Матосновы алгоритмов

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

22 января 2020 г.

Основные моменты.

Содержание

1 Лекция 2. **3**
1.1 Быстрая сортировка. 3
1.2 Сортировка кучей. 3
1.3 Скорость сортировки. 3
1.4 Нахождение i-го по величине элемента массива. 4
1.5 Метод динамического программирования. 4

2 Лекция 3. **4**
2.1 Продолжение динамического программирования. 4
2.2 Нахождение наибольшей общей подпоследовательности. 5
2.3 Поиск в ориентированном графе. 6

3 Лекция 4. **7**
3.1 Окончание поисков в орграфе. 7
3.2 Поиск в алгоритме с весами. 7
3.3 Окончание поиска с весами. 8
3.4 Нахождение минимального остовного дерева. 9
3.5 Структура данных для непересекающихся множеств. 10

4 Лекция 5. **11**
4.1 Пути между всеми парами вершин. 11
4.2 Быстрое умножение матриц. 11

1 Лекция 2.

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 3.

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 соответствующиих 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 4.

3.1 Окончание поисков в орграфе.

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

Алгоритм 13. ***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 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. □

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 верщин цикла можно записать не условие улучшения и сложить по все парам. □

Алгоритм 15. ***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.

3.3 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 16. ***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.

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

Определение 2. *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 8. 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 18. ***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 19. ***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 9. 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. □

3.5 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 20. ***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 10. 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 21. *Алгоритм пишется через представление компонент связности в виде структуры данных для непересекающихся множеств.*

4 Лекция 5.

4.1 Пути между всеми парами вершин.

Задача 9. Дан ориентированный граф $G = (V, E)$, где $V = \{1, \dots, n\}$, а $E \subseteq V \times V$. Ствaится 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 22. ***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 11. 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 23. ***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 12. 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 24. ***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.

4.2 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 - а 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 $O(n^{\log_b a})$. □

Задача 11. Умножение булевых матриц.

Алгоритм 25. *Можно записать матрицу в виде кучи нулейи 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 26. ***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 обохначаемыми 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 $O\left(\frac{n^3}{\log n}\right)$.