Матосновы алгоритмов

Мастера конспектов
на основе лекций А. С. Охотина и А. В. Тискина

22 января 2020 г.

Основные моменты.

Содержание

1 Лекция 1.

1.1 Быстрая сортировка.

Алгоритм 1. ***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

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

□

1.2 Сортировка кучей.

Алгоритм 2. ***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 1. Работает за $O(n \log n)$ (построение дерева - $O(\log n)$, вынесение вершин - $O(n)$), можно разогнать оценку до $2n$.

1.3 Скорость сортировки.

Теорема 2. *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 3. ***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 4. ***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.*

1.4 Нахождение i-го по величине элемента массива.

Алгоритм 5. ***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.*

1.5 Метод динамического программирования.

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

Алгоритм 6. *Начинаем с первого, и делаем полный перебор. Говнище*

Алгоритм 7. ***Жадный алгоритм.*** *Отпиливаем самый дорогой кусок, затем опять самый дорогой из возможных, и так далее. Не самый оптимальный.*

Алгоритм 8. ***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.

2 Лекция 2.

2.1 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.

2.2 Нахождение наибольшей общей подпоследовательности.

Определение 1. 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 9. ***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 10. ***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 $f(m,n)$.

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

□

2.3 Поиск в ориентированном графе.

Алгоритм 11. ***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 2. 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 3.

- 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 12. ***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 4. 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 5. Если в ациклическом графе есть дуга (u,v), то время завершения v ментше, чем время завершения u.

3 Лекция 3.

3.1 Окончание поисков в орграфе.

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

Алгоритм 13. ***Алгоритм Косараджу-Шарира.*** *Спервая 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 6. 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. □

3.2 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 14. ***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 7. После i-ой итерации внешнего цикла алгоритм Беллмана-Форда находит все пути наименьшего веса длины не более чем i.

Доказательство. Индукция по i. □