

Программа к экзамену по курсу "Алгоритмы
и структуры данных"

Содержание

1	(1) Префикс-функция. Линейный алгоритм подсчета. Алгоритм Кнута-Морриса-Прата.	8
2	(1) Z-функция. Линейный алгоритм подсчета. Алгоритм Кнута-Морриса-Прата.	10
3	(1) Полиномиальное хеширование. Алгоритм Рабина-Карпа.	13
4	(1) Бор. Сравнительный анализ времени работы в зависимости от выбора контейнера для хранения дочерних вершин.	14
5	(3) Суффиксный массив. Построение за $O(S \log S)$.	14
6	(3) Понятие LCP. Массив lcp. Алгоритм Касай-Аrimуры-Арикавы-Ли-Пака построения lcp (корректность б/д). Поиск LCP с использованием массива lcp и решения задачи RMQ.	15
7	(2) Суффиксные ссылки в боре. Функция перехода. Алгоритм Ахо-Корасик построения суффиксных ссылок.	17
8	(2) Сжатые суффиксные ссылки. Автомат Ахо-Корасик. Поиск множества паттернов в тексте.	18
9	(1) Понятие правого контекста. Классы эквивалентности по равенству правых контекстов. Теорема Майхилла-Нероуда (б/д). Определение суффиксного автомата.	19
10	(1) Атрибуты состояний. Устройство класса эквивалентности.	19

11 Понятие longest. Критерий longest. Устройство ребер в суффиксном автомате.	20
12 (2) Новые состояния в суффиксном автомате при дописывании символа. Алгоритм построения суффиксного автомата: случаи 1 (новая буква) и 2 (без порождения clone).	22
13 (3) Алгоритм построения суффиксного автомата: все случаи.	25
14 (2) Время построения суффиксного автомата.	26
15 (1) Задача интерполяции. Матричная постановка, существование и единственность решения. Комплексные корни из единицы. Обратная к матрице Вандермонда на комплексных корнях из единицы.	26
16 Дискретное преобразование Фурье. Быстрое преобразование Фурье. Преобразование бабочки.	28
17 Обратное преобразование Фурье. Перемножение многочленов.	31
18 Свертка последовательностей. Расстояние Хэмминга. Алгоритм поиска вхождений паттерна в текст не более k расхождениями.	32
19 Персистентные структуры данных. Частичная и полная персистентность. Персистентный стек.	34
20 Персистентные структуры данных. Частичная и полная персистентность. Персистентный массив	35

21 Процедура Merge. Оптимальное время работы: оценка.	35
22 Процедура Merge. Оптимальное время работы: оценка б/д, алгоритм.	37
23 Inplace Merge Sort.	37
24 Алгоритм Introsort.	38
25 Биномиальные деревья. Свойства: число вершин на уровне. Процедуры SiftDown, SiftUp и DecreaseKey в биномиальном дереве.	39
26 Биномиальная куча. Время работы слияния куч. Выполнение основных операций кучи.	40
27 Фибоначчиева куча. Топология узла. Структура кучи. Операции Insert, Merge, GetMin.	41
28 Фибоначчиева куча. Структура кучи. Операции Consolidate, ExtractMin. Оценка на $D(n)$	43
29 Фибоначчиева куча. Структура кучи. Операции Consolidate, DecreaseKey. Оценка на $D(n)$.	46
30 Алгоритмы во внешней памяти. Устройство памяти и понятие I/O-операции. Понятия latency, throughput. Модель оценивания времени на основе I/O-операций. Модельная задача: сумма последовательности во внешней памяти.	47
31 Алгоритмы во внешней памяти. Модель оценивания времени на основе I/O-операций. Сортировка во внешней памяти.	49

32 (a, b)-дерево. Операции SplitNode, Fuse, Share.	50
33 Алгоритмы во внешней памяти. Модель оценивания времени на основе I/O-операций. (a, b)- дерево. Основные операции с (a, b)-деревом.	52
34 Алгоритмы во внешней памяти. Модель оценивания времени на основе I/O-операций. BufferTree как (a, b)-дерево. Разбиение буфера на две части: основную и накопительную с родителями.	53
35 Задание полуплоскости. Пересечение полуплоскостей за $O(N^2)$.	53
36 Пересечение полуплоскостей за $O(N \log N)$ через построение огибающих.	54
37 Пересечение полуплоскостей за $O(N \log N)$ без построения огибающих.	54
38 Диаграмма Вороного за $O(N^3)$ или за $O(N^2 \log N)$.	55
39 Диаграмма Вороного. Линейность числа ребер и граней.	56
40 Диаграмма Вороного. Критерии того, что точка является вершиной или лежит на ребре в диаграмме Вороного.	58
41 Диаграмма Вороного. Алгоритм Форчуна.	58
42 Граф Делоне. Линейность числа ребер и граней в графе Делоне. Критерии того, что вершина/ребро будут в графе Делоне.	61

43 Независимость величины минимального угла от триангуляции граней графа Делоне. Два алгоритма построения: двойственный к диаграмме Вороного и модификация алгоритма Форчуна.	62
44 Легальное ребро. Критерий легальности ребра, триангуляции. Легальность триангуляции Делоне.	63
45 3D выпуклая оболочка. Алгоритм Джарвиса.	64
46 Построение триангуляции Делоне через 3D выпуклую оболочку.	65
47 Открытая адресация. Процедуры вставки, поиска. Артефакт <i>tombstone</i> . Время работы б/д.	66
48 Задача Perfect hashing. Решение для static версии. Алгоритм FKS. Доказательство времени работы для первого уровня.	68
49 Задача Perfect hashing. Решение для static версии. Алгоритм FKS. Доказательство времени работы для второго уровня.	69
50 k-независимое семейство хеш-функций. Пример. Обоснование отсутствия коллизий до взятия по модулю числа бакетов.	70
51 Cuckoo hashing. Время работы Insert б/д.	71
52 Задача фильтра. Фильтр Блума. Процедура выбора гиперпараметров.	72
53 XOR-filter. Алгоритм построения. Размер б/д.	73

1. (1) Префикс-функция. Линейный алгоритм подсчета. Алгоритм Кнута-Морриса-Прата.

Определение. Стока T называется супрефиксом строки S , если она является одновременно и префиксом, и суффиксом строки S .

Замечание. Пустая строка всегда является супрефиксом любой строки (кроме пустой).

Определение. Префикс-функцией от строки S называют такой массив $\pi(S)$ длины $|S|$, что $\pi(S)[i]$ равно длине максимального несобственного (то есть не равного всей строке) супрефикса строки $S[: i + 1]$.

Утверждение. $\pi[i + 1] \leq \pi[i] + 1$

Доказательство. Рассмотрим подстроку вида суффикс строки $S[: i + 2]$ длины $\pi[i + 1]$. Удалив последний символ этой подстроки, мы получим суффикс, оканчивающийся на позиции i и имеющий длину $\pi[i + 1] - 1 \leq \pi[i]$. \square

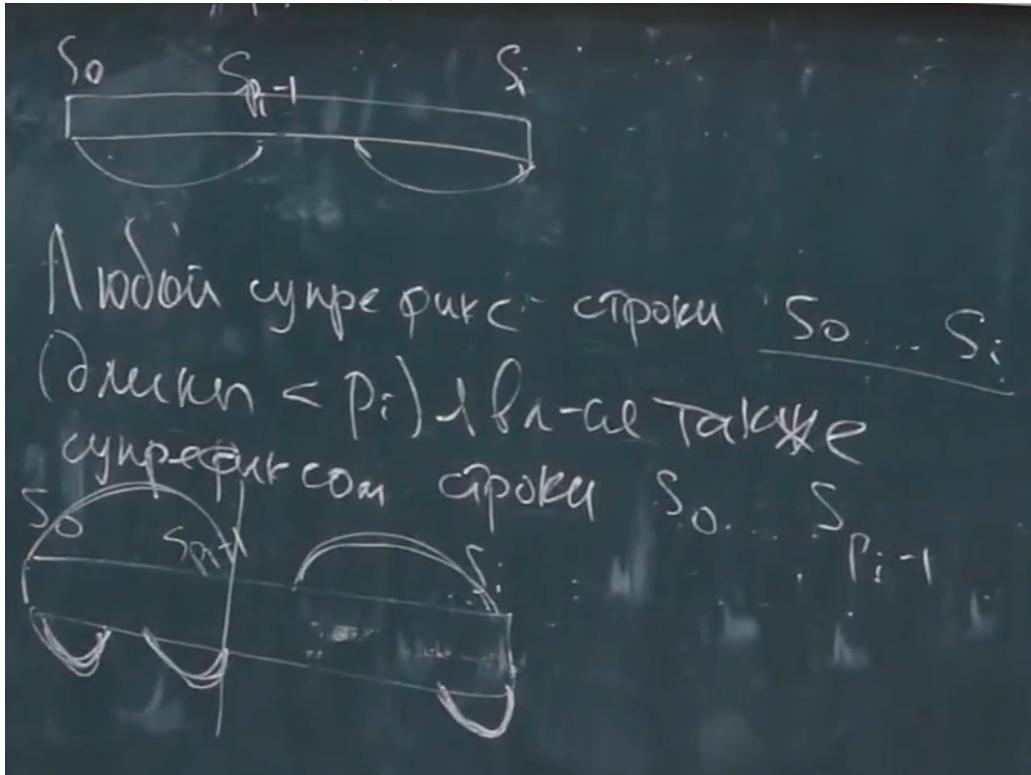
Линейный алгоритм подсчета.

Пусть $\forall j \leq i$ известно $\pi[j]$. Подсчитаем $\pi[i + 1]$:

- $S[i + 1] = S[\pi[i]] \Rightarrow \pi[i + 1] = \pi[i] + 1$.
- $S[i + 1] \neq S[\pi[i]]$. Цель - максимальный по длине супрефикс, заканчивающийся в $(i + 1)$ -й позиции. Такой супрефикс состоит из какого-то супрефикса строки $S[: i + 1]$ и символа $S[i + 1]$. Переберем все супрефиксы $S[: i + 1]$ в порядке вложенности, пусть они имеют длины $j_1 > j_2 > \dots > j_k$, тогда $\pi[i + 1] = j_l + 1$, где j_l таков, что $S[j_l + 1] = S[i + 1]$

Утверждение. Супрефиксы строки S в порядке вложенности имеют длины $\pi[|S| - 1], \pi[\pi[|S| - 1] - 1], \pi[\pi[\pi[|S| - 1] - 1] - 1], \dots$

Доказательство.



□

```
1 Array<int> PrefixFunction(String s) {  
2     Array<int> pi_func(len(s), 0);  
3     for (int i = 1; i < len(s); ++i) {  
4         int k = pi_func[i - 1];  
5         while (k > 0 and s[i] != s[k]) { k = pi_func[k - 1]; }  
6         pi_func[i] = k;  
7         if (s[i] == s[k]) { ++pi_func[i]; }  
8     }  
9     return pi_func;  
10 }
```

Так как k может увеличиться не более чем на 1, значит оно не больше $|S| - 1$. А внутри while оно убывает, значит суммарное число итераций while не превосходит $|S| - 1$.

Таким образом, время работы: $O(|S|)$.

Алгоритм Кнута-Морриса-Прата.

Требуется найти все позиции, начиная с которых P входит в T .

1. $S = P \# T$
2. Вычислим префикс функцию.
3. $\pi[i] = |P| \Rightarrow$ нашли конец вхождения.

2. (1) Z-функция. Линейный алгоритм подсчета.

Алгоритм Кнута-Морриса-Прата.

Определение. Z-функцией от строки S называют такой массив $z(S)$ длины $|S|$, что $z(S)[i]$ равно длине максимального префикса начинающегося $S[i :]$, который одновременно является и префиксом всей строки S .

Пример.

$$\text{zet_func(abcdabscabcdabia)} = [0, 0, 0, 0, 2, 0, 0, 0, 6, 0, 0, 0, 2, 0, 0, 1]$$

Линейный алгоритм подсчета.

Определение. Z-блоком назовем подстроку с началом в позиции i и длиной $z[i]$.

Будем поддерживать Z-блок строки S с максимальной позицией конца (среди них наибольший по длине), обозначим его границы за $left, right$. Пусть $\forall j < i$ известно $z[j]$. Подсчитаем $z[i]$:

- Пусть $i > right$, тогда наивно идем по строке S и сравниваем $S[i + j]$ и $S[j]$. Пусть j первая позиция в строке S для которой не выполняется равенство $S[i + j] = S[j]$, тогда $z[i] = j$. Тогда $left = i, right = i + j - 1$.

- Пусть $i \leqslant right$, тогда сравним $z[i - left] + i$ и $right$.
 - Если $right < z[i - left] + i$, то наивно идём по строке, начиная с $right$, и вычисляем значение $z[i]$.
 - Иначе мы уже знаем верное значение $z[i]$, так как оно равно значению $z[i - left]$.

```

1 Array<int> ZetFunction(String s) {
2     Array<int> z(len(s), 0);
3     int left = 0, right = 0;
4     for (int i = 1; i < len(s); ++i) {
5         z[i] = max(0, min(right - i, z[i - left]));
6         while (s[z[i]] == s[i + z[i]] and i + z[i] < len(s)) {
7             ++z[i];
8         }
9         if (i + z[i] > right) { left = i, right = i + z[i]; }
10    }
11    return z;
12
13 }

```

Заметим, что данный алгоритм обращается к каждому символу строки не более двух раз: когда наивно насчитываем $zet_func[i]$ и когда он только попадает в $[left, right]$.

Поэтому время работы: $O(|S|)$.

Алгоритм Кнута-Морриса-Прата.

Требуется найти все позиции, начиная с которых P входит в T .

1. $S = P \# T$
2. Вычислим Z-функцию.
3. $z[i] = |P| \Rightarrow$ нашли начало вхождения.

3. (1) Полиномиальное хеширование. Алгоритм Рабина-Карпа.

Определение. Полиномиальная хеш-функция для строки S определяется как

$$h(S) = \left(\sum_{i=0}^{|S|-1} S[i] \cdot x^i \right) \mod p$$

где p – простое число, а $x \in \mathbb{Z}_p^*$.

Замечание.

$$h(Sc) = \left(\sum_{i=0}^{|S|-1} S[i] \cdot x^i \right) \mod p + c \cdot x^{|S|} \mod p = (h(S) + c \cdot x^{|S|}) \mod p$$

Утверждение. Пусть $h[i] = h(S[:i+1])$, тогда

$$h(S[l:r]) = \frac{h[r-1] - h[l-1]}{x^l}$$

Алгоритм Рабина-Карпа.

1. Посчитаем полиномиальный хеш паттерна за $O(|P|)$.
2. Посчитаем массив префиксных хешей текста T за $O(|T|)$.
3. Переберем в тексте все подстроки размера $|P|$ (их $O(|T|)$) и для каждой сравним хеш с хешом P . Если не совпали, то точно в данном месте вхождения нет, иначе проверяем в лоб.

Теорема (б/д). Полиномиальная хеш-функция выше гарантирует, что вероятность коллизии не превосходит $\frac{|P|-1}{|P|}$

Следствие. Время работы в среднем: $O(|P| + |T|)$.

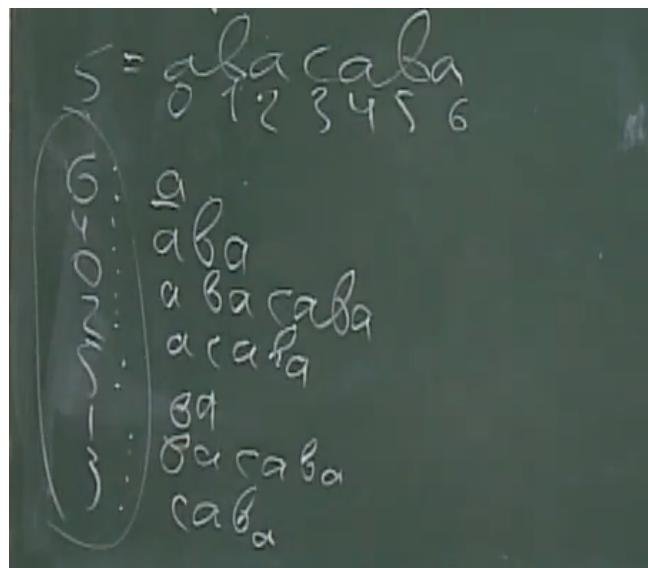
4. (1) Бор. Сравнительный анализ времени работы в зависимости от выбора контейнера для хранения дочерних вершин.

Определение. Бор - структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной.

Контейнер	Построение	Поиск	Память
vector	$O(\sum_{w \in dict} w)$	$O(S)$	$O(\Sigma \sum_{w \in dict} w)$
map	$O(\log \Sigma \sum_{w \in dict} w)$	$O(S \log \Sigma)$	$O(\sum_{w \in dict} w)$
hashmap	$O(\sum_{w \in dict} w)$	$O(S)$	$O(\sum_{w \in dict} w)$

5. (3) Суффиксный массив. Построение за $O(|S| \log |S|)$.

Определение. Суффиксный массив - индексы начал суффиксов, отсортированные в лексикографическом порядке.



Добавим в конец $\#$, зациклим и будем сортировать все подстроки.

Построение за $O(|S| \log |S|)$.

1. Сортируем все подстроки длины 1 (подсчетом).
 2. Знаем перестановку - массив p . Подсчитаем c - номера классов эквивалентности: идем по p , увеличиваем класс, если текущий символ в перестановке p не равен предыдущему.
 3. Пусть известна сортировка для 2^k . Отсортируем подстроки 2^{k+1} следующим образом:
 - (a) Разобьем подстроки длины 2^{k+1} на две части: первые 2^k символов и вторые 2^k символов.
 - (b) Каждую подстроку длины 2^{k+1} представим как пару $(c[i], c[i + 2^k])$.
 - (c) Отсортируем пары $(c[i], c[i + 2^k])$ лексикографически (подсчетом, сначала по второму, потом по первому).
 4. Пересчитаем p : $p[i] = p[i] - 2^k$. Повторяем 2-3. Только в шаге 2 классы эквивалентности не по строки, а по c .
 5. p - суффиксный массив
-
6. (3) Понятие LCP. Массив lcp. Алгоритм Касай-Аrimurы-Арикавы-Ли-Пака построения lcp (корректность б/д). Поиск LCP с использованием массива lcp и решения задачи RMQ.

Определение. LCP - longest common prefix. LCP(S, T) - длина наибольшего общего префикса строки S и T .

Определение. $lcp[i]$ - длина LCP i -го и $i + 1$ -го суффикса в лексикографическом порядке (суффмас).

Алгоритм Касаи-Аримуры-Арикавы-Ли-Пак.

Дана строка S и ее суффиксный массив p .

1. Находим pos - обратный к p .
2. Инициализируем $k = 0$. Для $i \in \overline{0, n - 1}$
 - $pos[i] == 0 \Rightarrow \text{continue}$
 - Иначе:
 - $i = p[pos[i] - 1]$
 - пока $s[i + k] == s[j + k] \Rightarrow k++$
 - $lcp[pos[i]] = k$
 - $k > 0 \Rightarrow k++$

Поиск LCP с использованием массива lcp и решения задачи RMQ.

Утверждение. Пусть $pos[l] < pos[r]$. Тогда

$$LCP(S[l :], S[r :]) = \min(lcp[pos[l]], lcp[pos[l] + 1], \dots, lcp[pos[r]] - 1)$$

7. (2) Суффиксные ссылки в боре. Функция перехода. Алгоритм Ахо-Корасик построения суффиксных ссылок.

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

Определение. Суффиксной ссылкой вершины u называют такую вершину $v = link(u)$, что $[v]$ является максимальным по длине суффиксом $[u]$, который можно прочесть, идя по бору из корня до, быть может, не терминальной вершины.

Замечание. Суффиксная ссылка для корня в общем случае не определена. Доопределим ее вершиной NIL.

Алгоритм Ахо-Корасик построения суффиксных ссылок.

Определение. Введем функцию $to(u, c)$, равную вершине, куда из вершины u можно перейти по букве c . Определяется она следующим образом:

$$to(u, c) = \begin{cases} \text{vertex}([u] + c), & \text{если из вершины } u \text{ есть переход по } c, \\ \text{to}(link(u), c), & \text{если из вершины } u \text{ нет перехода по } c. \end{cases}$$

Утверждение. Если переход из u по букве c есть в боре, то $\text{link}(\text{vertex}([u] + c)) = \text{vertex}[\text{to}(link(u), c)]$

Алгоритм.

1. Строим бор на данном наборе слов.
2. С помощью BFS из корня насчитаем функции $link$ и $to(\cdot, c)$ по утверждению выше.

Почему это работает? Из соотношения выше напрямую следует, что данные функции можно насчитать через вершины выше, чем текущая. А по предположению индукции мы верили, что для всех выше все посчитано корректно. База индукции - корень. Для него ручками посчитаем все по определению.

8. (2) Сжатые суффиксные ссылки. Автомат Ахо-Корасик. Поиск множества паттернов в тексте.

Определение. Сжатой суффиксной ссылкой вершины u называют такую вершину $v = \text{comp}(u)$, что

$$\text{comp}(u) = \begin{cases} \text{link}(u), & \text{если } \text{link}(u) \text{ терминальная,} \\ \text{comp}(\text{link}(u)), & \text{иначе.} \end{cases}$$

Поиск множества паттернов в тексте.

1. Построим Ахо-Корасик на паттернах за $O\left(|\Sigma| \sum_{i=1}^k |P_i|\right)$
2. Теперь будем идти по функции to текстом T из корня за $O(|T|)$.
3. При очередном переходе будем прыгать по сжатым суффиксным ссылкам вверх и насчитывать вхождения.

Время: $O\left(|\Sigma| \sum_{i=1}^k |P_i| + |T| + k\right)$, где k – кол-во вхождений.

9. (1) Понятие правого контекста. Классы эквивалентности по равенству правых контекстов. Теорема Майхилла-Нероуда (б/д). Определение суффиксного автомата.

Определение. Правый контекст слова x для языка L называют множество

$$R_L(x) = \{w | xw \in L\}$$

Определение. Два слова эквивалентны относительно языка L , то есть $x \sim_L y$, если $R_L(x) = R_L(y)$.

Теорема (Майхилла-Нероуда). *Если в языке L конечное число классов эквивалентности, равное k , то*

1. Язык автоматный и в любом ДКА, распознающем данный язык, хотя бы k состояний
2. Существует ДКА, распознающий L , в котором ровно k состояний, где каждое состояние отвечает за соответствующий класс эквивалентности

Определение. Суффиксным автоматом для строки S назовем минимальный по числу состояний ДКА, распознающий язык суффиксов L .

Замечание. Далее будем считать, что речь идет только о подстроках S , а вместо языка L будем обозначать эквивалентность по языку S , порожденному суффиксами строки S .

10. (1) Атрибуты состояний. Устройство класса эквивалентности.

Утверждение. Если $x \sim_L y$, то либо x суффикс y , либо y суффикс x

Доказательство. $\exists z : xz \in L, yz \in L$. □

Определение. $\text{longest}(v)$ - самая длинная строка, которую можно прочесть, дойдя до состояния v в суффиксном автомате.

Определение. $\text{len}(v)$ - длина $\text{longest}(v)$.

Определение. $\text{link}(v)$ - такая вершина, что в ней лежит самый длинный суффикс $\text{longest}(v)$, не лежащий в v .

Утверждение. Пусть v - вершина автомата. Тогда в v лежат $\text{longest}(v)$ и несколько её самых длинных суффиксов.

Доказательство. C - класс эквивалентности вершины v .

Пусть $u = \text{longest}(v)$. Тогда $\forall x \in C : R_L(x) = R_L(u) \Rightarrow x$ суффикс u .

Пусть y - самая короткая строка в C . Пусть w суффикс $u(|y| < |w| < |u|)$.

Тогда $R_L(u) \subseteq R_L(w) \subseteq R_L(y)$, а $R_L(u) = R_L(y) \Rightarrow R_L(u) = R_L(w) = R_L(y)$ □

11. Понятие longest . Критерий longest . Устройство ребер в суффиксном автомате.

Определения - см. предыдущий пункт.

Автомат состоит из:

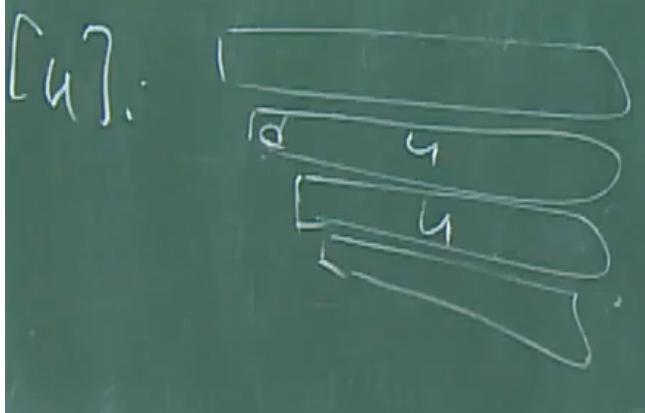
1. Вершины - классы эквивалентности.
2. Ребра - это тройки (C_1, C_2, d) .
3. В правом контексте начальной вершины - все суффиксы.
4. В правом контексте конечной вершины есть ϵ .

Утверждение. Если $u, v \in C_1 \Rightarrow ud, vd \in C_2$.

Доказательство. Пусть $z \in R_L(ud)$. Тогда udz - суффикс $s \Rightarrow dz \in R_L(u) = R_L(v) \Rightarrow z \in R_L(vd)$. \square

Утверждение (Критерий longest). *Пусть u - подстрока S . Тогда $u = \text{longest}([u]) \Leftrightarrow$ либо u - префикс S , либо $\exists a \neq b : au, bu$ - подстроки S .*

Доказательство. 1. Пусть $u \neq \text{longest}([u])$. Тогда $\exists d : du \sim_L u$. Значит, перед любым вхождением u в S стоит d .



2. Пусть $u = \text{longest}([u])$, u - не префикс S . Рассмотрим символы, предшествующие всем вхождениям $u \sim_L cu$. Противоречие.

\square

Следствие. Если u была longest в S , то она останется longest в Sc . Могут только появляться новые, гарантированно Sc .

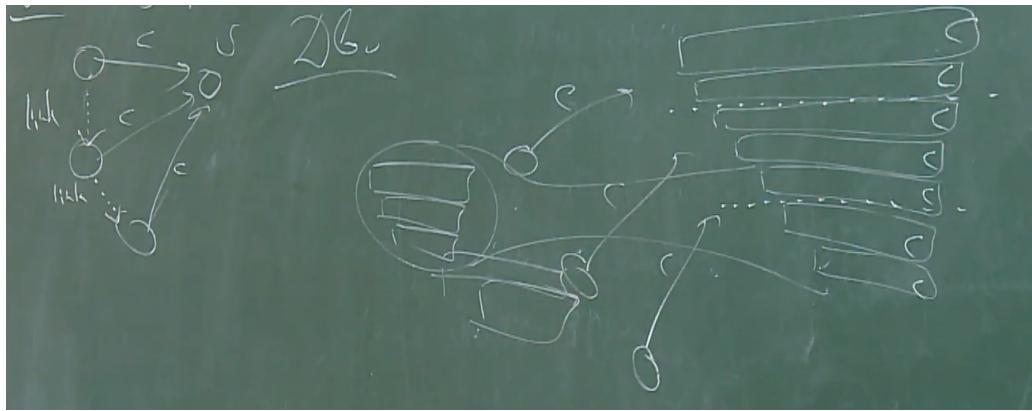
Утверждение. *Пусть в автоматае есть ребра $u_1 \rightarrow v, u_2 \rightarrow v, \dots, u_k \rightarrow v$ и $\text{len}(u_i) > \text{len}(u_{i+1})$. Тогда*

1. Все ребра идут по одной букве c .

2. $\text{link}(u_i) = u_{i+1}$.

Доказательство. Рассмотрим слова в классе v . Они заканчиваются на c .

Класс разбивается на ячейки $1 \dots k$. Каждой вершине u_i соответствует ячейка.



□

12. (2) Новые состояния в суффиксном автомате при дописывании символа. Алгоритм построения суффиксного автомата: случаи 1 (новая буква) и 2 (без порождения clone).

Классы эквивалентности \Leftrightarrow все longest.

Утверждение. При дописывании символа c строке S образуется не более двух новых состояний:

1. Состояние, у которого $\text{longest}(v) = Sc$, которое обязательно появится.
2. Состояние $[T]$, где $T = \text{longest}([T]_{Sc})$ - самый длинный суффикс Sc , который является подстрокой S (необязательно появится).

Доказательство. 1. Так как Sc - префикс Sc , значит, по критерию longest, $Sc = \text{longest}([Sc]_{Sc})$.

2. Если T не подстрока S , то она не могла быть рассмотрена ранее и обязательно попадет в $[Sc]_{Sc}$. Значит T - суффикс Sc , который является подстрокой S .

По критерию *longest*, чтобы T оказалась новым longest, у нее обязано появиться вхождение с новым предварением. Иначе нового состояния не возникнет в автомате.

□

Утверждение. $\text{link}([Sc]_{Sc}) = [T]_{Sc}$, где T из утверждения выше.

Алгоритм построения.

Алгоритм будет итеративным: перестраивание автомата при дописывании по одному символу. Потенциальные изменения:

1. Ребра, ведущие в $[Sc]_{Sc}$. (Ребер из нее быть не может по соображениям длины пути и строки Sc).
2. При возникновении нового состояния T , ребра, ведущие в/из него.

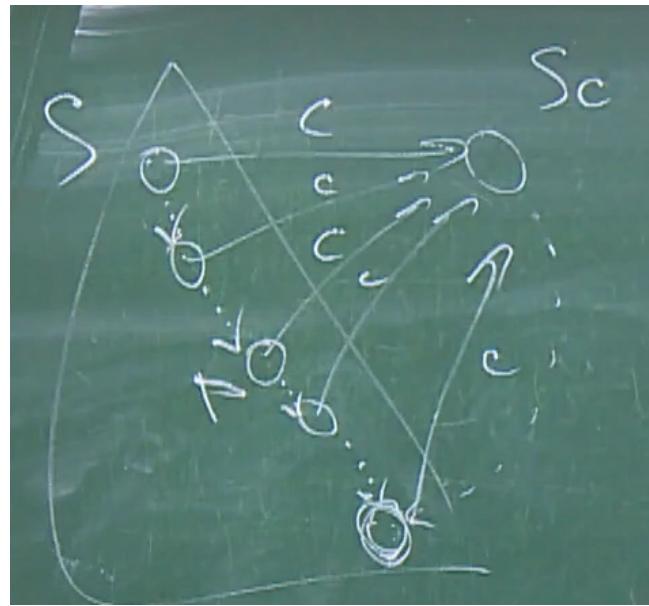
Осталось разобраться с тремя возможными ситуациями.

Случай 1 (новая буква).

Должно появиться ребро $([S]_{Sc}, c) \rightarrow [Sc]_{Sc}$.

По утверждению об устройстве ребер в одну вершину, все такие ребра бедутся из вершин, образующих суффиксный путь. Тогда, прыгая по суффиксным ссылкам из $[S]_S$, проводим ребра, пока не доберемся до вершины, из которой есть переход по c .

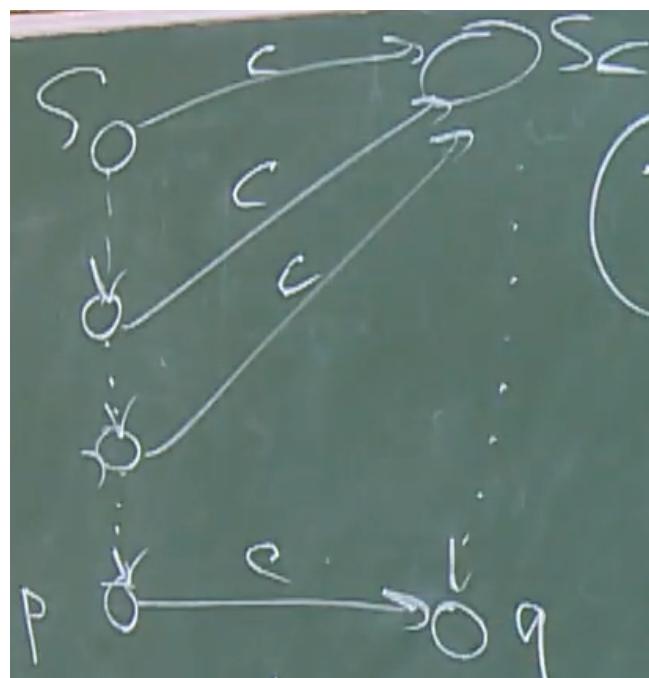
Допустим, что пришли в q_0 , из которого нет ребра по букве c , тогда не было буквы c в S и при этом $\text{link}([Sc]_{Sc}) = q_0$.



Случай 2 (без расщепления).

Пусть пришли в p , откуда есть ребро в q по c .

Из обозначений выше $T = \text{longest}(p) + c$. Вершину q надо будет расщепить, если $T \neq \text{longest}(q)$. Таким образом, критерий расщепления: $\text{len}(q) > \text{len}(p) + 1$. Допустим, что не надо расщеплять, то есть $\text{len}(q) = \text{len}(p) + 1$. Тогда достаточно провести $\text{link}([Sc]_{Sc}) = q$. На этом обработка окончена.



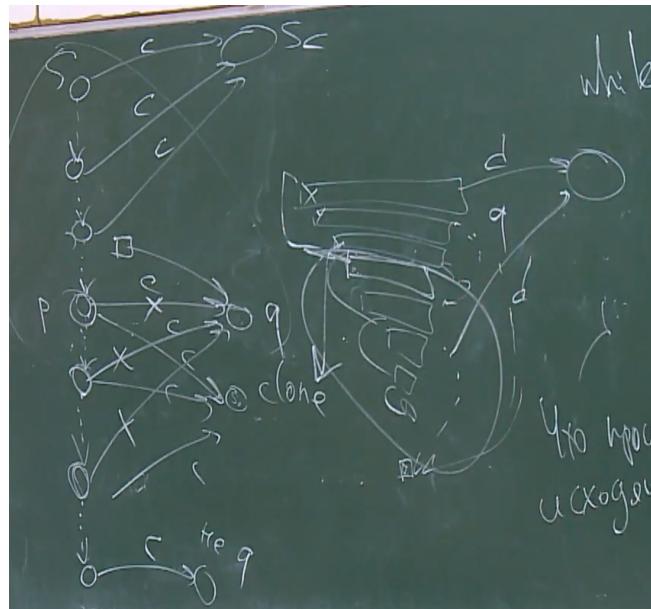
Случай 3 (с расщеплением).

Теперь $\text{len}(q) > \text{len}(p) + 1$. Тогда появится новое состояние clone такое, что $\text{longest}(\text{clone}) = T = \text{longest}(p) + c$, при этом в q будут лежать суффиксы $\text{longest}(q)$, которые длиннее T .

Рассмотрим суффиксный путь из p . Надо пропрыгать по нему, пока ребра по букве с шли в q , и перенаправить их в clone .

Заметим, что $R_{Sc}(\text{clone}) = R_{Sc}(q) \cup \{\varepsilon\}$, значит переходы из clone совпадут с переходами из q .

Остались ссылки. $\text{link}([Sc]_{Sc}) = \text{clone}, \text{link}(\text{clone}) = \text{link}(q), \text{link}(q) = \text{clone}, \text{len}(\text{clone}) = \text{len}(p) + 1$.



13. (3) Алгоритм построения суффиксного автомата: все случаи.

См. предыдущий пункт.

14. (2) Время построения суффиксного автомата.

Утверждение. При $|S| \geq 2$ число состояний в автомате не превосходит $2|S| - 1$.

Доказательство. Индукция по длине S . Для строки из двух букв число состояний равно трем. Далее дописывание одной буквы увеличивает не более чем на два. \square

Теорема (б/д). Начиная с некоторого n , в суффиксном автомате для строки длины n число ребер не превосходит $3n - 4$.

Заметим, что случаи первого и второго типов не меняют старых ребер, а только добавляют новые, которых $O(|S|)$. То есть они работают суммарно за $O(|S|)$.

Потенциал Φ - число вершин на пути по суфф. ссылкам от максимальной. Он уменьшается $\Rightarrow O(|S|)$.

15. (1) Задача интерполяции. Матричная постановка, существование и единственность решения. Комплексные корни из единицы. Обратная к матрице Вандермонда на комплексных корнях из единицы.

Задача 15.1. Известно, что многочлен $P(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1}$ в различных точках x_0, x_1, \dots, x_{n-1} принимает значения y_0, y_1, \dots, y_{n-1} . Надо восстановить многочлен $P(x)$, то есть найти a_i

Заметим, что перед нами система уравнений вида

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

Тогда данную систему уравнений можно записать в матричном виде

$$Xa = y,$$

где

$$y = (y_0, \dots, y_{n-1})^T, \quad a = (a_0, \dots, a_{n-1})^T, \quad X = (x_i^j)_{i,j=0}^{n-1}.$$

Формально решение можно записать в виде

$$a = X^{-1}y,$$

однако возникает вопрос: существует ли вообще матрица X^{-1} ?

Теорема (Вандермонда (б/д)). *Рассмотрим матрицу*

$$X = (x_i^j)_{i,j=0}^{n-1}.$$

Тогда

$$\det(X) = \prod_{0 \leq j < i < n} (x_i - x_j).$$

Определение. Комплексными корнями степени n из единицы называются решения уравнения $x^n = 1$.

Следствие. По основной теореме алгебры таких корней n

Утверждение. Комплексными корнями степени n из единицы равны $\omega_n^k = e^{\frac{2i\pi k}{n}}$

Рассмотрим задачу интерполяции, если известны значения многочлена

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

в комплексных корнях из единицы степени n ,

$$\omega_n^k, \quad k = 0, 1, \dots, n-1.$$

Рассмотрим матрицу W , которая будет фигурировать в матричном решении задачи интерполяции:

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

Утверждение. $W^{-1} = \frac{1}{n}V$, где $V = (\omega^{-ij})_{i,j=0}^{n-1}$

Доказательство. Посчитаем напрямую элементы матрицы WV и убедимся, что утверждение верно:

$$(WV)_{ij} = \sum_{k=0}^{n-1} W_{ik} V_{kj} = \sum_{k=0}^{n-1} \omega^{k(i-j)}.$$

Это геометрическая прогрессия, поэтому

$$\sum_{k=0}^{n-1} \omega^{k(i-j)} = \begin{cases} \sum_{k=0}^{n-1} \omega^0 = n, & i = j, \\ \frac{1 - \omega^{n(i-j)}}{1 - \omega^{i-j}} = \frac{1 - 1}{1 - \omega^{i-j}} = 0, & i \neq j. \end{cases}$$

Следовательно,

$$WV = nI_n.$$

□

16. Дискретное преобразование Фурье. Быстрое преобразование Фурье. Преобразование бабочки.

Определение. Пусть задан многочлен

$$P(x) = a_0 + a_1x + \dots + a_nx^n.$$

Тогда *дискретным преобразованием Фурье* называется вектор

$$\text{DFT}(P) = (P(\omega_0), P(\omega_1), \dots, P(\omega_{n-1}))^T,$$

где $\omega_k = \omega_n^k$, а ω_n - примитивный корень из единицы степени n .

Заметим, что преобразование Фурье можно записать в матричном виде как $Wa = y$, где a - вектор коэффициентов, а y - вектор, равный $\text{FFT}(P)$.

Таким образом, надо быстро научиться умножать матрицу на вектор. Рассмотрим три многочлена

$$P(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1},$$

$$P_0(x) = a_0 + a_2x + a_4x^2 + \dots,$$

$$P_1(x) = a_1 + a_3x + a_5x^2 + \dots.$$

Заметим, что

$$P(x) = P_0(x^2) + x P_1(x^2).$$

Следовательно, для вычисления значений $P(x)$ в корнях из единицы достаточно вычислить значения многочленов P_0 и P_1 в $n/2$ точках

$$\omega_0, \omega_2, \dots, \omega_{2(n-1)},$$

поскольку степени многочленов P_0 и P_1 в два раза меньше степени многочлена P .

Преобразование бабочки.

Мы получили стандартную процедуру по схеме «разделяй и властвуй». Осталось выразить $\text{DFT}(P)$ через $\text{DFT}(P_0)$ и $\text{DFT}(P_1)$.

Пусть

$$\text{DFT}(P_0) = (b_0, \dots, b_{n/2-1}), \quad \text{DFT}(P_1) = (c_0, \dots, c_{n/2-1}).$$

Первые $n/2$ значений вычисляются по формуле:

$$\text{DFT}(P)_k = b_k + \omega_n^k c_k, \quad k = 0, \dots, n/2 - 1.$$

Теперь разберём вторую половину:

$$\begin{aligned} \text{DFT}(P)_{n/2+k} &= P(\omega_n^{n/2+k}) \\ &= P_0((\omega_n^{n/2+k})^2) + \omega_n^{n/2+k} P_1((\omega_n^{n/2+k})^2) \\ &= P_0(\omega_n^{2k}) + \omega_n^{n/2} \omega_n^k P_1(\omega_n^{2k}) \\ &= b_k + \omega_n^{n/2} \omega_n^k c_k \\ &= b_k - \omega_n^k c_k, \end{aligned}$$

так как $\omega_n^{n/2} = -1$.

Таким образом, получили преобразование бабочки:

$$\text{DFT}(P)_k = \begin{cases} b_k + \omega_n^k c_k, & 0 \leq k < n/2, \\ b_{k-n/2} - \omega_n^{k-n/2} c_{k-n/2}, & n/2 \leq k < n. \end{cases}$$

Быстрое преобразование Фурье.

1. Дополним многочлен P нулевыми коэффициентами так, чтобы его степень была равна степени двойки.
2. Разбиваем вычисление $\text{DFT}(P)$ на рекурсивное вычисление $\text{DFT}(P_0), \text{DFT}(P_1)$.
3. Восстанавливаем ответ с помощью преобразования бабочки.

Время работы определяется рекуррентной $T(n) = 2T(n/2) + O(n)$, откуда время вычисления $\text{DFT}(P)$ составит $O(n \log n)$, где $n = \deg P$.

17. Обратное преобразование Фурье. Перемножение многочленов.

Обратное преобразование.

Вспомним, что

$$\text{DFT}(P) = Wa,$$

где a - вектор коэффициентов, а

$$W = (\omega^{ij})_{i,j=0}^{n-1}.$$

С помощью FFT умеем это делать за $O(n \log n)$.

В матричной постановке обратное преобразование имеет вид

$$W^{-1} \cdot \text{DFT}(P),$$

а уже доказано, что

$$W^{-1} = \frac{1}{n}V, \quad V = (\omega^{-ij})_{i,j=0}^{n-1}.$$

То есть нужно проделать всё то же самое, только заменить ω на ω^{-1} , и не забыть поделить на n в конце.

Перемножение матриц.

1. Считаем $\text{FFT}(A)$ и $\text{FFT}(B)$.
2. Посчитаем $C(\omega^k) = A(\omega^k) \cdot B(\omega^k)$
3. Вычислим обратное преобразование Фурье, то есть по $\text{DFT}(C)$ найдем C .

$$O((n+m) \log(m+n))$$

18. Свертка последовательностей. Расстояние Хэмминга. Алгоритм поиска вхождений паттерна в текст не более k расхождениями.

Определение. Пусть даны две последовательности

$$a = [a_0, \dots, a_{n-1}], \quad b = [b_0, \dots, b_{m-1}], \quad n \geq m.$$

Назовём *сверткой* $a * b = c$ последовательность

$$c = [c_0, \dots, c_{n-m+1}],$$

где

$$(a * b)_i = c_i = \sum_{k=0}^{m-1} a_{i+k} b_k.$$

Замечание. Перед нами скалярное произведение вектора b и всех подотрезков a как со скользящим окном.

Рассмотрим многочлены

$$A(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1}, \quad B(x) = b_0 + b_1x + \dots + b_{m-1}x^{m-1}.$$

Обозначим их произведение за

$$C(x) = A(x)B(x) = C_0 + C_1x + \dots + C_lx^l.$$

Рассмотрим, как устроены коэффициенты C :

$$C_0 = a_0b_0,$$

$$C_1 = a_0b_1 + a_1b_0,$$

⋮

$$C_{m-1} = a_0b_{m-1} + a_1b_{m-2} + \dots + a_{m-2}b_1 + a_{m-1}b_0,$$

$$C_m = a_1b_{m-1} + a_2b_{m-2} + \dots + a_{m-1}b_1 + a_mb_0,$$

⋮

То есть C_{m-1+k} почти совпадает с c_k , но $C_{m-1+k} = (a * b_R)_k$. То есть для вычисления свертки разворачиваем b , перемножаем многочлены и берем нужные коэффициенты.

Определение. Расстоянием Хэмминга для двух строк равной длины называют величину $\rho_H(S, T) = \sum_{i=0}^{|S|-1} I(S_i \neq T_i)$.

Задача 18.1. Необходимо в тексте T найти все вхождения паттерна P с точностью до k символов. То есть найти все подстроки S в T такие, что $\rho_H(S, P) \leq k$.

Алгоритм.

1. Для каждого $\sigma \in \Sigma$ построим вектор v_P^σ , где $(v_P^\sigma)_i = I(P_i = \sigma)$. Аналогично построим v_T^σ .
2. Подсчитаем $u_\sigma = v_\sigma^T * v_\sigma^P$ для каждого $\sigma \in \Sigma$. Заметим, что u_i^σ соответствует числу таких позиций, что ровно в u_i^σ позиций строки $T[i : i + |P|]$ и P совпадают по букве σ .
3. Подсчитаем $w = \sum_{\sigma \in \Sigma} u^\sigma$. w_i говорит о том, в скольких позициях совпадают $T[i : i + |P|]$ и P .
4. Если $w_i \geq |P| - k$, то позиция считается валидной.

Сложность алгоритма:

$$O(|\Sigma|(|P| + |T|) \log(|P| + |T|)) = O(|\Sigma||T| \log |T|).$$

19. Персистентные структуры данных. Частичная и полная персистентность. Персистентный стек.

Определение. Для структуры S назовем ее персистентной версией такую структуру $\text{pers}(S)$, что ее интерфейс совпадает с интерфейсом S и имеется операция $\text{Access}(k)$ - возможность обращаться к версии на момент k -го запроса изменения.

Определение. Структура частично персистентная, если результат $\text{Access}(k)$ дает read-only структуру.

Определение. Структура полностью персистентная, если результат $\text{Access}(k)$ дает доступ на изменение. Изменение применяется не только к версии, но и ко всем ее потомкам!

Замечание. Как можно заметить, частично персистентные структуры имеют граф версий в виде бамбука, тогда как полностью персистентные - дерево версий.

Персистентный стек.

Храним стек в виде дерева. Двигаем указатель. Ничего не удаляем.

1. $\text{pop}()$ просто сдвигает указатель назад
2. $\text{push}()$ просто порождает очередного ребенка в дереве версий

20. Персистентные структуры данных. Частичная и полная персистентность. Персистентный массив

См. предыдущий пункт.

Персистентный массив.

Строим ДО. Каждая версия - новый корень. Копируем только измененную ветку, на оставшиеся - ссылаемся.

21. Процедура Merge. Оптимальное время работы: оценка.

Пусть нам надо получить лишь позиции элементов, куда надо вставлять элементы второго массива в первый.

И пусть $N \gg M$, тогда нет смысла в классическом алгоритме, так как выгоднее сделать M бинпоисков за $O(\log N)$ времени каждый.

Нижняя оценка Merge

Построем решающее дерево $(A[i] <? B[j])$. Листьев - C_{N+M}^M , глубина - $\log C_{N+M}^M$.

Теорема (Формула Стирлинга - б/д). $N! \sim \sqrt{2\pi N} \left(\frac{N}{e}\right)^N$

Теорема (Нижняя оценка Merge). *Слияние двух массивов длины N и M , где $N > 2M$, работает за $\Omega(N \log \frac{M}{N})$ времени.*

Доказательство.

$$\begin{aligned}
\log_2 \binom{M+N}{M} &\sim \log_2 \frac{\sqrt{2\pi(M+N)} \left(\frac{M+N}{e}\right)^{M+N}}{\sqrt{2\pi N} \left(\frac{N}{e}\right)^N \cdot \sqrt{2\pi M} \left(\frac{M}{e}\right)^M} \\
&= \frac{1}{2} \log_2 \frac{M+N}{2\pi MN} + \log_2 \frac{(M+N)^{M+N}}{M^M N^N} \\
&= \frac{1}{2} \log_2 \frac{M+N}{2\pi MN} + \log_2 \left(\frac{M+N}{M}\right)^M + \log_2 \left(\frac{M+N}{N}\right)^N \\
&= \frac{1}{2} \log_2 \frac{M+N}{2\pi MN} + \log_2 \left(1 + \frac{N}{M}\right)^M + \log_2 \left(1 + \frac{M}{N}\right)^N \\
\log_2 \binom{M+N}{M} &\sim \frac{1}{2} \log_2 \frac{M+N}{2\pi MN} + M \log_2 \left(1 + \frac{N}{M}\right) + N \log_2 \left(1 + \frac{M}{N}\right).
\end{aligned}$$

Будем считать, что $N > 2M$. Тогда

$$M \log_2 \left(1 + \frac{N}{M}\right) = \Theta(M),$$

при этом первое слагаемое стремится к нулю при росте N, M .

$$\begin{aligned}
\log_2 \binom{M+N}{M} &\sim \frac{1}{2} \log_2 \frac{M+N}{2\pi MN} + M \log_2 \left(1 + \frac{N}{M}\right) + N \log_2 \left(1 + \frac{M}{N}\right) \\
&= \Theta(M) + N \log_2 \left(1 + \frac{M}{N}\right) = \Omega(N) + N \log_2 \left(1 + \frac{M}{N}\right) = \Omega\left(N \log \frac{M}{N}\right).
\end{aligned}$$

□

22. Процедура Merge. Оптимальное время работы: оценка б/д, алгоритм.

См. предыдущий пункт.

Galloping search.

Пусть k - позиция куда попадает ответ, а $p = 0$ - номер итерации.

1. Заведем указатель на нулевой элемент массива, сравним с искомым. Если все ок, то победа, иначе идем дальше.
2. Сдвинем указатель на 2^p . Если уже перескочили, то достаточно запустить обычный бинарный поиск на отрезке $[0, 2^p]$, иначе повтори этот шаг, увеличив номер итерации на единичку.

Время работы: $O(\log k)$.

Выше алгоритм, который вырождается в бинпоиск в одном экстремальном случае и в константу в другом.

Будем искать место вставки не бинарным, а галлопирующим поиском. Пусть k_i - место, куда вставили $a[i]$, тогда можно оценить время работы алгоритма следующим образом:

$$\begin{aligned} O\left(\sum_{i=1}^M \log k_i\right) &= O\left(M \cdot \frac{1}{M} \sum_{i=1}^M \log k_i\right) \\ &= O\left(M \log\left(\frac{1}{M} \sum_{i=1}^M k_i\right)\right) = O\left(M \log \frac{N}{M}\right). \end{aligned}$$

23. Inplace Merge Sort.

Хотим $O(1)$ доппамяти.

1. Сортируем правую половину, используя в качестве буфера левую - получаем массив C .
2. Сортируем первую четверть, используя в качестве буфера вторую четверть - получаем массив A .
3. Сливаем, чтобы элементы не перезатирались - получаем массив C длины $\frac{3N}{4}$:
 - Начинаем выписывать ответ со второй четверти
 - Пользуемся только swap, никаких присваиваний
4. Повторять шаги 2-3 до победного, только беря не четверти, а восьмые, шестнадцатые, etc.
5. Последний один элемент вставим за линейное время.

В алгоритме есть шаги двух видов: слияния и сортировки. Всего шагов $\log_2 N$.

- Слияние. Каждое работает за $O(N)$, их всего \log_2 штук, то есть $O(N \log N)$ времени.
- Сортировка. Сортируем массивы размерами $\frac{N}{2^k}, k \in \overline{1, \log_2 N}$.

24. Алгоритм Introsort.

1. Задаем два параметра: `max_depth` как функцию от N и `min_length`.
2. Если длина массива меньше `min_length`, то квадратичная сортировка (обычно сортировка вставками).
3. Если глубина рекурсии больше `max_depth(N)`, то запускаем HeapSort.
4. Иначе выбираем как-то быстро опорный элемент `pivot` (например, медиана из первого, последнего и серединного элементов). Далее `Partition` и рекурсивно левую и правую части сортируем.

25. Биномиальные деревья. Свойства: число вершин на уровне. Процедуры SiftDown, SiftUp и DecreaseKey в биномиальном дереве.

Определение. Биномиальное дерево ранга k T_k - корневое дерево следующего вида:

- Если $k = 0$, то это один корневой элемент
- Если $k > 0$, то это дерево T_{k-1} , подвешенное к такому же по структуре T_{k-1} .

В биномиальном дереве выполняется свойство кучи.

Свойства биномиального дерева:

1. В T_k 2^k элементов.
2. В T_k на i C_k^i элементов. $C_k^i = C_{k-1}^i + C_{k-1}^{i-1}$

SiftDown.

1. Выбираем с ребенка минимальным значением.
2. Если соотношение верно - останавливаемся.
3. Иначе - меняемся с корнем и рекурсивно запускаемся.

SiftUp.

Аналогично, пока вершина не корень - талкаем наверх.

`DecreaseKey`.

Надо уменьшить элемент - уменьшаем и `SiftUp`.

26. Биномиальная куча. Время работы слияния куч. Выполнение основных операций кучи.

Определение. Биномиальная куча - набор биномиальных деревьев попарно различных рангов с указателем на минимальный из корней.

Поиск минимума.

- Для поиска минимума достаточно найти наименьший корень.
- Функция будет работать за $O(1)$.

Слияние.

Даны две кучи размера N и M . Рассмотрим каждое из чисел в виде его двоичной записи.

Будем эмулировать сложение этих двоичных чисел начиная с младшего разряда.

При переносе разряда выполняется слияние двух деревьев одного ранга. А именно, T_k с большим корнем подвешивается к корню T_k с меньшим. Таким образом, получаем T_{k+1} или ту самую «единичку в уме».

Заметим, что все это выполняется за $O(\log(N + M))$ времени.

Вставка.

- Создаем кучу из одного биномиального дерева.

- Этую кучу можно слить с исходной.
- Работать это будет за $O(\log n)$.

Извлечение минимума.

- Ищем \min корень в списке корней.
- Заметим что его дети – это тоже биномиальные деревья.
- Нужно перевернуть список детей, и это будет биномиальной кучей.
- Сольем две кучи.

Время: $O(\log n)$.

27. Фибоначчиева куча. Топология узла. Структура кучи. Операции Insert, Merge, GetMin.

Определение. Фибоначчиева куча позволяет делать следующие операции:

1. GetMin за $O(1)$
2. Merge за $O(1)$
3. DecreaseKey за $O^*(1)$
4. ExtractMin за $O^*(\log n)$
5. Insert за $O(1)$

Топология.

Узел фибоначчиевой кучи состоит из элемента и двусвязного списка детей, на концы которого лежат указатели в узле. Также храним указатель на предка.

Если вершина не корневая, то в узле лежит еще указатели на соседние элементы в списке детей родительского узла (братья).

Фибоначчиева куча - двусвязный список почти биномиальных деревьев с поддержкой указателя на минимальный корень.

```
1 struct Node {  
2     T value;  
3     Node* parent;  
4     Node* left_brother;  
5     Node* right_brother;  
6     Node* children_start;  
7     Node* children_end;  
8     int rank; // number of children  
9     bool mark; // was one of children erased  
10};
```

Вырезание поддерева.

Вынесем вершину и все ее поддерево в конец списка корней. То есть удалим узел из списка детей родительской вершины и свяжем братьев друг с другом.

Insert.

Для вставки достаточно вставить в список корней новое дерево из одного элемента. Еще надо обновить указатель на минимум.

`GetMin.`

Храним указатель.

`Merge.`

Нужно сконкатенировать списки корней.

28. Фибоначчиева куча. Структура кучи. Операции `Consolidate`, `ExtractMin`. Оценка на $D(n)$

См. предыдущий пункт.

`Consolidate`.

Операция `Consolidate` нужна для того, чтобы сделать из списка элементов фибоначчиеву кучу.

Ранг дерева в фибоначчиевой куче - ранг (степень) корня. Запускаем процедуру аналогичную `Merge` в биномиальной куче: сливаем два дерева в одно рангом на 1 побольше.

Пусть $D(n)$ - максимально возможный ранг в куче на n элементах.

1. Заведем массив C размера $D(n)$, где $C[i]$ - указатель на корень дерева ранга i .
2. Пробегаемся по списку деревьев в куче и, если дерева такого ранга еще нет, то добавляем в массив указатель. Иначе сливаем деревья, пока нужно.
3. Еще поддерживаем указатель на минимальный корень.

Время работы: $O(D(n) + k)$, где k - число деревьев в куче. Число объединений $\leq k - 1$.

Отныне ранги всех корней различны.

Mark.

1. У корня всегда $mark = \text{false}$. Не важно, сколько детей у него вырезали.
2. Ставится в true , когда вершина - не корень, а у нее вырезается один из детей.

ExtractMin.

1. Получаем узел.
2. Удаляем вершину, а всех ее детей добавим в список корней (как в биномиальной куче).
3. У всех детей зануляем родителя, то есть $O(rank)$.
4. Обновляем указатель на минимум.
5. Вызываем `Consolidate`.

DecreaseKey.

Дан указатель на элемент, значение в котором надо уменьшить (куча на минимум). Значит может нарушиться неравенство с родительским узлом.

1. Вырезаем элемент, чье значение надо уменьшить, с поддеревом.
2. Меняем значение в новом корне, сбрасываем $mark$. Пересчитываем минимум.

3. Помечаем $\text{parent}.mark = \text{true}$. $\text{parent.rank} -= 1$.
4. Если у parent уже был $\text{mark} = \text{true}$, вырежем поддерево родителя в список корней и шаги 2-4 уже для него!

Время Consolidate.

На каждом корне лежит по монете, а на вершине с $\text{mark} = \text{true}$ по две монеты.

Тогда на каждое слияние в рамках `Consolidate` уйдет по монете и на новом корне монета останется.

Так что слагаемое k в $O(D(n) + k)$ «нивелируется»: монеты лишние не истрачены. Значит время `Consolidate`: $O^*(D(n))$.

Время DecreaseKey.

На каждом корне лежит по монете, а на вершине с $\text{mark} = \text{true}$ по две монеты.

Тогда на вырезание первой вершины уходит $O(1)$ действий - одна монета, еще одна монета останется на новом корне.

Когда закончим подъем, на последнюю вершину с $\text{mark} = \text{false}$ кладем две монеты.

Итого: $O(1)$ действий + 2 монеты, откуда выходит время работы: $O^*(1)$.

Оценка на $D(n)$.

$D(n)$ - максимальный ранг корня дерева на n вершинах. Решим обратную задачу: пусть максимальный ранг равен k , найдем $S(k)$ - минимальное число вершин в дереве ранга k .

$S(0) = 1, S(1) = 2$. Пусть у вершины v ранг равен k , значит у нее k детей u_0, \dots, u_{k-1}, u_i упорядочены по времени подвешивания к v .

Утверждение. Ранг u_i хотя бы $i - 1$.

Доказательство. В момент подвешивания u_i ранг v был хотя бы i , так как сливаем деревья одинакового ранга. А уменьшить ранг (вырезать дочерний узел) можно не больше чем один раз. \square

У детей массив рангов не меньше чем $[0, 0, 1, 2, \dots, k - 2]$. Значит

$$S(k) \geq 2 + \sum_{i=0}^{k-2} S(i)$$

Лемма. $S(k) \geq F_k$

Утверждение.

$$F_n = 1 + \sum_{i=0}^{k-2} F_i$$

Док-во по индукции.

$$S(k) \geq 2 + \sum_{i=0}^{k-2} S(i) \geq 1 + \sum_{i=0}^{k-2} F_i \geq F_k$$

29. Фибоначчиева куча. Структура кучи. Операции Consolidate, DecreaseKey. Оценка на $D(n)$.

См. предыдущий пункт.

30. Алгоритмы во внешней памяти. Устройство памяти и понятие I/O-операции. Понятия latency, throughput. Модель оценивания времени на основе I/O-операций. Модельная задача: сумма последовательности во внешней памяти.

Имеются три устройства: CPU, RAM и EM (External memory, диск) - HDD или SSD.

Объем RAM M порядка 100 гигов в мощных вычислителях, тогда как дисков N порядка 10 Тб.

В RAM-модели за одну операцию можно было считать одно машинное слово длины $w \in \{32, 64\}$ (длина регистра).

Модель диска.

1. Можно считать, что дисковая память - непрерывная лента, хотя это не так
2. Нет понятия машинного слова, так как чтение происходит не в регистры
3. Лента поделена на блоки размера $B >> w$
4. За одну операцию можно прочитать/записать один выровненный блок (I/O операция)

Определение. I/O-операция (Input/Output) - это передача блока данных между RAM и EM.

Определение. Latency (задержка) - время между запросом и началом передачи данных.

Определение. Throughput (пропускная способность) - скорость самой передачи данных.

Пусть N - размер входных данных, M - размер оперативной памяти, B - размер блока.

Сложность.

1. I/O операция объявляется элементарной
2. Сложность алгоритма - число I/O операций
3. Действия CPU с RAM не учитываются

Размер блока.

В силу устройства диска, I/O-операции состоят из трёх стадий:

1. вращение диска, скорость порядка 10^4 оборотов в минуту;
2. позиционирование головки;
3. собственно I/O-операция.

Первые два пункта занимают порядка 10 ms, а значит итоговое время работы равно $10 \text{ ms} + \frac{x}{\text{throughput}}$, где x - объём I/O-операции.

Однако в модели время работы равно $\frac{x}{B}$. Проблема в том, что здесь отсутствует аддитивное слагаемое, которое существенно влияет при малых объёмах данных.

Решение: запретить маленькие I/O-операции, так чтобы $\frac{x}{\text{throughput}} \geq 10 \text{ ms}$.

Тогда $B \geq 1 \text{ Mb}$, а на практике лучше брать порядка 10 Mb.

Сумма элементов.

Читаем блоками, суммируем. $O(\frac{N}{B})$.

31. Алгоритмы во внешней памяти. Модель оценивания времени на основе I/O-операций.

Сортировка во внешней памяти.

MergeSort.

Отличия:

1. В листьях висят блоки. Их сортируем во внутренней памяти.
2. Один уровень работает за $Scan(N) = \frac{N}{B}$

Сложность $\frac{N}{B} \log_2 \frac{N}{B}$.

Улучшение 1.

Давайте ограничим дерево не блоками по B , а величинами порядка M . Уже сложность $\frac{N}{B} \log_2 \frac{N}{M}$.

Улучшение 2.

Сливаем не по два, а по K массивов. Глубина $\log_K \frac{N}{M}$

Улучшение 3.

$$K = \frac{M}{B}$$

Теорема. Сортировка во внешней памяти работает за $\Omega\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}\right)$.

32. (a, b) -дерево. Операции SplitNode, Fuse, Share.

Определение. (a, b) -дерево, где $a \geq 2, b \geq 2a - 1$ - дерево поиска, которое удовлетворяет следующим требованиям:

1. У корня либо $[2; b]$ детей, либо 0.
2. У остальных элементов количество детей $[a; b]$ и число элементов в-node $\in [a - 1; b - 1]$.
3. Ключи в ноде упорядочены.
4. Все листья находятся на одной глубине.

Лемма. $h = O(\log_a n), h = \Omega(\log_b n)$

Доказательство. Скипнем первый уровень. Пусть на втором уровне k вершин. Тогда число элементов в дереве

$$n \geq \sum_{t=0}^h a^t k = \frac{k(1 - a^{h+1})}{1 - a} \sim k a^h \Rightarrow h = O(\log_a n)$$

□

SplitNode (расщипление).

Используется при переполнении узла (когда ключей стало b) - обычно при вставке.

1. Выбрать медиану.
2. Узел делится на левый и правый.

3. Медиана - в родителя.

4. Если родитель переполнен - рекурсивно вызываемся от него.

Fuse (слияние).

Используется при недостатке ключей (меньше $a - 1$) - обычно при удалении.

1. Берётся узел, его сосед (левый или правый) и разделяющий ключ родителя.
2. Все ключи объединяются в один узел.
3. Разделяющий ключ удаляется из родителя.
4. Если у родителя стало слишком мало ключей - операция продолжается вверх.

Share (перераспределение).

Операция балансировки, применяемая при удалении, когда в узле стало меньше $a - 1$ ключей, но соседний брат имеет больше минимума.

Рассмотрим с правым братом:

1. Разделяющий (с правым братом) ключ опускается в текущую вершину.
2. Минимальный ключ из правого брата поднимается в родителя.
3. Вершины переносятся из правого брата в текущую.

Аналогично с левым.

Поиск.

Спускаемся по дереву, выбираем нужного сына, бинпоиск.

Вставка.

1. Находим лист.
2. Вставляем ключ.
3. При переполнении - SplitNode.

Удаление.

1. Находим ключ.
2. Удаляем.
3. Пытаемся Share (длина узла $\geq a$), иначе - Fuse.

33. Алгоритмы во внешней памяти. Модель оценивания времени на основе I/O-операций. (a, b) - дерево. Основные операции с (a, b) -деревом.

См. предыдущий пункт.

34. Алгоритмы во внешней памяти. Модель оценивания времени на основе I/O-операций. BufferTree как (a, b) -дерево. Разбиение буфера на две части: основную и накопительную с родителями.

См. предыдущий пункт.

Определение. Buffer Tree - это $(\frac{M}{B}, \frac{4M}{B})$ -дерево, в каждом листе которого есть блок из B ключей. К тому же, в каждой нелистовой ноде есть буфер на $X \leq M$ элементов.

- Пока можем, буферизуем в памяти.
- Потом скидываем в буфер корня.
- Переполнили буфер - вызываем Flush:
 1. Считываем буфер в RAM.
 2. Сортируем запросы в RAM
 3. Скидываем запросы в соответствующие поддеревья (Scatter).

Проблема: Может быть, нам придётся делать Flush рекурсивно. Решение: к каждому узлу добавляем накоп (буфер побольше и отсортированный).

35. Задание полуплоскости. Пересечение полу плоскостей за $O(N^2)$.

Будем задавать полуплоскость неравенством $ax + by + c \geq 0$ и хранить как тройку (a, b, c) .

Тогда нормаль (a, b) смотрит в часть плоскости, содержащую полуплоскость.

Пересечение полуплоскостей за $O(N^2)$.

1. Изначально считаем, что результат - bounding box.
2. Добавляем полуплоскости по одной:
 - (a) Пересекаем полуплоскость с каждой стороной многоугольника и находим две точки P_i, P_j за $O(N)$.
 - (b) Все точки в порядке обхода между P_i и P_j удаляем за $O(N)$.

36. Пересечение полуплоскостей за $O(N \log N)$ через построение огибающих.

1. Разобьем полуплоскости на две группы по полярному углу ϕ нормали относительно оси OX : в первой группе $\phi \in [0, \pi)$, во второй: $\phi \in [\pi, 2\pi)$.
2. Внутри каждого класса сортируем по возрастанию ϕ . Если есть несколько паралельных прямых - оставляем одну с самым "сильным" неравенством.
3. В каждой группе: добавляем в порядке сортировки.
4. Пока пересечение не лежит в новой полуплоскости - удаляем последнюю.
5. Получили верхнюю и нижнюю огибающую - пересека скайлайном слева направо.

37. Пересечение полуплоскостей за $O(N \log N)$ без построения огибающих.

Аналогично предыдущему, но используем дек.

Замечание. 1. Если нормали двух полуплоскостей сонаправлены, то надо взять ту полуплоскость, у которой наименьший коэффициент c в уравнении $ax + by + c \geq 0$, единственное - надо не забыть сделать нормали единичными.

2. В момент опускания (останется в деке одна полуплоскость) надо проверить, что поворот против часовой от нормали в деке до нормали новой прямой меньше π , иначе пересечение пусто.

38. Диаграмма Вороного за $O(N^3)$ или за $O(N^2 \log N)$.

Определение. Пусть задан набор точек $P_1, \dots, P_n \in \mathbb{R}^2$ (сайты). Тогда локусом P_i назовем

$$\{X \in \mathbb{R}^2 \mid \forall j \neq i \ |P_i - X| < |P_j - X|\}$$

Или множество тех точек плоскости, которые ближе к заданной, чем к остальным по евклидовой метрике.

Определение. Диаграмма Вороного для заданного набора P_1, \dots, P_n - разбиение \mathbb{R}^2 на локусы для заданного набора.

Алгоритм за $O(N^2 \log N)$.

Для каждой точки P построим свой локус:

1. Проведем отрезки $[P, P_i]$, построим к каждому из них серединные перпендикуляры и выберем ту полуплоскость, в которой лежит P .
2. Пересекаем серперы.

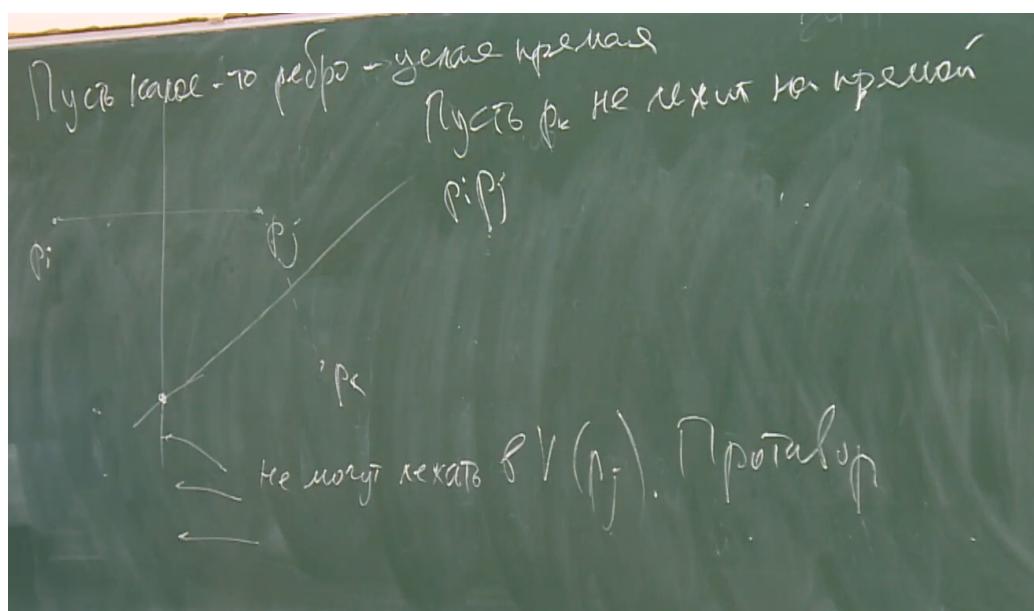
39. Диаграмма Вороного. Линейность числа ребер и граней.

См. предыдущий пункт.

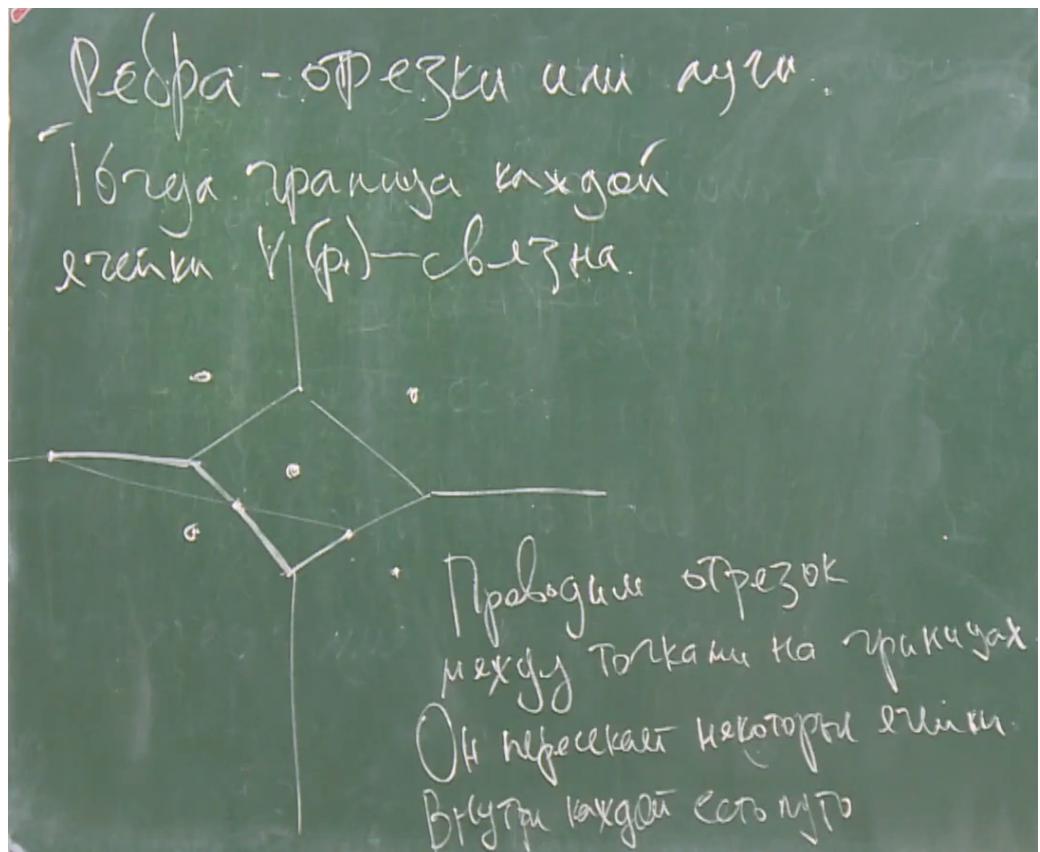
Утверждение. Пусть P_1, \dots, P_n - сайты. Тогда

1. Если все сайты лежат на одной прямой, то диаграмма Вороного - $n-1$ параллельная прямая.
2. Иначе, все ребра - либо отрезки или лучи и диаграмма связна.

Доказательство. Докажем, что ребра - либо отрезки или лучи.



Докажем, что граф связан.



□

Утверждение. В графе $\leq 2n - 5$ вершин и $\leq 3n - 6$ ребер.

Утверждение. Формула Эйлера: $|V| - |E| + |\Gamma| = 2$. Возьмем достаточно большой *bounding box* (*bbox*) и сузим диаграмму на него. Возьмем точку v_0 на бесконечном удалении и соединим все полубесконечные ребра с ней с сохранением планарности, верим, что можно?

Значит $|V| + 1 - |E| + N = 2$ или $|E| = |V| + N - 1$.

$\forall v \in V \deg(v) \geq 3$. Для внутренних очевидно. Теперь для v_0 . Пусть было только два бесконечных луча. Значит есть полуплоскость вне *bbox* не пересекающая эти лучи, то есть она содержится целиком в грани.

Значит $2|E| \geq 3(|V|+1)$. То есть $2|V| + 2N - 2 \geq 3|V| + 3$ или $|V| \geq 2N - 5$, отсюда $|E| \geq 3N - 6$.

40. Диаграмма Вороного. Критерии того, что точка является вершиной или лежит на ребре в диаграмме Вороного.

Определение. Пусть $\{P_1, \dots, P_N\}$ - набор точек и $Q \in \mathbb{R}^2$, тогда $C(Q)$ - круг максимального радиуса с центром в Q , не содержащий внутри себя точек из P .

Утверждение. 1. Q - вершина диаграммы Вороного $\Leftrightarrow C(Q)$ содержит на границе хотя бы три точки.

2. Кусок серпера l_{ij} к $P_i P_j$ является ребром диаграммы $\Leftrightarrow \exists Q \in l_{ij}$ такая, что $C(Q)$ содержит только P_i, P_j .

Утверждение. 1. \Rightarrow Если Q - вершина чьей-то грани, то ее степень хотя бы три, а значит она равноудалена от соответствующих точек.

\Leftarrow Раз внутри $C(Q)$ никого нет, а на границе P_i, P_j, P_k , значит $Q \in V(P_i) \cap V(P_j) \cap V(P_k)$, а это только вершина.

2. \Rightarrow Пусть l_{ij} содержит ребро, возьмем отрезок внутри ребра ненулевой длины и его середину Q . Построим круг с центром в Q , содержащий на границе P_i, P_j . Внутри никого нет, так как иначе Q ближе к этой точке.

\Leftarrow Напрямую из условия следует, что $\forall k \neq i, j \ dist(P_i, Q) = dist(P_j, Q) < dist(Q, P_k)$. Тогда внутри $\frac{\varepsilon}{2}$ окрестности Q вдоль l_{ij} нет других точек.

41. Диаграмма Вороного. Алгоритм Форчуна.

Сканирующая прямая π идет сверху вниз. Выше прямой диаграмма уже есть и по мере движения она эволюционирует.

Будем поддерживать диаграмму для тех точек, про которые уже все известно.

Диаграмма известна для тех точек X , что ближе к какой-то другой точке P_i , чем к π . Тогда граница области представляет из себя ГМТ равноудаленных от точки и прямой, а это парабола, где P_i - фокус, а π - директриса.

Определение. Береговая линия (beach line) - набор дуг парабол с фокусами в точках выше π , чья директриса π .

Будем хранить ее как дерево поиска по абсциссе фокуса.

В рамках движения π могут происходить события двух типов.

1. π достигла очередной точки диаграммы (событие точки, site event).
2. По мере движения π вниз какая-то из парабол схлопнулась между двух других (событие круга, circle event).

Обработка Site event.

1. Вставить новую параболу в береговую линию.
2. Посчитать для нее события круга с ее соседями.

При этом точки пересечения парабол удовлетворяют второму свойству границ и, следовательно, вычерчивают ребра диаграммы.

Обработка Circle event.

В событии круга две параболы зажали третью, тем самым два ребра пересеклись и породили третье.

Значит точка стала равноудалена от трех фокусов соседних парабол. То есть получаем новую вершину диаграммы Вороного.

Не забудем удалить схлопнутую параболу.

Так как вершина соответствует трем точкам P_i, P_j, P_k (последовательных в beach line), она равноудалена от них и от директрисы.

Найдем описанную около P_i, P_j, P_k окружность, тогда circle event появляется в момент прохождения π самой нижней точки окружности.

Если P_i, P_j, P_k на одной прямой, то параболы никогда не схлопнутся, так как линия пересечения двигаются вдоль параллельных серпиров.

Алгоритм Форчуна.

1. Сортируем точки по ординате и добавляем в кучу все site event.
2. Поддерживаем в ходе движения сканирующей прямой береговую линию.
3. Практически каждый (кроме случая одной прямой) новый site event порождает новый circle event, который добавляем в кучу.
4. Site event - добавление новой параболы в beach line.
5. Circle event - удаление параболы и добавление новых circle event для двух новых троек.
6. Если Circle event «протух» (параболы уже схлопнулись), то игнорим (роверяем, что его параболы существуют и последовательные).

Время работы.

1. Сортировка site events: $O(N \log N)$
2. Каждый site event порождает один circle event.
3. Каждый circle event порождает еще два circle event однако их всего $O(N)$.
4. Каждое событие обрабатывается за $O(\log N)$.

Итоговое время работы $O(N \log N)$.

42. Граф Делоне. Линейность числа ребер и граней в графе Делоне. Критерии того, что вершина/ребро будут в графе Делоне.

Определение. Пусть $S \subset \mathbb{R}^2$, $|S| = N$, тогда триангуляция S - максимальная по включению совокупность треугольников, вершины которых из S , пересекающихся по общим ребрам или общим вершинам.

Определение. Пусть $S \subset \mathbb{R}^2$, $|S| = N$, $Vor(S)$ - диаграмма Вороного, тогда граф Делоне:

- Вершины $V = S$
- Ребра P_i, P_j , если $V(P_i), V(P_j)$ - пересекаются по ребру положительной длины.

Утверждение. Пусть $|S| = N$ и K - число вершин на границе выпуклой оболочки, тогда в любой триангуляции S $2N - K - 2$ треугольника и $3N - K - 3$ ребра.

Доказательство. $|V| = N$, $\Gamma = M + 1$ (M треугольников и 1 внешняя), $|E| = \frac{3M+K}{2}$ Формула Эйлера: $N - \frac{3M+K}{2} + M + 1 = 2 \Rightarrow M = 2N - K - 2$ \square

Утверждение. 1. P_i, P_j, P_k являются вершинами одной грани графа Делоне $\Leftrightarrow \exists Q$ такая, что $C(Q)$ содержит их на границе. Более того, каждая грань графа Делоне - вписанный многоугольник

2. В графе Делоне есть ребро $P_iP_j \Leftrightarrow$ на серпине κP_iP_j такая, что $C(Q)$ содержит только P_i, P_j

Доказательство. См. аналогичное утверждение про диаграмму Вороного. \square

43. Независимость величины минимального угла от триангуляции граней графа Делоне. Два алгоритма построения: двойственный к диаграмме Вороного и модификация алгоритма Форчуна.

Определение. Триангуляция Делоне - произвольная триангуляция графа Делоне.

Замечание. Минимальный угол произвольно триангулированного вписанного многоугольника не зависит от триангуляции.

Алгоритм через диаграмму Вороного.

Пусть $S = \{P_1, \dots, P_N\} \subset \mathbb{R}^2$.

1. Строим диаграмму Вороного $\text{Vor}(S)$.
2. Проводим ребро триангуляции P_iP_j , если кусок серпера к P_iP_j участвует в диаграмме Вороного.
3. Дотриангилируем не треугольные грани графа Делоне.

Время: $O(N \log N)$.

Форчун-based алгоритм.

1. По мере алгоритма Форчуна если параболы с фокусами в P_i, P_j оказываются соседями в береговой линии, то проводим ребро графа Делоне.
2. Дотриангилируем не треугольные грани графа Делоне.

Время: $O(N \log N)$.

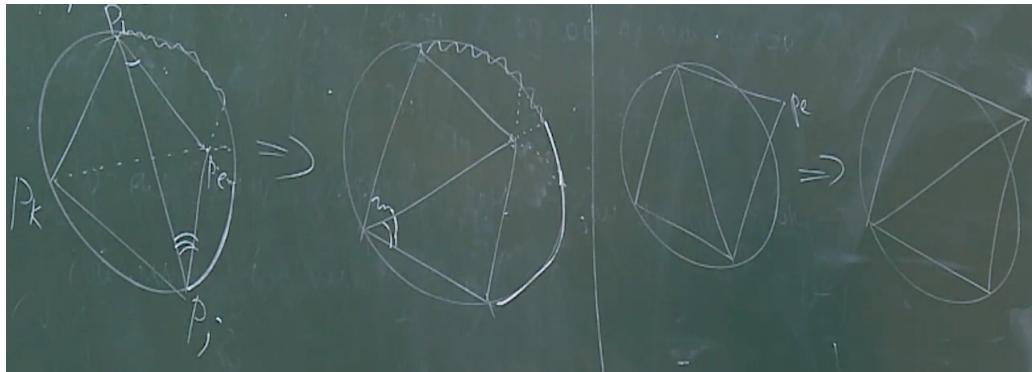
44. Легальное ребро. Критерий легальности ребра, триангуляции. Легальность триангуляции Делоне.

Определение. Рассмотрим ребро и смежные ему две грани $\triangle ABD$ и $\triangle ACD$ в триангуляции Делоне. Флип ребра - процедура замены диагонали AD четырехугольника $ABCD$ на диагональ BC .

Определение. Ребро нелегально, если флип ребра приводит к увеличению минимального угла.

Утверждение. Пусть $P_iP_jP_k, P_iP_jP_l$ - треугольники треангуляции. Ребро P_iP_j легально \Leftrightarrow окружность, описанная вокруг $P_iP_jP_k$ не содержит P_l .

Доказательство. Рассмотрим дуги:



□

Утверждение. Триангуляция легальна \Leftrightarrow триангуляция является триангуляцией Делоне.

Доказательство. \Leftarrow Предположим, что триангуляция легальна, но не является триангуляцией Делоне.

Тогда существует треугольник, в описанной окружности которого лежит некоторая точка P .

Рассмотрим путь от P к этому треугольнику через соседние треугольники триангуляции. Можно показать, что на этом пути обязательно найдётся ребро, для которого противоположная вершина лежит внутри описанной окружности соседнего треугольника.

Это означает, что такое ребро нелегально, что противоречит предположению.

\Rightarrow Так как триангуляция Делоне, ни одна вершина не лежит внутри описанной окружности соседнего треугольника.

Следовательно, данное ребро удовлетворяет локальному критерию Делоне, то есть является легальным. \square

45. 3D выпуклая оболочка. Алгоритм Джарви-са.

Определение. Выпуклой оболочкой $S \subset \mathbb{R}^3$ $\text{conv}(S)$ называется выпуклый многогранник минимального объема, содержащий все точки из S .

Теорема (б/д). Граф любого выпуклого многогранника планарен.

Алгоритм Джарвиса.

1. Находим грань P_i, P_j, P_k , заведомо являющуюся гранью оболочки.
2. Сложим ребра P_iP_j, P_jP_k, P_kP_i в очередь Q и запомним, что эти ребра уже были обработаны в множестве T .
3. Пока очередь Q не пуста:
 - Извлекаем из Q ребро AB
 - Находим крайние точки относительно ребра: C, D такие, что внутри двугранного угла $(ABC), (ABD)$ все точки

- Добавляем в очередь Q и в множество T ребра из $\{CA, CB, DA, DB\} \cap T$

Время.

Пусть K - число вершин выпуклой оболочки, тогда время работы $O(NK)$ - если грани только треугольные. А если нет?

Не треугольная грань может возникнуть, если окажется, что точки A, B, C, D на одной плоскости, тогда надо удалить ребро AB . Однако таких ребер суммарно $O(K)$, так как они являются диагоналями граней.

Значит время работы: $O(NK)$.

Утверждение (б/д). *Существует алгоритм, строящий выпуклую оболочку N точек в 3D за $O(N \log N)$.*

46. Построение триангуляции Делоне через 3D выпуклую оболочку.

Задача 46.1. Дан набор $S = \{P_1, \dots, P_N\} \subset \mathbb{R}^2$. Надо построить триангуляцию Делоне S .

Перейдем в 3D и $P_i = (x_i, y_i, 0)$. Спроектируем их на параболоид $z = x^2 + y^2$, то есть $\tilde{P}_i = (x_i, y_i, x_i^2 + y_i^2)$.

Алгоритм.

1. Построим выпуклую оболочку в 3D на точках $\{\tilde{P}_i\}_i^N$
2. Спроектируем нижнюю часть оболочки (нормаль к грани наружу направлена вниз относительно $z = 0$) на плоскость $z = 0$ - получим граф Делоне.
3. Дотриангулируем граф - получим триангуляцию Делоне.

47. Открытая адресация. Процедуры вставки, поиска. Артефакт tombstone. Время работы б/д.

Определение. Открытая адресация - способ реализации хеш-таблицы, при котором все элементы хранятся внутри массива, а при коллизиях ищется другая ячейка по некоторому правилу пробирования.

Определение. Функция пробы $g(x, i)$ определяет положение x в таблице с открытой адресацией, i - номер пробы.

Примеры функции пробы:

- Линейная пробирание: $g(x, i) = h(x) + i$.
- Квадратичное пробирание: $g(x, i) = h(x) + i^2$.
- Двойное хэширование: $g(x, i) = h_1(x) + ih_2(x)$.

Поиск.

1. Вычислим $pos_i = g(x, i) \% M$. Если в pos_i пусто, значит элемента
2. Иначе:
 - (a) Если в pos_i есть элемент и он равен x - нашли
 - (b) Если элемента нет, идем дальше к pos_{i+1}

Вставка.

1. Вычислим $pos_i = g(x, i) \% M$. Если в pos_i пусто, вставляем
2. Иначе:

- (a) Если в pos_i есть элемент и он равен x - закончили
- (b) Если элемента нет, идем дальше к pos_{i+1}

Удаление.

1. Вычислим $pos_i = g(x, i)\%M$. Если в pos_i пусто, закончили

2. Иначе:

- (a) Если в pos_i есть элемент и он равен x - вставляем `tombstone`
- (b) Если элемента нет, идем дальше к pos_{i+1}

Время.

Рассмотрим двойное хеширование. Логично, что для фиксированного хочется, чтобы $g(x, 0), \dots, g(x, N-1)$ образовывали перестановку от 0 до $N-1$. Как этого достичь?

Давайте считать, что число бакетов простое, тогда

$$\begin{aligned} \{g(x, 0), \dots, g(x, N-1)\} &= \\ &= \{h_1(x) + 0 \cdot h_2(x), \dots, h_1(x) + (N-1) \cdot h_2(x)\} = \\ &= h_1(x) + h_2(x) \cdot \{0, 1, \dots, N-1\} \end{aligned}$$

Данное множество равно $\{0, \dots, N-1\}$, так как сдвиг и умножение на взаимопростое не меняет систему вычетов.

Теорема (б/д). В варианте выше все операции работают за $O(1)$ в среднем.

48. Задача Perfect hashing. Решение для static версии. Алгоритм FKS. Доказательство времени работы для первого уровня.

Задача 48.1 (Perfect hashing). Структура данных, позволяющая проверять наличие элементов в множестве за $O(1)$ в худшем случае.

Разновидности:

1. Static - множество задано изначально.
2. Dynamic - можно вставлять элементы (необязательно за $O(1)$).

Решаем задачу static perfect hashing для множества на N элементах.

Хеш-таблица будет двухуровневой: хеш-таблица, где каждый бакет — хеш-таблица без коллизий.

1. Выберем хеш-функцию h_{out} из универсального семейства.
2. Пусть L_i — размер i -го из N бакетов.
3. Если $\sum_{i=0}^N L_i^2 > 4N$, то меняем функцию.
4. Для каждого бакета выберем хэш-функцию h_i из универсального семейства, число бакетов второго уровня L_i^2 .
5. Если h_i дала коллизию, то перевыбери.

Время построение (первый уровень).

Посчитаем матожидание суммы квадратов длинны цепочек:

$$\begin{aligned}\mathbb{E} \sum_{i=1}^N L_i^2 &= \mathbb{E} \sum_{i=1}^N \left(L_i + \frac{2L_i(L_i - 1)}{2} \right) = \mathbb{E} \sum_{i=1}^N (L_i + 2C_{L_i}^2) = \\ &= \mathbb{E} \sum_{i=1}^N L_i + 2\mathbb{E} \sum_{i=1}^N C_{L_i}^2 \leqslant \mathbb{E} N + 2 \frac{C_N^2}{N} = N + 2 \frac{N-1}{2} < 2N\end{aligned}$$

По неравенству Маркова

$$P \left(\sum_{i=1}^N L_i^2 > 4N \right) < \frac{2N}{4N} = \frac{1}{2}$$

Таким образом, число попыток подчинено распределению $Geom(p)$, где $p < 0.5$, откуда среднее число попыток $\leqslant 2$.

49. Задача Perfect hashing. Решение для static версии. Алгоритм FKS. Доказательство времени работы для второго уровня.

См. предыдущий

Время построение (второй уровень).

Теперь докажем, что если строить на L_i ключах таблицу на L_i^2 бакетов, то число коллизий мало. Пусть ε - число коллизий:

$$\begin{aligned}\mathbb{E}\varepsilon &= \sum_{x \neq y} \mathbb{E} I(h(x) = h(y)) = \sum_{x \neq y} P(h(x) = h(y)) < \sum_{x \neq y} \frac{1}{N} \\ \mathbb{E}\varepsilon &= \frac{C_{L_i}^2}{L_i} = \frac{L_i(L_i - 1)}{2L_i^2} < \frac{1}{2} \\ P(\varepsilon \geqslant 1) &\leqslant \frac{1}{2}\end{aligned}$$

50. k -независимое семейство хеш-функций. Пример. Обоснование отсутствия коллизий до взятия по модулю числа бакетов.

Определение. Семейство хеш-функций $\mathcal{H} = \{h : U \rightarrow \{0, \dots, n\}\}$ называется k -независимым, если $\forall x_1 \neq x_2 \neq \dots \neq x_n \in U, y_1, y_2, \dots, y_k \in \mathbb{Z}_n$

$$P_{h \in \mathcal{H}}(h_1(x_1) = y_1, \dots, h_k(x_k) = y_k) \leq \frac{1}{n^k}$$

То есть можно считать, что значения наборов размера k хеш-функций от ключа являются независимыми в совокупности.

Пример.

$$\mathcal{H} = h_{\alpha_0, \dots, \alpha_{k-1}}(x) = (\alpha_0 x^{k-1} + \alpha_1 x^{k-2} + \dots + \alpha_{k-1}) \mod p$$

Задача 50.1. 1. Заметим, что по набору x_i и y_i можно однозначно восстановить α_I , а значит на первом этапе нет коллизий. (вспоминаем FFT)

2. Всего наборов n^k .

$$\begin{cases} \alpha_{k-1} x_1^{k-1} + \alpha_{k-2} x_1^{k-2} + \dots + \alpha_1 x_1 + \alpha_0 = y_1 \\ \vdots \\ \alpha_{k-1} x_k^{k-1} + \alpha_{k-2} x_k^{k-2} + \dots + \alpha_1 x_k + \alpha_0 = y_k \end{cases}$$

$$\begin{pmatrix} x_1^{k-1} & x_1^{k-2} & \dots & x_1 & 1 \\ \vdots & \ddots & & \vdots & \\ x_k^{k-1} & x_k^{k-2} & \dots & x_k & 1 \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_{k-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_k \end{pmatrix}$$

$$\det X = \prod_{1 \leq i < j \leq k} (x_j - x_i)$$

$$\Rightarrow X^{-1} * Y \quad \text{if } \det X \neq 0$$

51. Cuckoo hashing. Время работы Insert б/д.

Определение. Cuckoo hash-table - пара хеш-таблиц на M бакетов каждая с хеш-функциями h_1, h_2 .

Данная хеш-таблица позволит решать dynamic perfect hashing.

Поиск.

1. Ищем в первой таблице. Не нашли - идем во вторую.
2. Ищем во второй. Не нашли - элемента нет.

Удаление.

Ищем, нашли - удаляем.

Вставка.

1. Пусть в H_1 в позиции $h_1(x)$ лежит элемент y , тогда положим y в H_2 по позиции $h_2(y)$.
2. Если же в H_2 в $h_2(y)$ лежит z , то переложим z в H_1 .
3. Повторить, пока не доберемся до ячейки без элемента.

Так делаем K итераций. Не помогло - идем в другую сторону не от $h_1(x)$, а от $h_2(x)$. Не помогло - Rehash.

Время работы.

Пусть $M \geq (1 + \varepsilon)N$, где M — число бакетов, а N — число элементов.

Теорема (б/д). Если ограничить число итераций при вставку как $3\lceil \log 1 + \varepsilon N \rceil$, то среднее время работы вставки составит $O(1)$.

52. Задача фильтра. Фильтр Блума. Процедура выбора гиперпараметров.

Определение. Фильтр — вероятностная структура данных, позволяющая по статическому множеству проверять, есть ли элемент в множестве с свойствами:

1. Если ответ от структуры «отсутствует», то элемента в структуре точно нет.
2. Если ответ от структуры «присутствует», то элемента в структуре может не быть.
3. Элементы не запоминаются для экономии памяти!!

Определение. FPR (false positive rate) — вероятность rate сказать ответ positive (элемент присутствует) и при этом сообщить ложь false.

Определение. Фильтр Блума — массив из M битов и k хеш-функций из k -независимого семейства хеш-функций.

Проверка наличия.

1. Вычислим $b_i = h_i(x)\%M$
2. Если все $b_i = 1$, то говорим, что элемент есть в множестве.

Считаем FPR.

Вероятность того, что i -й бит будет нулем по одной h_j : $1 - \frac{1}{M}$.

В силу k -независимости оценим грубо вероятность того, что i -й бит будет нулем: $(1 - \frac{1}{M})^k$.

Откуда для N элементов вероятность того, что i -й бит будет нулем: $(1 - \frac{1}{M})^{kN}$.

Откуда вероятность ошибки:

$$\left(1 - \left(1 - \frac{1}{M}\right)^{kN}\right)^k \approx \left(1 - e^{-\frac{kN}{M}}\right)^k$$

Оптимальное k : $k^* = \frac{M}{N} \log 2$. Откуда $FPR = \frac{1}{2^k}$.

53. XOR-filter. Алгоритм построения. Размер б/д.

Определение. Гиперграфом $H = (V, E)$ называют пару из конечного множества вершин V и конечного множества гиперребер E , где гиперребро — подмножество вершин или $\in 2^V$.

Определение. Степенью вершины v в гиперграфе H назовем число гиперребер, в которые v входит как элемент.

Определение. K -однородный гиперграф на N вершинах — гиперграф на N вершинах, чьи гиперребра содержат ровно K вершин каждое.

XOR-filter.

Введем массив B , в каждой ячейке которого хранится k -битное число **fingerprint**.

Пусть будет N элементов, тогда $|B| \geq 1.23N + 32$. Введем три хеш-функции

h_0, h_1, h_2 (из универсального семейства), каждая отображает ключ x в соответствующую третью диапазона от 0 до $|B|$.

Цель — построить такой массив B , что $\forall x \in S$.

$$B[h_0(x)] \oplus B[h_1(x)] \oplus B[h_2(x)] = \text{fingerprint}(x)$$

Заметим, что массив B задает собой вершины, а ключ $x \in S$ задает гиперребро на $h_i(x)$.

В случае универсального семейства получаем случайный 3-однородный гиперграф на $|B|$ вершинах с N ребрами.

А надо каждой вершине назначить битовые маски длины k , чтобы система была разрешима.

Построение.

1. Построим все гиперребра, вычислив $(h_0(x_j), h_1(x_j), h_2(x_j))$.
2. Проведем процедуру peeling
 - (a) Удаляем вершины степени ноль и кладем их в стек. Им рандомно назначим код.
 - (b) Удаляем вершины степени один и смежные ребра по очереди и кладем их в стек (декрементируя степени).
3. Если остались вершины степени 2 и выше, то у вашего гиперграфа имеется 2-ядро, делайте *rehash*.
4. Иначе вытаскиваем вершины из стека и решаем систему XOR-уравнений, ставя независимым переменным рандомные значения.

Определение. K -ядром в однородном R -гиперграфе назовем индуцированный подграф, полученный после peeling гиперграфа, состоящий из вершин степени $\geq K$.

54. XOR-filter. Алгоритм выбора решения.

См. предыдущий билет.