**1. Предмет теории алгоритмов.**

**Теория алгоритмов** — наука, находящаяся на стыке математики и информатики, изучающая общие свойства и закономерности алгоритмов и разнообразные формальные модели их представления. К задачам теории алгоритмов относятся формальное доказательство алгоритмической неразрешимости задач, асимптотический анализ сложности алгоритмов, классификация алгоритмов в соответствии с классами сложности, разработка критериев сравнительной оценки качества алгоритмов и т. п. Вместе с математической логикой теория алгоритмов образует теоретическую основу вычислительных наук.

**2. Этапы развития ТА. Цель и задачи теории алгоритмов.**

Первым дошедшим до нас алгоритмом в его интуитивном понимании считается предложенный Евклидом в III веке до нашей эры алгоритм нахождения наибольшего общего делителя двух чисел (алгоритм Евклида).

Начальной точкой отсчета современной теории алгоритмов можно считать работу немецкого математика Курта Гёделя (1931 год - теорема о неполноте символических логик).

Первые фундаментальные работы по теории алгоритмов были опубликованы независимо в 1936 году годы Аланом Тьюрингом, Алоизом Черчем и Эмилем Постом. Предложенные ими машина Тьюринга, машина Поста и лямбда-исчисление Черча были эквивалентными формализмами алгоритма. В 1950-е годы существенный вклад в теорию алгоритмов внесли работы Колмогорова и Маркова.

К 1960-70-ым годам оформились следующие направления в теории алгоритмов:

1. Классическая теория алгоритмов (формулировка задач в терминах формальных языков, понятие задачи разрешения, введение сложностных классов, исследования классов NP-полных задач).

2. Теория асимптотического анализа алгоритмов (понятие сложности и трудоёмкости алгоритма, критерии оценки алгоритмов, методы получения асимптотических оценок, в частности для рекурсивных алгоритмов, асимптотический анализ трудоемкости или времени выполнения).

3. Теория практического анализа вычислительных алгоритмов (получение явных функции трудоёмкости, интервальный анализ функций, практические критерии качества алгоритмов, методика выбора рациональных алгоритмов).

**Цели и задачи:**

1. Формализация понятия «алгоритм» и исследование формальных алгоритмических систем

2. Формальное доказательство алгоритмической неразрешимости ряда задач

3. Классификация задач, определение и исследование сложностных классов

4. Асимптотический анализ сложности алгоритмов

5. Исследование и анализ рекурсивных алгоритмов

6. Получение явных функций трудоемкости в целях сравнительного анализа алгоритмов

7. Разработка критериев сравнительной оценки качества алгоритмов

**3. Практическое применение результатов теории алгоритмов.**

Полученные в теории алгоритмов теоретические результаты находят достаточно широкое практическое применение, при этом можно выделить следующие два аспекта:

**Теоретический аспект:** при исследовании некоторой задачи результаты теории алгоритмов позволяют ответить на вопрос – является ли эта задача в принципе алгоритмически разрешимой – для алгоритмически неразрешимых задач возможно их сведение к задаче останова машины Тьюринга. В случае алгоритмической разрешимости задачи – следующий важный теоретический вопрос – это вопрос о принадлежности этой задачи к классу NP–полных задач, при утвердительном ответе на который, можно говорить о существенных временных затратах для получения точного решения для больших размерностей исходных данных.

**Практический аспект:** методы и методики теории алгоритмов (в основном разделов асимптотического и практического анализа) позволяют осуществить:

1) Рациональный выбор из известного множества алгоритмов решения данной задачи с учетом особенностей их применения (например, при ограничениях на размерность исходных данных или объема дополнительной памяти);

2) Получение временных оценок решения сложных задач;

3) Получение достоверных оценок невозможности решения некоторой задачи за определенное время, что важно для криптографических методов;

4) Разработку и совершенствование эффективных алгоритмов решения задач в области обработки информации на основе практического анализа.

**4. Основные определения.**

**А. Н. Колмогоров: Алгоритм** – это всякая система вычислений, выполняемых по строго определенном правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи.

**А. А. Марков: Алгоритм** – это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату.

Пусть D – область (множество) исходных данных задачи, а R – множество возможных результатов, тогда мы можем говорить, что алгоритм осуществляет отображение D --> R. Поскольку такое отображение может быть не полным, то вводятся следующие понятия:

Алгоритм называется **частичным** алгоритмом, если мы получаем результат только для некоторых d є D и **полным** алгоритмом, если алгоритм получает правильный результат для всех d є D.

**Алгоритм должен:**

1. Содержать конечное количество элементарно выполнимых предписаний

2. Выполнять конечное количество шагов при решении задач

3. Быть единым для всех допустимых исходных данных

4. Приводить к правильному решению

**5. Понятие об алфавитном операторе.**

**Алфавитным оператором** или **алфавитным отображением** называют всякое соответствие, сопоставляющее словам некоторого алфавита слова в том же или в другом фиксированном алфавите.

Алфавитные операторы, задаваемые с помощью конечной системы правил, принято называть **алгоритмами**.

**6. Рекурсивные функции. Основные определения.**

**Рекурсивной** называют функцию, которая прямо или косвенно сама вызывает себя.

Функция называется **косвенно** **рекурсивной** в том случае, если она содержит обращение к другой функции, содержащей прямой или косвенный вызов определяемой (первой) функции.

Если в теле функции явно используется вызов этой же функции, то имеет место **прямая** **рекурсия**.

Рекурсия называется **однократной**, если функция вызывает саму себя один раз. Если функция вызывает саму себя два раза, то рекурсия называется **двукратной** **и т.д.**

**7. Элементарные арифметические функции. Основные операции над ними.**

**Функции:**

1. Постоянная y = с (с - константа);

2. Степенная , a ϵ R;

3. Показательная , a > 0;

4. Логарифмическая , a > 0, a != 0;

5. Тригонометрические y = sin x, y = cos x, y = tg x, y = ctg x;

6. Обратные тригонометрические y = arcsin x, y = arccos x, y = arctg x, y = arcctg x;

называются **основными** **элементарными** **функциями**.

Всякая функция f, которая может быть задана с помощью формулы y = f(x), содержащей лишь конечное множество арифметических операций над основными элементарными функциями и композиций, называется **элементарной** **функцией**.

**В множестве элементарных функций выделяются следующие классы:**

1. Многочлены (полиномы) - функции вида

2. Рациональные функции (рациональные дроби) - функции f(x), представимые в виде

3. Иррациональные функции, т. е. такие функции, не являющиеся рациональными, которые могут быть заданы композицией конечного числа рациональных функций, степенных функций с рациональными показателями и четырех арифметических действий.

4. Трансцендентные функции - элементарные функции, не являющиеся рациональными или иррациональными.

**8. Понятие частично-рекурсивной функции.**

**Частично рекурсивными** называют функции, которые можно получить с помощью правил минимизации, подстановки и рекурсии из константной функции 0, функции I(x)=x+1, и набора функций , где .

Частично рекурсивная функция может быть не определена для некоторых значений аргументов.

**9. Тезис Чёрча.**

**Тезис Чёрча-Тьюринга:** Всякая эффективно вычислимая функция является вычислимой по Тьюрингу. Другими словами, для всякой функции, которую можно каким-либо способом вычислить, можно построить вычисляющую ее машину Тьюринга.

**10. Машина Поста. Основные понятия и операции.**

**Машина Поста** — абстрактная вычислительная машина, предложенная Эмилем Леоном Постом, которая отличается от машины Тьюринга большей простотой. Обе машины алгоритмически «эквивалентны» и были придуманы для уточнения понятия «алгоритм». В 1936 г. американский математик Эмиль Пост в статье описал систему, обладающую алгоритмической простотой и способную определять, является ли та или иная задача алгоритмически разрешимой. Если задача имеет алгоритмическое решение, то она представима также в форме последовательности команд для машины Поста.

Машина Поста состоит из каретки (или считывающей и записывающей головки) и разбитой на ячейки бесконечной в обе стороны ленты. Каждая ячейка ленты может находиться в 2 состояниях — быть либо пустой — 0, либо помеченной меткой 1. За такт работы машины каретка может сдвинуться на одну позицию влево или вправо, считать, изменить символ в своей текущей позиции.

Работа машины Поста определяется программой, состоящей из конечного числа строк. Для работы машины нужно задать программу и её начальное состояние (то есть состояние ленты и позицию каретки). Кареткой управляет программа, состоящая из пронумерованных не обязательно упорядоченных строк команд, если в каждой команде указана строка, на которую нужно перейти. Обычно принимается, что если в команде переход не указан, то переход происходит на следующую строку. Каждая команда имеет следующий синтаксис: i k j, где i — номер команды, K — действие каретки, j — номер следующей команды.

**Всего для машины Поста существует шесть типов команд:**

V j — поставить метку, перейти к j-й строке программы;

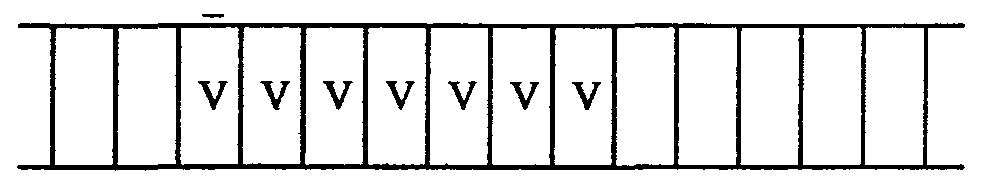
X j — стереть метку, перейти к j-й строке;

← j — сдвинуться влево, перейти к j-й строке;

→ j — сдвинуться вправо, перейти к j-й строке;

? j1; j2 — если в ячейке нет метки, то перейти к j1-й строке программы, иначе перейти к j2-й строке;

! — конец программы («стоп», останов).

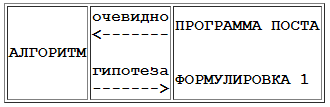


**11. Финитный 1 – процесс. Способ задания проблемы и формулировка 1.**

**Общая проблема называется по Посту 1-заданой**, если существует такой финитный 1 – процесс, что, будучи, применим к n є N в качестве исходной конфигурации ящиков, он задает n-ую конкретную проблему в виде набора помеченных ящиков.

Если общая проблема 1-задана и 1-разрешима, то, соединяя наборы инструкций по заданию проблемы, и ее решению мы получаем ответ по номеру проблемы – это и есть в терминах статьи Поста **формулировка** **1.**

Таким образом, **гипотеза Поста** состоит в том, что любые более широкие формулировки в смысле алфавита символов ленты, набора инструкций, представления и интерпретации конкретных проблем сводимы к формулировке 1.



Следовательно, если гипотеза верна, то любые другие формальные определения, задающие некоторый класс алгоритмов, эквивалентны классу алгоритмов, заданных формулировкой 1 Эмиля Поста.

**12. Машина Тьюринга.**

**Машина Тьюринга** — абстрактная вычислительная машина. Была предложена Аланом Тьюрингом в 1936 году для формализации понятия алгоритма.

Машина Тьюринга является расширением конечного автомата и, согласно тезису Чёрча — Тьюринга, способна имитировать всех исполнителей (с помощью задания правил перехода), каким-либо образом реализующих процесс пошагового вычисления, в котором каждый шаг вычисления достаточно элементарен.

**В состав Машины Тьюринга** входит бесконечная в обе стороны лента, разделённая на ячейки, и управляющее устройство с конечным числом состояний.

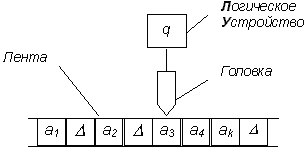
Управляющее устройство может перемещаться влево и вправо по ленте, читать и записывать в ячейки символы некоторого конечного алфавита. Выделяется особый пустой символ, заполняющий все клетки ленты, кроме тех из них (конечного числа), на которых записаны входные данные.

В **управляющем устройстве** содержится таблица переходов, которая представляет алгоритм, реализуемый данной Машиной Тьюринга. Каждое правило из таблицы предписывает машине, в зависимости от текущего состояния и наблюдаемого в текущей клетке символа, записать в эту клетку новый символ, перейти в новое состояние и переместиться на одну клетку влево или вправо. Некоторые состояния Машины Тьюринга могут быть помечены как **терминальные**, и переход в любое из них означает конец работы, остановку алгоритма.

Машина Тьюринга называется **детерминированной**, если каждой комбинации состояния и ленточного символа в таблице соответствует не более одного правила, и недетерминированной в противном случае.

**Работа машины Тьюринга:**

Конкретная машина Тьюринга задаётся перечислением элементов множества букв алфавита A, множества состояний Q и набором правил, по которым работает машина. Они имеют вид: (если головка находится в состоянии , а в обозреваемой ячейке записана буква , то головка переходит в состояние , в ячейку вместо записывается , головка делает движение , которое имеет три варианта: на ячейку влево (L), на ячейку вправо (R), остаться на месте (N)). Для каждой возможной конфигурации <, > имеется ровно одно правило (для недетерминированной машины Тьюринга может быть большее количество правил). Правил нет только для заключительного состояния, попав в которое, машина останавливается. Кроме того, необходимо указать конечное и начальное состояния, начальную конфигурацию на ленте и расположение головки машины.



**13. Алгоритмические неразрешимые проблемы.**

**Теорема.** Не существует алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма и его исходных данных (и алгоритм и данные заданы символами на ленте машины Тьюринга) определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.

Таким образом, фундаментально алгоритмическая неразрешимость связана с бесконечностью выполняемых алгоритмом действий, т.е. невозможностью предсказать, что для любых исходных данных решение будет получено за конечное количество шагов.

Тем не менее, можно попытаться сформулировать причины, ведущие к алгоритмической неразрешимости, эти причины достаточно условны, так как все они сводимы к проблеме останова, однако такой подход позволяет более глубоко понять природу алгоритмической неразрешимости:

а) Отсутствие общего метода решения задачи

б) Информационная неопределенность задачи

в) Логическая неразрешимость

**14. Проблема соответствий Поста над алфавитом Σ.**

В общем случае мы можем построить частичный алгоритм, основанный на идее упорядоченной генерации возможных последовательностей цепочек с проверкой выполнения условий задачи. Если последовательность является решающей – то мы получаем результативный ответ за конечное количество шагов. Поскольку общий метод определения отсутствия решающей последовательности не может быть указан, т.к. задача сводима к проблеме «останова» и, следовательно, является алгоритмически неразрешимой, то при отсутствии решающей последовательности алгоритм порождает бесконечный цикл.

В теории алгоритмов такого рода проблемы, для которых может быть предложен частичный алгоритм их решения, частичный в том смысле, что он возможно, но не обязательно, за конечное количество шагов находит решение проблемы, называются **частично разрешимыми проблемами**.

В частности, проблема останова так же является частично разрешимой проблемой, а проблемы эквивалентности и тотальности не являются таковыми.

**16. Сравнительные оценки алгоритмов. Система обозначений в анализе алгоритмов. Классификация алгоритмов по виду функции трудоёмкости.**

Под **трудоёмкостью алгоритма** для данного конкретного входа , будем понимать количество «элементарных» операций совершаемых алгоритмом для решения конкретной проблемы в данной формальной системе.

Комплексный анализ алгоритма может быть выполнен на основе комплексной оценки ресурсов формальной системы, требуемых алгоритмом для решения конкретных проблем. Очевидно, что для различных областей применения веса ресурсов будут различны, что приводит к следующей комплексной оценке алгоритма:

, где – веса ресурсов.

**Количество операций**, выполняемых над входом одинаковой длины N, зависит от конкретного набора данных.

Пусть – множество набора данных, заданное в формальной системе.

– конкретный набор входных данных.

– мощность множества .

Алгоритм, решая различные задачи для размерности N выполнит разное число операций:

– худший случай для

– наилучший случай для

– средний случай или среднее число операций, совершаемых алгоритмом А для решения задач размерностью V.

В зависимости от влияния исходных данных на функцию трудоемкости алгоритма может быть предложена следующая **классификация**, имеющая практическое значение для анализа алгоритмов:

**1. Количественно-зависимые по трудоемкости алгоритмы.** Это алгоритмы, функция трудоемкости которых зависит только от размерности конкретного входа, и не зависит от конкретных значений. Примерами алгоритмов с количественно-зависимой функцией трудоемкости могут служить алгоритмы для стандартных операций с массивами и матрицами – умножение матриц, умножение матрицы на вектор и т.д.

**2.** **Параметрически-зависимые по трудоемкости алгоритмы.** Это алгоритмы, трудоемкость которых определяется не размерностью входа, а конкретными значениями обрабатываемых слов памяти. Примерами алгоритмов с параметрически-зависимой трудоемкостью являются алгоритмы вычисления стандартных функций с заданной точностью путем вычисления соответствующих степенных рядов.

, где n – длина ввода, m – число параметров

**3.** **Количественно-параметрические по трудоемкости алгоритмы.** Однако в большинстве практических случаев функция трудоемкости зависит как от количества данных на входе, так и от значений входных данных, в этом случае. В качестве примера можно привести алгоритмы численных методов, в которых параметрически-зависимый внешний цикл по точности включает в себя количественно-зависимый фрагмент по размерности.

**4.** **Порядково-зависимые по трудоемкости алгоритмы.** Примерами таких алгоритмов могут служить ряд алгоритмов сортировки, алгоритмы поиска минимума и максимума в массиве.

, где и

**17. Асимптотический анализ функций. Трудоемкость алгоритмов и временные оценки.**

**Асимптотический анализ** применяется для оценки количества операций, лежащих в основе определения сложности.

**Целью** является оценка скорости роста числа операция при возрастании объема входных данных.

**Виды оценок:**

**1. Оценка снизу Ω.**

или , если существуют целые N и C, такие, что для всех

1) Определён класс функций, которые растут не медленнее, чем g(n) с точностью до постоянного множителя, начиная с некоторого объема данных N.

2) Запись Ω(k\*lnk) означает класс функций, которые растут не медленнее, нежели g(k)=k\*lnk.

3) Традиционно в данном классе заключены все полиномы, степени больше единичной и все степенные функции, с основанием, большим единицы.

**2. Оценка сверху O.**

или , если существуют целые N и C, такие, что для всех

1) Оценка О определяет класс функций, которые растут не быстрее, чем функция g(n) с точностью до некоторого постоянного множителя.

2) Функция g(n) мажорирует f(n).

**3. Оценка θ.**

или , если существуют целые N, и такие, что для всех

1) Оценка θ интерпретируется как одновременное выполнения оценок Ω и O.

2) Формально θ[g(n)] есть пересечение Ω и О.

3) Запись f(n)=θ(1) – означает, что функция либо равна константе, либо ограничена константой на бесконечности.

**18. Элементарные операции в языке записи алгоритмов.**

**Элементарные операции:**

1) Простое присваивание: a←b (или a:=b:)

2) Одномерная индексация a[i]: (адрес (a)+i\*длина элемента)

3) Арифметические операции: (\*, /, -, +)

4) Операции сравнения: a<b

5) Логические операции: or, and, not

**Конструкции:**

1) *Следование.* Трудоемкость конструкции есть сумма трудоемкостей блоков, следующих друг за другом.

2) *Ветвление.* Общая трудоемкость конструкции «Ветвление» требует анализа вероятности выполнения переходов на блоки «Then» и «Else» и определяется как:

3) *Цикл.* После сведения конструкции к элементарным операциям ее трудоемкость определяется как:

**19. Примеры анализа простых алгоритмов.**

**Пример 1:** Задача суммирования элементов квадратной матрицы

SumM (A,n;Sum)

Sum:=0

For i:=1 to n

For j:=1 to n

Sum:=Sum+A[i,j]

end for

Return(Sum)

End

Алгоритм выполняет одинаковое количество операций при фиксированном значении n, и, следовательно, является количественно-зависимым. Применение методики анализа конструкции «Цикл» дает:

**Пример 2:** Задача поиска максимума в массиве

MaxS(S,n;Max)

Max:=S[1]

For i:=2 to n

if Max<S[i]

then Max:=S[i]

end for

return Max

End

Данный алгоритм является количественно-параметрическим, поэтому для фиксированной размерности исходных данных необходимо проводить анализ для худшего, лучшего и среднего случая:

А) Худший случай.

Б) Лучший случай.

В) Худший случай.

Величина H называется n-ым гармоническим числом. Таким образом, точное значение среднего количества операций присваивания в алгоритме поиска максимума в массиве из n элементов определяется величиной Н:

**20. Переход к временным оценкам.**

**1) Пооперационный анализ:**

Идея пооперационного анализа состоит в получении пооперационной функции трудоемкости для каждой из используемых алгоритмом элементарных операций с учетом типов данных. Следующим шагом является экспериментальное определение среднего времени выполнения данной элементарной операции на конкретной вычислительной машине. Ожидаемое время выполнения рассчитывается как сумма произведений пооперационной трудоемкости на среднее время операции:

**2) Метод Гиббсона:**

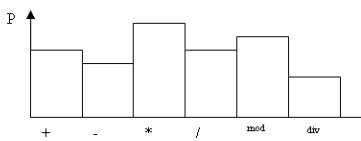
Метод предполагает проведение совокупного анализа по трудоемкости и переход к временным оценкам на основе принадлежности решаемой задачи к одному из следующих типов:

а) задачи научно-технического характера с преобладанием операций с операндами действительного типа

б) задачи дискретной математики с преобладанием операций с операндами целого типа

в) задачи баз данных с преобладанием операций с операндами строкового типа

Далее на основе анализа множества реальных программ для решения соответствующих типов задач определяется частотная встречаемость операций, создаются соответствующие тестовые программы, и определяется среднее время на операцию в данном типе задач –



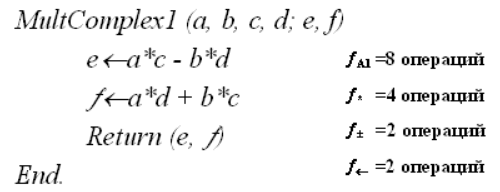
На основе полученной информации оценивается общее время работы алгоритма в виде:

**3) Метод прямого определения среднего времени:**

В этом методе так же проводится совокупный анализ по трудоемкости — определяется , после чего на основе прямого эксперимента для различных значений определяется среднее время работы данной программы и на основе известной функции трудоемкости рассчитывается среднее время на обобщенную элементарную операцию, порождаемое данным алгоритмом, компилятором и компьютером — . Эти данные могут быть интерполированы или экстраполированы на другие значения размерности задачи следующим образом:

**21. Пример пооперационного временного анализа.**

В качестве примера рассмотрим задачу умножения двух комплексных чисел: (a+bi)\*(c+di)=(ac-bd)+i(ad+bc)=e+fi



**22. Теория сложности вычислений и сложностные классы задач. Теоретический предел трудоемкости задачи.**

Рассматривая некоторую алгоритмически разрешимую задачу, и анализируя один из алгоритмов ее решения, мы можем получить оценку трудоемкости этого алгоритма в худшем случае —. . Такие же оценки мы можем получить и для других известных алгоритмов решения данной задачи. Рассматривая задачу с этой точки зрения, возникает резонный вопрос – а существует ли функциональный нижний предел для если «да», то существует ли алгоритм, решающий задачу с такой трудоемкостью в худшем случае.

**23. Сложностные классы задач.**

1) Класс P (задачи с полиномиальной сложностью или задачи, для решения которых достаточно детерминированных полиномиальных алгоритмов). Задача называется полиномиальной, т.е. относится к классу P, если существует константа kи алгоритм, решающий задачу с , где n—длина входа алгоритма в битах n= |D|. Преимущества алгоритмов из этого класса:

для большинства задач из класса P константа k меньше 6;

класс P инвариантен по модели вычислений (для широкого класса моделей);

класс P обладает свойством естественной замкнутости (сумма или произведение полиномов есть полином).

2) Класс NP (полиномиально проверяемые задачи или задачи, решаемые недетерминированными алгоритмами в течение полиномиального времени).

**24. Проблема P = NP. Класс NPC (NP – полные задачи)**

После введения в теорию алгоритмов понятий сложностных классов Эдмондсом P = NP? Словесная формулировка проблемы имеет вид: можно ли все задачи, решение которых проверяется с полиномиальной сложностью, решить за полиномиальное время? Любая задача, принадлежащая классу P, принадлежит и классу NP , т.к. она может быть полиномиально проверена – задача проверки решения может состоять просто в повторном решении задачи. На сегодня отсутствуют теоретические доказательства как совпадения этих классов ( P = NP ), так и их несовпадения. Предположение состоит в том, что класс P является собственным подмножеством класса NP, т.е. NP \ P не пусто. NP-трудные (NP-сложные) задачи — это класс задач, для которого детерминированный полиномиальный алгоритм решения задачи может быть использован для решения любой задачи из NP. Понятие NP–полноты основывается на понятии сводимости одной задачи к другой. Сводимость может быть представлена следующим образом: если мы имеем задачу 1 и решающий эту задачу алгоритм, выдающий правильный ответ для всех конкретных проблем, составляющих задачу, а для задачи 2 алгоритм решения неизвестен, то если мы можем переформулировать (свести) задачу 2 в терминах задачи 1, то мы решаем задачу 2.

**25. Примеры NP – полных задач.**

Задача о выполнимости схемы

Формулировка задачи — существует ли для данной схемы выполняющий набор значений входа. Очевидно, что задача принадлежит классу NP—проверка предъявленного выполняющего набора не сложнее количества функциональных элементов и, следовательно, не больше чем knO.

**26. Категории чисел по Колмогорову.**

Число **А** назовем **малым**, если возможно практически перебрать все схемы из А элементов с двумя входами и выходами (или, что-то же, выписать для них все функции алгебры логики с A аргументами);

Число **В** назовём **средним**, если мы оказываемся не в состоянии перебрать практически все схемы из В элементов, а можем перебрать лишь сами эти элементы или (что чуть-чуть сложнее) выработать систему обозначений для любой системы из В элементов;

Число **С** - **большое**, если мы не в состоянии практически перебрать такое число элементов, а можем лишь установить систему обозначений для этих элементов;

И, наконец, числа - **сверхбольшие**, если и этого практически нельзя сделать (они нам, как мы увидим дальше, и не понадобятся).

**27. Пример полного анализа алгоритма решения задачи о сумме. Формулировка задачи и асимптотическая оценка.**

Словесно задача о сумме формулируется как задача нахождения таких чисел из данной совокупности, которые в сумме дают заданное число, классически задача формулируется в терминах целых чисел.

Дано: Массив S[i], i={1,N} и число V. Требуется: определить такие Sj, что ∑Sj=V. Тривиальное решение определяется равенством V=Sum, где Sum=∑Sj, условия существования решения имеют вид: min Sj, i=1, N<=V<=Sum. Получим асимптотическую оценку сложности решения данной задачи для алгоритма, использующего прямой перебор всех возможных вариантов.

V содержит 1 слагаемое - вариантов;

V содержит 2 слагаемых - вариантов;

V содержит 3 слагаемых - вариантов;

**28. Алгоритм точного решения задачи о сумме (метод перебора). Анализ алгоритма точного решения задачи о сумме.**

TASKSUM(S,N,V; CNT,FL)  
FL <-- false  
i <-- 1  
repeat  
Cnt[i] <-- 0  
i <-- i+1  
Until i > N  
Cnt[N] <-- 1  
Repeat  
Sum <-- 0  
i <-- 1  
Repeat  
Sum <-- Sum + S[i] \* Cnt[i]

i <-- i+1  
Until i > N  
if Sum = V  
FL <-- true  
Return (Cnt,FL)  
j <-- N  
While Cnt[j] = 1  
Cnt[j] = 0  
j <-- j-1  
Cnt[j] <-- 1  
Until Cnt[0] = 1  
Return(Cnt,FL)

**29. Алгоритмы сортировки, слияния и поиска.**

**Сортировка**— упорядочение (перестановка) элементов в подмножестве данных по какому-либо критерию. Чаще всего в качестве критерия используется некоторое числовое поле, называемое ключевым.

**Поиск** — обработка некоторого множества данных с целью выявления подмножества данных, соответствующего критериям поиска.

**Все алгоритмы поиска делятся на:**

поиск в неупорядоченном множестве данных;

поиск в упорядоченном множестве данных.

**30. Сортировка вставками.**

Сортировка вставками — [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. [Вычислительная сложность](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C) — {\displaystyle O(n^{2})}O(n2).

**31. Сортировка методом пузырька.**

Сортировка простыми обменами, сортировка пузырьком — простой [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8). Для понимания и реализации этот алгоритм — простейший, но эффективен он лишь для небольших массивов. Сложность алгоритма: [{\displaystyle O}](https://ru.wikipedia.org/wiki/O_%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%BE%D0%B5){\displaystyle (n^{2})} O(n2). Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву

повторяются {\displaystyle N-1} до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива.

**32. Сортировка Шелла.**

При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии {\displaystyle d}d  {\displaystyle d}, а завершается сортировка Шелла упорядочиванием элементов при {\displaystyle d=1}d=1 (то есть обычной [сортировкой вставками](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%BE%D0%B9)). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, [пузырьковой](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BC_%D0%BF%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%B0), каждая перестановка двух элементов уменьшает количество [инверсий](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F_(%D0%BF%D0%B5%D1%80%D0%B5%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B0)) в списке максимум на 1, а при сортировке Шелла это число может быть больше).

Невзирая на то, что сортировка Шелла во многих случаях медленнее, чем [быстрая сортировка](https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0), она имеет ряд преимуществ:

1) отсутствие потребности в памяти под стек;

2) отсутствие деградации при неудачных наборах данных — быстрая сортировка легко деградирует до O(n²), что хуже, чем худшее гарантированное время для сортировки Шелла.

**33. Сортировка пирамидальная.**

Пирамидальная сортировка — [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за [*Θ*](https://ru.wikipedia.org/wiki/%D0%9E-%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D0%B8%D1%8F)*(n* log *n)* операций при сортировке *n* элементов.[[2]](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D1%80%D0%B0%D0%BC%D0%B8%D0%B4%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0#cite_note-2) Количество применяемой служебной памяти не зависит от размера массива (то есть, O(1)).

Сортировка пирамидой использует [бинарное сортирующее дерево](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0). Сортирующее дерево — это такое дерево, у которого выполнены условия:

1. Каждый лист имеет глубину либо {\displaystyle d}d, либо {\displaystyle d-1}d-1, {\displaystyle d}d — максимальная глубина дерева.

2. Значение в любой вершине не меньше (другой вариант — не больше) значения её потомков.

**34. Сортировка слиянием.**

**Сортировка слиянием** — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам, которых можно получать только последовательно, например — потоки) в определённом порядке.  
**Слияние**означает объединение двух (или более) последовательностей в одну упорядоченную последовательность при помощи циклического выбора элементов, доступных в данный момент.  
Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Затем их решения комбинируются, и получается решение исходной задачи.

**35. Быстрая сортировка.**

Быстрая сортировка — один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем {\displaystyle O(n\log n)}O(n\*log(n)) обменов при упорядочении {\displaystyle n}n элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками. QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «[Пузырьковая сортировка](https://ru.wikipedia.org/wiki/%D0%9F%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)» и «[Шейкерная сортировка](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D0%B9%D0%BA%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0" \o "Шейкерная сортировка)»), известного, в том числе, своей низкой эффективностью. Общая идея алгоритма состоит в следующем:

1) Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность (см. ниже).

2) Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие».

3) Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

**36. Последовательный поиск (линейный).**

Линейный, последовательный поиск — [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) нахождения заданного значения произвольной функции на некотором отрезке. Если отрезок имеет длину N, то найти решение с точностью до {\displaystyle \epsilon }έ можно за время {\displaystyle N \over \epsilon }. Т.о. [асимптотическая сложность алгоритма](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0) — O(n){\displaystyle O(n)}. В связи с малой эффективностью по сравнению с другими алгоритмами линейный поиск обычно используют, только если отрезок поиска содержит очень мало элементов, тем не менее, линейный поиск не требует дополнительной памяти или обработки/анализа функции, так что может работать в потоковом режиме при непосредственном получении данных из любого источника. Также линейный поиск часто используется в виде линейных алгоритмов поиска максимума/минимума.

**37. Двоичный поиск.**

Двоичный (бинарный) поиск (также известен как метод деления пополам и [дихотомия](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%85%D0%BE%D1%82%D0%BE%D0%BC%D0%B8%D1%8F)) — классический [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) поиска элемента в отсортированном массиве (векторе), использующий дробление массива на половины.

1. Определение значения элемента в середине структуры данных. Полученное значение сравнивается с ключом.

2. Если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй.

3. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.

4. Процесс продолжается до тех пор, пока не будет найден элемент со значением ключа или не станет пустым интервал для поиска.

**38. Комбинаторные алгоритмы.**

Индекс сочетания

Каждому сочетанию, перестановке, размещению и другим комбинаторным объектам можно сопоставить индекс — это номер, в котором он появляется при переборе данным алгоритмом. Есть вариант перебора «в лоб». Включаем счетчик сочетаний, берем алгоритм выше и пока не дойдем до нужного варианта перебираем все подряд. Такой вариант имеет очень большую временную сложность, поэтому такой вариант отбросим.  
Положим, на входе имеются числа i1, i2, i3. Нам, прежде всего, нужно их упорядочить в порядке возрастания (обратите внимание, что сам алгоритм генерации, приведенный выше, выдает их всегда в упорядоченном виде, хотя само понятие «сочетание» подразумевает произвольный порядок).

**40. Представление комбинаторных объектов.**

Комбинаторные объекты — конечные множества, на элементы которых могут накладываться определённые ограничения, такие как: различимость или неразличимость элементов, возможность повторения одинаковых элементов и т. п. Если два комбинаторных объекта, различающихся только порядком элементов, считаются различными, то они называются упорядоченными.

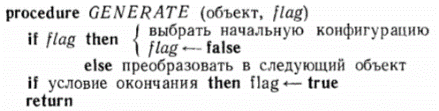
1) [Битовые вектора](https://neerc.ifmo.ru/wiki/index.php?title=%D0%9F%D0%BE%D0%BB%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%B0_%D0%BF%D0%BE_%D0%BD%D0%BE%D0%BC%D0%B5%D1%80%D1%83#.D0.91.D0.B8.D1.82.D0.BE.D0.B2.D1.8B.D0.B5_.D0.B2.D0.B5.D0.BA.D1.82.D0.BE.D1.80.D0.B0) — последовательность нулей и единиц заданной длины.

2) Перестановки — упорядоченный набор чисел 1,2,…,n1,2,…,n, обычно трактуемый как биекция на множестве {1,2,…,n}{1,2,…,n}, которая числу ii ставит соответствие ii-й элемент из набора.

3) Перестановки с повторениями — те же перестановки, однако некоторые элементы могут встречаться несколько раз.

4) Размещение из nn по kk — упорядоченный набор из kk различных элементов некоторого nn-элементного множества.

**41. Порождение элементарных комбинаторных объектов.**



**42. Рекурсивные алгоритмы. Примеры рекурсивных алгоритмов.**

**Рекурсия** – фундаментальное понятие в математике и компьютерных науках. В языках программирования рекурсивной программой называется программа, которая обращается *сама к себе*. Рекурсивная программа не может вызывать себя до бесконечности, следовательно, вторая важная особенность рекурсивной программы – наличие *условия завершения*, позволяющее программе прекратить вызывать себя.

Function Factorial(n:byte):longint;  
begin  
  if n=0   
    then Fact:=1  
    else Fact:=n\*Fact(n-1)  
end;

Function Fibonachi(n:byte):longint;  
begin  
  if n <= 1   
    then F:=1  
    else F:= F(n-1)+F(n-2)  
end;

**43. Анализ сложности рекурсивных алгоритмов.**

При относительной простоте написания, у рекурсивных подпрограмм часто встречается существенный недостаток – неэффективность. Так, сравнивая скорость вычисления чисел Фибоначчи с помощью итеративной и рекурсивной функции можно заметить, что итеративная функция выполняется почти «мгновенно», не зависимо от значения n. При использовании же рекурсивной функции уже при n=40 заметна задержка при вычислении, а при больших n результат появляется весьма нескоро.

Неэффективность рекурсии проявляется в том, что одни и те же вычисления производятся по многу раз.

**44. Трудоемкость рекурсивной реализации алгоритмов.**

Оценить сложность рекурсивных вычислений (количество рекурсивных вызовов) можно с помощью рекуррентных соотношений. Рекуррентное соотношение – это рекурсивная функция с целочисленными значениями. Значение любой такой функции можно определить, вычисляя все ее значения начиная с наименьшего, используя на каждом шаге ранее вычисленные значения для подсчета текущего значения.

Рекуррентные выражения используются, в частности, для определения сложности рекурсивных вычислений.

**45. Методика подсчета вершин рекурсивного дерева. Фундаментальные алгоритмы на графах и деревьях.**

Один из методов анализа трудоемкости рекурсивного алгоритма строится на основе подсчета вершин рекурсивного дерева. Для оценки трудоемкости рекурсивных алгоритмов строится **полное дерево рекурсии**. Оно представляет собой *граф*, вершинами которого являются наборы *фактических параметров* при всех вызовах функции, начиная с первого обращения к ней, а *ребрами* – пары таких наборов, соответствующих взаимным вызовам. Вершины дерева рекурсии соответствуют фактическим вызовам рекурсивных функций. Следует заметить, что одни и те же наборы параметров могут соответствовать разным вершинам дерева. *Корень полного дерева рекурсивных вызовов* – это *вершина* полного дерева рекурсии, соответствующая начальному обращению к функции.

Важной характеристикой рекурсивного алгоритма является **глубина рекурсивных вызовов** – наибольшее одновременное количество рекурсивных обращений функции, определяющее максимальное количество слоев рекурсивного стека, в котором осуществляется хранение отложенных вычислений. Количество элементов полных рекурсивных обращений всегда не меньше глубины рекурсивных вызовов. При разработке рекурсивных программ необходимо учитывать, что глубина рекурсивных вызовов не должна превосходить максимального размера стека используемой вычислительной среды.

**46. Основные понятия и определения теории графов. Алгоритм поиска минимального остовного дерева.**

Геометрическое представление графа — это схемы, состоящие из точек и соединяющих эти точки отрезков прямых или кривых.

Схема, состоящая из «изолированных» вершин, называется нулевым графом.

Графы, в которых не построены всевозможные ребра, называются неполными графами.

Изолированные вершины—это такие вершины, которые не имеют инцидентных ребер, т.е. их степень нулевая.

Висячие вершины—это такие вершины, которые имеют только одно инцидентное ребро.

Маршрут графа—это чередующаяся последовательность вершин и ребер. Эта последовательность начинается и кончается вершиной, в которой каждое ребро инцидентно двум вершинам.

**Минимальное остовное дерево** в связанном взвешенном [неориентированном графе](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B3%D1%80%D0%B0%D1%84) — это [остовное дерево](https://ru.wikipedia.org/wiki/%D0%9E%D1%81%D1%82%D0%BE%D0%B2%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE" \o "Остовное дерево) этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него рёбер.

Существует несколько алгоритмов для нахождения минимального остовного дерева. Некоторые наиболее известные из них перечислены ниже:

1) [Алгоритм Прима](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9F%D1%80%D0%B8%D0%BC%D0%B0)

2) [Алгоритм Краскала](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9A%D1%80%D0%B0%D1%81%D0%BA%D0%B0%D0%BB%D0%B0)

3) [Алгоритм Борувки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%91%D0%BE%D1%80%D1%83%D0%B2%D0%BA%D0%B8)

**47. Основные понятия и определения теории графов. Алгоритм поиска кратчайшего пути. Эвристические алгоритмы. Жадные алгоритмы.**

Эвристический алгоритм (эвристика) — [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) решения задачи, включающий практический метод, не являющийся гарантированно точным или оптимальным, но достаточный для решения поставленной задачи. Позволяет ускорить решение задачи в тех случаях, когда точное решение не может быть найдено. Эвристика, в отличие от корректного алгоритма решения задачи, обладает следующими особенностями.

1) Она не гарантирует нахождение лучшего решения.

2) Она не гарантирует нахождение решения, даже если оно заведомо существует (возможен «пропуск цели»).

3) Она может дать неверное решение в некоторых случаях.

Жадный алгоритм — алгоритм, заключающийся в принятии локально [оптимальных решений](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5) на каждом этапе, допуская, что конечное решение также окажется оптимальным. Известно, что если структура задачи задается [матроидом](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D1%80%D0%BE%D0%B8%D0%B4" \o "Матроид), тогда применение жадного алгоритма выдаст глобальный оптимум. В типичном случае доказательство оптимальности следует такой схеме:

1. Доказывается, что жадный выбор на первом шаге не закрывает пути к оптимальному решению: для всякого решения есть другое, согласованное с жадным выбором и не хуже первого.

2. Показывается, что подзадача, возникающая после жадного выбора на первом шаге, аналогична исходной.

3. Рассуждение завершается по [индукции](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%B8%D0%BD%D0%B4%D1%83%D0%BA%D1%86%D0%B8%D1%8F).