Матосновы алгоритмов

Кабашный Иван (@keba4ok)
на основе лекций А. С. Охотина и А. В. Тискина

22 января 2020 г.

https://users.math-cs.spbu.ru/~okhotin/teaching/algorithms_2020/ - источник с конспектами А. С. Охотина, где всё в тысячу раз подробнее и грамотнее.

Основные моменты.

Содержание

Билеты.

На каждом билте можно нажимать на то, куда вам надо (кроме того, что с первой и восьмой лекции, пришлите мне, пожалуйста, кто-нибудь, записи с них). К билету относится то, что от метки, по которой вы перешли, и до слова **подробнее**, ссылка рядом с которым перекинет вас на подробный конспект Охотина, который этот билет содержит (иногда билет на соседних лекциях, тогда два **подробнее**, соответственно).

Part 1.

1. Понятие алгоритма. Псевдокод. Хранение переменных при рекурсии. Машина с произвольным доступом к памяти.

2. Метод «разделяй и властвуй». Быстрое умножение Карацубы, анализ времени работы.

3. Сортировка вставкой, сортировка слиянием. Куча, действия над нею, сортировка кучей.

4. «Быстрая сортировка», среднее время работы.

5. Нижняя оценка числа сравнений при сортировке. Сортировка подсчётом. Поразрядная сортировка.

6. Нахождение i-го по величине элемента массива.

7. Метод динамического программирования. Задача о разделении стержня. Задача о порядке умножения матриц.

8. Нахождение наибольшей общей подпоследовательности: «народный» алгоритм, алгоритм Хиршберга.

9. Поиск в ориентированном графе: поиск в ширину, поиск в глубину, топологическая сортировка, нахождение компонентов сильной связности.

10. Кратчайшие пути в графе с весами: алгоритм Беллмана–Форда, алгоритм Дейкстры. Очередь с приоритетами и её реализация.

11. Нахождение минимального остовного дерева графа: алгоритм Прима, алгоритм Крускала.

12. Лес непересекающихся множеств, нахождение компонентов связности в неориентированном графе.

13. Кратчайшие пути в графе между всеми парами вершин. Транзитивное замыкание матриц. Алгоритм Варшалла. Кратчайшие пути с весами, алгоритм Флойда–Варшалла. Использование умножения матриц.

14. Быстрое умножение матриц, алгоритм Штрассена. (это я не стал переписывать, сразу смотрите в 5 лекции на странице 5)

15. Основная теорема о времени работы рекурсивных алгоритмов.

16. Быстрое умножение булевых матриц через числовые. Метод четырёх русских.

17. Структуры данных для представления множеств: вектор, список, двоичное дерево поиска. Основные операции (поиск, вставка, удаление, следующий, предыдущий, наибольший, наименьший), сложность их реализации. АВЛ-деревья. Реализация операций над ними, их сложность.

18. B-деревья. Реализация операций над ними, их сложность.

19. Полиномиальное хэширование строк. Алгоритм Рабина–Карпа. Нахождение наибольшей общей подстроки. Нахождение самого длинного палиндрома.

20. Алгоритм Кнута–Морриса–Пратта. Конечные автоматы, реализация алгоритма Кнута–Морриса–Пратта на конечном автомате.

21. Префиксное дерево. Алгоритм Ахо–Корасик. Его реализация на конечном автомате.

22. Префиксное дерево для множества строк. Построение упрощённого суффиксного дерева для строки. Суффиксное дерево и его применение.

23. Суффиксное дерево для строки, алгоритм Укконена.

24. Сжатие данных методом Хаффмана. Арифметическое кодирование.

`Билеты 22-24 см. в 8 лекции`

25. Сжатие данных со словарём, метод Лемпеля–Зива (LZ77, LZ78).

26. Преобразование Берроуза–Вилера, его реализация.

Part 2.

1. Первообразные корни из единицы. Дискретное преобразование Фурье (DFT), обратное DFT. Линейная, циклическая и косоциклическая свертки, их вычисление при помощи DFT. Умножение полиномов при помощи алгоритма Карацубы и DFT. Умножение целых чисел при помощи алгоритма Карацубы.

2. Быстрое преобразование Фурье (FFT). Общая шестиэтапная схема, схемы с прореживанием по времени и по частоте.

3. Преобразование DFT в кольце классов вычетов Z_N. Формулировка теоремы о выборе модуля N по заданной степени преобразования n и заданному первообразному корню ω из единицы; доказательство, что при таком выборе модуля значение n обратимо. Трудоемкость битовых операций FFT по модулю N, заданному теоремой.

4. Формулировка теоремы о выборе модуля N по заданной степени преобразования n и заданному первообразному корню ω из единицы; доказательство, что при таком выборе

модуля значение ω — первообразный корень из единицы.

5. Алгоритм Шенхаге-Штрассена для быстрого умножения двоичных чисел: сведение задачи умножения чисел к задаче вычисления косоциклической свертки в кольце классов вычетов.

6. Алгоритм Шенхаге-Штрассена для быстрого умножения двоичных чисел: вычисление косоциклической свертки по нечетному сомножителю модуля.

7. Алгоритм Шенхаге-Штрассена для быстрого умножения двоичных чисел: вычисление косоциклической свертки по четному сомножителю модуля; восстановление значения свертки по китайской теореме об остатках.

8. Вычисления схемами как модель параллельных вычислений. Схемы сравнения, слияния, сортировки. Принцип нулей-единиц. Эффективные схемы параллельной сортировки: сортировка нечетно-четным слиянием; сортировка битонным слиянием.

9. Задача префиксного накопления. Эффективная схема для параллельного решения этой задачи. Трудоемкость вычислений, коммуникации и синхронизации при решении этой задачи в модели барьерно-синхронного параллелизма (BSP). Параллельное вычисление обобщенных линейных рекуррентных соотношений и параллельное сложение двоичных чисел при помощи задачи префиксного накопления.

10. Задача линейного программирования (LP). Допустимые и оптимальные решения. Стандартная и каноническая форма задачи LP. Двойственная задача LP. Основная теорема линейного программирования (без доказательства). Общая идея симплекс-метода. Задача о кратчайших путях с одним источником как задача LP.

11. Задача о максимальном потоке в сети как задача LP. Алгоритм Форда-Фалкерсона.

12. Задача о минимальном вершинном покрытии. Ее приближенное решение при помощи LP-релаксации.

Part 3.

1. Алгоритм Фрейвальдса для проверки умножения матриц.

2. Проверка равенства полиномов. Лемма Шварца-Циппеля.

3. Randomized QuickSort.

4. Сравнение строк на расстоянии и алгоритм Рабина-Карпа.

5. Вероятностная проверка простоты: алгоритм Соловея-Штрассена.

6. Слабоэкспоненциальный детерминированный алгоритм для 3-SAT.

7. Слабоэкспоненциальный вероятностный алгоритм для 3-SAT, основанный на случайном блуждании.

8. Универсальные семейства хеш-функций, пример семейства (с доказательством). Хеш-таблица, основанная на универсальном семействе, оценка сложности операций.

9. Совершенное хеширование.

10. Линейный вероятностный алгоритм для минимального остовного дерева (без алгоритма верификации).

11. Оценки Чернова (Chernoff), без доказательства (достаточно самой простой формулировки). Уменьшение вероятности двусторонней ошибки.

12. Передача пакетов в кубической сети.

13. Онлайн-алгоритм MARKER для задачи кеширования (paging).

14. Алгоритм для онлайн-варианта задачи о покрытии множествами (случай единичных весов).

15. Алгоритм для онлайн-варианта задачи о покрытии множествами (общий случай).

Лекция 1.

Билет 1.

Определение 1. *Алгоритм* - понятие, которое мы формально не определили, но согласно википедии, это - конечная совокупность точно заданных правил решения некоторого класса задач или набор инструкций, описывающих порядок действий исполнителя для решения определённой задачи.

Определение 2. *Псевдокод* - удобный способ записи алгоритма, который, конечно, неполне формален.

Утверждение 1. Рассмотрим машину с произвольным доступом памяти (RAM). Это - абстрактная модель вычисления, которая состоит из ячеек с памятью (рисуем их в столбик и нумеруем целыми числами). Программа состоит из команд, и каждая команда имеет номер (скажем вообще, что они называются x_k). Пусть первая команда у нас пересылать что-то куда-то (*оператор присваивания* $A = B$, где B может быть константой, прямой адресацией (какая-то ячейка), или косвенной адресацией x_{x_n}, а A может быть всем этим же, кроме константы). Следующее, что нам нужно арифметические действия $A = B * C$, где $* \in \{+, -, \cdot, /, \div\}$.

Добавим также безусловный переход GOTO n - переход к строке программы, аналогично можно задать GOTO x_n, которая может пересылать к строчке команды, которая имеет номер, хранящийся в этой ячейке. Также нам нужен условный оператор IF; THEN, и в конец добавим оператор HALT, которая будет говорить остановиться, и в целом, всё, что можно, мы уже умеем делать с помощью такой тривиальной вещи.

Алгоритм 1. *Рассмотрим рекурсию, а конкретнее, как мы храним значения. Стэк - абстрактная структура данных, у которой есть дно, вершина, и мы всегда можем положить элемент наверх, или вынести его. Для рекурсии стэк состоит из нескольких stack-frame-ов, которые соответствуют каждому вызову рекурсивной программы, которые могут содержать ячейки со значениями, номерами строк или номерами вызова.*

Билет 2.

Алгоритм 2. ***Быстрое умножение Карацубы.*** *Система идей, которая позволяет значительно ускорить умножение (скажем, двоичных) чисел.*

Идея первая: при умножени двузначных чисел, вместо того, чтобы совершать 4 умножения и нескольких сумм, совершим 3 умножения и несколько сумм и разностей. Пусть у нас есть числа a_1a_0 и b_1b_0. Тогда рассмотрим $(a_1 + a_0)(b_1 + b_0)$ и вычтем $(a_1b_1 + a_0b_0)$, в итоге получим, что из результатов у нас выйдет собрать искомую сумму четырёх изначальных произведений.

Идея вторая: вообще, как мы могли заметить, нигде сильно нас не волновало, в какой системе счисления мы работали. Расмотрим большие числа $a_{n-1} \dots a_{\frac{n}{2}} a_{\frac{n}{2}-1} \dots a_0$ и $b_{n-1} \dots b_{\frac{n}{2}} b_{\frac{n}{2}-1} \dots b_0$ (двоичные), и разделим каждое из них пополам. Будем считать половины цифрами, тогда вот мы и свели к первой идее.

Наконец, идея третья: для умножения $\frac{n}{2}$-значных чисел мы используем тот же самый алгоритм, вызывая его рекурсивно.

Доказательство. Пусть у нас есть 2 n-значных числа, которые алгоритм K принимает на вход. Тогда у нас есть C_1 - K от первых половин, C_2 - K от вторых половин, и C_3 - K от сумм цифр пары чисел. В итоге мы получаем $D = C_3 - C_1 - C_2$, и ответ C_2DC_1. Нам нужно понять что-то про $T(n)$ - время на n-значных числах, $T(n) = 3T(\frac{n}{2}) + O(n)$. 3^i раз

мы вычисляем K от $\frac{n}{2^i}$-значных чисел, суммарно по этажам мы используем $\sum_{n=0}^{\log_2 n} C3^i \frac{n}{2^i}$, тчо можон допреобразовать к $Cn\frac{(\frac{3}{2})^{\log_2 n+1}-1}{\frac{3}{2}-1} = Cn^{\log_2 3}$. □

Алгоритм 3. *В ходе рассуждений мы коснулись идеи* ***метода "разделяй и властвуй"****, который состоит в том, что мы разбиваем большую задачу на несколько подзадач, которые ещё и можно распараллелить, если так можно, и таким образом, решать меньшие подзадачи вместо решения одной объёмной.*

Билет 3.

Алгоритм 4. ***Сортировка вставкой****. Пусть у нас есть массив a_i. Рассмотрим первый элемент, он отсортирован. Рассмотрим теперь два элемента; они либо отсортированы, либо нет, тогда мы их меняем. И теперь работаем, фактически, по индукции.*

Утверждение 2. Работает алгоритм за время n^2, причём зависит он от входных данных довольно сильно.

Алгоритм 5. ***Сортировка слиянием****. Разбиваем массив на две части, после чего отсортируем каждую половину и сольём. Делается это с помощью дополнительной памяти, ещё одного массива такого же размера, в который мы будем заносить итоговый результат слияния. Для самого слияния нам также нужны будут нужны указатели, которые двигаются и показывают на элементы, которые мы сравниваем. Тот, который получился меньше, заносится в новый массив, и указатель над ним сдвигается.*

Утверждение 3. Работает алгоритм за $n \log n$, раз уж $\sum_{i=0}^{\log_2 n} C2^i \frac{n}{2^i} = cn \log_2 n$.

Лекция 2.

Быстрая сортировка.

Алгоритм 6. ***Быстрая сортировка****. Выбираем опорный элемент, с которым сравниваем все остальные элементы (на это уходит линейное время). Затем рекурсивно работаем с тем, что справа от него и слева от него.*

Теорема 1. *Если все элементы массива различны и опопрный элемент выбирается случайно, то среднее время работы алгоритма* - $\Theta(n \log n)$.

Доказательство. Время работы пропорционально числу сравнениий между элементами. Расматриваем два элемента y_i и y_j, $i < j$, тогда они сравниваются только, если выбран один из них в качестве опорного. Если будет выбран какой-то y_k, $i < k < j$, то они никогда больше не будут сравнены, если что-то на отрезке не между ними - плевать, относительно отрезка между ними ничего не поменялось. Тогда среднее количество сравнений между этими элементами:

$$\frac{2}{j-i+1}.$$

Тогда всего среднее количество сравнений:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-1} \frac{2}{k+1} = O(n \log n).$$

Последняя оценка получается из

$$\sum_{k=1}^{n} \frac{1}{k} \approx \int_1^n \frac{1}{x} dx = \ln n.$$

□

Подробнее: 2 лекция, страница 1

Сортировка кучей.

Определение 3. *Куча* - двоичное дерево, представленное в виде массива. В этом дереве различаются левый и правый потомок каждой вершины, и потому вершины на каждом уровне упорядочены. Дерево почти сбалансировано: на каждом i-м уровне, кроме, может быть, последнего, есть все 2^i вершин.

Алгоритм 7. ***Сортировка кучей***. *Для начала, мы строим дерево: записываем по порядочку все вершины так, что у вершины x_i потомки - x_{2i} и x_{2i+1}. Затем начинаем на все вершины, кроме висячих смотреть и делать вот что: если она меньше потомка, то поменяем с ним (если меньше обоих, то с меньшим). Так доведём её до куда сможем, и продолжим рпссмотрение для оставшихся невисячих вершин (в изначальном дереве). Так сверху окажется наименьшая вершина, вынесем её, затем - по индукции.*

Утверждение 4. Работает за $O(n \log n)$ (построение дерева - $O(\log n)$, вынесение вершин - $O(n)$), можно разогнать оценку до $2n$.

Подробнее: 2 лекция, страница 3

Скорость сортировки.

Теорема 2. *(**Нижняя оценка скорости**).Всякий алгоритм сортировки, основанный на сравнении, требует $\Omega(n \log n)$ операций сравнения.*

Доказательство. Построим дерево того, как мы спускаемся к определению последовательности, его высота ограничивается $\log_2 n!$, оценим факториал $\left(\frac{n}{e}\right)^n < n! < n^n$, а после - оценим через это логарифм $n \log_2 n - O(n)$. □

Алгоритм 8. ***Сортировка подсчётом***. *Если у нас есть массив из конечного обозримого количества типов элементов, можно сначала посчитать количество первого, затем количество второго, и так далее. Время работы - $O(n + k)$, где k - количество типов, n - количество переменных.*

Алгоритм 9. ***Поразрядная сортировка***. *Сортируем числа сначала по первому разряду, затем по второму, и так далее... Время работы: $O(l(n + k))$, где сравниваются строки длины l, алфавит из k символов.*

Подробнее: 2 лекция, страница 4

Нахождение i-го по величине элемента массива.

Алгоритм 10. ***Нахождение i-го элемента.*** *Делим массив на пятёрки подряд идущих элементов (возможно, последняя пятёрка будет неполной). Теперь в каждой пятёрке выделяем медианы, и смотрим на медиану медиан. Сделаем её опорным элементом и как в быстрой сортировке, раскидаем всё по сторонам. Если этот элемент под номером i, то мы его нашли, иначе - действуем рекурсивно с одной из сторон. Время работы - линейное.*

Подробнее: 2 лекция, страница 8

Метод динамического программирования.

технические трудности

Задача 1. Имеется стержень длины n. Продав стержень длины i, можно выручить p_i денежных единиц. Как выгоднее всего распилить имеющийся стержень?

Алгоритм 11. *Начинаем с первого, и делаем полный перебор. Говнище*

Алгоритм 12. ***Жадный алгоритм.*** *Отпиливаем самый дорогой кусок, затем опять самый дорогой из возможных, и так далее. Не самый оптимальный.*

Алгоритм 13. ***Метод динамического программрования.*** *Суть этого метода такова. Пусть на каждом шаге надо сделать выбор (принять решение). Известно, что какой-то выбор приводит к оптимальному результату. Этому выбору соответствует некий набор подзадач. Тогда сперва находятся ответы для всех подзадач данной задачи, возникающих при различном выборе, после чего, имея все эти ответы перед глазами, можно будет в каждом случае сделать наилучший выбор.*

Пример(ы) 1. Пусть стержень длины 0 не стоит нисколько. Для j от 1 до n пока не найдено никаких способов продать стержень, для всякой длины отрезаемого куска, сложим его цену с выручкой за остаток. Если так можно выручить больше известного, то цена стержня длины j улучшается, и так рекурсивно мы дойдём до получения цены за весь стержень.

Подробнее: 2 лекция, страница 11

Лекция 3.

Продолжение динамического программирования.

Задача 2. Пусть нужно умножить n матриц $M_1 \times \ldots \times M_n$. В силу ассоциативности, скобки можно расставить как угодно. От их расстановки зависит общее число операций, и, чтобы умножить матрицы быстрее, надо заранее определить наилучший порядок их умножения.

Пример(ы) 2. Строим верхнедиагональную матрицу T, в которой $T_{i,j}$ - наименьшее число действий, необходимых для вычисления $M_{i+1} \times \ldots \times M_n$.

Внешний цикл по длине куска $l = j - i$, второй - по i, во внутреннем перебируются все разбиения произведения на два, и вычисляется следующее значение:

$$T_{i,j} = \min_{k=-+1}^{j-1} (T_{i,k} + T_{k,j} + m_i m_k m_j).$$

Разбираем так все по порядку и вычисляем наилучший способ. Время раюоты: $O(n^3)$ - строим таблицу, далее - $2n-1$ вызовов процедуры перемножить (i,j), в каждом - $O(n)$ итераций цикла. И ещё само умножение матриц.

Примечание 1. Для простого понимания - простой пример с кузнечиком, который прыгает на 1 или 2, и ему нужно пропрыгать n, сколькими способами это можно сделать? Мы заводим массив $dp[i]$ длины n (кол-во способов добраться до i), тогда $dp[i] = dp[i-1]+dp[i-2]$, и так насчитываем все значения, находим ответ для n.

Подробнее: 3 лекция, страница 1

Нахождение наибольшей общей подпоследовательности.

Определение 4. *Строкой над алфавитом* Σ называется всякая конечная последовательность $w = a_1 \dots a_l$, где $l \geq 0$, и $a_1, \dots, a_l \in \Sigma$ - символы.

Алгоритм 14. ***Народный алгоритм***. *Динамический способ нахождения наибольшей общей подпоследовательности. Заводим таблицу T и в ячейке $T_{i,j}$ записываем длину наибольшей общей подпоследовательности на префиксах длины i и j первого и второго слова соответственно. Заполняем таблицу последовательно от более коротких мар до самых длинных, и в итоге получим ответ в задаче.*

Строим таблицу так: берём $T_{i,j}$. Если у них одинаковые последние элементы, то получим $T_{i-1,j-1}+1$. Если они разные, то $\max(T_{i-1,j}, T_{i,j-1})$.

Саму последовательность элементов потом восстанавливаем с конца понятно как. Недостаток в том, что чтобы найти подпоследовательность, нужно хранить всю таблицу, а это $O(mn)$, и это много. Однако, если нужна только длина, то можно ограничиться лишь двумя столбцами (или двумя строчками).

Алгоритм 15. ***Алгоритм Хиршенберга***. *Построение наибольшей общей подпоследовательности за время $O(mn)$, используя память $O(\min(m,n))$. Пусть $u = u'u''$ - некоторое разбиение u. Тогда оптимальное совмещение u и v совмещает u' с каким-то начальным куском v - пусть это v', и u'' - с остатком v''. Нужно найти это разбиение $v = v'v''$, чтобы потом отдельно запустить совмещение двух соответствующих пар кусков.*

Алгоритм делит u на две подстроки примерно равной длины. Сперва динамическим программированием находится последняя строчка таблицы $T^{u',v}$, как в базовом алгоритме. Её j-ый элемент содержит длину наибольшей общей подпоследовательности u' и u_j - префикса длины j. Аналогично находится последняя строка таблицы $T^{(u'')^R,v^R}$ (R - reverse). Дальше складываем таблицы поэлементно и ссмотрим, где достигается максимум - это и есть искомое разбиение $v = v'v''$. Для этого вычисления алгоритс использовал $O(|v|)$ ячеек памяти, которые теперь можно освободить.

Далее алгоритм вызывается рекурсивно, чтобы вычислить лучшее совмещение u' и v', и u'' и v''. Полученные совмещения последовательно приписываются друг к другу.

Теорема 3. *Алгоритм Хиршберга работает за время $O(mn)$.*

Доказательство. Принимая за единицу времени время, затрачиваемое на вычисление значение одного элемента $T_{i,j}$ в "народном" алгоритме, утверждается, что в общей сложности будет выполнено не более, чем $2mn$ шагов.

Пусть $f(m,n)$ - время работы в наихудшем случае. Тогда индукцией по m и n доказывается неравенство $f(m,n) \leq 2mn$. При запуске на строках u и v, где их мощности соответственно равны m и n, вычисление таблицы $T_{u',v}$ займёт $\frac{1}{2}mn$ шагов, и за столько

же шагов будет вычисляться таблица $T^{(u'')^R,v^R}$. После этого проводятся два рекурсивных вызова, один из которых занимает $f(\frac{m}{2},k)$ шагов, а другой - $f(\frac{m}{2},n-k)$ шагов, для некоторого k. Время раюоты рекурсивных вызовов оценивается по предположению индукции, откуда получается оценка того же вида для $f(m,n)$.

$$f(m,n) = 2 \cdot \frac{1}{2}mn + \max_k(f(\frac{m}{2},k) + f(\frac{m}{2},n-k)) \leq mn + \max_k(mk + m(n-k)) = 2mn$$

□

Подробнее: 3 лекция, страница 3

Поиск в ориентированном графе.

Алгоритм 16. ***Поиск в ширину (BFS)***. *В каждый момент времени вершина графа может быть помечена или не помечена. Если вершина уже помечена, значит алгоритм нашёл путь из корня в неё. Кроме пометок на вершинах, алгоритм хранит очередь, в которой находятся все те помеченные вершины, для которых ещё не обработаны исходящие дуги. Таким образом, в каждый момент времени вершина может быть не помеченнной, помечанной и обработанной, и помечанной и необработанной. Идём из корня и последовательно отмечаем и заносим в очередь тех, к кому пришли.*

Утверждение 5. В каждый момент времени очередь состоит из некоторых вершин, находящихся на расстоянии l от s, вслед за которыми идут некоторые вершины, находящиеся на расстоянии $l+1$ от s. При этом все вершины на расстоянии, меньшем, чем l, уже обработаны, ровно как и все вершины на расстоянии l, не вошедшие в очередь. Из вершин на расстоянии $l+1$ в очереди есть ровно все потомки обработанных вершин.

Утверждение 6.

- алгоритм помечает врешину v тогда и только тогда, когда есть путь тз s в v;
- если алгоритм находит v по дуге (u,v), то один из кратчайших путей из s в v идёт через u;
- все пройденные дуги (u,v) образуют дерево.

Алгоритм 17. ***Поиск в глубину (DFS)***. *Ижём в глубину до конца, отмечаем вершины в чёрный, если из них начали идти вниз, серым, если они нам просто встретились на пути. После того, как дошли до конца, идём наверх до первой вершины. Время работы:* $O(|V|+|E|)$.

Задача 3. **Топологическая сортировка**. Нужно найти остовные деревья в орграфе (естественно, он должен быть ациклическим). Решается через DFS из следующих утверждений.

Утверждение 7. Граф ациклический тогда и только тогда, когда при поиске в глубину никогда не рассматривается дуга, ведущая в вершину, находящуюся в стеке возврата (дуга из серой в серую).

Утверждение 8. Если в ациклическом графе есть дуга (u,v), то время завершения v ментше, чем время завершения u.

Подробнее: 3 лекция, страница 6

Лекция 4.

Окончание поисков в орграфе.

Задача 4. Найти в данном графе его компоненты сильной связности.

Алгоритм 18. ***Алгоритм Косараджу-Шарира***. *Сперва запускается поиск в глубину для G, а затем запускается поиск в глубину для обращённого графа G^R, в котором направления всех дуг изменены на обратные (однако, компоненты связности те же). При поиске в глубину в обращённом графе, во внешнем цикле вершины рассматриваются в порядке их завершения при первом поиске в глубину, от конца к началу. После этого оказывается, что каждый запуск процедуры DFS во внешнем цикле будет находить очередной сильно связный компонент исходного графа. Время работы: $O(|V|+|E|)$, корректность обосновывается следующим:*

Утверждение 9. Пусть в графе G есть сильно связные компоненты C и D, и есть дуга $(u, v) \in E$ из C в D. Тогда при поиске в глубину в графе G самое позднее время завершения вершины в C превосходит таковое в D.

Доказательство. Рассматриваются два случая: самое ранне обнаружение в C меньше, чем в D, тогда рассматриваем первую обнаруженную вершину $x \in C$, у всех вершин из D время окончания меньше, чем у неё. Если же это не так, то рассмотрим самую раннюю $y \in D$. Все остальные из этой компоненты будут обнаружены на рекурсивных вызовах, а из C на этом этапе не обнаружатся, поэтому время окончания всех вершин из C больше. □

Теорема 4. *Вершины каждого дерева, найденного алгоритмом Косадаржу-Шарира при втором поиске в глубину - это и есть сильно связные компоненты исходного графа.*

Доказательство. Индукция по количеству найденных компонент связности. Надо доказать, что если первые k найденных компонент связности действительно таковы, то и следующая также обладает этим свойством. □

Подробнее: 4 лекция, страница 2

Поиск в алгоритме с весами.

Задача 5. Пусть в орграфе для каждой дуги задан вес. Нужно найти пуи наименьшего веса из данной вершины $s \in V$ во все вершины графа.

Алгоритм 19. ***Алгоритм Беллмана-Форда***. *Для каждой вершины вычисляются значения d_v - наименьший вес пути из s в v и π_v - предыдущая вершина на пути наименьшего веса из s в v. Изначально полагается, что $d_v = \infty$ и $\pi_v = NULL$ для всех вершин, и $d_s = 0$. Далее алгоритм постепенно находит пути меньшего веса в другие вершины, запоминая веса лучших из найденных путей в этих переменных, Значения уменьшаются с помощью элементарной операции улучшения пути, используя некоторую дугу $(u, v) \in E$. Если $d_u + w_{u,v} < d_v$, то $d_v = d_u + w_{u,v}$, а $\pi_v = u$. Эта операция применяется, пока можно что-то улучшить. Как будет показано, для этого достаточно рассмотреть все дуги $|V| - 1$ раз.*

Утверждение 10. После i-ой итерации внешнего цикла алгоритм Беллмана-Форда находит все пути наименьшего веса длины не более чем i.

Доказательство. Индукция по i. □

Теорема 5. *Алгоритм Беллмана-Форда за $|V| \cdot |E|$ шагов или правильно вычисляет пути наименьшего веса из вершины s во все вершины, или сообщает о наличии достижимого цикла отрицательного веса.*

Доказательство. Рассмотрим систему после $|V|-1$ итераций, и согласно утв. 7, нейдены все пути наименьшего веса, состоящие не более чем из $|V|-1$ дуг. Если какой-то путь ещё можно улучшить, то в этом пути какая-то вершина встретилась дважды, следовательно, найден цикл отрицательного веса. Если же ничего нельзя улучшить, то любой достижимый цикл будет иметь неотрицательный вес, так как для последовательный вершин цикла можно записать не условие улучшения и сложить по все парам. □

Алгоритм 20. ***Алгоритм Дейкстры.*** *Этот алгоритм решает ту же задачу, однако на каждом шаге находит очередную вершину u, путь наименьшего веса в которую уже известен. Тогда плгоритм рассматривает все дуги, исходящие из вершины u. Это позволяет ему рассматривать каждую дугу графа лишь однажды.*

Теорема 6. *Алгоритм Дейкстры работает правильно.*

Доказательство. Индукцией по длине вычисления доказываем, что для всякой вершины $u \notin Q$, путь наименьшего пути уже построен. Для этого рассматриваются две вершины на кратчайшем пути - одна в среди рассмотренных, другая - среди не рассмотренных (Q), и для последней проводятся вычисления, связанные с длиной самого короткого пути для неё. □

Примечание 2. Для представления множества Q алгоритм использует особую структуру данных: *очередь с приоритетами*. Каждый элемент Q находится там вместе со своим текущим значением d_v. Также заданы операции:

- $insert(x)$ - вставить новый элемент;
- $min()$ - выдать минимальный элемент;
- $extract_{min}()$ - выдать минимальный и удалить;
- $decrease(x, k)$ - изменить значение элемента $x \in Q$ на k, если k меньше текущего значения x_i.

Скорость работы алгоритма: $|V|$ раз $extract_{min}$ и $|E|$ раз $decrease$, поскольку каждая дуга обрабатывается лишь однажды.

Подробнее: 4 лекция, страница 3

Окончание поиска с весами.

Сложность алгоритма Дейкстры зависит от того, как реализвана очередь с приоритетами. Тупая реализация: хранить массив x_v, индекцированный по $v \in V$. Тогда $decrease$ работает за $O(1)$, но $extract_{min}$ требует время $|V|$. Отсюда общее время работы - $|V|^2 + |E| = O(|V|^2)$.

Алгоритм 21. ***Улучшение Дейкстры кучей.*** *Используется куча, в которой значение в каждой внутренней вершине не больше, чем значение в любом из её потомков (min-heap). Операции снатовятся следующими:*

- *insert*(x) - *вставить новый элемент (он становится листом), после чего дать ему всплыть до его законного места;*
- *min*() - *выдать минимальный элемент (просто вернуть* x_1*);*
- $extract_{min}()$ - *переместить* x_n *в* x_1*, убрав его в конце, а затем запустить исправление кучи из корня, то есть, heapify(1);*
- *decrease*(i, k) - *изменить значение элемента* x_i *на* k*, после чего дать элементу* x_i *всплыть наверх, пока возможно.*

Примечание 3. *Дать всплыть* - значит, если x_i меньше своего родителя, то он обменивается так до тех пор, пока не займёт положенное место.

Нахождение минимального остовного дерева.

Задача 6. Дан неориентированный связный граф, рёбрам которого сопоставлены числа - веса. Нужно найти одно из остовных деревьев с наименьшим весом.

Алгоритм 22. ***Общий принцип действия алгоритмов***. *Одно за другим присоединяются рёбра к поддереву какого-то минимального, чтобы опять получилось поддерево минимального.*

Определение 5. *Сечение* графа - разбиение множества вершин на два дизъюнктныхмножества. Ребро *пересекает сечение*, если один из его концов лежит в одной части, а другой - во второй.

Утверждение 11. Пусть T - одно из минимальных остовных деревьев графа, а F - его подмножество. Пусть также имеется какое-то сечение, что никакое ребро из FF его не пересекает. Тогда ребро с наименьшим весом, пересекающее это сечение, принадлежит некоторому минимальному остовному дереву T', которое также содержит F.

Доказательство. Если (u, v) - такое ребро, входит в T, то нам подойдёт T. Если его там нет, то рассмотрим цикл, который с нима полчуается и поменяем его на другое ребро из цикла, которое пересекает сечение. □

Алгоритм 23. ***Алгоритм Прима***. *Начинаем с произвольной вершины и на каждом шаге добавляем ребро из одной из уже имеющихся вершин в некоторую незадействованную (естественно, из всевозможных рёбер выбирается то, у которого минимальный вес).*

Время работы: $|E| \log |V|$*, так как выполняется* $|V|$ *операций внешнего цикла. Операции над очередью с приоритерами -* $\log |V|$*. Внутренний цикл по* v*: за всё время работы алгоритма каждое ребро рассматривается дважды: с одного конца и с другого. Поэтому тело цикла в общей сложности выполняется* $2|E|$ *раз, и всякий раз выполняется операция над очередью с приоритетами за время* $O(\log |V|)$.

Примечание 4. Структура данных алгоритма - очередь с приотитетеми, в которой хранятся все вершины, ещё не попавшие в дерево. Значение d_v каждойвершины v - это наименьший вес ребра, соединяющий её с деревом. Также для v запоминается вершина π_v, через которую v соединена с деревом ребром наименьшего веса.

Всякий раз, когда в дерево добавляется новая вершина u, для любой вершины v не из дерева моэет оказаться, что ребро (u, v) легче, чем ранее известное ребро наименьшего веса (π_v, v), соединяющее её с деревом, и это тка, если $w_{u,v} < d_v$. В этом случае значения d_v и π_v обновляются, переключая v на соединение с деревом через u.

Алгоритм 24. ***Алгоритм Крускала***. *Текущее подмножество остовного дерева - лес, соединяем компоненты самыми лёгкими путями.*
Теорема 7. *На каждом шаге работы алгоритма Крускала переменная T содержит подмножество одного из остовных деревьев минимального веса.*

Доказательство. Индукция по длине вычисления. □

Утверждение 12. При использовании леса непересекающихся множеств алгоритм работает за время $|E| \log |E|$ - и, стало быть, $|E| \log |V|$.

Доказательство. Сортировка займёт время $|E| \log |E|$. Далее алгоритм выполняет $3|E|$ операций над структурой данных, каждая из которых, при реализации через лес непересекающихся множеств, выполняется за время $O(\log n)$, и в общей сложности получается $|E| \log |V|$, что уже быстрее сортировки. □

Подробнее: 4 лекция, страница 8

Структура данных для непересекающихся множеств.

Задача 7. Абстрактная структура данных для представителя разбиения множества на непересекающиеся подмножества. У каждого множества есть выжеленный элемент - *представитель*. Заданы операции:

- $make\_set(x)$ - создать одноэлементное множество;
- $find\_set(x)$ - найти представитель множества, содержащего данный элемент;
- $set\_union(x, y)$ - объединение двух множеств.

Перед нами стоит задача: как эффективно реализовать эту абстрактную структуру данных?

Алгоритм 25. ***Лес непересекающихся множеств***. *У нас есть лес, если одноэлементно - просто вершинка, если представитель - корень дерева, если объединяем, то корень одного будет указывать на корень другого.*

Примечание 5. Используются следующие улучшения:
В каждой вершине хранится её условная сложность - *ранг*. Процедура $make\_set(x)$ устанавливает ранг в нуль. При объединении двух деревьев корень дерева меньшего ранга станет указывать на корень дерева большего ранга. Если оба корня имеют одинаковый ранг, то их объединение получит ранг, больший на единицу.
При выполнении каждой операции поиска все встреченные на пути вершины пренаправятся в корень для ускорения последующих операций цикла.

Утверждение 13. Пусь всего элементов - n, и к ним применяется m операций. Тогда, с применением первого улучшения, они выполняются за время $O(m \log n)$.

Доказательство. Смотрим на дерево, через логарифмы вычисляем. □

Теорема 8. *(**Тарьян**). Пусть всего элементов - n, и над ними выполняется m операций. Тогда, с применением первого и второго улучшения, эти операции выполняются за совокупное время $m\alpha(n)$, где $\alpha(n)$ - обратная функция к $A_n(n)$, а $A(k, n) = A_k(n)$ - функция Аккермана.*

Задача 8. Дан неориентированный граф - возможно, несвязный. Надо определить его компоненты связности и научиться быстро отвечать на вопрос о том, лежат ли две данные вершины в одной компоненте.

Алгоритм 26. *Алгоритм пишется через представление компонент связности в виде структуры данных для непересекающихся множеств.*

Подробнее: 4 лекция, страница 11

Лекция 5.

Пути между всеми парами вершин.

Задача 9. Дан ориентированный граф $G = (V, E)$, где $V = \{1, \dots, n\}$, а $E \subseteq V \times V$. Ствается задача проверить чуществование пути из каждой вершины в каждую - то есть: для каждой пары вершин (i, j) определить, есть ли путь из i в j. После решения мы получим *транзитивное замыкание* графа.

Алгоритм 27. ***Очевидный алгоритм***. *Перебираем все пары вершин (i, j) и для каждой пары все промежуточные вершины k. Если она соединена с обоими, то добавляем ребро (i, j). Делаем так пока можем.*

Утверждение 14. После каждой t-ой итерации внешнего цикла будут просчитаны все пути длины не более 2^t, и для каждого из них в граф будет добавлена дуга. Откуда время работы - не более, чем $n^3 \log_2 n$ шагов.

Алгоритм 28. ***Алгоритм Варшалла***. *Можно построить транзитивное замыкание за время n^3, обойдясь без внешнего цикла (без пока можем). Для этого достаточно поменять циклы местами и теперь рассматривать каждую вершину как смежную, соединяя все пары, смежные с ней.*

Утверждение 15. После k-ой итерации внешнего цикла найдены все пути, проходящие только через промежуточные вершины из множества $\{1, \dots, k\}$.

Доказательство. Доказательство индукцией по количеству итераций. □

Задача 10. Пусть граф теперь не только ориентированный, но и рёбра имеют вес. Нужно найти кратчайшие пути между вершинами.

Алгоритм 29. ***Алгоритм Флойда-Варшалла***. *То же самое, что и алгоритм варшала, но мы ещё и храним вес рёбер.*

Примечание 6. О достижимости можно говорит в рамках матриц смежности. Умножение рассматривается как

$$c_{i,j} = \bigvee_{k=1}^{l} a_{i,k} \wedge b_{k,l}.$$

Тогда если A - матрица смежности, то A^l - матрица достижимости по путям длины ровно l.

Подробнее: 5 лекция, страница 1

Быстрое умножение матриц.

Ну, какие-то способы усножения - ебанина какая-то, писать не буду.

Теорема 9. *(**Основная теорема о времени работы рекурсивных алгоритмов**). Пусть задача размера n рашается путём разбиения её на a подзадач того же типа, размера $\frac{n}{b}$ каждая, и процесс разбиения, а также соединения результатов занимает время $O(n^c)$.*

$$T(n) = aT(\frac{n}{b}) + O(n^c)$$

Тогда время работы алгоритма оценивается как

- *Если $c < \log_b a$, то $O(n^{\log_b a})$;*
- *Если $c = \log_b a$, то $O(n^c \log_b n)$;*
- *Если $c > \log_b a$. то (n^c).*

Доказательство. В дереве рекурсии $\lceil \log_b n \rceil$ уровней, на нулевом - одна подзадача размера n, рассчёты занимают время n^c, на первом - a подзадач размера $\lceil \frac{n}{b} \rceil$, расчёты для каждой занимают врем] $\lceil \frac{n}{b} \rceil^c$, и так далее, на уровне i - a подзадач размера $\lceil \frac{n}{b^i} \rceil$ каждая, расчёты каждой занимают время $\lceil \frac{n}{b^i} \rceil^c$. Общее время работа - сумма по всем уровням рекурсии.

$$T(n) = \sum_{i=0}^{\lceil \log_b n \rceil} a^i \left(\frac{n}{b^i}\right)^c = n^c \sum_{i=0}^{\lceil \log_b n \rceil} \left(\frac{a}{b^c}\right)^i$$

Таким образом, если $a = b^c$, то сумма геометрической прогрессии - $\log_b^n$, и отсюда $O(n^c \log_b n)$. Если $a < b^c$, то сумма ограничена сверху константой, отсюда $O(n^c)$. И если $a > b^c$, то это возрастающая геометрическая прогрессия, и применив формулу для вычисления получим $O(n^{\log_b a})$. □

Подробнее: 5 лекция, страница 8

Задача 11. Умножение булевых матриц.

Алгоритм 30. *Можно записать матрицу в виде кучи нулей и единиц, после чего перемножить в кольце вычетов по модулю $n+1$ или по любому удобному модулю, превосходящему n. Тогда вместо дизъюнкции конъюнкций будет вычислена сумма произведений по модулю $n+1$ (в точности равная оличества интинных конъюнкций в дизъюнкции конъюнкций, и потому она лежит в диапазоне от 0 до n, откуда следует, что по модулю $n+1$ она вычислится точно). Чтобы узнать значение дизъюнкции конъюнкций, достаточно будет проверить вычисленную сумму на равенство нулю.*

Алгоритм 31. ***Метод четырёх русских**. Чисто комбинаторный метод умножения булевых матриц за время $o(n^3)$. Пусть A и B - две булевых матрицы размера $n \times n$, цель - вычислить их произведение $C = AB$. Пусть k - число, намного меньшее, чем $\log_2 n$, и пусть n делится на k (если не делится - немного увеличим n до делимости). Каждая строка матрицы A делится на $\frac{n}{k}$ векторов размера $1 \times k$, называемых кусочками. Кусочки в i-ой строке обозначаются через $A_{i,l}, \ldots, A_{i,\frac{n}{k}} \in \mathbb{B}^{1\times k}$. Матрица B разделяется на $\frac{n}{k}$ подматриц размера $k \times n$, называемых полосами и обозначаемыми через $B_1, \ldots, B_{\frac{n}{k}} \in \mathbb{B}^{1\times k}$.*

Тогда всякая i-ая строка C, обозначаемая через $C_i \in \mathbb{B}^{1\times n}$, представима в виде следующей поэлементной дизъюнкции строк:

$$C_i = \bigvee_{r=1}^{\frac{n}{k}} A_{i,r}B_r.$$

Произведение кусочка $A_{i,r}$ на соответствующую полосу B_r - это строка размера $1 \times n$, и знак дизъюнкуции в формуле для C_i означает поразрядную дизъюнкцию таких строк,

Примечание 7. Казалось бы, никакого принципиального улучшения не получилось, однако с программистсткой точки зрения, это в чём-то удобнее.

Главная идея метода же состоит в том, что так как k невелико, возможных кусочков всего 2^k, и кусочки будут часто повторяться. Таким образом, если запомнить произведения таких кусочков на полосы B, в плане вычислений это займёт много меньше времени. Общее время работы будет $O\left(\frac{n^3}{\log n}\right)$.

Подробнее: 5 лекция, страница 10

Лекция 6.

Двоичные деревья поиска

Речь сейчас пойдёт о структурах данных для представления *множества* различными способами, с разными временем выполнения операций (строго говоря, мы будем говорить о *мультимножестве*, так как может быть несолько элементов с одинаковым значением). Элементы погут быть любого вида, на них определено отношение порядка $''\leq''$. Операции: найти элемент с данным значением; найти наибольший (наименьший) элемент; найти элемент, предшествующий или следующий за данным; вставить / удалить элемент.

Алгоритм 32. ***Бинарное дерево поиска***. *Представление мнлжества в виде дерева так, что вершина содержит одно значение и три указателя: на предка (если корень, то NULL), на левое поддерево и на правое (если их нет, то NULL). При этом все вершины в её левом поддереве содержат не меньшие значения, а все вершины в правом поддереве - не бóльшие. Понятно, как на таком дереве задаются требуемые операции.*

Примечание 8. Сложность каждой операции - не более, чем высота дерева. Если дерево сбалансированное, то есть, все пути примерно одинаковой длины, то все операции выполняются за логарифмическое время.

Задача 12. Даже если дерево сбалансированное, то операции над ним могут првести его к дизбалансу, надо как-то сложниить структуру, чтобы избежать критических изменений.

Алгоритм 33. ***АВЛ-деревья***. *Усложнённая разновидность двоичных деревьев поиска. Такую разновидность называют почти сбалансированной, то есть, высота деревьев-потомков одной вершины отличается не более, чем на 1. Этого ограничения достаточно, чтобы дерево мело логарифмическую высоту.*

Утверждение 16. АВЛ-дерево высоты h содержит не менее, чем $F_{h+3} - 1$ вершин, где F_n - n-ое число Фибоначчи.

Доказательство. Индукция по h. База очевидна, переход таков. Если дерево имело высоту h, то один из его потомков имеет высоту $h-1$, а другой - высоту не менее, чем $h-2$, то

есть, не менее чем $F_{h+2}-1$ и $F_{h+1}-1$ вершин соответственно. Всего, с учётом корня и по определению чисел Фибоначчи, как раз и получится то, что нужно. □

Утверждение 17. Высота АВЛ-дерева с n вершинами не превосходит $\log_\varphi n$, где φ - золотое сечение.

Примечание 9. Естественно, просто так АВЛ-дерево не получить, поэтому придумана операция *вращение*, которая состоит в том, что если у нас есть корень y, из которого направо идёт поддерево t_3, а налево - поддерево с корнем x и поддеревьями t_1 и t_2 налево и направо соответственно. Тогда мы подвешиваем дерево за x и перебрасываем его поддерево t_2 в левое поддерево y. При помощи такой операции можно исправить балансировку при различных операциях, подробно расписывать я это пока что не буду.

Задача 13. Двоичные деревья рассчитаны на то, чтобы храниться в оперативной памяти компьютера, а для того, чтобы хранить деревья во внешней, медленной памяти, требуется некая адаптация.

Подробнее: 6 лекция, страница 1

B-деревья.

Алгоритм 34. ***B-деревья***. *Адаптация деревьев поиска для хранения во внешней памяти. Основная мысль - использовать вершины большой степени - с тем, чтобы каждая вершина занимала один блок, а высота дерева уменьшилась бы.*

Пусть вершины в двоичном дереве - это 2-вершины, поскольку у каждой из них 2 потомка и 1 значение, по которому эти потомки разделяются. У m-вершины - m потомков (деревья $t_1,\dots,t_m$ - возможно, пустые), и в ней находится $m-1$ значение: $x_1,\dots,x_{m-1}$, где значения упорядочены по неубыванию. Все значения в каждом поддереве t_i, больше или равны x_i, и меньше или равны x_{i+1}.

В B-дереве могут одновременно соержаться вершины различных степеней: выбирается некоторое число $k \geq 2$, после чего корень может иметь степень от 0 до $2k$, а все остальные вершины - любые степени от k до $2k$. При этом, дерево сбалансировано.

Примечание 10. То, как производятся требуемые операции - лучше один раз увидеть и немного прочитать в конспекте А. С. Охотина, чем читать мои заметки.

Алгоритм 35. *B-дерево для $k=2$ называется $2-3-4$ деревом, и такое дерево очень удобно хранить в оперативной памяти.*

Алгоритм 36. *Красно-чёрное дерево - это представление 2-3-4 дерева в виде двоичного дерева, в котором каждая вершина представлена в виде одной или нескольких связанных между собою двоичных вершин, каждая из которых покрашена в один из двух цветом. Каждая 2-вершина остаётся собой и считается чёрной. Каждая 3-вершина разбивается на две двоичных: чёрную - корень поддерева, и красную вершину - правого или левого потомка чёрной. Далее - к этой паре вершин, так же как и к исходной 3-вершине, присоединяются три поддерева. Наконец, каждая 4-вершина разбивается на три двоичных - чёрную и два красных потомка, к которым присоединены четыре поддерева.*

Подробнее: 6 лекция, страница 7

Лекция 7.

Полиномиальное хэширование.

Задача 14. Дана длинная строка $w = a_1 \dots a_n$ и короткая искомая строка $x = b_1 \dots b_m$. Требуется найти все вхождения x в w в качестве подстроки.

Алгоритм 37. ***Хэш-функции.*** *Каждой строке ставится в соответствие некоторое число $w \mapsto h(w)$, и это число хранится вместе со строкой. Тогда, если нужно сравнить две строки, то мы сравниваем сперва соответствующие им числа. Если числа разные, то и строки точно разные. Если соответствующие числа одинаковые, то надо сравнить строки и получить итоговый результат. Самое тупое - сложить коды всех символов, но тогда распределение неравномерное, и вообще это не очень кайфовый вариант. Намного лучше полиномиальное хэширование. Берём некоторое основание степени p, пусть $w = a_1 \dots a_l$ - строка, тогда используется сумма $\sum_{i=1}^{l} a_i \cdot p^{l-i}$, взятая по некоторому модулю M.*

Алгоритм 38. ***Алгоритм Рабина-Карпа.*** *Алгоритм поиска подстроки $x = b_1 \dots b_m$ в строке $w - a_1 \dots a_n$, основанный на полиномиальном хэшировании степени $m - 1$: сперва вычисляется значение хэщ-функции для искомой строки, а затем для всех m-символьных подстрок данного текста последовательно вычисляется значение их хэш-функции. Когда значение для подстроки и искомой строки совпадают, производится прямое сравнение. Значение хэш-функции для искомой строки: $X = \sum_{i=1}^{m} b_i \cdot p^{m-i}$ по модулю q, а значение хэш-функции для подстроки со смещением s: $W_s = \sum_{i=1}^{m} a_{s+i} \cdot p^{m-1}$ по модулю q. Алгоритм сперва вычисляет X, а затем - последовательно все W_s. Из соотношения $W_{s+1} = pW_s - a_{s+1}p^m + a_{s+m+1}(\mod q)$. Сложность: $\Theta(m)$ - на подготовку, и затем в худшем случае $O(mn)$, если хэ=-функция выдаст одинаковые значения для всех подстрок. В среднем случае получается время работы $\Theta(n)$.*

Задача 15. (Наибольшая общая подстрока). Даны две строки: $u = a_1 \dots a_m$ и $v = b_1 \dots b_n$. Надо найти самую длинную их общую подстроку x.

Алгоритм 39. *Тупое решение: для каждой пары позиций найти самую длинную подстроку, время $((m+n)^3)$.*

Алгоритм 40. *Улучшение через полиномиальное хэширование - научиться отвечать на вопрос "Есть ли общая подстрока длины l?" за время $O((m+n)\log(m+n))$. Для этого находятся значения хэш-функции для всех подстрок и длины l, размещаются в двоичном дереве поиска - время $m \log m$. То же самое - для v за время $n \log n$.*

Алгоритм 41. *Улучшение через двоичный поиск по l. Его можно записать в виде рекурсивной процедуры, отвечающей на вопрос "Найти длину наибольшей общей подстроки, если известно, что её длина не меньше, чем l_1, и строго больше, чем l_2?". Процедура $f(l_1, l_2)$: если $l_2 - l_1 = 1$, то возвращаем l_1, $k = \lfloor \frac{l_1+l_2}{2} \rfloor$. Теперь, если есть общая подстрока длины k, то возвращаем $f(k, l_2)$, иначе возвращаем $f(l_1, k)$.*

Задача 16. Найти в строке самую длинную подстроку-палиндром.

Алгоритм 42. *Самый тупой алгоритм: искать вокруг каждого симыола на длину вплоть до $\frac{n}{2}$, время $O(n^2)$.*

Алгоритм 43. *Улучшение основано на решении подзадачи: "Есть ли подстрока-палиндром данной длины l?". После того, как будет построен алгоритм для решения подзадачи, останется использовать двоичный поиск по l.*

Алгоритм 44. *Решение через небольшую переформулировку: разворачиваем строку и для таковой и оригинальной рассматриваем хэш-ффункции для всех подстрок длины l. После чего, естественн, надо выполнить посимвольную проверку. Двоичный поис по l даст время $O(n \log n)$.*

Подробнее: 7 лекция, страница 2

Алгоритм Кнута-Морриса-Пратта.

Алгоритм 45. *Строка w читается слева направо, и после чтения i символов w он помнит длину j самого длинного префикса строки x, на которой заканчивается $a_1 \dots a_i$ - то есть, наибольшее число j, для которого верно $a_{i-j+1} \dots a_i = b_1 \dots b_j$.*

При чтении очередного символа a_{i+1} алгоритм сравнивает его с b_{j+1}, и если эти символы равны, это значит, что префикс x длины j продлевается на один символ. Если же $a_{i+1} \neq b_{j+1}$, то самый длинный префикс x, на который заканчивается строка $a_1 \dots a_{i+1}$, будет содержать строго меньше, чем $j+1$ символов. Чтобы найти длину этого префикса, нужно рассмотреть следующий по длине префикс x на который заканчивается $a_1 \dots a_i$, и пытаться продлевать уже его.

Чтобы иметь возможность находить длину такого префикса, алгоритм заранее строит по данной искомой строке x определённую стуктуру данных - префиксную сумму.

Определение 6. *Префиксная сумма* для строки $x = b_1 \dots b_m$ - это функция $\pi\{1, \dots, m\} \to \{0, \dots, m-1\}$, которая для всякого префикса $b_1 \dots b_i$ строки x выдаёт длину наибольшего суффикса подстроки $b_1 \dots b_i$, который также является префиксом x.

$$\pi(i) = \max\{j | j < i, b_1 \dots b_j = b_{i-j+1} \dots b_i\}$$

Примечание 11. Повторно применяя функцию π можно получить все префиксы x, являющиеся суффиксами $b_1 \dots b_i$, и алгоритм будет их перебирать, пока не найдёт такой, который можно продолжить следующим символом текста.

Утверждение 18. Имея готовую таблицу значений функции π, алгоритм КМР работает за время $\Theta(n)$.

Примечание 12. Осталось научиться быстро строить значение префиксной функции.

Подробнее: 7 лекция, страница 4

Поиск с помощью конечных автоматов.

Определение 7. *Детермированный конечный автомат (DFA)* - пятёрка $A = (\Sigma, Q, q_0, \delta, F)$, со следующим значением компонентов:

- Σ - алфавит;
- Q - конечное множество состояний;
- $q_0 \in Q$ - начальное состояние;

- $\delta : Q \times \Sigma \to Q$ - функция переходов. Если автомат находится в состоянии $q \in Q$ и читает символ $a \in \Sigma$, то его следующее состояние - $\delta(q, a)$;
- $F \subseteq Q$ - множество *принимающих состояний*.

Для всякой входной строки $w = a_1 \dots a_l$, где $l \geq 0$ и $a_1, \dots, a_l \in \Sigma$, вычисление - последовательность состояний $p_0 \dots, p_{l-1,p_l}$, где $p_0 = q_0$, и всякое следующее состояние p_i, где $i \in \{1, \dots, l\}$, однозначно определено как $p_i = \delta(p_{i-1}, a_i)$.

$$p_0 \to^{a_1} p_1 \to^{a_1} \dots \to^{a_l} p_l$$

Строка *принимается*, если последнее состоение p_l принадлежит множеству F - иначе *отвергается*. Множество строк, *распознаваемое* автоматом, обозначаемое через $L(A)$ - состоит из всех строк, которые он принимает.

Теорема 10. *Для всякой строки $x = b_1 \dots b_m$ над алфавитом Σ существует DFA с $m+1$ состояниями, распознающий множество $\Sigma^* x$ всех строк, заканчивающихся на x, и этот DFA можно построить за время $\Theta(|\Sigma| \cdot m)$.*

Доказательство. Пока что записывать не буду. □

Подробнее: 7 лекция, страница 6

Представления множеств строк.

Определение 8. *Лексикографический порядок*. Поэлементное сравнение слева направо.

Задача 17. Как хранить строки?

Алгоритм 46. *В виде списка или двоичного дерева поиска, однако это всё хуйня.*

Алгоритм 47. ***Префиксное дерево*** *- структура для хранения множеств строк. Корневое дерево с дугами, помеченными символами алфавита. Дуги, исходящие иизз всякой вершины, должны быть помечены различными символами. Каждая вершина соответствует некоторой строке, которая хранится неявно в виде последовательности дуг на пути к ней. Также всякая вершина хранит один бит информации, принадлежит ли эта строка ммножеству; вместе с битом можно хранить любые дополнительные значения, сопоставленный в этой строке. Корень соответствует пустой строке. Если вершина соответствует строке w, и исходящая из неё дуга помечена символом $a \in \Sigma$, то эта дуга идёт в вершину, соответствующую строке wa.*

Все операции с любой данной строкой длины l (поиск, вставка, удаление) выолняются за время $O(l)$, не зависящее от числа элементов в хранимом множестве.

Компактное префиксное дерево - объединяем несколько дуг в одну, если они идут одинаково параллельно.

Утверждение 19. В компактном префиксном дереве для множества из n элементов не более, чем n внутренних вершин.

Поиск нескольких строк одновременно: алгоритм Ахо-Корасик.

Задача 18. Пусть для данного текста нужно найти все вхождения не одной строки, а всех строк из конечного множества $K = \{x_1, \ldots, x_k\}$.

Алгоритм 48. ***Алгоритм Ахо-Корасик.*** *Данный алгоритм будет помнить самый длинный только что прочитанный префикс одной из строк $x_1, \ldots, x_k$, а точнее сказать, самую длинную такую строку u, что u - префикс какой-то строки из K, и алгоритм только что прочитал u.*

Примечание 13. Для решения обобщается префиксная сумма и построение функции π (работающее за линейное время), но пока что упущу подробности.

Подробнее: 7 лекция, страница 9

Суффиксные деревья.

Алгоритм 49. ***Суффиксное дерево*** *- структура данных, строящаяся по данной строке и обеспечивающая эффективный поиск подстрок в этой строке. Понятно, что это такое.*

Лекция 9.

Сжатие данных, основанное на повторении строк.

Задача 19. Нужно научиться использовать часто повторяющиеся подстроки, чтоы хранить их отдельно в памяти и эффективно использовать при хранении целого текста.

Алгоритм 50. ***Метод Лемпеля-Зива LZ77.*** *Алгоритм сжатия данных, использующий повторяемость подстрок. Основная идея в том, что каждый раз, когда ранее прочитанная подстрока встречается повторно, алгоритм пишет вместо неё ссылку на предыдущее вхождение. На каждом шаге выводится тройка (d, l, a), что означает: сперва повторяется подстрока длины l, ранее встречавшаяся d символов назад, а потом идёт символ a. Реализуется самым простым образом через суффиксное дерево, в котором на каждом шаге читаются следующие символы входной строки, пока не находится самая длинная раньше встречавшаяся подстрока. Обычно используется небольшое улучшение - "скользящее окно", то есть, подстроки ищутся (тем же суффиксным деревом, скорее всего) в окне из последних m элементов, а остальные забываются.*

Теорема 11. *Жадный LZ77 оптимален.*

Алгоритм 51. ***Метод Лемпеля-Зива LZ78.*** *Тут уже по мере чтения строки строится словарь из часто встречающихся строк в виде префиксного дерева. При декодировании строится в точности этот же словарь. В сжатом представлении строки словарь не хранится.*

В начале в словаре содержится только один элемент под номером нуль: $T_0 = \varepsilon$, иными словами, префиксное дерево состоит из корня, помеченного номером 0. На каждом шаге читается самая длинная строка $T_j = v$, и уже имеющаяся в словаре, и выводится её код j; также читается и выводится следующий символ a. При этом в словарь добавляется новая трока va - конкатенация только что прочитанной со следующим входным символом.

На префиксном дереве это выглядит так: после вывода очередной пары (j, a) алгоритм переходит в корень префиксного дерева, и дальше читает столько входных символов,

сколько возможно. Когда очередной символ прочитать нельзя, создаётся новый лист, при этом выводится номер предыдущей вершины и прочитанный символ.

При декодировании строится то же самое префиксное дерево. Алгоритму при этом потребуется находить строку по номеру в таблице: можно, например, завести для этого отдельный массиы строк, или же читать в префиксном дереве путь из вершины в корень. Прочитав очередную пару (j, a) алгоритм выводит строку T_j и символ a, после чего сразу перескакивает в вершину j и присоединяет к ней новый лист, с переходом по символу a.

Подробнее: 9 лекция, страница 1

Алгоритм 52. ***Преобразование Берроуза-Вилера***. *Перутся все циклические сдвиги строки $w = a_1 \dots a_n$. Они сортируются лексикографически, получается таблица T размера $n \times n$, где в каждой строчке - один из циклических сдвигов. наконец, берётся последовательность последних символов $b_1 \dots b_n$ в таблице T и номер t строки w в этой таблице.*

Если в исходной строке какая-то подстрока au часто повторялась, то есть много циклических сдвигов, начинающихся с u и заканчивающихся на a, и после сортировки они окажутся рядом. Поэтому в преобразованной строке будут длинные последовательности повторяющихся символов. Использование в качестве метода сжатия данных: преобразовать, а потом применять другие методы к преобразованной строке.

Утверждается, что по последовательности b_i, и t восстанавливается исходная w.

Утверждение 20. BWT можно вычислить за линейное время.

Утверждение 21. Обратное к BWT преобразование можно вычислить за линейное время.

Подробнее: 9 лекция, страница 4

Торжественно начинается часть конспекта, основанная на лекциях доцента ФМКН СПБГУ, кандидата физико-математических наук Александра Владимировича ТИСКИНА

Преобразование Фурье.

Введение в преобразование Фурье.

Алгоритм 53. ***Дискретное проеобразование Фурье степени*** n ***(DFT***$_n$***)*** - $F_{n,\omega} \cdot a = b$, *где* n - *обратим в* R, *коммутативном кольц без делителей нуля,* ω - *первообразный корень* $\sqrt[n]{1}$, $a = [a_0, \ldots, a_{n-1}]^T \in R^n$, $b = [b_0, \ldots, b_{n-1}]^T \in R^n$, *а* F_n, ω - *матрица*

$$[\omega^{ij}]_{i,j=0}^{n-1} = \begin{vmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{n-2} & \ldots & \omega \end{vmatrix}$$

Трудоёмкость: наивный алгоритм, время - $O(n^2)$.

Алгоритм 54. ***Обратное DFT***. $\frac{1}{n} F_{n,\omega^{-1}} \cdot b = a$.

Примечание 14. Используется для *вычисления значений* полинома $a(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ на $C = \{1, \omega, \ldots, \omega^{n-1}\}$. Аналогично для *интерполяции* полинома $a(x)$ по значениям на C - $\frac{1}{n} F_{n,\omega^{-1}} \cdot a$.

$\frac{1}{\sqrt{n}} F_{n,\omega^{-1}} \cdot a$ - разложение a по *базису Фурье* (строкам $\frac{1}{\sqrt{n}} F_{n,\omega}$).

Быстрое преобразование Фурье.

Алгоритм 55. ***Бвстрое преобразование Фурье (FFT)***. *Общая идея: исользовать разложение* n *на множители и структцту множества корней из единицы для ускорения вычислений. То есть, пусть* $n = n'n''$, $1 < n' \leq n'' < n$. *В алгоритме FFT*

- *вектор a записывается в виде $n' \times n''$-матрицы по строкам;*
- *вектор b записывается в виде $n'' \times n'$-матрицы по строкам;*
- *DFT_n выражается в виде $DFT_{n'}$, $DFT_{n''}$ над столбцами матриц;*
- *$DFT_{n'}$, $DFT_{n''}$ вычисляются рекурсивно.*

Алгоритм 56. *Некоторые классические варианты: $n = \frac{n}{2} \cdot 2$ - **FFT с прореживанием по времени** (FFT-DIT), и $n = 2 \cdot \frac{n}{2}$ - **FFT с прореживанием по частоте** (FFT-DIF).*

Алгоритм 57. *Наиболее общий слуяай - **шастиэтапное FFT**. Пусть $n = n'n''$, запишем матрицы в таком виде:*

$$A = \begin{vmatrix} a_0 & a_1 & \dots & a_{n''-1} \\ a_{n''} & a_{n''+1} & \dots & a_{2n''-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-n''} & a_{n-n''+1} & \dots & a_{n-1} \end{vmatrix},$$

и аналогично B. Тогда получим, что $B = F_{n'',\omega^{n'}} \cdot (G_{n',n'',\omega} \circ (F_{n',\omega^{n''}} \cdot A))^T$, где $G_{n',n'',\omega} = [\omega^{tv}]_{t,v} : n' \times n''$ - матрица поворотных множителей, а оператор $\circ$ - умножение Адамара (поэлементное умножение матриц).

И, действительно, данная схема почти шестиэтапная:

- *вектор a записывается в матрицу по строкам;*
- *n'' независимых $DFT_{n'}$ над столбцами, вычисляются рекурсивно;*
- *транспозиция; применение поворотных множителей (2 этапа);*
- *n' назависимых $DFT_{n''}$ над столбцами, вычисляются рекурсивно;*
- *вектор b считывается из матрицы по строкам.*

Примечание 15. Трудоёмкость FFT-DIT - $O(n \log n)$, трудоёмкость симметричного FFT ($n' = n''$) - $O(n \log n)$.

Примечание 16. Если вам интересен *граф-бабочка* и схемы FFT, то можете заглянуть в лекции Тискина. Я их, конечно же, перерисовывать не буду.

Задачи и алгоритмы решения.

Задача 20. Умножение полиномов. Пусть у нас есть два полинома с комплексными коэффициентами. Нас интересует их произведение. Тупейший алгоритм сделает это за время $O(n^2)$.

Алгоритм 58. ***Алгоритм Карацубы.*** *Поделим многочлены a и b (сделаем их одинаковой длины) на две равные части (по m), и запишем $a(x) = a'(x) + a''(x)x^m$, аналогично с $b(x)$. Тогда $c(x) =$*

$$= a'b' + ((a' + a'')(b' + b'') - a'b' - a''b'')x^m + a''b''x^n.$$

Таким образом, вместо 4 умножений полиномов с m членами получилось 3. Трудоёмкость процесса получилась $O(n^{\log_2 3})$.

Задача 21. Быстрая интерполяция.

Алгоритм 59. *Вот если мы посмотрим на ту красивую матрицу F, то заметим, что применив обратное преобразование Фурье к столбцу из значений в корнях из единицы, то как раз получим то, что нужно. В этом нетрудно убедится, провернув преобразования не в эту обратную, а в прямую сторону. Трудоёмкость: $O(n \log n)$.*

Определение 9. *Свёртка* (линейная) a и b - двух последовательностей длины n - $c = a * b$, что находится по формуле $c_i = \sum_{0 \le j \le i < 2n} a'_j b'_{i-j}$, где a' и b' - дополнения изначальных до длины $2n$. Есть ещё циклическая и косоциклическая свёртки, но они противные.

Алгоритм 60. *Суть - то же умножение полиномов. $c = \frac{1}{n} F_{2n,\omega^{-1}}((F_{2n,\omega} a') \circ (F_{2n,\omega} b'))$.*

Примечание 17. Немного про количество битов, необходимых для операций прямого и обратного DFT в $\mathbb{Z}_n$:

- сумма двух значений: в результате $(\frac{nr}{2} + 1) + 1 = \frac{nr}{2} + 2$ бит;
- умножение на поворотный множитель $\omega^q = 2^{qr}$, $0 \le q < n$: сдвиг на $qr < nr$ бит: в результате - не более $(\frac{nr}{2} + 1) + nr - 1 = \frac{3nr}{2}$ бит;
- приведение значений с записью из $\frac{3nr}{2} = O(nr)$ бит.

Трудоёкость каждой операции над значениями: $O(nr)$ битовых операций. Что касается трудоёмкости всего DFT_n в $\mathbb{Z}_N$: на вход/выход - $n \cdot O(nr) = O(n^2 r)$ бит, и $O(n \log n)$ сложений и умножений на поворотные множители, итого $O(n^2 r \log n)$ битовых операций.

Задача 22. Эффективное умножение многозначных чисел.

Алгоритм 61. ***Алгоритм Карацубы.*** *Как и умножение многочленов, просто сделаем из чисел многочлены поразрядно. В качестве значения на лекции рассматривалась двойка.*

Алгоритм 62. ***Алгоритм Шенхаге-Штрассена.*** *Основная мысль состоит в том, что мы разбиваем a и b на разряды правильно выбранной длины l и рассматриваем каждый разряд как коэффициент полинома, всего n/l коэффициентов в каждом полиноме. Используем быстрое преобразование Фурье для их умножения. Попарное умножение значений полиномов выполняется рекурсивно, затем учитываются переносы. Сам алгоритм, ну там пиздец намешали и КТО, и косоциклическую свёртку, и Карацубу. В сухом итоге, трудоёмкость - $O(n \log n \log \log n)$.*

Параллельные алгоритмы.

Базовые определения и понятия.

Определение 10. *Схема* (вычислительная) - ориентированный ациклический граф (dag), с фиксированным количеством входов и выходов. Вычисления неадаптивны (oblivious): последовательность операций не зависит от входа.

Вычисления над переменным количеством входов: (бесконечное) *семейство схем*. Семейство схем с конечным описанием - *алгоритм*.

Определение 11. *Размер* схемы - количество узлов, *глубина* - максимальная длина пути от входа до выхода.

Определение 12. *Схема сравнения* - схема, которой узлы - *элементы сравнения*. Сама схема выглядит как куча вертикальных полосок и перпендикулярных им отрезочков, которые соединяют некоторые пары длинных линий. Вертикальные линии - элементы сравнения, а отрезочки - сами операции сравнения.

Определение 13. *Схема слияния* - схема сравнения, которая берёт на вход две отсортированные последовательности и на выходе даёт отсортированную последовательность из всех их элементов.

Определение 14. *Схема сортировки* - схема сравнения, у которой на входе произвольная последовательность, а на выходе - отсортированная последовательность.

Примечание 18. Семейство схем с конечным описанием - алгоритм сортировки. Их размер и глубина - последовательная и параллельная сложность алгоритма.

Больше о сортировках и слияниях.

Теорема 12. ***Принцип нулей-единиц.*** *Схема сравнения является сортирующей тогда и только тогда, когда она сортирует все последовательности нулей и единиц.*

Доказательство. В одну сторону очевидно, в обратную пойдём от противного, ничего сложного. □

Примечание 19. Между прочим, очень применимая хуйня. Ей можно проверять сеть сортировки путём проверки только 2^n входных последовательностей, а не $n!$. А также, сеть слияния путём проверки $(n'+1)(n''+1)$ вместо много.

Алгоритм 63. ***Чётно-нечётное слияние (OEM).*** *Если $n' = n'' = 1$, то сравниваем (x_1, y_1), иначе рекурсивно*

- *слияние $\langle x_1, x_3, \ldots\rangle, \langle y_1, y_3, \ldots\rangle$;*
- *слияние $\langle x_2, x_4, \ldots\rangle, \langle y_2, y_4, \ldots\rangle$;*
- *попарное сравнение: $(u_2, v_1), (u_3, v_2), \ldots$*

Размер - $O(n \log n)$, глубина - $O(\log n)$.

Доказательство. Доказательство корректности приводится посредством индукции. □

Алгоритм 64. ***Сортировка нечётно-чётным слиянием.*** *Если $n = 1$ - останавливаемся, иначе рекурсивно*

- *сортировка $\langle x_1, \ldots, x_{\lceil n/2\rceil}\rangle$;*
- *сортировка второй половины;*
- *слияние результатов при помощи OEM.*

Размер - $O(n(\log n)^2)$, глубина - $O((\log n)^2)$.

Определение 15. *Битонная последовательность*: $\langle x_1 \geq \ldots \geq x_m \leq \ldots \leq x_n\rangle$, $1 \leq m \leq n$.

Алгоритм 65. ***Битонное слияние (BM)****: сортировка битонной последовательности. Если $n = 1$ - останавливаемся, иначе рекурсивно*

- *битонное слияние* $\langle x_1, x_3, \ldots \rangle$ *в упорядоченную последовательность;*
- *аналогично по чётным;*
- *попарное сравнение* $(u_1, v_1), (u_2, v_2), \ldots$.

Размер - $O(n \log n)$, *глубина* $O(\log n)$.

Алгоритм 66. ***Сортировка битонным слиянием.*** *Если* $n = 1$ - *останавливаемся, иначе рекурсивно*

- *сортируем первую половину в обратном порядке;*
- *сортируем вторую половину в прямом порядке;*
- *битонное слияние.*

Размер - $O(n(\log n)^2)$, *глубина* $O((\log n)^2)$.

Задача 23. Возможна ли неадаптивная сортировка схемой размера $o(n(\log n)^2)$? $O(n \log n)$?

Алгоритм 67. ***Схема AKS.*** *Размер* - $O(n \log n)$, *глубина* - $O(\log n)$. *Использует глубокие понятия теории графив (экспандеры). Асимптотически оптимальна, но имеет огромный константный множитель.*

Модели параллельных вычислений.

Алгоритм 68. ***Parallel Random Access Machine (PRAM).*** *Простая, идеализированная модель общих параллельных вычислений, включающая неограниченное количество процессоров (1 операция за единицу времени), и глобальную общую память. Вычисления полностью синхронны. Само вычисление PRAM - последовательность параллельных шагов, коммуникауя и синхронизация считаются "бесплантыми", однако эта хрень очень трудно реализуется на практике.*

Варианты PRAM:

- *concurrent/exclusive read;*
- *concurent/exclusive write.*

Алгоритм 69. ***Bulk-Synchronous Parallel (BSP) computer.*** *Простая, реалистичная модель общих параллельных вычислений - масштабируемая, переносимая, предсказуемая. Включает p процессоров, каждый с локальной памятью (1 операция за единицу времени), коммуникационную среду, состоящую из сети и (возможно) внешней памяти (1 единица данных за g единиц времени), и маханизм барьерной синхронизации (не чаще 1 раза за l единиц времени).*

Примеры коммуникационной среды:

- *g - communication gap (inverse bandwidth), время (в худшем случае) для единицы данных войти в сеть или покинуть сеть;*
- *l - latency, время (в худшем случае) для единицы данных быть переданной внутрь сети.*

Вычисление BSP - последовательность параллельных супершагов. Сначала происходят вычисления / коммуникация внутри шага (коммуникация включает обмен данными с внешней памятью), а затем происходит синхронизация между супершагами.

Альтернативная модель - CSP, взаимодействующие последовательные процессы.

Примечание 20. Рассмотрим композиционную модель стоимости вычислений. Для конкретного процессора *proc* на супершаге *sstep*:

- $comp(sstep, proc)$ - объём локальных вычислений и операций над локальной памятью процессора *proc* на супершаге *sstep*;
- $comm(sstep, proc)$ - объём данных, отправленных и полученных процессором *proc* на супершаге *sstep*.

Для компьютера BSP в целом на одном супершаге *sstep*:

- $comp(sstep) = \max_{0 \le proc < p} comp(sstep, proc)$;
- $comm(sstep) = \max_{0 \le proc < p} comm(sstep, proc)$;
- $cost(sstep) = comp(sstep) + comm(sstep) \cdot g + l$.

Для вычисления BSP, состоящего из sync супершагов:

- $comp - \sum_{0 \le sstep < sync} comp(sstep)$;
- $comm = \sum_{0 \le sstep < sync} comm(sstep)$;
- $cost = comp + comm \cdot g + sync \cdot l$.

Входные или выходные данные хранятся во внешней памяти, стоимость ввода / вывода включена в $comm$.

Примечание 21. Что касается разработки алгоритмов для BSP. Нас интересует минимизация $comp$, $comm$, $sync$ как функций от n, p.

Соглашения:

- размер задачи $n \gg p$ (*допуск*);
- входные или выходные во внейшней памяти, ввод или вывод - односторонние коммуникации.

Баланс вычислений:

- *work-optimal* $comp = O(\frac{seq\ work}{p})$.

Баланс коммуникации:

- цель *scalable* $comm = O(\frac{iinput+output}{p^c})$, $o < c \le 1$;
- в идеале $fully - scalable$, $c = 1$.

Крупоблочность:

- цель - $sync$, не зависящая от n (может зависеть от p);
- ещё лучше - *quasi-flat* $sync = O((\log p)^{O(1)})$;
- в идеале - *flat* $sync = O(1)$.

Всякая хуйня.

Алгоритм 70. *Схема на основе сбалансированного двоичного дерева - tree(n), 1 вход, n выходов (или наоборот). Размер - $n-1$, глубина - $\log n$. Каждый узел вычисляет произвольно заданную операцию за $O(1)$. Последовательная сложность - $O(n)$, сложность на PRAM - $O(\log n)$. Строим равномерно распределяя одинаковые количества операций на процессоры (n/p).*

Описанный на лекциях BSP-алгоритм вычисления сбалансированнного дерева полностью оптимален:

- *оптимальное comp* $= O(n/p) = O(\frac{sequential\ work}{p})$;
- *оптимальное comm* $= O(n/p) = O(\frac{input/output\ size}{p})$;
- *оптимальное sync* $= O(1)$.

Задача 24. Задача *префиксного накомпления.* Пусть нам дан массив $a = [a_0, \dots, a_{n-1}]$. Нам нужно вычислить $b_{-1} = 0$, а точнее $b_i = a_i + b_{i-1}$ для $0 \le i < n$. В более общем виде нуль заменяется на какой-то элемент, а сложение - на любой ассоциативный оператор.

Алгоритм 71. *Схема префиксного накопления - prefix(n) - красивая вещь, которая позволяет справиться с поставленной задачей за размер $2n-2$, глубину $2\log n$ и сложность на PRAM - $O(\log n)$.*

Рассматривая префиксное накопление на BSP, граф prefix(n) состоит Из

- *верхнего подерева, вычисляемого от листьев к корню tree(n);*
- *переноса значений от узлов верхнего поддерева к узлам нижнего;*
- *нижнего поддерева, вычисляемого от корня к листьям tree(n).*

Оба поддерева можно вычислить предыдущим алгоритмом. Перенос значений задаёт comm $= O(n/p)$, *comp* $= O(n/p)$ *и sync* $= O(1)$, *предполагая допуск* $n \ge p^2$.

Задача 25. Даны массивы $a = [a_0, \dots, a_{n-1}]$, $b = [b_0, \dots, b_{n-1}]$. Нам нудно вычислить $c_{-1} = 0$, а ещё лучше $c_i = a_i + b_i \cdot c_{i-1}$ для $i \le 0 < n$. Ещё это можно записать в виде произведений матриц, но это так, чуть более красивая форма. Пусть A_i - матрица 2×2, в верхней строке которой 1, 0, а во второй - a_i, b_i. C_i - матрица 2×1, сверху - 1, снизу - c_i. Тогда суть рассматриваются рекурренты $C_k = A_k \dots A_1 A_0 \cdot C_{-1}$.

Алгоритм 72. *Эта задача - суть предыдущая с префикснымми накоплениями. С итоговым размером $O(n)$, глубиной $O(\log n)$.*

Задача 26. Побитово складываем массивы. И переносим, а потом ещё и параллелим.

Алгоритм 73. *Параллельное двиочное сложение при помощи булевой логики. Если $x + y = z$, и каждый из них представлено соответственно как $[x_{n-1}, \dots, x_0]$ (понятно, как другие). Тогда если $c - [c_{n-1}, \dots, c_0]$, где c_i - i-ый бит переноса, то $x_i + y_i + c_{i-1} = z_i + 2c_i$, для $0 \le i < n$.*

Определим тогда битовые массивы $u = [u_{n-1}, \dots, u_0]$, $v = [v_{n-1}, \dots, v_0]$, $u_i = x_i \wedge y_i$, $v_i = x_i \oplus y_i$, $0 \le i \le n$. Тогда $z_0 = v_0$, а далее $z_{k-1} = v_{k-1} \oplus c_{n-2}$, $c_0 = u_0$, а далее - $c_{n-1} = u_{n-1} \vee (v_{n-1} \wedge c_{n-2})$. Это получилась схема сумматора со сквозным переносом размера и глубины $O(n)$.

Хотелось бы также её распареллелить. Для этого скажем, что $c_i = u_i \vee (v_i \wedge c_{i-1})$. *Вычисляем массив* c *при помощи линейного рекуррентного соотношения со входами* u, v *и операторами* $\vee, \wedge$*: размер* $O(n)$*, глудина -* $O(\log n)$*; массив* z *- дополнительный размер* $O(n)$*, глубина* $O(1)$*. Получилась схема сумматора с ускоренным переносом размера* $O(n)$*, глубины* $O(\log n)$.

Оптимизационные задачи и приближённые алгоритмы.

Задача 27. Задача *линейного программирования (LP)*: оптимизировать (максимизировать или минимизировать) многомерную вещественную линейную *целевую функцию* при линейных *ограничениях* (неравенствах или равенствах).

Вектор $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ - *допустимое решение* задачи LP, если оно удовлетворяет всем ограничениям. Задача LP разрешима, если у неё есть допустимое решение. Допустимое решение *оптимально*, если целевая функция достигает на нём (нестрогого) максимума (соответственно, минимума).

Стандартная форма задачи LP состоит из максимизации выражения $\sum c_i x_i$ при m условиях $\sum_j a_{ij} x_j \leq b_i$, и $x_1, x_2, \ldots, x_n \geq 0$. n переменных, $m + n$ лграничений, включая n ограничений неотрицательности (будем их опускать, предполагая по умолчанию). *Каноническая форма* задачи LP - то же самое, что и стандартная, только в куче тех рассматриваемых сумм у нас не знаки неравенства, а значи равенства. Чтобы привести задачу от стандартной формы к канонической вводятся вспомогательные переменные $s_i = x_{n+i}$.

Теорема 13. *(Основная теорема линейного программирования). Задача LP в стандартной форме имеет оптимальное решение, если она разрешима и ограничена. Если задача LP имеет оптимаьное решение, то хотя бы одно из них - вершина допустимого многогранника.*

Алгоритм 74. *(Симплекс-метод). Последовательные перемещения по вершинам допустимого могогранника, улучшающие целевую функцию. Общая идея симплекс-метода состоит в том, что мы берём одну из вершин в качестве начального решения, и если решение не оптимально, но оптимальное решение существует, то в какой-то из соседних вершин значение целевой функции лучше, чем в текущей - перемещаемся в эту вершину и повторяем.*

Определение 16. Задачи называются *двойственными*, если поменять местами b_i и c_j, и перевернуть неравенства в условиях. И если первичная задача разрешима, то и двойственная также разрешима и их оптимальные решения равны.

Всякие схемы про три задачи из билетов начинаются примерно на сотом слйде мистера Тискина, а поэтому я добавлю только некоторые важные определения и формулировки.

Определение 17. *Сохранение потока*: для любого $u \in V \backslash \{s, t\}$ суммарный входящий поток в u должен равняться суммарному исходящему потоку из u.

Теорема 14. *Следующие утверждения равносильны:*

- f *- максимальный поток;*
- f *не имеет увеличивающихся путей;*
- *существует* (s, t)*-разрез со значением* $c(S, T) = |f|$.

Алгоритм 75. *(Алгоритм Форда-Фалкерсона). Начинаем с произвоьлного допустимого потока (например, нулевого), затем*

- *строим для текущего потока остаточнкую сеть;*
- *ищем увеличивающий путь в остаточной сети. если его нет, то поток максимальный;*
- *включаем увеличивающий путь в поток, добавляя его значение к прямым рёбрам и вычитая из обратных;*
- *повторяем для нового потока.*

Теорема 15. *В целочисленной задаче о максимальном потоке значение оптимального потока $|f^*|$ - натуральное число. Алгоритм Форда-Фалкерсона находит его за $|f^*|$ итераций. Общая трудоёмкость алгоритма - $|E||f^*|$.*

Edward Hirsh

Проверки равенств

Алгоритм 76. *(Алгоритм Фрейвальдса). Если нам хочется проверить верно ли перемножены две матрицы ($A \times B = C$?) размера $n \times n$, то можно выполнить следующий алгоритм:*

Сгенерируем случайный вектор r из нулей и единиц размера $n \times 1$. Вычисляем $P = A \times (Br) - Cr$, и если получился нулевой столбик, то алгоритм скажет, TRUE, иначе - FALSE.

Если произведение выполнено, то алгоритм всегда выведет TRUE, но если произведение не выполнено, то вероятность того, что алгоритм выдаст TRUE - не больше $\frac{1}{2}$. Таким образом, если выполнить его k раз, и на каждой итерации получать TRUE, то полученный алгоритм будет работать со скоростью $O(kn^2)$ и вероятность ошибки будет составлять не больше 2^{-k}.

Алгоритм 77. *(Проверка равенства многочленов; Лемма Шварца-Зиппеля). Пусть $P \in F[x_1, x_2, \ldots, x_n]$ - ненулевой сногочлен степени $d \geq 0$ над полем F. Пусть S - конечное подмножество F, и пусть элементы $r_1, r_2, \ldots, r_n$ были выбраны из S равномерно и независимо друг от друга. Тогда $\Pr[P(r_1, \ldots, r_n) = 0] \leq \frac{d}{|S|}$.*

Доказательство. Случай одной переменной очевиден, а далее - доказываем по индукции по количеству переменных. □

Задача 28. Интересно нам сравнить две строки a, b (которые можно считать побитовыми), затратив как можно меньше информации на это сравнение. Основная идея - сравнивать не сами строки, а функции от них.

Определение 18. *Расстояние* между строками a, b будем считать количество несовпадающих у них битов.

Определение 19. Под *editing distance* между a, b будем понимать минимальное количество операций редактирования, необходимых для преобразования строки a в строку b (длин $m \leq n$ соответственно). Операциями редактирования мы считаем: вставку, удаление или уничтожение бита, а также перемещение сплошного блока битов.

Алгоритм 78. *(Алгоритм Рабина-Карпа). Определим $b(i) = b_i b_{i+1}..b_{i+m-1}$. Необходимо провести $n-m+1$ сравнений строк на равенсто, а это мы уже умеем делать: будем сравнивать $(a \mod p)$ и $(b(i) \mod p)$. Но можно упростить задачу, вычисляя $(b(i) \mod p)$ через $(b(i-1) \mod p)$.*

$$\begin{aligned} b(i) &= b_i + 2b_{i+1} + .. + 2^{m-1} b_{i+m-1} \\ b(i-1) &= 2b_i + 2^2 b_{i+1} + .. + 2^{m-1} b_{i+m-2} + b_{i-1} \\ \Rightarrow b(i) &= \frac{b(i-1) - b_{i-1}}{2} + 2^{m-1} b_{i+m-1} \end{aligned}$$

Опять будем брать простое число $p \in [2, r]$, где $r = n^2 m \log n^2 m$. Тогда

$$P_{ошибки} \leq \frac{m}{\frac{\tau}{\log \tau}} \leq \frac{2m \log n^2 m}{n^2 m \log n^2 m} = O\left(\frac{1}{n^2}\right) \tag{1}$$

Можно вообще избавить этот алгоритм от необходимости ошибаться. Если $a \equiv_p b(i)$, то просто проверим равенство $a = b$. Этот алгоритм уже не ошибается.

Математическое же ошидание его работы посчитать легко:

$$\mathbf{E}T(n,m) \leq mn * \frac{1}{n} + (m+n)(1-\frac{1}{n}) = O(m+n). \tag{2}$$

Проверка простоты.

Теорема 16. *(Теорема Эйлера). Если n - нечётное, то $\left(\frac{a}{n}\right) a^{\frac{n-1}{2}} \equiv_n 1$, для всех $a \in \{1,\ldots,n-1\}$.*

Алгоритм 79. *(Алгоритм Соловея-Штрассена). Получив на входе число n, случайно выбирается $\in \{1,\ldots,n-1\}$ и проверить условие теоремы Эйлера - то есть, вычислить $\left(\frac{a}{n}\right) a^{\frac{n-1}{2}} \mod n$, и если получилось 1, то сообщить, что n - простое.*

Минимальное остовное дерево.

Остовное дерево — это дерево, состоящее из некоторых ребер графа и содержащее все его вершины. Требуется построить остовное дерево с минимальным суммарным весом ребер. (Считаем, что все ребра разного веса: этого можно добиться, если к равным ребрам добавить достаточно малые ε_i.)

Сперва рассмотрим простой детерминированный алгоритм: **алгоритм Borůvka**. Выберем у каждой вершины ребро наименьшего веса. После этого в полученном графе каждую компоненту связности стянем в вершину. При каждой такой итерации число вершин уменьшается вдвое. Так что сложность этого алгоритма $O(m \log n)$. Фактически, этот алгоритм мало чем отличается от алгоритма Краскала, но его можно улучшить следующим образом:

Алгоритм 80. *По графу G_1 с n вершинами и m ребрами строим минимальный остовный лес (ибо граф может быть несвязен) для него.*

1. *Применим к G_1 3 шага алгоритма Borůvka - получим граф G_2 и некое множество ребер S, которые принадлежат остовному дереву. В графе G_2 максимум $\frac{n}{8}$ вершин.*
2. *Из G_2 выкинем примерно половину ребер (именно, каждое ребро выкинем с вероятностью $\frac{1}{2}$) — это будет граф $G_2(\frac{1}{2})$ (в нем $\leq \frac{n}{8}$ вершин, а мат. ожидание количества ребер $\leq \frac{m}{2}$.*
3. *Для $G_2(\frac{1}{2})$ применяем алгоритм рекурсивно; получим F — минимальный остовный лес для этого графа.*

   Далее требуется определение. Пусть в графе H есть остовное дерево (или лес) T. Ребро, соединяющее вершины v и w называется **тяжелым в** H **относительно** T*, если ее вес больше веса максимального ребра на пути из v в w в T. Иначе ребро называется* **легким**. *Понятно, что тяжелые ребра нас не интересуют.*

   Задача 29. Найти все легкие ребра за линейное время.

4. *Пусть V_2 — множество ребер G_2, легких относительно F. Возьмем только их — получим граф G_3.*

5. Рекурсивно обрабатываем G_3 и выдаем полученный результат.

□

Пусть $T(n,m)$ — мат. ожидание времени работы нашего алгоритма на графе с n вершинами и m ребрами (более строго, максимум этого мат. ожидания по всем графам). На рекурсивный вызов от $G_2(\frac{1}{2})$ тратится время $T(\frac{n}{8},\frac{m}{2})$. Покажем, что на рекурсивный вызов от G_3 тратится время $T(\frac{n}{8},\frac{n}{4})$.

Лемма 1. *Пусть $G(p)$ — граф, полученный из какого-то графа G выкидыванием каждого ребра с вероятностью $1-p$, и F — минимальный остовный лес в $G(p)$. Тогда мат. ожидание числа легких ребер в G относительно F не превосходит $\frac{n}{p}$, где n — число вершин.*

Доказательство. Лес F будем строить одновременно с графом $G(p)$. Упорядочим ребра G по возрастанию весов. Очередное ребро с вероятностью p добавляем в $G(p)$. Если добавленное ребро соединяет в F разные компоненты связности, то добавляем его в F (как в алгоритме Краскала). Таким образом, в F попадет $n-1$ ребро. Легкими в G будут эти $n-1 \le n$, а также те, которые попали бы в F, если бы их случайно не отбросили. Так как в среднем "отсев проходит" каждое $\frac{1}{p}$-е ребро, то всего легких ребер будет в среднем не более, чем $\frac{n}{p}$. □

Утверждение 22. Доказать лемму более формально.

Лемма 2. *Мат. ожидание количества ребер в графе G_2, легких относительно леса F, не превосходит $n/4$.*

Доказательство. В этом графе $\frac{n}{8}$ вершин, а $p=\frac{1}{2}$. Применяем предыдущую лемму. □

Таким образом, для мат. ожидания времени работы нашего алгоритма, очевидно, верно соотношение:

$$T(n,m) \le T\left(\frac{n}{8},\frac{m}{2}\right) + T\left(\frac{n}{8},\frac{n}{4}\right) + c(m+n).$$

Утверждение 23. Доказать, что $T(n,m) = O(n+m)$.

Предонлайн хрень.

Оценки Чернова.

Основной случай оценки Чернова для случайной величины X достигается применением неравенства Маркова к e^{tX}. Для каждого $t>0$

$$P(X \ge a) = P(e^{t\cdot X} \ge e^{t\cdot a}) \le \frac{\mathrm{E}\left[e^{t\cdot X}\right]}{e^{t\cdot a}}.$$

Когда X является суммой n случайных величин $X_1,\dots,X_n$, для любого $t>0$

$$P(X \ge a) \le e^{-ta}\mathrm{E}\left[\prod_i e^{t\cdot X_i}\right].$$

В частности, оптимизируя по t и предполагая, что X_i независимы, мы получаем

$$P(X \ge a) \le \min_{t>0} e^{-ta}\prod_i \mathrm{E}\left[e^{tX_i}\right].$$

Аналогично

$$P(X \le a) = P\left(e^{-tX} \ge e^{-ta}\right)$$

и, таким образом,

$$P(X \le a) \le \min_{t>0} e^{ta} \prod_i \mathrm{E}\left[e^{-tX_i}\right].$$

Конкретные значения оценок Чернова получаются вычислением $\mathrm{E}\left[e^{-t\cdot X_i}\right]$ для конкретных величин X_i.

On-line shit.

Задача кэширования (paging problem.)

Рассмотрим следующую задачу. Есть два вида памяти - дисковая, большая по объему, но медленная, и кэш, быстрая, но очень небольшая. Вся память разбита на блоки. Пусть в кэш-памяти k блоков, а на диске, соответственно, намного больше. К нам поступают запросы на те или иные блоки, и мы должны сразу же их обработать. Есть два варианта: либо этот блок уже есть в кэше, либо его там нет. Если он в кэше есть, то в этом случае мы почти не тратим время. И тогда время работы алгоритма можно измерять количеством обращений к винчестеру.

Алгоритмы отличаются друг от друга реакцией на запросы отсутствующих в кэше блоков. Понятно, что мы должны запрашиваемый блок подгрузить в кэш, непонятно только, какой из уже присутствующих там блоков мы должны для этого выгрузить. Конечно, хотелось бы выгрузить "ненужный" блок, но поскольку мы обрабатываем запросы в реальном времени, мы не знаем, какой блок нам понадобится в дальнейшем.

Встает и другой вопрос: как сравнить, какой алгоритм лучше.

Путь A — алгоритм, а r — последовательность запросов. Стоимость обработки последовательности запросов алгоритмом — $cost_A(r)$ — это количество обращений к диску. Пусть *opt* — некий оптимальный алгоритм. Тогда алгоритм A называется c-оптимальным (c-competitive), если для любого r, $cost_A(r) \le c \cdot cost_{opt}(r) + c_1$. Подразумевается, что c — константа относительно r, но она может зависеть от k или других параметров. Но это — для детерминированных алгоритмов, для вероятностных же рассматривается на стоимость, а ее математическое ожидание.

Мы будем рассматривать случай, когда A — вероятностный online алгоритм, а *opt* — детерминированный *offline* алгоритм ("oblivious adversary"), т.е. он *заранее* всю знает последовательность запросов r.

Алгоритм 81 (MARKER)**.** *Элементы кэша помечаются 0 или 1. Все время работы разделяется на периоды. В начале каждого периода все элементы кэша помечены 0.*

Приходит запрос. Если это запрос на блок, который уже есть в кэше, то он помечается 1. Если запрашивают элемент, которого нет в кэше, то мы случайным образом выбираем из непомеченных (то есть помеченных 0) блок, куда мы будем подгружать требуемый. Подгрузив, мы помечаем его 1.

Рано или поздно у нас не останется блоков, помеченных 0. В этом состоянии мы можем работать, пока у нас просят блоки, находящиеся в кэше. Как только поступит запрос на элемент, не содержащийся в кэше, мы обнулим все пометки и начнем новый период. □

Теперь попробуем оценить, насколько же хорош этот алгоритм. Разделим все поступающие запросы на три вида: помеченные, устаревшие и чистые. Помеченный запрос — это запрос на блок, образ которого находится в кэше и помечен 1. Устаревший — это запрос на блок, образ которого находится в кэше и помечен 0, или на блок, которого в кэше нет, но который там был в предыдущем периоде. И соответственно, чистый запрос — это запрос на блок, которого в кэше нет и не было в предыдущем периоде.

Обозначим l_i количество чистых запросов за i-тый период; $d_{I,i}$ — разница между кэшами A и *opt*, то есть можность симметрической разности этих двух множеств (количество элементов, входящих ровно в одно из них), в начале i-того периода, а $d_{F,i}$ — в конце.

Оценим снизу количество загрузок, которые сделает *opt*. С одной стороны, это не меньше чем $(l_i - d_{I,i})$, так как l_i блоков алгоритм А обязан подгрузить, и *opt* может выиграть у A только за счет того, что в начале периода у него в кэше уже будут блоки, на которые поступят запросы. А так как разница между кэшами в начале периода составляет $d_{I,i}$ элементов, то и выиграть *opt* может не более $d_{I,i}$ загрузок.

С другой стороны, число загрузок не менее $d_{F,i}$, так как все блоки, лежащие в кэше алгоритма A к концу периода, были запрошены. Следовательно, они должны были побывать и в кэше алгоритма *opt*, и если какого-то из них нет, значит, на его место был загружен другой блок, а нет там ровно $d_{F,i}$ элементов из кэша A.

Таким образом, число загрузок, которые должен сделать оптимальный алгоритм, составляет не менее

$$\max(l_i - d_{I,i}, d_{F,i}) \geq \frac{l_i - d_{I,i} + d_{F,i}}{2}.$$

Это — за один период. Просуммировав по все периодам, получим, что число загрузок, которые должен сделать *opt* за все периоды не менее

$$\sum_i \frac{l_i - d_{I,i} + d_{F,i}}{2} = \sum_i \frac{l_i}{2} + \frac{d_{F,n}}{2} \geq \frac{L}{2},$$

где $L = \sum_i l_i$.

Теперь оценим сверху, сколько загрузок сделает алгоритм A. Ясно, что на чистые запросы ему придется подгружать блоки, на помеченные — нет. Сложнее дело обстоит с устаревшими запросами. Во-первых, сколько таких запросов может быть? Ответ: $k - l$. При устаревшем запросе мы обращаемся к блоку, который на предыдущем периоде был в кэше, но есть ли он там сейчас, мы сказать не можем. Он мог быть выгружен, когда пришел чистый запрос, либо устаревший запрос на блок, который до этого был уже выгружен.

Рассмотрим **первый** устаревший запрос. Поскольку нам надо оценить вероятность сверху, то мы можем рассмотреть наиболее неблагоприятный вариант, когда все чистые запросы произошли до данного. Тогда вероятнось выгрузить нужный нам блок будет наибольшей. Посчитаем, какова вероятность в этом случае оставить наш блок в кэше. При первом чистом запросе мы оставим наш блок с вероятностью $(k-1)/k$, при втором — $(k-2)/(k-1)$ и т.д. Вероятность того, что мы не выгрузим нужный блок при последнем, l-том чистом запросе, равна $(k-l)/(k-l+1)$. Перемножая эти вероятности, получим вероятность того, что мы оставим наш блок в кэше: $(k-l)/k$. Таким образом, вероятность того, что мы выгрузим его, не превосходит l/k.

Общий случай оставляется для самостоятельных раздумий. Именно, утверждается, что вероятность выгрузить нужный нам блок при **j-том** устаревшем запросе не превосходит $l/(k-j+1)$.

Таким образом, математическое ожидание количества загрузок устаревших блоков меньше или равно

$$l/k + \ldots + l/(k-(k-l)+1) = l \cdot (1/k + \ldots + 1/(l+1)) = l \cdot (H_k - H_l + 1),$$

где $H_n = 1 + 1/2 + \ldots + 1/n$. Вспомнив, что мы должны еще загрузить l блоков при чистых запросах, мы получим, что

$$\mathbf{E}(\text{число загрузок}) \leq l \cdot (1 + H_k - H_l + 1) \leq l \cdot H_k.$$

Это за один период. Просуммируем по всем периодам. Получим: $\mathbf{E}cost_A = H_k \cdot L$. Но $cost_{opt} = L/2$, и мы доказали следующую теорему.

Теорема 17. *Алгоритм MARKER является $2H_k$-оптимальным.*

Определение 20. Рассмотрим ситуацию, когда adversary не только знает все запросы наперед, но и может их придумывать в зависимости от наших действий.

Adaptive offline adversary так и поступает; придумав все запросы, он порождает свой вариант их обработки стоимостью $cost_{opt}$. Заметим, что хотя он порождает новые запросы, зная наши ответы на предыдущие, он не может предугадать наших *будущих* действий, ибо они зависят от выпавших нам случайных чисел. Естественно, в определении *c-оптимальности относительно adaptive offline adversary* фигурируют не все r, а лишь r, порождаемые adversary.

Adaptive online adversary отличается от adaptive offline adversary тем, что он обязан обрабатывать (“порождать свой вариант обработки”) придумываемые им запросы online.

Задача 30. Оценить силу алгоритма MARKER относительно adaptive online и adaptive offline или показать, что этого сделать нельзя.

Задача 31. Придумать алгоритм лучше.