Оригинальная статья четырёх русских про быстрое умножение матриц... вернее про быструю композицию отображений.

[CF: оптимизации]. Про то, как ускорить умножение за куб в 50 раз. Можно скрещивать со Штрассеном.

7.1. Умножение матриц, простейшие оптимизации

Мы умеем за $\mathcal{O}(n^3)$. Из практически эффективных ещё есть алгоритм Штрассена за $\mathcal{O}(n^{\log_2 7})$, похожий на Карацубу, а из теоретических $\mathcal{O}(n^{2.37})$. Подробности в конспекте Охотина.

Мы не будем затрагивать решения за $\mathcal{O}(n^{3-\varepsilon})$, а сосредоточимся на технических оптимизациях метода за $\mathcal{O}(n^3)$ для матриц над $\mathbb{F}_2$.

• Обычное битовое сжатие и $\mathcal{O}(\frac{n^3}{w})$

Куб выглядит так.

```
for (i = 0; i < n; i++) // (i,k,j): можем выбрать любой порядок циклов
  for (k = 0; k < n; k++)
    for (j = 0; j < n; j++)
      c[i][j] ^= a[i][k] & b[k][j];
```

Если матрицы a, b, c представлены, как `bitset<n> a[n],b[n],c[n]`, то часть

```
for (j = 0; j < n; j++) // O(n)
  c[i][j] ^= a[i][k] & b[k][j];
```

эквивалентна версии за $\mathcal{O}(\frac{n}{w})$ (w – размер машинного слова):

```
if (a[i][k])
  c[i] ^= b[k]; // O(n/w)
```

То есть, c_i (i-я строка c) – сумма ровно тех строк b, которые помечены единицами в a_i.

• Предподсчёт и $\mathcal{O}(\frac{n^3}{\log n})$

Попробуем в решении за n^3 порядок циклов `for i, for j, for k`:
$c_{ij} = \oplus_k (a_{ik}$ & $b_{kj})$, разобьём эту сумму на части длины m.
Чтобы за $\mathcal{O}(1)$ посчитать сумму сразу m слагаемых, достаточно

1. Так хранить строки a и столбцы b, чтобы за $\mathcal{O}(1)$ получать целое число из нужных m бит.
2. Взять `AND` двух m-битных чисел.
3. Посчитать число бит в m-битном числе. Это предподсчёт за $\mathcal{O}(2^m)$: `bn[i] = bn[i>>1] + (i&1)`.

Возьмём $m = \log n$, получим $\mathcal{O}(m)$ на предподсчёт и $\mathcal{O}(n^3/\log n)$ на умножение.
На практике можно сделать предподсчёт, например, для $m = 20$ при $w = 32$
$\Rightarrow$ по скорости алгоритм работает также, как $\frac{n^3}{w}$.

7.2. Четыре русских

• Общие слова

Если задач какого-то вида мало, давайте предподсчитаем заранее ответы для всех возможных задач такого вида, а затем будем пользоваться предподсчётом в нужный момент за $\mathcal{O}(1)$.

• Задача

Даны A, B, найти $C = A \times B$. Умножаем матрицы над $\mathbb{Z}$ (на самом $\forall$ кольцо), предполагая, что A содержит только нули и единицы.

• Решение

Разобьём A на части по k **столбцов**: $A = A_1 A_2 \dots A_{n/k}$.
Разобьём B на части по k **строк**: $B = B_1 B_2 \dots B_{n/k}$.

Заметим, $A \times B = \sum_{i=1..n/k}(A_i \times B_i)$. Проверьте размерности: $(n \times k) \times (k \times n) = n \times n$.

Теперь сосредоточимся на умножении $C_1 = A_1 \times B_1$. Число k выберем в самом конце.

$\forall i$ строка $A_1[i]$ матрицы A_1 задаёт $C_1[i]$ (i-ю строку произведения C_1), которая является суммой (линейной комбинацией) строк B_1: $C_1[i] = \sum_j A_1[i,j] \cdot B_1[j]$.

Матрица A_1 состоит из $\{0,1\} \Rightarrow \exists$ всего 2^k различных строк $A_1[i]$. Предподсчитаем все 2^k сумм:

`sum[mask] = add(sum[mask ^ (1 << bit)], b[bit])`, где

`bit` – любой единичный бит `mask`, а функция `add` за $\mathcal{O}(n)$ складывает строки. После предподсчёта алгоритм умножения выглядит так: `for i: C[i] = sum[A[i]]` и работает за $\mathcal{O}(n^2)$.

Получили умножение $A_1 \times B_1$ за $2^k n + n^2 \Rightarrow$ оптимально взять $k = \log n$.
Итого $\frac{n}{k}$ умножений по $\Theta(n^2)$ каждое $\Rightarrow \mathcal{O}(n^3/\log n)$ на вычисление $A \times B$.

Замечание 7.2.1. Метод из данной главы подходит не только для $\mathbb{F}_2$ (см. постановку задачи).

7.3. Умножение матриц над $\mathbb{F}_2$ за $\mathcal{O}(n^3/(w \log n))$

Если в явном виде применить идеи четырёх русских и битового сжатия, то получится ровно $\mathcal{O}(\frac{n^2}{\log n})$ операций со строками, каждая операция за $\mathcal{O}(\frac{n}{w}) \Rightarrow \mathcal{O}(n^3/(w \log n))$.

Прикинем время для $n = 4\,000$ и $w = 64$: получаем $\approx 83 \cdot 10^6$ операций ≈ 0.1 секунда.

7.4. НОП за $\mathcal{O}(n^2/\log^2 n)$ (на практике)

Задача: даны две строки над алфавитом $\{0,1\}$, найти длину НОП.

Рассмотрим обычную динамику: $f[i,j] = \begin{cases} f[i{-}1, j{-}1] + 1 & \text{если } s[i] = t[j] \\ \max(f[i{-}1, j], f[i, j{-}1]) & \text{иначе} \end{cases}$

Идея: зафиксируем $k = \frac{1}{4}\log n$ и будем за $\mathcal{O}(1)$ сразу насчитывать кусок матрицы $k \times k$.

Заметим $\forall i, j\ 0 \leqslant f[i{+}1, j] - f[i,j] \leqslant 1 \ \wedge\ 0 \leqslant f[i, j{+}1] - f[i,j] \leqslant 1$. Также $\forall i\ f[i,0] = f[0,i] = 0$.
Давайте хранить только битовые матрицы $x[i,j] = f[i,j{+}1] - f[i,j],\ y[i,j] = f[i{+}1,j] - f[i,j]$.
Зафиксируем k и любую клетку $[i,j]$ пусть мы знаем «угол квадрата»: $y[i..i{+}k, j]$ и $x[i, j..j{+}k]$.
Противоположный «угол квадрата» ($y[i..i{+}k, j]$ и $x[i, j..j{+}k]$) зависит только от $4k$ бит:
$y[i..i{+}k, j], x[i, j..j{+}k], s[i..i{+}k], t[j..j{+}k] \Rightarrow$ используем четырёх русских и за $\mathcal{O}^*(2^{4k})$ всё предподсчитываем. Важно, что $y[i..i{+}k, j]$, $x[i, j..j{+}k]$, $y[i..i{+}k, j]$, $x[i, j..j{+}k]$, $s[i..i{+}k]$, $t[j..j{+}k]$ –

целые k-битные числа $\Rightarrow$ операции с ними происходят за $\mathcal{O}(1)$.

Заметьте, для каждого квадрата $k \times k$ мы получали только значения на границы. Другие нам и не нужны. Также мы ни в какой момент времени не пытались считать f, нам хватает x и y. Реальные значения f возникают внутри предподсчёта. Ещё нам в самом конце нужна собственно длина НОП – это сумма последней строки x.

7.5. (-) Схема по таблице истинности

Задача. Дана таблица истинности булевой функции от n переменных, вектор длины 2^n из нулей и единиц. Построить булеву схему с такой таблицей истинности.

Построить минимальную схему NP-трудно.

КНФ и ДНФ дадут $\mathcal{O}(2^n n)$ гейтов.

Разделяй и властвуй: построить формулу φ_0 от $n{-}1$ переменной для $x_n = 0$ и φ_1 для $x_n = 1$, ответ: $\varphi = (\varphi_0 \wedge x_n = 0) \vee (\varphi_1 \wedge x_n = 1)$. Получаем размер $S(n) = 2S(n{-}1) + 3 = \Theta(2^n)$.

К последнему решению можно применить оптимизацию предподсчёта: $n \leqslant k \Rightarrow$ все $m = 2^{2^k}$ возможных функций посчитано. Для $k = (\log n) - 1$ имеем $m = 2^{\frac{n}{2}}$, построим схемы для всех m возможных функций из k переменных, получим отсечение: на глубине рекурсии $n - k$ вместо построения φ взять уже готовую.

Замечание 7.5.1. Первые два решения могут строить и формулу, и схему. В последнем решении мы переиспользуем одну и ту же часть $\Rightarrow$ схему мы так можем построить, а формулу нет.

Замечание 7.5.2. Для «схемы от случайной таблицы истинности» мы получили асимптотически оптимальное решение.

7.6. (-) Оптимизация перебора для клик

Рассмотрим перебор для поиска максимальной клики

```
int go(int A): // A = маска вершин, которые можно добавить к клике
  if (A == 0) return 0 // больше никого не добавить...
  int i = anyBit(A) // например, младший бит мы точно умеем за O(1)
  return max(A & 2^i, 1 + go(A & graph[i])); // graph[i] - соседи i
go(2^n-1) // изначально можно брать все вершины
```

Если $|A| \leqslant 7 = k$, мы можем сказать «оставшийся граф мал, обратимся к предподсчёту». Нужно заранее предподсчитать ответы для всех $2^{k(k-1)/2} = 2^{21}$ возможных графов из $k = 7$ вершин.

Получили оптимизацию по времени работы $T(n) \to 2^{k(k-1)/2} + T(n-k)$.

7.7. (-) Транзитивное замыкание

На практике мы также изучим, как транзитивное замыкание

1. Сводить к $\mathcal{O}(\log n)$ умножениям матриц.
2. Считать за $\mathcal{O}$(одного умножения матриц).
3. Считать инкрементально за $\mathcal{O}(q\frac{n}{w} + \frac{n^3}{w})$.