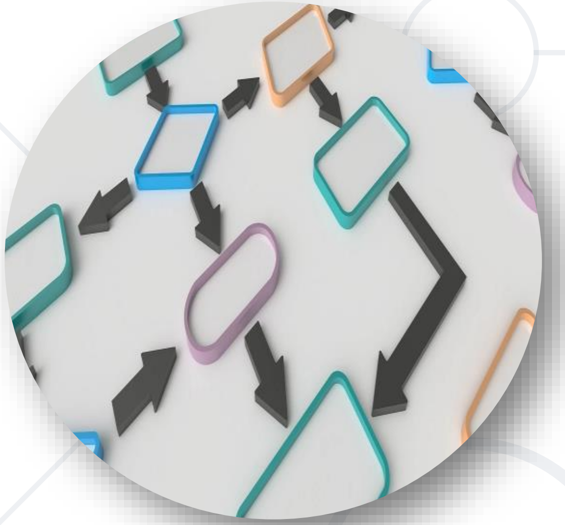
Basic Algorithms

Workshop



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://about.softuni.bg/

Have a Question?

sli.do

#csharp-advanced

Table of Contents

O(n)

Algorithmic Complexity

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
    - **Hard disk** operations
    - Other resources

Problem: Get Number of Steps

- Calculate maximum steps to find the result

```
long GetOperationsCount(int n)
{
  long counter = 0;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      counter++;
  return counter;
}
```

Solution:
$T(n) = 3(n ^ 2) + 3n + 3$

- The input(n) of the function is the main source of steps growth

Simplifying Step Count

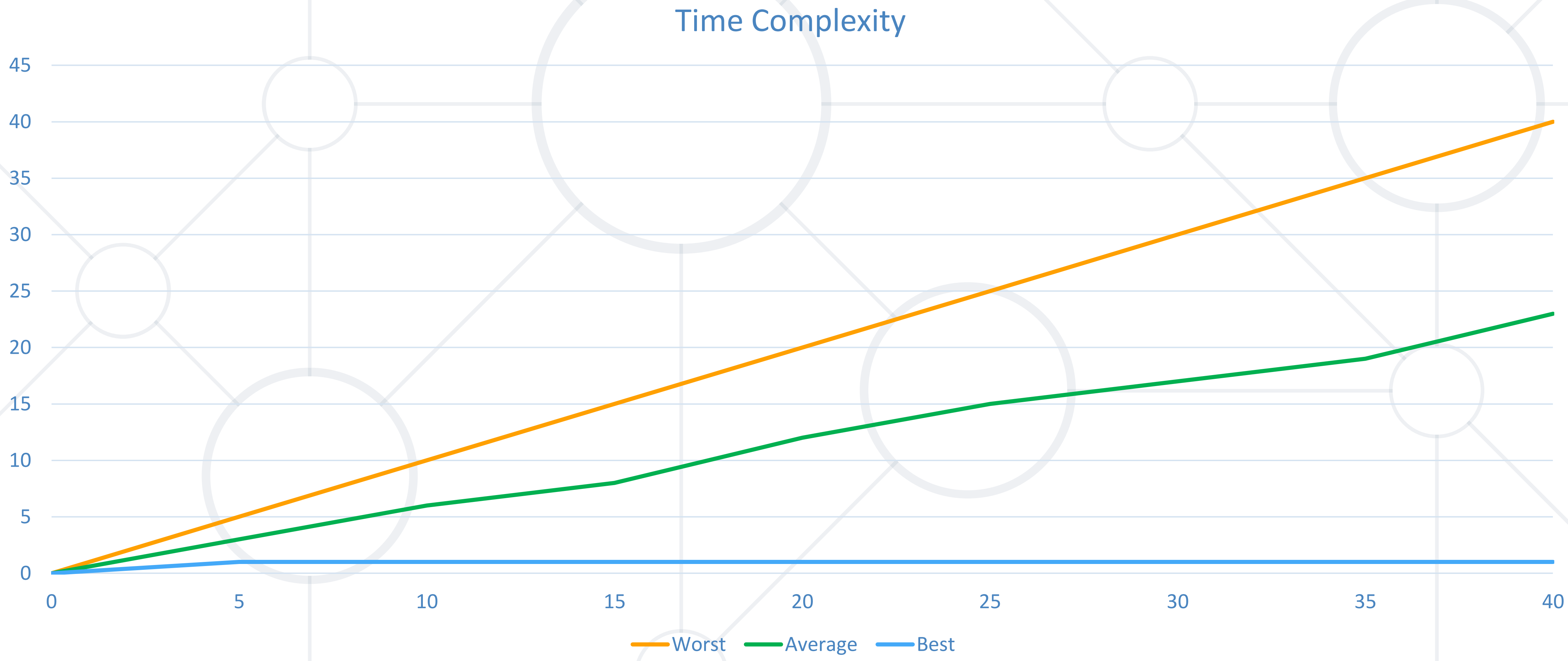
- Some parts of the equation **grow much faster** than others
  - $T(n) = \mathbf{3(n^2)} + 3n + 3$
  - We can **ignore** some part of this equation
  - Higher terms **dominate** lower terms – $\mathbf{n > 2}$, $\mathbf{n^2 > n}$, $\mathbf{n^3 > n^2}$
  - Multiplicative constants can be **omitted** – $\mathbf{12n \rightarrow n}$, $\mathbf{2n^2 \rightarrow n^2}$
- The previous solution becomes $\mathbf{\approx n^2}$

Time Complexity

- **Worst-case**
  - An **upper** bound on the running time
- **Average-case**
  - **Average** running time
- **Best-case**
  - The **lower** bound on the running time (the optimal case)

Time Complexity

- Therefore, we need to measure **all** the possibilities:



Time Complexity

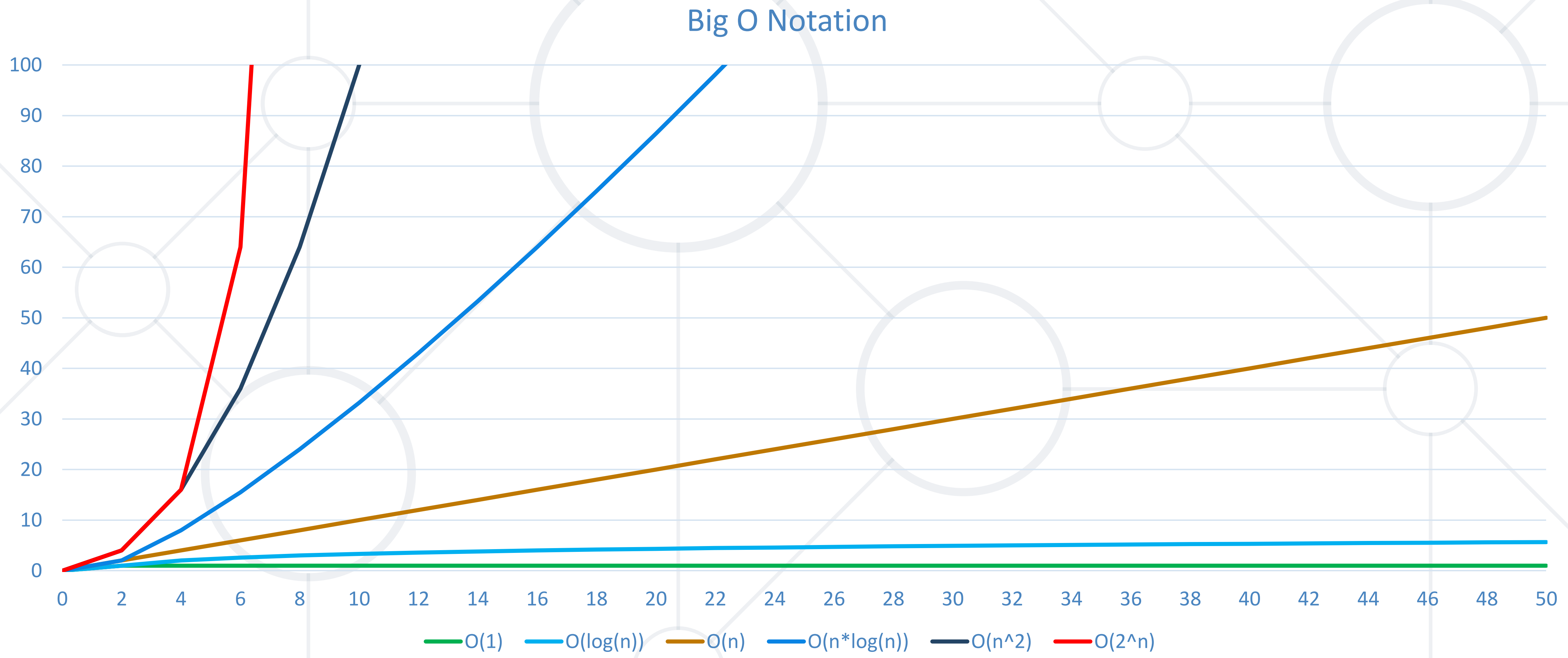
- From the previous chart we can deduce:
  - For smaller size of the input (**n**) we **don't care much for the runtime**
  - So we measure the time as **n** approaches **infinity**
  - If an algorithm **must scale**, it **should compute** the result within a **finite and practical time**
  - We're concerned about the **order of an algorithm's complexity**, not the actual time in terms of **milliseconds**

Asymptotic Notations

- Descriptions that allow us to examine an algorithm's running time
- There are **three** common asymptotic notations:
  - Big **O** – **O(f(n))**
  - Big **Theta** – **Θ(f(n))**
  - Big **Omega** – **Ω(f(n))**

Asymptotic Functions

- Below are some examples of **common algorithmic** grow:



Typical Complexities

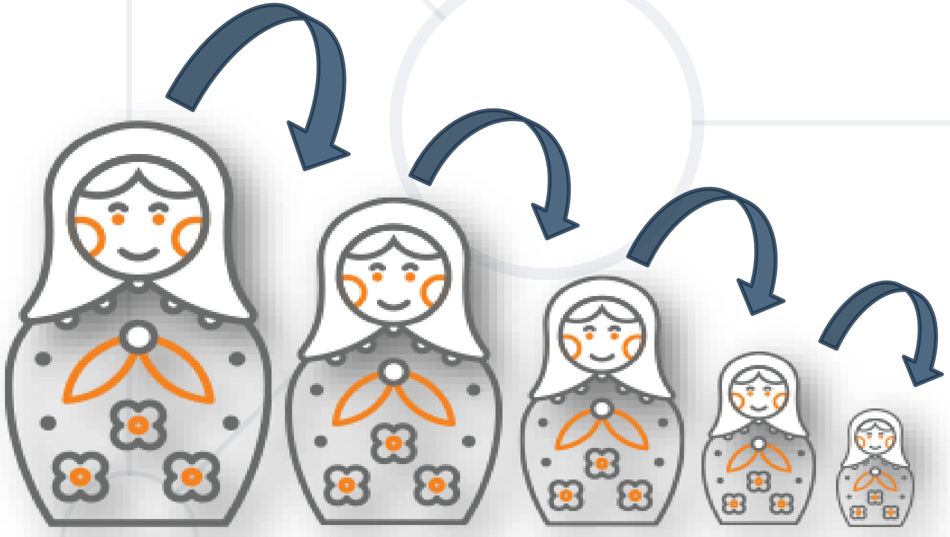
Complexity	Notation	Description
constant	O(1)	n = 1 000 → 1-2 operations
logarithmic	O(log n)	n = 1 000 → 10 operations
linear	O(n)	n = 1 000 → 1 000 operations
linearithmic	O(n*log n)	n = 1 000 → 10 000 operations
quadratic	O(n2)	n = 1 000 → 1 000 000 operations
cubic	O(n3)	n = 1 000 → 1 000 000 000 operations
exponential	O(n^n)	n = 10 → 10 000 000 000 operations

What is Recursion?

Recursion

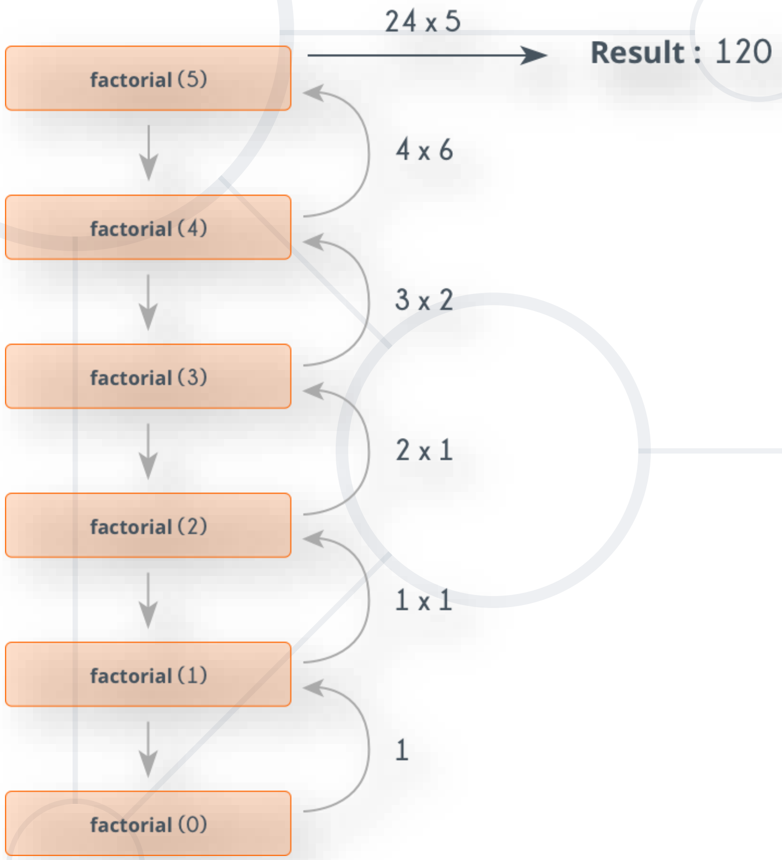
What is Recursion?

- A function or a method that **calls itself one or more** times until a specified **condition** is **met**
  - When the condition is met, the rest of **each** repetition is processed **from** the **last** one called **to** the **first**

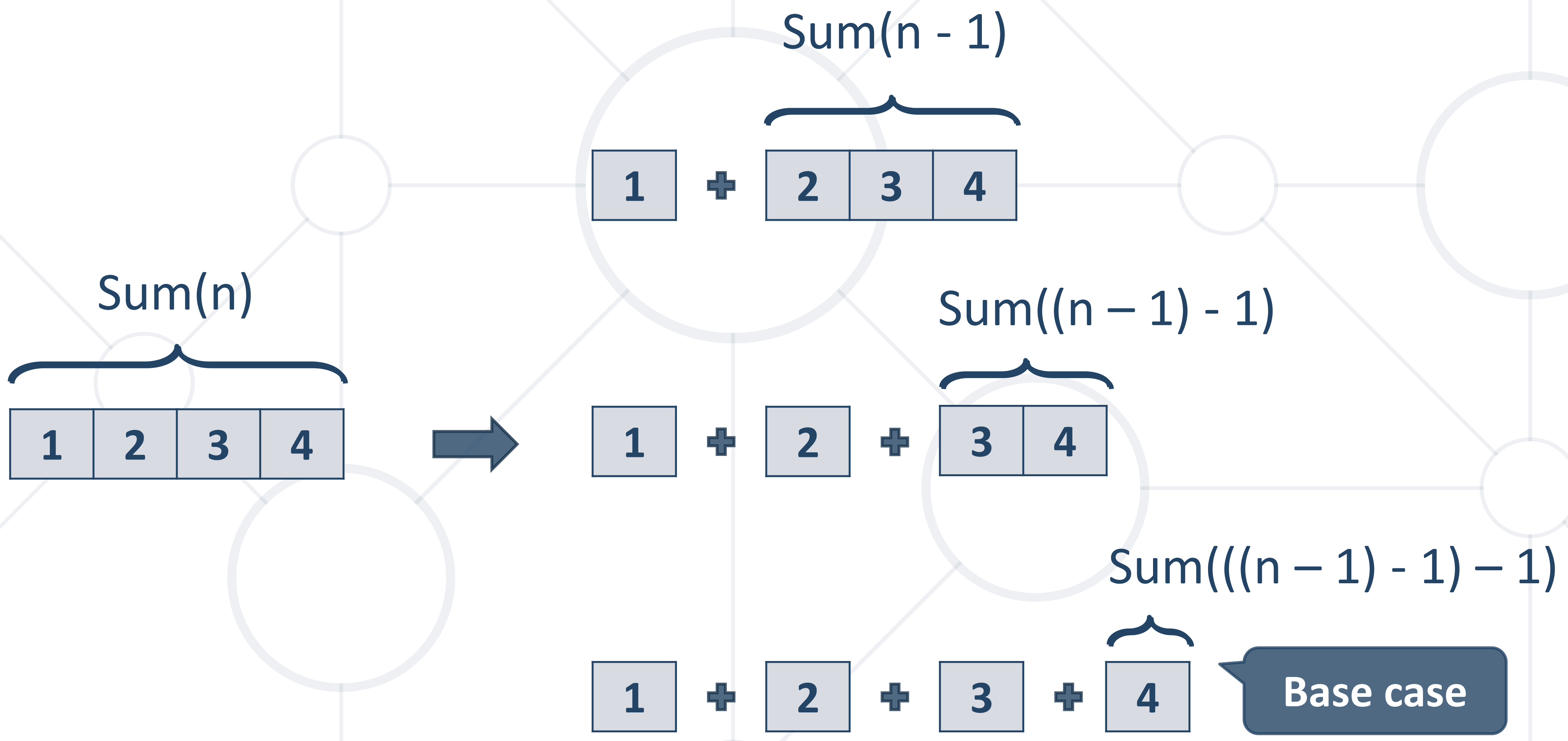


How Does It Work?

- The function or method has a **base case**
- **Each step** of the recursion should **move towards** the **base case**



Example: Array Sum



Problem: Recursive Array Sum

- Write a **recursive method** that:
  - Finds the sum of all numbers stored in an `int[] array`
  - Read numbers from the console

```
1 2 3 4
```
→
```
10
```

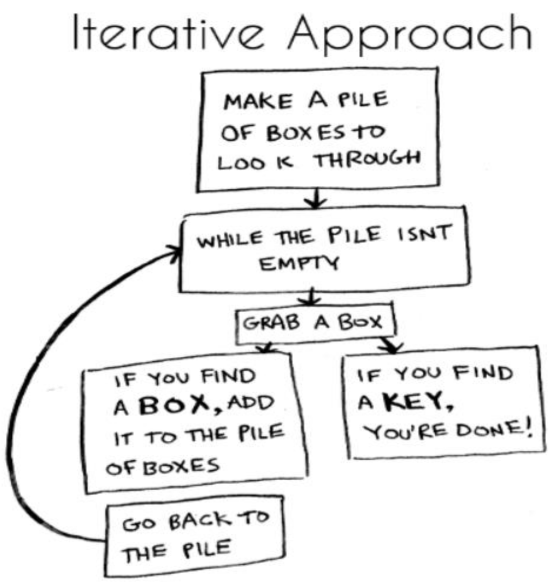
```
-1 0 1
```
→
```
0
```

Solution: Recursive Array Sum

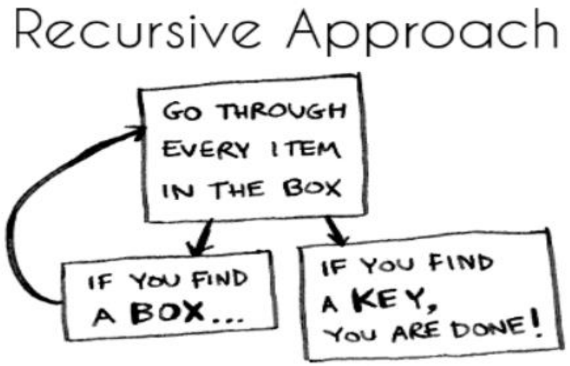
```
static int Sum(int[] array, int index)
{
  if (index == array.Length - 1)
  {
    return array[index];
  }

  return array[index] + Sum(array, index + 1);
}
```

Iterative vs. Recursive Approach

- A function **repeats** a defined process until a condition fails



- A function that **calls itself** repeatedly until a certain condition is met



Example: Recursive Factorial

- Recursive definition of n! (n factorial):

5	120
10	3628800

- Pseudocode

```
n! = n * (n-1)! for n > 0
0! = 1
```

N!

Problem: Recursive Factorial

- Create a **recursive method** that calculates **n!**
  - Read n from the console

Input	Output
5	120
10	3628800

Solution: Recursive Factorial

```
static long Factorial(int num)
{
  if (num == 0)
  {
    return 1;
  }

  return num * Factorial(num - 1)
}
```

Base case

Recursion Pre-Actions and Post-Actions

- Recursive methods have 3 parts:
  - **Pre-actions** (before calling the recursion)
  - **Recursive calls** (step-in)
  - **Post-actions** (after returning from recursion)

```
static void Recursion
{
  // Pre-actions
  Recursion();
  // Post-actions
}
```

Direct and Indirect Recursion

- **Direct recursion**
  - A method directly calls itself
- **Indirect recursion**
  - Method A calls B, method B calls A
  - Or even A → B → C → A

```
void A()
{
  …
  A();
}
```

```
void A()
{
  …
  B();
}
```

```
void B()
{
  …
  A();
}
```

Brute-Force Algorithms

Brute-Force Algorithms

- Trying all possible combinations
- Picking the best solution
- Usually slow and inefficient

0 0 0 0 0

Brute-Force Algorithms

0	0	0	0	0

Brute-Force Algorithms

0	0	0	0	1

Brute-Force Algorithms

0	0	0	0	2

Brute-Force Algorithms

9 9 9 9 9

10 x 10 x 10 x 10 x 10 = 100,000 combinations

Greedy Algorithms

Greedy Algorithms

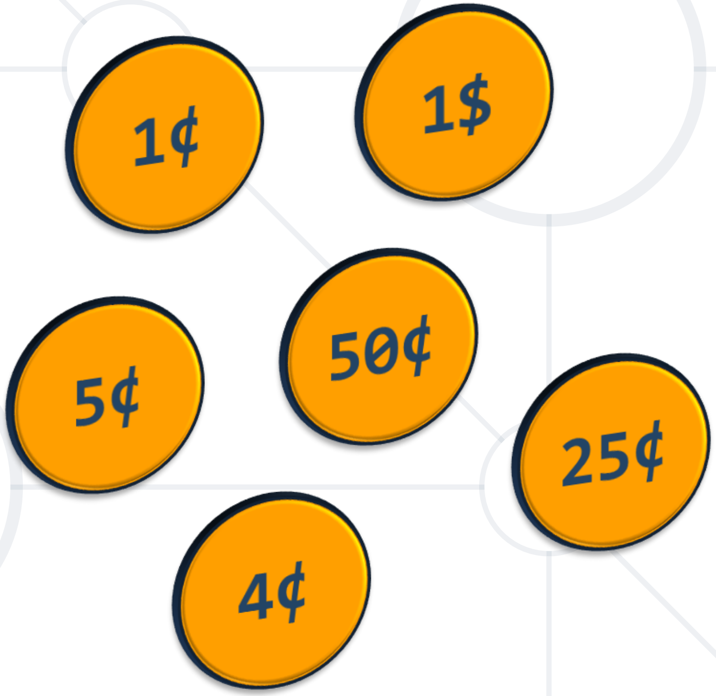
- Greedy algorithms assume that **always choosing a local optimum** leads to the global optimum
- Can produce a **non-optimal (incorrect)** result
- It is used in **optimization problems** as well
  - Find the **shortest** path from Sofia to Varna
  - Find the **maximum increasing subsequence**

Live Demo

Greedy Algorithms

Problem: Sum of Coins

- Write a program, which gathers a sum of money, using the least possible number of coins
- Consider the US **currency coins**
  - **0.01**, **0.02**, **0.05**, **0.10**
- **Greedy algorithm** for "Sum of Coins":
  - Take the largest coin while possible
  - Then take the second largest
  - Etc.



Sum of Coins Visualization

Target: 18

Actual: 0

Sum of Coins Visualization

Target: 18

Actual: 10

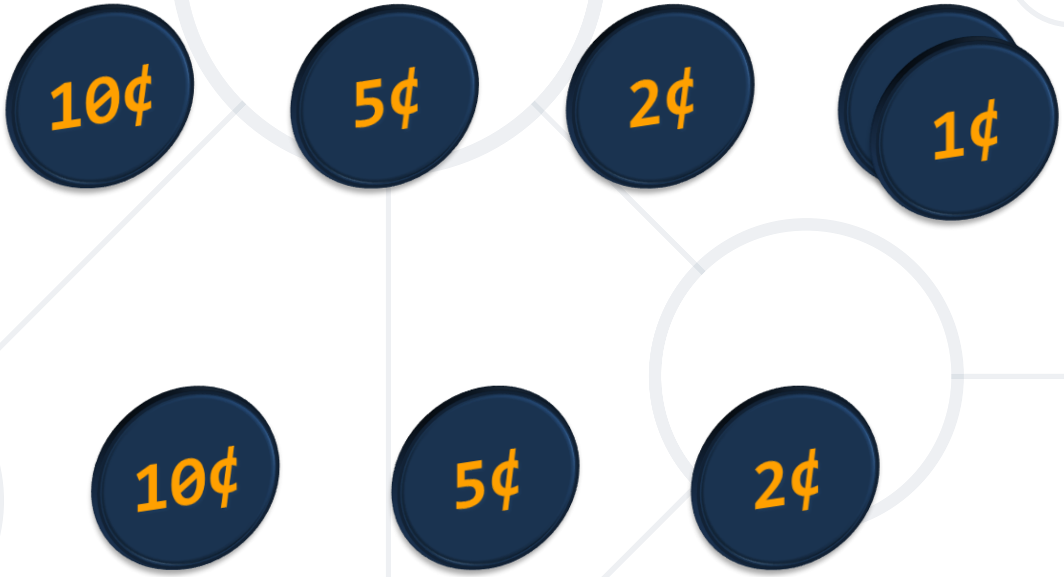
Sum of Coins Visualization

Target: 18

Actual: 15

Sum of Coins Visualization

Target: 18



Actual: 17

Sum of Coins Visualization

Target: 18

Actual: 18

Greedy Failure Cases

Sum of Coins Failure

Target: 18

Actual: 0

Sum of Coins Failure

Target: 18

Actual: 10

Sum of Coins Failure

Target: 18

Actual: 15

Sum of Coins Failure

Target: 18

Actual: 16

Sum of Coins Failure

Target: 18

Actual: 17

Sum of Coins Failure

Target: 18

Actual: 18

Sum of Coins Failure

Target: 18

Solution: Sum of Coins

Software University

```
int finalSum = 18;
int currentSum = 0;
int[] coins = { 10, 10, 5, 5, 2, 2, 1, 1 };
Queue<int> resultCoins = new Queue<int>();

// Next slide

Console.WriteLine("Sum not found");
```

Solution: Sum of Coins

Software University

```
for (int i = 0; i < coins.Length; i++)
{
  if (currentSum + coins[i] > finalSum) continue;

  currentSum += coins[i];
  resultCoins.Enqueue(coins[i]);

  if (currentSum == finalSum)
    // Sum found
}
```

Problem: Set Cover

- Write a program that finds the smallest subset of S, the union of which = **U** (if it exists)
- You will be given a **set** of integers **U** called "**the Universe**"
- And a set **S** of **n** integer sets whose union = **U**

```
Universe: 1, 2, 3, 4, 5
Number of sets: 4
1, 4
2, 4
5, 2
3
```

```
Sets to take (3):
{ 1, 4 }
{ 5, 2 }
{ 3 }
```

Solution: Set Cover

```
public static List<int[]> ChooseSets(List<int[]> sets,
 List<int> universe)
{
  List<int[]> selectedSets = new List<int[]>();
  while (universe.Count > 0)
  {
    // Next slide
  }
  return selectedSets;
}
```

Solution: Set Cover

```
int[] current = sets.OrderByDescending(set =>
set.Count(universe.Contains)).First();

  selectedSets.Add(current);
  sets.Remove(current);

  foreach (int i in current)
  {
    universe.Remove(i);
  }
```

Simple Sorting Algorithms

What is a Sorting Algorithm?

- An algorithm that rearranges elements in a set in a **specific order**
  - The elements must be **comparable**

Unsorted list

10	3	7	3	4

→  →

Sorted list

3	3	4	7	10

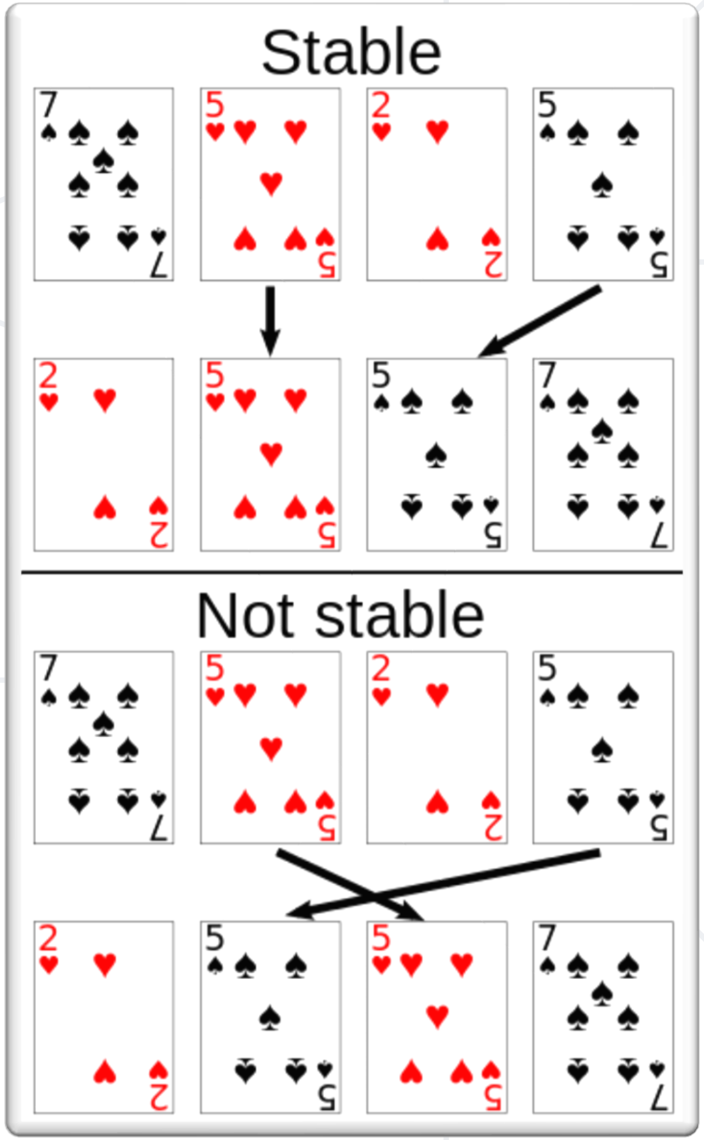
  - Sorting algorithms
    - Insertion, Exchange (Bubble sort and Quicksort), Selection (Heapsort), Merging, Serial/Parallel, etc.

Sorting Algorithms: Classification

- Sorting algorithms are often classified by
  - Computational **complexity** and memory usage
    - **Worst**, **average** and **best-case** behavior
  - **Recursive** / **non-recursive**
  - Stability – **stable** / **unstable**
  - **Comparison-based** sort / **non-comparison** based
- Fun fact - **Bogosort** is highly inefficient sorting algorithm with two versions
  - Deterministic version that enumerates all permutations until it hits a sorted one
  - Randomized version that randomly permutes its input

Stability of Sorting

- **Stable** sorting algorithms
  - Maintain the order of equal elements
  - If two items compare as equal, their relative order is preserved
- **Unstable** sorting algorithms
  - Rearrange the equal elements in unpredictable order
- Often **different elements** have the **same key** used for equality comparing

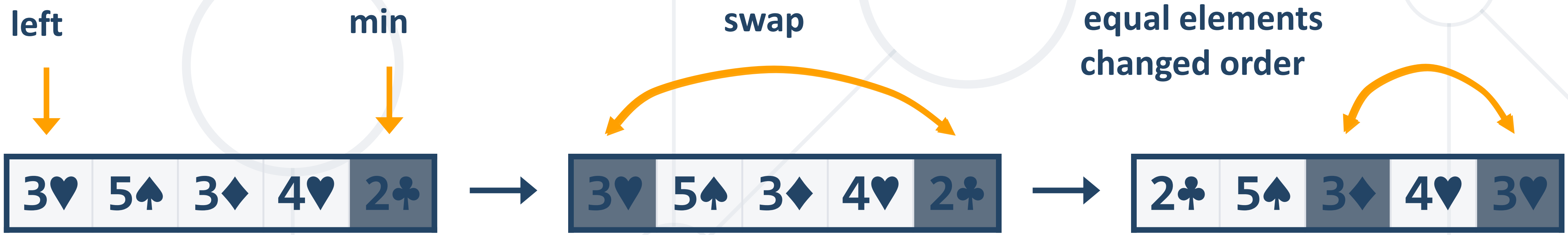


Selection Sort

- Selection sort – simple, but inefficient algorithm (visualize)
  - Swap the first with the min element on the right, then the second, etc.
  - Memory: O(1)
  - Stable: No
  - Method: Selection

Selection Sort: Why Unstable?

- Why the "selection sort" is **unstable**?
  - Swaps the first element with the min element on the right
  - Swaps the second element with the min element on the right
  - Etc.
- During the swaps equal elements can jump over each other

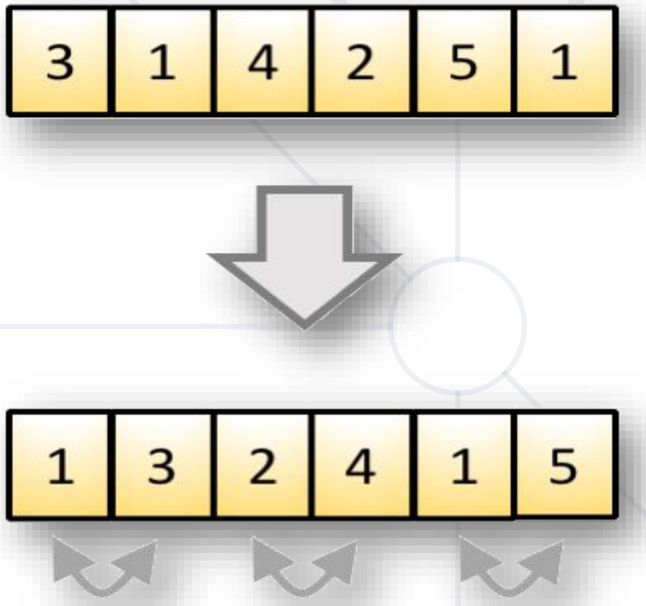


Selection Sort Code

```
for (int index = 0; index < collection.Length; index++)
{
  int min = index;
  for (int curr = index + 1; curr < collection.Length; curr++)
  {
    if (Less(collection[curr], collection[min]))
    {
       min = curr;
    }
  }
  Swap(collection, index, min);
}
```

Bubble Sort

- **Bubble sort** – simple, but inefficient algorithm (**visualize**)
  - Swaps to neighbor elements when not in order until sorted
  - Memory: **O(1)**
  - Stable: Yes
  - Method: Exchanging



Example: Bubble Sort

```
int[] numbers = { 1, 3, 4, 2, 5, 6 };
for (int i = 0; i < numbers.Length; i++)
{
  for (int j = i + 1; j < numbers.Length - 1; j++)
  {
    if (numbers[i] > numbers[j]) {
      int tempNumber = numbers[i];
      numbers[i] = numbers[j];
      numbers[j] = tempNumber; }
  }
}
Console.WriteLine(string.Join(" ", numbers));
```

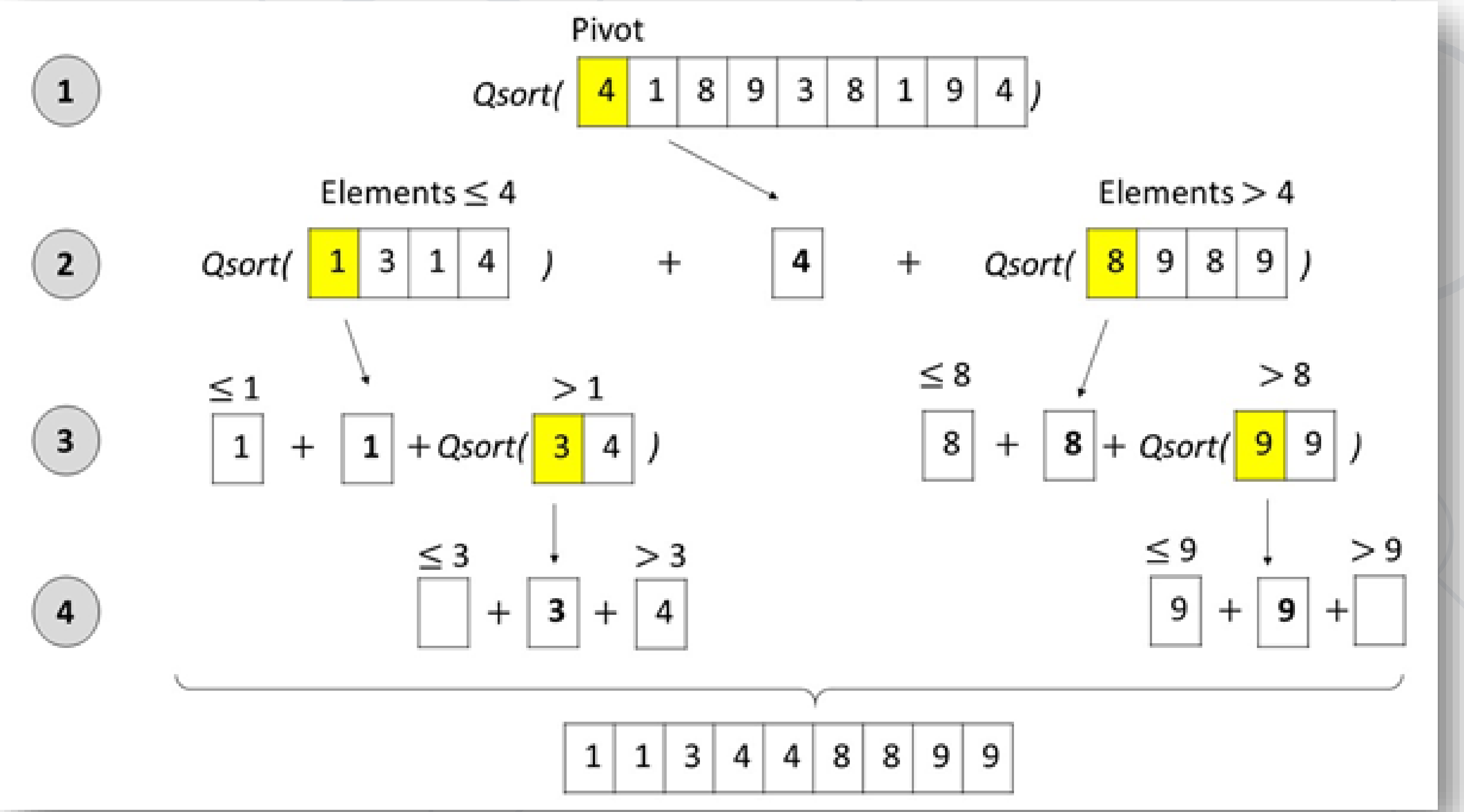
Comparison of Sorting Algorithms

Name	Best	Average	Worst	Memory	Stable	Method
Selection	n^2	n^2	n^2	1	No	Selection
Bubble	n	n^2	n^2	1	Yes	Exchanging

Quick Sort

- **QuickSort** – efficient sorting algorithm
  - Choose a pivot; move smaller elements left & larger right; sort left & right
  - Memory: **O(log(n))** stack space (recursion)
  - Time: **$O(n^2)$**
  - Stable: **Depends**
  - Method: **Partitioning**

Quick Sort: Conceptual Overview



Quick Sort

```
public static void QuickSortHelper(
  int[] array, int startIdx, int endIdx)
{
  if (startIdx >= endIdx)
    return;
  var pivotIdx = startIdx;
  var leftIdx = startIdx + 1;
  var rightIdx = endIdx;
  while (leftIdx <= rightIdx) {
    // TODO: Continues on the next slide
  }
  // TODO: Continues on slide Quick Sort (3)
}
```

Quick Sort

```
if (array[leftIdx] > array[pivotIdx] &&
      array[rightIdx] < array[pivotIdx]) {
  Swap(array, leftIdx, rightIdx);
}

if (array[leftIdx] <= array[pivotIdx]) {
  leftIdx += 1;
}

if (array[rightIdx] >= array[pivotIdx]) {
  rightIdx -= 1;
}
```

Quick Sort

```
Swap(array, pivotIdx, rightIdx);

var isLeftSubArraysSmaller =
  rightIdx - 1 - startIdx < endIdx - (rightIdx + 1);
if (isLeftSubArraysSmaller) {
  QuickSortHelper(array, startIdx, rightIdx - 1);
  QuickSortHelper(array, rightIdx + 1, endIdx);
} else {
  QuickSortHelper(array, rightIdx + 1, endIdx);
  QuickSortHelper(array, startIdx, rightIdx - 1);
}
```

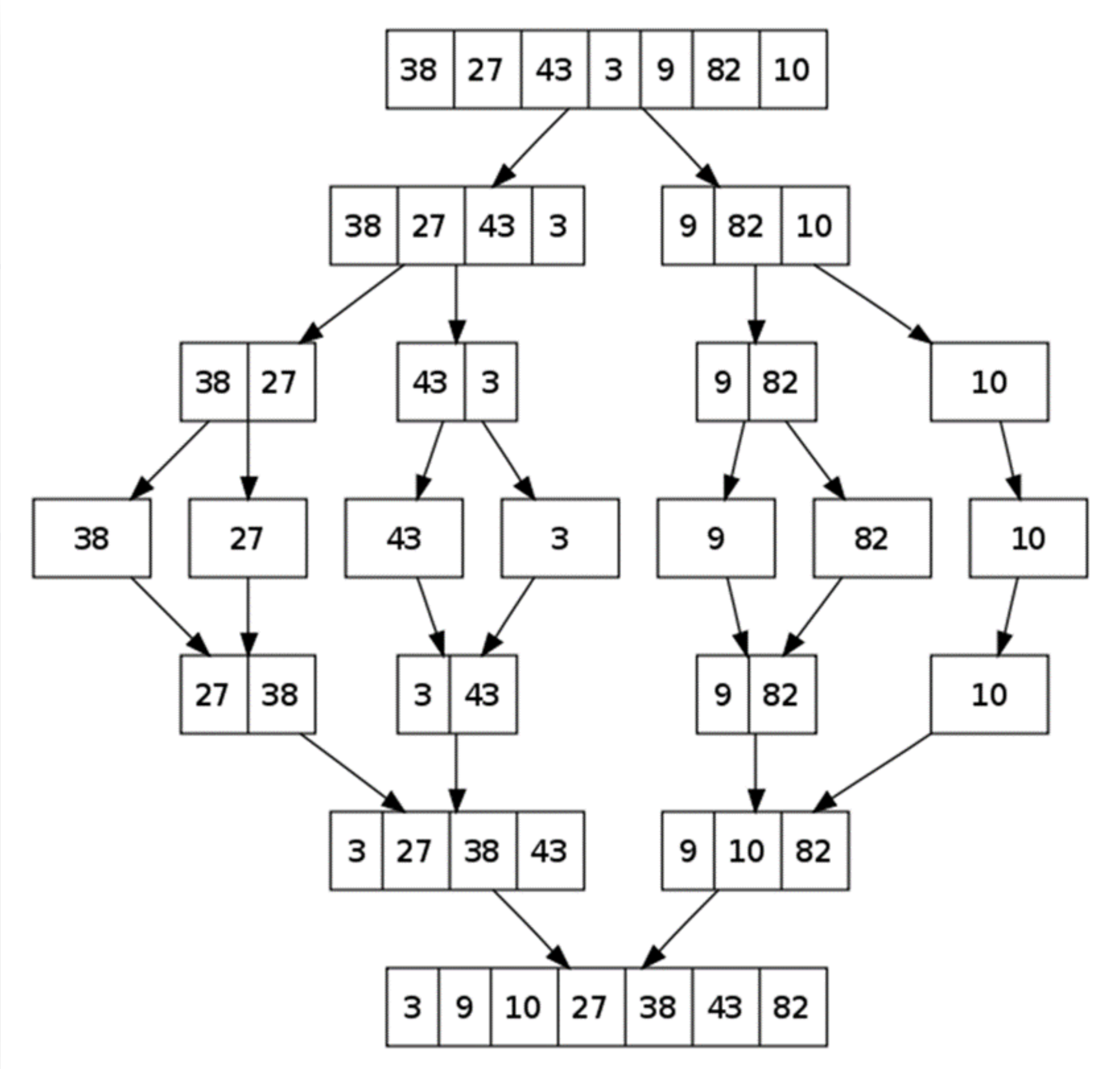
Comparison of Sorting Algorithms

Name	Best	Average	Worst	Memory	Stable	Method
Selection	n^2	n^2	n^2	1	No	Selection
Bubble	n	n^2	n^2	1	Yes	Exchanging
Quick	n * log(n)	n * log(n)	n^2	1	Depends	Partitioning

Merge Sort

- **Merge sort** is efficient sorting algorithm
- Divide the list into sub-lists (typically 2 sub-lists)
  1. Sort each sub-list (recursively call merge-sort)
  2. Merge the sorted sub-lists into a single list
- Memory: **O(n)** / **O(n*log(n))**
- Time: **O(n*log(n))**
- Highly parallelizable on multiple cores / machines → up to **O(log(n))**

Merge Sort: Conceptual Overview



Merge Sort

```
// Memory: O(n*log(n))
public static int[] MergeSort(int[] array)
{
  if (array.Length == 1)
    return array;

  var middleIdx = array.Length / 2;
  var leftHalf = array.Take(middleIdx).ToArray();
  var rightHalf = array.Skip(middleIdx).ToArray();

  return MergeArrays(MergeSort(leftHalf), MergeSort(rightHalf));
}
```

Merge Sort

```
public static int[] MergeArrays(int[] left, int[] right) {
  var sorted = new int[left.Length + right.Length];
  var sortedIdx = 0; var leftIdx = 0; var rightIdx = 0;
  while (leftIdx < left.Length && rightIdx < right.Length) {
    if (left[leftIdx] < right[rightIdx]) {
      sorted[sortedIdx++] = left[leftIdx++];
    } else {
      sorted[sortedIdx++] = right[rightIdx++];
    }
  }
  // TODO: Take remaining elements either from the left or right
  return sorted;
}
```

Merge Sort

```
while (leftIdx < left.Length) {
  sorted[sortedIdx] = left[leftIdx];
  sortedIdx += 1;
  leftIdx += 1;
}

while (rightIdx < right.Length) {
  sorted[sortedIdx] = right[rightIdx];
  sortedIdx += 1;
  rightIdx += 1;
}
```

Merge Sort

```
// Memory: O(n)
public static int[] MergeSort(int[] array)
{
  if (array.Length <= 1)
    return array;

  var copy = new int[array.Length];
  Array.Copy(array, copy, array.Length);

  MergeSortHelper(array, copy, 0, array.Length - 1);

  return array;
}
```

Merge Sort

```
public static void MergeSortHelper(
  int[] source, int[] copy, int leftIdx, int rightIdx)
{
  if (leftIdx >= rightIdx)
    return;

  var middleIdx = (leftIdx + rightIdx) / 2;
  MergeSortHelper(copy, source, leftIdx, middleIdx);
  MergeSortHelper(copy, source, middleIdx + 1, rightIdx);

  MergeArrays(source, copy, leftIdx, middleIdx, rightIdx);
}
```

Merge Sort

```
public static void MergeArrays(
  int[] source, int[] copy, int startIdx, int middleIdx, int endIdx)
{
  var sourceIdx = startIdx;
  var leftIdx = startIdx; var rightIdx = middleIdx + 1;
  while (leftIdx <= middleIdx && rightIdx <= endIdx) {
    if (copy[leftIdx] < copy[rightIdx])
      source[sourceIdx++] = copy[leftIdx++];
    else
      source[sourceIdx++] = copy[rightIdx++];
  }
    // TODO: Take remaining elements either from the left or right
}
```

Merge Sort

```
while (leftIdx <= middleIdx)
{
  source[sourceIdx] = copy[leftIdx];
  leftIdx += 1;
  sourceIdx += 1;
}

while (rightIdx <= endIdx)
{
  source[sourceIdx] = copy[rightIdx];
  rightIdx += 1;
  sourceIdx += 1;
}
```

Comparison of Sorting Algorithms

Name	Best	Average	Worst	Memory	Stable	Method
Selection	n^2	n^2	n^2	1	No	Selection
Bubble	n	n^2	n^2	1	Yes	Exchanging
Quick	n * log(n)	n * log(n)	n^2	1	Depends	Partitioning
Merge	n * log(n)	n * log(n)	n * log(n)	1	Yes	Merging

Searching Algorithms

Search Algorithm

- An algorithm for **finding** an item with specified properties among a collection of items
  - Typically answers either **True** or **False** to whether the item is present
    - It may return **where** the item is found

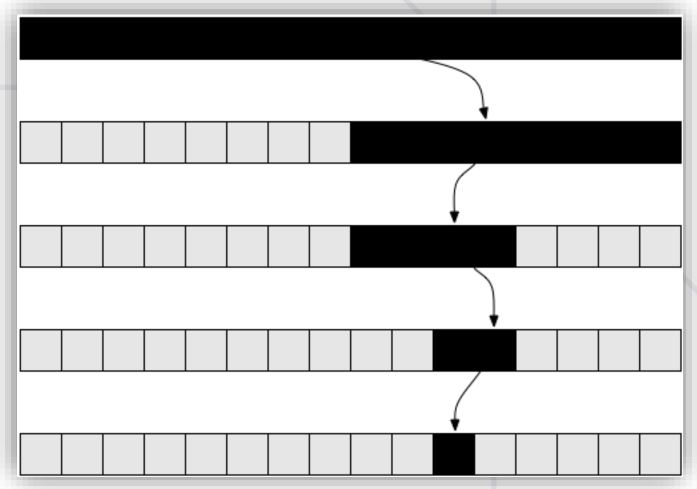
Linear Search

- **Linear search** finds a particular value in a list (**visualize**)
  - Searches the whole sequence
  - Checks every element **one** at a time
  - Searches until the desired one is **found**
- Worst and average performance: **O(n)**

```
for each item in the list:
  if that item has the desired value,
    return the item's location
return nothing
```

Binary Search

- **Binary search** finds an item within a ordered data structure
- At each step, compare the input with the middle element
  - The algorithm repeats its action to the left or right sub-structure
- See the **visualization**
- Complexity: O(log n)



Example: Binary Search (Iterative)

```
int BinarySearch(int arr[], int key, int start, int end)
{
  while (end >= start) {
    int mid = (start + end) / 2;
    if (arr[mid] < key)
      start = mid + 1;
    else if (arr[mid] > key)
      end = mid - 1;
    else
      return mid; }
  return KEY_NOT_FOUND;
}
```

Summary

- **Algorithm Complexity** – steps to execute: O(1), O(log n), O(n), O(n * log n), O(n^2), O(n^3), …
- **Recursion** – a function calls itself
- **Brute-Force** - trying all the possible solutions
- **Greedy** - picking a locally optimal solution
- **Sorting**
  - Slow algorithms: Selection and Bubble Sort
  - Fast algorithms: Quick and Merge Sort
- **Searching**
  - Linear and Binary

Software University

Questions?

SoftUni

- Software University
- SoftUni Creative
- SoftUni Digital
- SoftUni Foundation
- SoftUni Kids
- Finance Academy

SoftUni Diamond Partners

SmartIT

Coca-Cola HBC Bulgaria

SOFTWARE GROUP

DXC TECHNOLOGY

createX

Postbank

Решения за твоето утре

SUPER HOSTING .BG

DRAFT KINGS

VIVACOM

INDEAVR

Serving the high achievers

AMBITIONED

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License