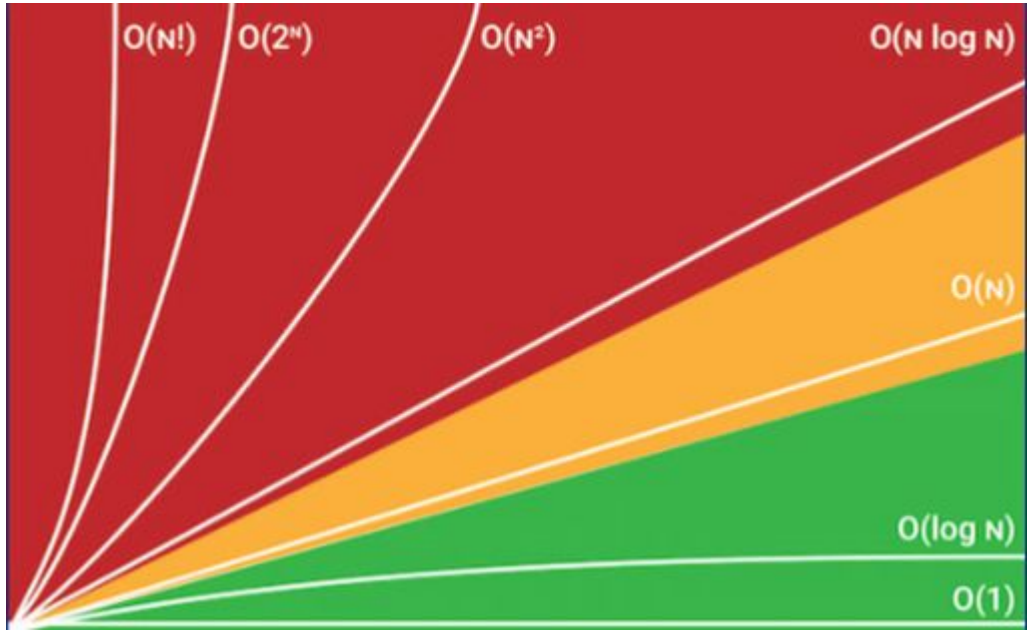
Трудоёмкость алгоритмов



Алгоритм

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

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

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

Определение

> **Трудоёмкость алгоритма** – это функция ***T*(*I*)**, которая оценивает сверху время, требуемое для решения задачи.

Аргументом функции ***T*** является размерность задачи ***I***.

Возникают вопросы:

1. С какими алгоритмами мы работаем, ведут ли они себя одинаково на одних и тех же входных данных при разных запусках алгоритма?
2. Как подсчитать время работы алгоритма? Какие входные данные надо учитывать?
3. Что такое размерность задачи ***I*** и как её подсчитать?

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**.

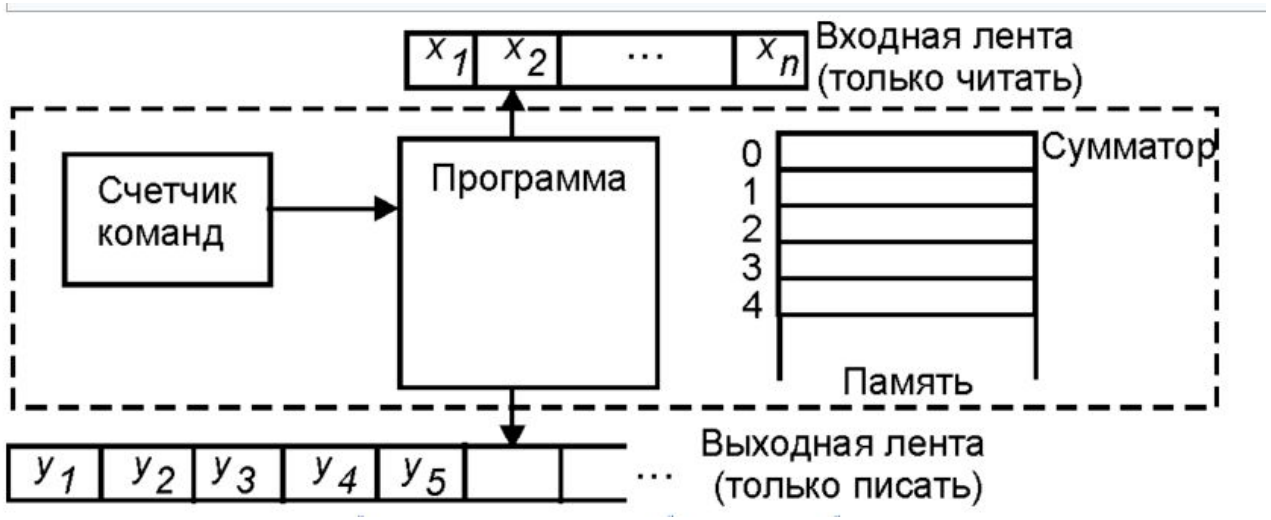
Как подсчитать время работы детерминированного алгоритма ?

На чём считать?

Модель вычислительного устройства:

Равнодоступная адресная машина (англ. Random-Access Machine - **RAM**).

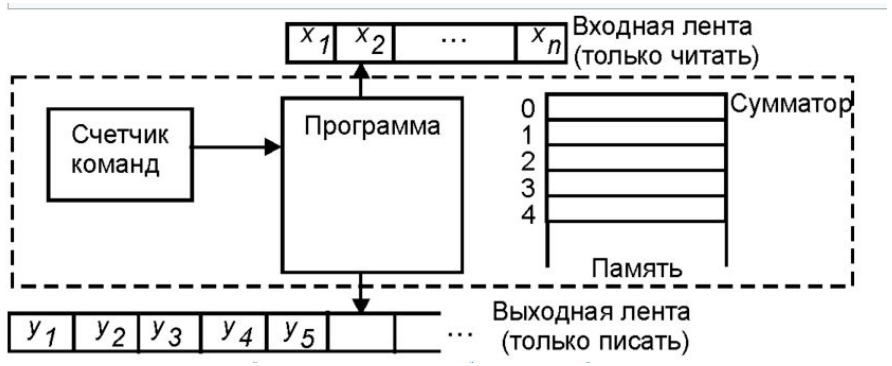
РАМ - универсальная математическая модель вычислений, которая является хорошим приближением к классу обычных вычислительных машин.



Равнодоступная адресная машина представляет собой вычислительную машину с одним сумматором, в котором команды программы не могут изменять сами себя.

PAM состоит из **входной ленты**, с которой она может считывать данные в соответствии с их упорядоченностью, **выходной ленты**, на которую она может записывать (в первую свободную клетку), и **памяти**, которая состоит из последовательности регистров (ячейка с номером 0 – сумматор).

Программа - это последовательность команд (команды занумерованы числами 1,2, ….).
Есть счетчик команд – целое число.
Сама программа не записывается в память.
Команды процессора выполняются последовательно, одновременно выполняемые команды отсутствуют (однопроцессорная машина).



Что считать?

Элементарный шаг вычисления:

Если в качестве модели вычислений взять неветвящуюся программу и предположить, что алгоритм – это последовательность арифметических операций и все арифметические операции эквивалентны, т.е. затрачивают на свое выполнение одну единицу времени (равномерный весовой критерий), то время работы алгоритма – число операций алгоритма.

В некоторых задачах (например, сортировка) удобно в качестве основной меры сложности брать число выполняемых команд разветвления.

На практике

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

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

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

Реальному ЦП требуется разное количество времени для выполнения различных операций.

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

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

mul, imul: 3–4 такта

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

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

Подробнее о времени выполнения операций: https://www.agner.org/optimize/instruction_tables.pdf

На практике

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

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

Информация подготовлена студентом ФПМИ
Владиславом Дмитриевичем Кощенко, 2022 год

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 ~ 10^8 абстрактных операций.

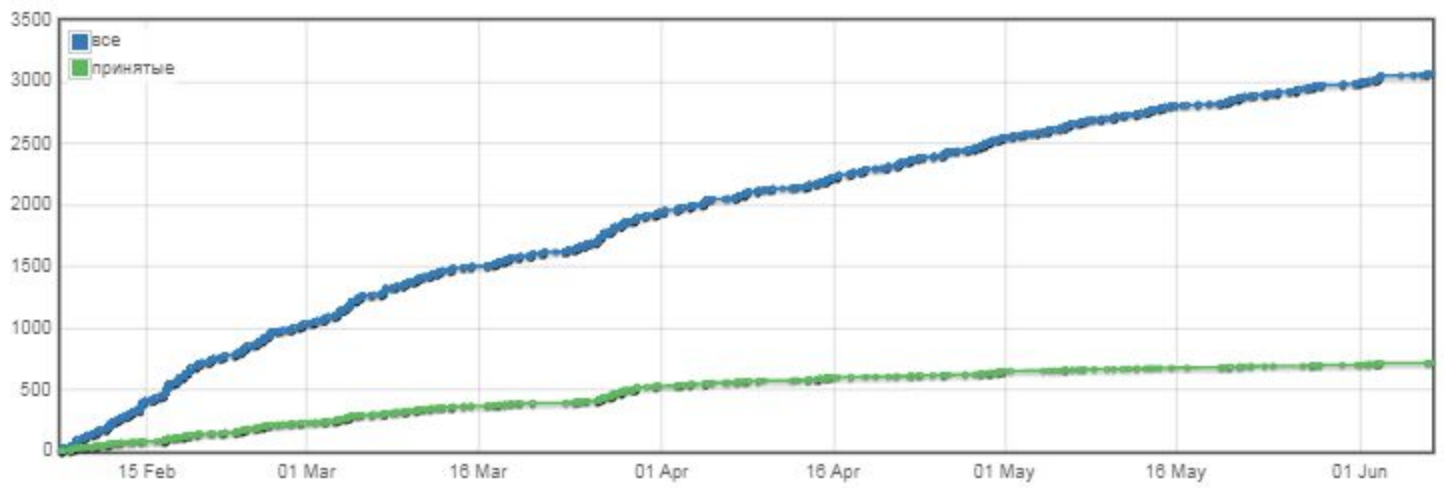
Если вы делаете много делений, если вы обращаетесь к большому количеству памяти в случайном порядке, то вы сможете сделать гораздо меньше, ~ 10^7.

Если операции простые, а обращения к памяти локальные или последовательные, то вы сможете выполнить ~ 10^9 операций.

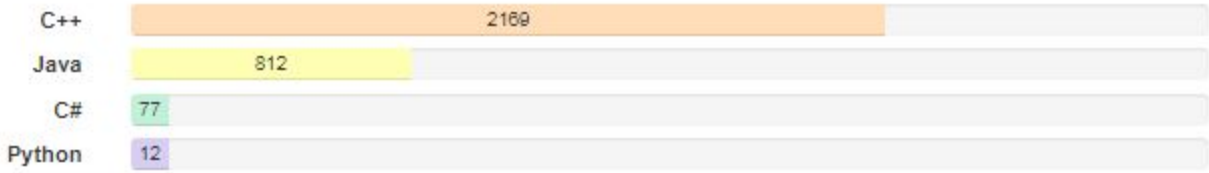
Популярность языков программирования у студентов

- 2-й курс 1-я группа ТА 2020–2021
- *В архиве*
- Задачи по курсу
- Сообщения 4
- Назначение задач
- Журнал
- Ведомость
- Все решения
- Мои решения
- Компиляторы
- Тесты
- Электронная очередь
- Настройки

Число отправленных решений



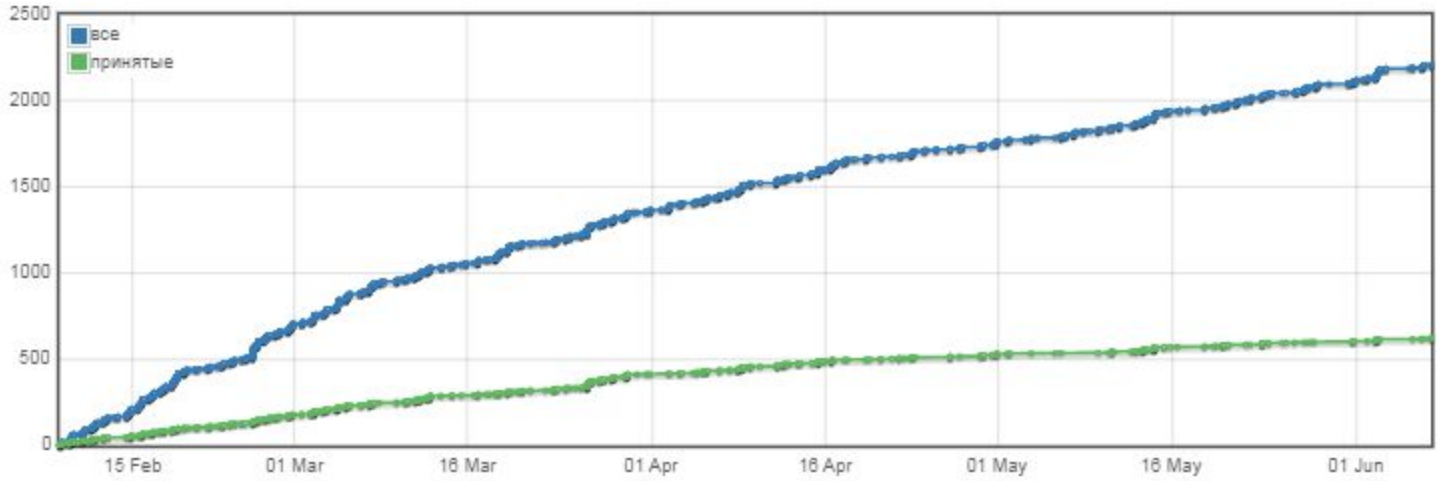
Популярность языков программирования



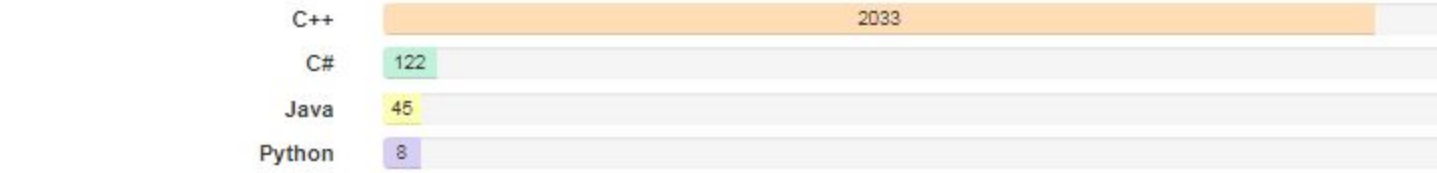
ФПМИ БГУ

- 2-й курс 2-я группа ТА 2020–2021
- *В архиве*
- Задачи по курсу
- Сообщения 2
- Назначение задач
- Журнал
- Ведомость
- Все решения
- Мои решения
- Компиляторы
- Тесты
- Электронная очередь
- Настройки

Число отправленных решений



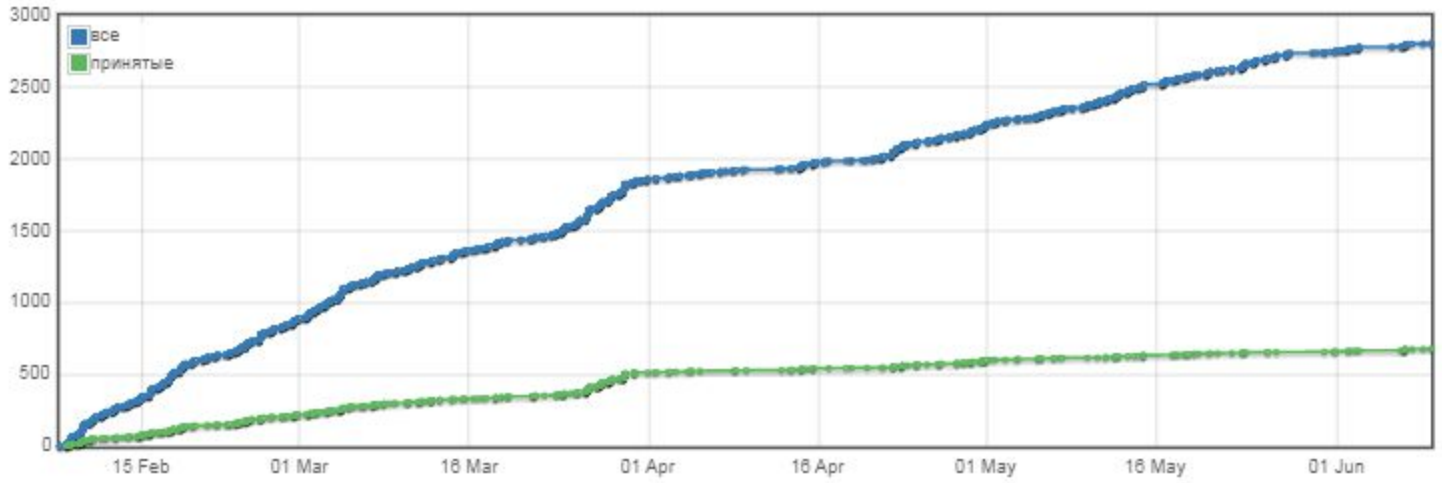
Популярность языков программирования



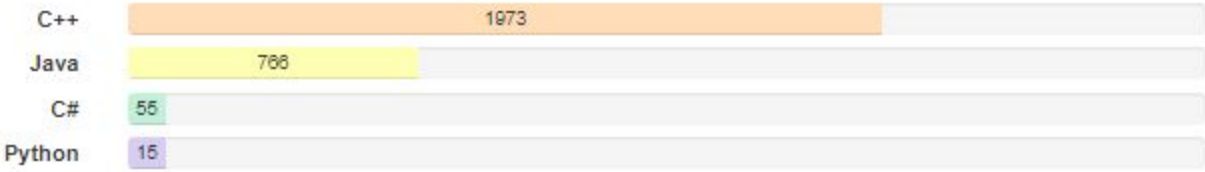
ФПМИ БГУ

- 2-й курс 3-я группа ТА 2020–2021
- *В архиве*
- Задачи по курсу
- Сообщения 3
- Назначение задач
- Журнал
- Ведомость
- Все решения
- Мои решения
- Компиляторы
- Тесты
- Электронная очередь
- Настройки

Число отправленных решений



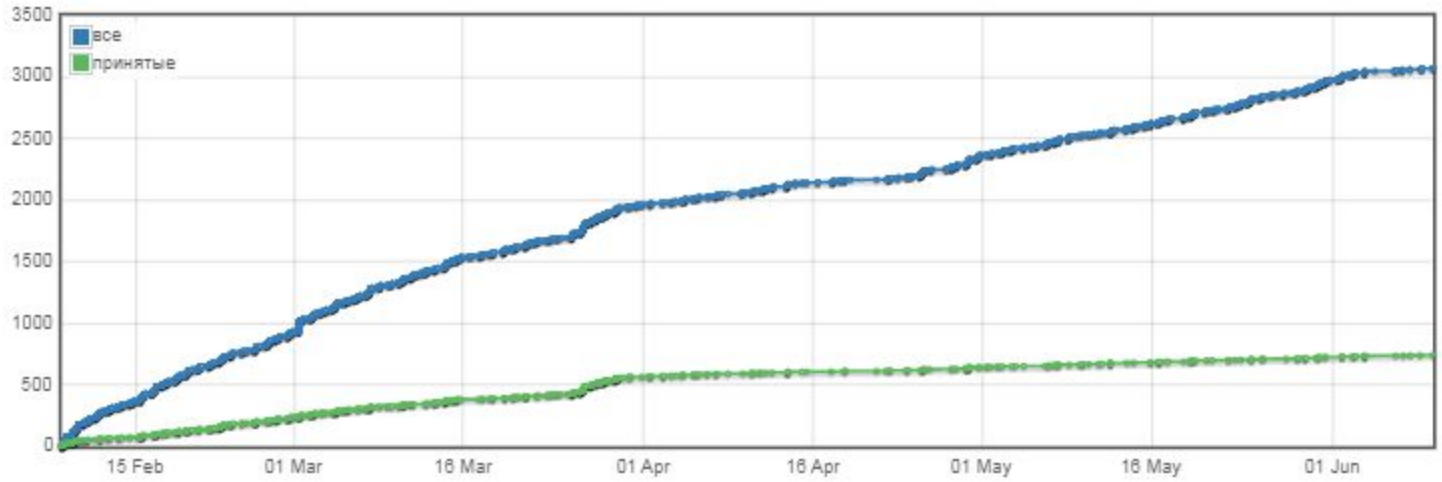
Популярность языков программирования



ФПМИ БГУ

- 2-й курс 4-я группа ТА 2020–2021
- *В архиве*
- Задачи по курсу
- Сообщения 1
- Назначение задач
- Журнал
- Ведомость
- Все решения
- Мои решения
- Компиляторы
- Тесты
- Электронная очередь
- Настройки

Число отправленных решений



Популярность языков программирования

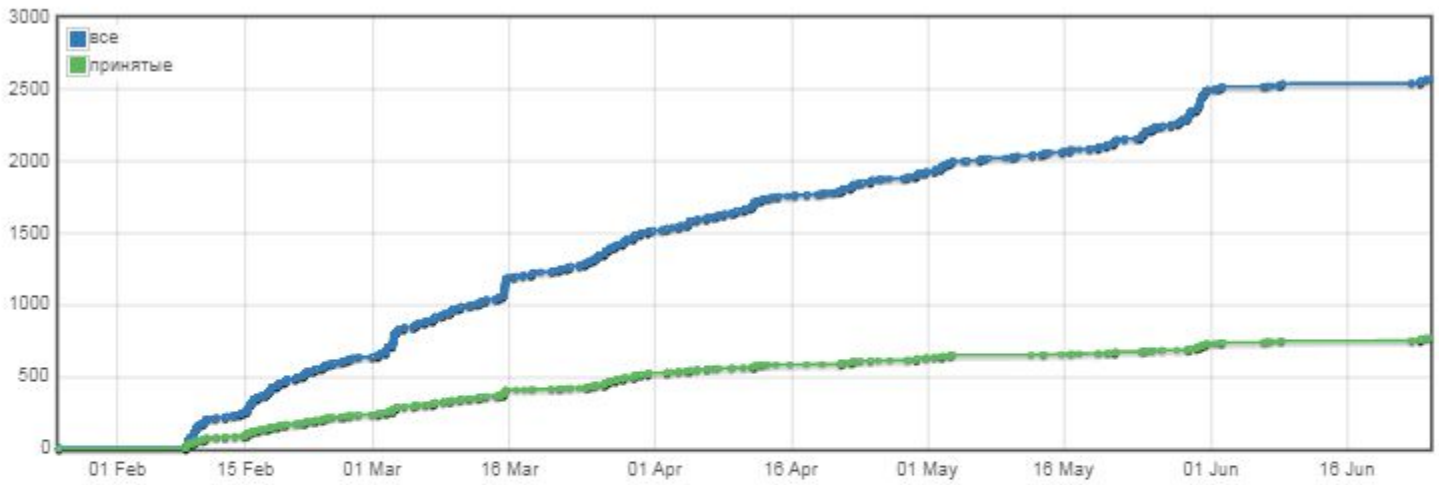
C++	2857
Java	120
Python	94
C#	3

ФПМИ БГУ

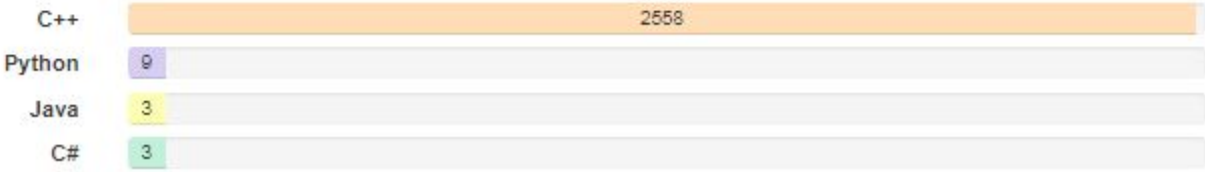
2-й курс 5-я группа ТА 2020–2021

- *В архиве*
- Задачи по курсу
- Сообщения 1
- Назначение задач
- Журнал
- Ведомость
- Все решения
- Мои решения
- Компиляторы
- Тесты
- Электронная очередь
- Настройки

Число отправленных решений



Популярность языков программирования



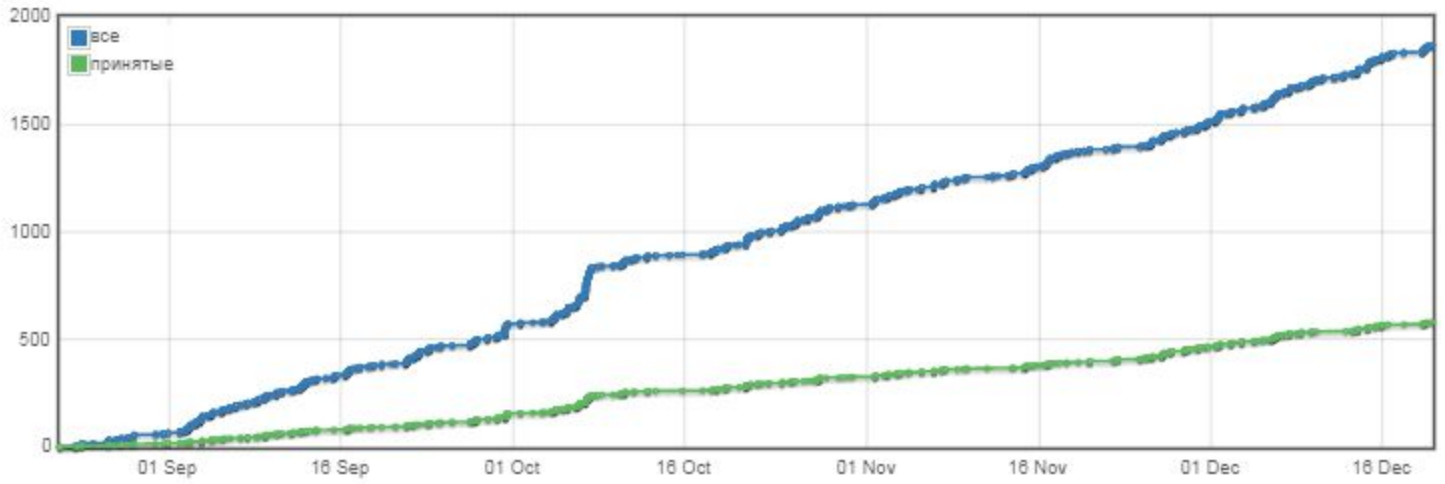
ФПМИ БГУ

2022
год

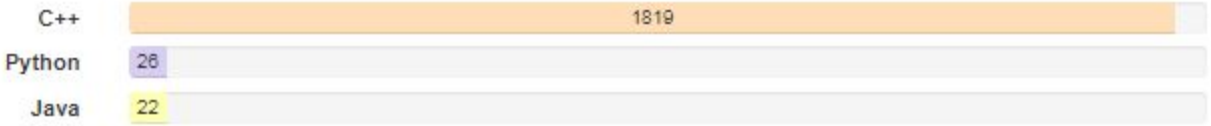
ФПМИ БГУ

- 2-й курс 13-я группа АиСД 2021–2022
- *В архиве*
- Задачи по курсу
- Сообщения
- Назначение задач
- Журнал
- Ведомость
- Все решения
- Мои решения
- Компиляторы
- Тесты
- Электронная очередь
- Настройки

Число отправленных решений



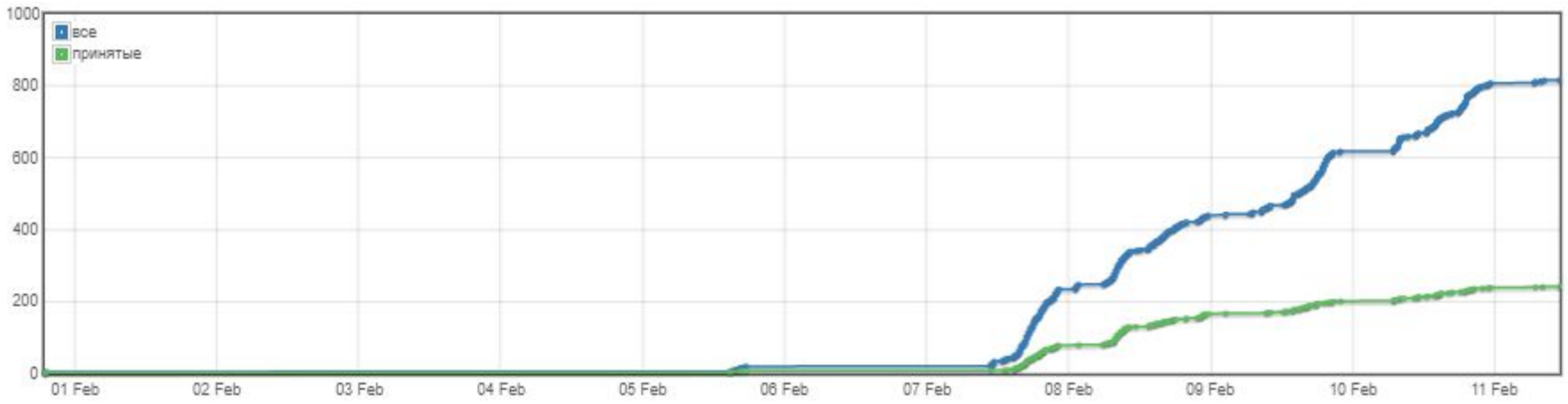
Популярность языков программирования



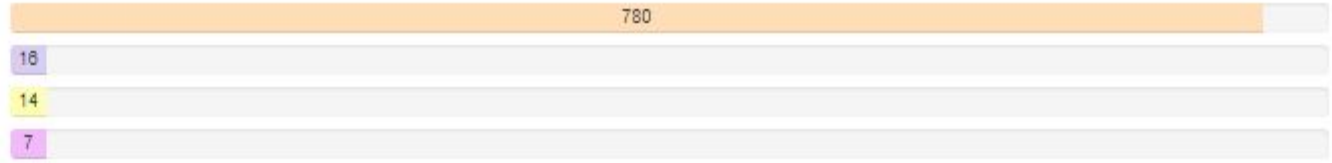
ФПМИ БГУ

2-й курс 5-я группа ТА 2021–2022
Задачи по курсу
Сообщения
Назначение задач
Журнал
Ведомость
Все решения
Мои решения
Отправить решение
Компиляторы
Тесты
Электронная очередь
Настройки

Число отправленных решений

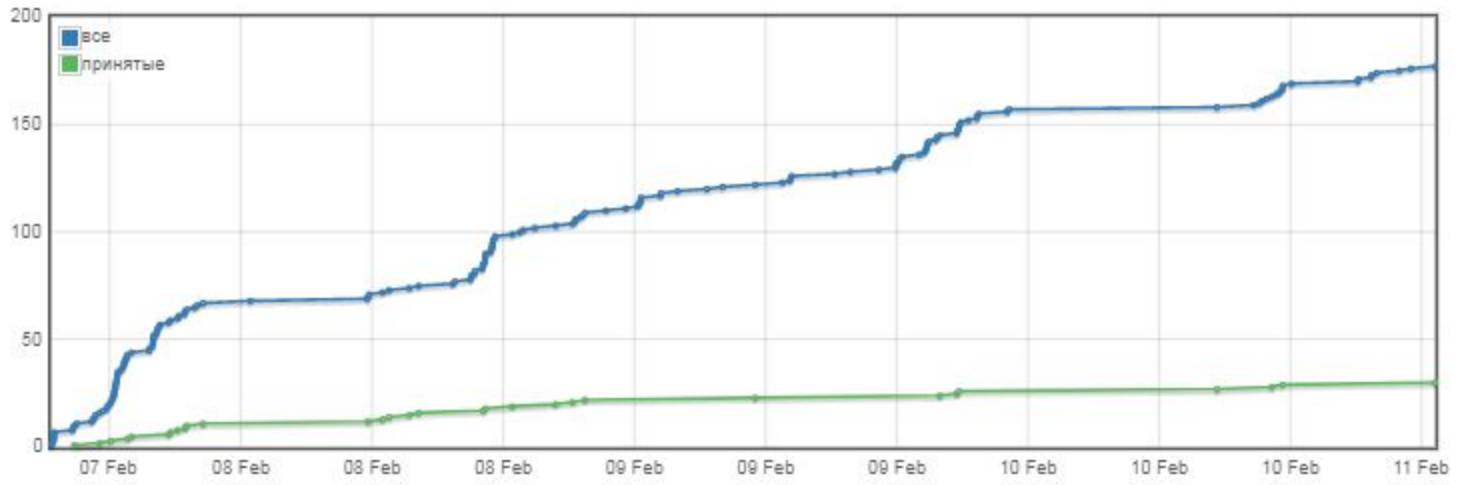


Популярность языков программирования



- 2-й курс 6-я группа АиСД 2021–2022
- Задачи по курсу
- Сообщения
- Назначение задач
- Журнал
- Ведомость
- Все решения
- Мои решения
- Отправить решение
- Компиляторы
- Тесты
- Электронная очередь
- Настройки

Число отправленных решений

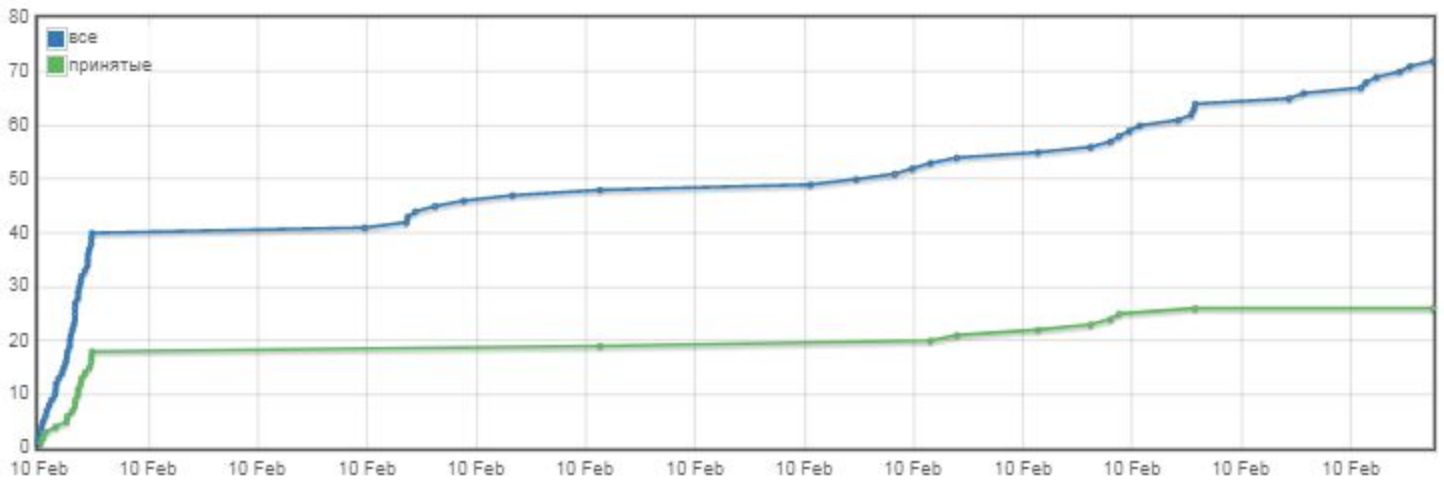


Популярность языков программирования

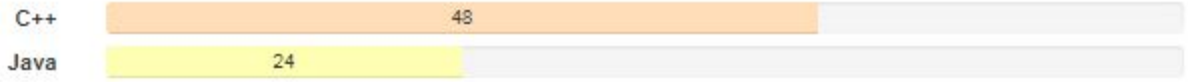
ФПМИ БГУ

- 2-й курс 7-я группа АиСД 2021–2022
- Задачи по курсу
- Сообщения
- Назначение задач
- Журнал
- Ведомость
- Все решения
- Мои решения
- Отправить решение
- Компиляторы
- Тесты
- Электронная очередь
- Настройки

Число отправленных решений



Популярность языков программирования



ФПМИ БГУ

Для оценки **<u>времени работы детерминированного алгоритма в худшем случае</u>** будем искать такой набор входных данных, на котором алгоритм работает дольше всего.

При этом нас будет интересовать порядок роста полученной функции, так как важна скорость роста функции при возрастании объема входных данных, т.е. оставляем только ту часть функции, которая с ростом аргумента к бесконечности, растёт быстрее всего.

Так порядок функции
$f(n) = n^2 + n\log n + 4$
есть n^2
(говорят, что функция $f(x)$
растёт как n^2).

Пример 1.
Последовательный поиск элемента ***x*** в произвольном массиве из ***n*** элементов. время $\leq C \cdot n$

Пример 2.
Сортировка массива из ***n*** элементов «пузырьком» (если на некоторой итерации нет ни одного обмена, то завершаем алгоритм). время $\leq C \cdot n^2$

Среднее время работы детерминированного алгоритма по всем возможным наборам входных данных

1) Все входные данные 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 $\mathbf{p_i}$ – вероятность, с которой данные попадают в **группу *i***.
3) Пусть $\mathbf{t_i}$ –время работы алгоритма для данных из группы ***i***.

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

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:

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

Пример.

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.

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

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

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

…

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

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

$$t_1 = 1$$

$$t_2 = 2$$

$$t_3 = 3$$

…

$$t_n = n$$

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

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

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

$$= \frac{1}{n+1}\left(\frac{1+n}{2} \cdot n + n\right) = \frac{n}{2} + \frac{n}{n+1} = \frac{n}{2} + 1 - \frac{1}{n+1} \approx \frac{n}{2} + 1$$

А как же подсчитать время работы рандомизированного алгоритма в худшем случае?

данные 1	данные 2	данные 3	…	данные m
1-й запуск алгоритма 2-й запуск алгоритма …… k_1-й запуск алгоритма	1-й запуск алгоритма 2-й запуск алгоритма …… k_2-й запуск алгоритма	1-й запуск алгоритма 2-й запуск алгоритма …… k_3-й запуск алгоритма	…	1-й запуск алгоритма 2-й запуск алгоритма …… k_m-й запуск алгоритма
$E_1 = \frac{\sum k_1}{k_1}$	$E_2 = \frac{\sum k_2}{k_2}$	$E_3 = \frac{\sum k_3}{k_3}$	…	$E_m = \frac{\sum k_m}{k_m}$

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

Функции можно сгруппировать по скорости роста в три основных класса (три асимптотики):

Асимптотик

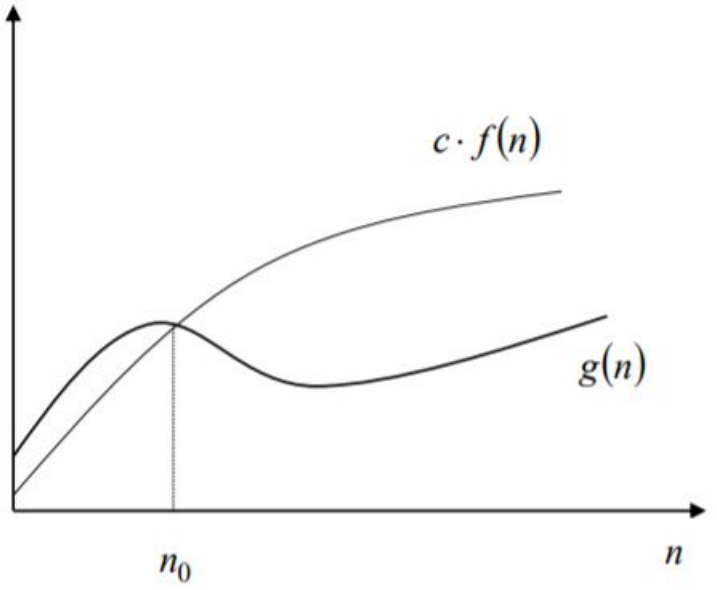
O, Ω, Θ

$O(f(n))$	**о большое** от *f* от *n*
$\Omega(f(n))$	**омега большое** от *f* от *n*
$\Theta(f(n))$	**тэтта большое** от *f* от *n*

O(f(n)) – это множество функций, которые растут не быстрее, чем функция f(n)

$$g(n) = \mathrm{O}\big(f(n)\big) \Leftrightarrow \exists \mathrm{c}, \mathrm{n}_0 > 0 : \forall n \geq \mathrm{n}_0$$

$$0 \leq g(n) \leq \mathrm{c} \cdot f(n)$$



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

$$n^2 = \mathrm{O}(n^3)$$

$$n \log n = \mathrm{O}(n^3)$$

$$4n^3 = \mathrm{O}(n^3)$$

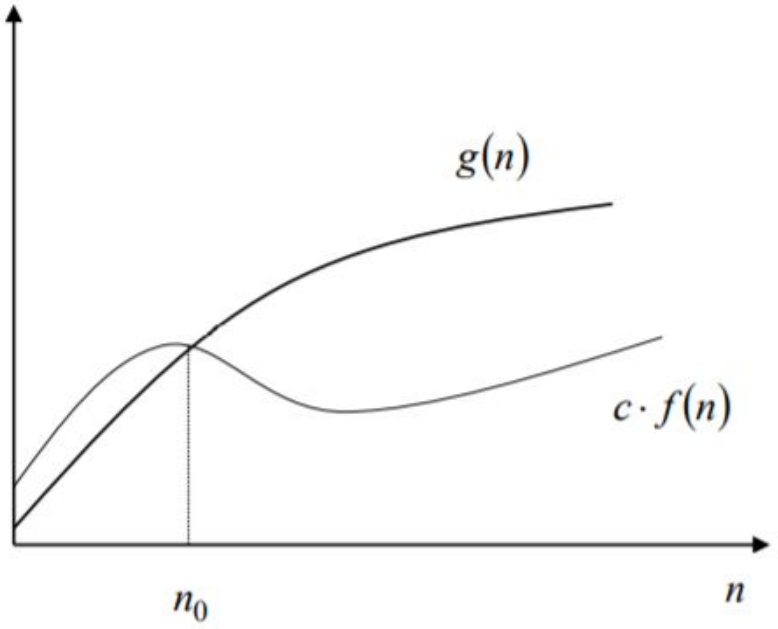
$$g(n) = o\big(f(n)\big) \Leftrightarrow \forall \mathrm{c}, \exists \mathrm{n}_0 > 0 : \forall n \geq \mathrm{n}_0$$

$$0 \leq g(n) < c \cdot f(n)$$

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

Ω (f(n)) – это множество функций, которые растут, по крайней мере, так же быстро, что и функция f(n)

$$g(n) = \Omega(f(n)) \Leftrightarrow \exists c, n_0 > 0 : \forall n \geq n_0$$
$$0 \leq c \cdot f(n) \leq g(n)$$



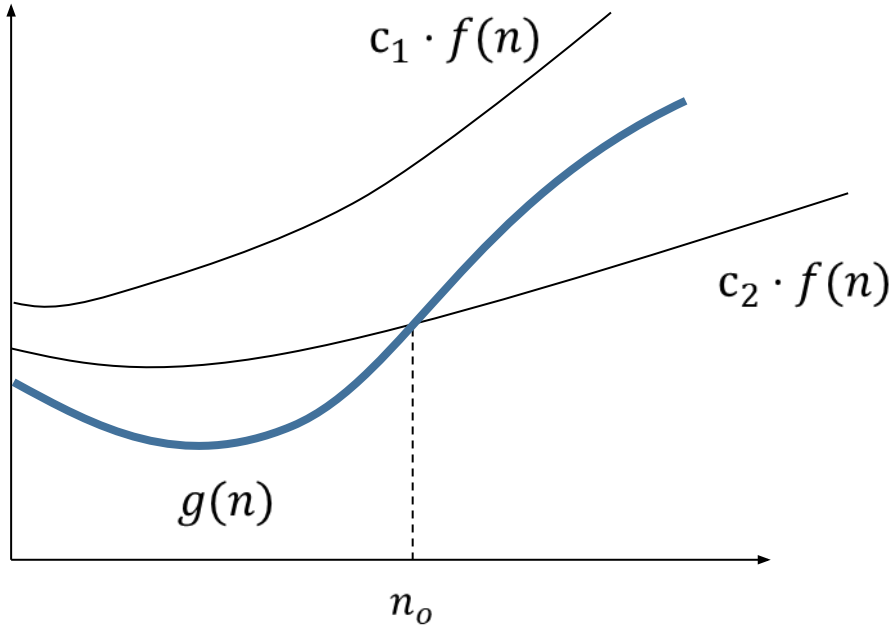
Говорят, что функция *f*(*n*) даёт **асимптотическую нижнюю границу** для функции *g*(*n*).

$$4n^3 = \Omega(n^3)$$
$$n^4 \log n = \Omega(n^3)$$
$$2^n = \Omega(n^3)$$

$$g(n) = \omega(f(n)) \Leftrightarrow \forall c, \exists n_0 > 0 : \forall n \geq n_0$$
$$0 \leq c \cdot f(n) < g(n)$$
$$4n^3 \neq \omega(n^3)$$

Θ(f(n)) – это множество функций, которые растут с той же скоростью роста, что и функция f(n)

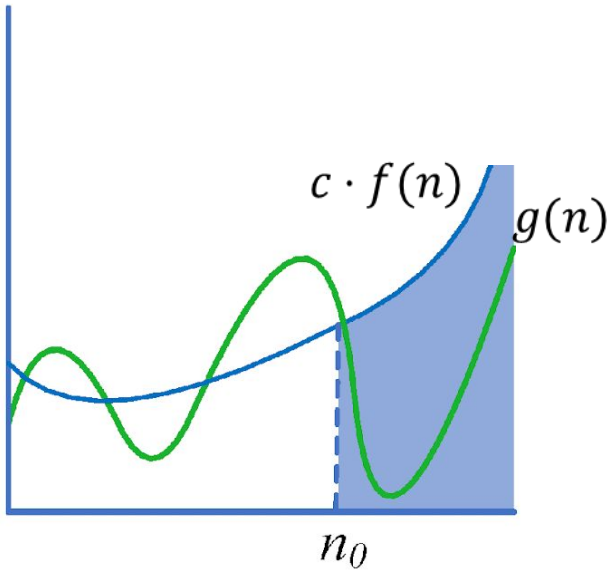
$$g(n) = \Theta(f(n)) \Leftrightarrow \exists c_1, c_2, n_0 > 0 : \forall n \geq n_0$$
$$0 \leq c_2 \cdot f(n) \leq g(n) \leq c_1 \cdot f(n)$$



Говорят, что функция *f*(*n*) является **асимптотически точной оценкой** для функции *g*(*n*).

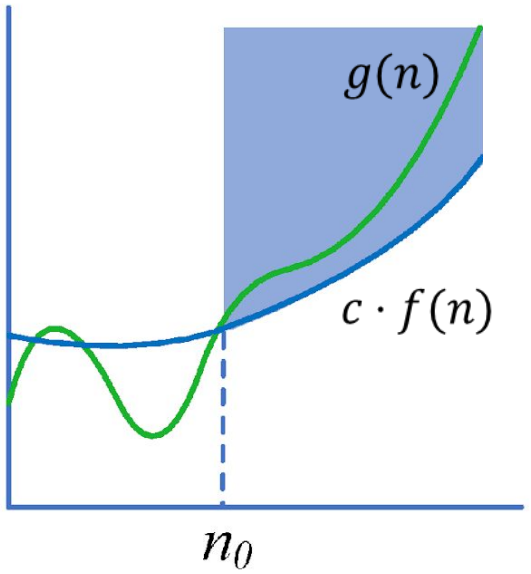
$$4n^3 = \Theta\left(n^3\right)$$

$$n^3 = \Theta\left(n^3\right)$$



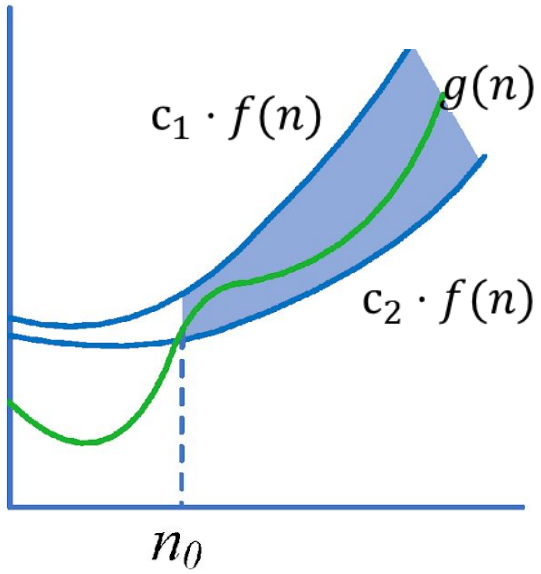
$$g(n) = O(f(n))$$

f(*n*) даёт **асимптотическую верхнюю границу** для функции *g*(*n*)



$$g(n) = \Omega(f(n))$$

f(*n*) даёт **асимптотическую нижнюю границу** для функции *g*(*n*)



$$g(n) = \Theta(f(n))$$

f(*n*) **асимптотическая точная оценка** для функции *g*(*n*)

Скрытые под асимптотикой константы …

Доказать формально, что для функции $g(n)=6n^4-2n^3+5$ праведливо $g(n)=\Theta\left(n^4\right)$

Доказательство

$$g(n)=\Theta\left(n^4\right)\Leftrightarrow\exists c_1,c_2,n_0>0:\forall n\geq n_0\quad 0\leq c_2\cdot n^4\leq g(n)\leq c_1\cdot n^4$$

$$0\leq c_1\cdot n^4\leq 6n^4-2n^3+5\leq c_2\cdot n^4$$

$$0\leq c_1\leq 6-\frac{2}{n}+\frac{5}{n^4}\leq c_2$$

Мы можем выбрать $c_1=5$, тогда для $n\geq 2$ левая часть неравенства выполняется.

Мы можем выбрать $c_2=11$, тогда для $n\geq 1$ правая часть неравенства выполняется.

Т.о., выбирая $c_1=5, c_2=11$, $n_0=2$ мы говорим, что $6n^4-2n^3+5=\Theta\left(n^4\right)$.

Константы можно выбрать и по-другому, однако важно не то, как их выбрать, а то, что такой выбор существует.

Для функции

$$f(n) = a \cdot n^2 + b \cdot n + c, a > 0$$

выберем константы по следующему правилу:

$$c_1 = \frac{a}{4}, c_2 = \frac{7 \cdot a}{4}, n_0 = 2 \cdot \max\left\{\frac{|b|}{a}, \sqrt{\frac{|c|}{a}}\right\}.$$

Тогда $\forall n \geq n_0$ выполняется неравенство

$$0 \leq c_1 \cdot n^2 \leq f(n) \leq c_2 \cdot n^2$$

т.е. $f(n) = \Theta\left(n^2\right)$.

Вообще говоря, для любого полинома

$$f(n) = \sum_{i=0}^{d} a_i n^i$$

где a_i – константы и $a_d > 0$,

справедливы неравенства

$$\exists c_1, c_2, n_0 > 0 : \forall n \geq n_0$$

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

т.е. $f(n) = \Theta\left(n^d\right)$

$$1 \ll \log n \ll \sqrt{n} \ll n \ll n \cdot \log n \ll n^2 \ll 2^n \ll n! \ll n^n$$

	n	n·log n	n^2	2^n
n=20	1c	1c	1c	1c
n=50	1c	1c	1c	13 дней
n=100	1c	1c	1c	10^{13} лет
n=10^6	1c	1c	17 мин	
n=10^9	1c	35 с	30 лет	
максимальное n , чтобы успеть решить за 1с.	10^9	10^7	$10^{4,5}$	30

Оцените асимптотически время работы в худшем случае следующих алгоритмов

$\Theta(n)$ 1. Время последовательного поиска элемента **x** в произвольном массиве из n элементов?

$\Theta(n)$ 2. Время нахождения суммы всех элементов массива. В массиве n элементов.

$\Theta(n^2)$ 3. Время построения бинарного поискового дерева для последовательности из n чисел,
Дерево строится последовательным добавлением вновь поступающих элементов?

$\Theta(n^2)$ 4. Время сортировки «пузырьком» последовательности из n элементов?

$\Theta(\sqrt{x})$ 5. Время алгоритма определения числа **x** на простоту: делим x на все числа от 2 до $\sqrt{x}$?

$\Theta(n)$ 6. Время вычисления n!: последовательно перемножаем числа от 1 до n?

Трудоёмкость алгоритма –

> это функция T(I), которая оценивает сверху время, требуемое для решения задачи.

Для того, чтобы найти трудоёмкость алгоритма, нужно время его работы

- для его вычисления мы использовали равномерный весовой критерий, при котором каждая операция выполняется за 1 единицу времени и каждая ячейка занимает 1 единицу памяти

выразить через размерность задачи

- логарифмический весовой критерий – объем памяти, необходимый для хранения числа, равен длине двоичного представления этого числа, а время выполнения команды пропорционально длине его операндов.

Определение

Размерность задачи l **-** минимальное число **бит**, которого достаточно, чтобы разрушить неопределённость о входных данных задачи.

Задача 1

На вход поступает одно целое число, которое выбирается из множества целых чисел 1,2, …. ,x (причём любой выбор является равновероятным).

Чему равна размерность задачи, т.е. какого минимального числа бит достаточно, чтобы определить, какое число мы ввели?

Решение

Известно, что в k битах можно закодировать 2^k различных исходов.

Предположим, что $l-1$ бит было мало, а l – достаточно, чтобы распознать x исходов, т.е.

$$2^{l-1} < x \le 2^l.$$

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

$$\log_2 x \le l < \log_2 x + 1$$

$$l = \lceil \log_2 x \rceil$$

Сведения из математики:

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

Задача 2

На вход поступает одно целое число, которое выбирается из множества целых чисел $0,1,\ldots,x$, причём любой выбор является равновероятным.

Чему равна размерность задачи?

Решение

Известно, что в k битах можно закодировать 2^k различных исходов.

Предположим, что $l-1$ бит было мало, а l – достаточно, чтобы распознать $x+1$ исход, т.е.

$$2^{l-1} < x+1 \le 2^l.$$

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

$$\log_2(x+1) \le l < \log_2(x+1)+1$$

$$l = \lceil \log_2(x+1) \rceil$$

Задача 3.

На вход поступает одно целое число, которое выбирается из множества целых чисел $\{-x, \dots, 0, \dots, x\}$ (причём любой выбор является равновероятным).
Чему равна размерность задачи?

Решение

Известно, что в k битах можно закодировать 2^k различных исходов.
Предположим, что $l-1$ бит было мало, а l – достаточно, чтобы распознать $2 \cdot x + 1$ исход, т.е.

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

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

$$\log_2(2x+1) \leq l < \log_2(2x+1) + 1$$

$$l = \lceil \log_2(2x+1) \rceil$$

В дальнейшем будем предполагать (если не оговорено иное), что, если на вход алгоритма поступает некоторое целое число $\boldsymbol{m}$, то предполагается, что это число может быть выбрано из множества $\{1,2,\dots,m\}$, причём любой выбор является равновероятным.

Задача 4

На вход поступает одно рациональное число a/b.

Чему равна размерность задачи?

Решение

Найдём множество возможных входных данных

$$S = \{(x, y) : x \in \{1, \ldots, a\}, \quad y \in \{1, \ldots, b\}\},$$

где входным числом будем считать

$$x/y.$$

Тогда

$$l = \lceil \log_2 (a \cdot b) \rceil = \lceil \log_2 a + \log_2 b \rceil.$$

Задача 5.

На вход поступает массив из n чисел.

Каждое число выбирается из множества целых чисел $\{1,2,\dots,x\}$, причём любой выбор является равновероятным.

Чему равна размерность задачи, т.е. какого количества бит достаточно, чтобы запрограммировать такой вход?

Решение

$$l = \lceil \log_2 x \rceil + \lceil \log_2 x \rceil + \dots + \lceil \log_2 x \rceil = \sum_{i=1}^{n} \lceil \log_2 x \rceil = n \cdot \lceil \log_2 x \rceil$$

$$l = n \cdot \lceil \log_2 x \rceil$$

Если x – константа и не зависит от n, например, $x = 2^{32}$,

тогда $l = C \cdot n$, где C – константа.

Оценка трудоёмкости алгоритма

1) Сформулировали задачу и описали алгоритм её решения.
2) Вычислили время работы алгоритма (в худшем случае).
3) Вычислили размерность задачи (по входным данным задачи).
4) Выразили время работы алгоритма через размерность задачи и получили функцию T(l) - трудоёмкость алгоритма.
5) Теперь нужно сделать вывод о том, какой был разработан алгоритм: полиномиальный или экспоненциальный.

Алгоритм называется **полиномиальным**, если его трудоемкость $T(l) = \mathrm{O}\big(p(l)\big),$

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

Сведения из математики

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

$$p(l) = \sum_{i=0}^{d} a_i \cdot l^i, a_d \neq 0.$$

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

2) **Функция $p(l)$ полиномиально ограничена**, если существует такая константа k, что

$$p(l) = \mathrm{O}\left(l^k\right).$$

3) Любая положительная полиномиальная функция возрастает быстрее, чем любая полилогарифмическая функция ($f(l) = \log^{\mathrm{k}} l$).

Задача.

На вход поступает массив из n чисел $a_1, a_2, \dots, a_n$.

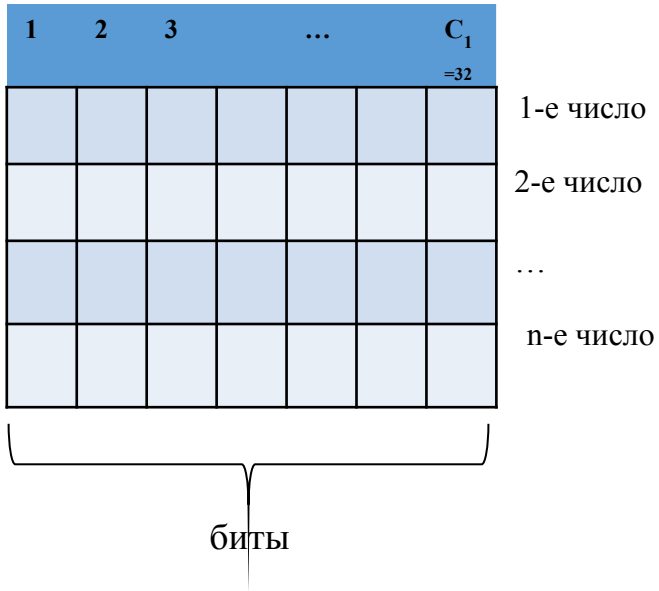
Нужно найти сумму этих чисел $a_1 + a_2 + \dots + a_n$

Какой это алгоритм: полиномиальный или экспоненциальный?

Размерность задачи

Входные данные: $a_i \in \{1,2,\dots,x\}, i = 1,\dots,n.$

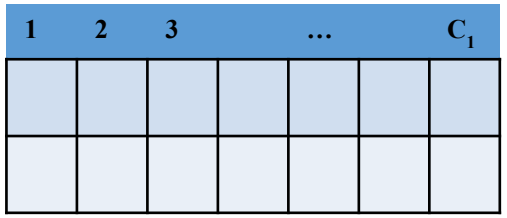
Если x не зависит от n (например, $x = 2^{32}$),
то $l = \lceil \log_2 x \rceil \cdot n = C_1 \cdot n$,
где C_1 – константа, не зависящая от n.



Время работы алгоритма

$O(n)$

Будем предполагать, что в результате суммирования не произойдёт переполнения.
Так как каждое число занимает C_1 бит, то сложение двух чисел будет выполнено за время $O(C_1)$.



Трудоёмкость

$$T(l) \le C_1 \cdot n = l$$

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

***Ответ**: алгоритм полиномиальный*

Алгоритм называется **экспоненциальным**, если его трудоемкость

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

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

Сведения из математики

1) **Экспоненциальная функция** это функция:

$$\exp(l) = a^l, \quad a > 1$$

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

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

2) Любая экспоненциальная функция возрастает быстрее полиномиальной функции.

3) Функция $n!$ возрастает быстрее, чем 2^n, но медленнее, чем функция n^n

Задача.

На вход поступает одно целое положительное число n.

Нужно вычислить $n! = 1 \cdot 2 \cdot \ldots \cdot n$

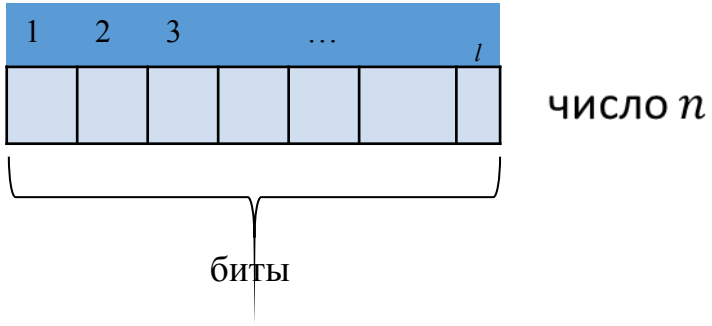
Какой это алгоритм: полиномиальный или экспоненциальный?

Размерность задачи

Входные данные:

единственное число n.

Предположим,что $2^{l-1} < n \leq 2^l$, тогда $l = \lceil \log_2 n \rceil$.



число n

Время работы алгоритма $\Omega(n)$

Сколько бы времени не занимало одно перемножение двух l-битных чисел, например,

(1) в «столбик» - l^2
(2) алгоритм Карацубы – $\Theta(l^{1,585})$
(3) используя быстрое преобразование Фурье – $l \cdot \log l$

число перемножений будет всегда n.

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.

→ Трудоёмкость

$$T(l) \geq n > 2^{l-1}$$

$$T(l) = \Omega\left(2^l\right)$$

***Ответ**: алгоритм экспоненциальный.*

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

$$T(l) = O(\ p(l, |Max(I)|)\ ), \text{где } p - \text{ полином от двух переменных:}$$

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).

Пример.

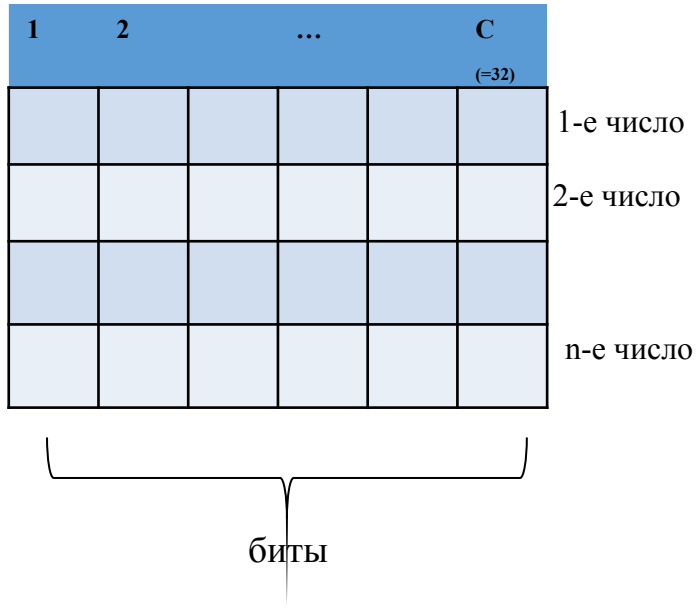
Задан массив A из n целых положительных чисел. Подсчитать частоту встречаемости всех элементов.

Псевдополиномиальный алгоритм решения задачи:

1) найдем максимальный элемент в массиве, предположим, что это число *M*;
2) выполним цикл от 1 до *M* и для каждого значения подсчитаем частоту его встречаемости в массиве;

$$\mathrm{T}(l) = \mathrm{O}(M \cdot l),$$

где $\mathrm{M} = \max_i a_i, l = C \cdot n$



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.

Следующие задачи являются примерами ***NP*-полных задач** комбинаторной оптимизации, для которых существуют **псевдополиномиальные алгоритмы** решения методом динамического программирования являются:

1) **Задача о разбиении** мультимножества положительных целых чисел. Задано мультимножество S из n целых положительных чисел. Можно ли его разбить на два подмножества S_1 и S_2 так, чтобы сумма элементов из S_1 равнялась сумме элементов из S_2?

$$\mathrm{T}(l) = \mathrm{O}(M \cdot l), \text{где } M = \sum_{i=1}^{n} s_i \ .$$

Например, для множества $S = \{1,3,3,5,6\}$
одним из вариантов разбиения может быть: $S_1 = \{1,3,5\}, S_2 = \{3,6\}$.

Так как дополнительная память, которая потребуется алгоритму $\Theta(M \cdot n)$, то для того, чтобы индивидуальную задачу можно было решить методом ДП нужно, чтобы размер множества S $(= n)$ и сумма элементов этого множества $(= M)$ были не слишком велики.

2) **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 ограничен, а стоимость набора максимальна (взять наиболее ценные предметы).

$$\mathrm{T}(l) = \mathrm{O}(W \cdot l)$$

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

Различие между полиномиальными и экспоненциальными алгоритмами заметно при решении задач большой размерности.

	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

Следует отметить очень быстрый рост двух приведенных экспонент.

Различие между полиномиальными и экспоненциальными алгоритмами проявляется еще более убедительно, если проанализировать влияние увеличения быстродействия ЭВМ на время работы алгоритмов. Так, для функции $f(n) = 2^n$ увеличение скорости вычислений в 1 000 раз приводит лишь к тому, что размерность наибольшей задачи, разрешимой за один час, возрастет только на 10, в то время как для функции $f(n) = n^5$ эта размерность возрастет почти в 4 раза. **Таким образом, колоссальный рост скорости вычислений, вызванный появлением нынешнего поколения цифровых вычислительных машин, не уменьшает значение эффективных алгоритмов**.

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

Например, при n ≤ 20 функция $f(n) = 2^n$ ведет себя лучше, чем функция $f(n) = n^5$.

Кроме того, **известны некоторые экспоненциальные алгоритмы, 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.

Упражнение.

На вход поступает одно целое положительное число n.

Нужно вычислить сумму $1 + 2 + \cdots + n$.

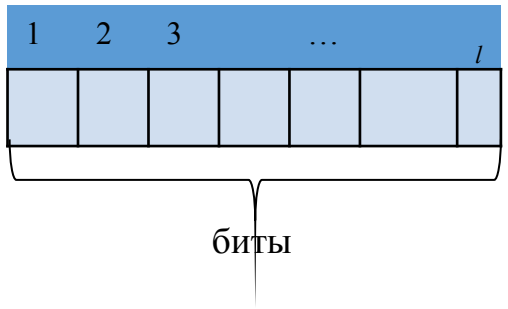
Алгоритм 1. Использовать формулу арифметической прогрессии: $\frac{(1+n)}{2} \cdot n$.

Алгоритм 2. Непосредственное суммирование $1 + 2 + \cdots + n$.

Размерность задачи

Предположим,что $2^{l-1} < n \leq 2^l$, тогда

$l = \lceil \log_2 n \rceil$.



число n

$T(l)$?

Алгоритм 1.

$\frac{(1+n)}{2} \cdot n$

полиномиальный или экспоненциальный?

$T(l)$?

Алгоритм 2.

$1 + 2 + \cdots + n$.

полиномиальный или экспоненциальный?

Спасибо за внимание!