# **Recursion and Backtracking**

Using recursion and backtracking, recursion vs Iteration



**SoftUni Team Technical Trainers** 







**Software University** 

https://softuni.bg

#### **Table of Contents**



- 1. Algorithmic Complexity
- 2. Recursion
- 3. Backtracking
  - The 8 Queens Problem
  - Finding All Paths in a Labyrinth Recursively
- 4. Recursion or Iteration?
  - Harmful Recursion and Optimizing Bad Recursion





# 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

### **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;
}</pre>
```

```
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
  - $T(n) = 3(n^2) + 3n + 3$
  - We can ignore some part of this equation
  - Higher terms dominate lower terms -n > 2,  $n^2 > n$ ,  $n^3 > n^2$
  - Multiplicative constants can be omitted 12n → n,
     2n² → n²
- The previous solution becomes  $\approx n^2$

### **Time Complexity**

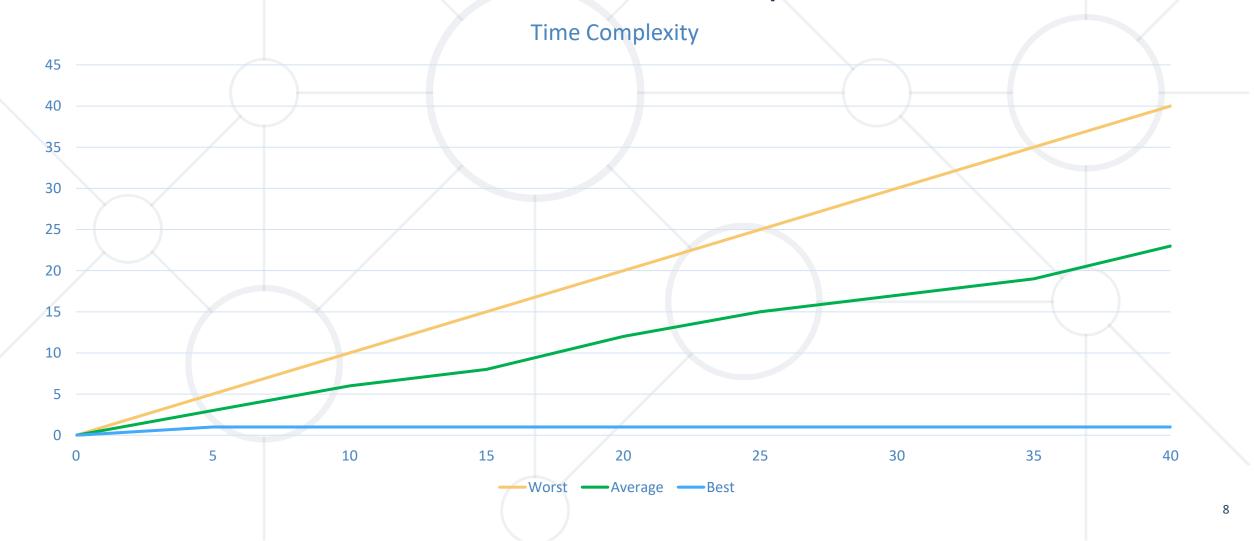


- Worst-case
  - An upper bound on the running time
- Average-case
  - Average running time
- Best-case
  - The lower bound on the running time (the optimal case)

# **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

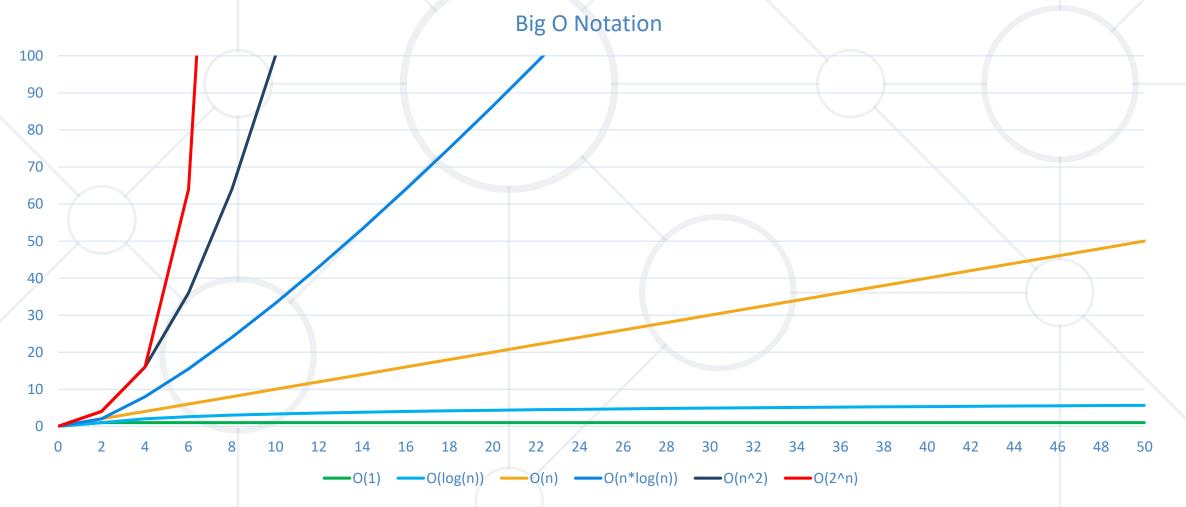


- Descriptions that allow us to examine an algorithm's running time
- There are three common asymptotic notations:
  - Big O O(f(n))
  - Big Theta  $-\Theta(f(n))$
  - Big Omega  $\Omega(f(n))$

## **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)	$n = 1000 \rightarrow 10$ operations
linear	O(n)	n = 1 000 → 1 000 operations
linearithmic	O(n*log n)	$n = 1000 \rightarrow 10000$ operations
quadratic	O(n2)	n = 1 000 $\rightarrow$ 1 000 000 operations
cubic	O(n3)	$n = 1000 \rightarrow 1000000000000000$
exponential	O(n^n)	n = 10 → 10 000 000 000 operations



#### What is Recursion?



 Method of solving a problem where the solution depends on solutions to smaller instances of the same problem

A common computer programing tactic is to divide a problem into sub-problems of the same type as the original, solve those sub-problems, and combine the results.



#### What is Recursion?



- A function or a method that calls itself one or more times until a specified condition is met
  - After the recursive call the rest code is processed from the last one called to the first



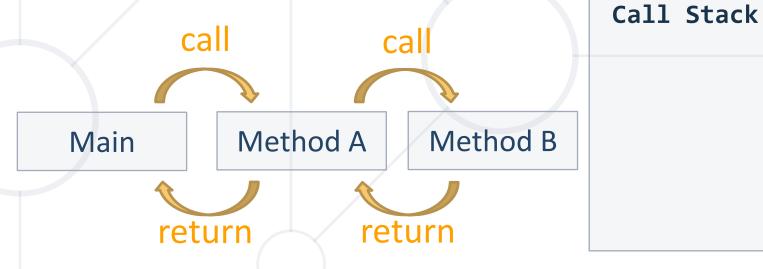


#### **Call Stack**



 "The stack" is a small fixed-size chunk of memory (e.g. 1MB)

Keeps track of the point to which each active subroutine should return control when it finishes executing



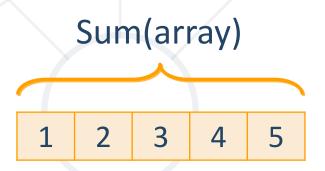




#### Other Definition

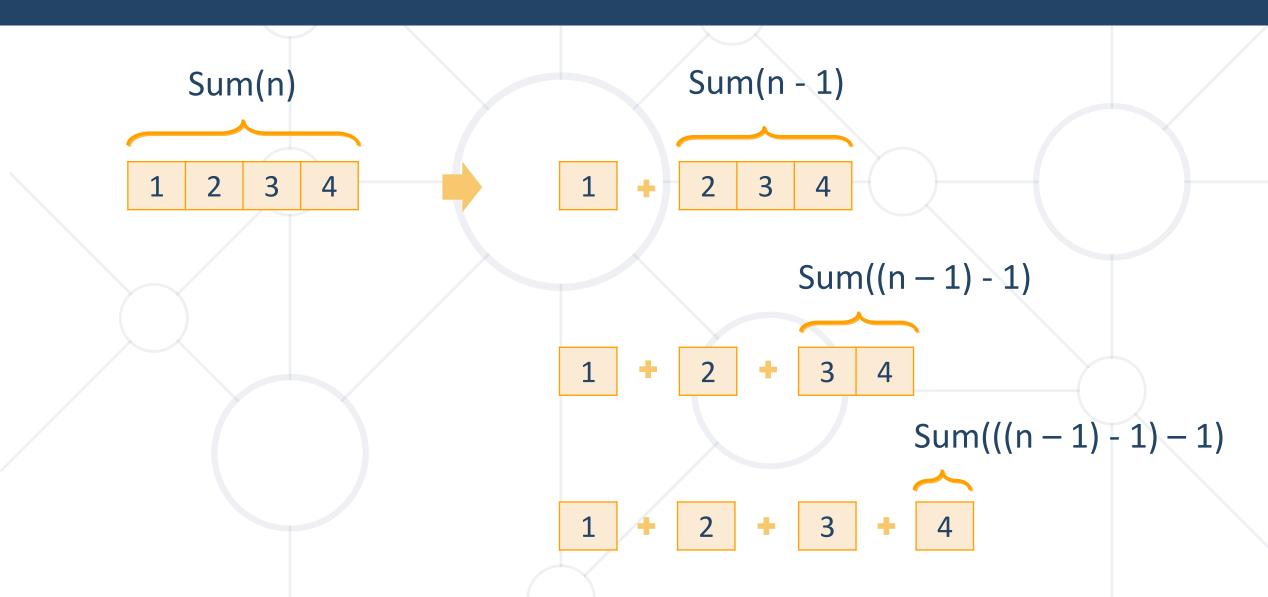


- Problem solving technique (In CS)
  - Involves a function calling itself
  - The function should have a base case
  - Each step of the recursion should move towards the base case



## Array Sum – Example

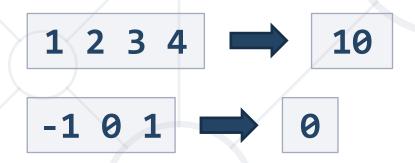




## **Problem: Array Sum**



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



#### **Solution: Array Sum**



```
static int Sum(int[] array, int index)
                                           Base case
   if (index == array.Length - 1)
      return array[index];
   return array[index] + Sum(array, index + 1);
                                              Recursive call
```

#### **Problem: Recursive Factorial**



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





#### Recursive Factorial – Example



Recursive definition of n! (n factorial):

#### **Solution: Recursive Factorial**



#### **Direct and Indirect Recursion**



- Direct recursion
  - A method directly calls itself
- Indirect recursion
  - Method A calls B, method B calls A
  - Or even  $A \rightarrow B \rightarrow C \rightarrow A$

#### **Recursion Pre-Actions and Post-Actions**



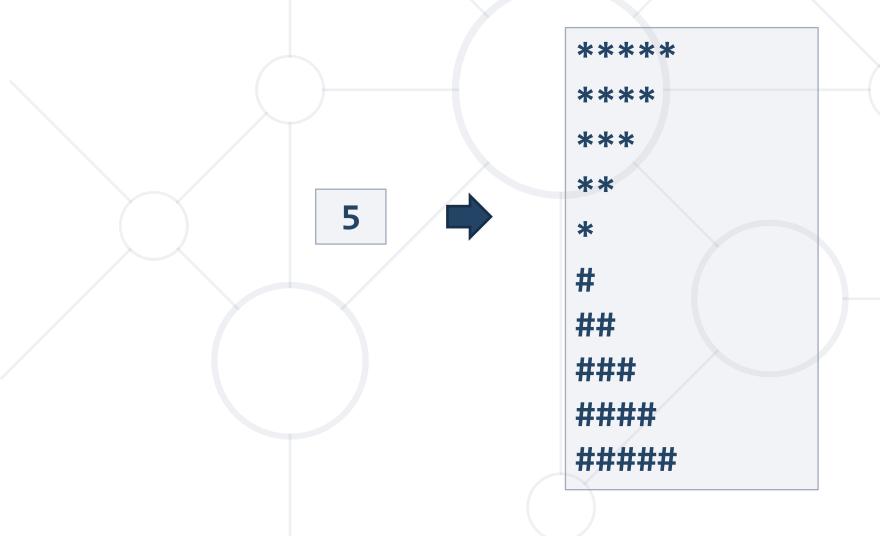
- Recursive methods have three parts:
  - Pre-actions (before calling the recursion)
  - Recursive calls (step-in)
  - Post-actions (after returning from recursion)

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

### **Problem: Recursive Drawing**



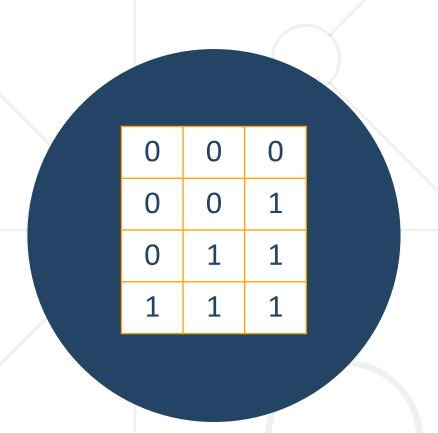
Create a recursive method that draws the following figure



#### **Pre-Actions and Post-Actions – Example**



```
static void PrintFigure(int n)
    if (n == 0)
      return;
    // TODO: Pre-action: print n asterisks
    PrintFigure(n - 1);
    // TODO: Post-action: print n hashtags
```



# Generating Simple Combinations Recursive Algorithm

# **Generating 0/1 Vectors**



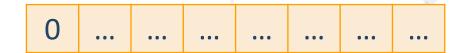
How to generate all 8-bit vectors recursively?

```
0 0 0 0 0 0 0
00000001
10000000
```

# **Generating 0/1 Vectors**



Start with a blank vector



Choose the first position and loop through all possibilities

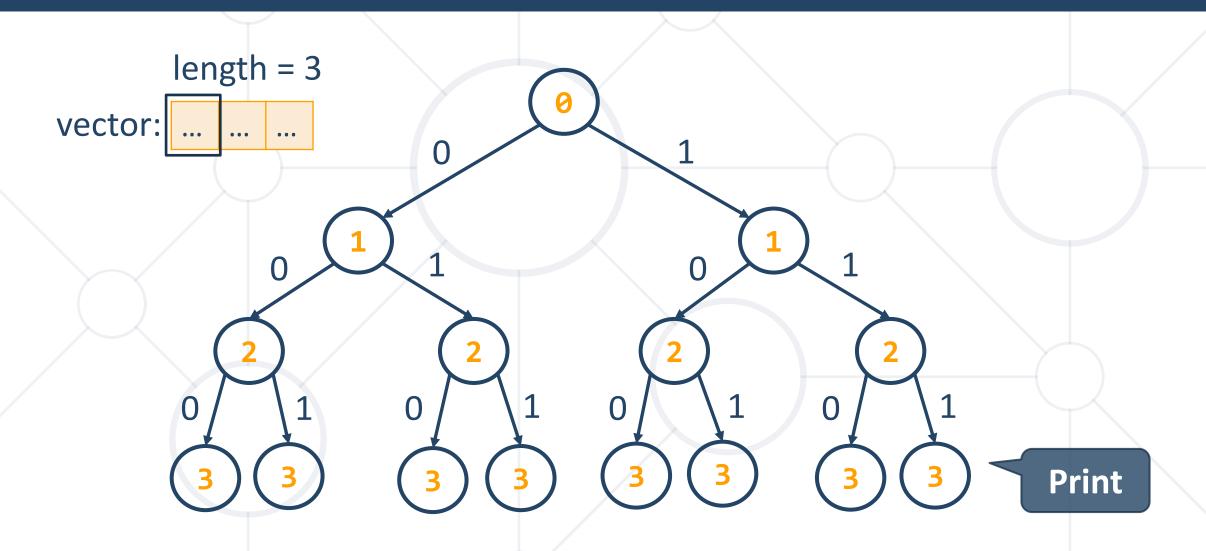




■ For each possibility, generate all (n − 1)-bit vectors

# **Generating 3-bit Vectors Recursion Tree**





#### **Solution: Generate n-bit Vectors**



```
static void Gen01(int index, int[] vector)
   if (index >= vector.Length)
      Console.WriteLine(string.Join(" ", vector));
   else
      for (int i = 0; i <= 1; i++)
          vector[index] = i;
          Gen01(index + 1, vector);
```



**Generating All Candidates** 

## Backtracking

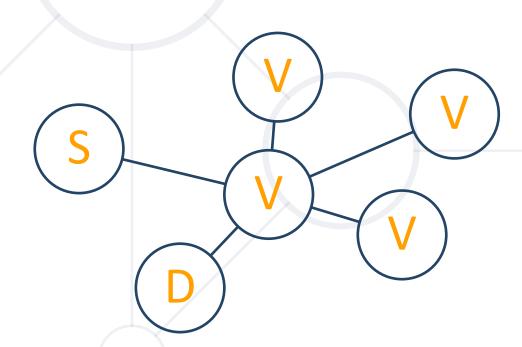


What is backtracking?



■ E.g., find all paths from Source to Destination





### Backtracking



- How does backtracking work?
  - At each step tries all perspective possibilities recursively
  - Drop all non-perspective possibilities as early as possible
- Backtracking has exponential running time

#### **Backtracking Algorithm (Pseudocode)**



```
static void recurrence(Node node) {
  if (node is solution) {
    printSolution(node);
  } else {
    for each child c of node
      if (c is perspective candidate) {
     markPositionVisited(c);
     recurrence(c);
     unmarkPositionVisited(c);
```

# Finding All Paths in a Labyrinth



- We are given a labyrinth
  - Represented as matrix of cells of size M x N
  - Empty cells '-' are passable, the others '\*' are not
- We start from the top left corner and can move in all
   4 directions (up, down, left, right)
- We want to find all paths to the exit, marked 'e'

# Finding All Paths in a Labyrinth (2)



There are 3 different paths from the top left corner to the bottom right corner:

0	1	2	*	-	-	-
*	*	3	*	-	*	-
6	5	4	-	-	_	-
7	*	*	*	*	*	-
8	9	10	11	12	13	14

0	1	2	*	8	9	10
*	*	3	*	7	*	11
-	-	4	5	6	-	12
_	*	*	*	*	*	13
-	-	-	-	-	-	14

0	1	2	*	-	-	-
*	*	3	*	-	*	-
-	-	4	5	6	7	8
_	*	*	*	*	*	9
-	-	-	-	-	-	10

RRDDLLDDRRRRRR

RRDDRRUURRDDDD

RRDDRRRRDD

# Find all Paths in a Labyrinth

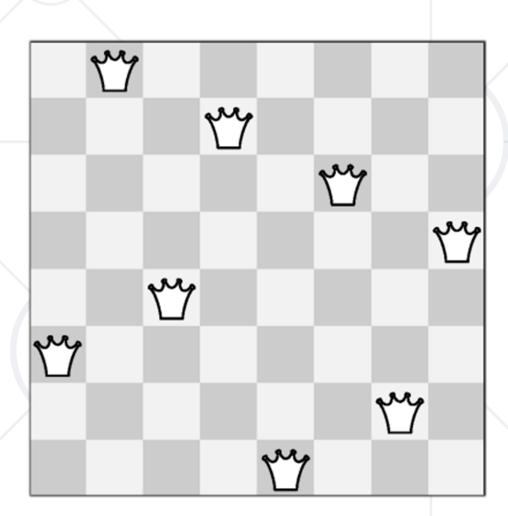


- Create a List<char> that will store the path
- Pass a direction at each recursive call (L, R, U or D)
- Add the direction at the start of each recursive call
- If you find an exit of the lab, then print the path
- Otherwise, mark the current cell as visited and try visit all possible directions
- Unmark the current cell and remove the last direction at the end of each recursive call

## The "8 Queens" Puzzle



- Write a program to find all possible placements of
  - 8 queens on a chessboard
  - So that no two queens can attack each other
  - http://en.wikipedia.org/wiki/
     Eight queens puzzle



# Solving The "8 Queens" Puzzle



- Find all solutions to "8 Queens Puzzle"
- At each step:
  - Put a queen at free position
  - Recursive call
  - Remove the queen

```
static void PutQueens(row) {
  if (row == 8)
     PrintSolution();
  else
    for (col = 0 ... 7)
      if (CanPlaceQueen(row, col)) {
        SetQueen(row, col);
        PutQueens(row + 1);
        RemoveQueen(row, col);
```



# Recursion or Iteration? When to Use and When to Avoid Recursion?

#### **Performance: Recursion vs. Iteration**



- Recursive calls are slower
- Parameters and return values travel through the stack
- Good for branching problems

```
static long Fact(int n)
{
   if (n == 0) {
     return 1;
   }
   return n * Fact(n - 1);
}
```

- No function call cost
- Creates local variables
- Good for linear problems (no branching)

```
static long Fact(int n)
{
  long result = 1;
  for (int i = 1; i <= n; i++)
    result *= i;
  return result;
}</pre>
```



#### **Infinite Recursion**



- Infinite recursion == a method calls itself infinitely
  - Typically, infinite recursion == bug in the program
  - The bottom of the recursion is missing or wrong
  - In C# / Java / C++ causes "stack overflow" error

```
Microsoft Visual Studio Debug Console

Stack overflow.

Press any key to close this window . . .
```

#### **Recursion Can be Harmful!**



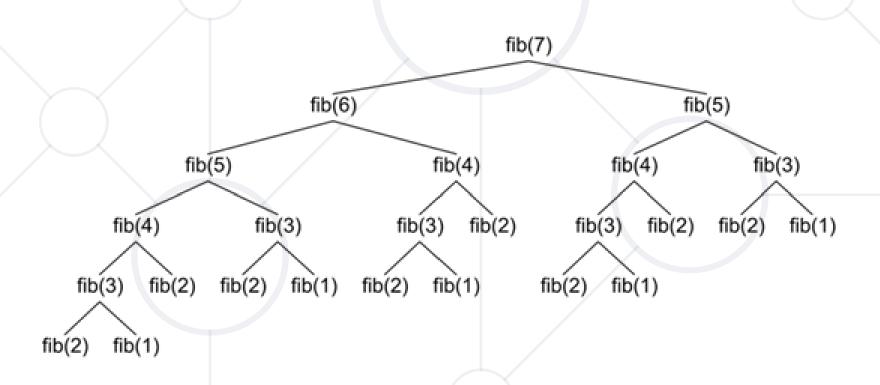
 When used incorrectly recursion could take too much memory and computing power

```
static long CalcFib(int number)
  if (number <= 1)</pre>
    return 1;
  return CalcFib(number - 1) + CalcFib(number - 2);
Console.WriteLine(CalcFib(10)); // 89
Console.WriteLine(CalcFib(50)); // This will hang!
```

#### How the Recursive Fibonacci Calculation Works?



- fib(n) makes about fib(n) recursive calls
- The same value is calculated many, many times!



#### When to Use Recursion?



- Avoid recursion when an obvious iterative algorithm exists
  - Examples: factorial, fibonacci numbers
- Use recursion for combinatorial algorithms were
  - At each step you need to recursively explore more than one possible continuation, i.e., branched recursive algorithms



## Summary

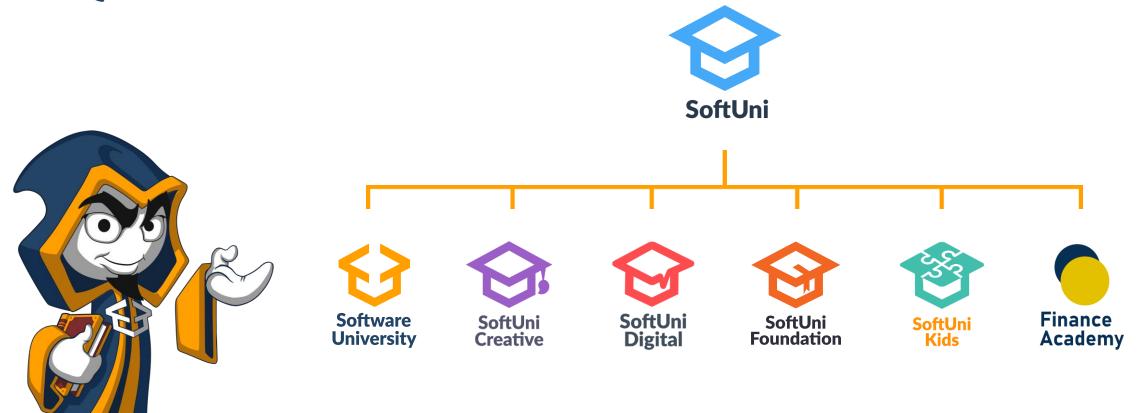


- Algorithmic Complexity
- Recursion
- Backtracking
- When to use recursion
- When to use iteration





# Questions?



#### **SoftUni Diamond Partners**



SUPER HOSTING .BG

















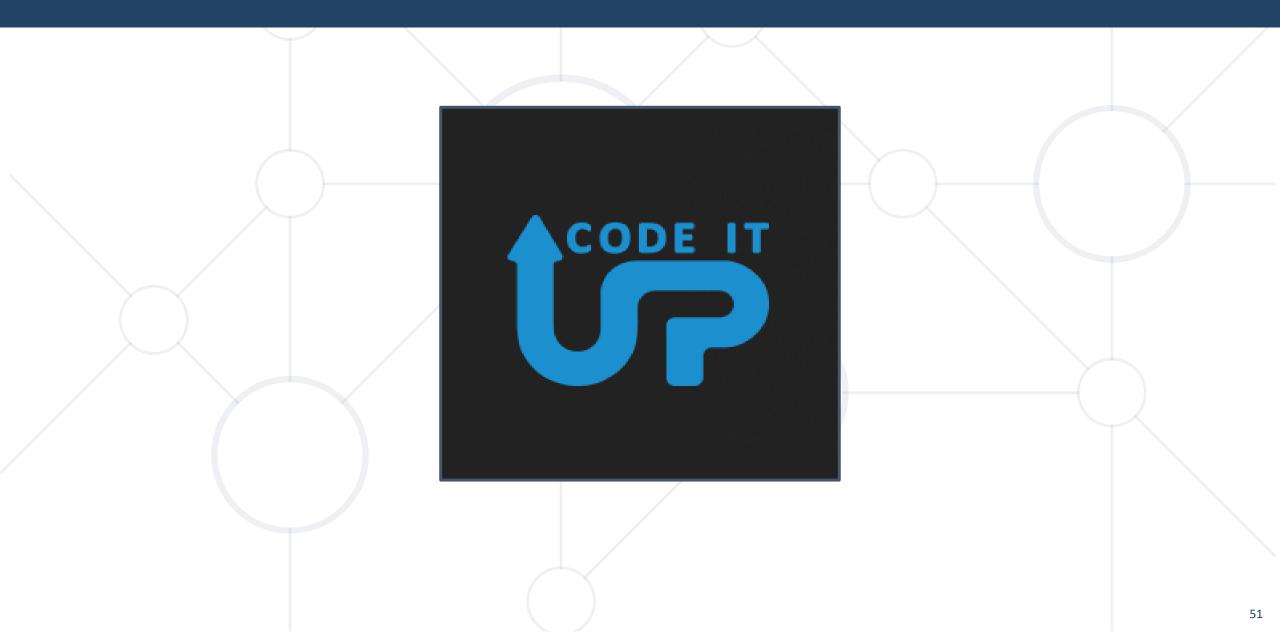






# **Educational Partners**





#### License



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is copyrighted content
- Unauthorized copy, reproduction or use is illegal
- © SoftUni <a href="https://about.softuni.bg/">https://about.softuni.bg/</a>
- © Software University <a href="https://softuni.bg">https://softuni.bg</a>





Software University – High-Quality Education, Profession and Job for

Software Developers

softuni.bg

- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg







