1. **ИП-3. Колоквиум - 1**
2. **Алгоритмы, виды сложности, виды асимптотических оценок,**

**Алгоритм** – точное предписание, которое задает вычислительный процесс, начинающийся с произвольного **исходного данного** и направленный на получение полностью определенного этим исходным данным **результата**.

**Алгоритм** – это четкое описание по выполнению некоторого **процесса обработки данных**, который через разумное конечное число шагов приводит к решению задачи данного типа для любых допустимых вариантов исходных данных.

**Свойства алгоритмов:**

1. **Дискретность (прерывность)** - алгоритм должен представлять процесс решения задачи как последовательное выполнение простых шагов.

2. **Определенность** - каждое правило алгоритма должно быть четким, однозначным и не оставлять места для вариаций.

3. **Результативность (конечность)** - алгоритм должен приводить к решению задачи за конечное число шагов.

4. **Детерминированность**. После каждого шага необходимо указывать, какой шаг выполняется следующим, либо давать команду остановки

5. **Массовость** - алгоритм решения задачи разрабатывается в общем виде и должен быть применим для некоторого класса задач, различающихся только входными данными. При этом входные данные могут выбираться из некоторой области, которая называется **областью применимости алгоритма**.

**Формы представления алгоритмов:**

1. **Словесная** (записи на естественном языке);

2. **Графическая** (изображения из графических символов);

3. **Псевдокоды** (описания алгоритмов на условном алгоритмическом языке, включающие в себя элементы языка программирования, фразы естественного языка, общепринятые математические обозначения и др.);

4. **Программная** (тексты на языках программирования)

**Сложность алгоритмов**

**Анализ трудоёмкости алгоритмов**

Целью анализа трудоёмкости алгоритмов является нахождение оптимального алгоритма для решения данной задачи.



**Теория сложности**, являясь частью **теории вычислений**, изучает ресурсы или стоимость вычислений, необходимые для выполнения поставленной проблемы.

**Вычислительная сложность алгоритма** — это функция, определяющая зависимость объёма работы, выполняемой некоторым алгоритмом, от свойств входных данных. Объём работы обычно измеряется абстрактными понятиями времени и пространства, называемыми вычислительными ресурсами.

**Время** определяется количеством элементарных шагов, необходимых для решения проблемы, тогда как **пространство** определяется объёмом памяти или места на носителе данных.

**Градации сложности**

𝑓 𝑛 = 𝑂(1) – константная;

𝑓 𝑛 = 𝑂(𝑙𝑜𝑔 𝑛) – логарифмическая; (см. пример ^ алгоритм 2)

𝑓 𝑛 = 𝑂 𝑛 – линейная; (см. пример \* алгоритм 2)

𝑓 𝑛 = 𝑂(𝑛 𝑐) - полиномиальная (квадратичная, кубическая …); (см. пример\* алгоритм 1)

𝑓 𝑛 = 𝑂(𝑐 𝑛) – экспоненциальная; (см. пример ~ алгоритм 1)

𝑓 𝑛 = 𝑂(𝑛!) – факториальная

Запись вида f(n) = O(g(n)) означает, что ф-ия f(n) возрастает медленнее чем ф-ия g(n), т.е если при 𝑔 𝑛 > 0 и 𝑛 > 0 существуют положительные с1и 𝑛0, такие что при любом 𝑛 > 𝑛0 выполняется 𝑐1 ∙ 𝑔 𝑛 ≥ 𝑓(𝑛). с1и 𝑛0 могут быть сколь угодно большими числами.

Такая **O оценка** дает нам верхнюю оценку временной трудоемкости алгоритма – его асимптотическую сложность.

1. **Правила оценки асимптотической сложности**

**Оценка сложности**

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.

**Подсчет операций. Классы входных данных**

Одним из способов оценки трудоемкости (*Tn*) является подсчет количества выполняемых операций. Рассмотрим в качестве примера алгоритм поиска минимального элемента массива.

**начало; поиск минимального элемента массива array из N элементов**

**min := array[1]**

**для i от 2 до N выполнять:**

**если array[i] < min**

**min := array[i]**

**конец; вернуть min**

При выполнении этого алгоритма будет выполнена:

1. N — 1 операция присваивания счетчику цикла i нового значения;
2. N — 1 операция сравнения счетчика со значением N;
3. N — 1 операция сравнения элемента массива со значением min;
4. от 1 до N операций присваивания значения переменной min.

Точное количество операций будет зависеть от обрабатываемых данных, поэтому имеет смысл говорить о наилучшем, наихудшем и среднем случаях. При этом худшему случаю всегда уделяется особое внимание, в том числе потому, что «плохие» данные могут быть намеренно поданы на вход злоумышленником.

Понятие среднего случая используется для оценки поведения алгоритма с расчетом на то, что наборы данных равновероятны. Однако, такая оценка достаточно сложна:

1. исходные данные разбиваются на группы так, что трудоемкость алгоритма (*ti*) для любого набора данных одной группы одинакова;
2. исходя из доли наборов данных группы в общем числе наборов, рассчитывается вероятность для каждой группы (*pi*);
3. оценка среднего случая вычисляется по формуле:

**O(n) — линейная сложность**

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

**O(log n) — логарифмическая сложность**

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

**O(n2) — квадратичная сложность**

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n \* n, т. е. n2.

1. Рекуррентные уравнения, способы реализации на примере расчета чисел Фибоначчи
2. Исследование рекуррентных соотношений на основе дерева рекурсии
3. Оценка решения рекуррентного соотношения методом подстановки
4. **Парадигма «Разделяй и властвуй». Основная теорема о рекуррентных соотношениях.**

Основная теорема о рекуррентных соотношениях — это формула, предназначенная для решения рекуррентных соотношений следующего вида:

T(n) = aT(n/b) + f(n), где

n = объем входных данных

a = количество подзадач в рекурсии

n/b = размер каждой подзадачи. Предполагается, что все подзадачи имеют одинаковый размер.

f(n) = оценка выполненной работы вне рекурсивных вызовов.

Также она включает в себя вычислительную стоимость деления на подзадачи объединения решений этих подзадач.

Здесь a ≥ 1 и b > 1 — константы, а f(n) — асимптотически положительная функция.

Асимптотически положительная функция — функция, где при достаточно больших значениях n f(n)>0.

Основная теорема о рекуррентных соотношениях — простой и быстрый способ вычисления временной сложности рекуррентных соотношений (например, «Разделяй и влавствуй»).

1. Постановка и решение задачи Рюкзака 0-1 методом ДП
2. Длинная арифметика. Алгоритм Карацубы.
3. Классический алгоритм Хаффмана на примере 4-значных чисел.
4. Простейшие структуры данных: очереди, стеки, деки; операции, структуры хранения.
5. Интерполяционный поиск в массиве чисел.
6. Задача кодирования сообщений, равномерные и неравномерные коды. Сжатие сообщений
7. DSA алгоритм, расчет открытого и закрытого ключа
8. АТД Бинарная куча. Назначение, операции, применение, алгоритм реализации
9. АТД DSU (СНМ), Назначение, операции, применение, алгоритм реализации
10. Деревья, термины и определения, основные операции
11. **N-арные деревья, представление, виды обходов**

В теории графов, дерево – это связный (ориентированный или неориентированный) граф, не содержащий циклов. Т.е. для любой вершины есть один и только один способ добраться до любой другой вершины. В программировании наиболее часто используются бинарные деревья, в которых число исходящих рёбер не превосходит 2, и N-арные деревья с произвольным количеством исходящих ребер.

Бинарное дерево — это конечное множество элементов, которое либо пусто, либо содержит элемент (корень), связанный с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями. Каждый элемент бинарного дерева называется узлом. Связи между узлами дерева называются его ветвями. N-арное дерево – это дерево, в котором степени вершин не превосходят N + 1.

В памяти компьютера деревья обычно представляют в виде связной структуры, где каждый узел помимо ключа (key) хранит указатели на дочерние узлы и иногда на родительский. Для хранения N-арных деревьев используют структуру с левым дочерним и правыми сестринским узлами (left-child, right-sibling representation). В этом случае вместо указателя на дочерние узлы каждый узел x хранит два указателя:

•в lef t\_child[x]– указатель на крайний левый дочерний узел узла x;

•в right\_sibling[x]– указатель на узел, расположенный на одном уровне с x справа от него.

Деревья можно обходить «в глубину» или «в ширину». Существует три основных способа обхода «в глубину»:

• прямой (pre-order)

• центрированный (in-order)

inorder(node)

if (node = null)

return

inorder(node.left)

visit(node)

inorder(node.right) iterativeInorder(node)

s ← empty stack

while (not s.isEmpty() or node ≠ null)

if (node ≠ null)

s.push(node)

node ← node.left

else

node ← s.pop()

visit(node)

node ← node.right

• обратный (post-order)

Деревья можно обходить также в порядке уровней, где мы посещаем каждый узел на уровне, прежде чем перейти на следующий уровень. Такой поиск называется поиском в ширину (breadth-first search, BFS).

Обход n-арного дерева. Алгоритмы обхода n-арного дерева.

1)Прямой обход - Посещается корень дерева, затем все узлы левого поддерева, затем узлы след поддеревьев до правого (корень-лево-право).

2)Обратный обход - Сначала все узлы левого поддерева, затем последовательно узлы остальных поддеревьев в обратном порядке, последним посещается корень (лево-прово-корень).

3)Симметричный обход – в симметричном порядке посещаются все узлы левого поддерева, потом корень, затем последовательно в симметричном порядке все узлы остальных поддеревьев (лево-корень-право).

4)Рекурсивный обход дерева(1)

**Прямой**:Preoder(n: узел);

1.Список обхода <- n(Обращение к узлу);

2.For каждого сына m-узла n в порядке слева направо doPreoder(m); (Рекурсивный обход левого поддерева. рекурсивный обход правого поддерева.)

**Обратный**: меняем местами пункты 1 и 2:

1. Рекурсивный обратный обход левого поддерева.

2.Рекурсивный обратный обход правого поддерева.

3. Обращение к узлу.

1. Бинарные деревья**, виды представлений в памяти**, виды обходов, операции добавления и удаления, поиск значения, минимума, максимума, расчет сумм, количества
2. Упорядоченное бинарное дерево, поиск значения, минимума, максимума, расчет сумм, количества
3. Постановка задачи о кратчайших путях, примеры применения
4. Алгоритм Дейкстры на основе сортировки, сложность, ограничения
5. Алгоритм Дейкстры на основе приоритетной очереди, сложность, ограничения
6. Постановка задачи о минимальном остовном дереве (МОД), примеры применения.
7. Алгоритм Прима на основе сортировки, сложность, ограничения
8. Алгоритм Прима на основе приоритетной очереди, сложность, ограничения
9. Алгоритм Крускала, сложность, ограничения
10. **Сортировка, назначение, основные виды и их особенности**

**Алгоритм сортировки** — это алгоритм для упорядочивания элементов в массиве. В случае, когда элемент в массиве имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

**Сортировка пузырьком / Bubble sort**

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 / Shaker sort**

(также известна как сортировка перемешиванием и коктейльная сортировка). 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.

**Сортировка вставками / Insertion sort**

Создадим массив, в котором после завершения алгоритма будет лежать ответ. Будем поочередно вставлять элементы из исходного массива так, чтобы элементы в массиве-ответе всегда были отсортированы. Асимптотика в среднем и худшем случае – O(n2), в лучшем – O(n). Реализовывать алгоритм удобнее по-другому (создавать новый массив и реально что-то вставлять в него относительно сложно): просто сделаем так, чтобы отсортирован был некоторый префикс исходного массива, вместо вставки будем менять текущий элемент с предыдущим, пока они стоят в неправильном порядке.

**Сортировка Шелла / Shellsort**

Используем ту же идею, что и сортировка с расческой, и применим к сортировке вставками. Зафиксируем некоторое расстояние. Тогда элементы массива разобьются на классы – в один класс попадают элементы, расстояние между которыми кратно зафиксированному расстоянию. Отсортируем сортировкой вставками каждый класс. В отличие от сортировки расческой, неизвестен оптимальный набор расстояний. Существует довольно много последовательностей с разными оценками. Последовательность Шелла – первый элемент равен длине массива, каждый следующий вдвое меньше предыдущего. Асимптотика в худшем случае – O(n2). Последовательность Хиббарда – 2n — 1, асимптотика в худшем случае – O(n1,5), последовательность Седжвика (формула нетривиальна, можете ее посмотреть по ссылке ниже) — O(n4/3), Пратта (все произведения степеней двойки и тройки) — O(nlog2n). Отмечу, что все эти последовательности нужно рассчитать только до размера массива и запускать от большего от меньшему (иначе получится просто сортировка вставками). Также я провел дополнительное исследование и протестировал разные последовательности вида si = a \* si — 1 + k \* si — 1 (отчасти это было навеяно эмпирической последовательностью Циура – одной из лучших последовательностей расстояний для небольшого количества элементов). Наилучшими оказались последовательности с коэффициентами a = 3, k = 1/3; a = 4, k = 1/4 и a = 4, k = -1/5.

**Сортировка выбором / Selection sort**

На очередной итерации будем находить минимум в массиве после текущего элемента и менять его с ним, если надо. Таким образом, после i-ой итерации первые i элементов будут стоять на своих местах. Асимптотика: O(n2) в лучшем, среднем и худшем случае. Нужно отметить, что эту сортировку можно реализовать двумя способами – сохраняя минимум и его индекс или просто переставляя текущий элемент с рассматриваемым, если они стоят в неправильном порядке. Первый способ оказался немного быстрее, поэтому он и реализован.

**Пирамидальная сортировка / Heapsort**

Развитие идеи сортировки выбором. Воспользуемся структурой данных «куча» (или «пирамида», откуда и название алгоритма). Она позволяет получать минимум за O(1), добавляя элементы и извлекая минимум за O(logn). Таким образом, асимптотика O(nlogn) в худшем, среднем и лучшем случае. Реализовывал кучу я сам, хотя в С++ и есть контейнер priority\_queue, поскольку этот контейнер довольно медленный.

**Быстрая сортировка / Quicksort**

Выберем некоторый опорный элемент. После этого перекинем все элементы, меньшие его, налево, а большие – направо. Рекурсивно вызовемся от каждой из частей. 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. Моя реализация этого алгоритма совершенно стандартна, идем одновременно слева и справа, находим пару элементов, таких, что левый элемент больше опорного, а правый меньше, и меняем их местами. Помимо чистой быстрой сортировки, участвовала в сравнении и сортировка, переходящая при малом количестве элементов на сортировку вставками. Константа подобрана тестированием, а сортировка вставками — наилучшая сортировка, подходящая для этой задачи (хотя не стоит из-за этого думать, что она самая быстрая из квадратичных).

1. **Алгоритм Шелла, оценки сложности, устойчивость**

**Сортировка Шелла** — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами.

При сортировке Шелла сначала сравниваются и сортируются между собой значения, отстоящие один от другого на некотором расстоянии d. После этого процедура повторяется для некоторых меньших значений d, а завершается сортировка Шелла упорядочиванием элементов при d = 1 (то есть, обычной сортировкой вставками). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).



**Устойчивость** – это один из параметров трудоемкости алгоритма, который характеризует то, что сортировка не меняет взаимного расположения равных элементов. Сортировка Шелла теряет устойчивость.

Реализация на Java:

**Сложность алгоритма: O(n log2 n).**

for (int step = n / 2; step > 0; step /= 2) {

for (int i = step; i < n; i++) {

for (int j = i - step; j >= 0 && a[j] > a[j + step] ; j -= step) {

int x = a[j];

a[j] = a[j + step];

a[j + step] = x;

}

}

}

1. Шейкер-сортировка, оценки сложности, устойчивость
2. Сортировка Хоара, оценки сложности, устойчивость
3. Пирамидальная сортировка, оценки сложности, устойчивость
4. Сортировка подсчетом, оценки сложности, устойчивость