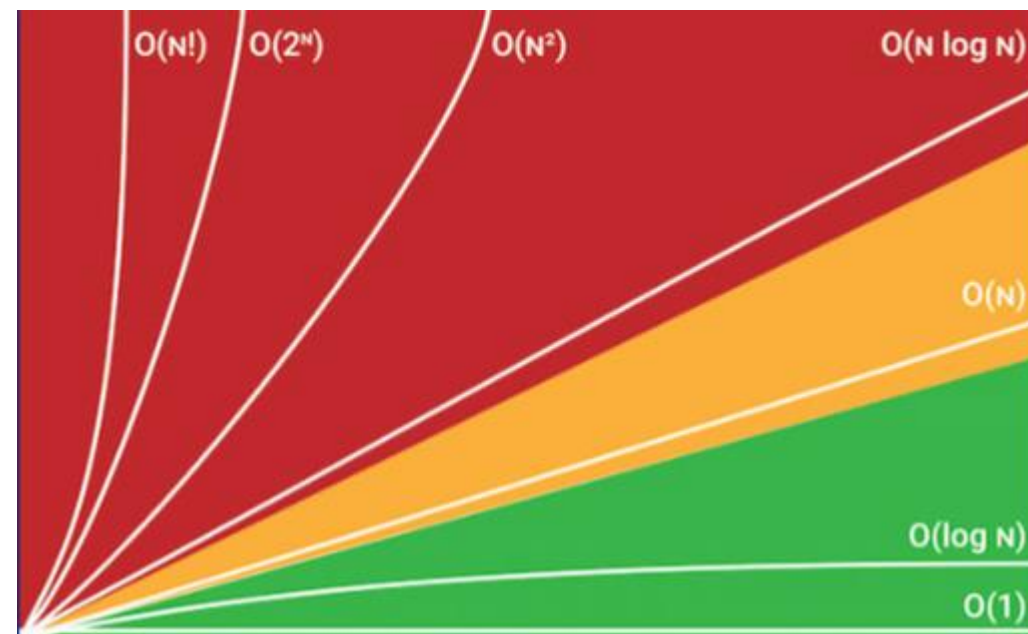


# Анализ сложности алгоритмов



## Алгоритм

это конечная последовательность чётко определенных, реализуемых компьютером инструкций, предназначенная для решения определенного класса задач.

Начиная с начального состояния и начального ввода (возможно, пустого), инструкции описывают вычисление, которое при выполнении проходит через конечное число чётко определённых последовательных состояний, в конечном итоге производя вывод и завершаясь в конечном состоянии.

Переход от одного состояния к другому не обязательно детерминирован; некоторые алгоритмы рандомизированы.

## Детерминированный алгоритм

Для одних и тех же входных данных все запуски алгоритма одинаковы по поведению.

## Рандомизированный алгоритм

Предполагает в своей работе некоторый случайный выбор и время работы рандомизированного алгоритма зависит от этого выбора.

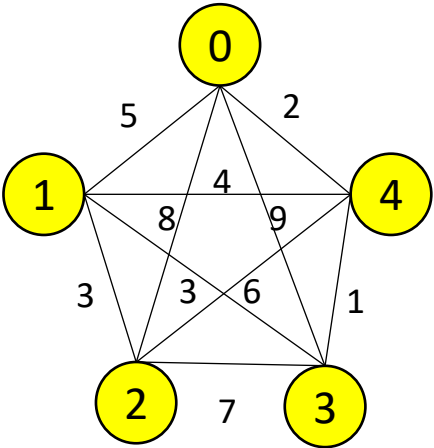
В рамках нашей дисциплины мы будем работать с **детерминированными алгоритмами**.

Все вычислительные задачи, которые мы будем решать в рамках нашей учебной дисциплины, разрешимы – существует алгоритм, который решает любой частный случай этой задачи.

Однако, этого не всегда достаточно, поскольку алгоритму может потребоваться так много времени, что он становится абсолютно бесполезным для практического применения.

# Задача коммивояжера

задан полный граф на  $n$  вершинах и матрица расстояний между всеми парами городов, необходимо найти замкнутый маршрут минимальной стоимости, проходящий через каждый город ровно один раз.



Задача разрешима, так как любую индивидуальную задачу можно решить, найдя наилучший обход среди конечного множества обходов. Если алгоритм будет перебирать все обходы, вычислять их длины и выбирать кратчайший обход, то реализация этого алгоритма на вычислительной машине потребует порядка  $\frac{(n-1)!}{2}$  шагов (элементарных команд) и уже для решения задачи среднего размера, например,  $n = 50$  (кратчайший обход столиц штатов США) потребовалось бы многих миллиардов лет при самых оптимистических прогнозах относительно скорости вычислительных машин в будущем, так как  $49! = 6.082818640342675e + 62$  имеет около 63 десятичных знаков.

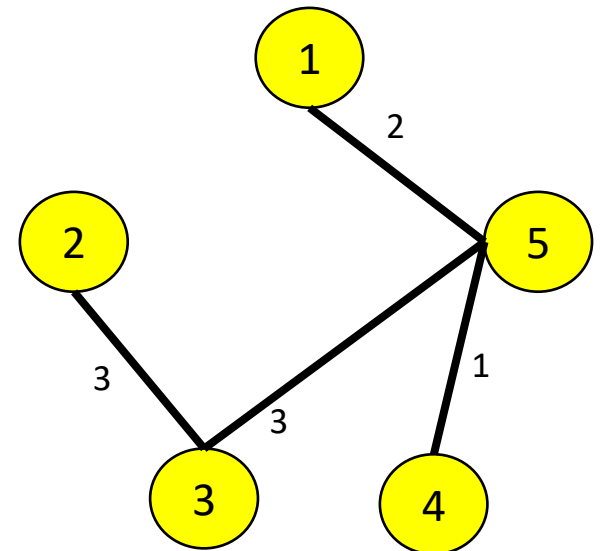
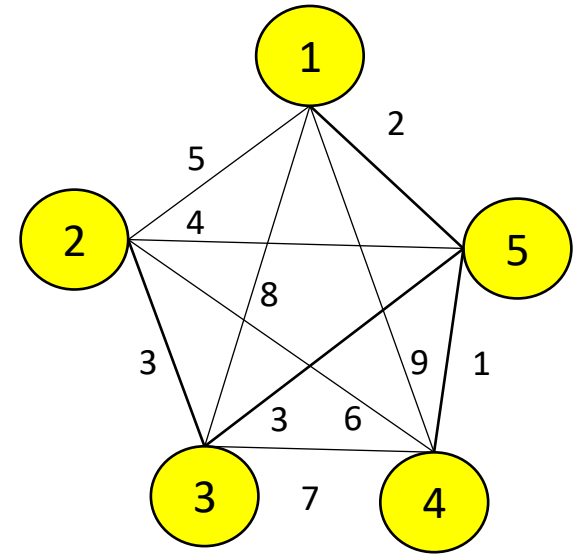
1!	1
2!	2
3!	6
4!	24
5!	120
6!	720
7!	5040
8!	40320
9!	362880
10!	3628800
11!	39916800
12!	479001600
13!	6227020800
14!	87178291200
15!	1307674368000
16!	20922789888000
17!	355687428096000
18!	6402373705728000
19!	121645100408832000
20!	2432902008176640000
21!	$5,10909 \cdot 10^{19}$
22!	$1,124 \cdot 10^{21}$
23!	$2,5852 \cdot 10^{22}$
24!	$6,20448 \cdot 10^{23}$
25!	$1,55112 \cdot 10^{25}$
26!	$4,03291 \cdot 10^{26}$
27!	$1,08889 \cdot 10^{28}$
28!	$3,04888 \cdot 10^{29}$
29!	$8,84176 \cdot 10^{30}$
30!	$2,65253 \cdot 10^{32}$

# Минимальное остовное дерево

заданы натуральное число  $n$  и матрица расстояний  $[d_{i,j}]$ , где  $d_{i,j} \in \mathbb{Z}^+$  между парами вершин, необходимо найти остовное дерево на  $n$  вершинах (неориентированный, связный, ациклический граф), имеющее наименьшую суммарную длину ребер. Задача разрешима: можно перебрать все остовные деревья с  $n$  вершинами и выбрать наилучшее из них.

Поскольку имеется  $n^{n-2}$  остовных деревьев с  $n$  вершинами, то время, необходимое для работы такого алгоритма перебора, снова неприемлемо: если  $n = 50$ , то  $50^{48} = 3,5527136788005007e+81$ .

Для этой задачи существует существенно лучший алгоритм, время работы которого пропорционально  $n^2$ .



# На практике при решении задач в iRunner

< 1. Рекуррентные соотношения (37 из 102) >

### Задача 32.2. Черепашка

Название

Свойства

Доступ

Папки

Условие

Разбор

Тесты

✓ Контролёр тестов

Решения

Взломы

Перетестирования

Файлы

> Редактор TeX

> Картинки

Отправить решение

Версия для печати

### 6 Задача 32.2. Черепашка

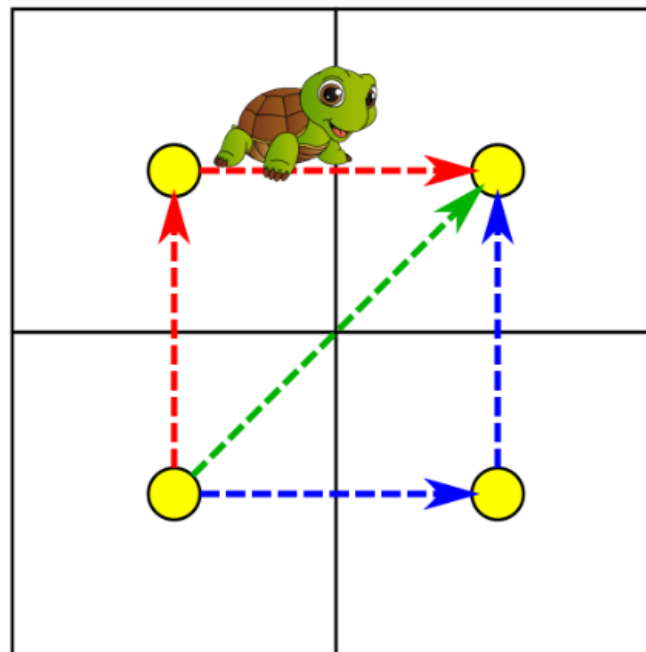
Имя входного файла: стандартный ввод

Имя выходного файла: стандартный вывод

Ограничение по времени: 1 с

Ограничение по памяти: 64 МБ

Черепашка гуляет по прямоугольному полю. Сейчас она находится в левом нижнем углу, но мечтает попасть в правый верхний угол. Черепашка очень любознательна, но слаба в точных науках, поэтому просит вас посчитать, сколько способов существует попасть в столь желанный ею угол, если она может перемещаться только вверх, вправо и вправо вверх по диагонали.



#### Формат входных данных

На входе заданы числа  $n$  строк и  $m$  столбцов в поле, разделенные пробелом ( $0 \leq n, m \leq 10\,000$ ).

#### Формат выходных данных

Выведите число способов добраться из квадрата с координатами  $(1, 1)$  до квадрата с координатами  $(n, m)$  по модулю  $1\,000\,000\,007$ .

#### Пример

стандартный ввод	стандартный вывод
2 2	3
3 2	5

#### Замечание

Картинка иллюстрирует способы для первого теста.



Программы, написанные на языках высокого уровня, нужно переводить в машинный код. Это можно делать по-разному.

**C++** уже на этапе компиляции переводит инструкции программы в хорошо оптимизированный машинный код.

**Python** выполняет преобразования в машинный код на этапе выполнения со значительными накладными расходами.

# Реальному ЦП требуется разное количество времени для выполнения различных операций.

Например, время выполнения операций на процессоре Intel Core десятого поколения (Ice Lake):

add, sub, and, or, xor, shl, shr...: 1 такт

mul, imul: 3–4 такта

div, idiv (32-битный делитель): 12 тактов

div, idiv (64-битный делитель): 15 тактов

Подробнее о времени выполнения операций: [https://www.agner.org/optimize/instruction\\_tables.pdf](https://www.agner.org/optimize/instruction_tables.pdf)

Даже имея готовый ассемблерный код реализации алгоритма, не представляется возможным узнать, какое время потребуется для его выполнения. Для этого необходимо было бы учесть, в частности,

- **Кеширование данных.** Процессоры имеют многоуровневую систему кешей (L1, L2, L3), постоянно сохраняющую те или иные ячейки для более быстрого доступа. В зависимости от того, закешировал ли процессор нужную ячейку, время доступа к данным может отличаться в десятки раз.
- **Out-of-order execution.** Процессор способен выполнять несколько не зависящих друг от друга команд одновременно (например, последовательные `mov eax, ecx` и `add edx, 5`). Процессор просматривает программу на сотни инструкций вперёд, выискивая те, что может выполнить без очереди.
- **Branch prediction и Speculative execution.** Большое препятствие для выполнения инструкций наперёд — ветвления (в частности, if'ы). Процессор не может заранее знать, в какую ветвь алгоритма ему придётся войти, и какой код ему нужно выполнять наперёд. Branch prediction модули следят за ходом выполнения программы и пытаются предсказать направления ветвлений (угадывают в >90% случаев). Штраф за ошибочное предсказание — потеря времени из-за избавления от десятков заранее подготовленных результатов операций и начала вычислений заново.



## Грубо говоря ....

Если вы пишете на C ++ и решаете типичную алгоритмическую задачу, то можете предположить, что за 1 секунду вы сможете выполнить  $\sim 10^8$  абстрактных операций.

Если вы делаете много делений, если вы обращаетесь к большому количеству памяти в случайном порядке, то вы сможете сделать за 1 секунду гораздо меньше,  $\sim 10^7$ .


Если операции простые, а обращения к памяти локальные или последовательные, то вы сможете за 1 секунду выполнить  $\sim 10^9$  операций.

# Анализ эффективности алгоритмов

1. Вы получили индивидуальную задачу.
2. Разработали алгоритм её решения и обосновали его корректность.
3. Программа, реализующая разработанный вами алгоритм, успешно прошла все тесты в iRunner.
4. Вы даже выяснили у преподавателя, что по всем тестам в iRunner ваша программа отработала быстрее, чем у других участников.



Итог	Принято
Очки	63 из 63
Автор	Даниил Матус
Задача	0.3. Является ли бинарное дерево поисковым? (🔗)
Ввод/вывод	bst.in/bst.out
Курс	2-й курс 3-я группа АИСД 2022–2023
Попытки (11)    Запуски (1)	

№	Вердикт	Очки	Время	Память	Сообщение программы проверк
1	OK	1 из 1	0.0 c	892 КБ	single line: 'NO'
2	OK	1 из 1	0.0 c	900 КБ	single line: 'YES'
3	OK	1 из 1	0.0 c	888 КБ	single line: 'YES'
4	OK	1 из 1	0.0 c	892 КБ	single line: 'NO'
5	OK	1 из 1	0.0 c	892 КБ	single line: 'NO'
6	OK	1 из 1	0.0 c	888 КБ	single line: 'YES'
7	OK	1 из 1	0.0 c	896 КБ	single line: 'NO'
8	OK	1 из 1	0.0 c	896 КБ	single line: 'NO'
9	OK	1 из 1	0.0 c	888 КБ	single line: 'YES'
10	OK	1 из 1	0.0 c	888 КБ	single line: 'YES'
11	OK	1 из 1	0.0 c	888 КБ	single line: 'NO'
12	OK	1 из 1	0.0 c	892 КБ	single line: 'NO'
13	OK	1 из 1	0.0 c	888 КБ	single line: 'NO'
14	OK	1 из 1	0.0 c	892 КБ	single line: 'NO'
15	OK	1 из 1	0.0 c	888 КБ	single line: 'NO'
16	OK	1 из 1	0.0 c	892 КБ	single line: 'YES'
17	OK	1 из 1	0.0 c	896 КБ	single line: 'NO'
18	OK	1 из 1	0.0 c	888 КБ	single line: 'NO'
19	OK	1 из 1	0.0 c	896 КБ	single line: 'NO'
20	OK	1 из 1	0.0 c	900 КБ	single line: 'YES'
21	OK	1 из 1	0.0 c	892 КБ	single line: 'NO'
22	OK	1 из 1	0.0 c	892 КБ	single line: 'NO'

Итог Принято  
Очки 63 из 63  
Автор Даниил Матус  
Задача 0.3. Является ли бинарное дерево поисковым?   
Ввод/вывод bst.in/bst.out  
Курс 2-й курс 3-я группа АиСД 2022–2023

Попытки (11)    Запуски (1)

№	Вердикт	Очки	Время	Память	Сообщение программы проверк
1 ▶	OK	1 из 1	0,0 с	892 КБ	single line: 'NO'
2 ▶	OK	1 из 1	0,0 с	900 КБ	single line: 'YES'
3 ▶	OK	1 из 1	0,0 с	888 КБ	single line: 'YES'
4 ▶	OK	1 из 1	0,0 с	892 КБ	single line: 'NO'
5 ▶	OK	1 из 1	0,0 с	892 КБ	single line: 'NO'
6 ▶	OK	1 из 1	0,0 с	888 КБ	single line: 'YES'
7 ▶	OK	1 из 1	0,0 с	896 КБ	single line: 'NO'
8 ▶	OK	1 из 1	0,0 с	896 КБ	single line: 'NO'
9 ▶	OK	1 из 1	0,0 с	888 КБ	single line: 'YES'
10 ▶	OK	1 из 1	0,0 с	888 КБ	single line: 'YES'
11 ▶	OK	1 из 1	0,0 с	888 КБ	single line: 'NO'
12 ▶	OK	1 из 1	0,0 с	892 КБ	single line: 'NO'
13 ▶	OK	1 из 1	0,0 с	888 КБ	single line: 'NO'
14 ▶	OK	1 из 1	0,0 с	892 КБ	single line: 'NO'
15 ▶	OK	1 из 1	0,0 с	888 КБ	single line: 'NO'
16 ▶	OK	1 из 1	0,0 с	892 КБ	single line: 'YES'
17 ▶	OK	1 из 1	0,0 с	896 КБ	single line: 'NO'
18 ▶	OK	1 из 1	0,0 с	888 КБ	single line: 'NO'
19 ▶	OK	1 из 1	0,0 с	896 КБ	single line: 'NO'
20 ▶	OK	1 из 1	0,0 с	900 КБ	single line: 'YES'
21 ▶	OK	1 из 1	0,0 с	892 КБ	single line: 'NO'
22 ▶	OK	1 из 1	0,0 с	892 КБ	single line: 'NO'

Итог Принято  
Очки 63 из 63  
Автор   
Задача 0.3. Является ли бинарное дерево поисковым?   
Ввод/вывод bst.in/bst.out  
Курс 2-й курс 4-я группа АиСД 2022–2023

Попытки (7)    Запуски (1)    Исх

№	Вердикт	Очки	Время	Память	Сообщение программы проверки
1 ▶	OK	1 из 1	0,015 с	748 КБ	single line: 'NO'
2 ▶	OK	1 из 1	0,0 с	748 КБ	single line: 'YES'
3 ▶	OK	1 из 1	0,015 с	752 КБ	single line: 'YES'
4 ▶	OK	1 из 1	0,015 с	748 КБ	single line: 'NO'
5 ▶	OK	1 из 1	0,0 с	748 КБ	single line: 'NO'
6 ▶	OK	1 из 1	0,0 с	752 КБ	single line: 'YES'
7 ▶	OK	1 из 1	0,0 с	748 КБ	single line: 'NO'
8 ▶	OK	1 из 1	0,015 с	748 КБ	single line: 'NO'
9 ▶	OK	1 из 1	0,0 с	748 КБ	single line: 'YES'
10 ▶	OK	1 из 1	0,0 с	752 КБ	single line: 'YES'
11 ▶	OK	1 из 1	0,0 с	752 КБ	single line: 'NO'
12 ▶	OK	1 из 1	0,015 с	748 КБ	single line: 'NO'
13 ▶	OK	1 из 1	0,0 с	748 КБ	single line: 'NO'
14 ▶	OK	1 из 1	0,0 с	748 КБ	single line: 'NO'
15 ▶	OK	1 из 1	0,0 с	756 КБ	single line: 'NO'
16 ▶	OK	1 из 1	0,015 с	748 КБ	single line: 'YES'
17 ▶	OK	1 из 1	0,0 с	752 КБ	single line: 'NO'
18 ▶	OK	1 из 1	0,0 с	748 КБ	single line: 'NO'
19 ▶	OK	1 из 1	0,0 с	744 КБ	single line: 'NO'
20 ▶	OK	1 из 1	0,0 с	748 КБ	single line: 'YES'
21 ▶	OK	1 из 1	0,0 с	748 КБ	single line: 'NO'

Можно ли утверждать, что вы разработали эффективный алгоритм решения вашей индивидуальной задачи?

	n=10	n=20	n=30
n	0,00001с	0,00002с	0,00003с
n <sup>2</sup>	0,0001с	0,0004с	0,0009с
n <sup>3</sup>	0,001с	0,008с	0,027с
<b>n<sup>5</sup></b>	0,1с	<b>3,2с</b>	24,3с
<b>2<sup>n</sup></b>	0,001с	<b>1с</b>	17,5 мин
3 <sup>n</sup>	0,059с	58 мин	6,5 лет



# **Теоретический анализ сложности алгоритма**

## Абстрактное вычислительное устройство

**РАМ** – одноадресная машину с произвольным доступом к памяти  
(англ. Random-Access Machine – **RAM**)

Предположим, что у нас есть некоторый алгоритм решения задачи и мы хотим на некотором устройстве реализовать его.

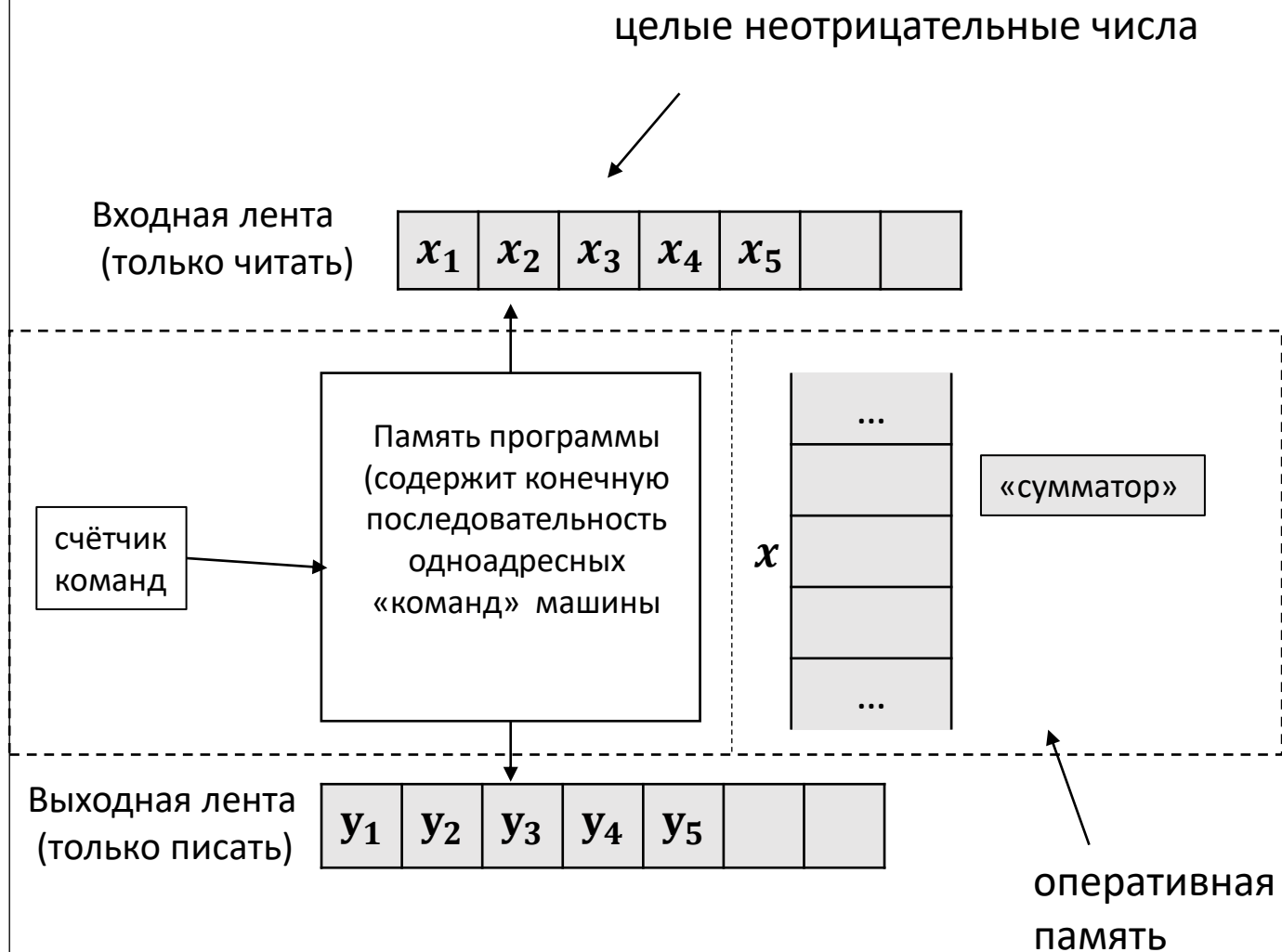
Как определить время выполнения и объём памяти программной реализации алгоритма на некотором **абстрактном вычислительном устройстве**?

В качестве модели такого вычислительного устройства возьмём **РАМ** – одноадресную машину с произвольным доступом к памяти (англ. Random-Access Machine – **RAM**).

**РАМ** - универсальная математическая модель вычислений, которая является хорошим приближением к классу обычных вычислительных машин;

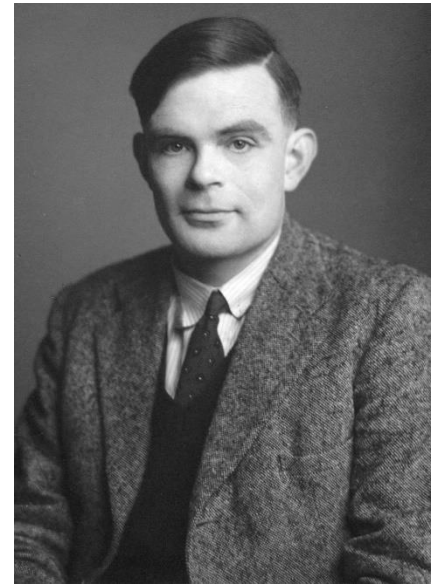
РАМ - вычислительная машина с одним сумматором, в котором команды программы не могут изменять сами себя, состоит из:

- ✓ **процессора с набором команд**, все команды – одноадресные;
- ✓ **входной ленты**, с которой она может считывать данные в соответствии с их упорядоченностью (последовательный доступ);
- ✓ **выходной ленты**, на которую она может записывать (в первую свободную клетку);
- ✓ **оперативной памяти**, которая состоит из потенциально бесконечного числа ячеек, в каждой из которых может быть записано произвольное целое число, индексы ячеек могут быть и отрицательными, все ячейки равнодоступны; **сумматор** – это специальный выделенная тип памяти, в котором выполняются все вычисления.



**РАМ** существенно отличается от другой известной абстрактной модели вычислений – **машины Тьюринга** (MT) (предложена в качестве абстрактной модели вычислений А. Тьюрингов в 1936 г.)

Для MT время перехода между ячейками ленты зависит от расстояния между ячейками, а одна ячейка ленты содержит один символ фиксированного конечного алфавита.



Алан Тьюринг

*Alan Mathison Turing*  
(1912-1954)

## Программа для РАМ

- ✓ конечная последовательность одноадресных команд, которая изменяет ячейки памяти (команды программы занумерованы числами 1,2, ...);
- ✓ есть счетчик команд – целое число;
- ✓ программа не записывается в память (т. е. не может менять саму себя);
- ✓ команды процессора выполняются последовательно, одновременно выполняемые команды отсутствуют (однопроцессорная машина);
- ✓ результат работы программы - набор чисел на выходной ленте.

Формально – любое число вмещается в любую ячейку.

# Набор команд для РАМ-машины

<b>READ <math>x</math></b>	<b>считывает</b> в ячейку $x$ очередной символ из входной ленты
<b>WRITE <math>x</math></b>	<b>запись</b> содержимого ячейки $x$ на выходную ленту
<b>LOAD <math>x</math></b>	<b>загрузить</b> операнд <b>из</b> ячейки $x$ памяти <b>в сумматор</b>
<b>STORE <math>x</math></b>	<b>отослать</b> содержимое <b>из сумматора</b> в указанную ячейку $x$ памяти
<b>ADD <math>x</math></b> <b>SUB <math>x</math></b>	<b>сложить</b> содержимое ячейки $x$ и <b>сумматора</b> <b>вычесть</b> содержимое ячейки $x$ из <b>сумматора</b>
<b>MUL <math>x</math></b> <b>DIV <math>x</math></b>	<b>умножить</b> содержимое <b>сумматора</b> и ячейки $x$ <b>разделить</b> число <b>из сумматора</b> на содержимое ячейки $x$
<b>JGTZ M</b> <b>JZERO M</b> <b>JUMP M</b> <b>HALT</b> <b>WAIT</b>	<b>переход</b> на команду с меткой <b>M</b> , если число в <b>сумматоре</b> <b>&gt;0</b> <b>переход</b> на команду с меткой <b>M</b> , если число в <b>сумматоре</b> <b>=0</b> <b>безусловный переход</b> на команду с меткой <b>M</b> <b>останов (завершение работы программы)</b> <b>ожидание</b>

Операнды могут быть одного из следующих типов:

$=x$  — число

$i$  — содержимое  $i$ -ой ячейки памяти

$*i$  — содержимое ячейки памяти, номер которой записан в ячейке  $i$  (косвенная адресация)

# Равномерная мера сложности.

Время выполнения каждой команды и размер каждой ячейки полагается равным единице.

Теперь

**время работы программы** = общему числу выполненных команд;  
**объем памяти** = числу ячеек, к которым было обращение.

входная лента

2 <sup>100</sup>	2 <sup>10</sup>					
------------------	-----------------	--	--	--	--	--

<b>READ 1</b>	<b>сумма двух чисел (равномерная мера сложности)</b>
<b>LOAD 1</b>	время работы программы = 6;
<b>READ 2</b>	объем памяти = 3;
<b>ADD 2</b>	
<b>STORE 3</b>	
<b>WRITE 3</b>	

Если не оговорено специально, то в дальнейшем будем использовать эту меру сложности.



# Логарифмическая мера сложности

Время выполнения каждой команды полагается по порядку равным логарифму величины максимального операнда, а общее время равно сумме времен выполнения всех выполненных команд.

Объем памяти равен сумме максимальных разрядностей числа в каждой используемой ячейке оперативной памяти плюс максимальная разрядность числа в сумматоре.

входная лента

$2^{100}$	$2^{10}$					
-----------	----------	--	--	--	--	--

READ 1

LOAD 1

READ 2

ADD 2

STORE 3

WRITE 3

**сумма двух чисел**

(логарифмическая мера сложности)

время выполнения =  $510? 516?$ ;

объем памяти =  $314?$   
(1яч.) 101 бит + (2яч.) 11 бит +  
(3яч.) 101 бит + (сумматор) 101 бит  
= 314

READ 1

LOAD 1

READ 2

MUL 2

STORE 3

WRITE 3

**произведение двух чисел**

(логарифмическая мера сложности)

время выполнения =  $530? 536?$ ;

объем памяти =  $334?$   
(1яч.) 101 бит + (2яч.) 11 бит +  
(3яч.) 111( бит ) + (сумматор) 111 бит =  
334

Объясните, почему такая схема корректно учитывает по длине в битах все операции для корректно завершенной программы?

Одна из характеристик для сравнения эффективности алгоритмов —

**исследование (математическими методами)  
поведения алгоритма при увеличении размера  
входных данных, так как именно эти входы  
определяют границы применимости алгоритма.**

Для характеристики «громоздкости данных» можно ввести некоторую **неотрицательную числовую характеристику возможных входов** алгоритма и оценивать трудоёмкость алгоритма с помощью функций, зависящих от этой числовой характеристики.

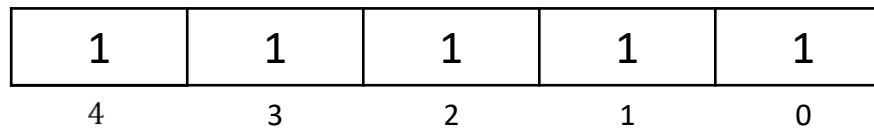
Эту числовую характеристику называют ***размерностью задачи*** (*размер входа*) и обозначают буквой  ***$l$*** .

**В качестве размерности задачи  $l$  принято брать число бит, необходимое для кодировки входных данных задачи.**

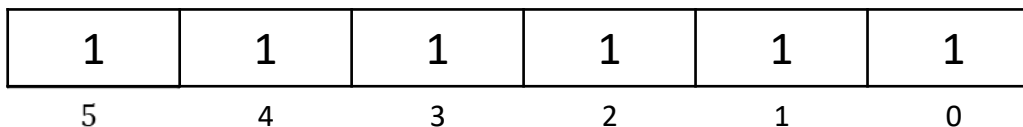
На вход поступает одно целое число  $n$ , какова битовая длина  $l$  этого числа?

**Унарная** система кодирования:

длина в битах равна самому числу



$n=5$  ( $l = 5$ )



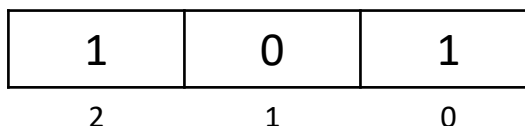
$n=6$  ( $l = 6$ )

Вход: 1 1 0 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1

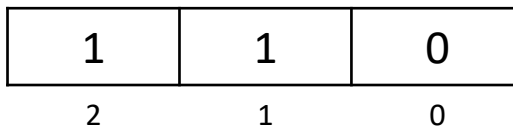
---

**Бинарная** система кодирования:

длина в битах равна количеству цифр в его двоичной записи



$n = 5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101_2$  ( $l = 3$ )

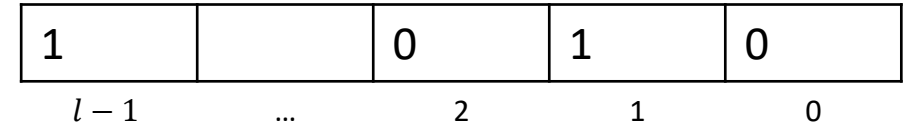


$n = 6 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 110_2$  ( $l = 3$ )

Будем рассматривать только **компактное представление числа**:

старший бит в двоичной записи числа равен 1, т.е. нет незначащих 0.

**На вход поступает одно целое число  $n$ .  
Какова битовая длина этого числа в  
бинарной системе кодирования?**



Предположим, что для компактного задания числа  $n$  в двоичной системе счисления  $l + 1$  бита много, а  $l$  – достаточно. Тогда справедливы неравенства:

$$2^{l-1} \leq n < 2^l.$$

Логарифмируем и, учитывая, что число бит является целым числом, получаем

$$\log_2 n < l \leq \log_2 n + 1$$

$$l = \lfloor \log_2 n + 1 \rfloor$$

Таким образом, если на вход поступает одно целое число  $n \geq 0$ , то битовая длина в бинарной системе кодирования:

$$l = \begin{cases} 1, & n = 0 \\ \lfloor \log_2 n + 1 \rfloor, & n > 0 \end{cases}$$

Сведения из математики:  
 $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$

На вход поступает натуральное число  $n$  ( $\mathbb{N} = \{1, 2, 3, \dots\}$ ) и массив чисел:  $a_1, a_2, \dots, a_n$ .  
Какова битовая длина этого входа в бинарной системе кодирования?

$$l = \lfloor \log_2 n + 1 \rfloor + \lfloor \log_2 a_1 + 1 \rfloor + \lfloor \log_2 a_2 + 1 \rfloor + \dots + \lfloor \log_2 a_n + 1 \rfloor$$

Если предположить, что  $\max_{1 \leq i \leq n} \lfloor \log_2 a_i + 1 \rfloor = \text{const}$  (например,  $\text{const} = 32$ ), то полагается, что

$$l = \lfloor \log_2 n + 1 \rfloor + \text{const} \cdot n$$

В данной модели для увеличения объема входных данных будем устремлять  $n \rightarrow +\infty$ , а входными данными задачи размерности  $l$  будут только массивы из  $n$  элементов.

- ✓ если на вход поступает массив чисел  $a_1, a_2, \dots, a_n$  и нам нужно их упорядочить или что-то найти в массиве, то в предположении, что  $\max_{1 \leq i \leq n} \lfloor \log_2 a_i + 1 \rfloor = \text{const}$ , в качестве такой числовой характеристики входных данных может выступать число  $n$  элементов массива, а такую меру измерения размерности задачи называют **равномерной**; при анализе поведения алгоритма с ростом объема входных данных будем устремлять к бесконечности число  $n$ ;
- ✓ в задачах на графы в качестве числовой характеристики входных данных задачи берут число вершин  $|V|$  и/или ребер  $|E|$  (рассуждения будут аналогичны, предыдущему случаю, так как, например,
  - если граф задан матрицей смежности, то элементы матрицы принимают всего два значения 1 или 0 и для кодирования достаточно константной памяти;
  - если граф задан списками смежности, то так как мы работаем с конечными графами, т.е.  $|V|$  – конечно, то для кодирования также достаточно константной памяти;
  - при анализе поведения алгоритма с ростом объема входных данных будем устремлять к бесконечности число рёбер  $|E|$ ;
- ✓ в арифметических задачах размером входа может быть максимум абсолютных величин входных чисел или количество цифр в его двоичной записи (бинарная система кодирования);
  - например, если на вход поступает целое число  $n$  и необходимо определить, является ли это число простым, то в качестве числовой характеристики входных данных  $l$  удобно брать число бит двоичной записи компактного представления числа  $n$ ; при анализе поведения алгоритма при увеличении объема входных данных будем устремлять к бесконечности битовую длину числа  $n$ ;

**Выбор, что взять в качестве числовой характеристики возможных входов алгоритма, делается в зависимости от характера задачи.**

## Временная сложность алгоритма

это функция, которая задаче размерности  $l$  ставит в соответствие время  $T(l)$ , затрачиваемое алгоритмом для её решения.

Если при данной размерности  $l$  в качестве меры сложности берётся наибольшая из сложностей (по всем входам этой размерности), то она называется **временной сложностью в худшем случае**.

Если при данной размерности  $l$  в качестве меры сложности берётся средняя сложность (по всем входам этой размерности), то она называется **средней сложностью**.

## Ёмкостная сложность алгоритма –

объем памяти, требуемый для реализации алгоритма, как функция от размерности задачи.

Поведение временной (ёмкостной) сложности при увеличении размерности задачи до бесконечности называется **асимптотической временной (ёмкостной) сложностью**.



**Задача.**

Для заданного числа  $n$  проверить, является ли число простым.

Решение.

Алгоритм: проверить, есть ли среди чисел  $2, 3, \dots, \lfloor \sqrt[n]{n} \rfloor$  хотя бы одно, делящее  $n$ .

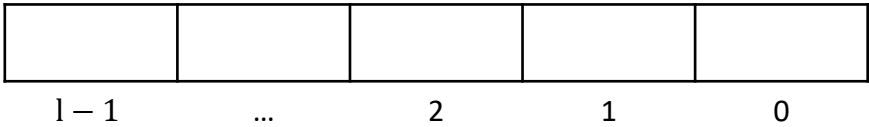
Оценку проведем по числу мультипликативных операций: деления.  
Если в качестве размерности задачи  $l$  взять само число  $n$  (унарная система кодирования входа), то не получится выписать точную формулу для временной сложности  $T(l)$ .

$n$	111	112	<b>113</b>	114	115	116	117	118	119	120
$T(n)$	<b>2</b> n:2 - нет n:3 - да	<b>1</b> n:2 - да	<b>9</b> n: 2 - нет n: 3 –нет ... n:10 - нет	<b>1</b> n: 2 - да	<b>4</b> n: 2 - нет n: 3 –нет n: 4 - нет n: 5 –да	<b>1</b> n: 2 - да	<b>2</b> n:2 - нет n:3 - да	<b>1</b> n: 2 - да	<b>6</b> n: 2 - нет n: 3 –нет n: 4 - нет n: 5 –нет n: 6 –нет n: 7 –да	<b>1</b> n: 2 - да

Можно лишь показать, что, для любых  $n \rightarrow +\infty, T(n) \leq \lfloor \sqrt[n]{n} \rfloor - 1$ .

(продолжение)

Предположим, что в задаче проверки числа  $n$  на простоту, в качестве размерности задачи взяли его битовую длину в двоичной записи:



Но одним и тем же размером в  $l$  бит обладают несколько входов задачи и временная сложность алгоритма для каждого из этих входов может быть разной:

$$2^{10\dots00} \leq n \leq 2^{11\dots11} - 1 < 2^l.$$

Для вычислительной сложности алгоритма в худшем случае нужно найти тот вход размера  $l$ , для которого у алгоритма будут самые большие затраты.

$l$ бит	2 --	3 ---	4 ----	5 -----	6 -----	7 -----
$n$ диапазон чисел размерности $l$	2 – 3	4 – 7	8 – 15	16 – 31	32 – 63	64 – 127
$\hat{n}$ число, для которого самые большие затраты	3	7	13	31	61	127
$T(l)$	1	1	2	4	6	10

Постулат Бертрана гласит, что при любом целом  $N > 1$  имеется простое число, принадлежащее интервалу  $(N, 2 \cdot N)$ .

Пусть  $p_l$  – самое большое простое число битовой длины  $l$ , тогда число выполненных делений при проверке этого числа известно и оно равно  $\lfloor \sqrt{p_l} \rfloor - 1$ .

Так как  $2^{l-1} \leq p_l < 2^l$ , то справедливо

$$\lfloor \sqrt{p_l} \rfloor - 1 \geq \lfloor 2^{\frac{l-1}{2}} \rfloor - 1,$$

$$T(l) \geq \lfloor \sqrt{p_l} \rfloor - 1 \geq \lfloor 2^{\frac{l-1}{2}} \rfloor - 1 > 2^{\frac{l-1}{2}} - 2 = 2^{\left(\frac{l}{2} - \frac{1}{2}\right)} - 2 = \frac{2^{\frac{l}{2}}}{\sqrt{2}} - 2 = \left(\frac{1}{\sqrt{2}} - \frac{2}{2^{l/2}}\right) \cdot 2^{l/2} = C_1 \cdot 2^{l/2}, \text{ где}$$

при  $4 \leq l \leq +\infty$ ,  $0 < \left(\frac{1}{\sqrt{2}} - \frac{1}{2}\right) \leq C_1 < \frac{1}{\sqrt{2}}$ .

В тоже время для всех входных данных задачи размерности  $l$  выполняется:

$$T(l) < \lfloor 2^{l/2} \rfloor - 1 \leq 2^{l/2} - 1 < 2^{l/2}.$$

### **Вывод:**

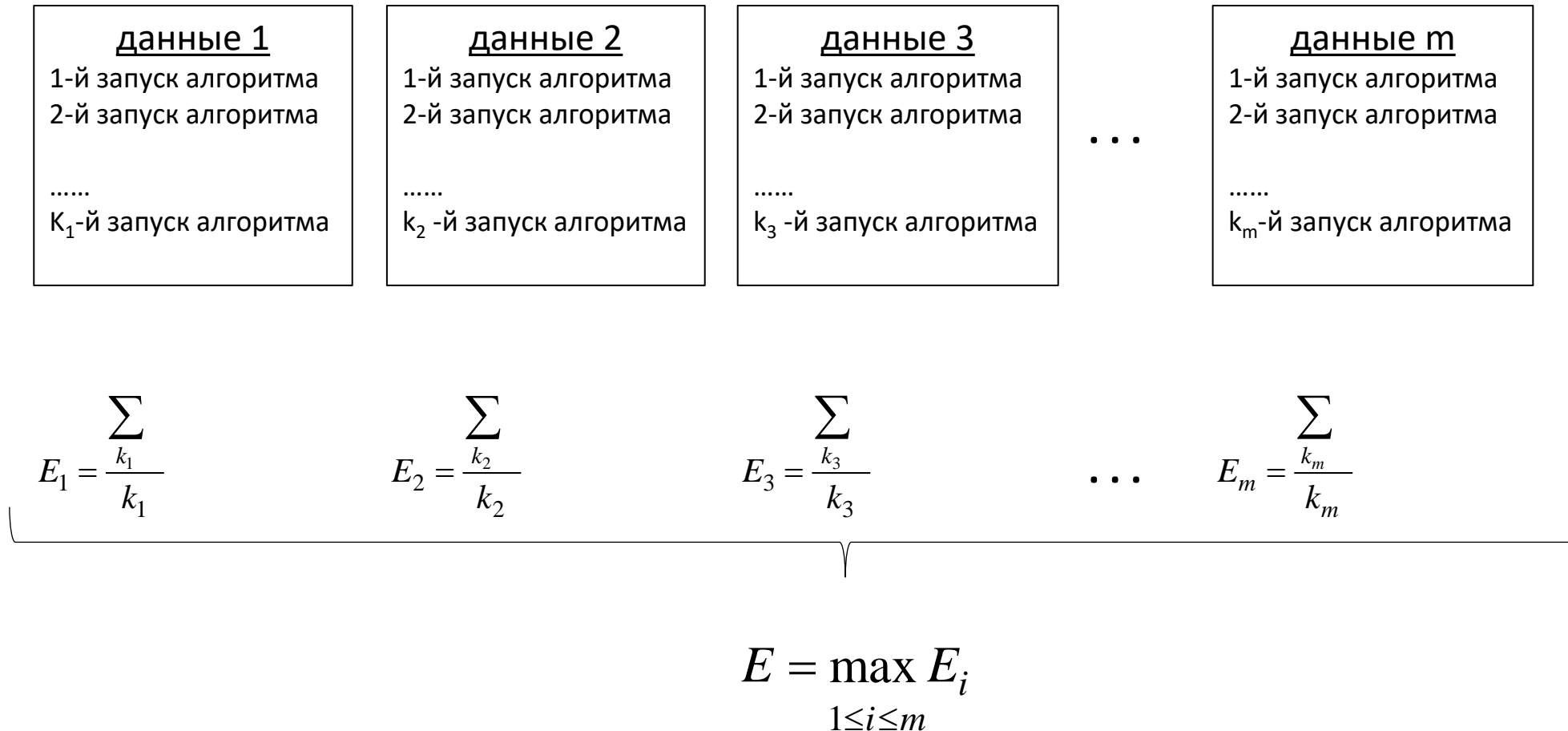
при бинарной системе кодирования входа удалось получить оценки сверху и снизу для временной сложности в худшем случае алгоритма проверки числа на простоту:

$$C_1 \cdot 2^{l/2} < T(l) < 2^{l/2}, \text{ при } l \geq 4.$$

В последующем, мы покажем, что этот результат записывается  $T(l) = \Theta(2^{l/2})$  и данный алгоритм при бинарной системе кодирования входных данных является экспоненциальным.

(для информации)

## Временная сложность в худшем случае рандомизированного алгоритма



## Среднее время работы детерминированного алгоритма по всем возможным наборам входных данных размерности $l$

- 1) Все входные данные размерности  $l$  разбиваем на группы так, чтобы время работы алгоритма для всех данных из одной группы было одним и тем же.

Предположим, что у нас  $m$  групп.

- 2) Пусть  $p_i$  – вероятность, с которой данные попадают в группу  $i$ .
- 3) Пусть  $t_i$  – время работы алгоритма для данных из группы  $i$ .

$$A(l) = \sum_{i=1}^m p_i \cdot t_i$$

*Сведения из теории вероятности*

Если у нас  $m$  групп и входные данные могут оказаться с равной вероятностью в любой из них, то

$$p_i = 1/m, \forall i = \overline{1, m}$$

В этом случае среднее время работы алгоритма по всем возможным наборам входных данных:

$$A(l) = \frac{\sum_{i=1}^m t_i}{m}$$

## Задача

Задан массив из  $n$  уникальных элементов и некоторое число  $x$ .

Необходимо определить есть ли число  $x$  в массиве.

Оценить **среднее время работы алгоритма** последовательного поиска по всем возможным наборам входных данных размерности  $l$ .

1-я группа: искомый элемент  $x$  стоит на 1-й позиции

$$t_1 = 1$$

2-я группа: искомый элемент  $x$  стоит на 2-й позиции

$$t_2 = 2$$

3-я группа: искомый элемент  $x$  стоит на 3-й позиции

$$t_3 = 3$$

...

$n$ -я группа: искомый элемент  $x$  стоит на  $n$ -й позиции

$$t_n = n$$

$(n + 1)$ -я группа: искомого элемента  $x$  нет

$$t_{n+1} = n$$

$$p_i = \frac{1}{(n + 1)}, \quad \forall i = \overline{1, n + 1}$$

$$\begin{aligned} A(l) &= \sum_{i=1}^{n+1} p_i \cdot t_i = \frac{1}{n+1} (1 + 2 + \dots + n + n) = \\ &= \frac{1}{n+1} \cdot \left( \frac{1+n}{2} \cdot n + n \right) = \frac{n}{2} + \frac{n}{n+1} = \frac{n}{2} + 1 - \frac{1}{n+1} \approx \frac{n}{2} + 1, \\ &\text{при } n \rightarrow +\infty \quad C_1 \cdot n \leq l \leq C_2 \cdot n \end{aligned}$$

Для функции  $T(l)$ , описывающей время работы алгоритма в худшем случае важна скорость роста функции  $T(l)$  при возрастании объема входных данных, т.е. оставляем только ту часть функции  $T(l)$ , которая с ростом аргумента к бесконечности, растёт быстрее всего (не оставляют и постоянный коэффициент у главного члена, так как с ростом аргумента к бесконечности он не вносит существенный вклад в значение функции).

Например, порядок функции

$$T(l) = 8 \cdot l^2 + l \cdot \log l + 2$$

есть  $l^2$ , в этом случае говорят, что функция  $T(l)$  растёт как  $l^2$ .

Математические обозначения (нотации) **O**, **Ω**, **Θ** введены для функций по аналогии с тем, как для чисел были введены обозначения ( $\lceil \cdot \rceil$  округление к большему/вверх/англ. *ceiling* досл. «потолок»;  $\lfloor \cdot \rfloor$  —округление к меньшему/вниз/ англ. *floor* досл. «пол»).

**Асимптотики  $O(f(n))$ ,  $\Omega(f(n))$ ,  $\Theta(f(n))$  –**

способ оценки функции  $g(n)$  при  $n \rightarrow +\infty$  сверху и снизу, чтобы упростить её внешний вид.

Произносится:

$g(n) \in O(f(n))$   $g$  от  $n$  равно **о большое** от  $f$  от  $n$

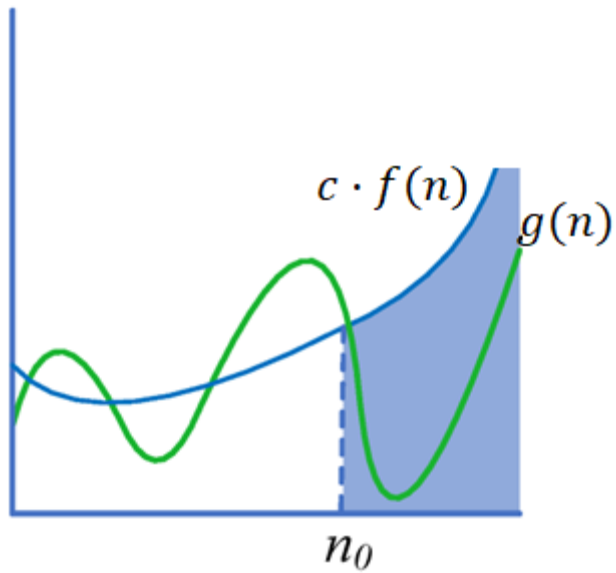
$g(n) \in \Omega(f(n))$   $g$  от  $n$  равно **омега большое** от  $f$  от  $n$

$g(n) \in \Theta(f(n))$   $g$  от  $n$  равно **тетта большое** от  $f$  от  $n$

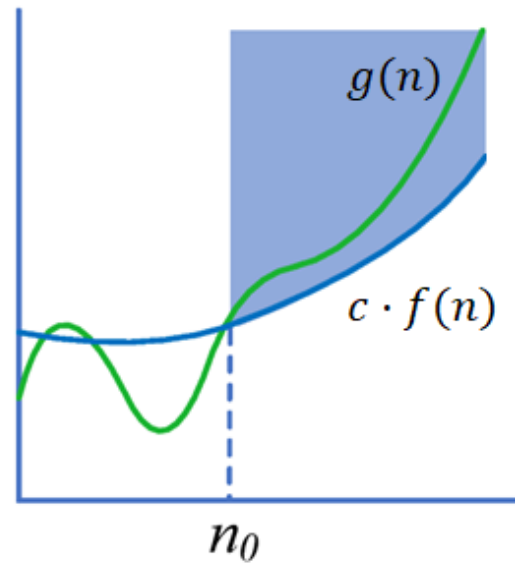
здесь и далее  $n \rightarrow +\infty$



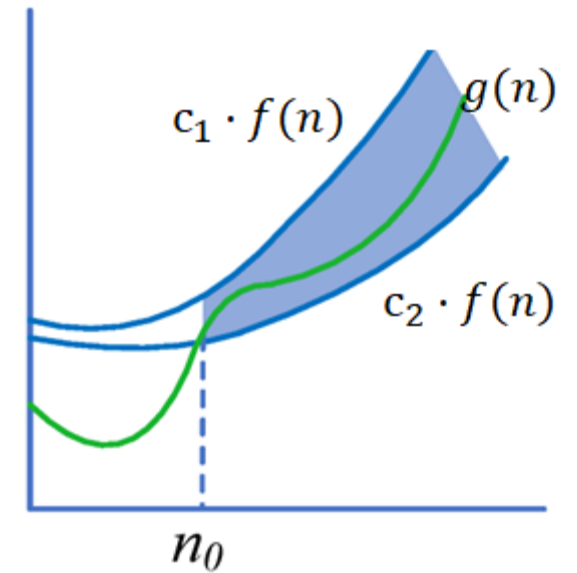
$O(f(n))$  – это множество функций, которые растут не быстрее, чем функция  $f(n)$  при достаточно больших  $n$ .



$\Omega(f(n))$  – это множество функций, которые растут, по крайней мере, так же быстро, что и функция  $f(n)$  при достаточно больших  $n$ .



$\Theta(f(n))$  – это множество функций, которые растут с той же скоростью, что и функция  $f(n)$  при достаточно больших  $n$ .



Каждый из рассматриваемых трёх классов:  $\mathbf{O}(f(n))$ ,  $\mathbf{\Omega}(f(n))$ ,  $\mathbf{\Theta}(f(n))$  – множество, поэтому правильнее писать

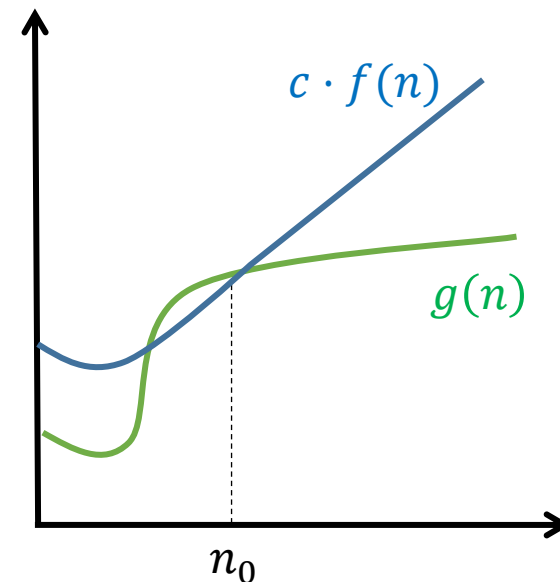
$$g(n) \in \mathbf{O}(f(n)),$$

но мы будем также употреблять эквивалентную запись:

$$g(n) = \mathbf{O}(f(n)).$$

Пусть  $f(n)$  и  $g(n)$  — функции, определённые на множестве целых положительных чисел и принимающие положительные действительные значения.

Будем писать  $g(n) \in O(f(n))$ , если существуют такие константы  $c > 0$ ,  $n_0 > 0$ , что  $\forall n \geq n_0$  выполняется  $0 \leq g(n) \leq c \cdot f(n)$ .



Говорят, что функция  $f(n)$  даёт **асимптотическую верхнюю границу** для функции  $g(n)$ .

Пусть  $g(n) = 4 \cdot n^3$ , тогда можно записать

$$4 \cdot n^3 \in O(2^n)$$

$$4 \cdot n^3 \in O(n^3 \cdot \log n)$$

$$4 \cdot n^3 \in O(n^3)$$

Будем писать  $g(n) \in o(f(n)) \Leftrightarrow \forall c, \exists n_0 > 0: \forall n \geq n_0$  выполняется  $0 \leq g(n) < c \cdot f(n)$ .

Пусть  $g(n) = O(f(n))$ , тогда  $f(n)$  является точной оценкой для функции  $g(n)$  только если  $g(n) \neq o(f(n))$ .

для функции  $g(n) = 4 \cdot n^3$  функция  $f(n) = n^3$  является точной оценкой:

$$4 \cdot n^3 = O(n^3)$$

$$4 \cdot n^3 \neq o(n^3)$$

Пусть  $f(n)$  и  $g(n)$  — функции, определённые на множестве целых положительных чисел и принимающие положительные действительные значения.

Будем писать  $g(n) \in \Omega(f(n))$ , если существуют такие константы  $c > 0$ ,  $n_0 > 0$ , что  $\forall n \geq n_0$  выполняется  $0 \leq c \cdot f(n) \leq g(n)$ .

Говорят, что функция  $f(n)$  даёт **асимптотическую нижнюю границу** для функции  $g(n)$ .

Пусть  $g(n) = 4 \cdot n^3$ , тогда можно записать

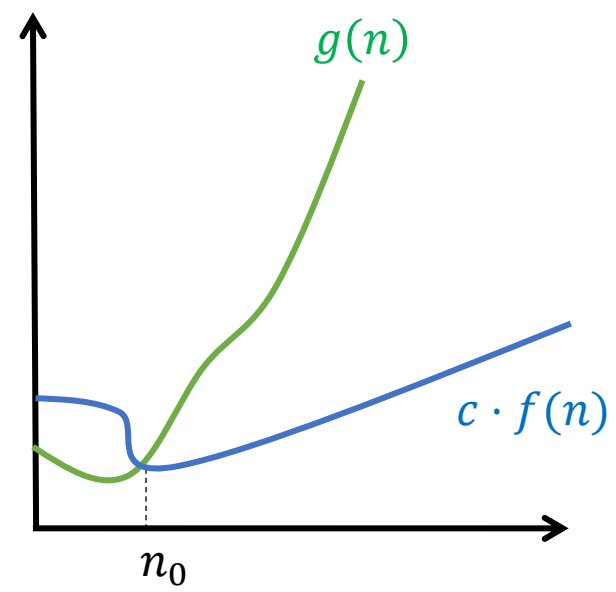
- $4 \cdot n^3 \in \Omega(n)$
- $4 \cdot n^3 \in \Omega(n^2 \cdot \log n)$
- $4 \cdot n^3 \in \Omega(n^3)$

Будем писать  $g(n) \in \omega(f(n)) \Leftrightarrow \forall c, \exists n_0 > 0: \forall n \geq n_0$  выполняется  $0 \leq c \cdot f(n) < g(n)$ .

Пусть  $g(n) \in \Omega(f(n))$ , тогда  $f(n)$  является точной оценкой для функции  $g(n)$  только если  $g(n) \notin \omega(f(n))$ .

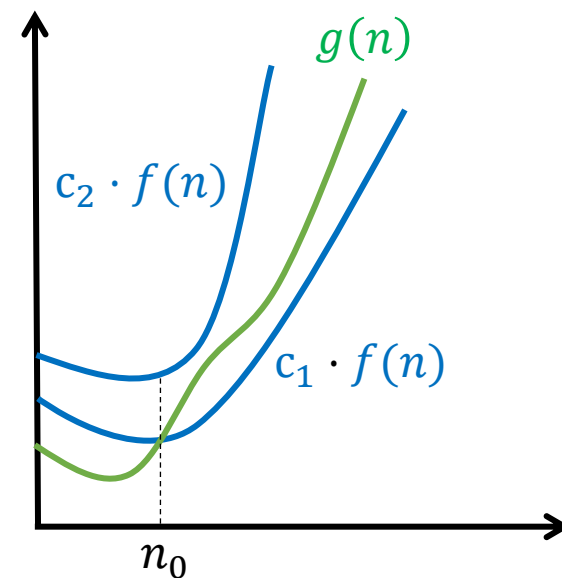
для функции  $g(n) = 4 \cdot n^3$  функция  $f(n) = n^3$  является точной оценкой:  $4 \cdot n^3 \in \Omega(n^3)$  и  $4n^3 \notin \omega(n^3)$ .

Если  $g(n) \in \Omega$ , то  $f(n) \in O(g(n))$ .



Пусть  $f(n)$  и  $g(n)$  — функции, определённые на множестве целых положительных чисел и принимающие положительные действительные значения.

Будем писать  $g(n) \in \Theta(f(n))$ , если существуют такие константы  $c_1 > 0, c_2 > 0, n_0 > 0$ , что  $\forall n \geq n_0$  выполняется

$$0 \leq c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$


Говорят, что функция  $f(n)$  является **асимптотически точной оценкой** для функции  $g(n)$  (про функции  $f(n)$  и  $g(n)$  говорят, что они имеют одинаковый порядок роста).

$$4n^3 \in \Theta(n^3)$$

$$2n^3 \in \Theta(n^3)$$

$$n^3 \in \Theta(n^3)$$

про функции, которые попадают в это множество  $\Theta(n^3)$ , говорят, что они являются величинами порядка  $n^3$

Если  $g(n) \in \Theta(f(n))$ , то  $f(n) \in \Theta(g(n))$ .

Вообще говоря, для любого полинома

$$f(n) = \sum_{i=0}^d a_i \cdot n^i, \text{ где } a_i \text{ — константы и } a_d > 0,$$

справедливы неравенства

∃ такие константы  $c_1 > 0, c_2 > 0, n_0 > 0$ , что  $\forall n \geq n_0$  выполняется

$$0 \leq c_1 \cdot n^d \leq f(n) \leq c_2 \cdot n^d$$

т.е.  $f(n) = \Theta(n^d)$

Смысл нижней и верхней оценки, полученной для алгоритма  $A$  с вычислительной сложностью в худшем случае  $T(l)$  :

$$\begin{array}{ccccc} \text{нижняя} & & & & \text{верхняя} \\ \text{оценка} & & & & \text{оценка} \\ \Omega(f_1(l)) & = & T(l) & = & O(f_2(l)) \\ \uparrow & & & & \uparrow \\ \text{быстрее работать не будет} & & & & \text{медленнее работать не будет} \end{array}$$

$T(l) = O(n^2)$ , алгоритм  $A$  медленнее  $const \cdot n^2$  работать не будет

$T(l) = \Omega(n \cdot \log n)$ , алгоритм  $A$  быстрее  $const \cdot n \cdot \log n$  работать не будет

Алгоритмы, предназначенные для решения определенного класса задач, могут иметь разную временную сложность. В каком случае вычислительную задачу можно считать удовлетворительно решённой?

Для сравнения эффективности алгоритмов часто применяют подход, который заключается в различии между **полиномиальными** и **экспоненциальными** алгоритмами.

Среди специалистов по вычислительным наукам широко распространено мнение, что алгоритм окажется практически полезным для вычислительной задачи только в том случае, если его сложность растет полиномиально относительно размера входа (в то же время продолжается полемика, в ходе которой выдвигаются серьёзные контраргументы утверждению о том, что *полиномиальный* и *практический* – синонимы).



Алгоритм называется **полиномиальным**, если его асимптотическая временная сложность

$$T(l) = O(p(l)),$$

где  $p(l)$  – полином или полиномиально ограниченная функция.

#### Сведения из математики

- 1) Полиномом степени  $k$  от аргумента  $l$  называется функция следующего вида:

$$p(l) = \sum_{i=0}^k a_i \cdot l^i, \quad a_k \neq 0, k - \text{константа.}$$

Полиномом является асимптотически положительной функцией тогда и только тогда, когда  $a_k > 0$ .

- 2) Функция  $p(l)$  полиномиально ограничена, если существует такая константа  $d$ , что

$$p(l) = O(l^d).$$

- 3) Полилогарифмическая функция

$$p(\log l) = \sum_{i=0}^k a_i \cdot (\log l)^i, \quad a_k > 0, k - \text{константа}$$

растёт медленнее, чем любой положительный показатель степени  $l$ .

Алгоритм называется **экспоненциальным**, если его асимптотическая временная сложность

$$T(l) = \Omega(\exp(l)),$$

где  $\exp(l)$  – экспоненциальная функция.

#### *Сведения из математики*

- 1) Экспоненциальная функция это функция:

$$\exp(l) = a^l, \quad a > 1, a - \text{константа.}$$

Например, экспоненциальными являются такие функции  $2^l, 3^l$ .

Экспонента – показательная функция  $e^l$ , где  $e \approx 2,718281$  – основание натурального логарифма.

- 2) Любая экспоненциальная функция асимптотически возрастает быстрее полиномиальной функции:

$$\lim_{l \rightarrow \infty} \frac{p(l)}{\exp(l)} = 0.$$

- 3) Функция  $l!$  Асимптотически возрастает быстрее, чем  $2^l$ , но медленнее, чем функция  $l^l$

$$\lim_{l \rightarrow \infty} \frac{2^l}{l!} = 0,$$

$$\lim_{l \rightarrow \infty} \frac{l!}{l^l} = 0.$$

Различие между полиномиальными и экспоненциальными алгоритмами заметно при решении задач большой размерности, кроме того полиномиальные алгоритмы лучше используют успехи технологий.

Функция	Размер индивидуальной задачи, решаемой за 1 день	Размер индивидуальной задачи, решаемой за 1 день на ЭВМ, скорость которой в 10 раз больше
$n$	$10^{12}$	$10^{13}$
$n \cdot \log n$	$0,948 \cdot 10^{11}$	$0,87 \cdot 10^{12}$
$n^2$	$10^6$	$3,16 \cdot 10^6$
$n^3$	$10^4$	$2,15 \cdot 10^4$
$10^8 \cdot n^4$	10	18
$2^n$	40	43
$10^n$	12	13
$n^{\log n}$	79	95
$n!$	14	15

Для полиномиальных алгоритмов, когда технологические достижения приводят к увеличению скорости вычислений в 10 раз, наибольший размер задач, решаемых полиномиальными алгоритмами, скажем за час, умножается на константу, заключённую между 1 и 10. В противоположность этому для экспоненциального алгоритма произойдет всего лишь аддитивное увеличение размера задачи, которую алгоритм сможет решить за фиксированное время. Таким образом, колоссальный рост скорости вычислений, вызванный появлением нынешнего поколения цифровых вычислительных машин, не уменьшает значение эффективных (полиномиальных) алгоритмов.

С другой стороны, некоторые экспоненциальные алгоритмы достаточно эффективны на практике, когда размеры решаемых задач невелики.

	n=10	n=20	n=30
n	0,00001с	0,00002с	0,00003с
n <sup>2</sup>	0,0001с	0,0004с	0,0009с
n <sup>3</sup>	0,001с	0,008с	0,027с
n <sup>5</sup>	0,1с	3,2с	24,3с
2 <sup>n</sup>	0,001с	1с	17,5 мин
3 <sup>n</sup>	0,059с	58 мин	6,5 лет

Например, при  $n \leq 20$  функция  $f(n) = 2^n$  ведет себя лучше, чем функция  $f(n) = n^5$ . Или будет ли алгоритм с оценкой  $f(n) = n^{80}$  иметь практическое применение, если время, необходимое для решения индивидуальной задачи размера  $n = 3$  уже выражается астрономическим числом, и может оказаться, что некоторый экспоненциальный алгоритм работает лучше при всех разумных входных?

Кроме того, **известны некоторые экспоненциальные алгоритмы** (например, симплекс-метод), **весьма хорошо зарекомендовавшие себя на практике**. Дело в том, что трудоемкость определена как мера поведения алгоритма в наихудшем случае. Утверждение о том, что алгоритм имеет трудоемкость  $2^n$ , означает, что решение по крайней мере одной задачи размерности  $n$  требуется времени порядка  $2^n$ , но на самом деле может оказаться, что для большинства других задач затраты времени значительно меньше.

Следует отметить, что большинство экспоненциальных алгоритмов – это просто варианты полного перебора.

На практике ...

При решении большинства задач как только обнаруживается некоторый полиномиальный алгоритм, так сразу же различные исследователи улучшают идею алгоритма и степень полинома быстро претерпевает ряд уменьшений. Обычно окончательно получается степень роста  $O(n^3)$  или лучше.

Экспоненциальные алгоритмы, напротив, требуют на практике столько же времени, сколько и в теории, и от них, как правило, тут же отказываются, как только для той же задачи обнаруживается полиномиальный алгоритм.

# ПРИМЕРЫ

## Задача

На вход поступает натуральное число  $n$  и массив чисел  $a_1, a_2, \dots, a_n$ .

Нужно найти сумму:  $S = a_1 + a_2 + \dots + a_n$ .

Какой это алгоритм: полиномиальный или экспоненциальный?

Входные данные:  $n$  и  $a_1, a_2, \dots, a_n$ .

$$l = \lfloor \log_2 n + 1 \rfloor + C_1 \cdot n, \quad \max_{1 \leq i \leq n} \lfloor \log_2 a_i + 1 \rfloor = C_1 - \text{конст.}$$

При  $n \rightarrow +\infty$  верно, что  $\lfloor \log_2 n + 1 \rfloor < n$ ,

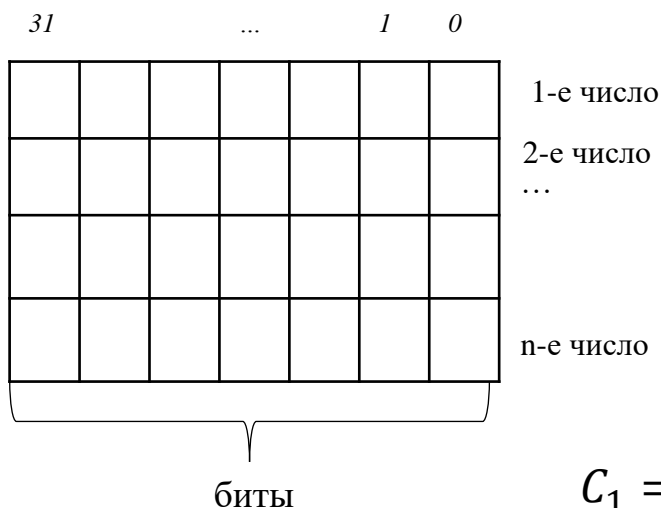
поэтому при  $n \rightarrow +\infty$  получим, что

$$C_1 \cdot n + 1 \leq l < n + C_1 \cdot n = (C_1 + 1) \cdot n$$

$$C_1 \cdot n + 1 \leq l < (C_1 + 1) \cdot n$$

$$\frac{1}{C_1 + 1} \cdot l < n \leq \frac{l - 1}{C_1} < \frac{1}{C_1} \cdot l$$

$$l = \Theta(n)$$



Предполагаем, что в результате суммирования не произойдёт переполнения: все промежуточные результаты вычислений вмещаются в  $C_1$  бит. Так как каждое число занимает  $C_1$  бит, то сложение двух чисел будет выполнено:

- ✓ за время 1, если у нас в RAM равномерная мера измерения времени выполнения операции;

Равномерная мера:

$$T(l) \leq 1 \cdot C_2 \cdot n < \frac{C_2}{C_1} \cdot l$$

$$T(l) = O(l)$$

- ✓ за время  $C_1$  - при логарифмической мере.

Логарифмическая мера:

$$T(l) \leq C_1 \cdot (C_2 \cdot n) < < C_1 \cdot C_2 \cdot \frac{1}{C_1} \cdot l < C_2 \cdot l$$

$$T(l) = O(l)$$

**Ответ:** алгоритм полиномиальный

## Задача

На вход поступает одно натуральное число  $n$ .

Нужно вычислить  $n! = 1 \cdot 2 \cdot \dots \cdot n$ .

Какой это алгоритм: полиномиальный или экспоненциальный?

Входные данные:  
единственное число  $n$ .

Предположим  
сначала, что  
размерность задачи  
 $l = n$   
(унарная система  
кодирования).

Число мультипликативных операций умножения равно  $n$  (оценку выполним по числу операций умножения + учтём начальное присваивание и запись результата в память).

Так как каждое число занимает  $l$  бит, то умножение двух чисел будет выполнено:

✓ за время  $=1$ , если в РАМ  
равномерная мера измерения  
времени выполнения  
операции;

✓ за время равное максимуму  
числа бит операндов - при  
логарифмической мере.

Равномерная мера:

$$T(l) = 1 \cdot l = \Theta(l)$$

Логарифмическая мера:

$$\begin{aligned} T(l) &= (\lfloor \log_2 1 + 1 \rfloor + \lfloor \log_2 2 + 1 \rfloor + \lfloor \log_2 3 + 1 \rfloor) + \\ &+ (\lfloor \log_2 3! + 1 \rfloor + \lfloor \log_2 4! + 1 \rfloor + \dots + \lfloor \log_2 n! + 1 \rfloor) \leq \\ &= (1 + 2 + 2) + (\log_2 3! + \log_2 4! + \dots + \\ &+ \log_2 n! + (n - 3)) \leq \sum_{i=3}^n \log_2 i! + (n - 3) + 5 \end{aligned}$$

Так как  $\log_2 x! = \Theta(x \cdot \log x)$ , то

$$\begin{aligned} T(l) &\leq (3 + \dots + n) \cdot \text{const}_1 \cdot \log_2 n + (n - 3) + 5 \leq \\ &\leq \text{const}_2 \cdot n^2 \cdot \log_2 n + (n - 3) + 5 \end{aligned}$$

$$T(l) = O(l^2 \cdot \log_2 l)$$

**Ответ:** алгоритм – полиномиальный.



# Задача

Сведения из математики:  $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$

На вход поступает одно натуральное число  $n$ .

Нужно вычислить  $n! = 1 \cdot 2 \cdot \dots \cdot n$ .

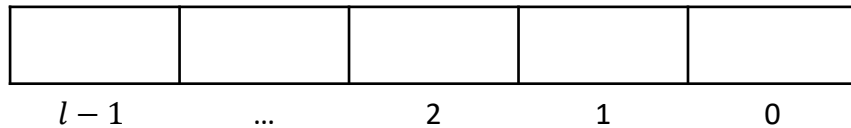
Какой это алгоритм: полиномиальный или экспоненциальный?

Входные данные: единственное число  $n$ .

Предположим, что размерность задачи

$l = \lfloor \log_2 n + 1 \rfloor$  (бинарная система кодирования).

Одной размерности  $l$  соответствуют несколько входов:



$$2^{l-1} \leq n \leq 2^l - 1.$$

Наибольшее число мультипликативных операций (умножение) для задачи размерности  $l$  у входных данных:  $n = 2^l - 1$ .  
Необходимо также учесть, что длина чисел растёт.

Например, на последнем шаге мы умножаем число  $n$  (в нём столько бит, сколько на входе алгоритма) на число  $(n-1)!$ , которое гораздо длиннее.  
Так, значение  $21!$  уже не помещается в int64.

Число мультипликативных операций умножения равно  $2^l - 1$ .

Умножение двух чисел будет выполнено:

- ✓ за время =1, если у нас в RAM равномерная мера измерения времени выполнения операции;

Равномерная мера:

$$T(l) = 1 \cdot (2^l - 1)$$

$$T(l) = \Theta(2^l)$$

- ✓ за время равное максимуму числа бит операндов - при логарифмической мере.

Логарифмическая мера:

$$\begin{aligned} T(l) &= (\lfloor \log_2 1 + 1 \rfloor + \lfloor \log_2 2 + 1 \rfloor + \lfloor \log_2 3 + 1 \rfloor + \dots + \lfloor \log_2 (2^l - 1)! + 1 \rfloor) \\ &> 5 + (\log_2 3! + \dots + \log_2 (2^l - 1)!) \\ &[\text{т. к. } \log_2(x!) = \Theta(x \cdot \log x)] \\ &> 5 + (3 + \dots + (2^l - 1)) \cdot \log_2 3 \cdot \text{const}_1 = \\ &= (2^{l-1} + 1) \cdot (2^l - 3) \cdot \text{const}_2 + 5 \end{aligned}$$

**Ответ:** алгоритм экспоненциальный.

Алгоритм называется **псевдополиномиальным**,  
если для любой его индивидуальной задачи  $I$

$$T(I) = \Theta( p(l, |Max(I)|) ),$$

где  $p$  — полином от двух переменных:

(1) размерности задачи  $l$  (длина в битах входных данных индивидуальной задачи  $I$ );

(2)  $|Max(I)|$  — значение наибольшего по абсолютной величине числового параметра индивидуальной задачи  $I$  (если у задачи несколько числовых параметров, то иногда берут среднее из них).

Для задач, имеющих числовые параметры, псевдополиномиальные алгоритмы на практике ведут себя как экспоненциальные только при очень больших значениях числовых параметров индивидуальных задач.

Во всех случаях, кроме очень больших значений числового параметра (которые могут и не встречаться в реальных задачах), они работают, как полиномиальные.

## Задача

На вход поступает натуральное число  $n$  и массив натуральных чисел  $a_1, a_2, \dots, a_n$ .  
Подсчитать частоту встречаемости всех элементов массива.

**Рассмотрим следующий псевдополиномиальный алгоритм**

- 1) найдем максимальный элемент в массиве, предположим, что это число  $M$ :

$$M = \max_{1 \leq i \leq n} a_i$$

- 2) выполним цикл по  $i$  от 1 до  $M$  и для каждого значения  $i$  подсчитаем частоту встречаемости числа  $i$  в исходном массиве;

	1	2	3	4	5	6
A	2	5	2	2	1	5

$$M = \max_{1 \leq i \leq n} a_i = 5$$
$$i = 1..M$$

↓	$i = 1$	частота (1)=1
	$i = 2$	частота (2)=3
	$i = 3$	частота (3)=0
	$i = 4$	частота (4)=0
	$i = 5$	частота (5)=2

(продолжение)

Размерность задачи:

$$l = \lfloor \log_2 n + 1 \rfloor + \sum_{i=1}^n \lfloor \log_2 a_i + 1 \rfloor$$

Если предположить, что

$$\max_{1 \leq i \leq n} \lfloor \log_2 a_i + 1 \rfloor = \text{const},$$

то величина

$$M = \max_{1 \leq i \leq n} a_i = 2^{\text{const}} - \text{константа}.$$

Вычислительная сложность алгоритма для такой модели:

$$T(l) = \Theta(M \cdot n), \text{ где } n = \Theta(l), M = 2^{\text{const}} - \text{константа}.$$

При такой мере измерения длины входа **алгоритм на практике будет вести себя, как полиномиальный.**

Следующая задача является примером ***NP*-полной задачи** комбинаторной оптимизации, для которой существует **псевдополиномиальный алгоритм** (основан на методе динамического программирования):

## Задача о рюкзаке:

заданы рюкзак ограниченной вместимости  $W$  и  $n$  предметов. Для каждого предмета задан его вес  $w_i > 0$  и стоимость  $p_i > 0$  (например, если стоимость выше, то вещь более ценная).

Требуется определить набор предметов суммарный вес которых не превосходит  $W$  (т.к. вместимость рюкзака ограничена), а стоимость набора максимальна (взять наиболее ценные предметы).

$$\text{Размерность задачи: } l = \underbrace{\lfloor \log_2 n + 1 \rfloor + \sum_{i=1}^n \lfloor \log_2 w_i + 1 \rfloor + \sum_{i=1}^n \lfloor \log_2 p_i + 1 \rfloor}_{l_1} + \underbrace{\lfloor \log_2 W + 1 \rfloor}_{l_2}$$

(продолжение)

	1	2				$W$
1						
...						
$n$						

Так как дополнительная память, которая потребуется алгоритму  $\Theta(W \cdot n)$  , то для того, чтобы индивидуальную задачу можно было решить методом ДП нужно, чтобы число предметов (  $= n$  ) вместимось рюкзака ( $= W$ ) были не слишком велики.

наибольшая  
суммарная ценность  
вещей рюкзака веса= $j$   
собранного из первых  
 $i$  предметов



$dp[i, j] = dp[i - 1, j]$

если  $w_i \leq j$ , то

$dp[i, j] = \max\{dp[i, j], dp[i - 1, j - w_i] + p_i\}$

Ответ:

наибольший элемент в последней строке матрицы.



(продолжение)

Размерность задачи:

$$l = \underbrace{\lfloor \log_2 n + 1 \rfloor + \sum_{i=1}^n \lfloor \log_2 w_i + 1 \rfloor + \sum_{i=1}^n \lfloor \log_2 p_i + 1 \rfloor}_{l_1} + \underbrace{\lfloor \log_2 W + 1 \rfloor}_{l_2}$$

**(1) если предположить, что**

$$\max_{1 \leq i \leq n} \lfloor \log_2 w_i + 1 \rfloor = C_1,$$

$$\max_{1 \leq i \leq n} \lfloor \log_2 p_i + 1 \rfloor = C_2,$$

где  $C_1$  и  $C_2$  — константы, то битовую длину  $W$  также можно ограничить:

$$\lfloor \log_2 W + 1 \rfloor \leq C_1 \cdot n.$$

**Тогда в данной модели размерность задачи  $l = \Theta(n)$ , а алгоритм будет вести себя, как полиномиальный:**

$$T(l) = \Theta(W \cdot n), \text{ где } W = O(n), n = \Theta(l).$$

(продолжение)

Размерность задачи:

$$l = \underbrace{\lfloor \log_2 n + 1 \rfloor + \sum_{i=1}^n \lfloor \log_2 w_i + 1 \rfloor}_{l_1} + \underbrace{\sum_{i=1}^n \lfloor \log_2 p_i + 1 \rfloor + \lfloor \log_2 W + 1 \rfloor}_{l_2}$$

(2) если не ограничивать константами битовую длину параметров задачи, то не сложно увидеть, что

$$l_1 > n$$

Поэтому выполняется  $n = o(l_1)$ .

Так как  $l_2 \stackrel{\text{def}}{=} \lfloor \log_2 W + 1 \rfloor \leq \log_2 W + 1$ , то  
 $W \geq 2^{l_2-1}$ .

Получаем алгоритм, который для данной модели будет работать, как экспоненциальный по  $W$ :

$$T(l) = \Theta(W \cdot n), W = \Omega(2^{l_2}), n = o(l_1) (n = o(l))$$
$$T(l) = \Theta(\exp(l_2) \cdot p(l_1)).$$



БЕЛОРУССКИЙ  
ГОСУДАРСТВЕННЫЙ  
УНИВЕРСИТЕТ

Спасибо за внимание!