ОТВЕТИКИ НА ЭКЗАМЕНАЦИОННЫЕ ВОПРОСЫ ПО ДИСЦИПЛИНЕ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ» ДЕЛАЛА ПОД СЕБЯ ЧЕТА-НЕ НРАВИТСЯ ИДИ В ЖОПУ

1. Хеш-таблицы. Хеш. Вычисление хеша. Коллизии и разрешение их.

**Хеш-таблица** — это структура данных, в которой все элементы хранятся в виде пары ключ-значение, где:

* **ключ**— уникальное число, которое используется для индексации значений;
* **значение**— данные, которые с этим ключом связаны.

Главное свойство hash-таблиц — все три операции: вставка, поиск и удаление — в среднем выполняются за время *O*(1), среднее время поиска по ней также равно *O*(1) и *O*(n) в худшем случае.

В хеш-таблице обработка новых индексов производится при помощи ключей. А элементы, связанные с этим ключом, сохраняются в индексе(некое натуральное число). Этот процесс называется **хешированием**.

Пусть k — ключ, а h(x) — хеш-функция.

Тогда h(k) в результате даст индекс, в котором мы будем хранить элемент, связанный с k.

Хорошие хеш-функциями, которые помогут уменьшить число коллизий:

* Метод деления - Если k — ключ, а m — размер хеш-таблицы, то хеш-функция h() вычисляется следующим образом: h(k) = k mod m
* Метод умножения - h(k) = ⌊m(kA mod 1)⌋, где А – произвольная константа
* Универсальное хеширование - хеш-функция выбирается случайным образом и не зависит от ключей.

**Где применяются**

* Когда необходима постоянная скорость поиска и вставки.
* В криптографических приложениях.
* Когда необходима индексация данных.

Когда хеш-функция генерирует один индекс для нескольких ключей, возникает конфликт: неизвестно, какое значение нужно сохранить в этом индексе. Это называется **коллизией хеш-таблицы**.

Есть несколько методов борьбы с коллизиями:

* метод цепочек;
* метод открытой адресации: линейное и квадратичное зондирование.

**Метод цепочек.** Суть этого метода проста: если хеш-функция выделяет один индекс сразу двум элементам, то храниться они будут в одном и том же индексе, но уже с помощью двусвязного списка.

Если j — ячейка для нескольких элементов, то она содержит указатель на первый элемент списка. Если же j пуста, то она содержит NIL.

**Метод открытой адресации.** В отличие от метода цепочек, в открытой адресации несколько элементов в одной ячейке храниться не могут. Суть этого метода заключается в том, что каждая ячейка либо содержит единственный ключ, либо NIL.

Существует несколько видов открытой адресации:

#### a) Линейное зондирование

Линейное зондирование решает проблему коллизий с помощью проверки следующей ячейки. h(k, i) = (h′(k) + i) mod m, где

* i = {0, 1, ….}
* h'(k) — новая хеш-функция

Если коллизия происходит в h(k, 0), тогда проверяется h(k, 1). То есть значение i увеличивается линейно.

Проблема линейного зондирования заключается в том, что заполняется кластер соседних ячеек. Это приводит к тому, что при вставке нового элемента в хеш-таблицу необходимо проводить полный обход кластера. В результате время выполнения операций с хеш-таблицами увеличивается.

#### b) Квадратичное зондирование

Работает оно так же, как и линейное — но есть отличие. Оно заключается в том, что расстояние между соседними ячейками больше (больше одного). Это возможно благодаря следующему отношению:

h(k, i) = (h′(k) + c1i + c2i2) mod m,где

* c1 и c2 — положительные вспомогательные константы,
* i = {0, 1, ….}

#### c) Двойное хэширование

Если коллизия возникает после применения хеш-функции h(k), то для поиска следующей ячейки вычисляется другая хеш-функция.

h(k, i) = (h1(k) + ih2(k)) mod m

1. Графы. Представление графа. Матрица смежности. Измерение размера графа.

**Граф** — это структура, представляющая собой набор объектов, в котором некоторые пары объектов в некотором смысле «связаны».

Объекты, называемые **вершинами** (также называемыми *узлами* или *точками*).

Каждая из связанных пар вершин называется **ребром**.

**Параллельные вершины** - Два или более ребра, соединяющие одну и ту же пару вершин.

**Петля** - Ребро, которое начинается и заканчивается в одной вершине.

Графы бывают двух типов: ориентированные и неориентированные. Если ребра в графе ориентированы, т. е. указывают только в одном направлении, граф **называется ориентированным графом** или **орграфом**. Если ребра в графе не имеют направления, граф называется**неориентированным графом.**

Граф, в котором каждое ребро имеет числовой «вес», называется **взвешенным графом**.

Вершины ***u*** и ***v*** называются ***смежными***, если ***u*** и ***v*** соединены некоторым ребром.

Количество ребер, инцидентных вершине, называется ***степенью вершины*** или ***deg(v)***. Вершина инцидентна ребру, если эта вершина является одной из двух вершин, которые соединяет ребро. Например, град(3) = 3

**Листовая вершина**— это вершина с deg(v) = 1, например, 5 — листовая вершина.

**Изолированная вершина**— это вершина с deg(v) = 0 , например, 6 — изолированная вершина.

Граф называется ***связным***, если в нем нет вершины с deg = 0.

Каждый связный подграф называется компонентом.

**Путь** в графе — это последовательность ребер, соединяющая последовательность вершин. Например, 4 3 2 5

**Цикл** — это **путь**, который начинается и заканчивается в одной и той же вершине. Например, 3 2 1 3

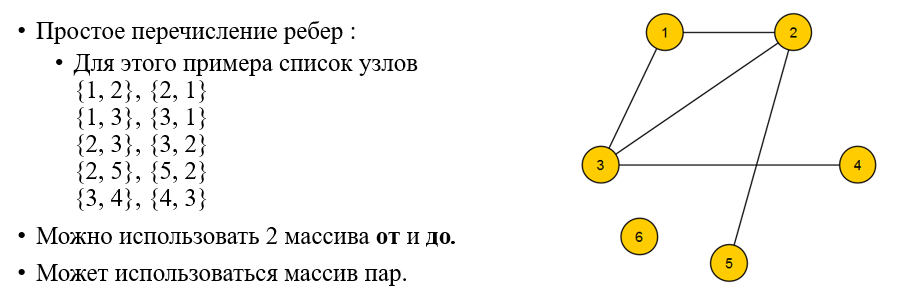
Лемма о рукопожатии:

где |E| количество ребер.

Другими словами, сумма степени всех вершин равна количеству ребер, умноженному на 2.

**3 обычных вида представления**։

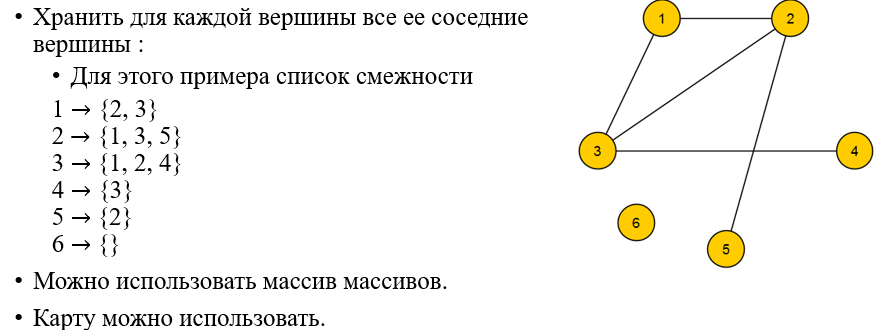
* 1. **Список ребер**



* 1. **Матрица смежности**

Если между вершинами ***i*** и ***j*** есть ребро, то (***i***, ***j***)-я матрица устанавливается в true, а в противном случае - в false. В случае ориентированного графа представление остается прежним, но ребра добавляются только в одном направлении. Для неориентированного графа (***i***, ***j***) **всегда** равно (***j***, ***i***). В случае направленного такого правила нет.

* 1. **Список смежности**



3. Поиск в графе и его применения. Обобщенный графовый поиск. Поиск в ширину и в глубину.

Обход или поиск — это одна из фундаментальных операций, выполняемых на графах. **Поиск в ширину**(BFS) начинается с определённой вершины, затем исследуются все её соседи на данной глубине и происходит переход к вершинам следующего уровня. В графах, в отличие от деревьев, могут быть циклы — пути, в которых первая и последняя вершины совпадают. Поэтому необходимо отслеживать посещённые алгоритмом вершины. При реализации алгоритма поиска в ширину используется структура данных «очередь».

Алгоритм: BFS(*v*)

{  
 добавить ***v*** в очередь ***q***

пока ***q*** не пусто  
 взять первую вершину ***u*** из ***q***

удалить первую вершину из ***q***

пометить ***u*** как посетившую

*для всех непосещенных смежных вершин* ***w*** из ***u***

добавить ***w*** в ***q***

}

## **Применяется для:**

* определения кратчайших путей и минимальных остовных деревьев;
* индексации веб-страниц поисковыми ботами;
* поиска в соцсетях;
* нахождения доступных соседних узлов в одноуровневых сетях, таких как BitTorrent

**Поиск в глубину**начинается с определённой вершины, затем уходит как можно дальше вдоль каждой ветви и возвращается обратно. Здесь тоже необходимо отслеживать посещённые алгоритмом вершины. Для того, чтобы стало возможным возвращение обратно, при реализации алгоритма поиска в глубину используется структура данных «стек». Алгоритм поиска в глубину базируется на рекурсии, т. е. функция обхода, по мере выполнения, вызывает сама себя, что делает код в целом довольно компактным.

**DFS(st){  
 Вывести (st)  
 visited[st] ← посещена;  
 Для r=1 до n выполнять  
 Если (graph[st, r] ≠ 0) и (visited[r] не посещена) то DFS(r)**

**}**

Единственный ее параметр st – стартовый узел, передаваемый из главной части программы как аргумент. Каждому из элементов логического массива visited заранее присваивается значение false, т. е. каждая из вершин изначально будет значиться как не посещенная.

Двумерный массив graph – это матрица смежности графа. Большую часть внимания следует сконцентрировать на последней строке. Если элемент матрицы смежности, на каком то шаге равен 1 (а не 0) и вершина с тем же номером, что и проверяемый столбец матрицы при этом не была посещена, то функция рекурсивно повторяется. В противном случае функция возвращается на предыдущую стадию рекурсии.

## Применяется:

* для нахождения пути между двумя вершинами;
* для обнаружения циклов на графе;
* в топологической сортировке;
* в головоломках с единственным решением (например, лабиринтах).

Сложность для обоих алгоритмов: ***O(|V| + |E|)*** если мы используем список соединений и ***O()*** если мы используем матрицу смежности (потому что нам нужно найти каждое ребро, поэтому нужно перебрать всю матрицу).

**Обобщенный графовый поиск**. Основной принцип достаточно прост: мы снова обращаемся к описанию поиска в ширину, но теперь вместо понятия *очередь (queue)*мы употребляем обоб­щенный термин**(fringe)**для описания множества ребер, которые являются кан­дидатами для следующего включения в дерево поиска. Начав с петли исходной вершины в **fringe** и пустого дерева, до тех пор, пока **fringe** не станет пустой, выполняется следующая операция:

Берем очередное ребро из **fringe** и переносим его в дерево. Если вершина, в которую оно ведет, еще не посещалась, переходим на эту вершину и помещаем в **fringe** все ребра, которые ведут из этой вершины в вершины, на которых мы еще не были.

Эта стратегия описывает семейство алгоритмов поиска, которые обеспечивают посе­щение всех вершин и ребер связного графа *независимо*от того, какой тип обобщенной очереди используется для хранения ребер в **fringe**.

Когда для реализации **fringe** используется очередь, мы получаем поиск в ширину. Когда для реализации **fringe** используется стек, получается поиск в глубину.

4. Поиск в глубину. Топологическая сортировка. Вычисление топологического упорядочивания.

**Поиск в глубину**начинается с определённой вершины, затем уходит как можно дальше вдоль каждой ветви и возвращается обратно. Здесь тоже необходимо отслеживать посещённые алгоритмом вершины. Для того, чтобы стало возможным возвращение обратно, при реализации алгоритма поиска в глубину используется структура данных «стек». Алгоритм поиска в глубину базируется на рекурсии, т. е. функция обхода, по мере выполнения, вызывает сама себя, что делает код в целом довольно компактным.

**DFS(st){  
 Вывести (st)  
 visited[st] ← посещена;  
 Для r=1 до n выполнять  
 Если (graph[st, r] ≠ 0) и (visited[r] не посещена) то DFS(r)**

**}**

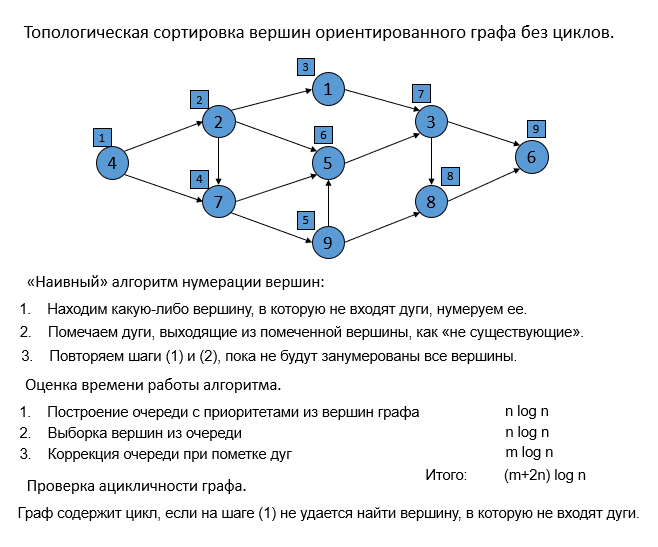
Единственный ее параметр st – стартовый узел, передаваемый из главной части программы как аргумент. Каждому из элементов логического массива visited заранее присваивается значение false, т. е. каждая из вершин изначально будет значиться как не посещенная.

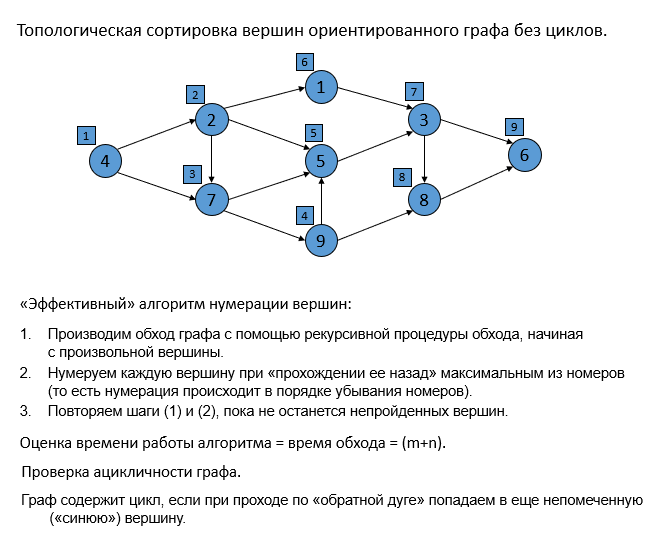
Двумерный массив graph – это матрица смежности графа. Большую часть внимания следует сконцентрировать на последней строке. Если элемент матрицы смежности, на каком то шаге равен 1 (а не 0) и вершина с тем же номером, что и проверяемый столбец матрицы при этом не была посещена, то функция рекурсивно повторяется. В противном случае функция возвращается на предыдущую стадию рекурсии.

## Применяется:

* для нахождения пути между двумя вершинами;
* для обнаружения циклов на графе;
* в топологической сортировке;
* в головоломках с единственным решением (например, лабиринтах).

Сложность для алгоритма: ***O(|V| + |E|)*** если мы используем список соединений и ***O()*** если мы используем матрицу смежности (потому что нам нужно найти каждое ребро, поэтому нужно перебрать всю матрицу).





5. Алгоритм кратчайшего пути Дейкстры.

Алгоритм Дейкстры работает на ориентированных (с некоторыми дополнениями и на неориентированных) графах, и призван искать кратчайшие пути между заданной вершиной и всеми остальными вершинами в графе. Работает только для графов без рёбер отрицательного веса.

**Алгоритм:**

1. Выбрать начальную вершину, присвоить стоимость пути до нее – 0, остальным вершинам ∞;
2. Все вершины являются не выделенными;
3. Объявить первую вершину текущей;
4. Стоимости путей до всех невыделенных вершин находятся след. образом: стоимость пути до невыделенной вершины есть минимальное число из стоимости старого пути до данной вершины, равное сумме стоимости пути до текущей вершины и веса ребра соединяющего текущую и невыделенную вершины.
5. Среди невыделенных вершин выбирается вершина с минимальной стоимостью пути до нее. Если такой вершины нет (стоимость путей до всех вершин равна ∞), то путь не существует и алгоритм завершается, иначе текущей вершиной становится найденная, и она же выделяется.
6. Если все вершины являются выделенными (до всех них найден кратчайший путь), то алгоритм завершается, иначе переход на шаг 4.

Основной цикл выполняется максимум *n* раз, в каждом из них на нахождение минимума тратится порядка *n* операций.

На циклы поиска по соседям тратится количество операций, пропорциональное количеству рёбер *m* (поскольку каждое ребро встречается в этих циклах ровно дважды и требует константное число операций). Таким образом, общее время работы алгоритма:



6. Алгоритм кратчайшего пути Флойда – Уоршелла.

В отличии от алгоритма Дейкстры, который позволяет построить ориентированное дерево кратчайших путей от некоторой вершины, метод Флойда позволяет найти длины всех кратчайших путей в графе.

Обозначим через **di,jm** длину кратчайшего пути из вершины ***i*** в вершину ***j*,** который в качестве промежуточных может содержать только первые ***m*** вершин графа (промежуточной вершиной пути является любая принадлежащая ему вершина, не совпадающая с его начальной или конечной вершинами).

Если между вершинами ***i*** и **j** не существует ни одного пути указанного типа, то условно будем считать, что **di,jm=∞.**

Величина di,j0, представляет длину кратчайшего пути из вершины *i* в вершину *j*, не имеющего промежуточных вершин, т. е. длину кратчайшей дуги, соединяющей *i* с *j* (если такие дуги присутствуют в графе).

Обозначим через Dmматрицу размера NxN, элемент (i,j) которой совпадает с di,jm.

Если в исходном графе нам известна длина каждой дуги, то мы можем сформировать матрицу D0, которая в алгоритме Флойда выступает в качестве исходной.

Вначале из этой матрицы вычисляется матрица D1. Затем по матрице D1 вычисляется матрица D2 и т. д. по формуле:

***di,jm =* min{ *di,mm-1 + dm,jm-1; di,jm-1*}**

Процесс повторяется до тех пор, пока по матрице DN-1 не будет вычислена матрица DN.

Пусть есть три вершины *– i, j,m* – и расстояния между ними: *dij, dim, dmj*.

Если выполняется неравенство ***dim + dmj < dij***, то целесообразно заменить путь ***i j*** путём ***i m + m j***

**Основной этап:**

Задаём строку ***m*** и столбец ***m*** как *ведущую строку* и *ведущий столбец*.

Для всех элементов ***dij***матрицы ***Dm-1*** рассматриваем возможность применения *треугольного оператора*. Если выполняется неравенство:

***dim + dmj < dij*** (i≠j, j≠m, i≠m),

то делаем следующее:

а) в матрице ***Dm*** заменяем элемент ***dij*** матрицы ***Dm-1*** суммой ***dim + dmj***;

b) в матрице ***Sm*** меняем элемент ***sij*** матрицы ***Sm-1*** на ***m***;

c) полагаем ***m=m+1***, повторяем основной этап, пока ***m < N***

После реализации ***N*** этапов алгоритма определение по матрицам ***Dn*** и ***Sn*** кратчайшего пути между узлами ***i*** и ***j*** выполняется по следующим правилам:

1. Кратчайшее расстояние между узлами ***i*** и ***j*** равно элементу ***dij*** в матрице ***Dn***.
2. Промежуточные узлы пути от узла ***i*** к узлу ***j*** определяем по матрице ***Sn***. Пусть ***sij = k***, тогда имеем путь ***i -> j-> k***.

Если далее ***sik*** = ***k*** и ***skj*** = ***j***, то считаем, что весь путь определён. В противном случае повторяем процедуру для путей от узла ***i*** к узлу ***k*** и от узла ***k*** к узлу ***j***.

Сложность – О(n3)

(треш конечно, пока все эти эм и джи выговоришь, легче показывать на пальцах решеткой как это работает)

7. Волновой алгоритм (Алгоритм Ли).

Рассматривается алгоритм построения ортогонального пути.

Алгоритм состоит из двух частей.

1. В первой от источника к приемнику распространяется волна.
2. Во второй выполняется обратный ход, в процессе которого из ячеек волны формируется путь.

Волна, идущая от источника к приемнику, на каждом шаге первой части алгоритма пополняется свободными ячейками, которые, во-первых, еще не принадлежат волне, и, во-вторых, являются 4-соседями ячеек, попавших в волну на предыдущем шаге.

**Инициализация**

Пометить стартовую ячейку *d* := 0

**Распространение волны**

**ЦИКЛ**

**ДЛЯ** каждой ячейки *X*, помеченной числом *d*

пометить все соседние свободные непомеченные ячейки числом *d* + 1

**КЦ**

*d* := *d* + 1

**ПОКА** (финишная ячейка не помечена) **И** (есть возможность распространения волны)

**Восстановление пути**

**ЕСЛИ** финишная ячейка помечена

**ТО**

перейти в финишную ячейку

**ЦИКЛ**

выбрать среди соседних ячейку, помеченную числом на 1 меньше числа в текущей ячейке

перейти в выбранную ячейку и добавить её к пути

**ПОКА** текущая ячейка — не стартовая

**ВОЗВРАТ** путь найден

**ИНАЧЕ**

**ВОЗВРАТ** путь не найден

* Алгоритм кратчайшего пути Форда – Фалкерсона.

**Сеть** – это ориентированный нагруженный граф, в котором нагрузка имеет интерпретацию «пропускная способность» (разумеется, положительная; можно считать, что отсутствующее ребро соответствует нулевой нагрузке). Будем обозначать эту нагрузку c (u, v).

Мы будем считать, что:

a) в сети есть две выделенные вершины – исток (только исходящие дуги) и сток (только входящие дуги);

b) любая вершина лежит на каком-нибудь пути из истока в сток (нет «бесполезных» вершин).

**Поток в сети** – это задание некоторой дополнительной нагрузки на ребра.

Свойства потока:

a) поток по ребру не может превышать пропускной способности ребра и всегда неотрицателен;

b) в любую вершину количество втекающей жидкости равно количеству вытекающей.

**Величина потока** – это сумма исходящего потока из истока. Очевидно, она равна сумме входящего потока в сток. Основная задача – найти максимальный поток в сети с заданной пропускной способностью.

Будем искать *дополняющие пути* из истока в сток, по которому можно пропустить дополнительное количество вещества. Тогда схема алгоритма может быть записана следующим образом:

f = 0;

**while** (*существует дополняющий путь p*) {

*дополнить f вдоль p*;

}

(рассказать как поняла с дискры)

9. Минимальное остовное дерево. Алгоритм Прима.

**Минимальное остовное дерево** — это подмножество рёбер графа, которое соединяет все вершины, имеющие минимальную сумму весов рёбер, и без циклов.

Суть самого алгоритма Прима тоже сводится к жадному перебору рёбер, но уже из определенного множества. На входе так же имеется пустой подграф, который и будем достраивать до потенциального минимального остовного дерева.

* Изначально наш подграф состоит из одной любой вершины исходного графа.
* Затем из рёбер инцидентных этой вершине, выбирается такое минимальное ребро, которое связала бы две абсолютно разные компоненты связности, одной из которых и является наш подграф. То есть, как только у нас появляется возможность добавить новую вершину в наш подграф, мы тут же включаем ее по минимальмально возможному весу.
* Продолжаем выполнять предыдущий шаг до тех пор, пока не найдем искомое MST.

10. Минимальное остовное дерево. Алгоритм Краскала.

**Минимальное остовное дерево** — это подмножество рёбер графа, которое соединяет все вершины, имеющие минимальную сумму весов рёбер, и без циклов.

Механизм, по которому работает данный алгоритм, очень прост. На входе имеется пустой подграф, который и будем достраивать до потенциального минимального остовного дерева. Будем рассматривать только связные графы, в другом случае при применении алгоритма Краскала мы будем получать не минимальное остовное дерево, а просто остовной лес.

* Вначале мы производим сортировку рёбер **по неубыванию** по их весам.
* Добавляем i-ое ребро в наш подграф только в том случае, если данное ребро соединяет две разные компоненты связности, одним из которых является наш подграф. То есть, на каждом шаге добавляется минимальное по весу ребро, один конец которого содержится в нашем подграфе, а другой - еще нет.
* Алгоритм завершит свою работу после того, как множество вершин нашего подграфа совпадет с множеством вершин исходного графа.

Данный алгоритм называется жадным из-за того, что мы на каждом шаге пытаемся найти оптимальный вариант, который приведет к оптимальному решению в целом.

11. Парадигма проектирования жадных алгоритмов. Жадный алгоритм Хаффмана.

**Жадный алгоритм**(greedy algorithm) - метод решения оптимизационных задач, основанный на том, что процесс принятия решения можно разбить на элементарные шаги, на каждом из которых принимается отдельное решение. Решение принимаемое на каждом шаге должно быть оптимальным только на текущем шаге и должно приниматься без учета предыдущих или последующих решений.

**Коды Хаффмана** (Huffman codes) — широко распространенный и очень эффективный метод сжатия данных, который, в зависимости от характеристик этих данных, обычно позволяет сэкономить от 20% до 90% объема. Мы рассматриваем данные, представляющие собой последовательность символов. В жадном алгоритме Хаффмана используется таблица, содержащая частоты появления тех или иных символов. С помощью этой таблицы определяется оптимальное представление каждого символа в виде бинарной строки.

Идея состоит в том, чтобы использовать “кодирование переменной длины”. Мы можем использовать тот факт, что одни символы встречаются в тексте чаще, чем другие для разработки алгоритма, который может представлять тот же фрагмент текста, используя меньшее количество битов. При кодировании с переменной длиной мы присваиваем символам переменное количество битов в зависимости от их частоты в данном тексте. Таким образом, некоторые символы могут в конечном итоге занимать один бит, а некоторые — два бита, некоторые могут быть закодированы с использованием трех битов и так далее.

Техника работает, создавая [бинарное дерево](https://www.techiedelight.com/ru/binary-tree-interview-questions/) узлов. Узел может быть листовым узлом или внутренним узлом. Изначально все узлы являются листовыми узлами, которые содержат сам персонаж, вес (частоту появления) символа. Внутренние узлы содержат вес символов и ссылки на два дочерних узла. По общему соглашению, бит 0 представляет следующий левый дочерний элемент, и 1 представляет следующий правый ребенок.

12. Парадигма проектирования жадных алгоритмов. Задача о составлении расписания. Задача о рюкзаке.

13. Динамическое программирование. Числа Фибоначчи.

Динамическое программирование – это особый подход к решению сложных рекурсивных задач, состоящих из повторяющихся подзадач.

Классическая проблема, которая демонстрирует мощь динамического программирования, – вычисление [чисел Фибоначчи](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8).

Мемоизация

Один из способов избежать постоянного пересчитывания одних и тех же данных – кешировать результаты вычислений. Технически это реализуется так:

* Если кеш содержит результат для полученных входных данных, использовать значение из кеша.
* В противном случае, вычислить результат и сохранить его в кеше, поставив в соответствие с входными данными.

Простая мемоизация результатов, с которой мы только что имели дело – это классический подход **сверху вниз.** Но можно коварно зайти и с другой стороны.

Формализация алгоритма

1. Заведем две локальные переменные, которые будут представлять последние два числа Фибоначчи, с которыми мы работаем.
2. Поместим в них F0(=1) и F1(=1)
3. Изменяя i от 2 до n вычислим Fi. Каждая операция зависит только от Fi-1 и Fi-2, которые хранятся в локальных переменных. Получая результат, мы выбрасываем Fi-2, заменяем его на Fi-1, а в оставшуюся переменную записываем Fi.

Этот подход называется **"снизу-вверх"** (**bottom-up approach**), так как основывается на решении небольших подзадач с целью построения более крупных.

14. Динамическое программирование. Возрастающая подпоследовательность.

Динамическое программирование – это особый подход к решению сложных рекурсивных задач, состоящих из повторяющихся подзадач.

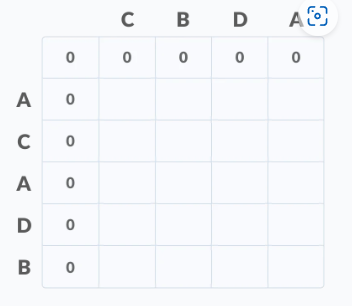
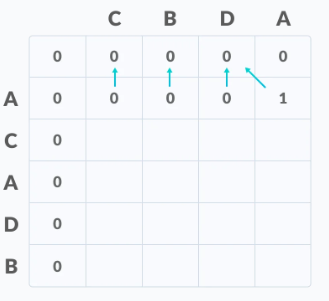
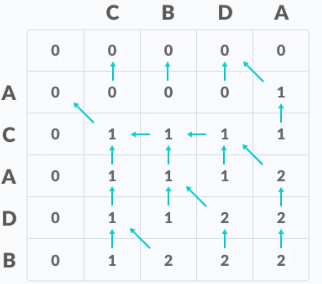
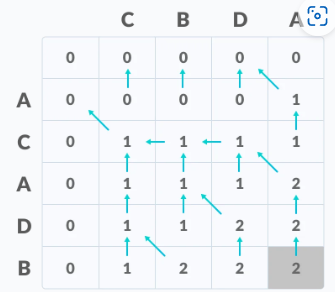
Вроде нормально описано тут [﻿Решаем задачу нахождения длины наибольшей возрастающей подпоследовательности (itnan.ru)](https://itnan.ru/post.php?c=1&p=343210&ysclid=lcjdaup7tz886168437)

15. Динамическое программирование. Путь в лабиринте.

16. Динамическое программирование. Общая подпоследовательность.

Следующие шаги выполняются для поиска самой длинной общей подпоследовательности.

1. Создайте таблицу размеров, где n и m — длины и соответственно. Первая строка и первый столбец заполняются нулями. n+1\*m+1XY
2. Заполните каждую ячейку таблицы, используя следующую логику.
3. Если символ, соответствующий текущей строке и текущему столбцу, совпадает, заполните текущую ячейку, добавив ее к диагональному элементу. Наведите стрелку на диагональную ячейку.
4. В противном случае возьмите максимальное значение из предыдущего столбца и предыдущего элемента строки для заполнения текущей ячейки. Наведите стрелку на ячейку с максимальным значением. Если они равны, укажите на любой из них.
5. **Шаг 2** повторяется до тех пор, пока таблица не будет заполнена.
6. Значение в последней строке и последнем столбце — это длина самой длинной общей подпоследовательности.
7. Чтобы найти самую длинную общую подпоследовательность, начните с последнего элемента и следуйте направлению стрелки. Элементы, соответствующие символу (), образуют самую длинную общую подпоследовательность.

17. Динамическое программирование. Подзадачи и рекуррентные отношения.

18. Динамическое программирование. Задача о лягушке.

19. Динамическое программирование. Расстояние редактирования. Взвешенное расстояние редактирования.

Идея состоит в том, чтобы обрабатывать все символы один за другим, начиная с левой или правой стороны обеих строк.  
Давайте пройдем из правого угла, есть две возможности для каждой пары пересекаемых символов.

1. Если последние символы двух строк одинаковы, делать особо нечего. Игнорируйте последние символы и получайте счетчик оставшихся строк. Таким образом, мы повторяемся для длин m-1 и n-1.
2. Else (если последние символы не совпадают), мы рассматриваем все операции на 'str1', рассматриваем все три операции на последнем символе первой строки, рекурсивно вычисляем минимальную стоимость для всех трех операций и берем минимум три значения.
   1. Вставка: Повтор для m и n-1
   2. Удалить: Повторить для m-1 и n
   3. Заменить: Повтор для m-1 и n-1

20. Динамическое программирование. Задача о рюкзаке. Задача о рюкзаке с повторением и без повторения.

21. Динамическое программирование. Перемножение матриц.

Умножение цепочки матриц (или проблема упорядочения цепочки матриц, MCOP) — это задача оптимизации, которая заключается в том, чтобы найти наиболее эффективный способ умножения заданной последовательности матриц. На самом деле проблема не в том, чтобы выполнить умножение, а просто в том, чтобы определить последовательность задействованных матричных умножений.

Однако порядок, в котором произведение заключено в скобки, влияет на количество простых арифметических операций, необходимых для вычисления произведения.

Пусть матрица A имеет размер *m* × *n*, матрица B имеет размер *n* × *p*. Тогда матрица С будет иметь размер *m* × *p*, а для ее вычисления следует совершить *m* \* *n* \* *p* умножений.

Идея состоит в том, чтобы использовать **запоминание**. Теперь каждый раз, когда мы вычисляем минимальную стоимость, необходимую для умножения определенной подпоследовательности, сохраняйте ее. Если нас когда-нибудь попросят вычислить его снова, просто дайте сохраненный ответ и не пересчитывайте его.

Временная сложность приведенного выше нисходящего решения равна O(n3) и требует O(n2) дополнительное пространство, где n это общее количество матриц.

Следующее [подход «снизу вверх](https://www.techiedelight.com/ru/introduction-dynamic-programming/#bottom-up) вычисляет, для каждого 2 <= k <= n, минимальные затраты всех подпоследовательностей длины k, используя уже вычисленные цены меньших подпоследовательностей. Он имеет такое же асимптотическое время выполнения и не требует рекурсии.

22. Недетерминированные полиномиальные задачи (NP-задачи). Задача коммивояжера. Генетический алгоритм.

Генетический алгоритм — это в первую очередь эволюционный алгоритм, другими словами, основная фишка алгоритма — скрещивание (комбинирование). Алгоритм делится на три этапа:

* Скрещивание
* Селекция (отбор)
* Формирования нового поколения

Если результат нас не устраивает, эти шаги повторяются до тех пор, пока результат нас не начнет удовлетворять или произойдет одно из ниже перечисленных условий:

* Количество поколений (циклов) достигнет заранее выбранного максимума
* Исчерпано время на мутацию

Чаще всего скрещивание производят над двумя лучшими особями. Результатом является также обычно две особи с компонентами, взятыми от их родителей. Цель этого оператора - распространение хороших генов по популяции и стягивание плотности популяции к тем областям, где она и так велика в том предположении, что "нас много там, где хорошо".

* В одноточечном варианте, результатом скрещивания родителей p_i^k=\{a_1,a_2,... a_n\},\qquad p_j^k=\{b_1,b_2,...b_n\}\in P^k в k-ой популяции станут два

элемента популяции k+1, такие что p_i^{k+1}=\{a_1,... ,a_c,b_{c+1},.. ,b_n\},\qquad p_j^{k+1}=\{b_1,... ,b_c,a_{c+1},...a_n\}\in P^{k+1}, где точка c выбирается случайно. В двухточечном варианте, соответственно, точек пересечения будет две, и они также выбираются случайно. Легко расширить эту конструкцию и до n точек. Нужно заметить, что в случае нечётного n, происходит n+1-точечный кроссинговер с n+1-ой точкой между последней и первой компонентами.

Оператор мутаций просто меняет произвольное число элементов в особи на другие произвольные. Фактически он является неким диссипативным элементом, с одной стороны вытягивающим из локальных экстремумов, с другой - приносящим новую информацию в популяцию.

* Инвертирует бит в случае бинарного признака.
* Изменяет на некоторую величину числовой признак. Причём, скорее на ближайший.
* Заменит на другой номинальный признак.

23. Недетерминированные полиномиальные задачи (NP-задачи). Задача коммивояжера. Метод ветвей и границ (алгоритм Литтла).

Имеется матрица расстояний M. Диагональ заполняется бесконечными значениями, т.к. не должно возникать преждевременных циклов. Также имеется переменная, хранящая нижнюю границу.

Стоит оговориться, что нужно вести учет двух видов бесконечностей — одна добавляется после удаления строки и столбца из матрицы, чтобы не возникало преждевременных циклов, другая — при отбрасывании ребер. Случаи будут рассмотрены чуть позже. Первую бесконечность обозначим как inf1, вторую — inf2. Диагональ заполнена inf1.

1. Из каждого элемента каждой строки вычитается минимальный элемент данной строки. При этом минимальный элемент строки прибавляется к нижней границе
2. Из каждого столбца аналогично вычитается минимальный элемент и прибавляется к нижней границе.
3. Для каждого нулевого элемента M(i, j) вычисляется коэффициент, равный сумме минимальных элементов строки i и столбца j, исключая сам элемент (i, j). Этот коэффициент показывает, насколько гарантированно увеличится нижняя граница решения, если исключить из него ребро (i, j)
4. Ищется элемент с максимальным коэффициентом. Если их несколько, можно выбрать любой (все равно оставшиеся будут рассмотрены на следующих шагах рекурсии)
5. Рассматриваются 2 матрицы — M1 и M2. M1 равна M с удаленными строкой i и столбцом j. В ней находится столбец k и строка l, в которых не содержится inf1 и элемент M(k, l) приравнивается inf1. Как было сказано ранее, это необходимо во избежание преждевременных циклов (т.е. на первых этапах (k, l) == (j, i)). Матрица M1 соответствует множеству, содержащему ребро (i, j). Вместе с удалением столбца и строки ребро (i, j) включается в путь.
6. M2 равна матрице M, у которой элемент (i, j) равен inf2. Матрица соответствует множетсву путей, не содержащих ребро (i, j) (важно понимать, что ребро (j, i) при этом не исключается).
7. Переход к п.1 для матриц M1 и M2.

Эвристика состоит в том, что у матрицы M1 нижняя граница не больше, чем у матрицы M2 и в первую очередь рассматривается ветвь, содержащая ребро (i, j).

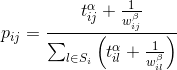
24. Недетерминированные полиномиальные задачи (NP-задачи). Задача коммивояжера. Муравьиный алгоритм.

Все муравьи колонии формируют так называемый *роевой интеллект*. Особи, составляющие колонию, не должны быть умными: они должны лишь взаимодействовать по определенным – крайне простым – правилам, и тогда колония целиком будет эффективна.

Каждый раз проходя от муравейника до пищи и обратно, муравьи оставляют за собой дорожку феромонов. Другие муравьи, почувствовав такие следы на земле, будут инстинктивно устремляться к нему. Поскольку эти муравьи тоже оставляют за собой дорожки феромонов, то чем больше муравьев проходит по определенному пути, тем более привлекательным он становится для их сородичей. При этом, чем короче путь до источника пищи, тем меньше времени требуется муравьям на него – а следовательно, тем быстрее оставленные на нем следы становятся заметными.

Каждый муравей хранит в памяти список пройденных им узлов. Этот список называют ***списком запретов*** *(tabu list)* или просто ***памятью*** муравья. Выбирая узел для следующего шага, муравей «помнит» об уже пройденных узлах и не рассматривает их в качестве возможных для перехода. На каждом шаге список запретов пополняется новым узлом, **а перед новой итерацией алгоритма – то есть перед тем, как муравей вновь проходит путь – он опустошается**.

Кроме списка запретов, при выборе узла для перехода муравей руководствуется **«привлекательностью» ребер**, которые он может пройти. Она зависит, во-первых, от расстояния между узлами (то есть от веса ребра), а во-вторых, от следов феромонов, оставленных на ребре прошедшими по нему ранее муравьями. Естественно, что в отличие от весов ребер, которые являются константными, следы феромонов обновляются на каждой итерации алгоритма: как и в природе, со временем следы испаряются, а проходящие муравьи, напротив, усиливают их.

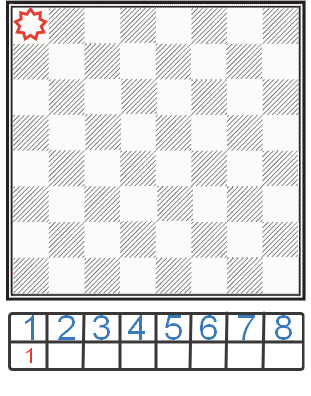
Пусть муравей находится в узле , а узел  – это один из узлов, доступных для перехода. Обозначим вес ребра, соединяющего узлы  и , как , а интенсивность феромона на нем – как . Тогда вероятность перехода муравья из  в  будет равна:

где  и  – это регулируемые параметры, определяющие важность составляющих (веса ребра и уровня феромонов) при выборе пути.

После того, как муравей успешно проходит маршрут, он оставляет на всех пройденных ребрах след, обратно пропорциональный длине пройденного пути:,

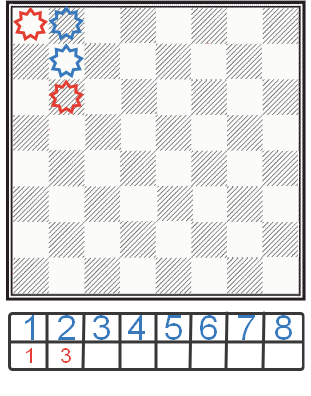
где  – длина пути, а  – регулируемый параметр. Кроме этого, следы феромона испаряются, то есть интенсивность феромона на всех ребрах уменьшается на каждой итерации алгоритма. Таким образом, в конце каждой итерации необходимо обновить значения интенсивностей

25. Алгоритмы с возвратом. Задача о ходе коня. Задача о ферзях.

[](http://1.bp.blogspot.com/-MgPZiEFLQNs/VB8TNemGAII/AAAAAAAAAVQ/AtpsRU6jUEw/s1600/%D0%BF%D1%83%D1%81%D1%82%D0%BE%D0%B5%2B%D0%BF%D0%BE%D0%BB%D0%B5.png)

Поставим первого ферзя в первую клетку первой вертикали. Соответственно в первый элемент нашего массива сохраним единичку. Хорошо, первый ферзь установлен. Самое время попытаться установить второго ферзя.

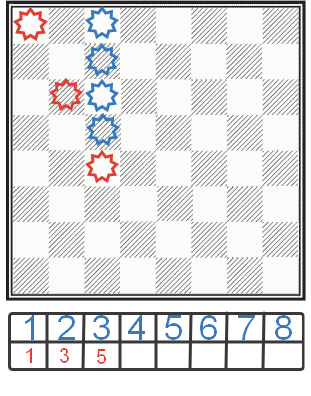
Он должен стоять на второй горизонтали.

[](http://4.bp.blogspot.com/--gJY3A2qOZg/VB8TLvikYMI/AAAAAAAAAVY/1hDE38fmM40/s1600/%D0%BF%D1%83%D1%81%D1%82%D0%BE%D0%B5%2B%D0%BF%D0%BE%D0%BB%D0%B5%2B(2)%2B(2).png)

Поставим его в первую клетку второй вертикали. Вызов функцию проверки, даст отрицательный результат на первом же условии, так как это строка бьется ранее установленным ферзем.

Проверим вторую клетку, второй вертикали. Результат проверки неудовлетворительный, это поле тоже бьется ранее установленным ферзем.

Теперь проверим третью клетку второй вертикали, может быть она нам подойдет? Действительно подходит. Устанавливаем туда ферзя.    Переходим к установке третьего ферзя на третью вертикаль.

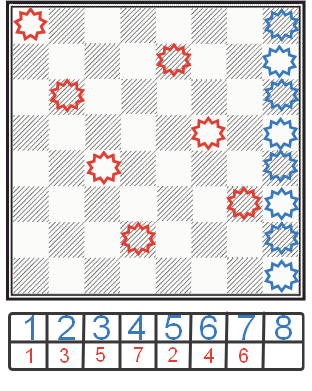
[](http://3.bp.blogspot.com/-uK8AaU3EsHY/VB8TLo1Wd5I/AAAAAAAAAVI/JzZKZEgggok/s1600/%D0%BF%D1%83%D1%81%D1%82%D0%BE%D0%B5%2B%D0%BF%D0%BE%D0%BB%D0%B5%2B(2).png)

Проверку начинаем осуществлять с 1 клетки третей вертикали.

Думаю уже очевидно, что ни первая, ни вторая, ни третья, ни даже четвертая клетки нам не подойдут, так как они бьются ранее установленными ферзями.  А вот пятая клетка, окажется в самый раз, так как её ни первый, ни второй ферзи не бьют. Значит, устанавливаем туда нашего третьего ферзя.

Аналогичным способом заполняем четвертую, пятую, шестую и седьмую вертикали.  Пока что это всё можно реализовать стандартным перебором в два цикла. Внешний цикл двигается по вертикалям от 1 до 8, а внутренний по горизонталям.

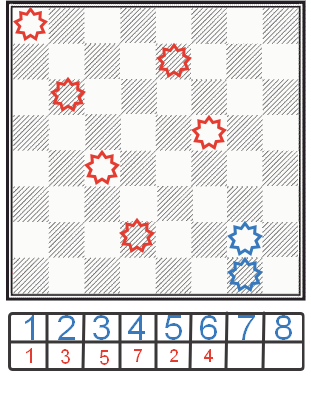
Отдельно рассмотрим установку ферзя на последней вертикали.

[](http://3.bp.blogspot.com/-ObEMxiO_G8A/VB8TMED9Z-I/AAAAAAAAAU4/X9SDw-amxuo/s1600/%D0%BF%D1%83%D1%81%D1%82%D0%BE%D0%B5%2B%D0%BF%D0%BE%D0%BB%D0%B5%2B(3).png)

Если вы всё делали согласно алгоритму, то на этом шаге у вас получится ситуация, изображенная на рисунке справа. Выходит, что текущего ферзя некуда установить, так как все клетки данной вертикали бьются ранее установленными семью ферзями.

Что же делать? Нужно вернуться на шаг назад и попытаться установить седьмого ферзя в другое место.

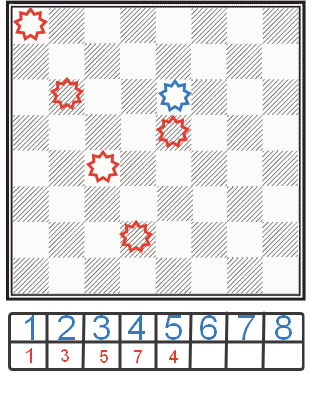
Это операция называется ***возврат***.

[](http://4.bp.blogspot.com/-EQdoHH3kNtA/VB8TMZ5XAGI/AAAAAAAAAVU/XFL5nb6KvRQ/s1600/%D0%BF%D1%83%D1%81%D1%82%D0%BE%D0%B5%2B%D0%BF%D0%BE%D0%BB%D0%B5%2B(4).png)

Нет необходимости, снова проверять для седьмого ферзя клетки с 1 по 6, так как они уже были проверены ранее, и первые шесть ферзей остались на своих местах. Проверив седьмую и восьмую клетки, убеждаемся, что установить в них седьмого ферзя не получится, а поэтому снова делаем возврат, и теперь уже пробуем поставить на другое место шестого ферзя.

 Будем проверять лишь клетки с 5 по 8. Нетрудно убедиться, что ни одна из них не подходит. А значит, выполняем еще один возврат и пытаемся установить пятого ферзя на новое место.

Проверку, как вы уже наверное догадались будем начинать с третьей клетки пятой вертикали. Она нам не подойдет, так как находится под боем, причем от двух ферзей сразу, от второго и третьего. А вот четвертая клетка свободна, и поэтому в неё и будем ставить нашего ферзя.

[](http://2.bp.blogspot.com/-YQIPU7vMTSA/VB8TM_CoOjI/AAAAAAAAAVE/ygQ8InpCjpw/s1600/%D0%BF%D1%83%D1%81%D1%82%D0%BE%D0%B5%2B%D0%BF%D0%BE%D0%BB%D0%B5%2B(5).png)

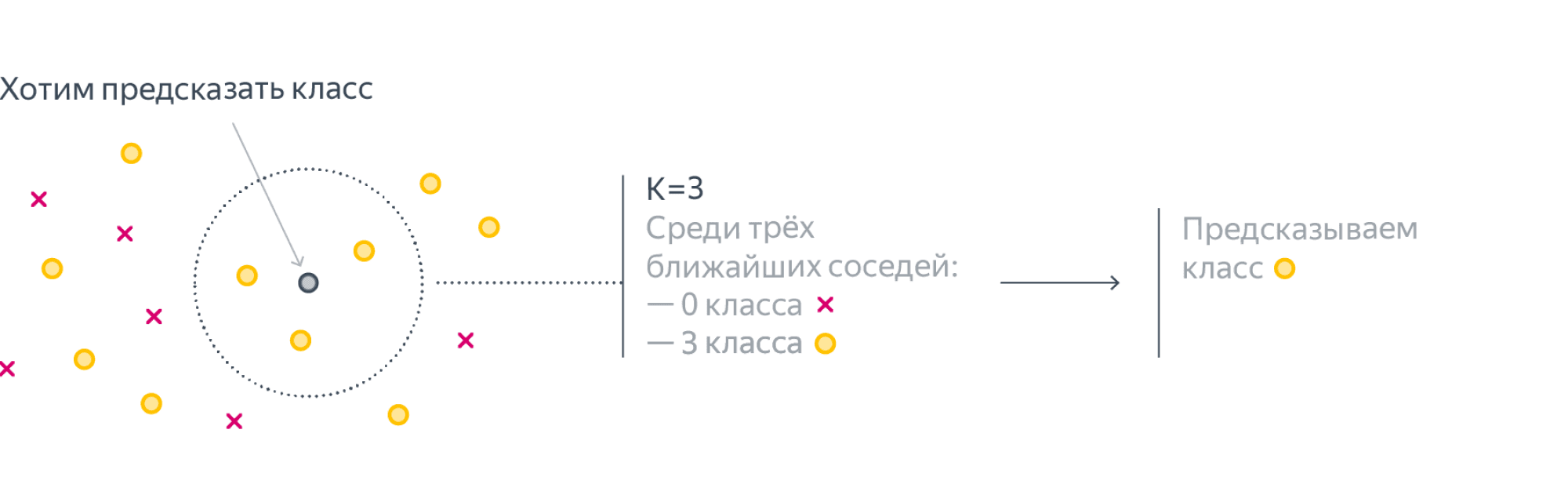
Продолжая в таком духе (чередуя установки с возвратами), рано или поздно мы наткнемся на такую расстановку ферзей, которая удовлетворяет всем нашим требованиям. Другими словами на доске будет размещено 8 ферзей, которые не будут бить друг друга. Наткнувшись на такую расстановку, мы сохраним её в отдельный массив, или же сразу можем вывести на экран. После чего, нам необходимо будет продолжить выполнение перебора.

26. Алгоритм k-ближайших соседей.

Один из самых известных метрических алгоритмов — методу **k-ближайших соседей**, или **k-nearest neighbors (KNN)**

Для метрических методов очень важно уметь эффективно находить ближайшие объекты, поэтому задача их поиска неизбежно возникает при применении любого такого алгоритма.

Представим, что мы проводим классификацию объектов на два класса — красный или жёлтый. Нам дана некоторая обучающая выборка и целевой объект (серый):



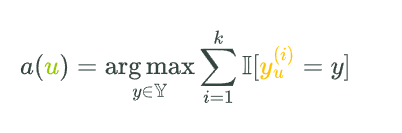
Мы хотим определить, к какому классу относится серый объект. Интуитивно очевидно, что он должен быть жёлтым, потому что все его соседи жёлтые. Эта интуиция и отражает суть метода KNN — классифицировать целевой объект, исходя из того, какие классы у объектов, которые максимально похожи на него.

Перейдём теперь к более формальному описанию алгоритма. Рассмотрим сначала задачу многоклассовой классификации, а регрессией займёмся позже.

Пусть дана обучающая выборка X=(xi,yi)iN =1, где xi∈X, yi∈Y={1,…,C}. Пусть также задана некоторая симметричная по своим аргументам функция расстояния ρ:X×X→[0,+∞). Предположим, что требуется классифицировать новый объект u. Для этого найдём k наиболее близких к u в смысле расстояния ρ объектов обучающей выборки Xk(u)={xu(1),…,xu(k)}:

(1)∀xin∈Xk(u) ∀xout∈X∖Xk(u)ρ(u, xin)⩽ρ(u, xout).

Метку класса объекта xu(i) будем обозначать yu(i). Класс нового объекта тогда естественным образом определим как наиболее часто встречающийся класс среди объектов из Xk(u):



Формула может показаться страшной, но на самом деле все довольно просто: для каждой метки класса y∈Y количество соседей u с такой меткой можно посчитать, просто просуммировав по всем соседям индикаторы событий, соответствующих тому, что метка соседа равна y.

Может возникнуть закономерный вопрос, как же правильно выбрать функцию расстояния ρ. В подавляющем большинстве случаев обычное евклидово расстояние ρ(x,y)=∑i(xi−yi)2 будет хорошим выбором.

27. Асимптотические обозначения. Математическое определение. Обозначение Омега-большое, Тета-большое и о-малое.

**Асимптотические нотации** — это графики функций. Они служат для описания времени работы алгоритма, когда размер входных данных стремится к определенному значению или пределу.

Длительность выполнения алгоритма описывается асимптотическими нотациями.

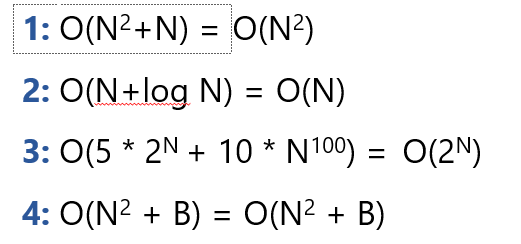
В основном используются три нотации:

* большое «О»,
* омега-нотация,
* тета-нотация.

**Большое «О»**— это верхняя граница скорости выполнения алгоритма. Эта нотация показывает скорость алгоритма в худшем случае.

Так как эта нотация дает нам представления о худшей скорости выполнения алгоритма, то ее анализ обязателен — нам всегда интересна эта характеристика. **O-большое (Big O)** описывает **только** скорость роста. Поэтому мы отбрасываем константы при оценке сложности. Поэтому алгоритм, описываемый как **О(2N)** должен описываться как **O(N)**.

***Неважная сложность. Примеры***



**Омега нотация** — противоположность большому «О». Она показывает нижнюю границу скорости выполнения алгоритма. Она описывает лучший случай выполнения алгоритма.

**Тета-нотация** объединяет в себе сразу две функции — верхнюю и нижнюю. Эта нотация отражает и верхнюю, и нижнюю границу скорости выполнения алгоритма. Именно поэтому она используется для анализа средней скорости выполнения алгоритма.

28. Сортировки данных. Классы алгоритмов сортировки. Оценка алгоритмов сортировки. Обменная сортировка. Пузырьковая сортировка. Быстрая сортировка. Сортировка вставками.

**Сортировка** *-* это упорядочивание набора однотипных данных по воз­растанию или убыванию.

Категории алгоритмов сортировки:

* алгоритмы, сортирующие объекты с ***произвольным*** доступом (например, массивы или дис­ковые файлы произвольного доступа)
* алгоритмы, сортирующие ***последова­тельные*** объекты (например, файлы на дисках и лентах или связанные списки )

Существует 3 метода сортировки массива: **обмен, выбор, вставка**

Обменная сортировка – сортировка, которая некоторым систематическим образом меняет местами пары элементов, не отвечающие порядку, до тех пор, пока такие пары существуют. Наиболее известными обменными сортировками являются сортировка пузырьком, шейкерная сортировка, быстрая сортировка.

**Сортировка пузырьком**

Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован. Заметим, что после первой итерации самый большой элемент будет находиться в конце массива, на правильном месте. После двух итераций на правильном месте будут стоять два наибольших элемента, и так далее. Очевидно, не более чем после n итераций массив будет отсортирован. Таким образом, асимптотика в худшем и среднем случае – O(n2), в лучшем случае – O(n).

**Шейкерная сортировка**

Разновидность пузырька. Заметим, что сортировка пузырьком работает медленно на тестах, в которых маленькие элементы стоят в конце (их еще называют «черепахами»). Такой элемент на каждом шаге алгоритма будет сдвигаться всего на одну позицию влево. Поэтому будем идти не только слева направо, но и справа налево. Будем поддерживать два указателя begin и end, обозначающих, какой отрезок массива еще не отсортирован. На очередной итерации при достижении end вычитаем из него единицу и движемся справа налево, аналогично, при достижении begin прибавляем единицу и двигаемся слева направо. Асимптотика у алгоритма такая же, как и у сортировки пузырьком, однако реальное время работы лучше.

**Быстрая сортировка**

Самый продвинутый обменный алгоритм.

1. Делим массив пополам. Средний элемент — опорный.
2. Движемся от левого края массива вправо, до тех пор пока не обнаружим элемент, который больше опорного.
3. Движемся от правого края массива влево, до тех пор пока не обнаружим элемент, который меньше опорного.
4. Меняем местами два элемента, найденные в пунктах 2 и 3 .
5. Продолжаем выполнять пункты 2-3-4 пока в результате обоюдного движения не произойдёт встреча.
6. В точке встречи массив делится на две части. К каждой части рекурсивно применяем алгоритм быстрой сортировки.

В итоге получим отсортированный массив, так как каждый элемент меньше опорного стоял раньше каждого большего опорного. Асимптотика: O(nlogn) в среднем и лучшем случае, O(n2). Наихудшая оценка достигается при неудачном выборе опорного элемента.

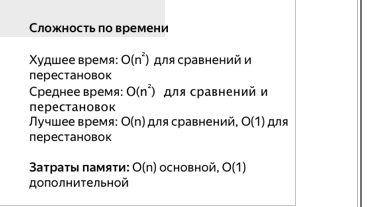
Общая суть сортировок вставками такова:

1. Перебираются элементы в неотсортированной части массива.
2. Каждый элемент вставляется в отсортированную часть массива на то место, где он должен находиться.

То есть, сортировки вставками всегда делят массив на 2 части — отсортированную и неотсортированную. Из неотсортированной части извлекается любой элемент. Поскольку другая часть массива отсортирована, то в ней достаточно быстро можно найти своё место для этого извлечённого элемента. Элемент вставляется куда нужно, в результате чего отсортированная часть массива увеличивается, а неотсортированная уменьшается. По такому принципу работают все сортировки вставками.

**Сортировка простыми вставками**

Проходим по массиву слева направо и обрабатываем по очереди каждый элемент. Слева от очередного элемента наращиваем отсортированную часть массива, справа по мере процесса потихоньку испаряется неотсортированная. В отсортированной части массива ищется точка вставки для очередного элемента. Сам элемент отправляется в буфер, в результате чего в массиве появляется свободная ячейка — это позволяет сдвинуть элементы и освободить точку вставки.

****

29. Динамические структуры данных. Вектор. Связанный список. Двухсвязный список. Стек. Очередь. Словарь. Множество.

**Структура данных** – программная единица, позволяющая хранить и обрабатывать однотипные и/или логически связанные данные.

**Динамические структуры данных** – структуры данных, память под которые выделяется и высвобождается по мере необходимости.

**Вектор** (он же одномерный массив) — упорядоченный набор элементов с произвольным доступом по числовому индексу. однако они обладают одним весомым преимуществом перед массивом - их размер не фиксирован.

**Список** — это динамическая структура данных, у каждого элемента может быть только один предок и только один потомок. По сути своей это очень похоже на обыкновенный массив, с той лишь разницей, что размер его не имеет ограничений. [Списки](https://zen.yandex.ru/media/codeblog/sviaznyi-spisok-linked-list-na-iazyke-c-5abb5b72e44a9479f81e72a8)также подразделяются на несколько типов.

* Односвязный список — элемент имеет указатель только на своего потомка.
* Двусвязный список — элемент имеет указатели и на потомка, и на родителя.
* Замкнутый (кольцевой, циклический) список — головной и хвостовой элементы которого указывают друг на друга.

На базе простого однонаправленного списка могут быть построены такие структуры данных, как очередь (queue) и стек (stack).

**Очередь** есть ничто иное, как список, операции чтения и добавления элементов в котором подвержены определенным правилам. При этом, при чтении элемента, он удаляется из очереди. Все операции проводятся по принципу «Первый пришел, первый вышел» (FIFO — first in, first out). Таким образом, для чтения в очереди доступна только голова, в то время как добавление проводится только в хвост.

**Стек**во многом похож на очередь, с той лишь разницей, что извлечение и добавление элементов в нем происходит по правилу «Последний пришел, первый вышел» (LIFO — last in, first out). Добавление и извлечение элементов проводится от головы.

**Словарь** (он же ассоциативный массив) — это тот-же вектор, но с небольшими отличиями. В качестве индекса (который в словаре будет называться ключ) могут выступать не только числа, но и любые другие типы данных (даже другие коллекции!). Также допустимы пропуски, если мы все-таки будем использовать в качестве ключа целое число, например у нас может быть элемент связанный с ключом 5, но при этом отсутствовать элемент связанный с ключом 4

**Множество** — неупорядоченный набор элементов, без повторов. В множестве можно быстро проверить, есть какой-либо элемент внутри, или его нет. Если мы хотим узнать, находится ли конкретный элемент в моем списке элементов, то и для списка и для вектора мне пришлось бы перебрать (в худшем случае) все элементы.

30. Поиск. Последовательный поиск. Бинарный поиск. Интерполяционный поиск. Поиск по деревьям (дописать про дерево поиска).

Под поиском в массиве будем понимать задачу нахождения индекса, по которому в массиве располагается некоторый заданный элемент.

**Последовательный метод** поиска называют также методом *последо­вательного перебора.*Это самый универсальный, наиболее простой и самый длительный метод поиска. Последовательный поиск можно использовать при любом способе организации информационного масси­ва, для любых структур данных, при любой форме аргумента поиска. В процессе поиска осуществляется последовательное обращение к каж­дой записи массива и при этом определяется, удовлетворяет ли данная запись критерию выдачи.

Последовательный поиск в массиве из *N*записей требует в среднем *(N*+ 1)/2 сравнений (единица в числителе появляется при нечетном N). В худшем случае, когда искомая запись окажется последней в массиве или ее не будет там вовсе, потребуется N сравнений.

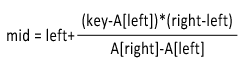
Если данные, в которых производится поиск, отсортированы, для нахождения элемента можно применять метод, намного превосходящий предыдущий — **бинарный поиск.** В нем применяется метод половинного деления. Сначала проверим средний элемент. Если он больше, чем искомый ключ, проверим средний элемент первой половины, в противном случае — средний элемент второй половины. Будем повторять эту процедуру до тех пор, пока искомый элемент не будет найден либо пока не останется очередного элемента.

В двоичном поиске количество сравнений в худшем случае равно log2*n.*

В основе **интерполяционного поиска** лежит операция интерполирование. Интерполирование – нахождение промежуточных значений величины по имеющемуся дискретному набору известных значений. Интерполяционный поиск работает только с упорядоченными массивами; он похож на бинарный, в том смысле, что на каждом шаге вычисляется некоторая область поиска, которая, по мере выполнения алгоритма, сужается.

Но в отличие от двоичного, интерполяционный поиск не делит последовательность на две равные части, а вычисляет приблизительное расположение ключа (искомого элемента), ориентируясь на расстояние между искомым и текущим значением элемента.

**Формула, определяющая алгоритм интерполяционного поиска выглядит следующим образом:**



Здесь mid – номер элемента, с которым сравнивается значение ключа, key – ключ (искомый элемент), A – массив упорядоченных элементов, left и right – номера крайних элементов области поиска. Важно отметить, операция деления в формуле строго целочисленная, т. е. дробная часть, какая бы она ни была, отбрасывается.

Интерполяционный поиск в эффективности, как правило, превосходит двоичный, время его работы составляет O(log(log(N))).

**Бинарное дерево** — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

**Бинарное дерево поиска** — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

31. Рекурсия. Виды рекурсивных функций. Перебор с помощью рекурсии. Решение задач с помощью.

**Рекурсия** – это функция, которая вызывает сама себя.

**Рекурсивное определение состоит из двух частей:**

1. Базовый случай т.е. не зацикленная часть

Обычное определение, четко определяющее объект

2.Рекурсия т.е. зацикленная часть

Позволяет использовать объекты, так, как новые получившиеся объекты из старого тоже подпадают под определение данного объекта.

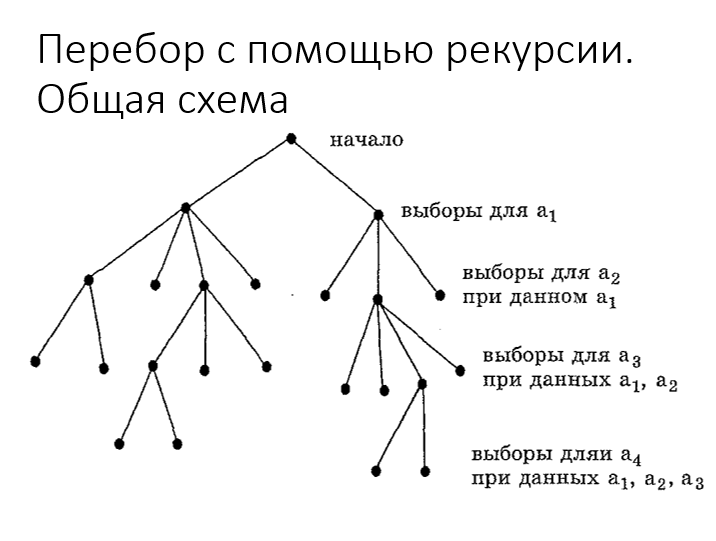
Рекурсивные функции бывают двух видов:

* Прямая (когда функция вызывает саму себя)
* Косвенная (например, когда функция А вызывает функцию Б, а функция Б вызывает саму функцию А. Вызов через посредника)

Простейшая рекурсивная функция – это конструкция вида if – else, где:

* **if** содержит базовый случай
* **else** содержит рекурсивный случай

Распространенными задачами, которые решаются с помощью рекурсии являются задачи на нахождение числа Фибоначчи и задача о Ханойских башнях. Также можно использовать для решения различных алгебраических задач: нахождение суммы чисел от 1 до n, нахождение суммы цифр числа,



Прием, который позволяет уменьшить количество рассматриваемых вариантов в переборе, называется **отсечением**. Существует огромное количество разных по сложности и эффективности отсечений и эвристик.

Вот самые популярные из них:

* Отсечение по ответу (откидывание заведомо ненужных вариантов)
* Отсечение по времени.
* Оптимальный порядок перебора.
* Жадных поиск каких-то решений еще до запуска перебора.

Все что можно закодить рекурсией, можно в теории закодить итеративно (и наоборот).Если вы можете без особых проблем написать итеративное решение задачи, то, скорее всего, оно будет работать лучше рекурсивного.