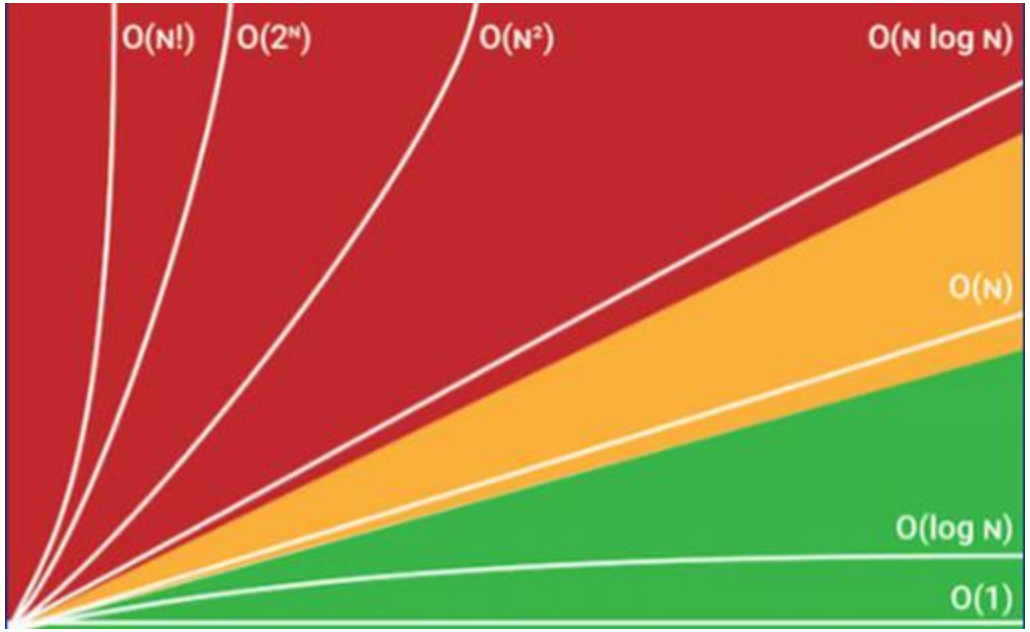
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Анализ сложности алгоритмов



Алгоритм

это конечная последовательность чётко определенных, 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.

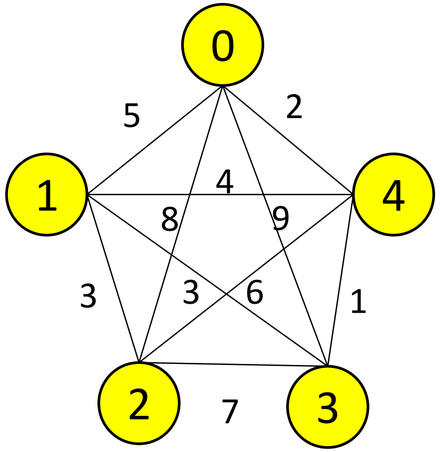
В рамках нашей дисциплины мы будем работать с **детерминированными алгоритмами**.

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 коммивояжёра

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.

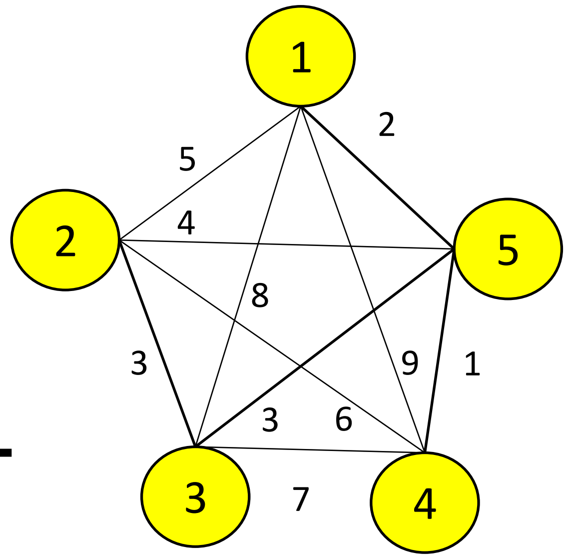


Задача разрешима, так как любую индивидуальную задачу можно решить, найдя наилучший обход среди конечного множества обходов. Если алгоритм будет перебирать все обходы, вычислять их длины и выбирать кратчайший обход, то реализация этого алгоритма на вычислительной машине потребует порядка $\boxed{\frac{(n-1)!}{2}}$ **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!	$2{,}65253 \cdot 10^{32}$

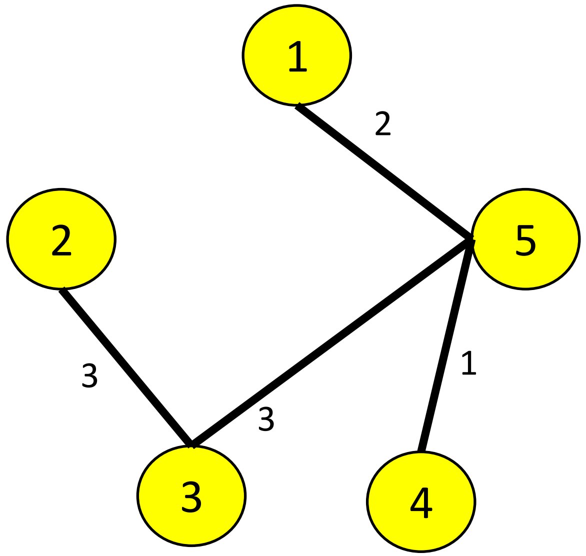
Минимальное остовное дерево

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 $\boxed{\boldsymbol{n^{n-2}}}$ 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.**

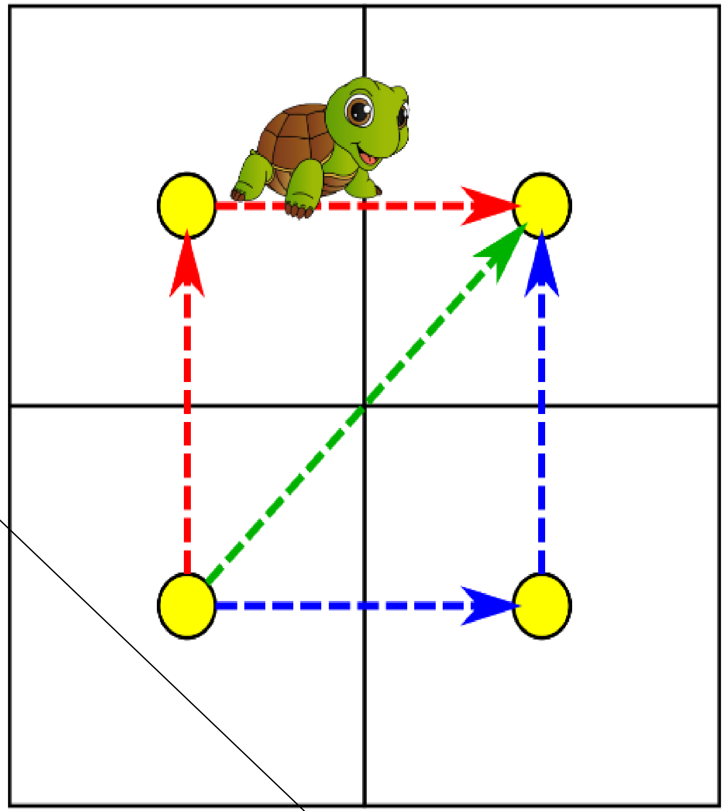
Имя входного файла: стандартный ввод
Имя выходного файла: стандартный вывод
Ограничение по времени: **1 с**
Ограничение по памяти: **64 МБ**

$$0 \le n, m \le 10^4$$

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.



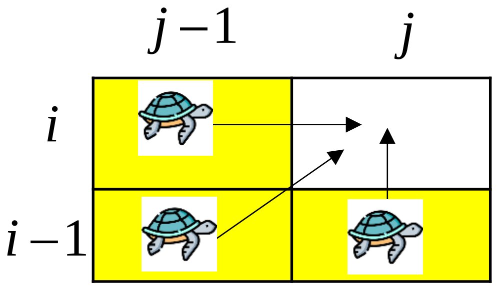
Формат входных данных

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

Формат выходных данных

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

	1	2	3	4
5	1	9	41	129
4	1	7	25	63
3	1	5	13	25
2	1	3	5	7
1	1	1	1	1



$$f[i, j] = f[i, j-1] + f[i-1, j-1] + f[i-1, j]$$

Ограничение по памяти: 64 Мб.

1 Мб $= 10^6$ байт

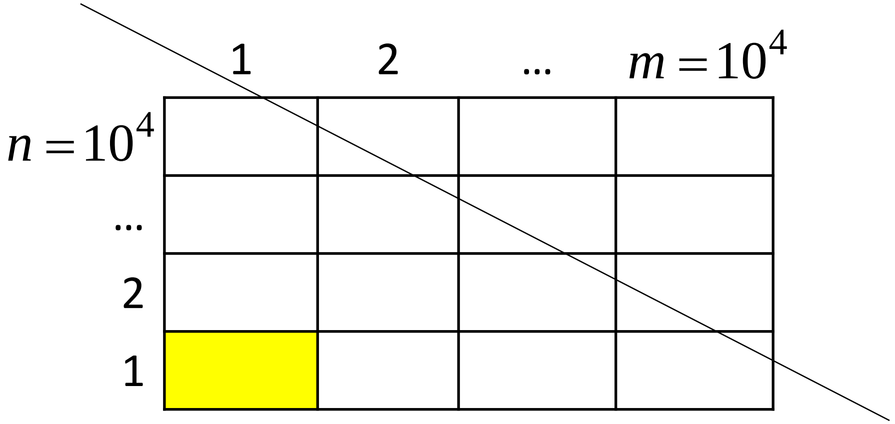
64 Мбайт = $64 \cdot 10^6$ байт

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

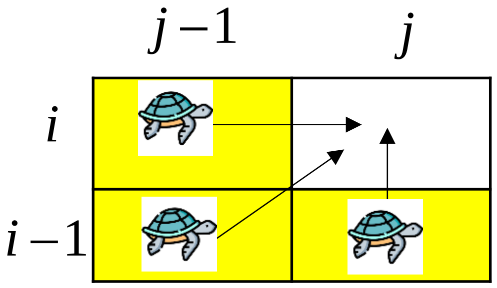
Ограничение по памяти в 64 Мб позволит выделить в памяти участок не боле, чем для

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

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



	1	2	3	4
5	1			
4	1	7	?	
3	1	5	13	25
2	1	3	5	7
1	1	1	1	1



$$f\left[i,j\right]=f\left[i,j-1\right]+f\left[i-1,j-1\right]+f\left[i-1,j\right]$$

1	5	13	25

1	3	5	7

1	1	1	1

Тактовая частота процессора

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

$$4 \cdot 10^{9} \quad \text{тактовых циклов (операций)}$$

1 тактовый цикл (1 операция) выполняется за

$$\tfrac{1}{4} \cdot 10^{-9}\,\text{секунду} = \tfrac{1}{4}\,\text{наносекунду}$$

Процессор суммирует два 64-х битных числа за 1 тактовый цикл, т.е. за ¼ 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**

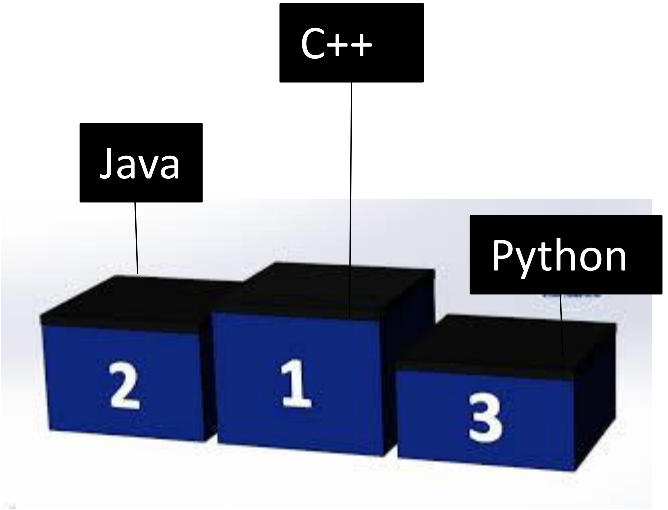
div, idiv (64-битный делитель): **15 тактов**

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.

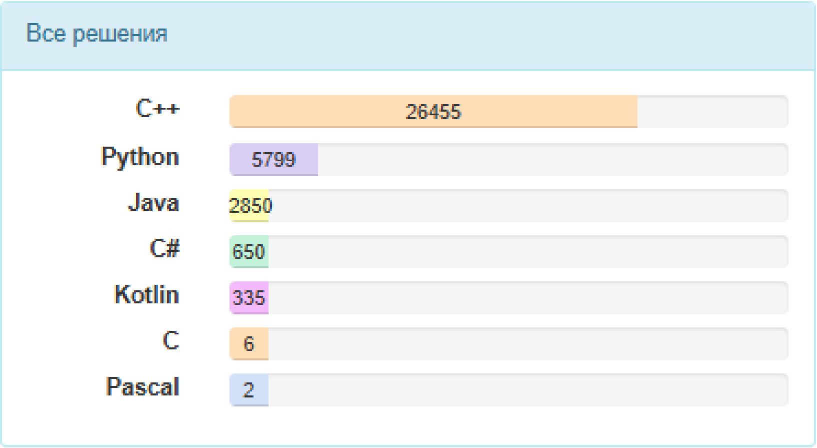
C++ уже на этапе компиляции переводит инструкции программы в хорошо оптимизированный машинный код.

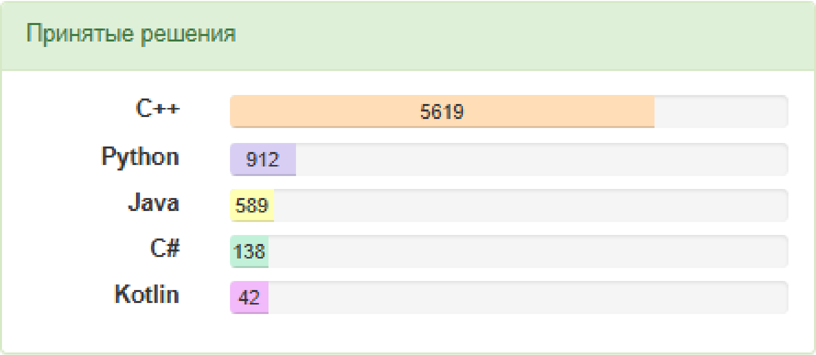
Python выполняет преобразования в машинный код на этапе выполнения со значительными накладными расходами.



Число **принятых** решений на 26.01.2025	
C++	**154 394**
Java	**15 616**
Python	**8 742**
Pascal	**3 217**
Delphi	**2 622**
C#	**2 417**
C	**1 997**
Kotlin	**433**

<u>Данные с 1.09.2024-26.01.2025 г.</u>





Вердикты

WA	Неправильный ответ	13454
OK	Принято	7300
RTE	Ошибка во время выполнения	4800
TLE	Нарушен предел времени	4382
CE	Ошибка компиляции	3282
PE	Ошибка представления	2120
MLE	Нарушен предел памяти	746
ILE	Нарушен предел ожидания	2

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

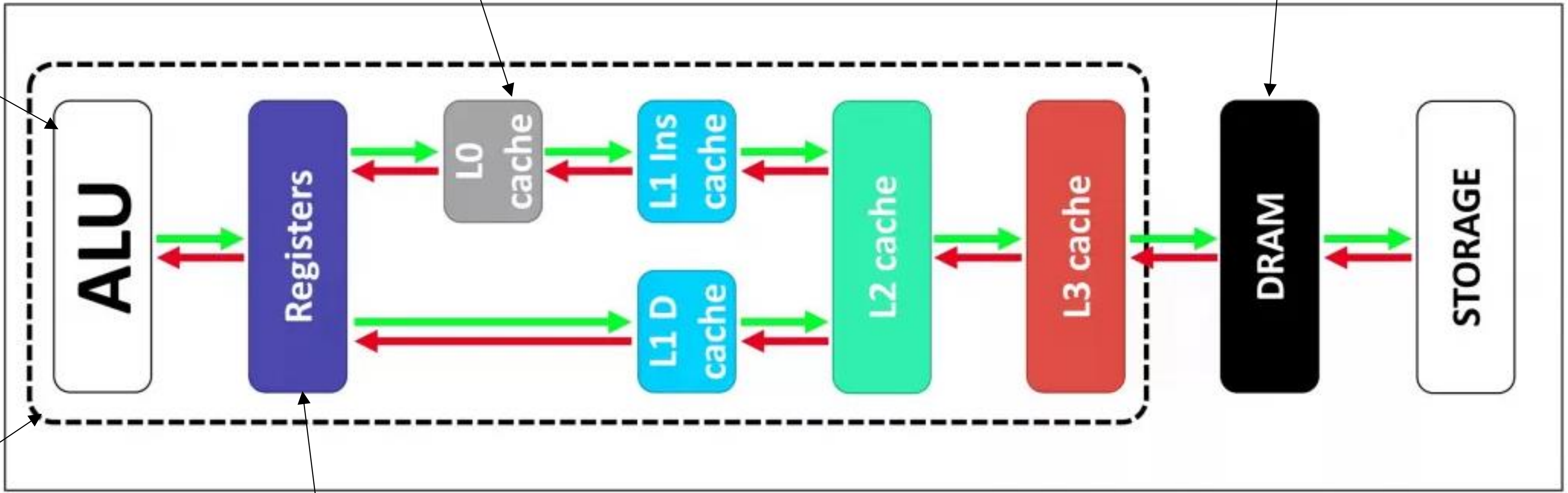
Процессоры имеют многоуровневую систему кэшей (L1, L2, L3), постоянно сохраняющую те или иные ячейки для более быстрого доступа. В зависимости от того, закэшировал ли процессор нужную ячейку, время доступа к данным может отличаться в десятки раз.

Уровни (**L**ayers) памяти компьютера

оперативная память (энергозависимая)
Dynamic **R**andom **A**ccess **M**emory
(микроскопические кондерсаторы)

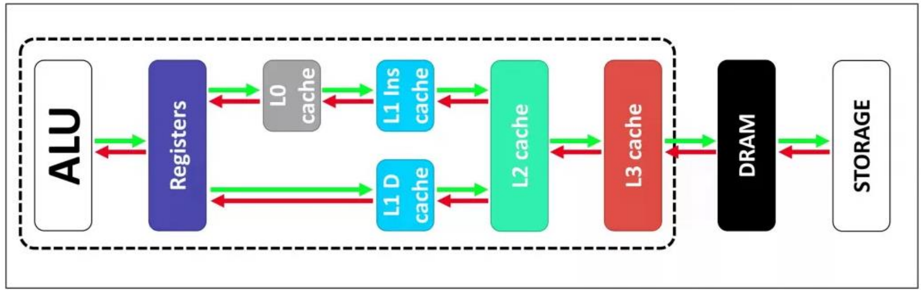
блок памяти всего
на 1 500 µops (микроопераций)

arithmetic
logic units
(арифметико-
логическое
устройство)



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

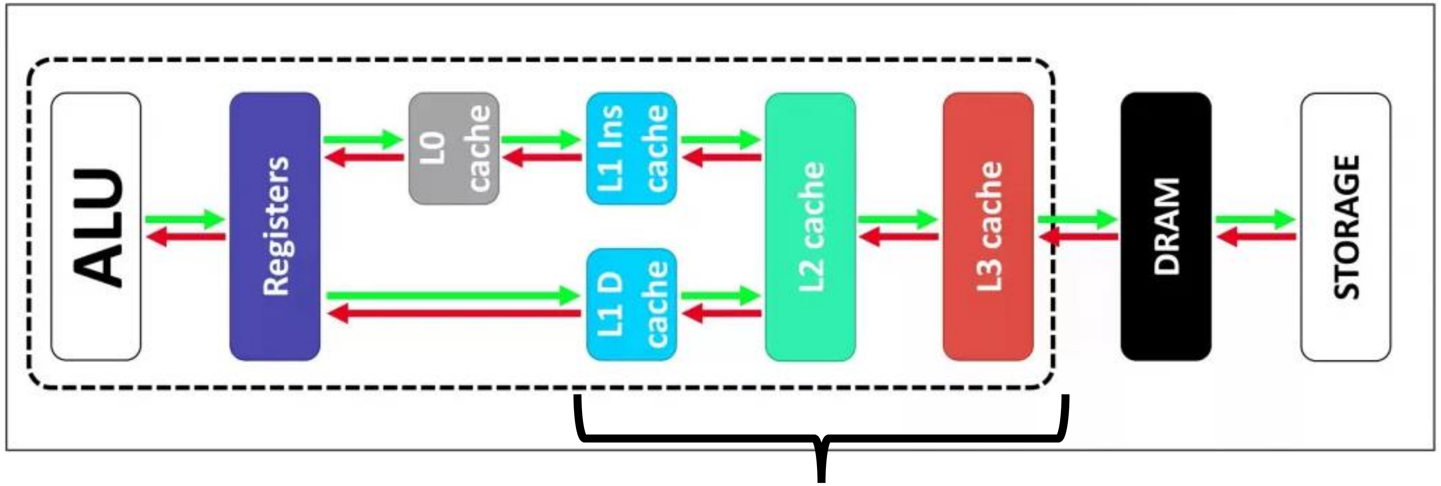


Доступ к данным

	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), 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: 64-256 Кб
два блока:
кэш-команд **Instruction cache**,
кэш-данных **Data cache**
быстрая (почти в 100 раз быстрее DRAM, но в среднем в 5 раз медленнее регистров), маленькая по объему

L2: от 256 Кб (до 8 Мб)
примерно в 2 раза медленнее, чем L1

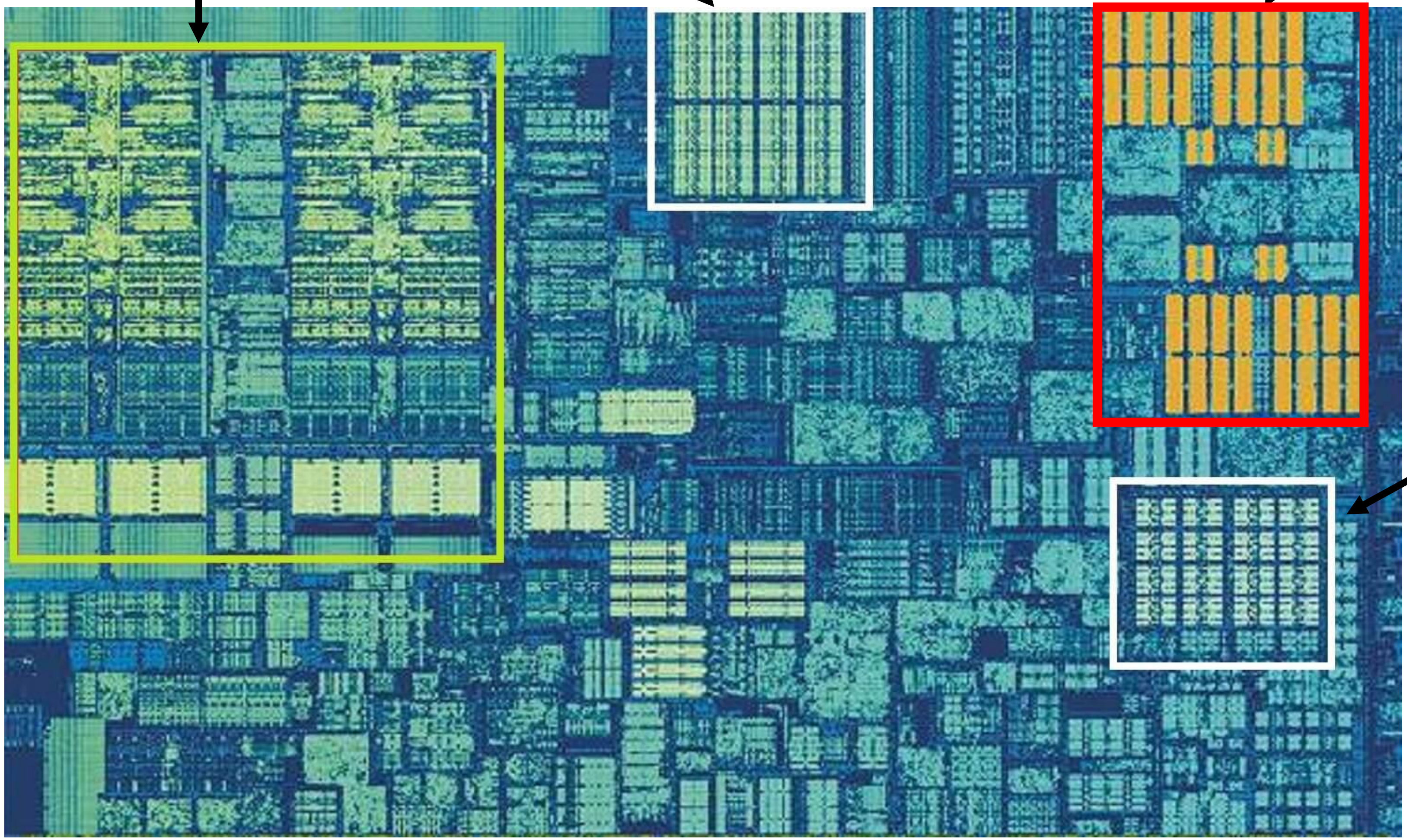
больше по объему, чем L1

L3: 4 Мб (до 50 Мб)
(для многоядерных процессоров доступ к L3 разрешен из любого ядра ЦП), самая медленная, но ее много

Arithmetic Logic Units
Registers

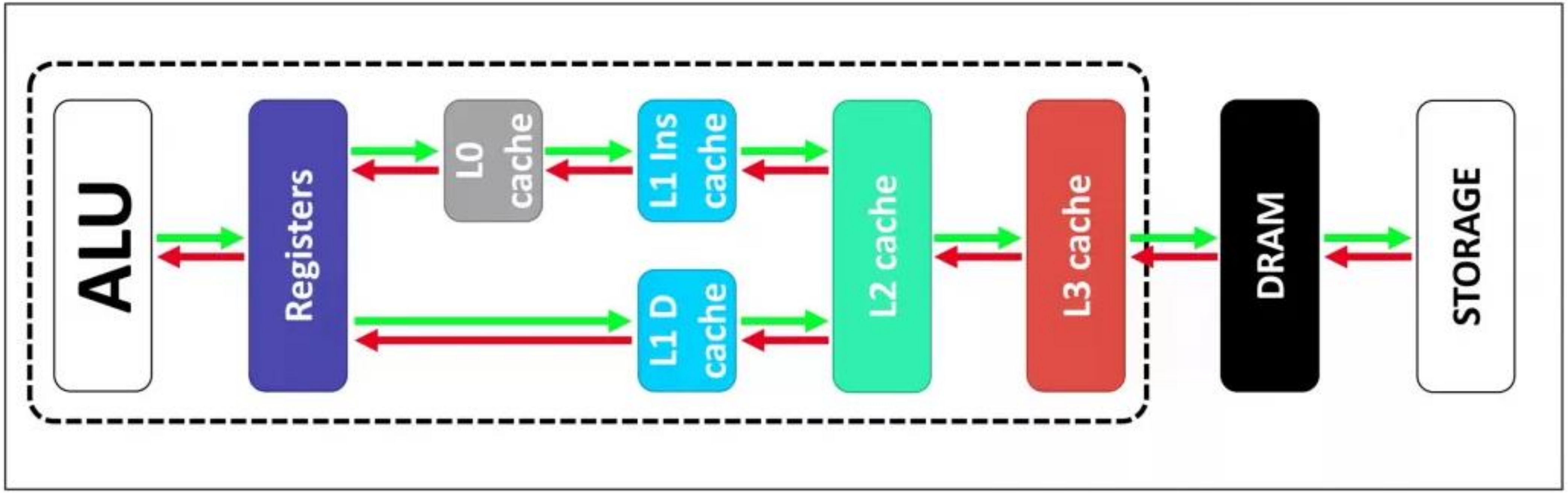
L1 (Data cache, 32Кб)

L2 (Data cache, до 512 Кб)



L1 (Instruction cache, 32Кб)

Одно ядро процессора Intel Skylake



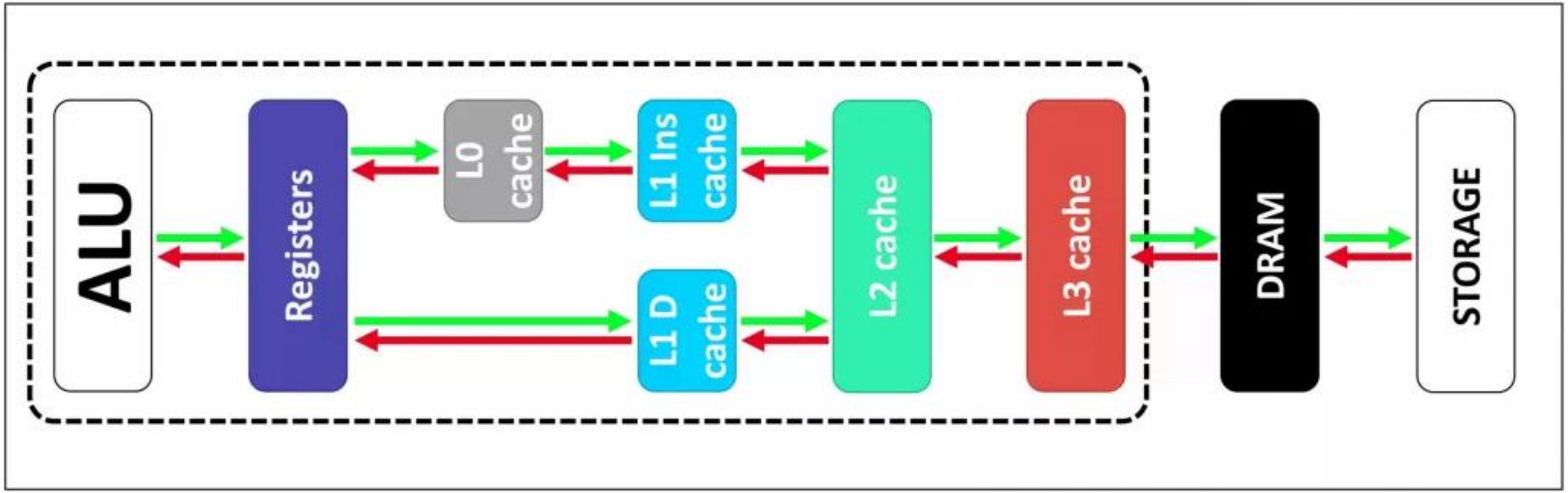
В кэш помещаются информация, к которой часто обращается процессор (информация из ОЗУ помещается в кэш, а потом к ней обращается процессор).

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.

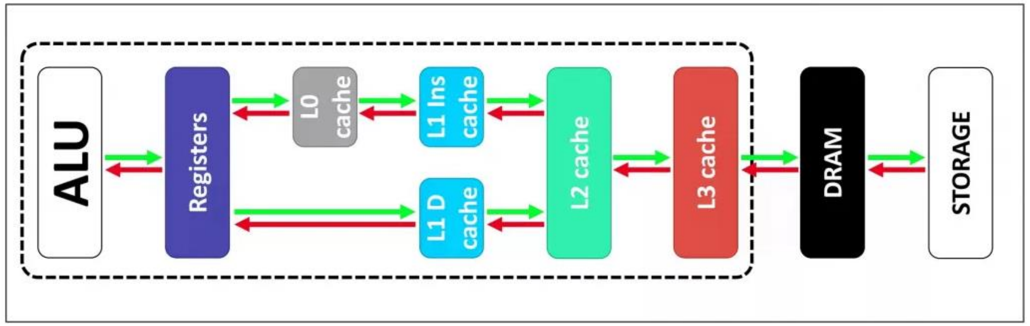


Взаимодействие уровней кэш-памяти между собой (сложный процесс, в ходе которого используются разные технологии и математические алгоритмы):

эксклюзивное (уникальность информации);

инклюзивное (дублирование информации);

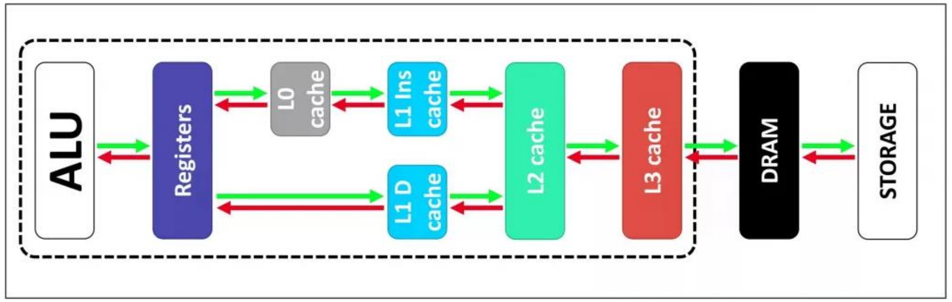
Эксклюзивная кэш-память предполагает уникальность информации, находящейся в L1 и L2.



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 **считывании 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 удаления наиболее "старых данных" - **LRU** (Least-Recently Used)), а затем, передается в процессор.
Если нужная информация не найдена и в кэше второго уровня, то она считывается из ОЗУ по схеме, описанной далее.

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

Грубо говоря ….

Если вы пишете на C ++ и решаете типичную алгоритмическую задачу, то можете предположить, что за 1 секунду вы сможете выполнить ~ $\mathbf{10^8}$ 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, **~** $\mathbf{10^7}$.

Если операции простые, а обращения к памяти локальные или последовательные, то вы сможете за 1 секунду выполнить **~** $\mathbf{10^9}$ 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	[redacted]
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 $n \to +\infty$ 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**.

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

Предположим, что за сутки $(= 24\text{ ч} \cdot 60\text{ мин} \cdot 60\text{ с} = 86\,400\text{ с})$ современная ЭВМ может выполнить $\approx 10^{12}$ абстрактных операций.

Если алгоритму требуется для решения задачи выполнить $\boldsymbol{f(n) = n^3}$ 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
$\tilde{n} = \sqrt[3]{10 \cdot 10^{12}} = \boldsymbol{\sqrt[3]{10} \cdot n} \approx 2{,}15 \cdot n$.

Если алгоритму требуется для решения задачи выполнить $\boldsymbol{f(n) = \beta^n}$ операций, то за сутки ЭВМ выполняет $\beta^n = 10^{12}$ операций и успевает 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
$\tilde{n} = log_\beta(10 \cdot 10^{12}) = log_\beta 10 + log_\beta 10^{12} = \boldsymbol{log_\beta 10 + n}$.

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}	$\mathbf{10} \cdot 10^{12}$
$n \cdot \log n$	$0{,}948 \cdot 10^{11}$	$\mathbf{8,7} \cdot 10^{11}$
n^2	10^6	$\mathbf{3,16} \cdot 10^6$
n^3	10^4	$\mathbf{2,15} \cdot 10^4$
$10^8 \cdot n^4$	10	$\mathbf{1{,}8} \cdot 10$
2^n	40	$\mathbf{3} + 40 = 43$
10^n	12	$\mathbf{1} + 12 = 13$
$n^{\log n}$	79	$\mathbf{16} + 79 = 95$
$n!$	14	$\mathbf{1} + 14 = 15$

В общем случае справедливы следующие утверждения.

1. Если время работы алгоритма оценивается функцией $\boldsymbol{f(n) = log_2 n}$, то увеличение скорости вычислений ЭВМ $\boldsymbol{v}$ раз позволит решить за такое же время задачу, размер которой в $\boldsymbol{2^v}$ **раз больше**.

2. Если время работы алгоритма оценивается функцией $\boldsymbol{f(n) = n^\alpha}$, то увеличение скорости вычислений ЭВМ в $\boldsymbol{v}$ раз позволит решить за такое же время задачу, размер которой **в** $\boldsymbol{\sqrt[\alpha]{v}}$ **раз больше** (мультипликативное увеличение размера задачи, которую алгоритм может решить за фиксированное время).

3. Если время работы алгоритма оценивается функцией $\boldsymbol{f(n) = \beta^n}$, то увеличение скорости вычислений ЭВМ в $\boldsymbol{v}$ раз позволит решить за такое же время задачу, размер которой лишь на $\boldsymbol{log_\beta v}$ **больше** (аддитивное увеличение размера задачи, которую алгоритм может решить за фиксированное время).

ВЫВОД

Использование алгоритмов, время работы которых ограничено функцией вида $\boldsymbol{n^\alpha}$, позволяет кратно 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.**

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 $\boldsymbol{l}$.

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

1	1	1	1	1
4	3	2	1	0

$\boldsymbol{n}$=5 ($l = 5$)

1	1	1	1	1	1
5	4	3	2	1	0

$\boldsymbol{n}$=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 0 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

1	0	1
2	1	0

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

1	1	0
2	1	0

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

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?

1		0	1	0
$l-1$	…	2	1	0

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:

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

<u>Сведения из математики:</u>

$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$

На вход поступает натуральное число n ($\mathbb{N} = \{1,2,3, \ldots\}$) и массив чисел: $a_1, a_2, \ldots, a_n$.

Какова битовая длина входа в бинарной системе кодирования?

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

Если предположить, что $\max\limits_{1 \le i \le n} \lfloor \log_2 |a_i| + 1 \rfloor = const$ (где, например, $const = 32$), то полагается, что каждый элемент массива занимает $const$ бит (в этой модели для элементов массива не предполагается компактная форма записи).

Тогда размерность задачи:

$$l = \lfloor \log_2 n + 1 \rfloor + const \cdot n.$$

В данной модели для увеличения объема входных данных будем устремлять $n \to +\infty$, 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 $\max\limits_{1\le i\le n}\lfloor \log_2 |a_i| + 1\rfloor = const$, 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.

Временна́я сложность алгоритма

это функция, которая задаче размерности l ставит в соответствие время $\boldsymbol{T(l)}$, 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

данные 1	данные 2	данные 3		данные m
1-й запуск алгоритма	1-й запуск алгоритма	1-й запуск алгоритма		1-й запуск алгоритма
2-й запуск алгоритма	2-й запуск алгоритма	2-й запуск алгоритма	…	2-й запуск алгоритма
……	……	……		……
K_1-й запуск алгоритма	k_2 -й запуск алгоритма	k_3 -й запуск алгоритма		k_m-й запуск алгоритма

$$\underbrace{E_1 = \frac{\sum_{k_1}}{k_1} \qquad E_2 = \frac{\sum_{k_2}}{k_2} \qquad E_3 = \frac{\sum_{k_3}}{k_3} \qquad \ldots \qquad E_m = \frac{\sum_{k_m}}{k_m}}$$

$$E = \max_{1 \le i \le m} E_i$$

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 $\boldsymbol{n}$ уникальных элементов и некоторое число $\boldsymbol{x}$.
Необходимо определить есть ли число $\boldsymbol{x}$ в массиве.
Оценить **среднее время работы алгоритма** последовательного поиска по всем возможным наборам входных данных размерности l .

1-я группа: искомый элемент $\boldsymbol{x}$ стоит на 1-й позиции $\quad t_1 = 1$

2-я группа: искомый элемент $\boldsymbol{x}$ стоит на 2-й позиции $\quad t_2 = 2$

3-я группа: искомый элемент $\boldsymbol{x}$ стоит на 3-й позиции $\quad t_3 = 3$

…

n-я группа: искомый элемент $\boldsymbol{x}$ стоит на n-й позиции $\quad t_n = n$

$(n+1)$-я группа: искомого элемента $\boldsymbol{x}$ нет $\quad t_{n+1} = n$

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

$$A(l) = \sum_{i=1}^{n+1} p_i \cdot t_i = \frac{1}{n+1}(1 + 2 + \ldots + n + n) =$$

$$= \frac{1}{n+1} \cdot \left(\frac{1+n}{2} \cdot n + n\right) = \frac{n}{2} + \frac{n}{n+1} = \frac{n}{2} + 1 - \frac{1}{n+1} \approx \frac{n}{2} + 1, \quad \text{при } n \to +\infty \qquad C_1 \cdot n \le l \le C_2 \cdot n$$

Модель абстрактного вычислительного устройства

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

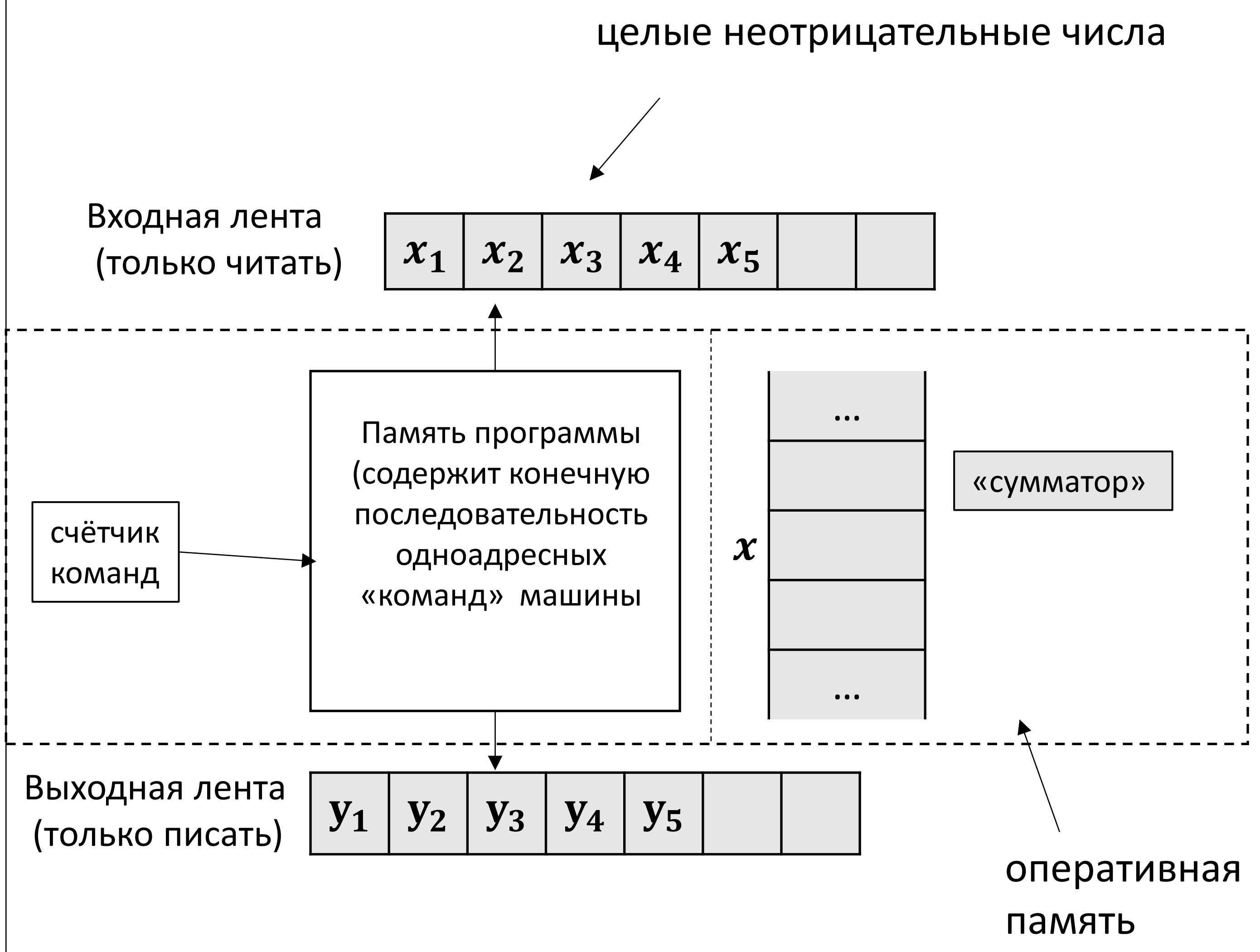
Как определить время выполнения и объём памяти программной реализации алгоритма на некотором **абстрактном вычислительном устройстве**?

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;

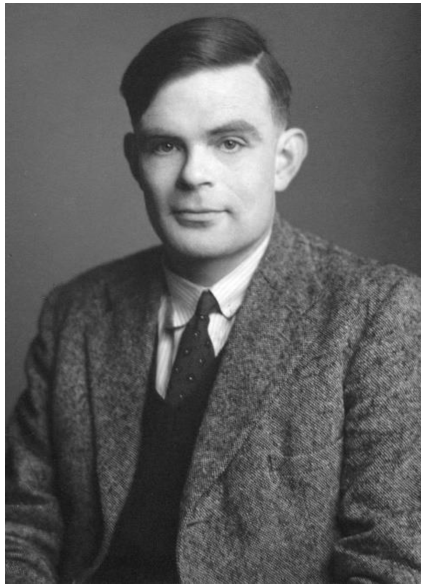
РАМ - вычислительная машина с одним сумматором, в котором команды программы не могут изменять сами себя, состоит из:

- ✓ **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** (МТ) (предложена в качестве абстрактной модели вычислений А. Тьюрингом в 1936 г.)

Для МТ 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 $\boldsymbol{x}$	**считывает** в ячейку $\boldsymbol{x}$ очередной символ из входной ленты
WRITE $\boldsymbol{x}$	**запис**ь содержимого ячейки $\boldsymbol{x}$ на выходную ленту
LOAD $\boldsymbol{x}$	**загрузит**ь операнд **из** ячейки $\boldsymbol{x}$ памяти **в сумматор**
STORE $\boldsymbol{x}$	**отослать** содержимое **из сумматора** в указанную ячейку $\boldsymbol{x}$ памяти
ADD $\boldsymbol{x}$ **SUB** $\boldsymbol{x}$	**сложит**ь содержимое ячейки $\boldsymbol{x}$ и **сумматора** **вычесть** содержимое ячейки $\boldsymbol{x}$ из **сумматора**
MUL $\boldsymbol{x}$ **DIV** $\boldsymbol{x}$	**умножит**ь содержимое **сумматора и** ячейки $\boldsymbol{x}$ **разделить** число **из сумматора** на содержимое ячейки $\boldsymbol{x}$
JGTZ M **JZERO** M **JUMP** M **HALT** **WAIT**	**перехо**д 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**

<u>Операнды могут быть одного из следующих типов:</u>

$=x$ – 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 $\mathbf{1}$
LOAD $\mathbf{1}$
READ $\mathbf{2}$
ADD $\mathbf{2}$
STORE $\mathbf{3}$
WRITE $\mathbf{3}$

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
LOAD 1
READ 2
ADD 2
STORE 3
WRITE 3

сумма двух чисел
(логарифмическая мера сложности)

время выполнения = 516?;
101+101+11+101+101+101

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 = 536?;
101+101+11+101+111+111

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

Асимптотики

Рассмотрим две неотрицательные функции $\boldsymbol{f(n)}$ и $\boldsymbol{g(n)}$.

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

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

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

$$\lim_{n\to+\infty}\frac{f(n)}{g(n)}\le const$$

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

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

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

$$\lim_{n\to+\infty}\frac{f(n)}{g(n)}\ge const$$

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

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

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

Математические обозначения (нотации) $\mathbf{O}, \boldsymbol{\Omega}, \boldsymbol{\Theta}$ введены для функций по аналогии с тем, как для чисел были введены обозначения ($\lceil \quad \rceil$ округление к большему/вверх/англ. *ceiling* досл. «потолок»; $\lfloor \quad \rfloor$ –округление к меньшему/вниз/ англ. *floor* досл. «пол»).

Асимптотики $\mathbf{O}\big(\boldsymbol{f}(\boldsymbol{n})\big), \boldsymbol{\Omega}\big(\boldsymbol{f}(\boldsymbol{n})\big), \boldsymbol{\Theta}(\boldsymbol{f}(\boldsymbol{n}))$ –

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

Произносится:

$g(n) \in \mathrm{O}\big(f(n)\big)$ g от n равно **о большое** от f от n

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

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

здесь и далее $n \to +\infty$

Асимптотика
$\boldsymbol{\mathrm{O}(f(n))}$ –
это множество функций, которые растут не быстрее, чем функция $f(n)$.

$$4 \cdot n^3 \in \mathrm{O}\left(2^n\right)$$

$$4 \cdot n^3 \in \mathrm{O}\left(n^3\right)$$

Асимптотика
$\boldsymbol{\Omega(f(n))}$ –
это множество функций, которые растут не медленнее, чем функция $f(n)$.

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

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

Асимптотика
$\boldsymbol{\Theta(f(n))}$ –
это множество функций, которые растут не быстрее и не медленнее, чем функция $f(n)$.

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

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

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

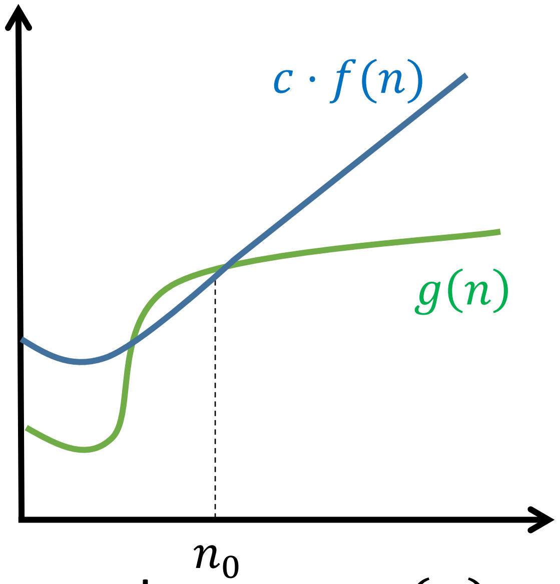
$$\boldsymbol{g}(\boldsymbol{n}) \in \mathbf{O}(\boldsymbol{f}(\boldsymbol{n})),$$

но мы будем также употреблять эквивалентную запись:

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

Запишем математически определения асимптотик.
Пусть $f(n)$ и $g(n)$ – функции, определённые на множестве целых положительных чисел и принимающие положительные действительные значения.

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



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

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

$$4 \cdot n^3 \in \mathrm{O}(2^n)$$
$$4 \cdot n^3 \in \mathrm{O}(n^3 \cdot \log n)$$
$$4 \cdot n^3 \in \mathrm{O}(n^3)$$

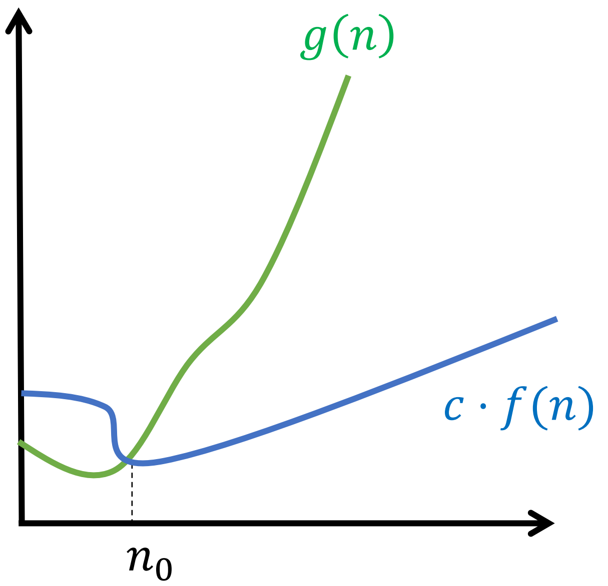
Будем писать $g(n) \in \mathrm{o}(f(n)) \;\Leftrightarrow \forall \mathrm{c}, \exists n_0 > 0 \colon \forall n \geq n_0$ выполняется $0 \leq g(n) < c \cdot f(n)$.

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

для функции $g(n) = 4 \cdot n^3$ функция $f(n) = n^3$ является точной оценкой: $4 \cdot n^3 = \mathrm{O}(n^3)$, $4 \cdot n^3 \neq \mathrm{o}(n^3)$

ФПМИ БГУ

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



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

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

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

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

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

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

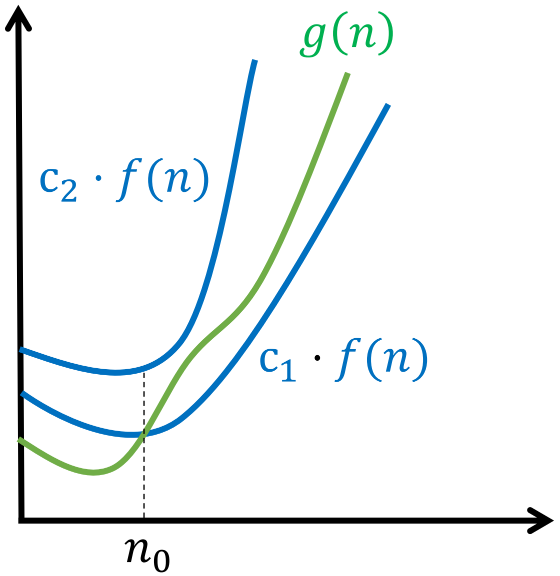
Будем писать $g(n) \in \omega(f(n)) \Leftrightarrow \forall \mathrm{c}, \exists n_0 > 0: \forall n \geq n_0$ выполняется $0 \leq \mathrm{c} \cdot f(n) < g(n).$

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 $f(n) \in \mathrm{O}(g(n))$.

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



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

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

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

$\exists$ такие константы $c_1 > 0, c_2 > 0, n_0 > 0,$ что $\forall n \geq n_0$ выполняются неравенства:

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

т.е. $f(n) = \Theta\left(n^d\right).$

Смысл нижней оценки, полученной **для алгоритма** $\boldsymbol{A}$ с вычислительной сложностью в худшем случае $T_A(l)$:

если $\mathbf{T}_{\boldsymbol{A}}(\boldsymbol{l}) = \boldsymbol{\Omega}\left(\boldsymbol{f}_{\mathbf{1}}(\boldsymbol{l})\right)$, то алгоритм $\boldsymbol{A}$ 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 $\mathbf{T}_{\boldsymbol{A}}(\boldsymbol{l}) = \mathbf{O}\left(\boldsymbol{f}_{\mathbf{2}}(\boldsymbol{l})\right)$, то алгоритм $\boldsymbol{A}$ 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.

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

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

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

Сведения из математики

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

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

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

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

3) Полилогарифмическая функция

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

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:

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

3) Функция $l!$ асимптотически возрастает быстрее, чем 2^l, но медленнее, чем функция l^l:

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

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

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^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

Например, при n ≤ 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 1.

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

Необходимо вычислить сумму элементов массива: $\boldsymbol{S} = \boldsymbol{a_1} + \boldsymbol{a_2} + \dots + \boldsymbol{a_n}$.

Оцените вычислительную сложность алгоритма.

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

$$l = \lfloor log_2 n + 1 \rfloor + C_1 \cdot n,$$

где C_1 —константа (например, 32 бита или 64 бита).

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

поэтому, при $n \to +\infty$ верно:

$$l = \lfloor log_2 n + 1 \rfloor + C_1 \cdot n <$$
$$< n + C_1 \cdot n = (C_1 + 1) \cdot n.$$

Справедливы неравенства:

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

откуда

$$\frac{\mathbf{1}}{\boldsymbol{C_1 + 1}} \cdot \boldsymbol{l} < \boldsymbol{n} < \frac{\mathbf{1}}{\boldsymbol{C_1}} \cdot \boldsymbol{l}.$$

Таким образом $l = \Theta(n)$ $(n = \Theta(l))$.

Предполагаем, что в результате суммирования не произойдёт переполнения: все промежуточные результаты вычислений вмещаются в C_1 бит.

Тогда сложение двух чисел будет выполнено:

- ✓ за время 1, если в РАМ <u>равномерная мера</u> измерения времени выполнения одной операции, и в этом случае вычислительная сложность алгоритма:

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

- ✓ за время C_1, если в РАМ <u>логарифмическая мера</u>, и в этом случае вычислительная сложность алгоритма:

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

***Ответ**: алгоритм – полиномиальный*

Задача 2.

На вход поступает натуральное число $\boldsymbol{n}$.

Нужно вычислить: $\boldsymbol{n}! = \mathbf{1} \cdot \mathbf{2} \cdot \ldots \cdot \boldsymbol{n}$.

Оцените вычислительную сложность алгоритма.

$\boldsymbol{n! = 1 \cdot 2 \cdot \ldots \cdot n}$

Входные данные: n.

Размерность задачи: унарная система кодирования $l = n$.

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

Умножение двух чисел будет выполнено

- ✓ за время 1, если в РАМ равномерная мера, и тогда **вычислительная сложность алгоритма**:

$$\boldsymbol{T(l) = 1 \cdot l = \Theta(l)}.$$

- ✓ за время, равное максимуму числа бит операндов операции, если в РАМ логарифмическая мера, и тогда **вычислительная сложность алгоритма**:

$$T(l) = (\lfloor \log_2 1 + 1 \rfloor + \lfloor \log_2 2 + 1 \rfloor + \lfloor \log_2 3 + 1 \rfloor) + (\lfloor \log_2 3! + 1 \rfloor + \lfloor \log_2 4! + 1 \rfloor + \cdots + \\ + \lfloor \log_2 l! + 1 \rfloor) \leq (1 + 2 + 2) + (\log_2 3! + \log_2 4! + \cdots + \log_2 l! + (n - 3)) \leq \\ \leq \sum_{i=3}^{l} \log_2 i! + (l - 3) + 5 \,.$$

Учитывая, что $\log_2 x! = \Theta(x \cdot \log x)$, получим

$$T(l) \leq (3 + \cdots + l) \cdot const_1 \cdot \log_2 l + (l - 3) + 5 \leq const_2 \cdot l^2 \cdot \log_2 l + (l - 3) + 5.$$

$$\boldsymbol{T(l) = O\left(l^2 \cdot log_2 l\right)}.$$

***Ответ**: алгоритм – полиномиальный.*

$\boldsymbol{n! = 1 \cdot 2 \cdot \ldots \cdot n}$

Входные данные: n.

Размерность задачи: бинарная система кодирования входных данных

$$\boldsymbol{l = \lfloor log_2\, n + 1 \rfloor}.$$

Размерности l в задаче соответствует несколько входов:

$l-1$	…	2	1	0

$$\boldsymbol{2^{l-1} \le n \le 2^l - 1.}$$

Наибольшее число мультипликативных операций (умножение) для задачи размерности l у входных данных:

$$\boldsymbol{n = 2^l - 1.}$$

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.

Вычислительная сложность алгоритма:

число мультипликативных операций умножения равно $(2^l - 1)$.

Умножение двух чисел будет выполнено:

- ✓ за время 1, если в РАМ равномерная мера измерения времени выполнения операции, и тогда вычислительная сложность алгоритма:

$$\boldsymbol{T(l) = 1 \cdot (2^l - 1), T(l) = \Theta\left(2^l\right);}$$

- ✓ за время равное максимуму числа бит операндов, если логарифмическая мера, и тогда

$$\boldsymbol{T(l)} = (\lfloor \log_2 1 + 1 \rfloor + \lfloor \log_2 2 + 1 \rfloor + \lfloor \log_2 3 + 1 \rfloor) +$$
$$+ \left(\lfloor \log_2 3! + 1 \rfloor + \lfloor \log_2 4! + 1 \rfloor + \cdots + \left\lfloor \log_2 \left(2^l - 1\right)! + 1 \right\rfloor\right) >$$
$$> 5 + \left(\log_2 3! + \ldots + \log_2 \left(2^l - 1\right)!\right)$$
$$[\text{т. к. } \log_2(x!) = \Theta(x \cdot \log x)]$$
$$> 5 + (3 + \cdots + (2^l - 1)) \cdot \log_2 3 \cdot const_1 =$$
$$\boldsymbol{= 5 + (2^{l-1} + 1) \cdot (2^l - 3) \cdot \log_2 3 \cdot const_1.}$$

Ответ: *алгоритм – экспоненциальный.*

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

Задача 3

Для заданного натурального числа $\boldsymbol{n}$ проверить, **является ли число простым.**

Алгоритм:

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

Оценить вычислительную сложность алгоритма.

Оценку вычислительной сложности алгоритма проведём **по числу** мультипликативных **операций деления**.

При унарной системе кодировки входных данных **размерность задачи**

$$\boldsymbol{l = n}.$$

Выписать точную формулу для **временной сложности** $\boldsymbol{T(l)}$ не получится.

$\boldsymbol{l}$	111	112	113	114	115	116	117	118	119	120
$\boldsymbol{T(l)}$	**2** l:2 - нет l:3 - да	**1** l:2 - да	**9** l: 2 - нет l: 3 –нет … l:10 - нет	**1** l: 2 - да	**4** l: 2 - нет l: 3 –нет l: 4 - нет l: 5 –да	**1** l: 2 - да	**2** l:2 - нет l:3 - да	**1** l: 2 - да	**6** l: 2 - нет l: 3 –нет l: 4 - нет l: 5 –нет l: 6 –нет l: 7 –да	**1** l: 2 - да

Функция $T(l)$ не является монотонной, поэтому можно лишь показать, что, при $n \to +\infty$, для $T(l)$ справедлива оценка сверху:

$$\boldsymbol{T(l) \le \left\lfloor \sqrt[2]{l} \right\rfloor - 1}.$$

Алгоритм при унарной системе кодировки входных данных и равномерной мере сложности для РАМ-машины является *полиномиальным*.

Размерность задачи: бинарная система кодирования входных данных, $\boldsymbol{l = [\log_2 n + 1]}$.
Задаче размерности l соответствуют несколько входов (это множество натуральных чисел, компактная запись которых в двоичном представлении содержит l бит.
Все возможные входы для задачи размерности l описываются неравенствами:

$$\boldsymbol{2^{l-1} \le n \le 2^l - 1}.$$

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

В таблице показано, сколько операций в худшем случае будет выполнено алгоритмом для задачи размерность l при бинарной системе кодирования входа.

$\boldsymbol{l}$ бит	**2** – –	**3** – – –	**4** ––– –	**5** ––– –	**6** ––––– –	**7** –––––– –
диапазон входов для задачи размерности l	$2-3$	$4-7$	$8-15$	$16-31$	$32-63$	$64-127$
число из диапазона, для которого самые большие затраты	3	7	13	31	61	127
$\boldsymbol{T(l)}$	**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 для задачи размерности l, тогда число выполненных делений при проверке этого числа известно и оно равно $\left\lfloor \sqrt[2]{p_l} \right\rfloor - 1$.

Так как $2^{l-1} \le p_l < 2^l$, то $\left\lfloor \sqrt[2]{p_l} \right\rfloor - 1 \ge \left\lfloor 2^{\frac{l-1}{2}} \right\rfloor - 1$, откуда получаем, что

$$T(l) \ge \lfloor \sqrt[2]{p_l} \rfloor - 1 \ge \left\lfloor 2^{\frac{l-1}{2}} \right\rfloor - 1 > 2^{\frac{l-1}{2}} - 2 = 2^{\left(\frac{l}{2} - \frac{1}{2}\right)} - 2 = \frac{2^{\frac{l}{2}}}{\sqrt[2]{2}} - 2 = \left(\frac{1}{\sqrt[2]{2}} - \frac{2}{2^{l/2}}\right) \cdot 2^{l/2} = \text{C}_1 \cdot 2^{l/2},$$

где при $4 \le l \le +\infty$, выполняется $0 < \left(\frac{1}{\sqrt[2]{2}} - \frac{1}{2}\right) \le \text{C}_1 < \frac{1}{\sqrt[2]{2}}$.

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

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

Таким образом, при бинарной системе кодирования входных данных задачи получены оценки сверху и снизу для временной сложности алгоритма:

$$\mathbf{C_1 \cdot 2^{l/2} < T(l) < 2^{l/2}}, \textbf{ при } \boldsymbol{l \ge 4}.$$

Вывод: алгоритм при бинарной системе кодирования входных данных является экспоненциальным:

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

Псевдополиномиальные алгоритмы

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

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

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 4

На вход поступает натуральное число $\boldsymbol{n}$ и массив натуральных чисел $\boldsymbol{a_1, a_2, \ldots, a_n}$. Подсчитать частоту встречаемости элементов массива.

Алгоритм

1) найдем максимальный элемент в массиве, предположим, что это число $\boldsymbol{M = \max\limits_{1 \le i \le n} a_i}$;
2) выполним **цикл** по $\boldsymbol{i}$ от **1** до $\boldsymbol{M}$ и для каждого значения $\boldsymbol{i}$ подсчитаем частоту встречаемости числа $\boldsymbol{i}$ в исходном массиве;

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

$M = \max\limits_{1 \le i \le 6} a_i = 5$

$i = 1..M$

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

Размерность задачи:

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

В нашей модели в предположении, что $\max\limits_{1 \le i \le n} \lfloor \log_2 a_i + 1 \rfloor = const$ (например, $const = 32$), полагаем, что каждый из элементов массива занимает число бит $= const$.

В этом случае $\boldsymbol{M = \max\limits_{1 \le i \le n} a_i}$ можно считать константой: $\boldsymbol{M \le (2^{32} - 1)}$, хотя и достаточно большой $(2^{32} - 1 = 4\,294\,967\,295)$.

Вычислительная сложность приведенного алгоритма для такой модели:

$\mathrm{T}(l) = \Theta(M \cdot n)$, где $n = \Theta(l)$, M – константа.

Алгоритм на практике будет вести себя, как полиномиальный.

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

Задача о рюкзаке:

> заданы рюкзак ограниченной грузоподъёмности $\boldsymbol{W}$ и $\boldsymbol{n}$ неделимых предметов. Для каждого предмета задан его вес $\boldsymbol{w_i} > \boldsymbol{0}$ и стоимость $\boldsymbol{p_i} > \boldsymbol{0}$ (например, если стоимость выше, то вещь более ценная). Числа – целые неотрицательные.
>
> Требуется определить набор предметов суммарная масса которых не превосходит $\boldsymbol{W}$ (т. к. грузоподъёмность рюкзака ограничена), а стоимость набора максимальна (взять наиболее ценные предметы).

Размерность задачи:

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

Алгоритм

(динамическое программирование)

	0	1	2	…	j	…	W
0	0	0	0	0	0	0	0
1	0						
…	0						
i	0						
…	0						
n	0						

Максимальная суммарная стоимость предметов, которые можно вложить в рюкзак грузоподъемностью $\boldsymbol{j}$, если можно использовать только первые $\boldsymbol{i}$ предметов:

$$\text{если } w_i > j, \text{то } dp[i,j] = dp[i-1,j]$$
$$\text{если } w_i \le j, \text{то } dp[i,j] = max\{dp[i-1,j], dp[i-1,j-w_i] + p_i\}$$

Наибольшая ценность $- \; dp[n,W]$, а сам набор восстанавливается обратным ходом.

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 $(= n)$ и вместимость рюкзака $(= W)$ были не слишком велики.

Пример

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

	w_i	p_i
1	**4**	5
2	**5**	7
3	**3**	4
4	**7**	9
5	**6**	8

```
function findAns(int i, int j)
 if dp[i][j] == 0 return
 if dp[i - 1][j] == dp[i][j]
     findAns(i - 1, j)
 else
     findAns(i - 1, j - w[i])
     ans.push(i)
```

Оптимальный набор:
(1-й предмет, 2-й предмет, 4-й предмет),
вес набора = 16, ценность = 21.
Решение неоднозначно: можно взять набор
(3-й предмет, 4-й предмет, 5-й предмет).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$W = 16$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5	5
2	0	0	0	0	5	7	7	7	7	12	12	12	12	12	12	12	12
3	0	0	0	4	5	7	7	9	11	12	12	12	16	16	16	16	16
4	0	0	0	4	5	7	7	9	11	12	13	14	16	16	18	20	21
$n = 5$	0	0	0	4	5	7	8	9	11	12	13	15	16	17	19	20	21

Размерность задачи:

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

В нашей модели сначала предположим, что

$$\max_{1\le i\le n}\lfloor\log_2 w_i+1\rfloor=C_1,\ \max_{1\le i\le n}\lfloor\log_2 p_i+1\rfloor=C_2,$$

где C_1 и C_2 – константы (например, 32 или 64 бита) и тогда полагаем, что каждый из параметров w_i занимает C_1 бит, а каждый из параметров p_i занимает C_2 бит.

Тогда

$$\lfloor\log_2 W+1\rfloor\le\log_2 W+1\le\log_2((2^{C_1}-1)\cdot n)+1<(C_1+1)+\log_2 n\cdot$$

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

$$\mathrm{T}(l)=\Theta(W\cdot n),\text{ где } W\le(2^{C_1}-1)\cdot n=O(n), n=\Theta(l).$$

Размерность задачи:

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

Если не ограничивать константами битовую длину параметров $\boldsymbol{w_i}$ и $\boldsymbol{p_i}$, то не сложно увидеть, что $l_1 > n$. Поэтому выполняется $n = \mathrm{o}(l_1)$.

Так как $l_2 \stackrel{\text{def}}{=} \lfloor \log_2 W + 1 \rfloor \le \log_2 W + 1$, то $\boldsymbol{W \ge 2^{(l_2 - 1)}}$.

Алгоритм для данной модели будет работать, как экспоненциальный по W:

$$\mathrm{T}(l) = \Theta(W \cdot n), W = \Omega\left(2^{l_2}\right), n = \mathrm{o}(l_1)\ (n = \mathrm{o}(l))$$

$$\mathrm{T}(l) = \Theta(\exp(l_2) \cdot p(l_1)).$$

БЕЛОРУССКИЙ
ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

Спасибо за внимание!