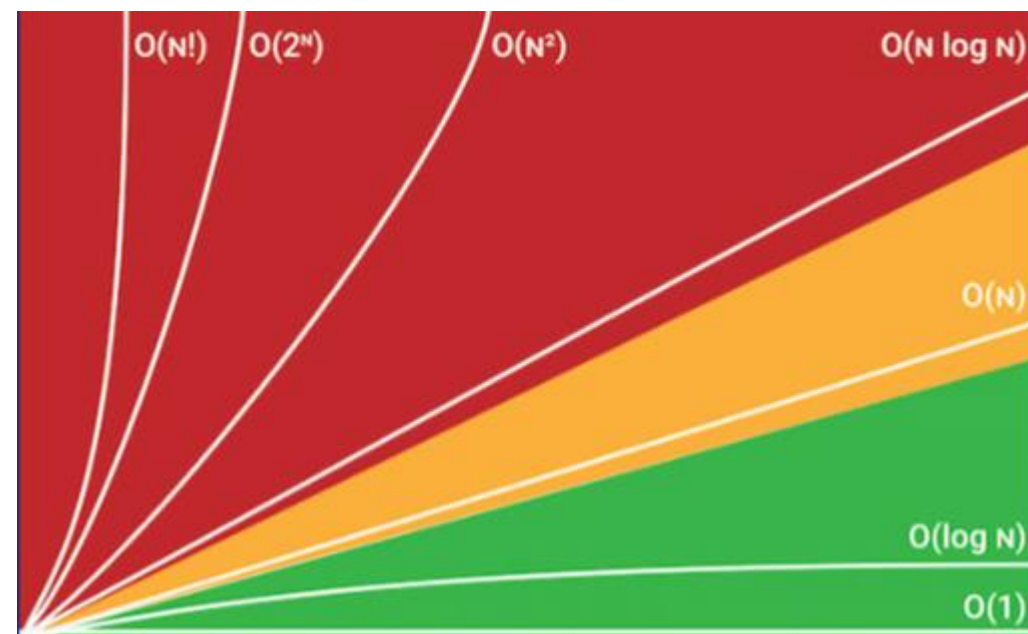


Анализ сложности алгоритмов



Алгоритм

это конечная последовательность чётко определённых, реализуемых компьютером инструкций, предназначенная для решения определённого 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.

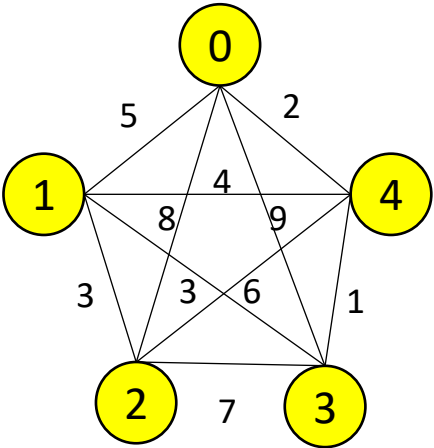
В рамках нашей дисциплины мы будем работать с **детерминированными алгоритмами**.

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.

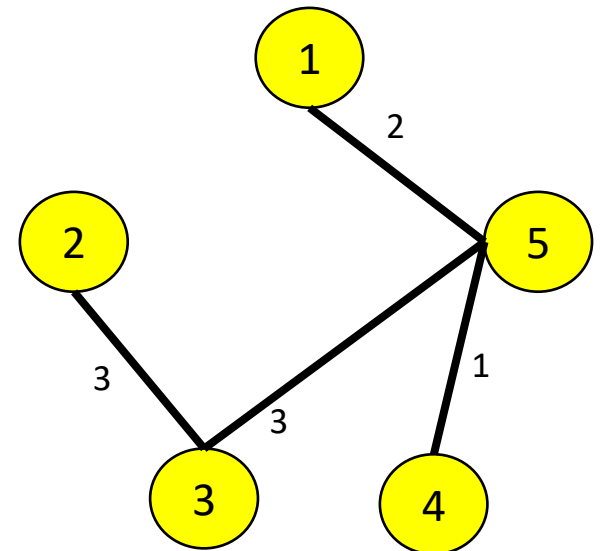
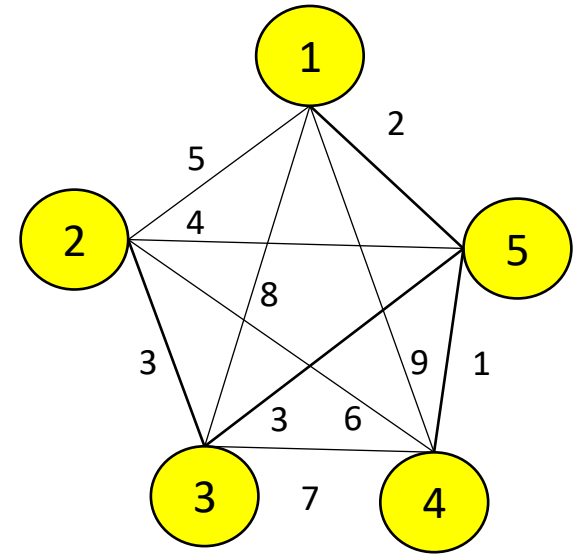
1!	1
2!	2
3!	6
4!	24
5!	120
6!	720
7!	5040
8!	40320
9!	362880
10!	3628800
11!	39916800
12!	479001600
13!	6227020800
14!	87178291200
15!	1307674368000
16!	20922789888000
17!	355687428096000
18!	6402373705728000
19!	121645100408832000
20!	2432902008176640000
21!	$5,10909 \cdot 10^{19}$
22!	$1,124 \cdot 10^{21}$
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 .



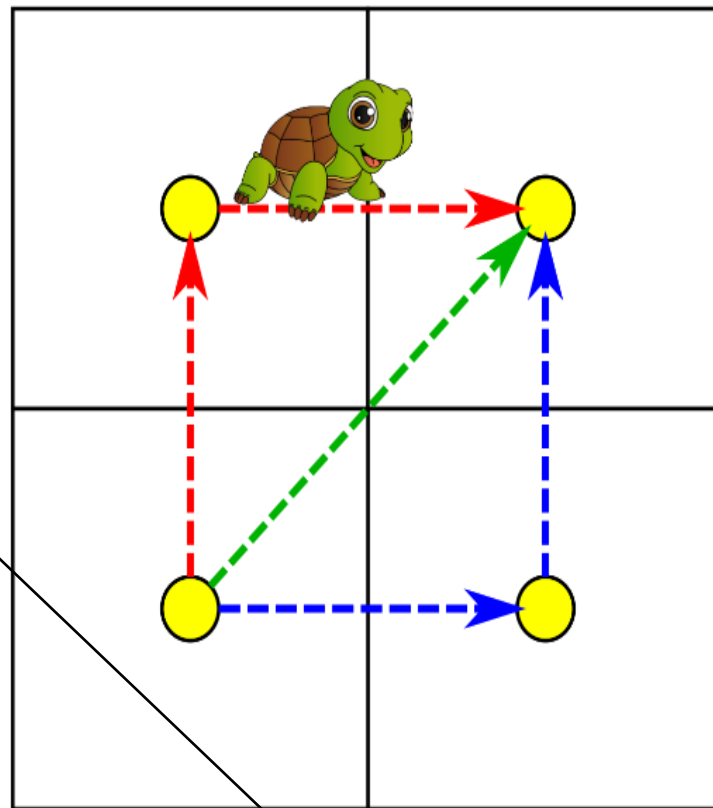
Решение задач в системе автоматического тестирования **iRunner**

При решении индивидуальных задач в системе автоматического тестирования iRunner для каждой задачи в условии заданы ограничения на входные данные, а также указаны ограничения по памяти и времени, в которые должна уложиться программа.

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 $1\,000\,000\,007$.

Ограничение по памяти: 64 Мб.

$$1 \text{ Мб} = 10^6 \text{ байт}$$

$$64 \text{ Мбайт} = 64 \cdot 10^6 \text{ байт}$$

одно целое число (int) занимает в памяти компьютера 4 байта

Ограничение по памяти в 64 Мб позволит выделить в памяти участок не более, чем для

$$\frac{64 \cdot 10^6}{4} = 16 \cdot 10^6 \approx 10^7 \text{ целых чисел.}$$

В задаче $0 \leq n, m \leq 10^4$, значит выделить память под матрицу размера $n \times m$ не получится.

Тактовая частота процессора

4 ГГц (гигагерц) означает, что вычислительное устройство выполняет **за 1 секунду**

$4 \cdot 10^9$ тактовых циклов (абстрактных операций)

1 тактовый цикл (1 операция) выполняется за

$$\frac{1}{4} \cdot 10^{-9} \text{ секунд} = \frac{1}{4} \text{ наносекунда}$$

Процессор суммирует два 64-х битных числа за 1 тактовый цикл, т.е. за $\frac{1}{4}$ наносекунды.

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

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.

Принятые решения



Вердикты

WA	Неправильный ответ	21461
OK	Принято	9135
RTE	Ошибка во время выполнения	6559
TLE	Нарушен предел времени	5803
CE	Ошибка компиляции	5585
PE	Ошибка представления	2493
MLE	Нарушен предел памяти	963
ILE	Нарушен предел ожидания	4
SV	Нарушение безопасности	4
CF	Ошибка тестирования	1

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,

1. Кэширование данных

Процессоры имеют многоуровневую систему кэшей (L1, L2, L3), постоянно сохраняющую те или иные ячейки для более быстрого доступа.

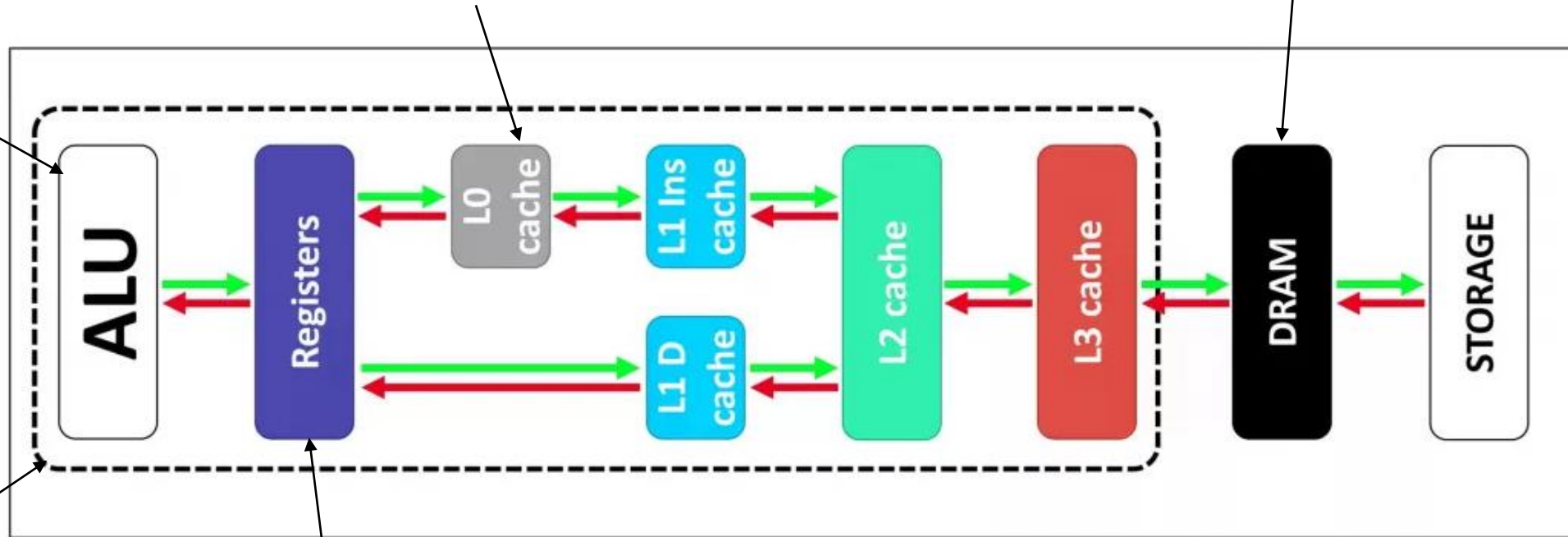
В зависимости от того, закэшировал ли процессор нужную ячейку, время доступа к данным может отличаться в десятки раз.

Уровни (Layers) памяти компьютера

**Arithmetic
Logic Units**
(арифметико-
логическое
устройство)

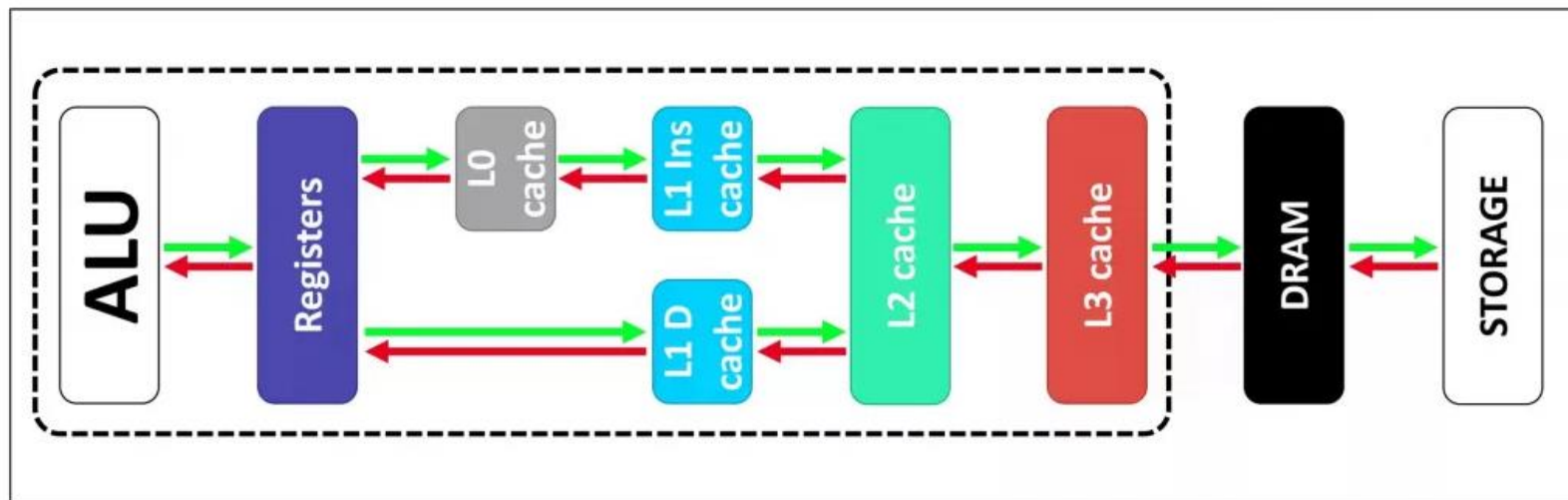
блок памяти всего
на 1 500 мопс (микроопераций)

оперативная память (энергозависимая)
Dynamic Random Access Memory
(микроскопические конденсаторы)



кристалл
CPU

файл регистров (около 7Кб); каждый регистр хранит
одно число, например, 64-битное целое число (код
инструкции, фрагмент данных, адрес памяти др. данных),
передают данные ALU за 1 тактовый цикл



Доступ к данным

на жестком диске:

1 000 наносекунд

DRAM – объем до 4-х Гб,
микр. конденсаторы

оперативная память компьютера

100 наносекунд (400 тактовых циклов)

SRAM
(транзисторы)

кэш-память (L3)

7 ½ наносекунд (более 30 тактовых циклов)

кэш-память (L2)

2 ½ наносекунд (более 10 тактовых циклов)

кэш-память (L1)

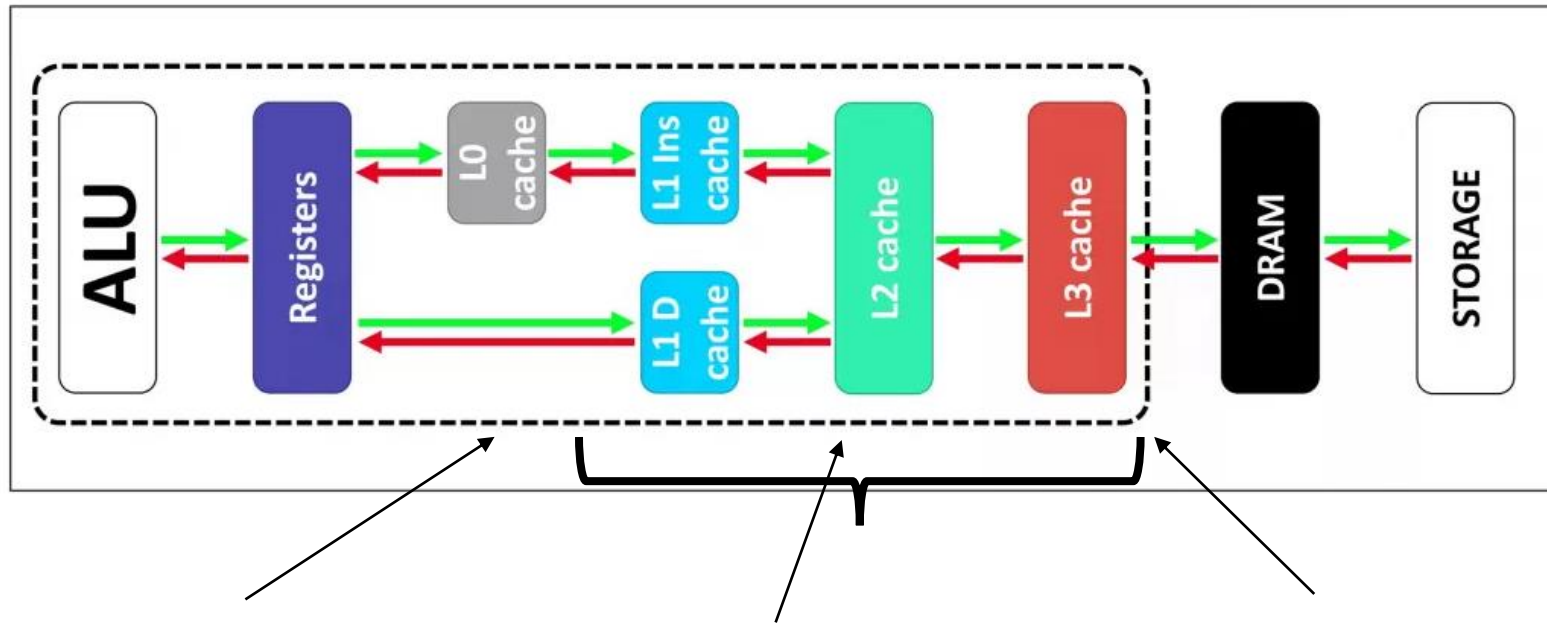
1 ¼ наносекунда (примерно 5 тактовых циклов)

регистры ЦП

¼ наносекунда (1 тактовый цикл процессора)

Кэш-память (сверхоперативная память, «тайник»), это отдельные микросхемы, расположенные непосредственно на кристалле ЦП в непосредственной близости от его логических блоков (**SRAM** – Static Random Access Memory); это транзисторная память, поэтому требует много места на кристалле и поэтому ее много не поставить:

100 Мб SRAM = 4 Гб DRAM



L1 – кэш первого уровня:

несколько десятков Кб (маленькая по объему)

делится на 2 блока:

кэш-команд **Instruction cache**,

кэш-данных **Data cache**

быстрая (почти в 100 раз быстрее DRAM, но в среднем в 5 раз медленнее регистров),

L2:

несколько сотен Кб

(больше по объему, чем L1)

примерно в 2 раза медленнее, чем L1

L3:

до нескольких десятков Мб

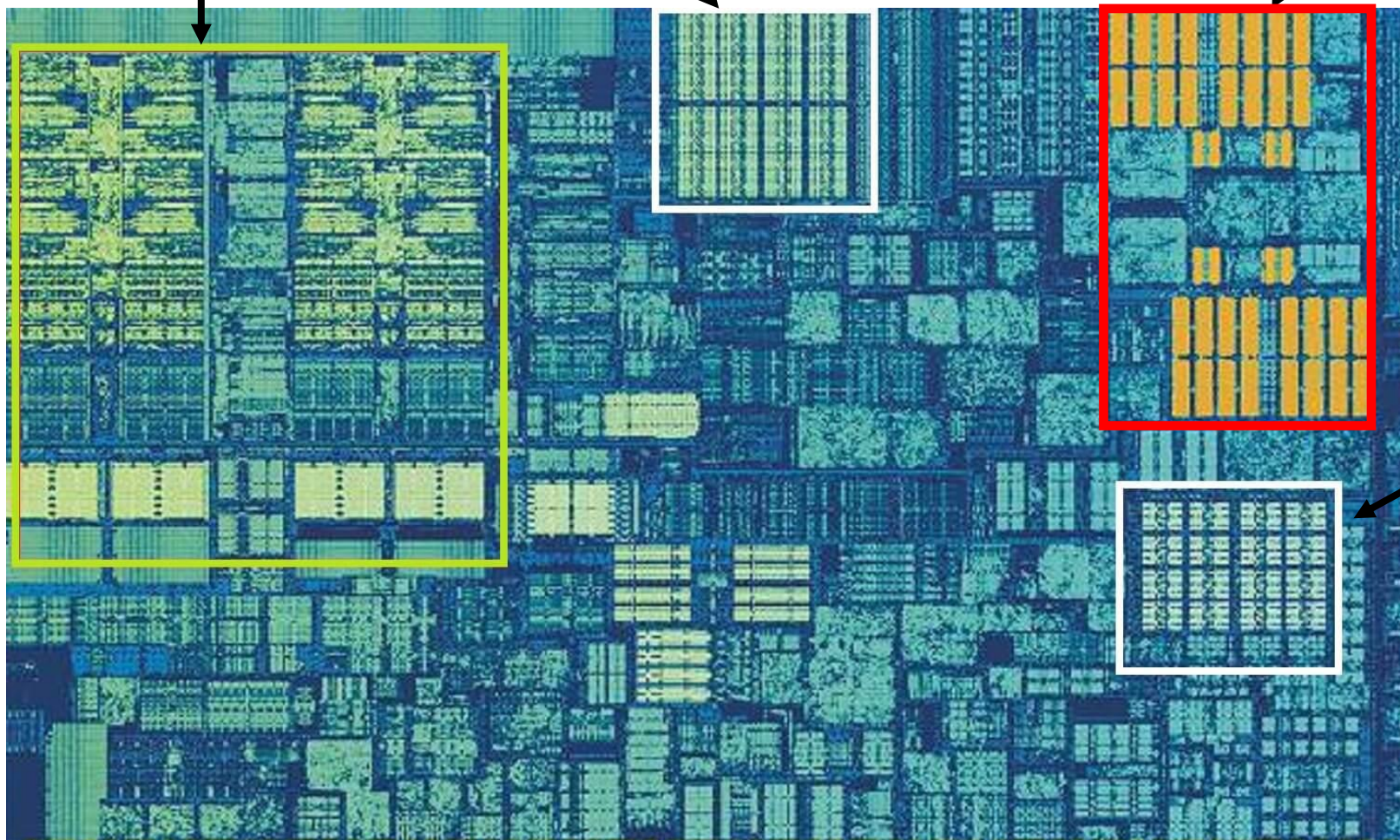
самая медленная, но ее много

для многоядерных процессоров доступ к L3 разрешен из любого ядра ЦП

Arithmetic Logic Units
Registers

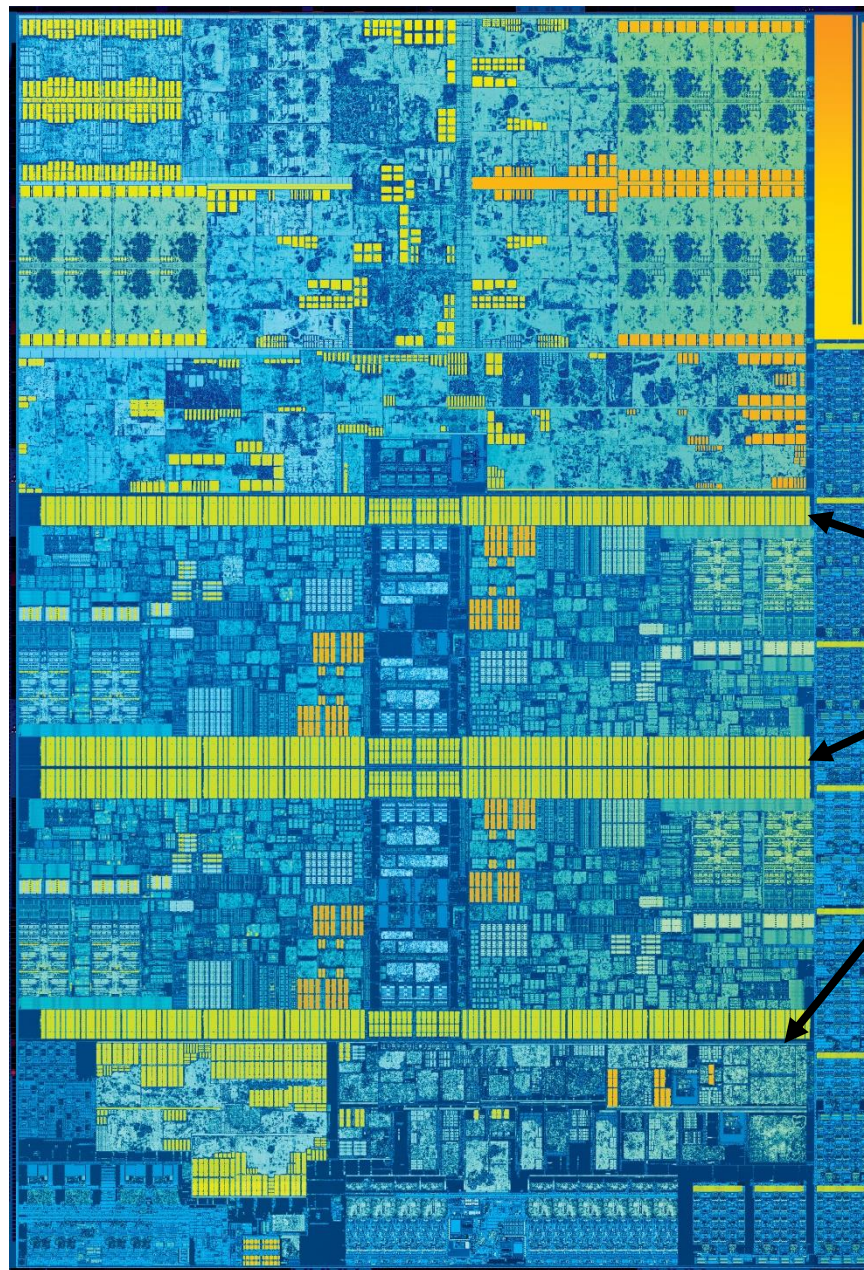
L1 (Data cache, 32K6)

L2 (Data cache, до 512 K6)



L1 (Instruction cache, 32K6)

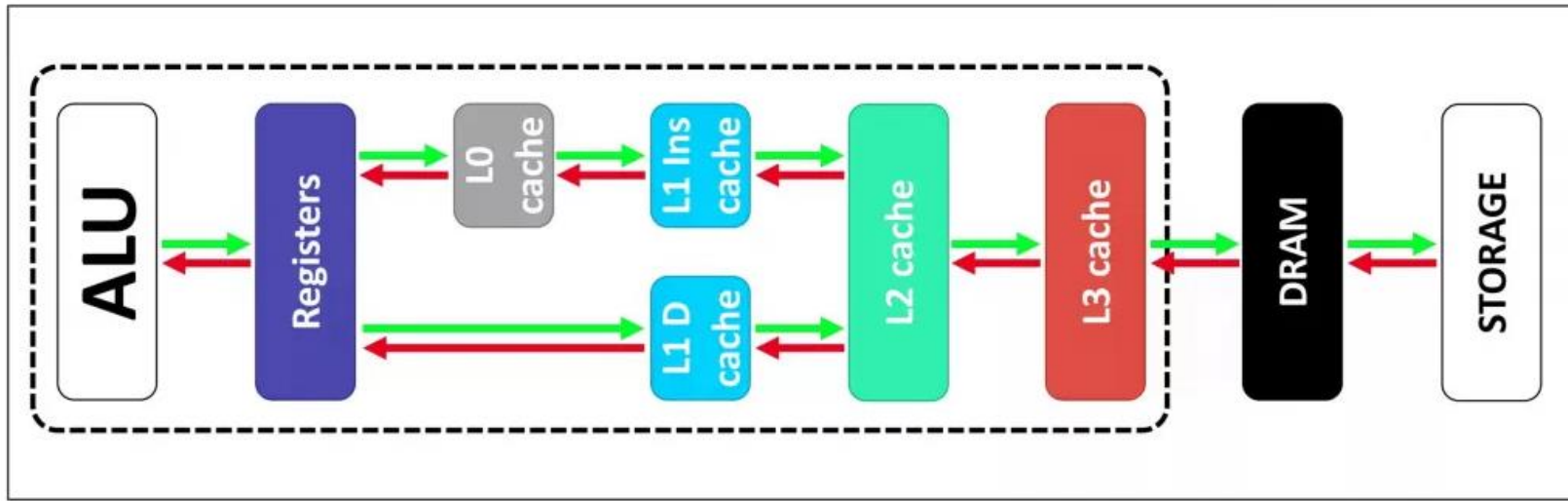
Одно ядро процессора Intel Skylake



L3

для многоядерных процессоров
доступ к L3 разрешен из любого
ядра ЦП

6th Generation Intel Core processor



Идея кэширования опирается на *свойство локальности данных и инструкций*:

если происходит обращение по некоторому адресу, то велика вероятность, что в ближайшее время произойдет обращение к памяти по тому же адресу или соседним.

В кэш помещается информация, к которой часто обращается процессор (информация из ОЗУ помещается в кэш, а потом к ней обращается процессор).

Система процессорного кэша состоит из:

- (1) контроллера кэша, который управляет движением данных между ЦП, ОЗУ и кэш-памятью
- (2) кэш-памяти.

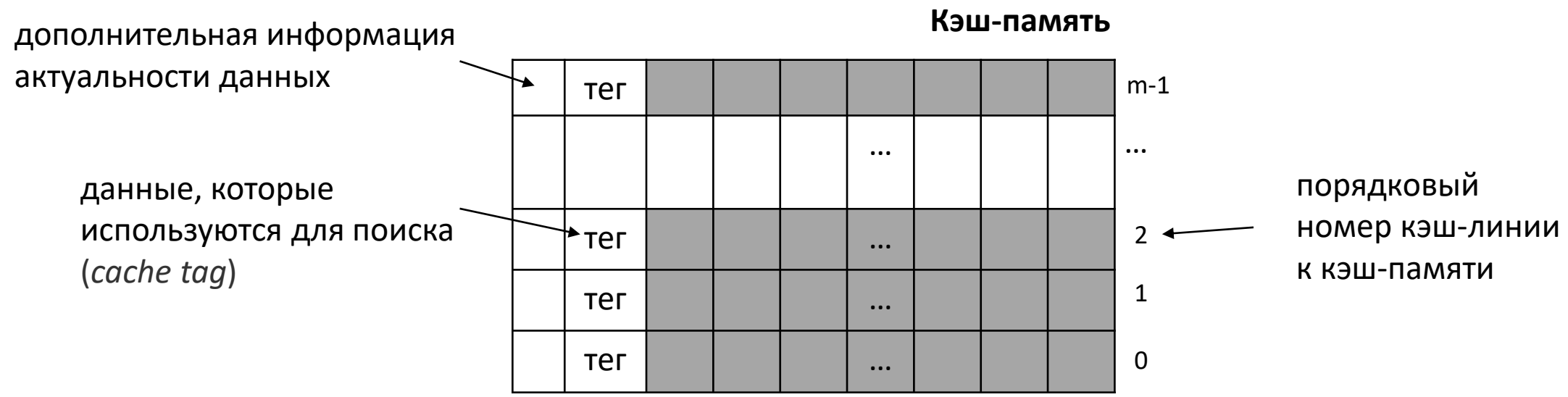
Кэш-попадание: при запросе нужные данные есть в кэш-памяти.

Кэш-промах: данных нет в кэш-памяти.

Кэш-память делится на строки одинаковой длины, которые называются **кэш-линиями** (*cache lines*).

Каждая кэш-линия сначала содержит вспомогательную информацию, а затем идут сами данные.

Размер кэш-линии определяется размером блока данных, который там хранится (как правило, размер кэш-линии - **64 байта**).



Ассоциативное отображение DRAM на кэш-память
(fully associative mapping): любой блок RAM памяти может быть записан в любую кэш-линию.

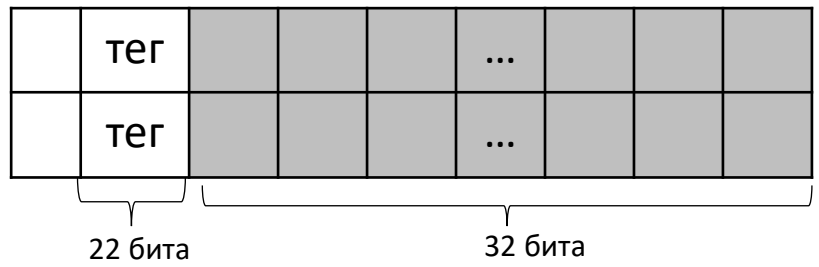
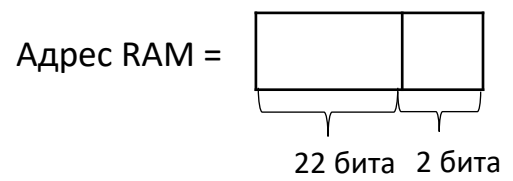
		Кэш-память						
2	тег				...			
1	тег				...			
0	тег				...			

Оперативная память (RAM)

RAM делится на **блоки (линии)**, размер блока = размеру кэш-линии

Адрес каждого байта в RAM состоит из:
 порядкового № блока (**тег**),
 порядкового № байта внутри блока.

Если RAM=16Мб, КЭШ=64Кб, кэш-линия=4 байта, то число блоков=16Мб/4б=4 · 10⁶ = 2²², значит для задания номера блока (тега) нужно = 22 бита, а для задания смещения байта внутри блока нужно 2 бита.



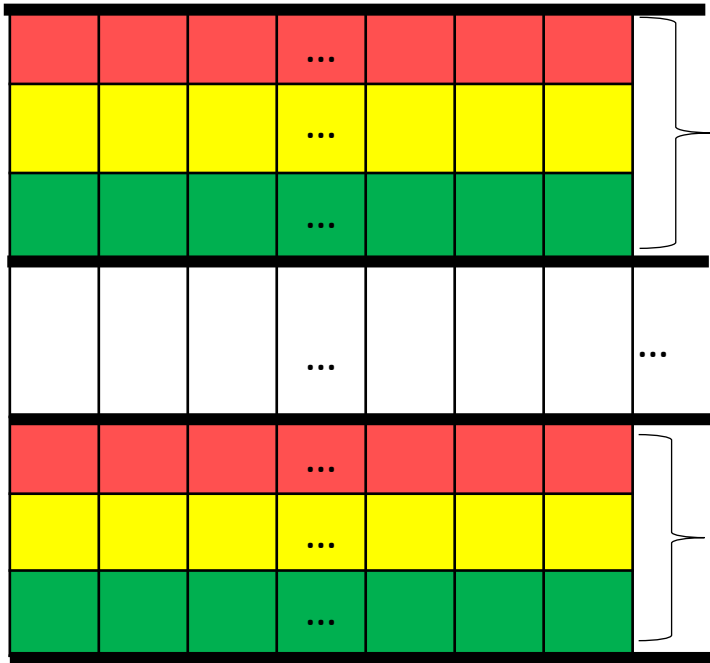
Поиск данных в кэше при ассоциативном отображении:

Тэги всех линий сравниваются со старшей частью адреса одновременно (одновременное сравнение - это сложная аппаратная задача, поэтому время доступа к ассоциативной кэш-памяти, как правило, значительно больше, чем время доступа к кэш-памяти с прямым отображением, которое будет рассмотрено далее).

Прямое отображения DRAM на кэш-память (direct mapping):

блоки из разных сегментов, но с одинаковыми порядковыми номерами в этих сегментах, всегда будут отображаться на одну и ту же кэш-линию кэш-памяти.

Оперативная память (RAM)

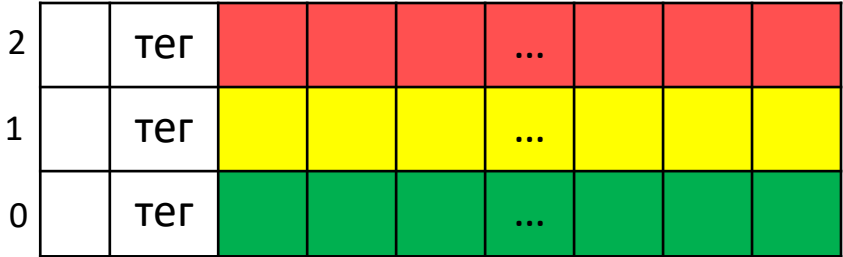


RAM делится на **сегменты**
размер сегмента = размеру кэша

сегменты делятся на **блоки (линии)**
размер блока = размеру кэш-линии

Адрес каждого байта RAM состоит из:
порядкового № сегмента (**тег**),
порядкового № блока внутри сегмента (**индекс**),
порядкового № байта внутри блока (**смещение**).

Кэш-память



Тег каждой кэш-линии хранит старшую часть адреса первого байта этой линии (**порядковый номер сегмента в RAM, где этот байт располагался**).

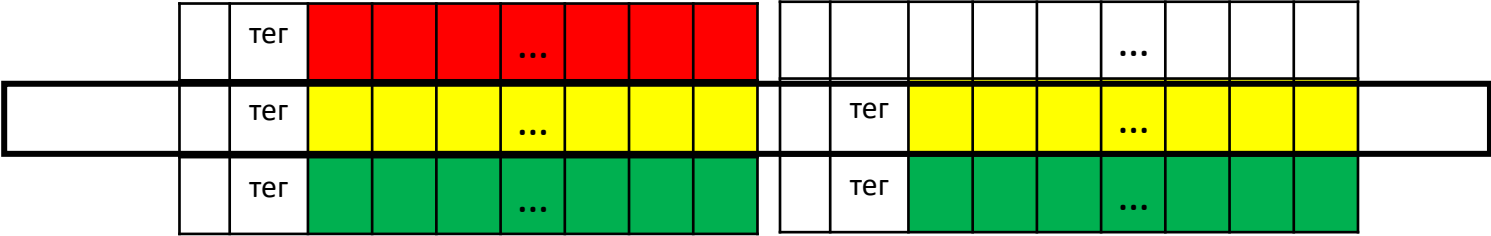
Поиск данных в кэше при прямом отображении:

1. Извлекается средняя часть адреса байта, по ней определяется номер кэш-линии (на рисунке – цвет).
2. Тег кэш-линии сравнивается со старшей частью адреса байта.

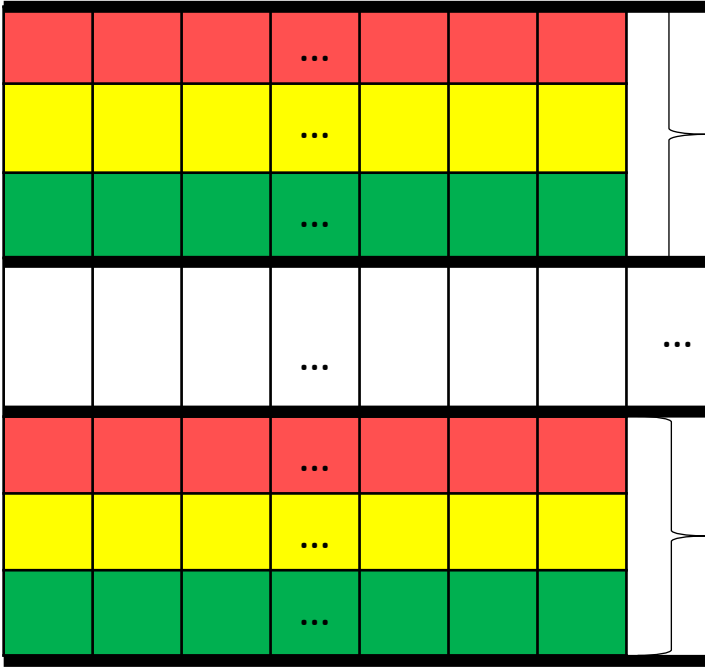
Недостатки: если в программе последовательно происходят обращения к элементам памяти, отстоящим на величину, кратную размеру кэш-памяти, то все эти элементы будут претендовать на одну и ту же строку кэш-памяти (буксирование кэш-памяти).

Множественно-ассоциативное отображение DRAM на кэш-память (set associative mapping)

Кэш-память делится на «*банки*»



Оперативная память (RAM)



RAM делится на **сегменты**
размер сегмента = размеру кэша
одного банка

сегменты делятся на **блоки**
размер блока = размеру кэш-линии

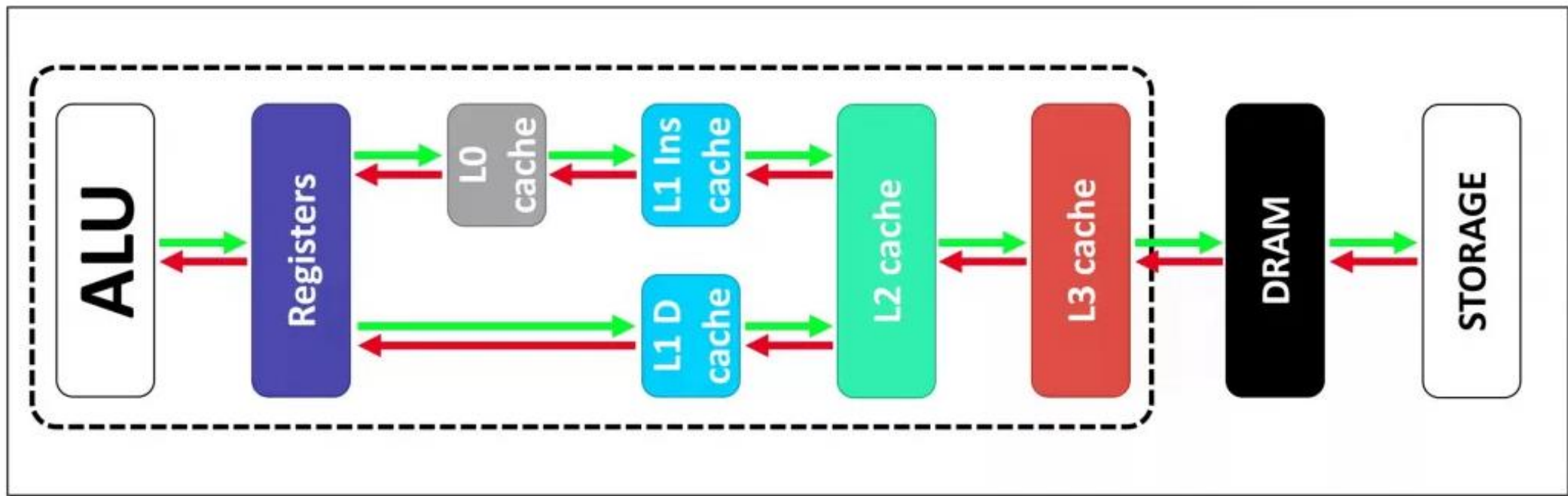
Адрес каждого байта RAM состоит из:
порядкового № сегмента (**тег**),
порядкового № блока внутри сегмента (**индекс**),
порядкового № байта внутри блока (**смещение**).

Блоки из разных сегментов, но с одинаковыми индексами (один цвет), всегда будут записаны в одно множество (набор) (если все занято, то работает алгоритм замещения), количество блоков в множестве называется *степенью ассоциативности* (2,4,8,16).

Тэг каждой кэш-линии хранит старшую часть адреса первого байта этой линии (**порядковый номер сегмента в RAM**, где этот байт располагался).

Поиск данных в кэше при наборно ассоциативном отображении:

1. Извлекается средняя часть адреса байта, по ней определяется номер множества в кэше.
2. Теги всех кэш-линии данного множества сравниваются со старшей частью адреса одновременно.



Взаимодействие уровней кэш-памяти между собой это достаточно сложный процесс, в ходе которого используются разные технологии и математические алгоритмы):

эксклюзивное (уникальность информации);

инклюзивное (дублирование информации);

и др.

Эксклюзивная архитектура предполагает уникальность информации, находящейся в L1 и L2.

При считывании процессором информации из кэша она берется из L1.

Если в L1 нужная информация не найдена, то она ищется в L2.

Если нужная информация найдена в L2, то по принципу **LRU** (принцип удаления наиболее "старых данных" - LRU (Least-Recently Used)) кэши первого и второго уровня обмениваются между собой строками (самая "старая" строка из L1 помещается в L2, а на ее место записывается нужная строка из L2).

Если нужная информация не найдена и в L2, то обращение идет к ОЗУ по схеме, описанной выше.

При копировании информации из ОЗУ в кэш - информация сразу заносится в L1.

Когда L1 заполнен, то, по принципу **LRU** информация переносится из L1 в L2.

Инклюзивная архитектура предполагает дублирование информации, находящейся в L1 и L2.

При считывании процессором информации из кэша, она берется из L1.

Если нужной информации в кэше первого уровня нет, то она ищется в L2.

Если нужная информация в кэше второго уровня найдена, то она дублируется в L1 по принципу **LRU**, а затем, передается в процессор.

Если нужная информация не найдена и в кэше второго уровня, то она считывается из.

Во время копирования информации из ОЗУ в кэш делается две копии:

одна копия заносится в L2, другая копия - в L1.

Когда L1 полностью заполнен, информация замещается по принципу **LRU**.

Аналогично происходит и с кэшем второго уровня, но, поскольку его объем больше, то и информация хранится в нем дольше.

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,

1. 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.

2. 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.

3. 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.

Подробнее: <https://www.agner.org/optimize/microarchitecture.pdf>



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.

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, уложившись в выделенные лимиты по времени и памяти.
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	
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'

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^2	0,0001с	0,0004с	0,0009с
n^3	0,001с	0,008с	0,027с
n^5	0,1с	3,2с	24,3с
2^n	0,001с	1с	17,5 мин
3^n	0,059с	58 мин	6,5 лет

Как следует из данных таблицы, если число абстрактных операций алгоритма 1 оценивается полиномиальной функцией n^5 , а алгоритма 2 – экспоненциальной функцией 2^n , то при ограничениях на входные данные $n \leq 20$ быстрее работает алгоритм 2.

Однако, при $n \rightarrow +\infty$ алгоритм 2 будет работать существенно дольше, чем алгоритм 1.

Более того, если функция, описывающая число абстрактных операций, выполняемых алгоритмом, растет экспоненциально, то при увеличении объема входных данных программа может не завершиться за миллиарды лет, хотя при небольших данных она работает за секунду и успешно проходит по времени все тесты в iRunner. **Такой алгоритм решения задачи на практике не применим.**

Теоретический анализ сложности алгоритма

Теоретический анализ сложности алгоритма заключается в анализе поведения алгоритма при увеличении объема входных данных.

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 некий **неотрицательный числовой параметр**, характеризующий входные данные для алгоритма, и оценивать трудоёмкость алгоритма с помощью функций, зависящих от этого числового параметра.

Этот параметр называют ***размерностью задачи (размер входа)*** и обозначают буквой ***l*** .

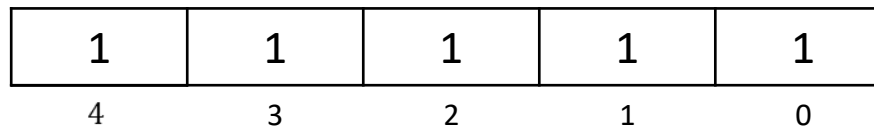
В качестве размерности задачи l принято брать число бит, необходимое для кодировки входных данных задачи.

Для того, чтобы понять, сколько бит нужно для кодировки входных данных, надо определиться с системой кодирования входных данных. Рассмотрим унарную и бинарную системы кодирования.

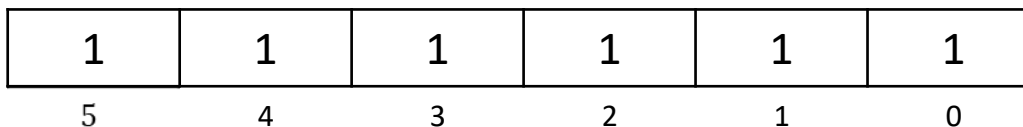
На вход поступает одно целое число n , какова битовая длина l этого числа?

Унарная система кодирования:

длина в битах равна самому числу



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

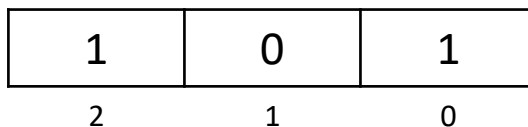


$n=6$ ($l = 6$)

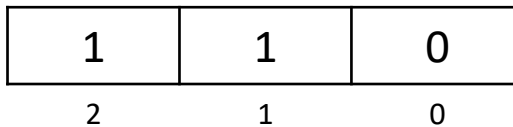
Вход: 1 1 0 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1

Бинарная система кодирования:

длина в битах равна количеству цифр в его двоичной записи



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



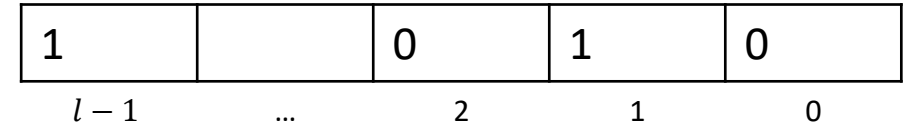
$n = 6 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 110_2$ ($l = 3$)

Будем рассматривать только **компактное представление числа**:

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

На вход поступает одно целое неотрицательное 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 входа 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 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 $\max_{1 \leq i \leq n} \lfloor \log_2 |a_i| + 1 \rfloor = \text{const}$ (где, например, $\text{const} = 32$ или 64),
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 $\max_{1 \leq i \leq n} [\log_2 |a_i| + 1] = \text{const}$, в качестве такой числовой характеристики входных данных может выступать число n элементов массива; 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 ребер $|E|$;
- ✓ 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 наибольшее из времен (по всем входам этой размерности), то она называется **временной сложностью в худшем случае**.

Если при данной размерности l в качестве меры сложности берётся среднее время (по всем входам этой размерности), то она называется **средней сложностью** (среднее время работы алгоритма по всем возможным наборам входных данных задачи размерности l).

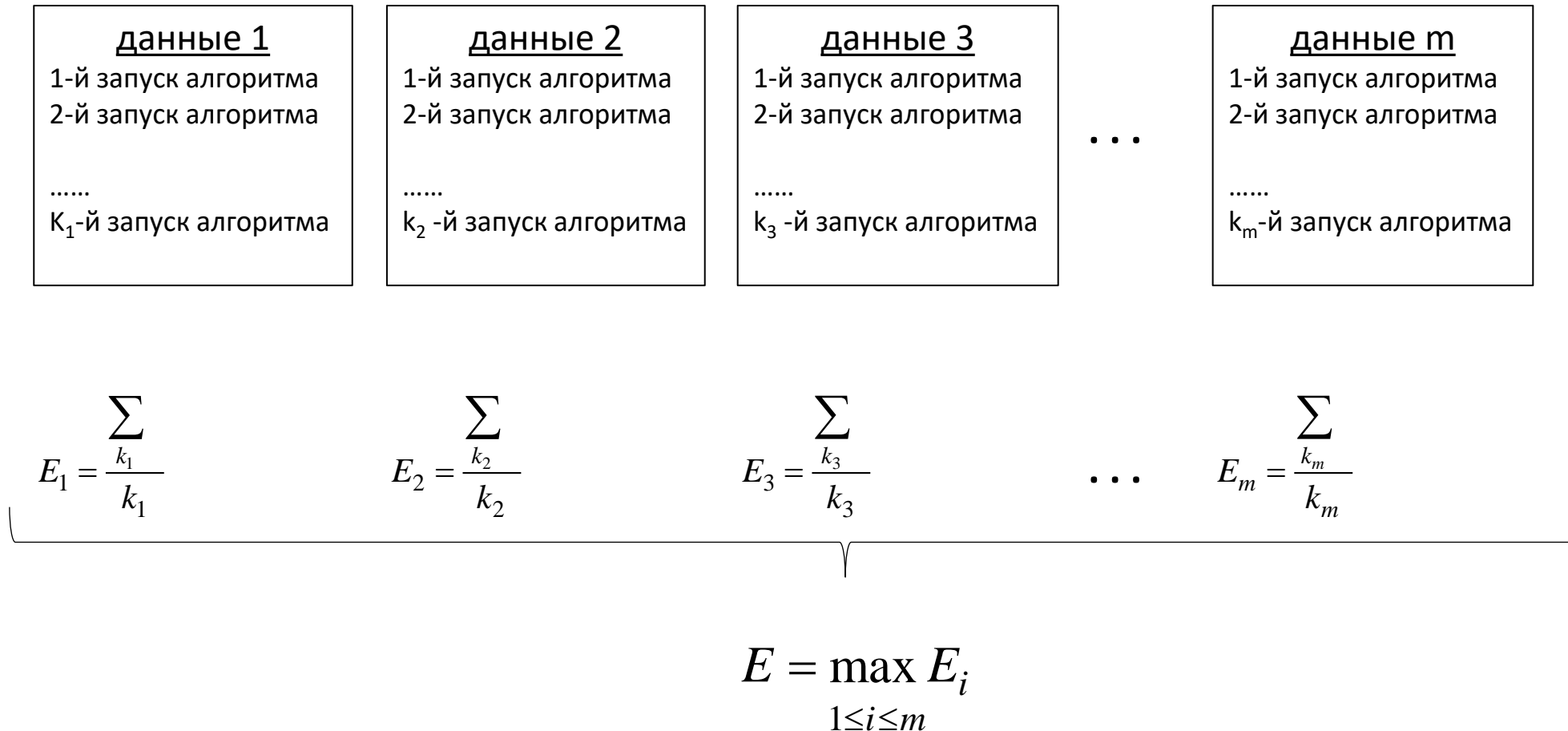
Ёмкостная сложность алгоритма –

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

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

(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

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

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

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

Модель абстрактного вычислительного устройства

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

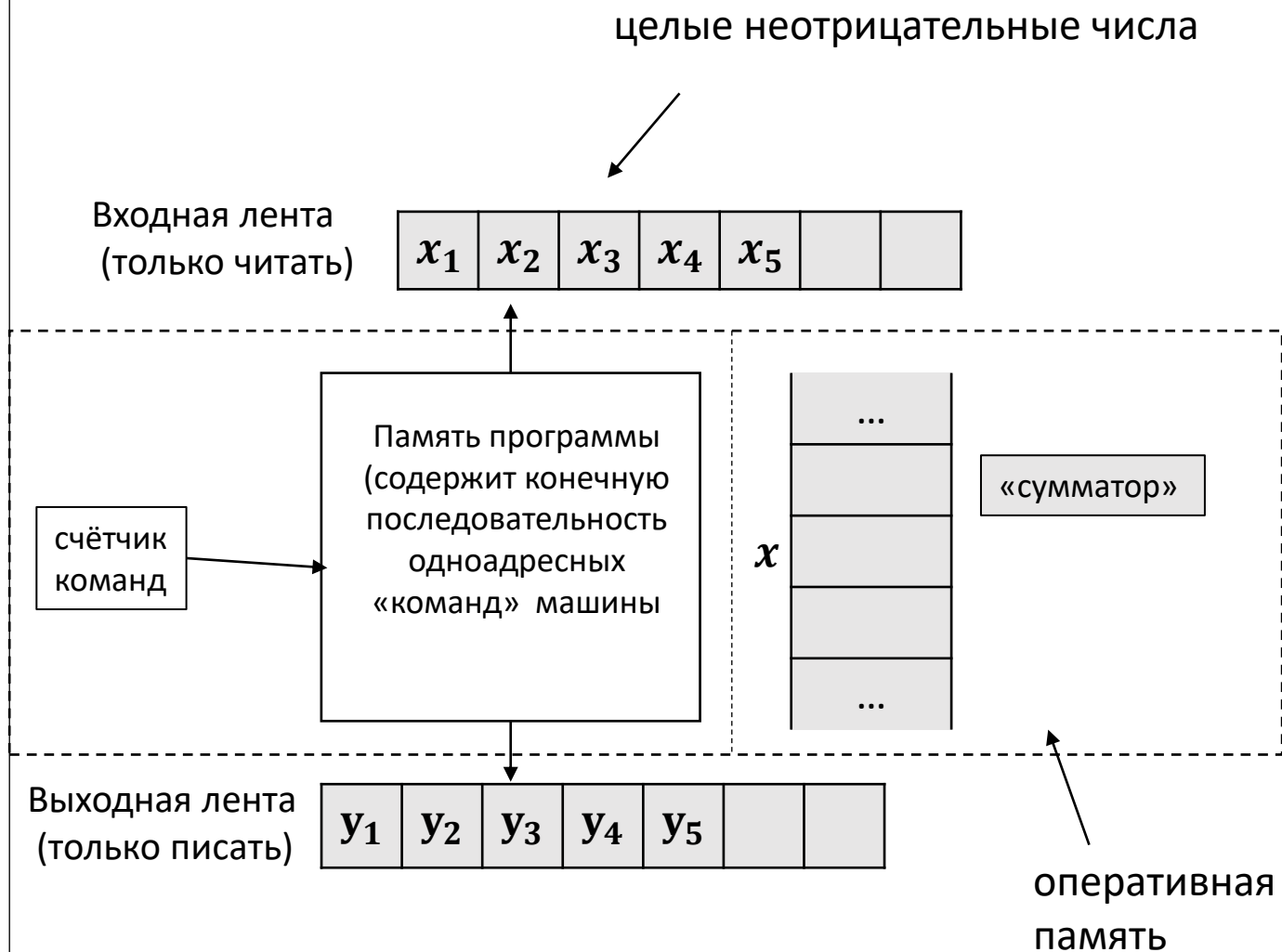
Как определить время выполнения и объём памяти программной реализации алгоритма на некотором **абстрактном вычислительном устройстве**?

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;

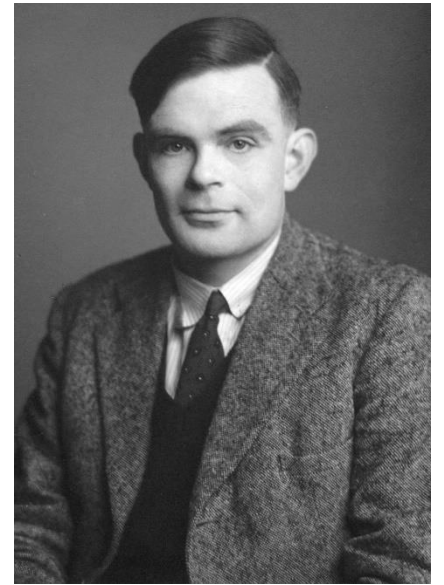
РАМ - вычислительная машина с одним сумматором, в котором команды программы не могут изменять сами себя, состоит из:

- ✓ **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. Тьюрингом 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

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

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.

При равномерной мере сложности **время выполнения каждой команды** и **размер каждой ячейки** для РАМ-машины полагается равным единице.

Теперь

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

входная лента

2 ¹⁰⁰	2 ¹⁰					
------------------	-----------------	--	--	--	--	--

READ 1

LOAD 1

READ 2

ADD 2

STORE 3

WRITE 3

сумма двух чисел (равномерная мера сложности)

время работы программы равно 6 (в программе шесть команд);

объем памяти равен 3 (к трем ячейкам памяти шло обращение);

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

Логарифмическая мера сложности

При логарифмической мере сложности время выполнения каждой команды для РАМ-машины полагается по порядку равным логарифму величины максимального операнда плюс 1, 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 **сумма двух чисел**
LOAD 1 (логарифмическая мера сложности)

READ 2 время выполнения = 516?;
 $101 + 101 + 11 + 101 + 101 + 101 = 516$
ADD 2 объем памяти = 314?
STORE 3 (1яч.) 101 бит + (2яч.) 11 бит +
WRITE 3 (3яч.) 101 бит + (сумматор) 101 бит
 = 314

READ 1 **произведение двух чисел** (логарифмическая
LOAD 1 мера сложности)

READ 2 время выполнения = 536?;
 $101 + 101 + 11 + 101 + 111 + 111 = 536$
MUL 2 объем памяти = 334?
STORE 3 (1яч.) 101 бит + (2яч.) 11 бит +
WRITE 3 (3яч.) 111(бит) + (сумматор) 111 бит
 = 334

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

Асимптотики

Рассмотрим две неотрицательные функции $f(n)$ и $g(n)$.

Будем говорить, что функция $f(n)$ растет **асимптотически медленнее**, чем функция $g(n)$, если

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$$

Будем говорить, что функция $f(n)$ растет **асимптотически не быстрее**, чем функция $g(n)$, если для некоторой константы $const$ справедливо

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} \leq const$$

Будем говорить, что функция $g(n)$ растет **асимптотически быстрее**, чем функция $f(n)$, если

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$$

Будем говорить, что функция $f(n)$ растет **асимптотически не медленнее**, чем функция $g(n)$, если для существует такая константа $const > 0$, для которой

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} \geq const$$

Для функции $T(n)$, характеризующей время работы алгоритма в худшем случае, важна скорость её роста при возрастании объема входных данных.

Поэтому, если функция $T(n)$ – сложная, то будем учитывать только ту её часть, которая с ростом аргумента к бесконечности, растёт не медленнее и не быстрее, чем $T(n)$. Эту часть функции $T(n)$ будем называть **порядком роста функции**.

Например, для функции

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

порядок роста можно определить, например, как $const \cdot n^2$ (говорят, что функция $T(n)$ растёт как n^2).

Покажем, насколько важна скорость роста функций, несмотря на технический прогресс, приводящий к увеличению роста быстродействия ЭВМ.

Предположим, что за сутки современная ЭВМ можно выполнять 10^{12} абстрактных операций.

Если алгоритму требуется для решения задачи выполнить $f(n) = n^3$ операций, то за сутки ЭВМ успеет решить задачу для $n = \sqrt[3]{10^{12}} = 10^4$.

Если предположить, что скорость вычислений за сутки возрастает в 10 раз, то ЭВМ за сутки выполняет $\tilde{n}^3 = 10 \cdot 10^{12}$ операций и теперь ЭВМ успевает решить задачу для $\tilde{n} = \sqrt[3]{10 \cdot 10^{12}} = \sqrt[3]{10} \cdot n = 2,15 \cdot n$

Если алгоритму требуется для решения задачи выполнить $f(n) = \beta^n$ операций, то за сутки ЭВМ успеет решить задачу для $n = \log_{\beta} 10^{12}$.

Если предположить, что скорость вычислений за сутки возрастает в 10 раз, то ЭВМ за сутки выполняет $\beta^{\tilde{n}} = 10 \cdot 10^{12}$ и теперь ЭВМ успевает решить задачу для $\tilde{n} = \log_{\beta}(10 \cdot 10^{12}) = \log_{\beta} 10 + n$

В таблице, показано, как растет размер индивидуальной задачи, решаемой за один день ЭВМ, если скорость вычислений увеличивается в 10 раз:

Функция	Размер индивидуальной задачи, решаемой на ЭВМ за 1 день	Размер индивидуальной задачи, решаемой за 1 день на ЭВМ, скорость которой в 10 раз больше
n	10^{12}	$10 \cdot 10^{12}$
$n \cdot \log n$	$0,948 \cdot 10^{11}$	$8,7 \cdot 10^{11}$
n^2	10^6	$3,16 \cdot 10^6$
n^3	10^4	$2,15 \cdot 10^4$
$10^8 \cdot n^4$	10	$1,8 \cdot 10$
2^n	40	$3 + 40 = 43$
10^n	12	$1 + 12 = 13$
$n^{\log n}$	79	$16 + 79 = 95$
$n!$	14	$1 + 14 = 15$

В общем случае справедливы следующие утверждения.

1. Если время работы алгоритма оценивается функцией $f(n) = \log_2 n$, то увеличение скорости вычислений ЭВМ ν раз позволит решить за такое же время задачу, размер которой в 2^ν раз больше.
2. Если время работы алгоритма оценивается функцией $f(n) = n^\alpha$, то увеличение скорости вычислений ЭВМ в ν раз позволит решить за такое же время задачу, размер которой в $\sqrt[\alpha]{\nu}$ раз больше (мультипликативное увеличение размера задачи, которую алгоритм может решить за фиксированное время).
3. Если время работы алгоритма оценивается функцией $f(n) = \beta^n$, то увеличение скорости вычислений ЭВМ в ν раз позволит решить за такое же время задачу, размер которой лишь на $\log_\beta \nu$ больше (аддитивное увеличение размера задачи, которую алгоритм может решить за фиксированное время).

ВЫВОДЫ

Видно, что использование алгоритмов, время которых ограничено функцией вида n^α позволилократно увеличить размер решаемых задач при увеличении быстродействия ЭВМ.

Таким образом, колоссальный рост скорости вычислений, вызванный появлением нынешнего поколения цифровых вычислительных машин, не уменьшает важность разработки эффективных алгоритмов.

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 $n \rightarrow +\infty$

Асимптотика

$O(f(n))$ –

это множество функций, которые растут не быстрее, чем функция $f(n)$.

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

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

Асимптотика

$\Omega(f(n))$ –

это множество функций, которые растут не медленнее, чем функция $f(n)$.

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

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

Асимптотика

$\Theta(f(n))$ –

это множество функций, которые растут не быстрее и не медленнее, чем функция $f(n)$.

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

$$25 \cdot n \cdot \log_2 n \in \Theta(n \cdot \log n)$$

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

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

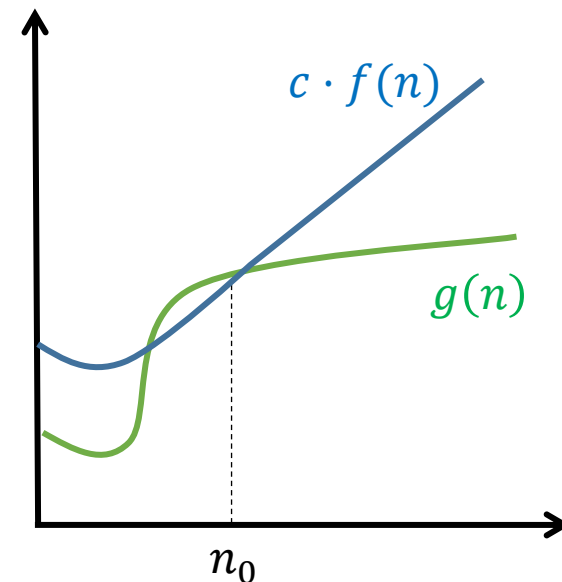
но мы будем также употреблять эквивалентную запись:

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

Запишем математически определения асимптотик.

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

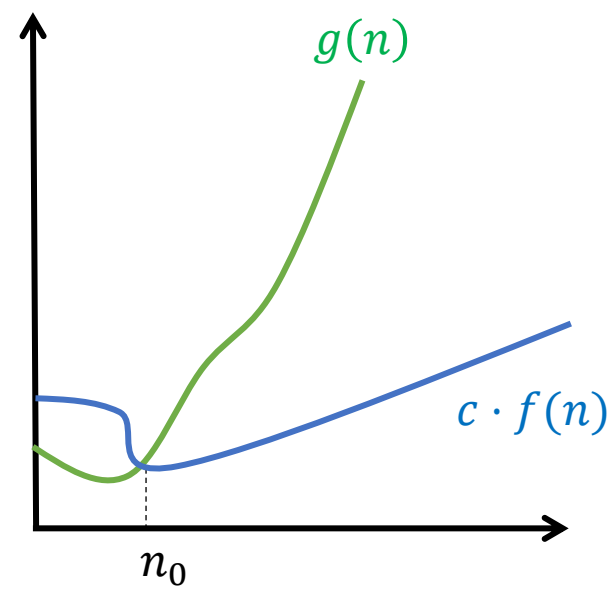
- $4 \cdot n^3 \in \Omega(n)$
- $4 \cdot n^3 \in \Omega(n^2 \cdot \log n)$
- $4 \cdot n^3 \in \Omega(n^3)$

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 $g(n) \notin \omega(f(n))$.

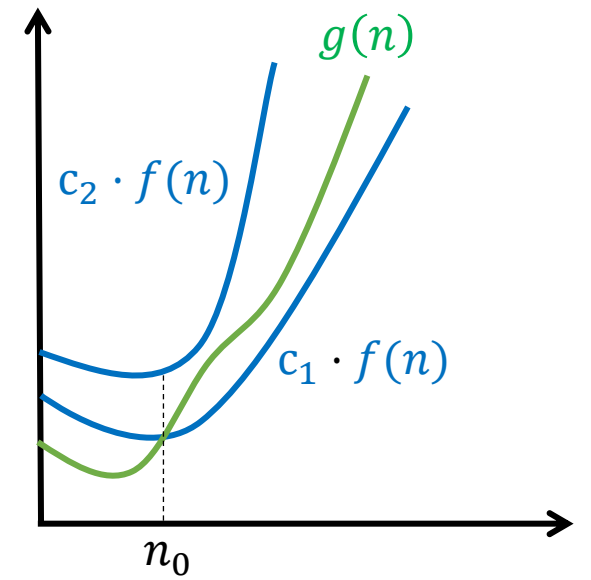
для функции $g(n) = 4 \cdot n^3$ функция $f(n) = n^3$ является точной оценкой: $4 \cdot n^3 \in \Omega(n^3)$ и $4n^3 \notin \omega(n^3)$.

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).

$$\begin{aligned} 4n^3 &\in \Theta(n^3) \\ 2n^3 &\in \Theta(n^3) \\ n^3 &\in \Theta(n^3) \end{aligned}$$

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 оценки, полученной для алгоритма A с вычислительной сложностью в худшем случае $T_A(l)$:

если $T_A(l) = \Omega(f_1(l))$, то алгоритм A быстрее, чем $f_1(l)$ работать не будет.

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

если $T_A(l) = O(f_2(l))$, то алгоритм A медленнее, чем $f_2(l)$ 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.

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) **Полилогарифмическая функция** – это функция следующего вида:

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

Полилогарифмическая функция растет медленнее, чем любой положительный показатель степени l

$$p(\log l) = O(l^\alpha), \alpha > 1.$$

3) **Функция $f(l)$ полиномиально ограничена**, если существует такая константа $\alpha > 1$, что

$$f(l) = O(l^\alpha).$$

Например, $f(l) = l \cdot \log l$ – полиномиально ограничена, так как $f(l) = l \cdot \log l = O(l^2)$.

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

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

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

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 **экспоненциальным**,
если его асимптотическая временная сложность

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

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

Сведения из математики (продолжение)

4) Напомним, что экспоненциальная функция это показательная функция следующего вида:

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

Например, функции $2^l, 3^l$ являются экспоненциальными функциями (экспонента – это показательная функция e^l , где $e \approx 2,718281$ – основание натурального логарифма).

Любая экспоненциальная функция асимптотически растёт быстрее полиномиальной функции:

$$\lim_{l \rightarrow +\infty} \frac{l^\alpha}{\beta^l} = 0.$$

Функция $l!$ асимптотически растёт быстрее, чем β^l , но медленнее, чем l^l

$$\lim_{l \rightarrow +\infty} \frac{\beta^l}{l!} = 0, \quad \lim_{l \rightarrow +\infty} \frac{l!}{l^l} = 0.$$

Алгоритм называется **псевдополиномиальным**, если для любой его индивидуальной задачи I размерности l его асимптотическая временная сложность

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

где p — полином от двух переменных

- (1) размерности задачи l (длина в битах входных данных индивидуальной задачи I);
- (2) $|\text{Max}(I)|$ — 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.

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

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.

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

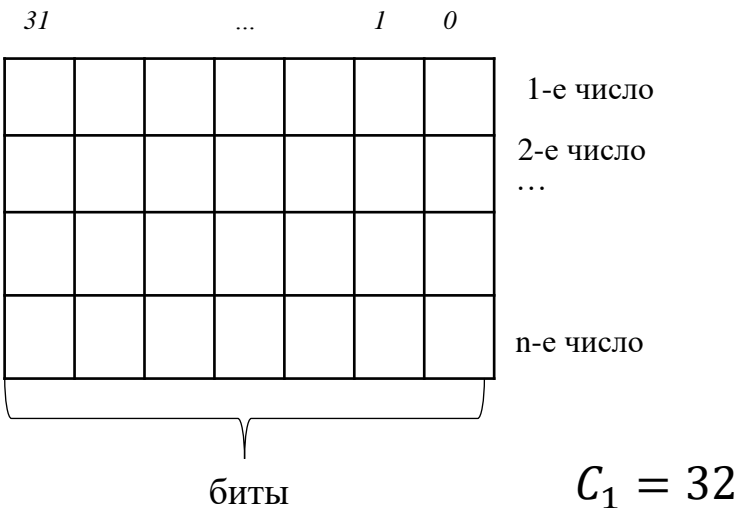
ПРИМЕРЫ ПОЛУЧЕНИЯ ОЦЕНОК ВЫЧИСЛИТЕЛЬНОЙ СЛОЖНОСТИ АЛГОРИТМА

Задача. Сумма элементов.

На вход поступает натуральное число n и массив чисел a_1, a_2, \dots, a_n .
Необходимо найти сумму: $S = a_1 + a_2 + \dots + a_n$. Определить асимптотическую сложность алгоритма. 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;
- ✓ за время C_1 - при логарифмической мере.

Равномерная мера:

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

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. Факториал числа.

На вход поступает одно натуральное число n .

Нужно вычислить $n! = 1 \cdot 2 \cdot \dots \cdot n$. Определить асимптотическую сложность алгоритма. Какой это алгоритм: полиномиальный или экспоненциальный?

Входные данные:
единственное
натуральное число n .

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

Число мультипликативных операций умножения равно n (оценку выполним по числу операций умножения, а также 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 RAM
равномерная мера измерения
времени выполнения
операции;

✓ за время равное максимуму
числа бит операндов - при
логарифмической мере
измерения времени
выполнения операции.

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.

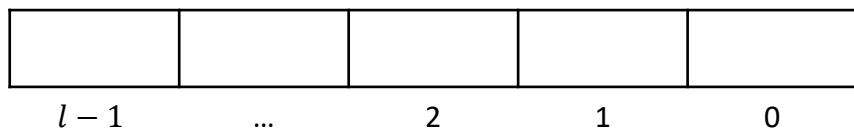
Задача. Факториал числа (продолжение).

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

Входные данные: единственное число n .

Предположим, что размерность задачи $l = \lfloor \log_2 n \rfloor + 1$ (бинарная система кодирования).

Задаче размерности l могут соответствовать несколько входов:



Все возможные входы описываются неравенствами:

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

Наибольшее число мультипликативных операций (умножение) для задачи размерности l будет для входа: $n = 2^l - 1$.

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:

$$\begin{aligned} T(l) &= (\lfloor \log_2 1 \rfloor + 1) + (\lfloor \log_2 2 \rfloor + 1) + (\lfloor \log_2 3 \rfloor + 1) + (\lfloor \log_2 3! \rfloor + 1) + (\lfloor \log_2 4! \rfloor + 1) + \dots + (\lfloor \log_2 (2^l - 1)! \rfloor + 1) \\ &> 5 + (\log_2 3! + \dots + \log_2 (2^l - 1)!) \\ &[\text{т.к. } \log_2(x!) = \Theta(x \cdot \log x)] \\ &> 5 + (3 + \dots + (2^l - 1)) \cdot \log_2 3 \cdot \text{const}_1 = \\ &= (2^{l-1} + 1) \cdot (2^l - 3) \cdot \text{const}_2 + 5 \end{aligned}$$

Ответ: алгоритм – экспоненциальный.

Задача. Проверить, является ли натуральное число n простым.

Алгоритм: проверить, есть ли среди чисел $2, 3, \dots, \lfloor \sqrt[n]{n} \rfloor$ хотя бы одно, делящее n .
 Определить асимптотическую сложность алгоритма. Какой это алгоритм: полиномиальный или экспоненциальный?

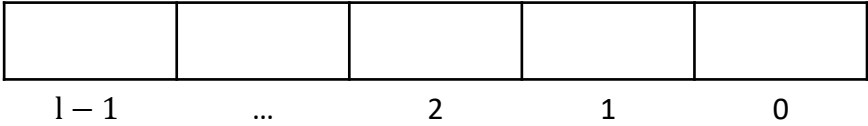
Решение. Оценку проведем по числу мультипликативных операций: деления. Если в качестве размерности задачи l взять само число n (унарная система кодирования входа), то не получится выписать точную формулу для временной сложности $T(l)$.
 В таблице 1.4 приведено число операций деления, которые нужно выполнить алгоритму для проверки на простоту числа l .

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

Видно, что функция $T(l)$ не является монотонной, поэтому можно лишь показать, что при $l \rightarrow +\infty$ для $T(l)$ справедлива оценка сверху $T(l) \leq \lfloor \sqrt[n]{l} \rfloor - 1$.
 Алгоритм – полиномиальный.

(продолжение)

Рассмотрим теперь бинарную систему кодирования входных данных. Задаче размерности l соответствует несколько входов (это множество натуральных чисел, компактная запись которых в двоичном представлении содержит ровно l бит):



Все возможные входы для задачи размерности задачи l описываются неравенствами:

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

$10\dots00$

$11\dots11$

Для асимптотической оценки временной сложности алгоритма в худшем случае необходимо найти такой вход, для которого число мультипликативных операций (деления) наибольшее.

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 $[\sqrt{p_l}] - 1$.

Так как справедливы неравенства $2^{l-1} \leq p_l < 2^l$, то верно следующее соотношение

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

Учитывая которое, получаем следующие неравенства:

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

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:

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

Таким образом, бинарной системе кодирования входа удалось получить оценки сверху и снизу для временной сложности алгоритма проверки числа на простоту:

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

Таким образом, бинарной системе кодирования входа удалось получить оценки сверху и снизу для временной сложности алгоритма проверки числа на простоту:

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

Откуда можно сделать вывод, что алгоритм при бинарной системе кодирования входных данных и равномерной мере для РАМ-машины является экспоненциальным:

$$T(l) = \Theta\left(\sqrt[2]{2^l}\right)$$

или

$$T(l) = \Theta(\beta^l), \text{ где } \beta = \sqrt[2]{2}.$$

Задача. Частота встречаемости всех элементов массива.

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

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

Вычислительная сложность алгоритма для такой модели:

$$T(l) = \Theta(M \cdot n), \text{ где } n = \Theta(l), M = (2^{\text{const}} - 1) - \text{константа}.$$

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).

Размерность задачи:

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

(продолжение) задача о рюкзаке

	1	2	...	j	...	W
1						
...						
i						
...						
n						

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

если $w_i \leq j$, то

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

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

наибольшая суммарная ценность вещей рюкзака веса j , который собран из первых i предметов (при этом не обязательно все из них были взяты в рюкзак)

Ответ:

наибольший элемент в последней строке матрицы dp .

Размерность задачи:

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

(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:

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

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!