Второй курс, осенний семестр 2017/18
Конспект лекций по алгоритмам

Собрано 15 июля 2018 г. в 19:06

Содержание

Лекция #1: Mincost

2 октября 2017

1.1. Mincost k-flow в графе без отрицательных циклов

Сопоставим всем прямым рёбрам вес (стоимость) $w_e \in \mathbb{R}$.

Def 1.1.1. *Стоимость потока* $W(f) = \sum_e w_e f_e$. *Сумма по прямым рёбрам.*

Обратному к e рёбру $\overline{e}$ сопоставим $w_{\overline{e}} = -w_e$.

Если толкнуть поток сперва по прямому, затем по обратному к нему ребру, стоимость не изменится. Когда мы толкаем единицу потока по пути `path`, изменение потока и стоимости потока теперь выглядят так:

```
for (int e : path):
  edges[e].f++
  edges[e ^ 1].f--
  W += edges[e].w;
```

Задача mincost k-flow: найти поток $f\colon |f| = k, W(f) \to \min$

При решении задачи мы будем говорить про веса путей, циклов, “отрицательные циклы”, кратчайшие пути... Везде вес пути/цикла – сумма весов рёбер (w_e).

Решение #1. Пусть в графе нет отрицательных циклов, а также все $c_e \in \mathbb{Z}$.
Тогда по аналогии с алгоритмом Ф.Ф., который за $\mathcal{O}(k \cdot \texttt{dfs})$ искал поток размера k, мы можем за $\mathcal{O}(k \cdot \texttt{FordBellman})$ найти mincost поток размера k. Обозначим f_k оптимальный поток размера $k \Rightarrow f_0 \equiv 0, f_{k+1} = f_k + path$, где $path$ – кратчайший в G_{f_k}.

Lm 1.1.2. $\forall k, |f| = k \quad (W(f) = \min) \Leftrightarrow (\nexists$ отрицательного цикла в $G_f)$

Доказательство. Если отрицательный цикл есть, увеличим по нему поток, $|f|$ не изменится, $W(f)$ уменьшится. Пусть $\exists f^*\colon |f^*| = |f|, W(f^*) < W(f)$, рассмотрим поток $f^* - f$ в G_f.
Это циркуляция, мы можем декомпозировать её на циклы $c_1, c_2, \dots, c_k$.
Поскольку $0 > W(f^* - f) = W(c_1) + \dots + W(c_k)$, среди циклов c_i есть отрицательный. ■

Теорема 1.1.3. Алгоритм поиска mincost потока размера k корректен.

Доказательство. База: по условию нет отрицательных циклов $\Rightarrow f_0$ корректен.
Переход: обозначим f^*_{k+1} mincost поток размера $k+1$, смотрим на декомпозицию $\Delta f = f^*_{k+1} - f_k$. $|\Delta f| = 1 \Rightarrow$ декомпозиция = путь p + набор циклов. Все циклы по 1.1.2 неотрицательны $\Rightarrow W(f_k + p) \leqslant W(f^*_{k+1}) \Rightarrow$, добавив, кратчайший путь мы получим решение не хуже f^*_{k+1}. ■

Lm 1.1.4. Если толкнуть сразу $0 \leqslant x \leqslant \min_{e \in p}(c_e - f_e)$ потока по пути p,
то получим оптимальный поток размера $|f| + x$.

Доказательство. Обозначим f^* оптимальный поток размера $|f| + x$, посмотрим на декомпозицию $f^* - f$, заметим, что все пути в ней имеют вес $\geqslant W(p)$, а циклы вес $\geqslant 0$. ■

1.2. Потенциалы и Дейкстра

Для ускорения хотим Форда-Беллмана заменить на Дейкстру.
Для корректности Дейкстры нужна неотрицательность весов.
В прошлом семестре мы уже сталкивались с такой задачей, когда изучали алгоритм Джонсона.

• Решение задачи mincost k-flow.

Запустим один раз Форда-Беллмана из s, получим массив расстояний d_v, применим потенциалы d_v к весам рёбер:

$$e\colon a \to b \Rightarrow w_e \to w_e + d_a - d_b$$

Напомним, что из корректности d имеем $\forall e\ d_a + w_e \geqslant d_b \Rightarrow w_e^{'} \geqslant 0$.
Более того: для всех рёбер e кратчайших путей из s верно $d_a + w_e = d_b \Rightarrow w_e^{'} = 0$.

В G_f найдём Дейкстрой из s кратчайший путь p и расстояния d'_v.
Пустим по пути p поток, получим новый поток $f^{'} = f + p$.
В сети $G_f^{'}$ могли появиться новые рёбра (обратные к p). Они могут быть отрицательными.
Пересчитаем веса:

$$e\colon a \to b \Rightarrow w_e \to w_e + d'_a - d'_b$$

Поскольку d' – расстояния, посчитанные в G_f, все рёбра из G_f останутся неотрицательными.
p – кратчайший путь, все рёбра p станут нулевыми $\Rightarrow$ рёбра обратные p тоже будут нулевыми.

• Псевдокод

```
def applyPotentials(d):
  for e in Edges:
    e.w = e.w + d[e.a] - d[e.b]
d <-- FordBellman(s)
applyPotentials(d)
for i = 1..k:
  d, path <-- Dijkstra(s)
  for e in path: e.f += 1, e.rev.f -= 1
  applyPotentials(d)
```

1.3. Графы с отрицательными циклами

Задача: найти mincost циркуляцию.

Алгоритм Клейна: пока в G_f есть отрицательный цикл, пустим по нему $\min_e(c_e - f_e)$ потока.

Пусть $\forall e\ c_e, w_e \in \mathbb{Z} \Rightarrow W(f)$ каждый раз уменьшается хотя бы на $1 \Rightarrow$ алгоритм конечен.

Задача: найти mincost k-flow циркуляцию в графе с отрицательными циклами.

Решение #1: найти за $|W(f)|$ итераций mincost циркуляцию, перейти от f_0 за k итераций к f_k.
Решение #2: найти любой поток $f\colon |f| = k$, в G_f найти mincost циркуляцию, сложить с f.

1.4. Mincost flow

Задача: найти $f\colon W(f) = \min$, размер f не важен.
Обозначим f_k – оптимальный поток размера k, p_k кратчайший путь в G_{f_k}.

Lm 1.4.1. $W(p_k)\nearrow$, как функция от k.

Доказательство. Аналогично доказательству леммы для Эдмондса-Карпа **??**.

От противного. Был поток f, мы увеличили его по кратчайшему пути p.
Расстояния в G_f обозначим d_0, в G_{f+p} – d_1.
Возьмём v: $d_1[v] < d_0[v]$, а из таких ближайшую к s в дереве кратчайших путей.
Рассмотрим кратчайший путь q в G_{f+p} из s в v: $s \rightsquigarrow \dots \rightsquigarrow x \to v$.
$e = (v \to x), d_1[v] = d_1[x] + w_e, d_1[x] \geqslant d_0[x] \Rightarrow d_1[v] \geqslant d_0[x] + w_e \Rightarrow$ ребра $(x \to v)$ нет в $G_f \Rightarrow$ ребро $(v \to x) \in p \Rightarrow d_0[x] = d_0[v] + w_{\overline{e}} = d_0[v] - w_e \Rightarrow$
$d_1[v] = d_1[x] + w_e \geqslant d_0[x] + w_e = (d_0[v] - w_e) + w_e = d_0[v]$. Противоречие. ∎

Следствие 1.4.2. $(W(f_k) = \min) \Leftrightarrow (W(p_{k-1}) \leqslant 0 \wedge W(p_k) \geqslant 0)$.

Осталось найти такое k бинпоиском или линейным поиском. На текущий момент мы умеем искать f_k или за $\mathcal{O}(k \cdot VE)$ с нуля, или за $\mathcal{O}(VE)$ из $f_{k-1} \Rightarrow$ линейный поиск будет быстрее.

1.5. Полиномиальные решения

Mincost flow мы можем бинпоиском свести к mincost k-flow.
Mincost k-flow мы можем поиском любого потока размера k свести к mincost циркуляции.
Осталось научиться за полином искать mincost циркуляцию.

• **Решение #1:** модифицируем алгоритм Клейна, будем толкать $\min_e(c_e - f_e)$ потока по циклу min среднего веса. Заметим, что ($\exists$ отрицательный цикл) $\Leftrightarrow$ (min средний вес < 0).

Решение работает за $\mathcal{O}(VE\log(nC))$ поисков цикла. Цикл ищется алгоритмом Карпа за $\mathcal{O}(VE)$.
Доказано будет на практике.

• **Решение #2:** Capacity Scaling.

Начнём с графа $c'_e \equiv 0$, в нём mincost циркуляция тривиальна.
Будем понемногу наращивать c'_e и поддерживать mincost циркуляцию. В итоге хотим $c'_e \equiv c_e$.

```
1  for k = logU..0:
2    for e in Edges:
3      if c_e содержит бит 2^k:
4        c'_e += 2^k  // e: ребро из a_e в b_e
5        Найдём p - кратчайший путь a_e → b_e
6        if W(p) + w_e ⩾ 0:
7          нет отрицательных циклов ⇒ циркуляция f оптимальна
8        else:
9          пустим 2^k потока по циклу p + e (изменим f)
10         пересчитаем потенциалы, используя расстояния, найденные Дейкстрой
```

Время работы алгоритма $E\log U$ запусков Дейкстры $= E(E + V\log V)\log U$.

Lm 1.5.1. После 9-й строки циркуляция f снова минимальна.

Доказательство. f – минимальная циркуляция до 4-й строки, f' – после.
Как обычно, рассмотрим $f' - f$. Это тоже циркуляция. Декомпозируем её на единичные циклы.
Любой цикл проходит через e (иначе f не оптимальна). Через e проходит не более 2^k циклов.
Каждый из этих циклов имеет вес не меньше веса $p + e \Rightarrow W(f') \geqslant W(f + 2^k(p + e))$. ∎

1.6. (*) Cost Scaling

Cost scaling (часть 1)
Cost scaling (часть 2)

Лекция #2: Суффиксный массив

16 октября 2017

Def 2.0.1. *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 $\mathcal{O}(n^2 \log n)$.

2.1. 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 2.1.1. Заодно научились за $\mathcal{O}(\log n)$ сравнивать на больше/меньше любые подстроки.

2.2. 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 $\min k$: $t[p_k:] \geqslant s_i$. 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. 2.6 мы улучшим время обработки запроса до $\mathcal{O}(|s_i| + \log |t|)$.

2.3. 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 $c[i]$: $(0 \leqslant c[i] < n) \wedge (\texttt{s[i:i+k)} < \texttt{s[j:j+k)} \Leftrightarrow c[i] \leqslant c[j])$.

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 2.3.1. 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 2.3.2. При написании суффмассива в контесте рекомендуется, прочтя конспект, написать код самостоятельно, без подглядывания в конспект.

2.4. 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 2.4.1. 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 2.4.2. 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) $\overset{2.4.1}{\Rightarrow}$
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 $i \searrow$ 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 $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$.

2.5. 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:

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

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:
$3 \to 1{+}1,\ 4 \to 2{+}1,\ 5 \to 2{+}2,\ \ldots.$

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 $\mathcal{O}(n)$.

2.6. 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 2.6.1. 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 $x = z_i \wedge z_{i+1} = lcp(s, m_i)$. ■

Лекция #3: Быстрое преобразование Фурье

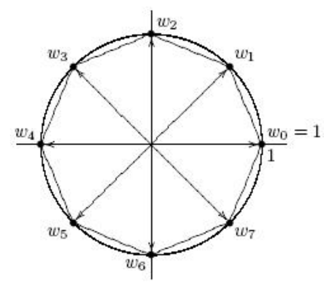
4 декабря 2017

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:

$e^{2\pi i\alpha}e^{2\pi i\beta} = e^{2\pi i(\alpha+\beta)}$ $e^{2\pi i\varphi} = (\cos\varphi, \sin\varphi)$, $\overline{(a,b)} = (a, -b) \Rightarrow \overline{e^{2\pi i\varphi}} = e^{2\pi i(-\varphi)}$
Извлечение корня k-й степени: $\sqrt[k]{z} = \sqrt[k]{e^{2\pi i\varphi}} = e^{2\pi i\varphi/k}$
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: $e^{2\pi ij/k}$.

3.1. Прелюдия к FFT

Возьмём $\min N = 2^t \geqslant n$ и $w_j = e^{2\pi ij/N}$. 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:

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

3.2. 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[0] # 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 $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$.

3.3. Крутая реализация FFT

Чтобы преобразование работало быстро, нужно заранее предподсчитать все $w_j = e^{2\pi i j/N}$.
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] + ((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:

```
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;
      }
}
```

3.4. 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 $\sum\limits_{k=0}^{n-1} w^{jk} = n$.

Поэтому $fa_0 + fa_1 + fa_2 + \dots = a_0 n + a_1 \sum_k w^k + a_2 \sum_k w^{2k} + \dots = a_0 n$
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 --> f
  FFT(a, fa)
  reverse(fa + 1, fa + N) // w^j <--> w^{-j}
  for (int i = 0; i < N; i++) fa[i] /= N;
}
```

3.5. 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:
$\overline{x+iy} = x - iy, \overline{a \cdot b} = \overline{a} \cdot \overline{b}, w^{n-j} = w^{-j} = \overline{w^j} \Rightarrow \overline{fc_{n-j}} = \overline{C(w^{n-j})} = \overline{C}(w^j) \Rightarrow$
$fc_j + \overline{fc_{n-j}} = 2A(w^j) = 2fa_j$. Аналогично $fc_j - \overline{fc_{n-j}} = 2B(w^j) = 2fb_j$.

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

3.6. 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.

Лекция #4: Длинная арифметика

11 декабря 2017

Мы займёмся целыми беззнаковыми числами. Целые со знаком – ещё отдельно хранить знак. Вещественные – то же, но ещё 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 $\texttt{BASE} = 10^k$, $k \to \max$: 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  void 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  int divide( int *a, int k ) { // деление на короткое за O(n/k), делим со старших разрядов
8    long long carry = 0; // перенос с более старшего разряда, он же остаток
9    for (int i = N - 1; i >= 0; i--) {
10     carry = carry * BASE + a[i]; // максимальное значение carry < BASE^2
11     a[i] = carry / k, carry %= k;
12   }
13   return carry; // как раз остаток
14 }
15 int mul_slow( int *a, int *b, int *c ) { // умножение за (n/k)^2
16   fill(c, c + N, 0);
17   for (int i = 0; i < N; i++)
18     for (int j = 0; i + j < N; j++)
19       c[i + j] += a[i] * b[j]; // здесь почти наверняка произойдёт переполнение
20   for (int i = 0; i + 1 < N; i++) // сначала умножаем, затем делаем переносы
21     c[i + 1] += c[i] / BASE, c[i] %= BASE;
22 }
23 void out( int *a ) { // вывод числа за O(n/k)
24   int i = 0;
25   while (i && !a[i]) i--;
26   printf("%d", a[i--]);
27   while (i >= 0) printf("%0*d", BASE_LEN, a[i--]); // воспользовались таки BASE_LEN!
28 }
```

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

4.1. Бинарная арифметика

Пусть у нас реализованы простейшие процедуры: “+, -, *2, /2, %2, >, $\geqslant$, 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 1; // 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), так как длины a и b убывают от n до 1
      sub(a, b), add(res, c); O(n)
    div2(b), div2(c); O(n)
  }
  return {res, a};
}
```

Глубина рекурсии равна $n-m$. 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(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
  }
  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: $\mathcal{O}(\max(n, m)^2/k)$.

4.2. 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--) { // выводим коэффициенты частного в порядке убывания
    q[i] = a[i + m] / b[m]; // m - степень ⇒ b[m] ≠ 0.
    for (int j = 0; j <= m; j--) // конечно, вычитать имеет смысл, только если q[i] ≠ 0
      a[i + j] -= b[j] * q[i]; // можно соптимизить, перебирать только ненулевые b[j]
  }
  return {q, a}; // в 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 4.2.1. $\deg A = m, \deg B = n \Rightarrow \deg Q = m - n$, и Q зависит только от $m{-}n{+}1$ старших коэффициентов A и $m{-}n{+}1$ коэффициентов B.

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

Теперь будем решать задачу:
Даны $A, B \in \mathbb{R}[x]$: $\gamma(A) = \gamma(B) = n$, найти $C \in \mathbb{R}[x]$: $\gamma(C) = n$,
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.

4.3. 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.
Сегодня лучшими результатами будут $\mathcal{O}((n/k)^2)$ и $\mathcal{O}(n\log^2 n)$.

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

1. Бинпоиск по ответу: n^3/k^2 при простейшем умножении, $n^2\log n$ при Фурье внутри.
2. Двоичное деление: n^2/k времени.
3. Деление в столбик: n^2/k^2 времени. На нём остановимся подробнее.

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

Делить будем в столбик. У нас уже было деление многочленов за квадрат. Если мы научимся вычислять за $\mathcal{O}(n/k)$ старшую цифру частного, мы сможем воспользоваться им без изменений. Пусть даны числа $a, b, |a| = n, |b| = m$.

Lm 4.4.1. Старшая цифра $\frac{a}{b}$ отличается от $x = \frac{a_n a_{n-1}}{b_m b_{m-1}}$ не более чем на 1.

Доказательство. $\frac{a_n a_{n-1}}{b_m b_{m-1} + \frac{1}{base}} \leqslant \frac{a}{b} \leqslant \frac{a_n a_{n-1} + \frac{1}{base}}{b_m b_{m-1}} \Rightarrow |\frac{a}{b} - x| \leqslant (\frac{a_n a_{n-1} + \frac{1}{base}}{b_m b_{m-1}} - x) + (x - \frac{a_n a_{n-1}}{b_m b_{m-1} + \frac{1}{base}}) := y$

Заметим, $b_m \neq 0 \Rightarrow b_m b_{m-1} \geqslant base$. Продолжаем преобразования:
$y \leqslant \frac{1}{base} \cdot \frac{1}{b_m b_{m-1}} + \frac{a_n a_{n-1}}{base(b_m b_{m-1})^2} = \frac{1}{base \cdot b_m b_{m-1}}(1 + \frac{a_n a_{n-1}}{b_m b_{m-1}}) \leqslant \frac{1}{base^2}(1 + \frac{base^2}{base}) \leqslant 1.$ ■

• **Алгоритм деления:**

Длина частного, т.е. $\frac{n-m}{k}$, раз вычисляем α – приближение старшей цифры частного за $\mathcal{O}(1)$, затем умножением за $\mathcal{O}(\frac{n}{k})$ вычитаем $(\alpha - 1) b\, 10^{ki}$ из a и не более чем двумя вычитаниями $b\, 10^{ki}$ доводим дело до конца. Важно было начать с $\alpha - 1$, чтобы не уйти в минус при вычитании.