Analýza a Zložitosť Algoritmov

AZA

Doc. RNDr. Silvester Czanner, PhD

silvester.czanner@stuba.sk (subject AZA)

Summary – session 1

- Lecturer introduction
- Refresher
  - Data structure taxonomy
  - Data structure landscape
  - C++ programming
  - Algorithms, pseudo-code, code
- Logistics
  - Semester and weekly schedule
  - Labs organization
- New stuff
  - **Complexity Analysis**

Who am I ? Overview of my academic journey

1989-00 Univerzita Komenského, Slovakia

2000-02 University of Aizu, Japan

2002-06 Carnegie Mellon University, USA

Pittsburgh University, USA

Harvard University, USA

2007-11 University of Warwick, UK

EPSRC, Robert's fund, IAS European Frontiers, Reinvention Centre Collaboration Fund,
RCUK academic fellowship

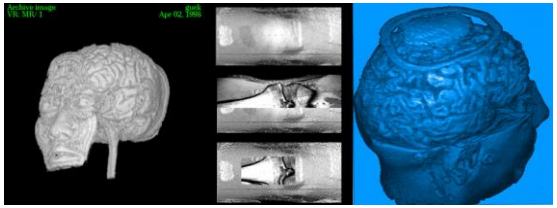
2011 - 2018 Manchester Met University, UK

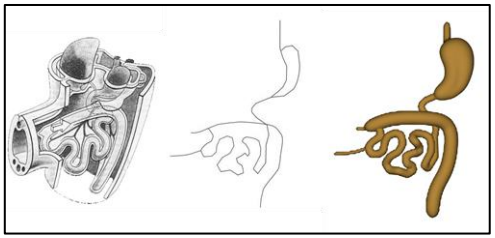
KTP, H2020, MAN MET research fund, Erasmus teaching grant

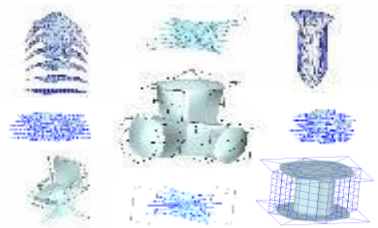
2019 - 2024 Liverpool John Moores Uni, UK

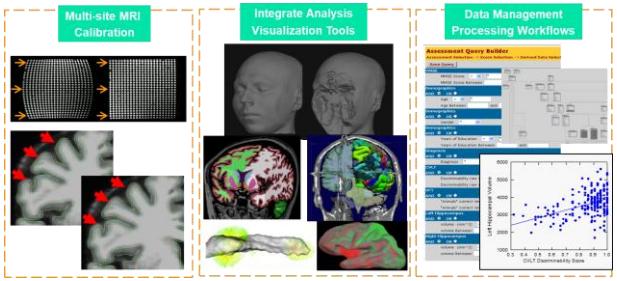
2024 – now Chester Uni, UK

2021 - now FIIT, STU

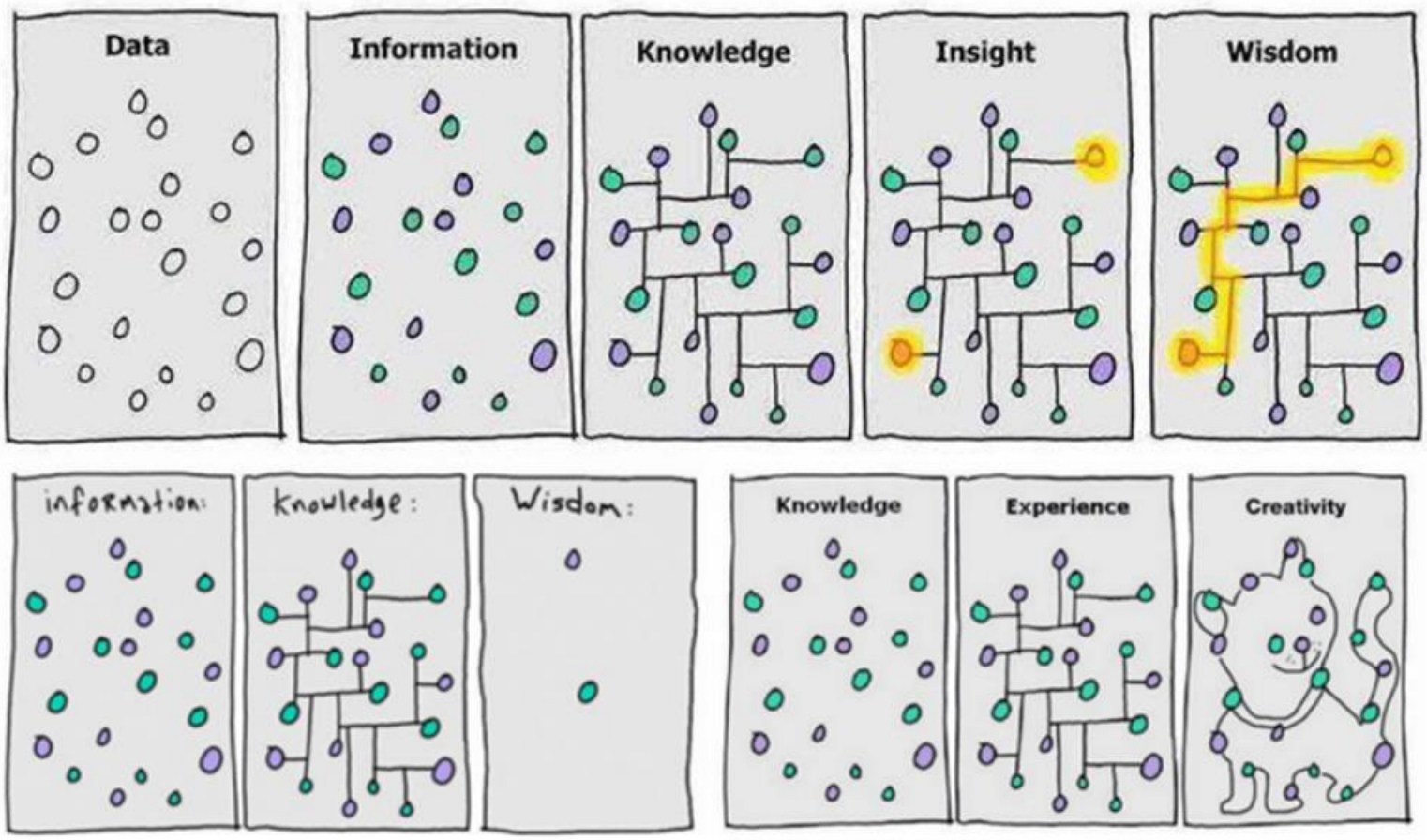








Refresher – Data Structure I.



https://www.gapingvoid.com

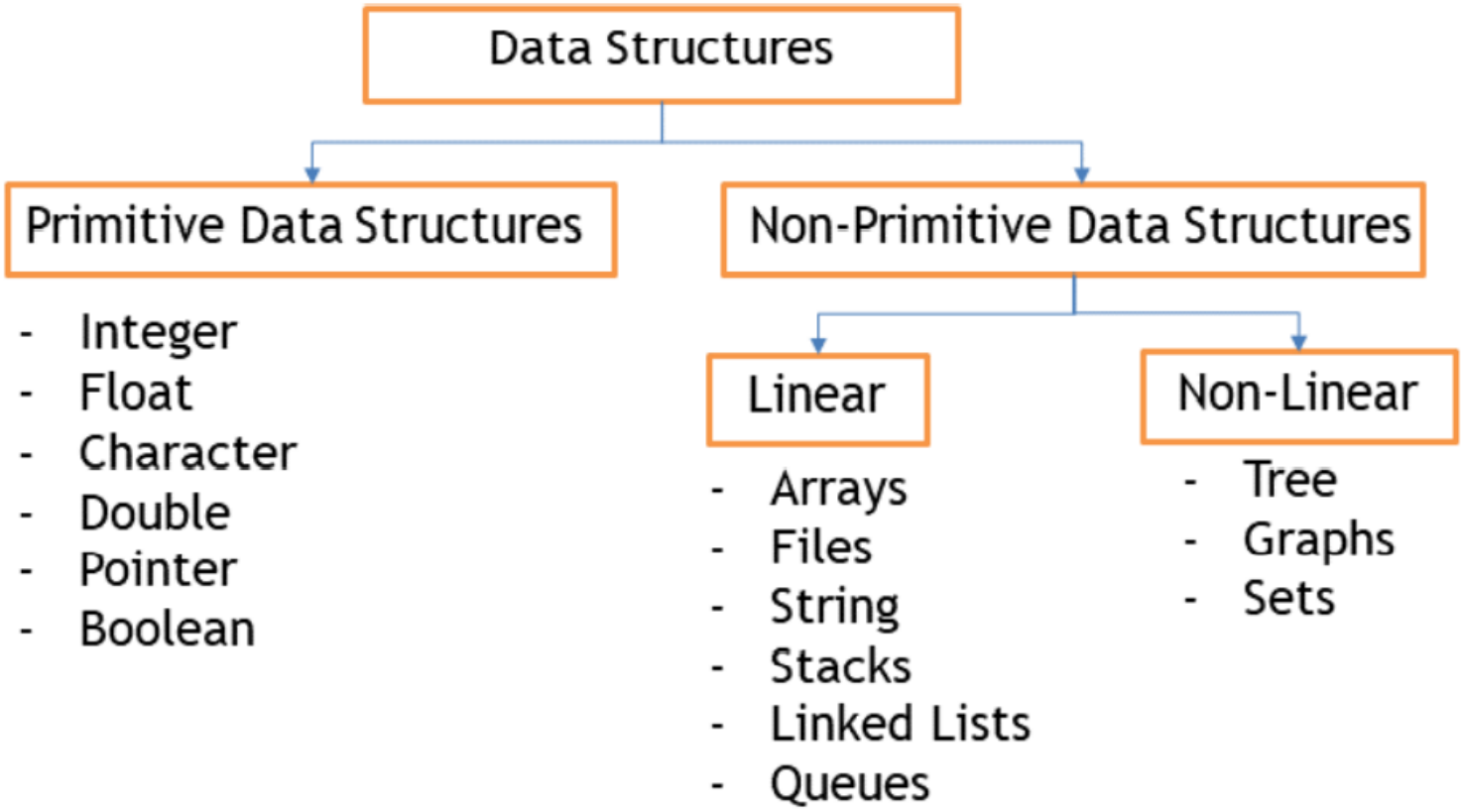
Refresher – Data Structure II

Data
Data is information that has been translated into a form that is efficient for movement or processing.

Data Type
A data type is a classification that specifies which type of value a variable has and what type of operations can be applied to it.

Data Structure
A data structure is a data organization, management, and storage format that enables efficient access and modification.

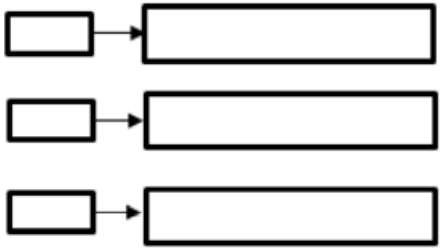
Refresher – Data Structure Taxonomy



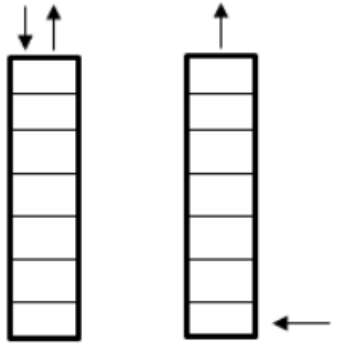
Refresher – Data Structure Landscape

Arrays

Hash Table

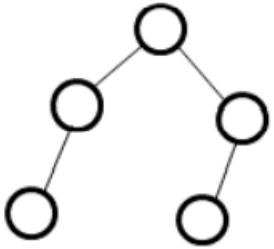


Stack and Queue

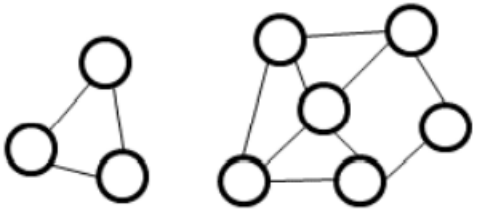


Linked List

Tree



Graph



Data Structure - Requirements

The five requirements of any data structure:

1. How to **Access** (one item / all items)
2. How to **Insert** (at end / at position)
3. How to **Delete** (form end / from position)
4. How to **Find** (if exists / what location)
5. How to **Sort** (sort in place / create sorted version)

C++ Programming

- Datatypes
- Structure and Union
- Functions
- Decision Structure (if-then-else, for, while)
- Class
- Template
- Memory Address
- Dynamic Memory management
- Static and Dynamic allocation
- Pointers

Algorithm, Pseudo-Code and Code

Algorithm

An algorithm expresses computational processes for solving a problem in terms of the actions. An algorithm is merely the sequence of steps taken to solve a problem.

Pseudo-Code

Pseudocode is human-friendly and informal but structured english for describing algorithms. It cannot be understood by a machine, but helps programmers develop algorithms.

Code

Implementation of a given algorithm in a specific programming language that can be interpreted or compiled by a machine.

Example of Pseudo-Code

Algorithm 1 Mystery Procedure

Input: Array A of n elements.
Output: Array sorted

```
1:  procedure Mystery(A, n)
2:      for j←1 to n do
3:          key ← A[j]
4:          i ← j − 1
5:          while i > 0 and A[i] > key do
6:              /* swap element */
7:              A[i + 1] ← A[i]
8:              i ← i − 1
9:          end while
10:         A[i + 1] ← key
11:     end for
12: end procedure
```

Example of Code

```
void Mystery(int A*, int n)
{
 int i, key, j;
 for (j = 1; j < n; j++)
     {
      key = A[j];
      i = j - 1;

      while (i >= 0 && A[i] > key)
            {
             A[i + 1] = A[i];
             i = i - 1;
            }
      A[i + 1] = key;
     }
}
```

Comparison – Pseudo-code vs CODE

Algorithm 1 Mystery Procedure

Input: Array A of n elements.
Output: Array sorted

```
1: procedure Mystery(A, n)
2:     for j←1 to n do
3:         key ← A[j]
4:         i ← j − 1
5:         while i > 0 and A[i] > key do
6:             /* swap element */
7:             A[i + 1] ← A[i]
8:             i ← i − 1
9:         end while
10:        A[i + 1] ← key
11:    end for
12: end procedure
```

```
void Mystery(int A*, int n)
{
 int i, key, j;
 for (j = 1; j < n; j++)
     {
      key = A[j];
      i = j - 1;

      while (i >= 0 && A[i] > key)
            {
             A[i + 1] = A[i];
             i = i - 1;
            }
      A[i + 1] = key;
     }
}
```

Semester Schedule

Lectures: **16/9, 30/9, 14/10, 21/10, 04/11, 11/11, 18/11**

- Monday 8-10
- Monday 11-13

Labs: Martina Billichová, Dmytro Furman, Timotej Kralik, Michal Lüley a Martin Druzbacky

Assessment:

- 2 x assignments during the semester
  - Written test on theoretical analysis of an algorithm (20%) – **21/10/2024**
  - Practical implementation and analysis of an algorithm (30%) – deadline **6/12/2024**
- 1 x exam
  - Theoretical analysis and pseudocode of algorithms (50%) – **January 2025**

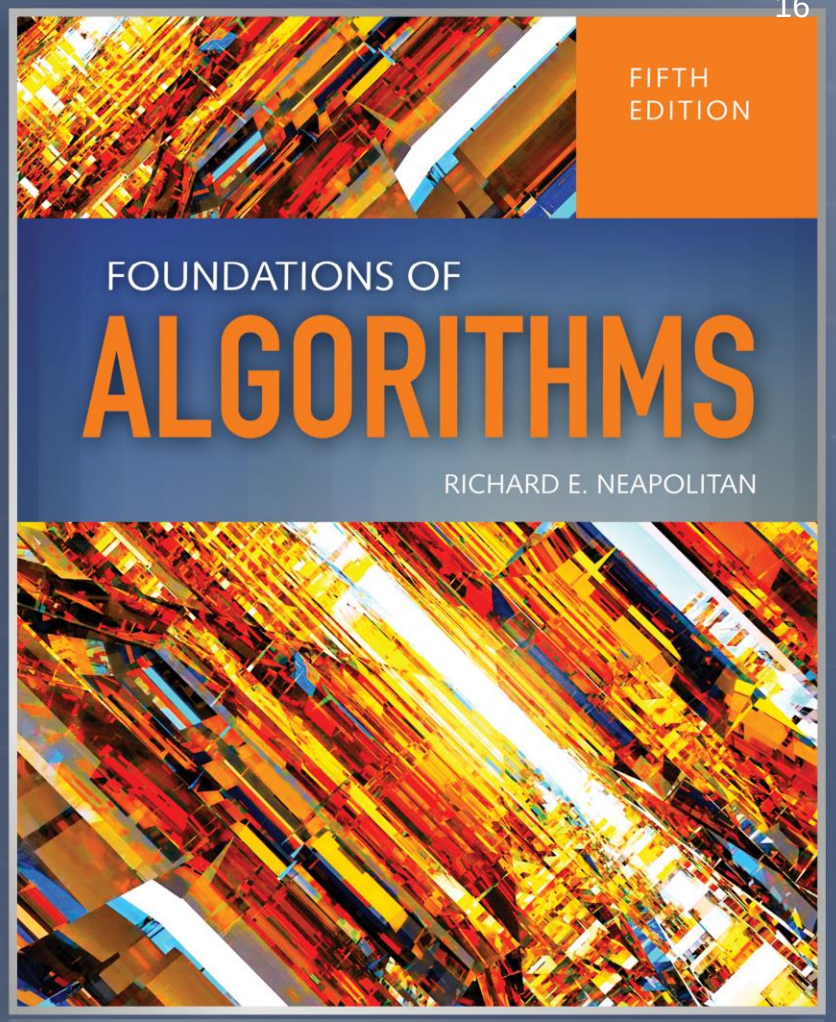
Semester Schedule II.

- 16/9
  - Session 1: Intro, Logistics, Algorithms: Efficiency, Analysis, and Order
  - Session 2: Divide-and-Conquer
- 30/09
  - Session 1: Dynamic programming,
  - Session 2: The Greedy approach
- 14/10
  - Session 1: Backtracking
  - Session 2: Branch-and-Bound
- **21/10**
  - Session 1: written test (1h) – material from first 5 labs
  - Session 2: written test (1h) – material from first 5 labs
- 04/11
  - Session 1: Computational Complexity: The Sorting Problem
  - Session 2: Computational Complexity: The Searching Problem
- 11/11
  - Session 1: An introduction to the theory of NP
  - Session 2: Limitations of Algorithm Power
- 18/11
  - Session 1: Wrap up, Q&A session

- Practical work submission deadline **06/12 5PM**

Algorithms:
Efficiency, Analysis, and Order

Chapter 1



Objectives

- Analyze techniques for solving problems
- Define an algorithm
- Define growth rate of an algorithm as a function of input size
- Define Worst case, average case, and best-case complexity analysis of algorithms
- Classify functions based on growth rate
- Define growth rates: Big O, Theta, and Omega

Methodology

- Approach to solving a problem
- Independent of Programming Language
- Independent of Style
- Sequential Search versus Binary Search
- Which technique results in the most efficient solution?

Problem?

- A question to which an answer is sought
- Parameters
  - Input to the problem
  - Instance: a specific assignment of values to the input parameters
- Algorithm:
  - Step-by-step procedure
  - Solves the Problem

Sequential Search vs Binary Search – Worst Case

- Input Array S size n
- $X \notin S$
- Sequential Search: n operations
- Binary Search: lg n + 1 operations

Fibonacci: Iterative vs Recursive

- $Fib_0 = 0$
- $Fib_1 = 1$
- $Fib_n = Fib_{n-1} + Fib_{n-2}$
- Calculate the nth Fibonacci Term:
  - Recursive calculates $2^{n/2}$ terms
  - Iterative calculates n+1 terms

Fast computers *vs* efficient algorithms [CLRS 1]

Many recent innovations rely on

- fast computers
- efficient algorithms.

Which is more important?

The importance of efficient algorithms

The *cost* of an algorithm can be quantified by the number of steps $T(n)$ in which the algorithm solves a problem of size n.

Imagine that a certain problem can be solved by four different algorithms, with $T(n) = n, n^2, n^3$, and 2^n, respectively.
Question: what is the maximum problem size that the algorithm can solve in a given time?

Assume that a computer is capable of 10^{10} steps per second.

Cost $T(n)$ (Complexity)	Maximum problem size solvable in		
	1 second	1 hour	1 year
n	10^{10}	3.6×10^{13}	3×10^{17}
n^2	10^5	6×10^6	5×10^8
n^3	2154	33000	680000
2^n	33	45	58

Faster computers *vs* more efficient algorithms

Suppose a faster computer is capable of 10^{16} steps per second.

Cost $T(n)$	Max. size before	Max. size now
n	s_1	$10^6 \times s_1$
n^2	s_2	$1000 \times s_2$
n^3	s_3	$100 \times s_3$
2^n	s_4	$s_4 + 20$

A $10^6\times$ increase in speed results in only a factor-of-100 improvement if cost is n^3, and only an additive increase of 20 if cost is 2^n.

Conclusions As computer speeds increase ...

1. ... it is algorithmic efficiency that really determines the increase in problem size that can be achieved.
2. ... so does the size of problems we wish to solve.
   Thus, designing efficient algorithms becomes even more important!

From *Algorism* to *Algorithms*

Invented in India around AD 600, the *decimal system* was a revolution in quantitative reasoning. Arabic mathematicians helped develop arithmetic methods using the Indian decimals.

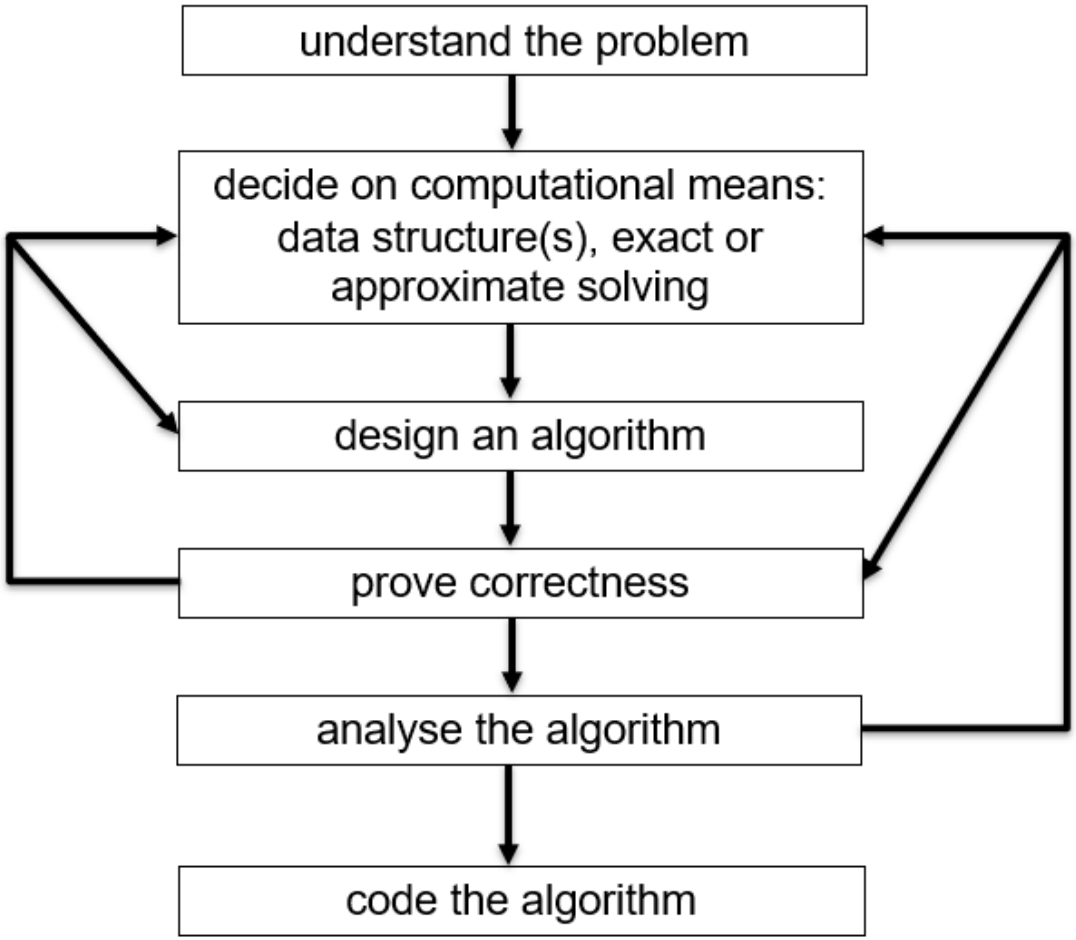
A 9th-century Arabic textbook by the Persian *Al Khwarizmi* was the key to the spread of the Indian-Arabic decimal arithmetic. He gave methods for basic arithmetic (adding, multiplying and dividing numbers), even the calculation of square roots and digits of π.

Derived from 'Al Khwarizmi', *algorism* means rules for performing arithmetic computations using the Indian-Arabic decimal system.

The word "algorism" devolved into *algorithm*, with a generalisation of the meaning to

Algorithm: a finite set of well-defined instructions for accomplishing some task.

Algorithm Design and Analysis



Evaluating algorithms

Two questions we ask about an algorithm

1. Is it correct?
2. Is it efficient?

Correctness is of utmost importance.
It is easy to design a highly efficient but incorrect algorithm.

Efficiency with respect to:

- Running time
- Space (amount of memory used)
- Network traffic
- Other features (e.g. number of times secondary storage is accessed)

Proving correctness and analysing the efficiency of programs are difficult problems, in general. Take for example the Collatz program: starting from a positive integer x repeat "if x is even then $x = x/2$, else $x = (3x + 1)/2$" until $x = 1$. It is an open problem whether it terminates for every positive x.

Computational Complexity

- Computational complexity **is the number of resources required to run an algorithm.**
- Focus is given to time and memory requirements.
- **Complexity of a problem** = complexity of the optimal algorithms solving the problem.
- The resource that is most commonly considered is **time**.
- Another important resource is the **size of memory** needed for running algorithms.
- Complexity in **time** or **space/memory**.
- Complexity expressed with respect to the **size of the input**.
- The **number of arithmetic operations** is another resource that is commonly used.
- Impossible to count the number of steps of an algorithm on all inputs.
- Therefore, several complexity functions are commonly used.

Space Complexity

- When memory was expensive, we focused on making programs as space efficient as possible and developed schemes to make memory appear larger than it really was

ATARI 800XL – 64KB memory
HDD ➔ magnetic tape

- Space complexity is still important in the field of embedded computing (handheld computer based equipment like cell phones, palm devices, etc.)

Time Complexity

- Is the algorithm "**fast enough**" for my needs
- How much longer will the algorithm take if I increase the amount of data it must process
- Given a set of algorithms that accomplish the same thing, which is the right one to choose

Degrees of Complexity

An algorithm shows different degrees of complexity when exposed to different configuration of inputs.

Best-Case Analysis
Pure Optimism. In the most ideal case, the minimum number of primitive operations for any input size n (with n not equal to 1).

Average-Case Analysis
Being Practical. The maximum number of primitive operations performed on an input of size n (with n not equal to infinity).

Worst-Case Analysis
Pure Pessimism. In the worst case, the weighted average number of primitive operations on input size n.

Asymptotic Analysis

- Asymptotic analysis is the process of calculating the running time of an algorithm in mathematical units to find the program's limitations, or run-time performance.
- Asymptotic analysis defines the mathematical boundation of an algorithm run-time.
- We want to know is how long these algorithms take.
- Asymptotic Analysis is the evaluation of the performance of an algorithm in terms of just the input size n.
- Give an idea of the limiting behavior of a function.
- Determine the best case, worst case and average case time required to execute a given function.
- Focus on how fast a function grows with the input size.
- We call this the rate of growth of the running time.

Tilde Approximations

- Tilde notation is used when we want to make a simple approximation of a complex function.
- Assume that the lower order terms contributes relatively small and thus considers only leading terms.
- Use tilde notation to develop simpler approximate expressions.
- Throw away low-order terms that complicate formulas.
- Simply drops the lower order terms.
- Tilde notation ($\sim$) defines both the upper bound and lower bound of an algorithm.
- It gives a sharper approximation of an algorithm.
- In asymptotic analysis we are only interested to know about the growth of a function for a larger value of n.

Tilde Approximations II

Tilde Notation ($\sim$)

Tilde notation is used when we want to make a simple approximation of a complex function. It simply drops the lower order terms. It is denoted by $\sim g(n)$. We write $\sim f(n)$ to represent any function that when divided by $f(n)$ approaches 1 as n grows. We write $g(n) \sim f(n)$ to indicate that $g(n)/f(n)$ approaches 1 as n grows. Mathematically, we can express it as :

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$$

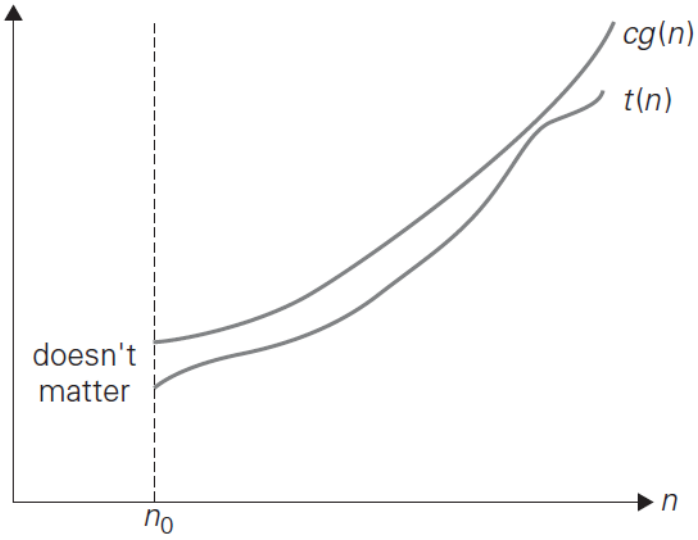
Asymptotic notations and basic efficiency classes

Informally, $O(g(n))$ is the set of all functions with a lower or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). Thus, to give a few examples, the following assertions are all true:

$$n \in O(n^2), \qquad 100n + 5 \in O(n^2), \qquad \frac{1}{2}n(n-1) \in O(n^2).$$

$$n^3 \notin O(n^2), \qquad 0.00001n^3 \notin O(n^2), \qquad n^4 + n + 1 \notin O(n^2).$$

O-notation



DEFINITION A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \leq 100n + n \text{ (for all } n \geq 5) = 101n \leq 101n^2.$$

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively.

Properties of Big-O

Lemma

Let $f, g, h : \mathbb{N} \longrightarrow \mathbb{R}^+$. *Then:*

1. *For every constant* $c > 0$, *if* $f \in O(g)$ *then* $cf \in O(g)$.
2. *For every constant* $c > 0$, *if* $f \in O(g)$ *then* $f \in O(cg)$.
3. *If* $f_1 \in O(g_1)$ *and* $f_2 \in O(g_2)$ *then* $f_1 + f_2 \in O(g_1 + g_2)$.
4. *If* $f_1 \in O(g_1)$ *and* $f_2 \in O(g_2)$ *then* $f_1 + f_2 \in O(\max(g_1, g_2))$.
5. *If* $f_1 \in O(g_1)$ *and* $f_2 \in O(g_2)$ *then* $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$.
6. *If* $f \in O(g)$ *and* $g \in O(h)$ *then* $f \in O(h)$.
7. *Every polynomial of degree* $l \geq 0$ *is in* $O(n^l)$.
8. *For any* $c > 0$ *in* $\mathbb{R}$, *we have* $\lg(n^c) \in O(\lg(n))$.
9. *For every constant* $c, d > 0$, *we have* $\lg^c(n) \in O(n^d)$.
10. *For every constant* $c > 0$ *and* $d > 1$, *we have* $n^c \in O(d^n)$.
11. *For every constant* $0 \leq c \leq d$, *we have* $n^c \in O(n^d)$.

Example

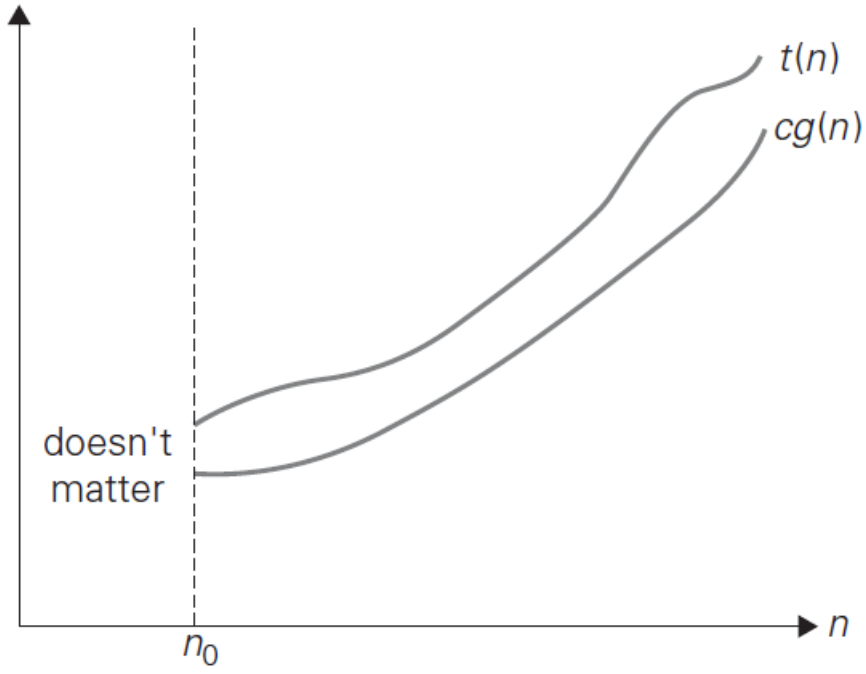
Example. Show that

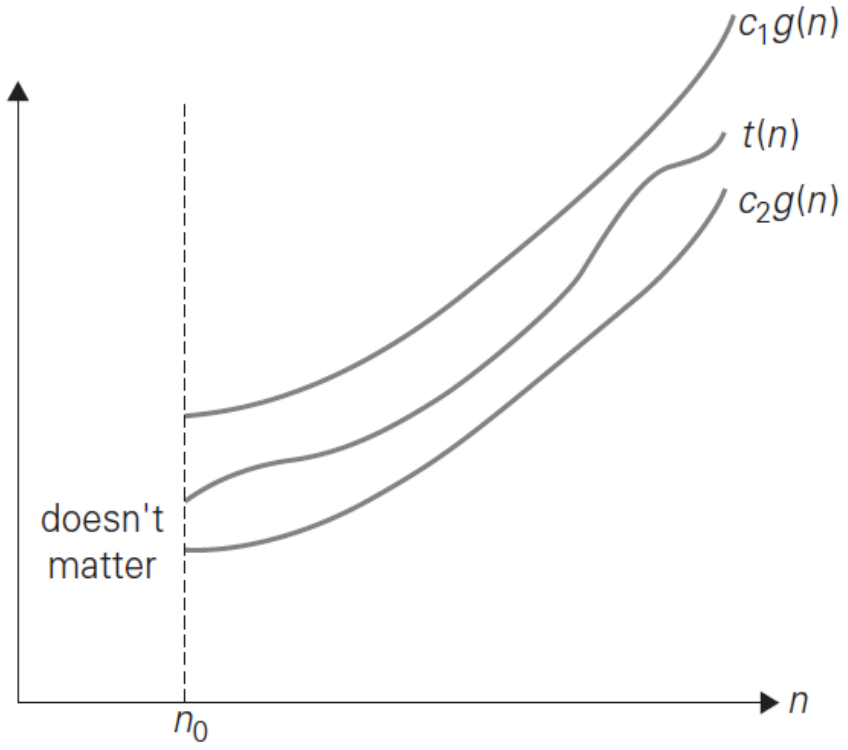
$$57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 \in O(n^3)$$

by appealing to Lemma 1.

$$\begin{aligned}
\lg^5(n) &\in O(n) && \because 9 \\
4n^2 \cdot \lg^5(n) &\in O(4n^3) && \because 5 \\
57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 &\in O(57n^3 + 4n^3 + 17n + 498) && \because 3 \\
57n^3 + 4n^3 + 17n + 498 &\in O(n^3) && \because 7 \\
57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 &\in O(n^3) && \because 6
\end{aligned}$$

Ω and θ notations





$t(n) \in \Omega(g(n))$.

$t(n) \geq cg(n)$ for all $n \geq n_0$.

$t(n) \in \Theta(g(n))$.

$c_2 g(n) \leq t(n) \leq c_1 g(n)$ for all $n \geq n_0$.

Examples

1. $5n^3 + 88n = \Theta(n^3)$
2. $2 + \sin(\lg n) = \Theta(1)$
3. $n! = \Theta(n^{n+1/2}e^{-n})$.
   Consequence of Stirling's Approximation.
4. For all $a, b > 1$, $\log_a n = \Theta(\log_b n)$.
   Consequence of the relation $\log_b a = \frac{\log_c a}{\log_c b}$.
   From now on, we will be "neutral" and write $\Theta(\log n)$, without specifying the base of the logarithm.

Example

1. $n^n = \Omega(n!)$
2. $2^n = \Omega(n^{10})$.

Question. OK, we have seen that $\log_a n$ is a Big-Theta of $\log_b n$. But is $2^{\log_a n}$ a Big-Theta of $2^{\log_b n}$?

No! Recall that $a^{\log_b c} = c^{\log_b a}$ for all $a, b, c > 0$.

Using this relation, we have $2^{\log_a n} = n^{\log_a 2}$ and $2^{\log_b n} = n^{\log_b 2}$.
If $a \neq b$, we have two different powers of n, but $n^c = \Theta(n^d)$ only if $c = d$.

Revision

Logarithms.

$\log_2 n$ is sometimes written $\lg n$, and $\log_e n$ is sometimes written $\ln n$.

Recall the following useful facts. Let $a, b, c > 0$.

$$\begin{aligned} a &= b^{\log_b a} \\ \log_b a &= \frac{\log_c a}{\log_c b} \\ \log_b a &= \frac{1}{\log_a b} \\ a^{\log_b c} &= c^{\log_b a} \end{aligned}$$

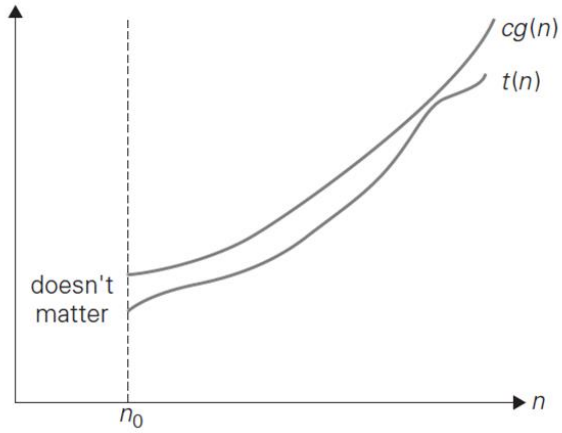
A form of Stirling's approximation.

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

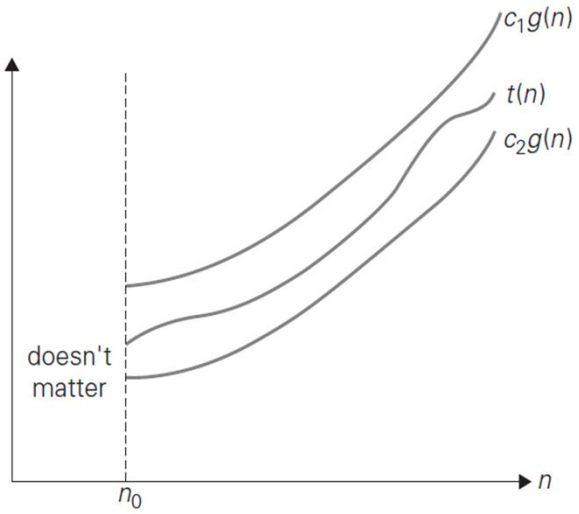
Comparing Orders of Growth

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

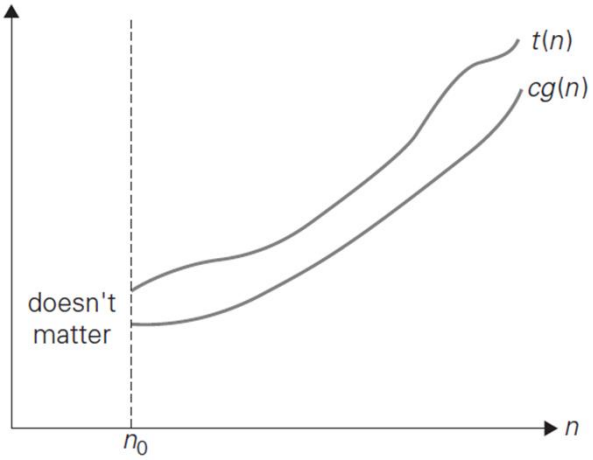
Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.



$t(n) \in O(g(n))$



$t(n) \in \Theta(g(n))$.

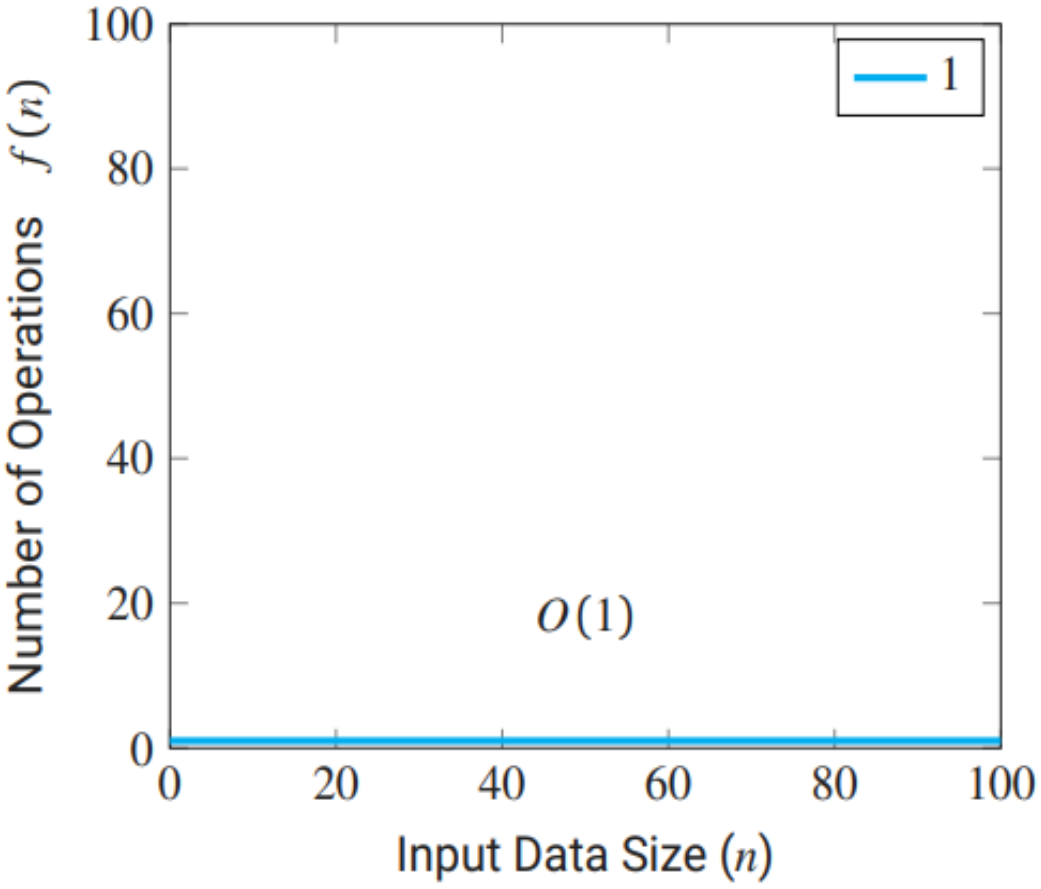


$t(n) \in \Omega(g(n))$.

Big-O Analysis

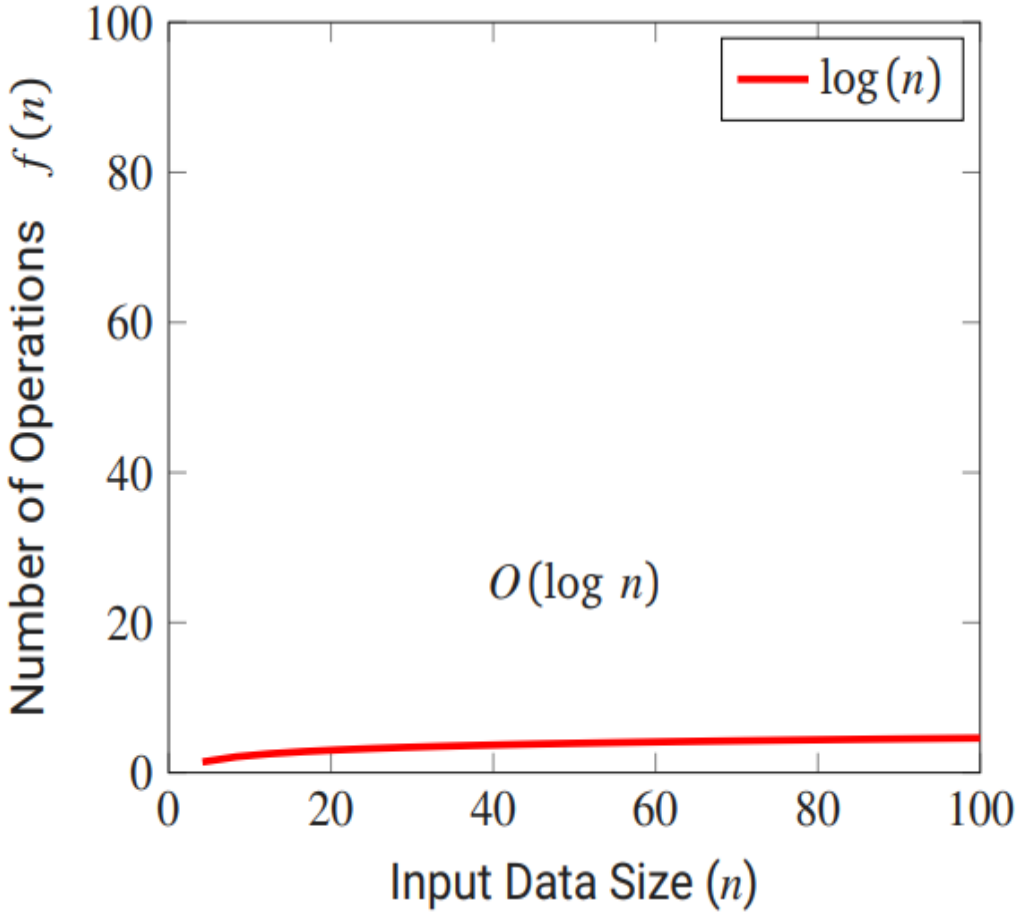
- Given the input of size n, Big-O describes an upper bound on number of relevant, primitive operations algorithm needs to perform.
- Big-O can be used to describe space complexity as well as time complexity, or any other complexity.
- This notation is basically used to describe the asymptotic upper bound.
- The time complexity of an algorithm is commonly expressed using Big-O notation.
- Big-O notation describes the execution time required or the spaced used by an algorithm.
- Big-O actually represents how fast things can go really bad.
- Big-O notation specifically describes the worst-case scenario.

Constant Time Complexity



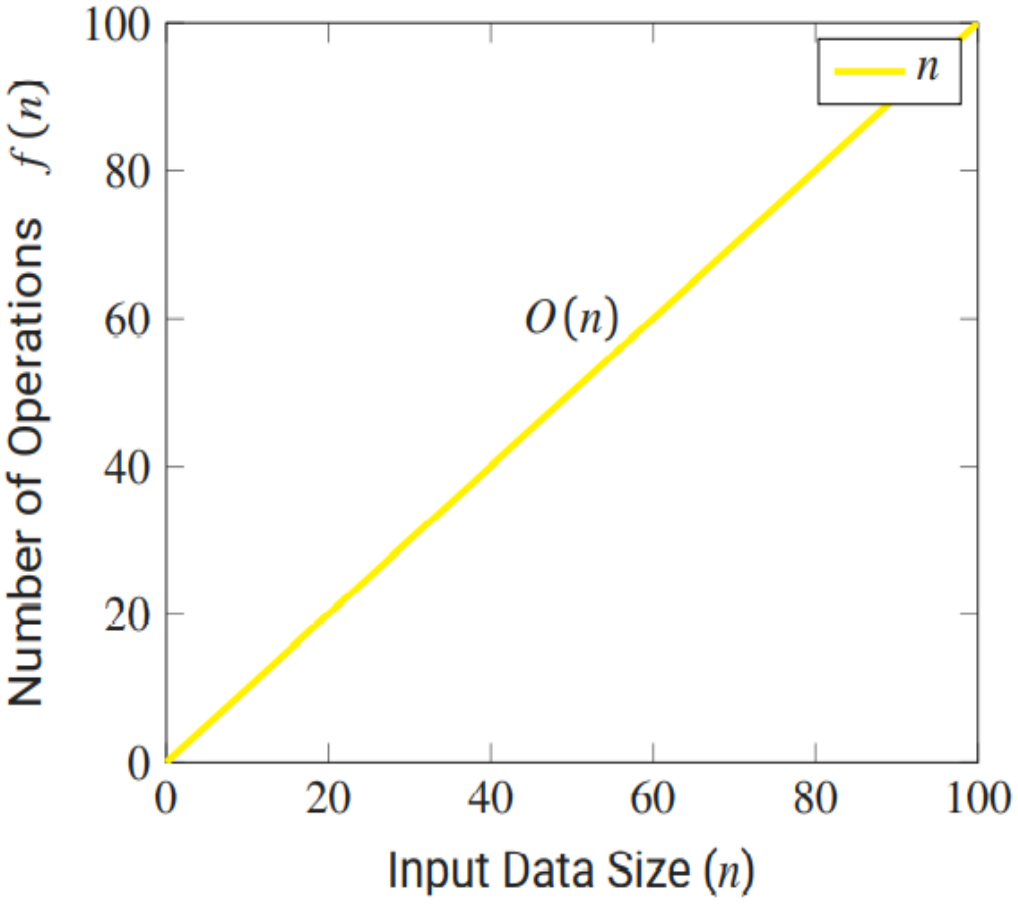
- Constant Time Complexity $O(1)$.
- Constant growth rate.
- Fastest time complexity.
- Constant-time method is order 1.
- Not dependent on the input size n.
- Irrespective of the input size n.
- The runtime will always be the same.

Logarithmic Time Complexity



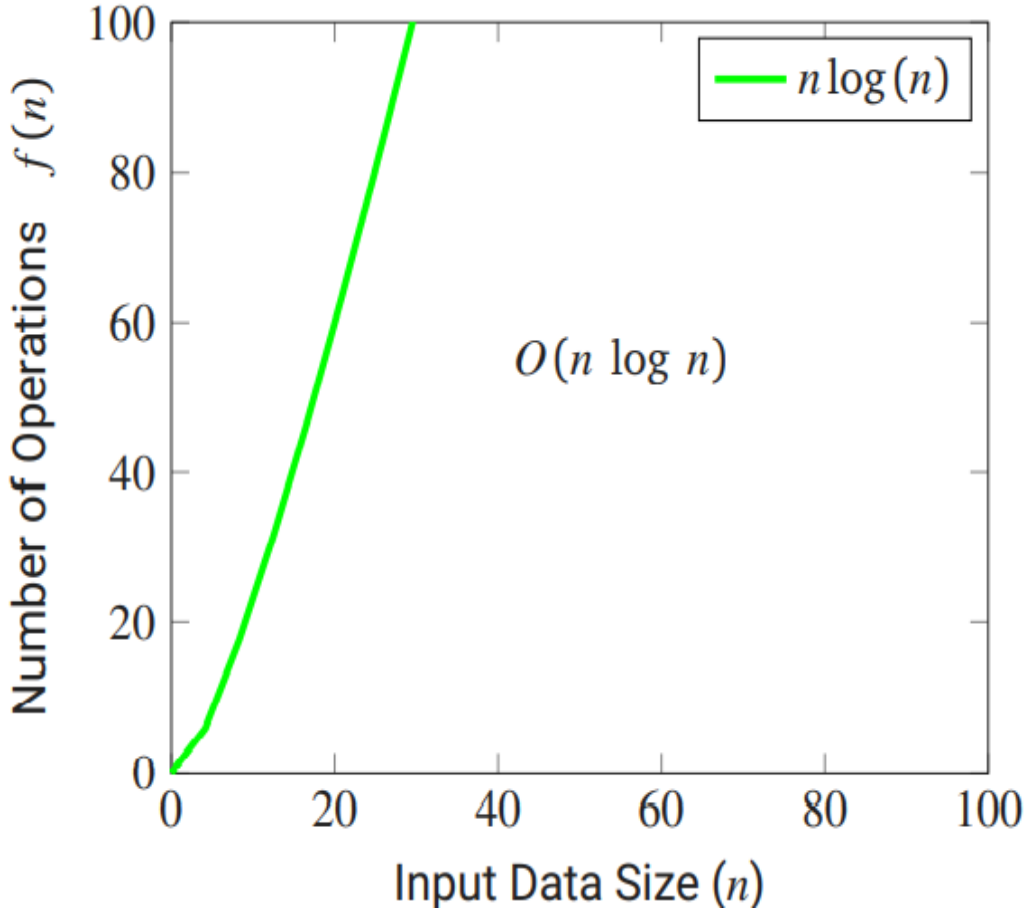
- Time Complexity $O(\log\ n)$.
- Logarithmic order complexity.
- Considered highly efficient.
- The ratio of the number of operations to the size of the input decreases and tends to zero when n increases.
- Execution time increases, but at a decreasing rate.
- Commonly found in operations on binary trees or search.

Linear Time Complexity



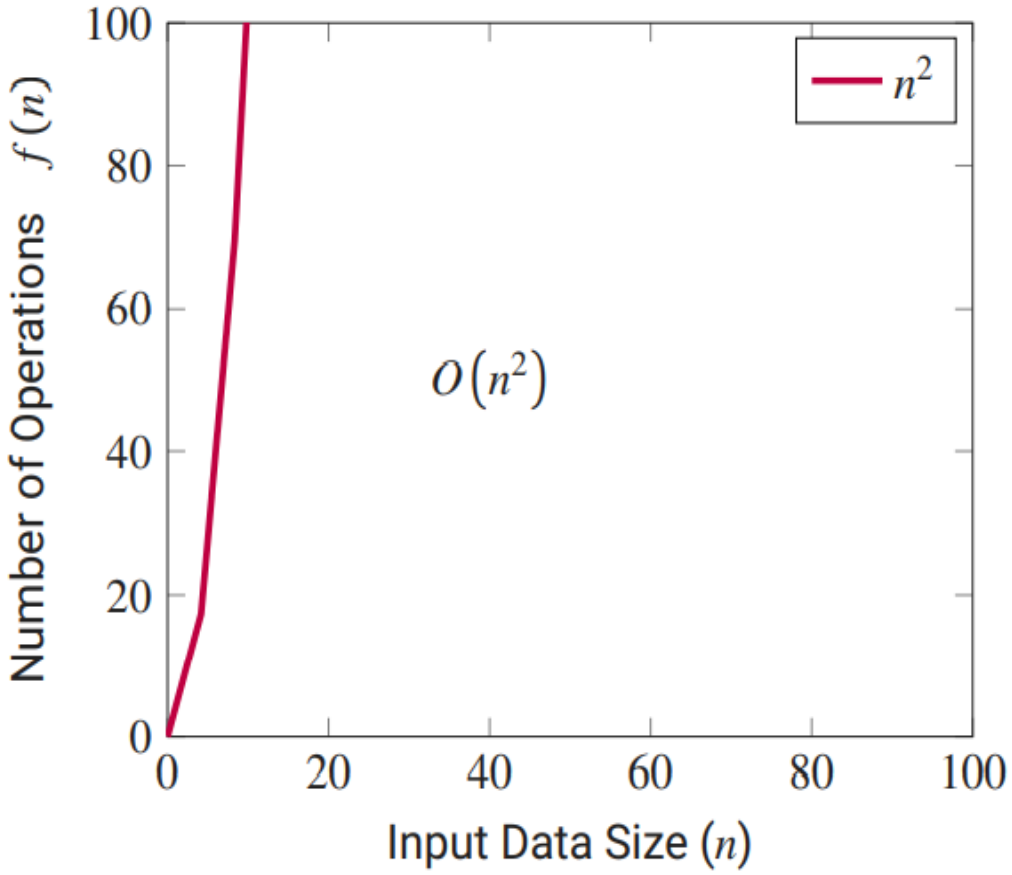
- Time Complexity $O(n)$.
- The running time increases at most linearly with the length of the input.
- Takes proportionally longer to complete as the input grows.
- Number of elements and number of steps are linearly dependent.
- Best possible time complexity to sequentially read its entire input.

Linearithmic Time Complexity



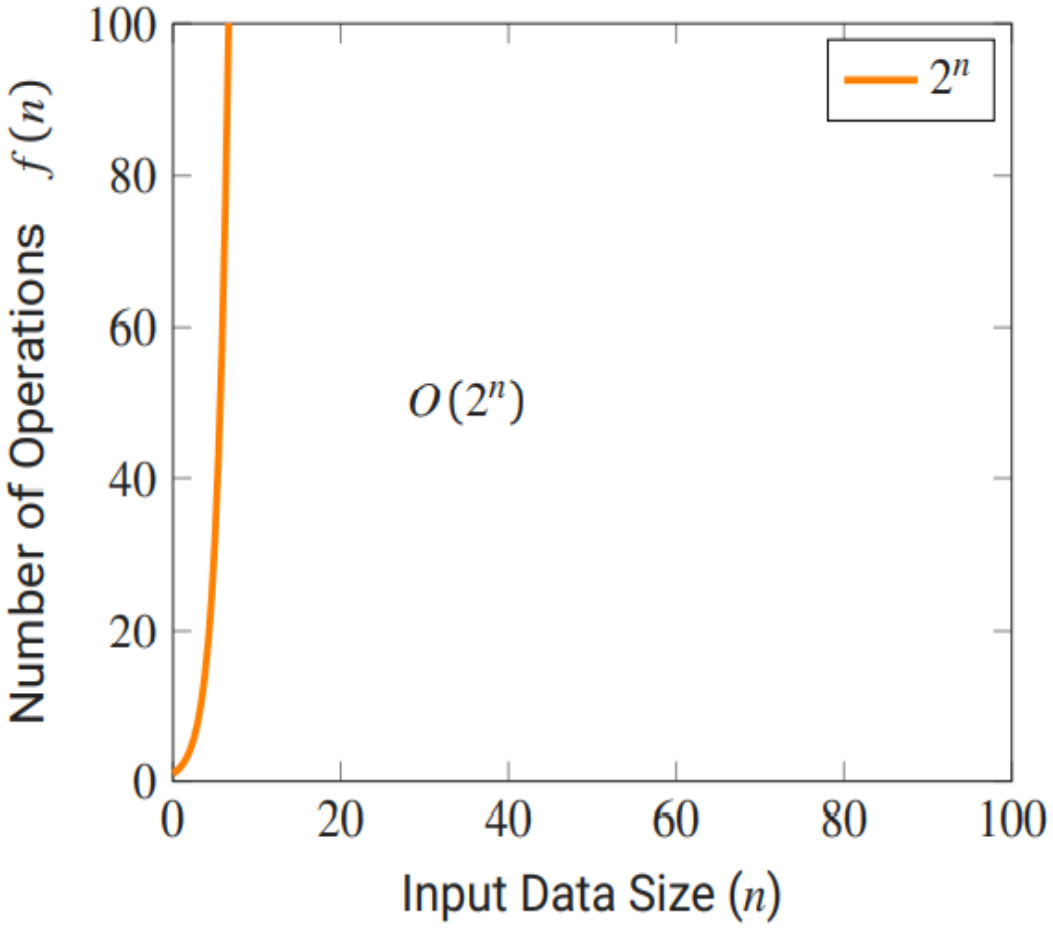
- Time Complexity $O(n \log n)$.
- Takes an amount of time between linear and polynomial.
- The worst of the best complexity.
- A moderate complexity that floats around linear time.
- Slightly slower than a linear algorithm.
- Still much better than a quadratic algorithm.

Quadratic Time Complexity



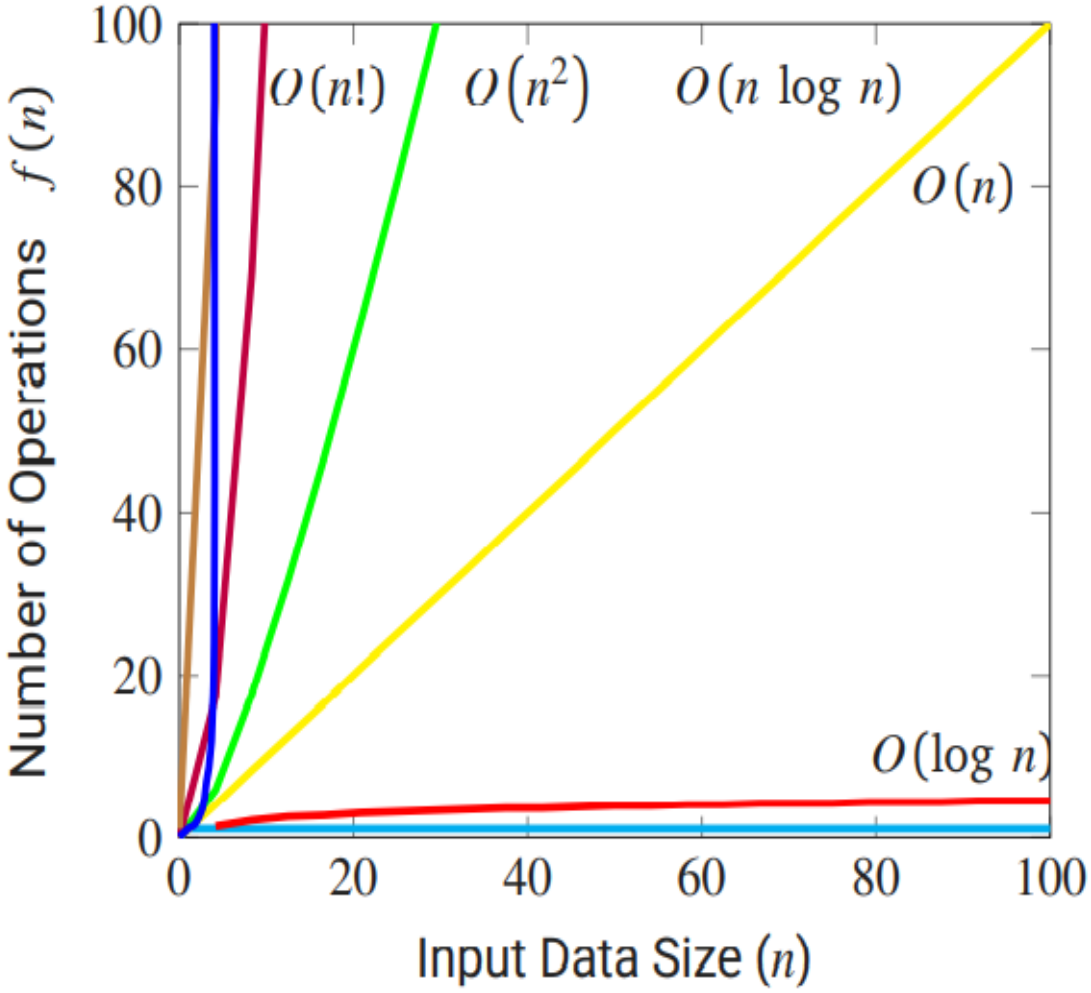
- Time Complexity $O(n^2)$.
- Growth rate of n^2.
- Performance is directly proportional to the squared size of the input.
- If the input is size 2, it will do four operations.
- With quadratic complexity execution time increases at an increasing rate.
- It is slower than logarithmic time.

Exponential Time Complexity



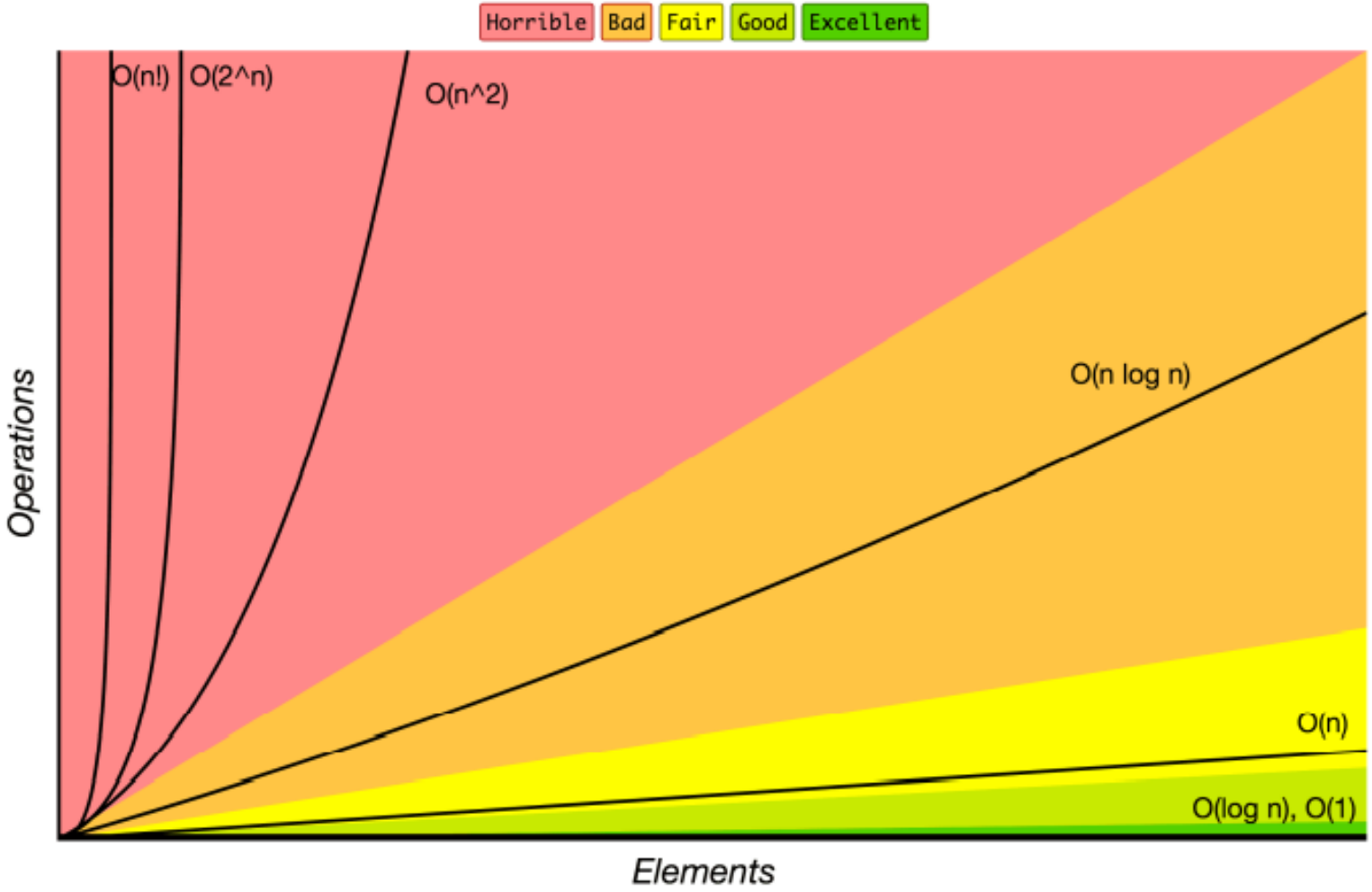
- Time Complexity $O(2^n)$.
- Denotes an algorithm whose growth doubles anytime an input unit increases by 1.
- Exponential time complexity is usually seen in Brute-Force algorithms.
- When the input is big enough, resource consumption will quickly approach infinity.

Order of Growth



- $O(1)$: constant
- $O(\log n)$: logarithmic
- $O(n)$: linear
- $O(n \log n)$: linearithmic
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
- $O(n!)$: factorial
- $O(n^c)$: polynomial

Order of Growth II



Order of Growth and Time

Some order magnitude on a specific (not generalizable) example:

- $O(1) \implies T(n) = 1$ second.
- $O(\log n) \implies T(n) = (1 \cdot \log 16) / (\log 8) = 4/3$ seconds.
- $O(n) \implies T(n) = (1 \cdot 8) = 2$ seconds.
- $O(n \log n) \implies T(n) = (1 \cdot 16 \cdot \log 16) / (8 \cdot \log 8) = 3$ seconds.
- $O(n^2) \implies T(n) = (1 \cdot 16^2) / 8^2 = 4$ seconds.
- $O(n^3) \implies T(n) = (1 \cdot 16^3) / 8^3 = 8$ seconds.
- $O(2^n) \implies T(n) = (1 \cdot 2^{16}) / 2^8 = 256$ seconds.

Big-O vs Tilde

Big-O Notation

- It generally defines the upper bound of an algorithm.
- It defines the worst case.
- It show the order of growth of algorithms.

Tilde Notation

- It defines both the upper bound and lower bound of an algorithm.
- It mostly defines the average case.
- It simply drops the lower order terms.

Proving Tilde Complexity

$$
\begin{aligned}
8n + 24 &\sim 8n \\
40n + 24 &\sim 40n \\
8n + 48 &\sim 8n \\
2n + 56 &\sim 2n \\
n^2 + 32n + 24 &\sim n^2 \\
4n^2 + 32n + 24 &\sim 4n^2 \\
6n^3 + 20n + 16 &\sim 6n^3 \\
6n^3 + 100n^{4/3} + 16 &\sim 6n^3 \\
6n^3 + 17n^2 \log(n) + 7n &\sim 6n^3
\end{aligned}
$$

Proving Big-O Complexity

$$t(n) \in O(g(n))$$
$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$

Using the definition of big-O notation:

- $7n-2$ is $O(n)$

need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$.
This is true for $c = 7$ and $n_0 = 1$.

- $3n^3 + 20n^2 + 5$ is $O(n^3)$

need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$.
This is true for $c = 4$ and $n_0 = 21$.

- $3\log(n) + 5$ is $O(log\ n)$

need $c > 0$ and $n_0 \geq 1$ such that $3\log(n) + 5 \leq c \cdot log\ n$ for $n \geq n_0$.
This is true for $c = 8$ and $n_0 = 2$.

Tilde vs Order of Growth

Function	Tilde Approximation	Order of Growth
$n^3/6 - n^2/2 + n/3$	$\sim n^3/6$	$O(n^3)$
$n^2/2 - n/2$	$\sim n^2/2$	$O(n^2)$
$\log n + 5$	$\sim \log n$	$O(\log n)$
3	~ 3	$O(1)$

Elementary Operations

- Primitive Operations ⟺ Elementary Operations.
- A primitive operation is a basic step that can be performed in constant time.
- Including: mathematical constants and arithmetic operations.
- Including: assigning a value to a variable, as well as comparisons and tests.
- Including: evaluating an expression.
- Including: indexing into an array.
- Including: calling a method, returning from a method.
- Easily identifiable and independent from the programming language.
- Non-elementary operations: have a unit cost and bit costs.
- Non-elementary operations: for and while loop.
- Non-elementary operations: nested loops, simple recursive functions.

Elementary Operations Count

- examine a piece of code and predict the number of instructions to be executed
- e.g. for each instruction predict how many times each will be encountered as the code runs

Inst #	Code	F.C.
1	for (int i=0; i< n ; i++)	n+1
2	{ cout << i;	n
3	p = p + i;	n
	}	———
		3n+1

totaling the counts produces the F.C. (frequency count)

discarding constant terms produces : 3n

clearing coefficients : n

picking the most significant term: n

Big O = O(n)

Elementary Operations Count II

Inst #	Code	F.C.	F.C.
1	for (int i=0; i< n ; i++)	n+1	n+1
2	for (int j=0 ; j < n; j++)	n(n+1)	n^2+n
3	{ cout << i;	n*n	n^2
4	p = p + i;	n*n	n^2
	}		———
			$3n^2+2n+1$

discarding constant terms produces : $3n^2+2n$

clearing coefficients : n^2+n

picking the most significant term: n^2

Big O = $O(n^2)$

Examples of Algorithm Complexity I

```
int function(int n)
 { if (n>1)
    { return 3*function(n/2) + 1; }
 }
```

```
  int count=0;
  for (int i=n; i>0 ; i/=2)
     { count++; }
```

```
  int i=1;
  while (i<n)
        { sum=sum+1;
          i=i*2;
        }
```

$O(\log n)$

Examples of Algorithm Complexity II

```
void function(int n)
{
 int i,j,k;
 int count=0;

 for (i=n/2 ; i<=n ; i++)
     {
      for (j=1; j<=n ; j=j*2)
          {
            for (k=1 ; k=n; k=k*2)
                { count++; }
          }
     }
}
```

$O(n \log n)$

Examples of Algorithm Complexity III

```
void function(int n)
{
 int x=0;
 for (int i=1 ; i<n ; i++)
    {
     for (int j=1 ; j<=i ; j++)
        {
         for (int k=1 ; k<=j ; k++)
            {
             x=x+1;
            }
        }
    }
}
```

$O(n^3)$

Examples of Algorithm Complexity IV

```
int function{int n}
{
 int a=0;
 int b=0;

 if (n==0)
   {
    return 1;
   }
 else
   {
    a=function(n-1);
    b=function(n-1);
    return a+b;
   }
}
```

$O(2^n)$

Time Measurement Method

```
#include <chrono>
using namespace std::chrono;

int64_t millisSinceEpoch(void)
{
 steady_clock::duration dur{steady_clock::now().time_since_epoch()};
 return duration_cast<milliseconds>(dur).count();
}

int64_t microsSinceEpoch(void)
{
 steady_clock::duration dur{steady_clock::now().time_since_epoch()};
 return duration_cast<microseconds>(dur).count();
}
```

Time Measurement Method II

```
#include <chrono>
using namespace std;

auto start = chrono::steady_clock::now();
/* call your functions here */
auto end = chrono::steady_clock::now();

cout << "Microsecond: " <<
chrono::duration_cast<chrono::microseconds>(end-start).count()
<< endl;

cout << "Second: " <<
chrono::duration_cast<chrono::seconds>(end-start).count()
<< endl;
```

Count vs. Time

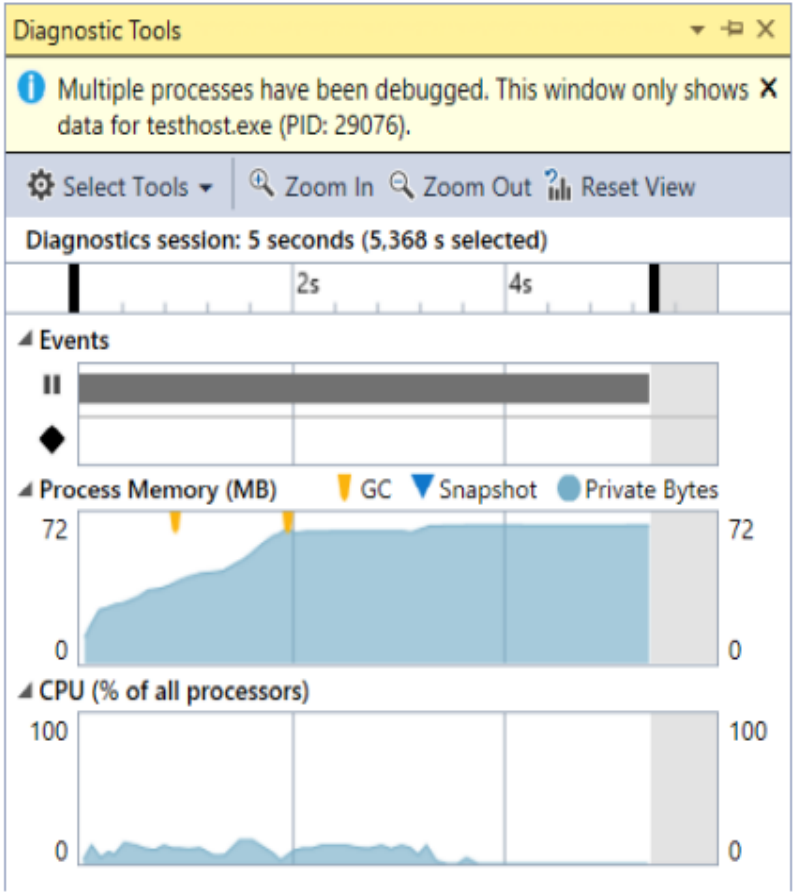
Step Count Method

The operation-count method of estimating time complexity omits accounting for the time spent on all but the chosen operations relates to the problem size. A step is any computation unit that is independent of the problem size. The success of this method depends on the ability to identify the operations that contribute most to the time complexity.

Measure Execution Time

The running time of an algorithm for a specific input depends on the number of operations executed. We measure the running time of a program or an algorithm in term of CPU (time) usage. To measure the execution time, we make a measurement in two different time instants on the clock cycles and find the difference. This depends on the machine, compiler as well as the code. We have somewhat mitigated the machine-dependency of CPU time.

Memory Usage



When you start debugging in Visual Studio by selecting *Debug* > *Start Debugging*, or pressing *F5*, the Diagnostic Tools window appears by default.

To open it manually, select *Debug* > *Windows* > *Show Diagnostic Tools*. The Diagnostic Tools window shows information about events, process memory, and CPU usage.

Memory Usage II

sizeof()

The `sizeof()` predefined built-in function tell us the size in bytes of whatever datatype is specified and occupied the memory. This function (also named operator) is machine dependent, and returns different values according to the specific hardware. For instance, what can take 2 bytes in most PC will take 4 bytes for mainframes and 16 bytes for supercomputers. This function is working for primitives and user-defined datatypes.

```
int a;
int b=sizeof(a);
```

Recap

- Algorithms have different types of voracity for consuming time and memory.
- Assess algorithms with time and space complexity analysis.
- Calculate time complexity by finding the exact $T(n)$ function, the number of operations performed by an algorithm.
- Express time complexity using the Big-O notation.
- Perform simple time complexity analysis of algorithms using this notation.
- Calculating $T(n)$ is not necessary for inferring the Big-O complexity of an algorithm.
- The cost of running exponential algorithms explode and not runnable for big inputs.
- If an algorithm is too slow, would optimizing the algorithm or using a supercomputer help?