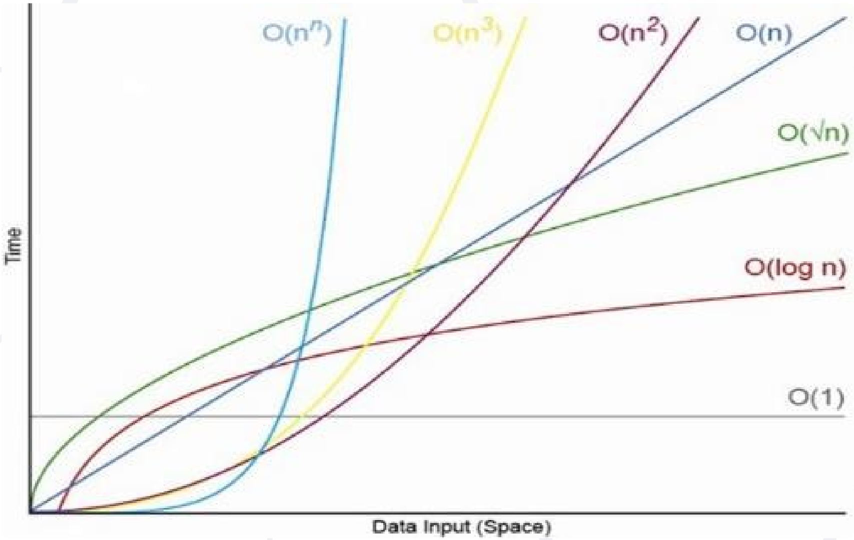
Algorithms and Complexity

Analyzing Algorithm Complexity. Asymptotic Notation



SoftUni Team

Technical Trainers

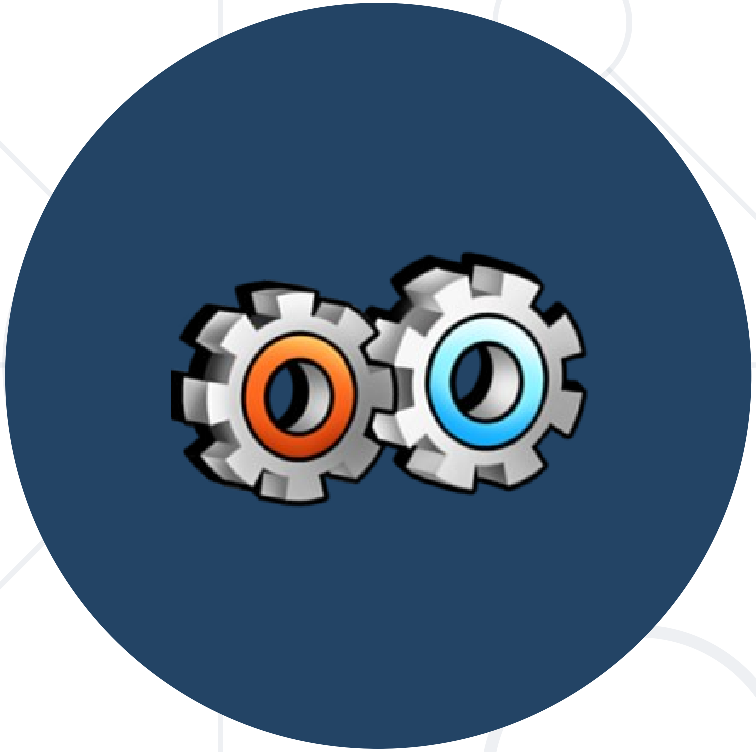
Software University

Software University

https://about.softuni.bg/

Table of Contents

1. **Algorithms**
   - Sorting and Searching, Combinatorics, Dynamic Programming, Graphs, Others
2. **Complexity of Algorithms**
   - Time and Space Complexity
   - Mean, Average and Worst Case
   - Asymptotic Notation **O(g)**



Overview

What is an Algorithm?

- The term "**algorithm**" means "**a sequence of steps**"
  - Derived from **Muḥammad Al-Khwārizmī'**, a Persia mathematician and astronomer
    - He described an algorithm for solving quadratic equations in 825

> **"In mathematics and computer science, an algorithm is a step-by-step procedure for calculations. An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function."**
>
> ***-- Wikipedia***

Algorithms in Computer Science

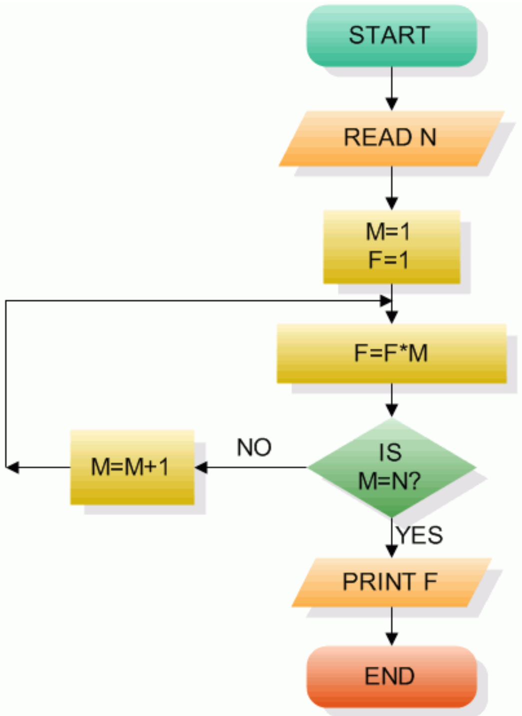
- Algorithms are fundamental in programming
  - **Imperative** (traditional, algorithmic) programming means to **describe in formal steps** how to do something
  - **Algorithm** == sequence of operations (steps)
    - Can include branches (conditional blocks) and repeated logic (loops)
- **Algorithmic thinking** (mathematical thinking, logical thinking, engineering thinking)
  - Ability to decompose the problems into formal sequences of steps (algorithms)

Pseudocode and Flowcharts

- **Algorithms** can be expressed as **pseudocode**, through **flowcharts** or **program code**

```
BFS(node)
{
  queue ← node
  while queue not empty
    v ← queue
    print v
    for each child c of v
      queue ← c
}
```

Pseudo-code



Flowchart

```
public void DFS(Node node)
{
  Print(node.Name);
  for (int i = 0; i < node.
    Children.Count; i++)
  {
    if (!visited[node.Id])
      DFS(node.Children[i]);
  }
  visited[node.Id] = true;
}
```

Source code

Some Algorithms in Programming

- Sorting and searching
- Combinatorial algorithms
  - Recursive algorithms
- Dynamic programming
- Graph algorithms
  - DFS and BFS traversals
- Other algorithms
  - Greedy algorithms, computational geometry, randomized algorithms, parallel algorithms, genetic algorithms

Asymptotic Notation

Algorithm Analysis

- **Why should we analyze algorithms?**
  - Predict the **resources** the algorithm will need
    - Computational **time** (CPU consumption)
    - **Memory** space (RAM consumption)
    - Communication **bandwidth** consumption
  - The expected **running time** of an algorithm is:
    - The total number of **primitive operations** executed (machine independent steps)
    - Also known as **algorithm complexity**

Algorithmic Complexity

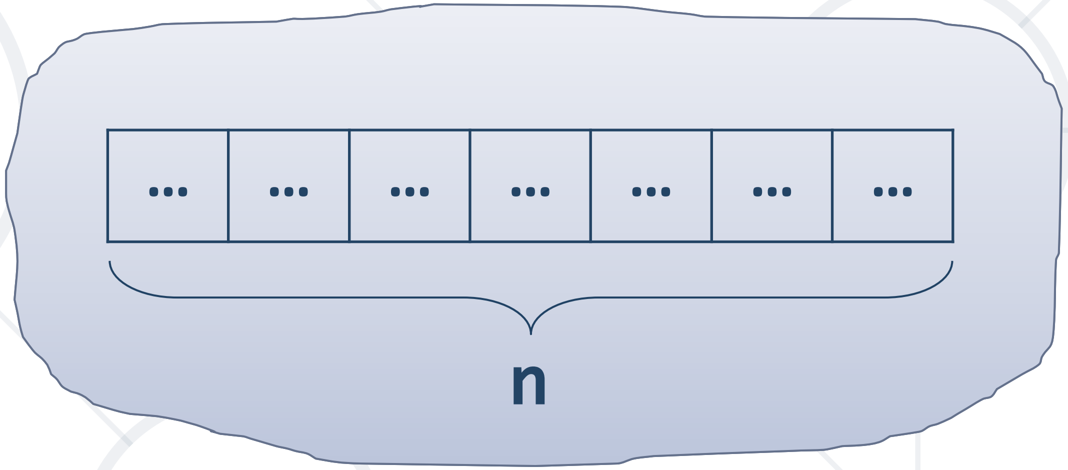
- **What to measure?**
  - CPU time
  - Memory consumption
  - Number of steps
  - Number of particular operations
    - Number of disk operations
    - Number of network packets
  - Asymptotic complexity

Time Complexity

- **Worst-case**
  - An upper bound on the running time for any input of given size
  - Typically, algorithms performance is measured for their worst case
- **Average-case**
  - The running time averaged over all possible inputs
  - Used to measure algorithms that are repeated many times
- **Best-case**
  - The lower bound on the running time (the optimal case)

Time Complexity: Example

- Sequential search in a list of size **n**
  - Worst-case:
    - **n** comparisons
  - Best-case:
    - **1** comparison
  - Average-case:
    - **n/2** comparisons
- The algorithm runs in **linear time**
  - Linear number of operations



Algorithms Complexity

- **Algorithm complexity** - rough estimation of the **number of steps** of a given computation, depending on the **size of the input**
  - Measured with asymptotic notation
    - **`O(g)`** where **g** is a function of the size of the input data
- Examples:
  - Linear complexity **$O(n)$**
    - All elements are processed once (or constant number of times)
  - Quadratic complexity **$O(n^2)$**
    - Each of the elements is processed **n** times

Asymptotic Notation: Definition

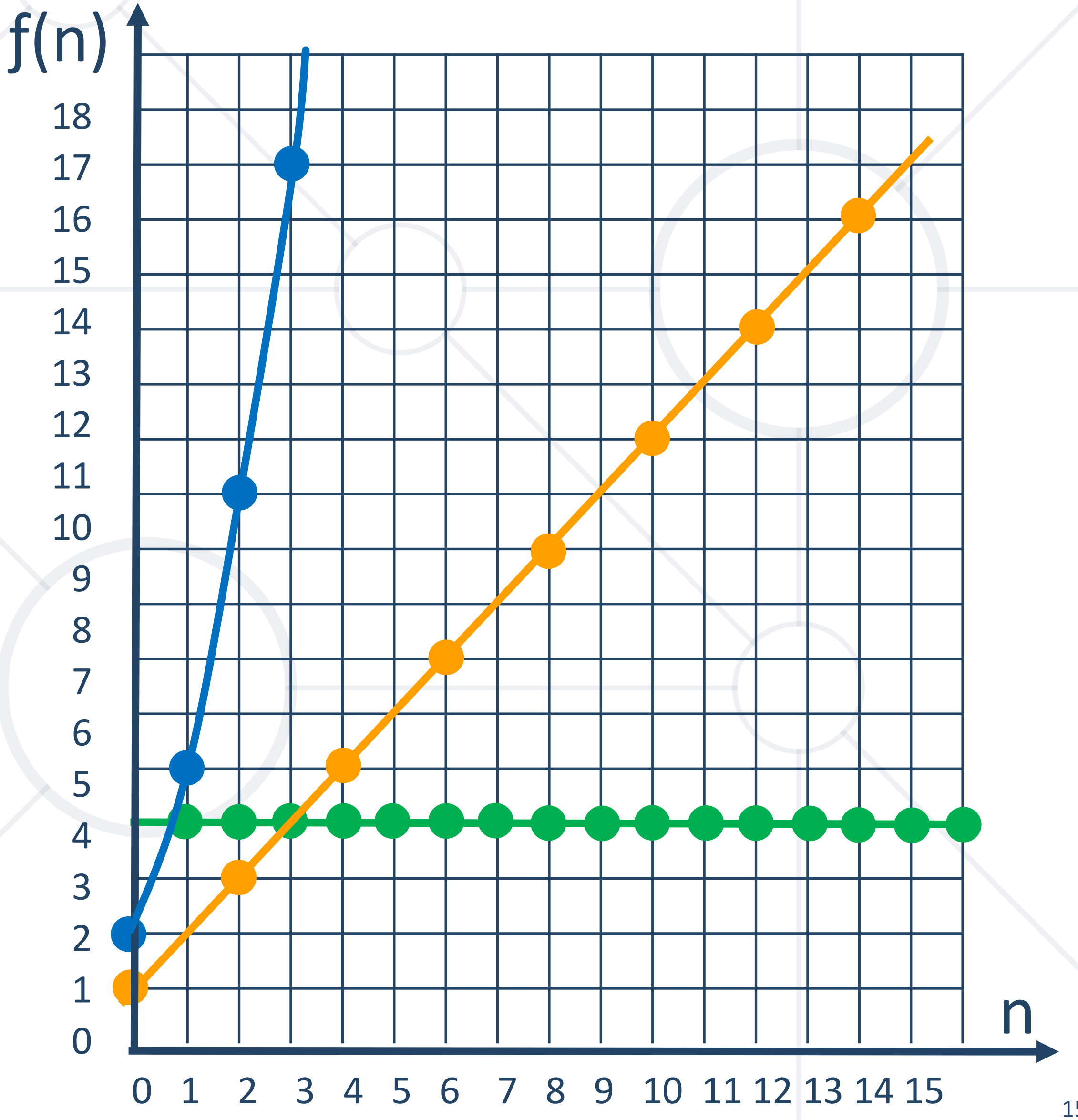
- Asymptotic upper bound
  - O-notation **(Big O notation)**
- For a given function $g(n)$, we denote by $O(g(n))$ the set of functions that are different than $g(n)$ by a constant

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $f(n) <= c*g(n)$ for all $n >= n_0\}$

- Examples:
  - $3 * n^2 + n/2 + 12 \in O(n^2)$
  - $4*n*\log_2(3*n+1) + 2*n-1 \in O(n * \log n)$

Functions Growth Rate

- **O(n)** means a function grows linearly when n increases
  - E.g. $f(n)=n+1$
- **O(n^2)** means a function grows exponentially when n increases
  - E.g. $f(n)=n^2+2n+2$
- **O(1)** means a function does not grow when n changes
  - E.g. $f(n)=4$



Asymptotic Notation: Examples

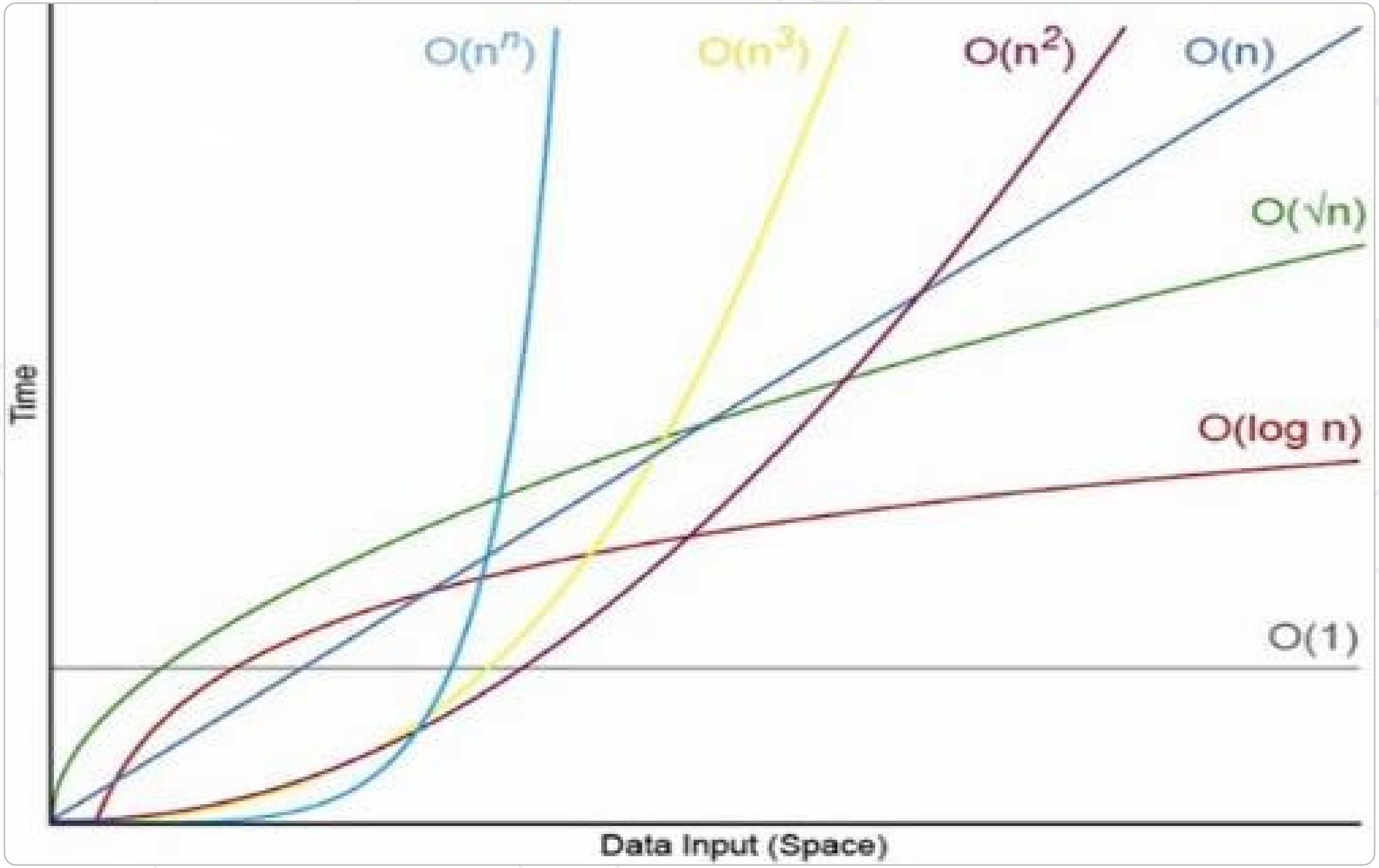
Positive examples:

$$10n \in O(n),$$
$$10n \in O(n^2),$$
$$10n \in O(n^4)$$
$$10n \in O(3n^4 - 10n^2 + 7)$$
$$10n+3 \in O(n)$$
$$4n^2 - 5n + 2 \in O(n^2)$$
$$4n^3 + 5n^2 + 5 \in O(n^3)$$
$$\sqrt{n} \in O(n)$$
$$\log n \in O(\sqrt{n})$$

Negative examples:

$$2n \notin O(1)$$
$$4n^2 - 5n + 2 \notin O(n)$$
$$5n + 1 \notin O(\sqrt{n})$$
$$\sqrt{n^3} \notin O(n)$$

Asymptotic Functions



Typical Complexities

Complexity	Notation	Description
constant	O(1)	Constant number of operations, not depending on the input data size, e.g. n = 1 000 000 → 1-2 operations
logarithmic	O(log n)	Number of operations proportional to $\log_2(n)$ where n is the size of the input data, e.g. n = 1 000 000 000 → 30 operations
linear	O(n)	Number of operations proportional to the input data size, e. g. n = 10 000 → 5 000 operations

Typical Complexities (2)

Complexity	Notation	Description
quadratic	$O(n^2)$	Number of operations proportional to the square of the size of the input data, e.g. n = 500 → 250 000 operations
cubic	$O(n^3)$	Number of operations propor-tional to the cube of the size of the input data, e.g. n = 200 → 8 000 000 operations
exponential	$O(2^n)$, $O(k^n)$, $O(n!)$	Exponential number of operations, fast growing, e.g. n = 20 → 1 048 576 operations

Function Values

Software University

Function	*Value*				
	$n = 1$	$n = 2$	$n = 10$	$n = 100$	$n = 1000$
5	5	5	5	5	5
$\log n$	0	1	3,32	6,64	9,96
n	1	2	10	100	1000
$n \log n$	0	2	33,2	664	9966
n^2	1	4	100	10000	10^6
n^3	1	8	1000	10^6	10^9
2^n	2	4	1024	10^{30}	10^{300}
$n!$	1	2	3628800	10^{157}	10^{2567}
n^n	1	4	10^{10}	10^{200}	10^{3000}

Time Complexity and Program Speed

Complexity	10	20	50	100	1 000	10 000	100 000
O(1)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(log(n))	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n*log(n))	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
$O(n^2)$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	2 s	3-4 min
$O(n^3)$	< 1 s	< 1 s	< 1 s	< 1 s	20 s	5 hours	231 days
$O(2^n)$	< 1 s	< 1 s	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 s	hangs	hangs	hangs	hangs	hangs	hangs
$O(n^n)$	3-4 min	hangs	hangs	hangs	hangs	hangs	hangs

$O(n)$

$O(n+m)$

$O(n^2)$

$O(n!)$

$O(n*m)$

$f(x)$

Examples

Complexity Examples

```
int FindMaxElement(int[] array)
{
  int max = array[0];
  for (int i = 1; i < array.length; i++)
  {
    if (array[i] > max)
      max = array[i];
  }
  return max;
}
```

- Runs in **O(n)** where **n** is the size of the array
- The number of elementary steps is **~ n**

Complexity Examples (2)

```
long FindInversions(int[] array)
{
  long inversions = 0;
  for (int i = 0; i < array.Length; i++)
    for (int j = i + 1; j < array.Length; i++)
      if (array[i] > array[j])
        inversions++;
  return inversions;
}
```

- Runs in **$O(n^2)$** where **n** is the size of the array
- The number of elementary steps is **~ n * (n+1) / 2**

Complexity Examples (3)

```
decimal Sum3(int n)
{
    decimal sum = 0;
    for (int a = 0; a < n; a++)
        for (int b = 0; b < n; b++)
            for (int c = 0; c < n; c++)
                sum += a * b * c;
    return sum;
}
```

- Runs in cubic time $O(n^3)$
- The number of elementary steps is $\sim n^3$

Complexity Examples (4)

```
decimal SpecialCalculation(int n)
{
    decimal sum = 0;
    for (int a = 0; a < n; a++)
        for (int b = 0; b < n; b++)
            if (a == b)
              for (int c = 0; c < n; c++)
                sum += a * b * c;
    return sum;
}
```

- Runs in quadratic time **$O(n^2)$** – think why!
- The number of elementary steps is **$\sim n^2$**

Summary

- **Algorithms** are sequences of steps for calculating / doing something
- **Algorithm complexity** is a rough estimation of the **number of steps** performed by given computation
  - Can be **logarithmic linear**, **n log n**, **square** (n^2), **cubic** (n^3), **exponential**, etc.
  - Complexity predicts the speed of given code before its execution

Software University

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Resources

1. "Fundamentals of Computer Programming with C#“ → "Data Structures and Algorithm Complexity" → pages 787-798
   - https://introprogramming.info/wp-content/uploads/2018/07/CSharp-Principles-Book-Nakov-v2018.pdf

License