ВВЕДЕНИЕ

Современное формальное определение **алгоритма** было дано в 30 - 50-х гг. XX века в работах А. Тьюринга, Э. Поста, А. Чёрча, Н. Винера, А. А. Маркова.

Само слово «алгоритм» происходит от имени учёного Абу Абдуллах Мухаммеда ибн Муса аль-Хорезми. Около 825 г. он написал сочинение, в котором впервые дал описание придуманной в Индии позиционной десятичной системы счисления. Аль-Хорезми сформулировал правила вычис-лений в новой системе и, вероятно, впервые использовал цифру 0 для обозначения пропущенной позиции в записи числа (её индийское название арабы перевели как as-sifr или просто sifr, отсюда такие слова, как «цифра» и «шифр»). Приблизительно в это же время

В первой половине XII века книга аль-Хорезми в латин-ском переводе проникла в Европу. Переводчик дал ей название Algoritmi de numero Indorum («Алгоритми о счёте индийском»). По-арабски же книга именовалась Китаб аль-джебр валь-мукабала («Книга о сложении и вычитании»). Из оригинального названия книги проис-ходит слово Алгебра.

Алгоритм (процедура) – решение задач в виде точных последовательно выполняемых предписаний. Это интуитивное определение сопровождается описанием интуитивных **свойств (признаков)** алгоритмов: эффективность, определенность, конечность.

Эффективность – возможность исполнения предписаний за конечное время.

Например, алгоритм – процедура, состоящая из “конеч-ного числа команд, каждая из которых выполняется меха-нически за фиксированное время и с фиксированными затратами”.

Функция алгоритмически эффективно вычислима, если существует механическая процедура, следуя которой для конкретных значений ее аргументов можно найти значе-ние этой функции .

Определенность – возможность точного математического определения или формального описания содержания команд и последовательности их применения в этой процедуре.

Конечность – выполнение алгоритма при конкретных исходных данных за конечное число шагов.

В **формальных** описаниях алгоритм конструктивно связы-вают с понятием **машины,** предназначенной для автома-тизированных преобразований символьной информации.

Для автоматических вычислений разрабатываются модели алгоритмов распознавания языков и машина, работающая с этими моделями. Таким образом, соединяют математи-ческое, формальное определение алгоритма и конструк-тивное, позволяющее реализовать модели вычислитель-ными машинами.

Общая теория алгоритмов занимается проблемой эффек-тивной **вычислимости.** Разработано несколько формаль-ных определений алгоритма, в которых эффективность и конечность вычислений может быть определена количест-венно – числом элементарных шагов и объемом требу-емой памяти.

Подобными моделями алгоритмических преобразований символьной информации являются:

- конечные автоматы;
- машина Тьюринга;
- машина Поста;
- ассоциативное исчисление или нормальные алгоритмы Маркова;
- рекурсивные функции.

Некоторые из этих моделей лежат в основе методов программирования и используются в алгоритмических языках.

В современной **программной инженерии** алгоритмы, как методы решения задач, по сравнению с традиционной математикой занимают ведущее место.

РЕГУЛЯРНЫЕ ЯЗЫКИ

Для того, чтобы представить формальное описание алго-ритма необходимо формальное описание решаемой зада-чи. В большинстве случаев описание задачи неформаль-ное (вербальное) и переход к алгоритму неформальный и требует верификации и тестирования и многократных ите-раций для приближения к решению.

Верификация (от лат. *verus* – «истинный» и *facere* – «де-лать») – проверка, способ подтверждения каких-либо тео-ретических положений, алгоритмов, программ и процедур путем их сопоставления с опытными (эталонными или эмпирическими) данными, алгоритмами и программами.

Тестирование применяется для определения соответствия предмета испытания заданным спецификациям .

К сожалению, теория алгоритмов не дает и не может дать как универсального, формального способа описания зада-чи, так и ее алгоритмического решения. Однако ценность теории состоит в том, что она дает примеры таких описа-ний и дает определение алгоритмически неразрешимых задач.

Один из примеров представлен регулярным языком, и задача формулируется как разработка алгоритма для распознавания принадлежности любого предложения к конкретному регулярному языку. Доказывается, что регулярный язык может быть формально преобразован в **модель алгоритма** решения этой задачи за конечное число шагов. Этой моделью является **конечный автомат.**

Алфавит языка обозначается как конечное множество символов. Например: **Σ={a,b,c,d}, Σ={0,1}.**

Символ и цепочка символов образуют слово – a, b, 0, abbcd, 0111000.

Пустое слово **(e)** не содержит символов.

Множество слов **S**={a, ab, aaa, bc} в алфавите **Σ** называют языком **L(Σ).**

Языки S=L(**Σ**) могут содержать неограниченное число слов, для их определения используют различные фор-мальные правила. В простейшем случае это **алгебра-ическая формула**, которая содержит операции формиро-вания слов из символов алфавита и ранее полученных слов.

Рассмотрим следующие операции формирования новых множеств из существующих множеств слов:

1) Символы алфавита могут соединяться **конкатенацией** (сцепление, соединение) в цепочки символов-слов, кото-рые соединяются в новые слова.

Конкатенация двух слов **x|y** обозначает, что к слову x справа приписано слово y или x|y=xy, причем xy≠yx.

Произведение S1|S2=S1S2 множеств слов S1 и S2 - это множество всех различимых слов, построенных конкатенацией соответствующих слов из S1 и S2.
Если S1={a, aa, ba}, S2={e, bb, ab}, то
S1S2={a, aa, ba, abb, aabb, baab, …}.

Для конкатенации выполняется ассоциативность, но коммутативность и идемпотентность не выполняются:

S1S2 ≠ S2S1;

SS ≠ S.

2) **объединение** (S1 ∪ S2) или (S1+ S2) множеств S1={a, aa, ba}, S2={e, bb, ab}, S1 ∪ S2={a, aa, ba, e, bb, ab}.

Для операции объединения выполняются следующие законы:

Коммутативность объединения S1 ∪ S2=S2 ∪ S1.

Идемпотентность объединения S ∪ S=S.

Ассоциативность объединения

S1 ∪(S2 ∪ S3) =(S1 ∪ S2) ∪ S3.

Дистрибутивность конкатенации и объединения

S1(S2 ∪ S3) = S1S2 ∪ S1S3.

3) **Итерация множества** {S}* состоит из пустого слова и всех слов вида S^0=e, S^1=S, S^2=SS, S^3=SSS.

Ассоциативность итерации S1 * (S2 * S3) =(S1 * S2)* S3.

Дистрибутивность объединения с итерацией

$S1 *(S2 \cup S3) = S1*S2 \cup S1*S3.$

Если a, b – любые регулярные выражения, то

$(a \cup b)^* = (a^* È b^*)^* = (a^*b^*)^* = (a^*b)^*a^*;$

$a^*=a^*a^*=(a^*)^*=(a \cup a^2 \cup .. \cup a^k)^*;$

$(a^*b)^*=(a \cup b)^*b.$

Таким образом, формулы могут содержать скобки и могут быть преобразованы с использованием этих законов.

Формулы, содержащие эти операции с множествами слов, называют **регулярными выражениями.**

Регулярные выражения допускают формальные алгебра-ические преобразования.

Языки, определяемые регулярными выражениями, называются **регулярными языками**, а множество слов - **регулярными множествами**.

Пример.

Регулярные выражения регулярного языка в алфавите

Σ={0,1}

(1(0)*)) = 0 ∪ 10*;

(0 ∪ 1)* = (0* ∪ 1*)*;

(0 ∪ 1)*011 - все слова из 0 и 1, заканчивающиеся на 011;

(a ∪ b)(a ∪ +b)* = (a ∪ b)(a* ∪ b*)* - слова, начинающиеся с a или b;

(00 ∪ 11)*((01 ∪ 10)(00 ∪ 11)*(01 ∪ 10)(00 ∪ 11)*)* - все слова, содержащие четное число 0 и 1.

Пример.

Регулярное выражение, определяющее правильное арифметическое выражение.

Входной алфавит **Σ={i, +, -},** где i – идентификатор;

(+, -) – знаки арифметических операций.

Примеры правильных арифметических выражений

i , - i , i + i , i - i , - i - i , i + i - i, …

Обозначим знаки арифметических действий буквами p = (+), m=(-).

Тогда соответствующие правильные (регулярные) арифметические выражения имеют вид **i, mi, ipi, imi, mimi, ipimi,** … и регулярное выражение, определяющее регулярный язык,

L(M)=(mi + i)((p + m)i)*.

КОНЕЧНЫЕ АВТОМАТЫ

Алгоритм распознавания предложений регулярного языка называют **конечным автоматом (КА)** .

Определение. Конечный автомат определяется символами

M=(Q, Σ, δ, q0, F), где

Q={q0, q1, …, qn} – конечное множество состояний;

Σ={a, b, c, …} – входной алфавит (конечное множество);

δ: Q* Σ →{Pj } – функция переходов, Pj - подмножество Q.

q0 – начальное состояние;

F – множество заключительных состояний

Конечное множество отношений для этого функционала-называемых может быть определено перечислением в **таблице переходов**:

Q	Σ	Pj
qi	a	qj

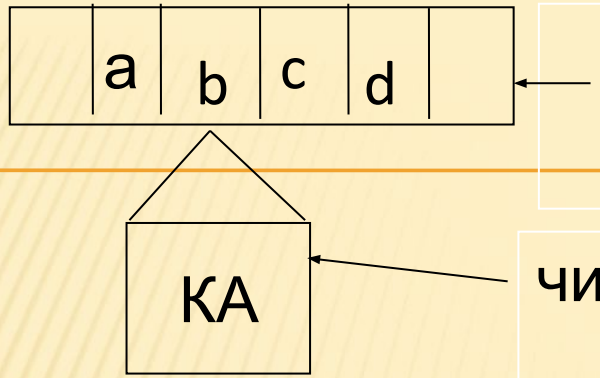
Конечный автомат называется **недетерминированным** (НДКА), если Pj содержит

КА называется **детерминированным** (ДКА), если Pj содержит не более одного состояния.

КА **полностью определенный**, если Pj в детерминирован-ном автомате не пустое. Если есть пустые элементы мно-жества Pj, то автомат **частично определен**.

Работа КА или выполнение алгоритма распознавания слов регулярного языка могут быть представлены последо-вательностью шагов, которые определяются текущим сос-тояние Q, входным символом **Σ** и следующим состоянием Pj.

Используется конструктивное описание принципа работы КА как машины М, имеющей следующую организацию:



лента (память с возможностью чтения)

читающая головка, управляемая КА

КА читает входной символ в текущем состоянии qi ∈ Q, переходит в следующее состояние qj ∈ Q и сдвигает читающую головку к следующему символу. Автомат допускает входное слово, если приходит в зак-лючительное состояние из F, последовательно считывая символы из памяти и переходя в следующие состояния в соответствии с таблицей переходов. При этом входное слово исчерпано и автомат остановился.

Конфигурация КА K=(q,ω), где q-текущее состояние КА, ω - непрочитанная цепочка символов слова на ленте, включая символ под читающей головкой.

k = (q, ω) текущая конфигурация;

k_0 = (q_0, ω0) начальная конфигурация;

k_f = (q, e), где q ∈ F, - заключительная конфигурация и (e) – символ, обозначающий конец строки.

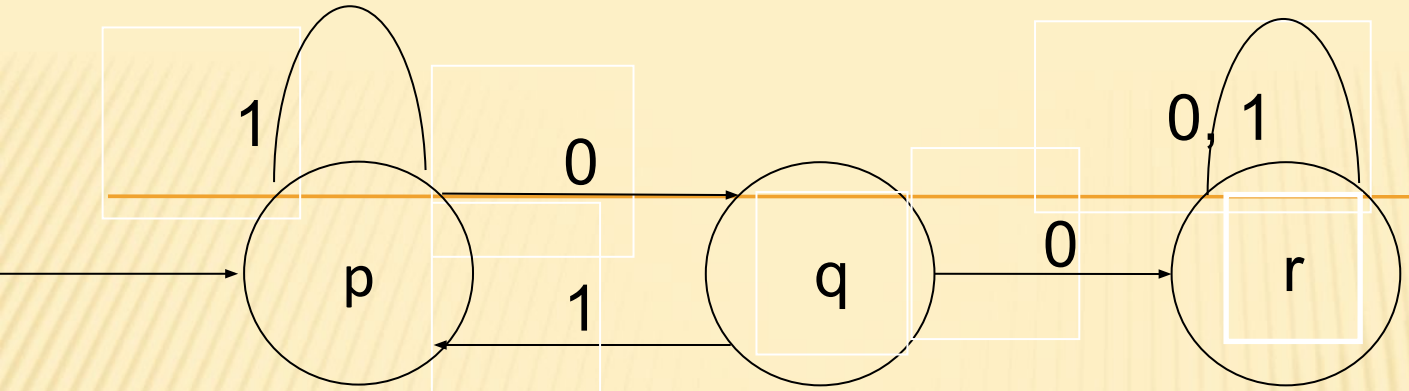
Шаг алгоритма - переход из одной конфигурации КА в другую

Ki →Kj или (qi, ωi) → (qj, ωj).

Функция переходов, заданная в табличной форме, может быть представлена графом переходов G=(Q, R), где Q - вершины графа, R - бинарное отношение между парой вершин, которое представлено множеством дуг (qi, qj)

(qi, qj) ∈ Q*Q, если существует символ **a** ∈ **Σ** и **δ** (qi, a) = qj.

На дугах графа (qi, qj) отмечаются соответствующие сим-волы алфавита.



Для ДКА, приведенного на рисунке состояния Q={p, q, r}, входной алфавит **Σ**={0,1}, начальное состояние p, конечное - r.

Таблица переходов ДКА

Q	Σ	
	0	1
p	q	p
q	r	p
r	r	r

Исполнение алгоритма это последовательность шагов, в которых изменяется конфигурация КА:

(p, 01001) →(q, 1001) → (p, 001) →(q, 01) → (r, 1) → (r, e);

(p, 01001)- начальная конфигурация;

(r, e) – конечная конфигурация.

В результате применения слова 01001 в начальном состоянии **p** автомат переходит в следующее состояние **q** и следующее значение цепочки символов на входе 1001.

Автомат M допускает слово ω0 , если существует (q0, ω0) → *(qf, e), где → *( ), обозначает **транзитивное замыкание** и существует путь, соединяющий q0 и qf для входного слова ω0.

Язык L(M), определяемый (распознаваемый, допускаемый) автоматом M включает множество всех слов, допускаемых M.

МАШИНА ТЬЮРИНГА

Универсальный КА, применяемый для решения любой алгоритмически разрешимой задачи, в теории алгоритмов и вычислений называется **машиной Тьюринга**.

Память машины допускает как чтение, так и запись на ленту в одну и ту же ячейку.

Множество входных и выходных символов не различа-ются - единый алфавит на входе (чтение) и на выходе (запись) $\Sigma=W$.

Функция перехода δ: $Q^* \Sigma \rightarrow Q$.

Функция выхода λ: $Q^* \Sigma \rightarrow K^* \Sigma$, где K - команды управления памятью (применительно к ленте: L -сдвиг влево, R - сдвиг вправо, N - лента неподвижна).

Принципы работы машины:

Начальное слово - в алфавите Σ размещается на ленте

При чтении очередного символа с ленты выполняется определенная команда K, автомат переходит в следующее состояние и в ту же позицию на ленте записывается символ;

Машина применима к входной последовательности, если достигает конечного состояния и останавливается.

Машина Тьюринга реализует алгоритм, если она всегда применима к начальной информации (слову), изобража-ющей условия задачи, и перерабатывает входное слово в результирующую информацию (выходное слово).

Конфигурация машины при исполнении алгоритма – K(qi,s), где qi -состояние автомата, s - текущая строка в памяти. Шаг алгоритма – переход от одной конфигурации к другой после чтения символа.

Машина Тьюринга имеет фундаментальное значение в теории алгоритмов как формальное определение алго-ритма в строгой и точной форме - в виде **схемы машины**.

Основная **гипотеза** теории алгоритмов: Машина Тьюринга решает любую алгоритмически разрешимую задачу.

Вопрос **алгоритмической разрешимости** (существования алгоритма решения задачи) сводится к доказательству существования машины Т, решающей задачу за конечное число шагов. Если число шагов бесконечно, то задача трудно разрешимая или алгоритмически неразрешимая проблема.

Однако в такой постановке проблема доказательства раз-решимости задачи сама является неразрешимой, так как отсутствует алгоритм построения такой машины (аналогия с формальным выводом в логике).

Поэтому задача сводится к частной и более простой зада-че, для которой можно доказать, что соответствующей машины Тьюринга построить нельзя и тогда, и более общая задача алгоритмически не разрешима.

РЕКУРСИВНЫЕ ФУНКЦИИ

Если зафиксировать алфавит A из r букв, то всякое слово R в этом алфавите можно рассматривать как запись некото-рого натурального числа в **r-ичной** системе счисления.

Таким образом, исходные данные – слова R в алфавите A можно интерпретировать как натуральные числа. Также можно интерпретировать результат выполнения алгорит-ма как вычисление значения числовой функции по заданному значению аргумента.

Рекурсивные фуйкции позволяют определить функцию от неизвестного к известному (от сложного к простому).

Функция **вычислима**, если существует алгоритм – **эффек-тивная** последовательная процедура вычисления от простого к сложному.

Выделяется **базис элементарных функций**, интуитивно вычислимых, и средства-**операторы** получения из них более сложных функций.

1. При вычислении значения натурального числа $N_n = N_{n-1} + 1$, $N_0 = 0$; можно записать ряд выражений для $N_{n-1}, N_{n-2}, \ldots N_4, N_3, N_2, N_1, N_0 = 0$ и затем последовательно вычислить N_n

2. Для суммы **n** чисел $\{a_1, a_2, \ldots, a_n\}$: $S_n = S_{n-1} + a_n, \ldots,$ $S_1 = S_0 + a_1$, $S_0 = 0$;

К элементарным вычислимым функциям относятся:

1) $S(x) = x+1$ - следование – вычисление следующего натурального числа;

2) $0(x) = 0$ – константа нуля;

3) $I_m(x_1x_2..x_n) = x_m$ - выбор аргумента x_m из n аргументов.

Операторы получения более сложных функций:

1) $H(x, y) = f(x)$ введение вспомогательной **фиктивной переменной**, при вычислении стирается y и вычисляется $f(x)$;

2) $\Phi(x) = g(f(x))$ - **суперпозиция**, для вычисления $\Phi(x)$ используются алгоритмы вычисления более простых функций $y = f(x)$ и $g(y)$;

3) **Итерация (рекурсия)** $\Phi(0) = C$ - базис, $\Phi(x+1) = f(x,\Phi(x))$ – рекуррентное (возвратное) соотношение, шаг индукции.

Утверждение. Всякая рекурсивная функция эффективно вычислима – существует метод, который по рекурсивному описанию строит алгоритм вычисления этой функции.

Эффективное вычисление рекурсивных функций

Применение рекурсии и формирование трассы вычислений Ф(N), Ф(N-1), …, Ф(0).

Возврат-вычисление функций по трассе проводится:

- если трасса известна, то строится циклическая программа;
- прямое вычисление по формуле Ф(N), если она известна.

Примеры построения сложных рекурсивных функций на основе элементарных и рекурсии:

1. **Сумма** - вычисляется Ф(x, C) = x+C для заданного x

   Ф(0, C) = C базис,

   итерация Ф(x+1, C) = Ф(x, C) +1.

2. **Произведение** - вычисляется $\Phi(x, C) = x * C$ для заданного множителя x

$\Phi(0, C) = 0$ базис

итерация $\Phi(x+1, C) = \Phi(x, C) + C$.

3. **Факториал** - вычисляется $\Phi(x) = x!$

$\Phi(0) = 1$

итерация $\Phi(x+1) = (x+1)*\Phi(x)$.

4. **Вычисляется** $\Phi(x, C) = x!C^{x+1}$

$\Phi(0, C)=C$

итерация $\Phi(x+1, C) = (x+1)*\Phi(x, C)*C$.

5. **Вычисляется** $\Phi(x, C) = C^x$

$\Phi(0, C) =1$

итерация $\Phi(x+1, C) = \Phi(x, C)*C$.

В теории алгоритмов показано, что все эффективно вычис-лимые рекурсивные функции могут быть вычислены машинами Тьюринга.

На практике рекурсивное описание задач является тру-доемким и интуитивным, вместе с тем уровень сложности решаемых задач практически доступен, применяется в высокоуровневых алгоритмических языках и компили-руется в эффективные вычислительные процедуры.

Примеры рекурсивных вычислений

1. Факториал:

Ф(0) =1

Ф(x+1) = (x+1)*Ф(x).

Формируется и сохраняется трасса

Ф(10) =10* Ф(9) = 10*(9* Ф(8)) =10*(9*(8* ……1* Ф(0)

Затем последовательно вычисляется факториал.

Простая циклическая программа в Си

```
for(int i=1 ; S=1; i<11; i++)
    S=S*i;
```

2. ***Ряд чисел Фибоначчи*** $F_1, F_2, \ldots, F_n$:

Трасса строится по рекуррентной формуле

$F_n = F_{n-1} + F_{n-2}$, $n \geq 2$;

Общая формула Бине

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}, \quad \text{где } \varphi = \frac{1+\sqrt{5}}{2}.$$

3. ***Арифметическая и геометрическая прогрессии:***

трассы членов ряда $A_n, A_{n-1}, \ldots A_1, A_0$ и рекуррентные формулы:

возрастающей арифметической прогрессии $A_n = A_{n-1} + d$;

возрастающей геометрической прогрессии $A_n = A_{n-1} * d$;

формулы для общего члена ряда:

арифметической прогрессии $A_n = A_0 + d(n-1)$;

геометрической прогрессии $A_n = A_0 d^{n-1}$.

4. *Степенные ряды и полиномы*.

Широко используются в виде производящих функций, в приближениях функций рядами Тейлора, в позиционных системах счисления.

Вычисления рядов во многих случаев можно представить рекуррентными формулами и привести их к итерацион-ным алгоритмам.

Пример.

Десятичное число $a_{m-1}a_{m-2}..a_1a_0$ представленное в полиномиальной форме обозначает количество N

$$N = \sum_{i=0}^{m-1} a_i d^i = a_{m-1}10^{m-1} + a_{m-2}10^{m-2} + \ldots + a_1 10 + a_0 =$$

$$= (\ldots((0 + a_{m-1})10 + a_{m-2})10 + \ldots + a_1)10 + a_0.$$

КЛАССЫ СЛОЖНОСТИ

В рамках классической теории алгоритмические задачи различаются по классам сложности (P-сложные, NP-сложные, экспоненциально сложные и др.).

Классы сложности - множества вычислительных задач, примерно одинаковых по сложности вычисления. Более узко, классы сложности — это множества предикатов (функций, получающих на вход слово и возвращающих ответ 0 или 1), использующих для вычисления примерно одинаковые количества ресурсов. Каждый класс сложности (в узком смысле) определяется как множество предикатов, обладающих некоторыми свойствами.

Классом сложности K называется множество предикатов P(x), вычислимых на машинах Тьюринга и использующих для вычисления O(f(n)) ресурса, где n — длина слов

В качестве ресурсов обычно берутся время вычисления (количество рабочих тактов машины Тьюринга) или рабочая зона (количество использованных ячеек на ленте во время работы).

Класс P – задачи, которые могут быть решены за время, полиномиально зависящее от объёма исходных данных, с помощью детерминированной вычислительной машины (например, машины Тьюринга).

Класс NP — задачи, которые могут быть решены за полиномиально выраженное время с помощью недетерминированной вычислительной машины, то есть машины, следующее состояние которой не всегда однозначно определяется предыдущими.

К классу NP относятся задачи, решение которых с помощью дополнительной информации полиномиальной длины, данной нам свыше, мы можем проверить за поли-номиальное время. В частности, к классу NP относятся все задачи, решение которых можно проверить за полино-миальное время. Класс P содержится в классе NP. Классическими примерами NP-задач являются задачи о коммивояжёре, нахождение гамильтонова цикла, раскраска вершин графа.

Работу такой машины можно представить как разветвля-ющийся на каждой неоднозначности процесс: задача считается решённой, если хотя бы одна ветвь процесса пришла к ответу.

Поскольку класс P содержится в классе NP, принадлеж-ность той или иной задачи к классу NP зачастую отражает наше текущее представление о способах решения данной задачи и носит неокончательный характер. В общем слу-чае нет оснований полагать, что для той или иной NP-за-дачи не может быть найдено P-решение.

Вопрос о возможной эквивалентности классов P и NP (то есть о возможности нахождения P-решения для любой NP-задачи) считается многими одним из основных вопросов современной теории сложности алгоритмов. Ответа на этот вопрос нет. Сама постановка вопроса об эквивалентности классов P и NP возможна благодаря введению понятия NP-полных задач. NP-полные задачи отличаются тем свойством, что все NP-задачи могут быть тем или иным способом сведены к ним.

Наиболее часто встречающиеся классы сложности в зависимости от числа входных данных n таковы:
O(1) - количество шагов алгоритма не зависит от количес-тва входных данных. Обычно это алгоритмы, использую-щие определённую часть данных входного потока и игно-рирующие все остальные данные. Например, чистка 1 квадратного метра ковра вне зависимости от его размеров.
Ряд алгоритмов имеют порядок, включающий $\log_2 n$, и называются логарифмическими (logarithmic). Эта слож-ность возникает, когда алгоритм неоднократно подразде-ляет данные на подсписки, длиной 1/2, 1/4, 1/8, и так далее от оригинального размера списка. Логарифмические порядки возникают при работе с бинар-ными деревьями. Бинарный поиск имеет сложность сред-него и наихудшего случаев $O(\log_2 n)$.

Сложность $O(n\log_2 n)$ имеют алгоритмы быстрой сортиров-ки, сортировки слиянием и "кучной" сортировки, алго-ритм Краскала - построение минимального связываю-щего дерева, n - число ребер графа.

Алгоритм со сложностью $O(n)$ - алгоритм линейной слож-ности. Например, просмотр обложки каждой поступаю-щей книги - то есть для каждого входного объекта выпол-няется только одно действие;

Алгоритмы, имеющие порядок $O(n^2)$, являются квадратич-ными. К ним относятся наиболее простые алгоритмы сор-тировки; алгоритм Дейкстры - нахождение кратчайших пу-тей в графе, n – число вершин графа; алгоритм Прима - построение минимального связывающего дерева, n – чис-ло вершин графа. Всякий раз, когда n удваивается, время выполнения такого алгоритма увеличивается на

Алгоритм показывает кубическое время, если его поря-док равен $O(n^3)$, и такие алгоритмы очень медленные. Всякий раз, когда n удваивается, время выполнения алгоритма увеличивается в восемь раз. Алгоритм Флойда –Уоршелла (динамический алгоритм для нахождения кратчайших расстояний между всеми вершинами взве-шенного ориентированного графа)

Алгоритм с временной сложностью $O(2^n)$ имеет экспоненциальную сложность. Такие алгоритмы выполняются настолько мед-ленно, что они используются только при малых значениях n. Этот тип сложности часто ассоциируется с проблемами, требующими неоднократного поиска дерева решений.

Алгоритмы со сложностью $O(n!)$ – факториальные алгорит-мы, в основном, используются в комбинаторике для опре-деления числа сочетаний, перестановок.

В таблице сравниваются значения n^2 и $n\log_2 n$.

n	n^2	$n \log_2 n$
5	25	11,6
10	100	33,2
100	10000	664,3
1000	1000000	9965,7
10000	100000000	132877,1

Заметьте, насколько более эффективным является алго-ритм сортировки $O(n\log_2 n)$, чем обменная сортировка. Например, в случае со списком из 10 000 элементов ко-личество сравнений для обменной сортировки ограничи-вается величиной 100 000 000, тогда как более эффектив-ный алгоритм имеет количество сравнений, ограниченное величиной 132

В следующей таблице приводятся линейный, квадратич-ный, кубический, экспоненциальный и логарифмический порядки величины для выбранных значений n.

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	4294967296
128	7	896	16384	2097152	3.4×10^{38}
1024	10	10240	1048576	1073741824	1.8×10^{308}
65536	16	1048576	4294967296	2.8×10^{14}	Избегайте!

Из таблицы очевидно, что следует избегать использования кубических и экспоненциальных алгоритмов, если только значение n не мало.

Важность проведения резкой границы между полиномиальными и экспоненциальными алгоритмами вытекает из сопоставления числовых примеров роста допустимого размера задачи с увеличением быстродействия Б исполь-зуемых ЭВМ (в табл. указаны размеры задач, решаемых за одно и то же время *T* на ЭВМ с быстродействием $Б_1$ при различных зависимостях сложности *Q* от размера *n*)

$Q(n)$	$Б_1$	$Б_2 = 10 Б_1$	$Б_3 = 1000 Б_1$
n	n_1	$100n_1$	$1000n_1$
n^2	n_2	$10n_2$	$31{,}6n_2$
n^3	n_3	$4{,}64n_3$	$10n_3$
2^n	n_4	$6{,}64 + n_4$	$9{,}97 + n_4$

Эти примеры показывают, что, выбирая ЭВМ в *K* раз более быстро-действующую, получаем увеличение размера решаемых задач при линейных алгоритмах в *K* раз, при квадратичных алгоритмах в $K^{1/2}$ раз и т. д.

Иначе обстоит дело с неэффективными алгоритмами. Так, в случае сложности 2^n для одного и того же процессорного времени размер задачи увеличивается только на *lgK / lg2* единиц. Следовательно, переходя от ЭВМ с $Б_1$ = 1 Гфлопс к суперЭВМ с $Б_3$ = 1 Тфлопс, можно увеличить размер реша-емой задачи только на 10.

Исследования сложности алгоритмов позволили по-ново-му взглянуть на решение многих классических математи-ческих задач и найти для ряда таких задач (умножение многочленов и матриц, решение линейных систем урав-нений и др.) решения, требующие меньше ресурсов, нежели традиционные.