Recursion and Backtracking

Using recursion and backtracking, recursion vs Iteration

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

http://softuni.bg

Table of Contents

Software University

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

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

Problem: Get Number of Steps

- Calculate maximum steps to find the result

```
def get_operations_count(n):
    counter = 0
    for i in range(n):
        for j in range(n):
            counter += 1
    return counter
```

Solution:
$T(n) = 3(n \wedge 2) + 3n + 3$

- 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 $\mathbf{\approx n^2}$

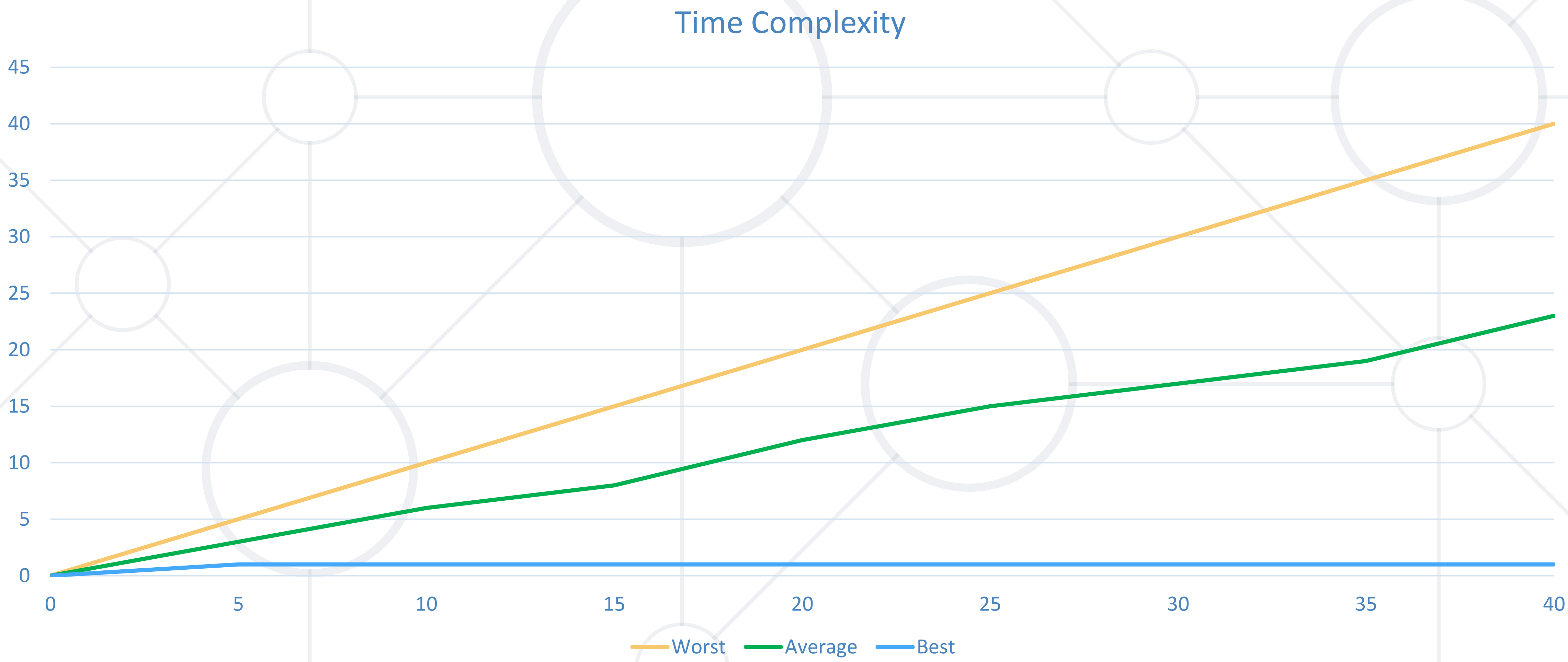
Time Complexity

- 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

Software University

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



Time Complexity

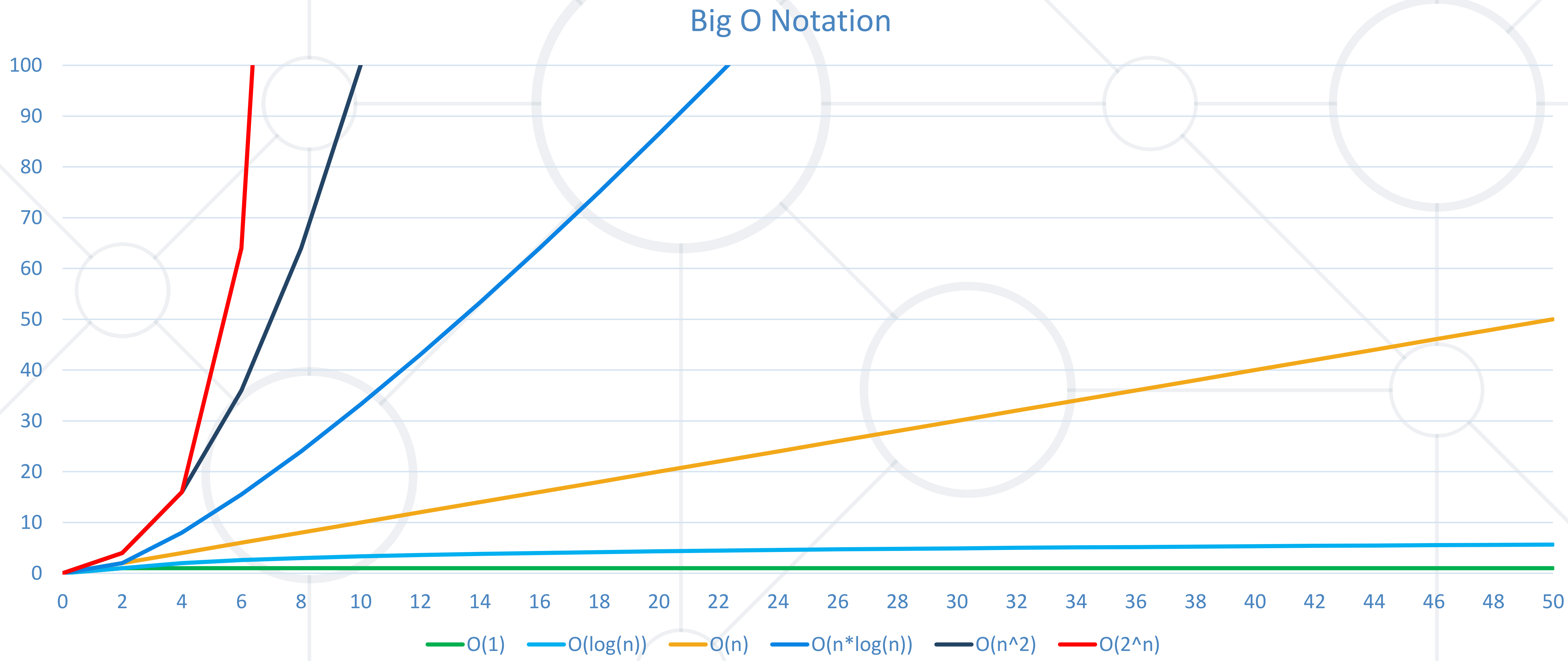
- From the previous chart we can deduce:
  - For smaller size of the input (**n**) we **don't care much for the runtime**
  - So we measure the time as **n** approaches **infinity**
  - If an algorithm **must scale**, it **should compute** the result within a **finite and practical time**
  - We're concerned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

Recursion

What is Recursion?

What is Recursion?

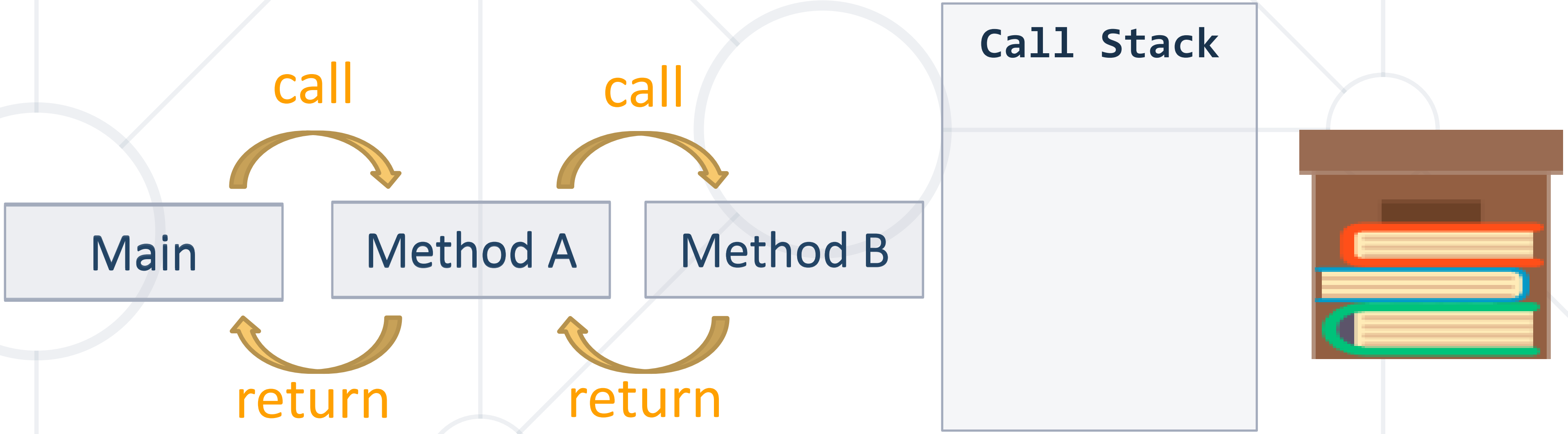
- **Method** of solving a problem where the solution depends on solutions to smaller instances of the same problem
- A common **computer programing tactic** is to **divide** a problem into **sub-problems** of the same type as the original, **solve** those sub-problems, and **combine** the **results**

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

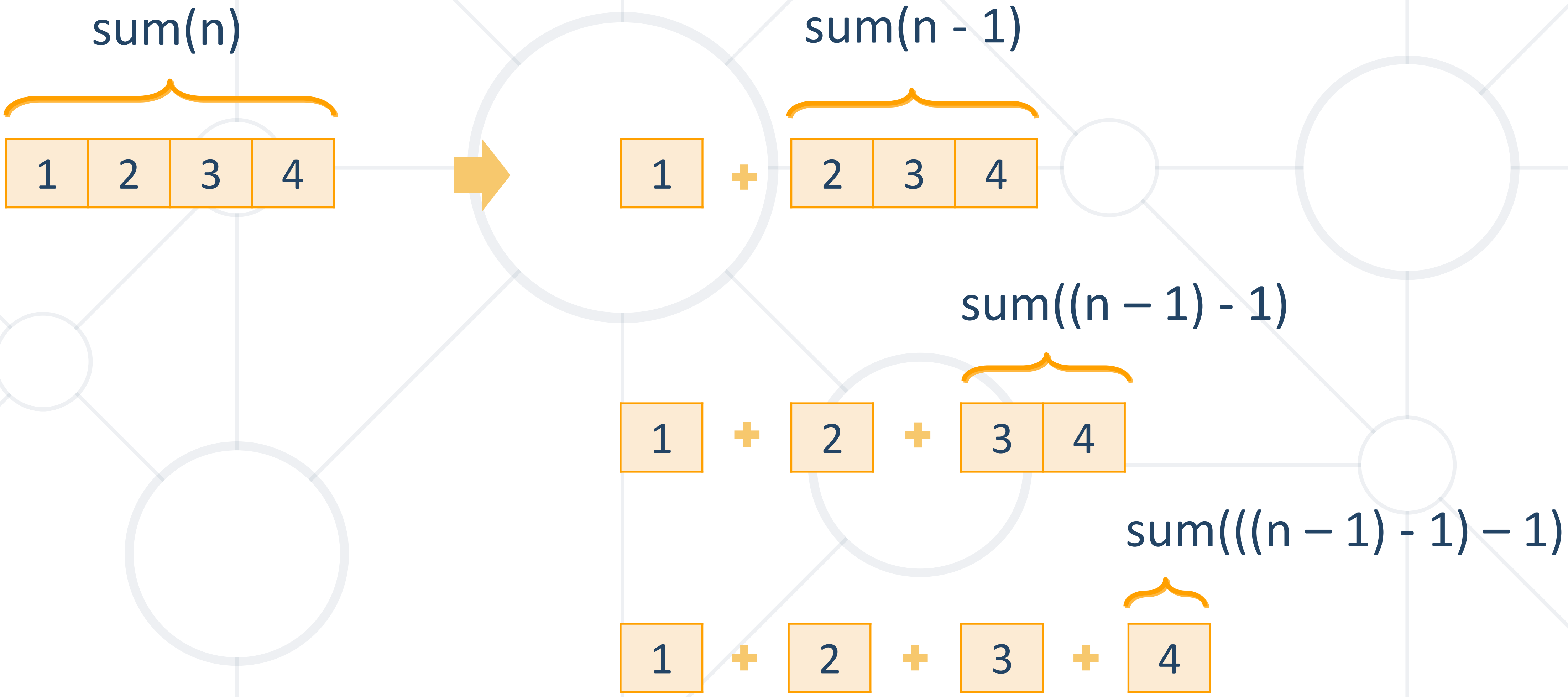
- "The stack" is a small **fixed-size** chunk of memory (e.g. 1MB)
- Keeps track of **the point** to which each active subroutine should **return control** when it **finishes executing**



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 `array`
  - Read numbers from the console

```
1 2 3 4  →  10
```

```
-1 0 1  →  0
```

Solution: Array Sum

```
def calc_sum(numbers, idx):
    if idx == len(numbers) - 1:
        return numbers[idx]
    return numbers[idx] + calc_sum(numbers, idx + 1)
```

Base case

Recursive call

Problem: Recursive Factorial

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

5	→	120
10	→	3628800

Recursive Factorial – Example

- Recursive definition of **n!** (n factorial):

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

3!	=	3 * 2!
2!	=	2 * 1!
1!	=	1 * 0!
0!	=	1

Solution: Recursive Factorial

```
def get_factorial(num):
    if num == 0:
        return 1
    return num * get_factorial(num - 1)
```

Base case

Recursive call

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**

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)

```
def recursion()
  # Pre-actions
  recursion()
  # Post-actions
```

Problem: Recursive Drawing

- Create a **recursive method** that draws the following figure

```
5
```

```
*****
****
***
**
*
#
##
###
####
#####
```

Pre-Actions and Post-Actions – Example

```python
def print_figure(n):
    if n == 0:
        return
    # TODO: Pre-action: print n asterisks
    print_figure(n - 1)
    # TODO: Post-action: print n hashtags
```

0	0	0
0	0	1
0	1	1
1	1	1

Generating Simple Combinations

Recursive Algorithm

Generating 0/1 Vectors

- How to generate all 8-bit vectors **recursively**?

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
...
0 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0
...
1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1
```

Generating 0/1 Vectors

- Start with a **blank vector**

0	…	…	…	…	…	…	…

- Choose the **first position** and **loop through all possibilities**

0	…	…	…	…	…	…	…

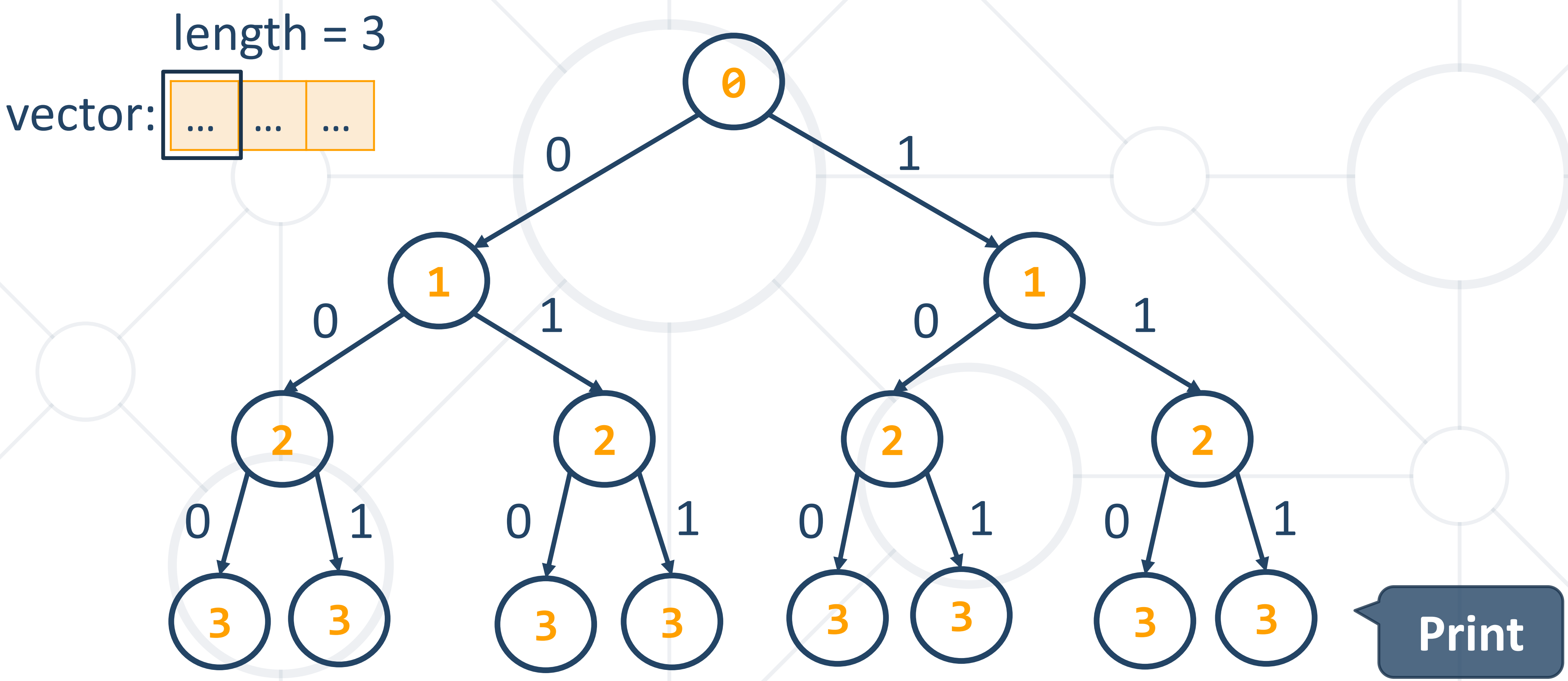
n - 1

1	…	…	…	…	…	…	…

n - 1

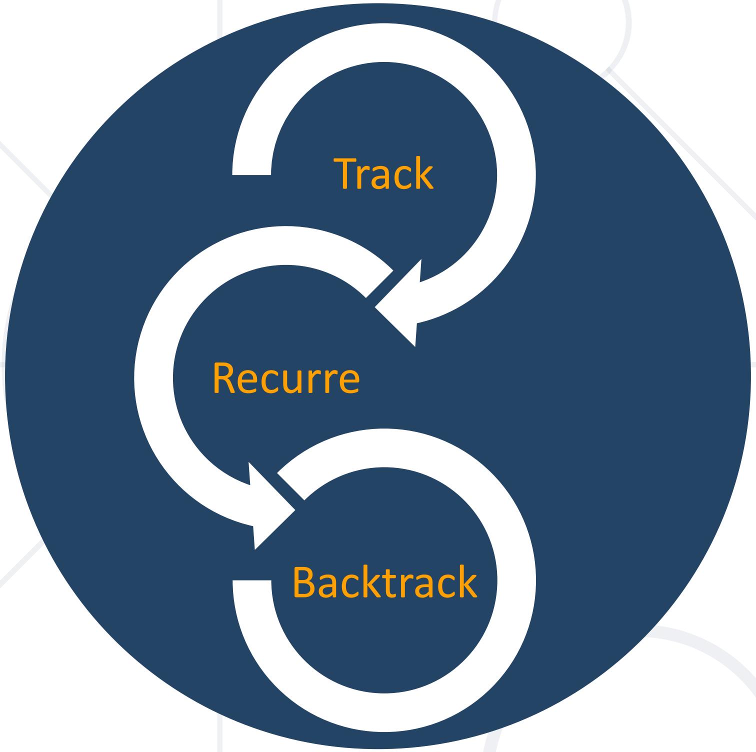
- For each possibility, generate all **(n – 1)-bit** vectors

Generating 3-bit Vectors Recursion Tree



Solution: Generate n-bit Vectors

```python
def gen01(idx, vector):
    if idx >= len(vector):
        print("".join([str(x) for x in vector]))
        return
    for number in range(0, 2):
        vector[idx] = number
        gen01(idx + 1, vector)
```



Backtracking

Generating All Candidates

Backtracking

- What is **backtracking**?
  - Class of algorithms for **finding all solutions**
    - E.g., find all paths from Source to Destination
- How does backtracking work?
  - At each step **tries all perspective possibilities** recursively
  - **Drop** all **non-perspective possibilities** as early as possible

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

- 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

RRDDLLDDRRRRRR

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

RRDDRRUURRDDDD

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

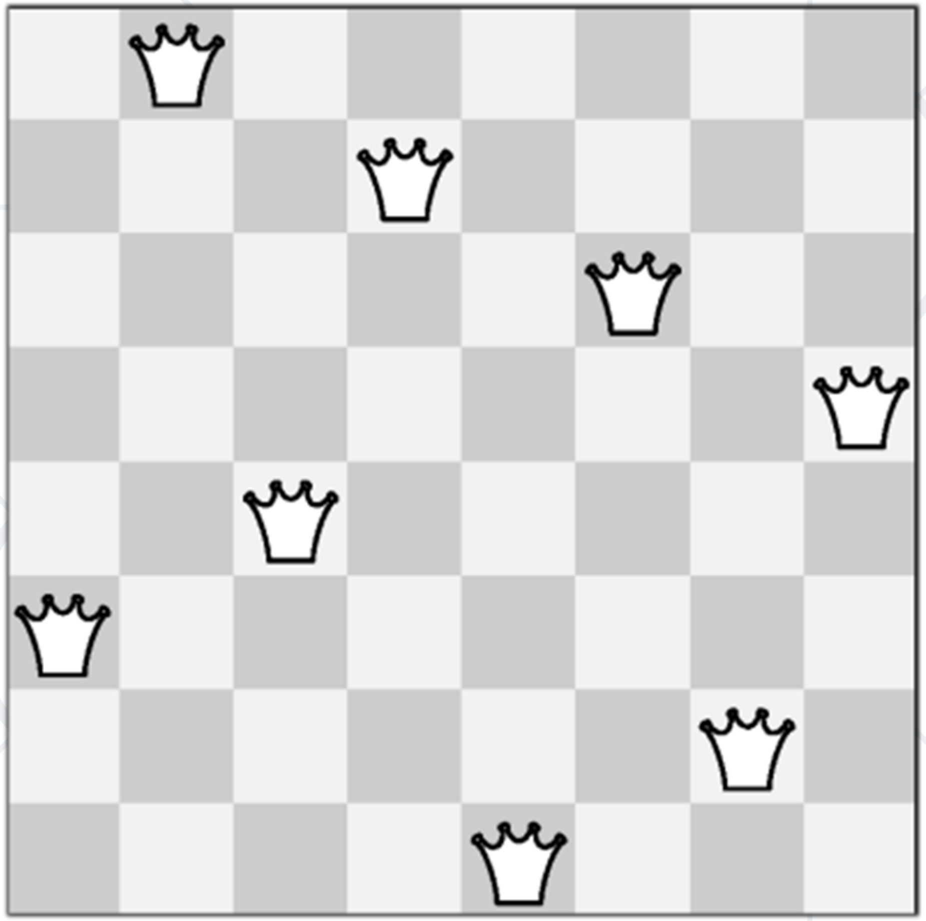
RRDDRRRRDD

Find all Paths in a Labyrinth

- Create a **list** 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

```
def put_queens(row):
    if row == 8:
        print_solution()
        return
    for col in range(0, 8):
        if can_place_queen(row, col):
            set_queen(row, col)
            put_queens(row + 1)
            remove_queen(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

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)
```

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

```
def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

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 Python / C# / Java / C++ causes "stack overflow" error

```
Traceback (most recent call last):
  File "C:\Users\aatanasov\PycharmProjects\demo\main.py", line 5, in <module>
    infinite_recursion()
  File "C:\Users\aatanasov\PycharmProjects\demo\main.py", line 2, in infinite_recursion
    infinite_recursion()
  File "C:\Users\aatanasov\PycharmProjects\demo\main.py", line 2, in infinite_recursion
    infinite_recursion()
  File "C:\Users\aatanasov\PycharmProjects\demo\main.py", line 2, in infinite_recursion
    infinite_recursion()
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded

Process finished with exit code 1
```

Recursion Can be Harmful!

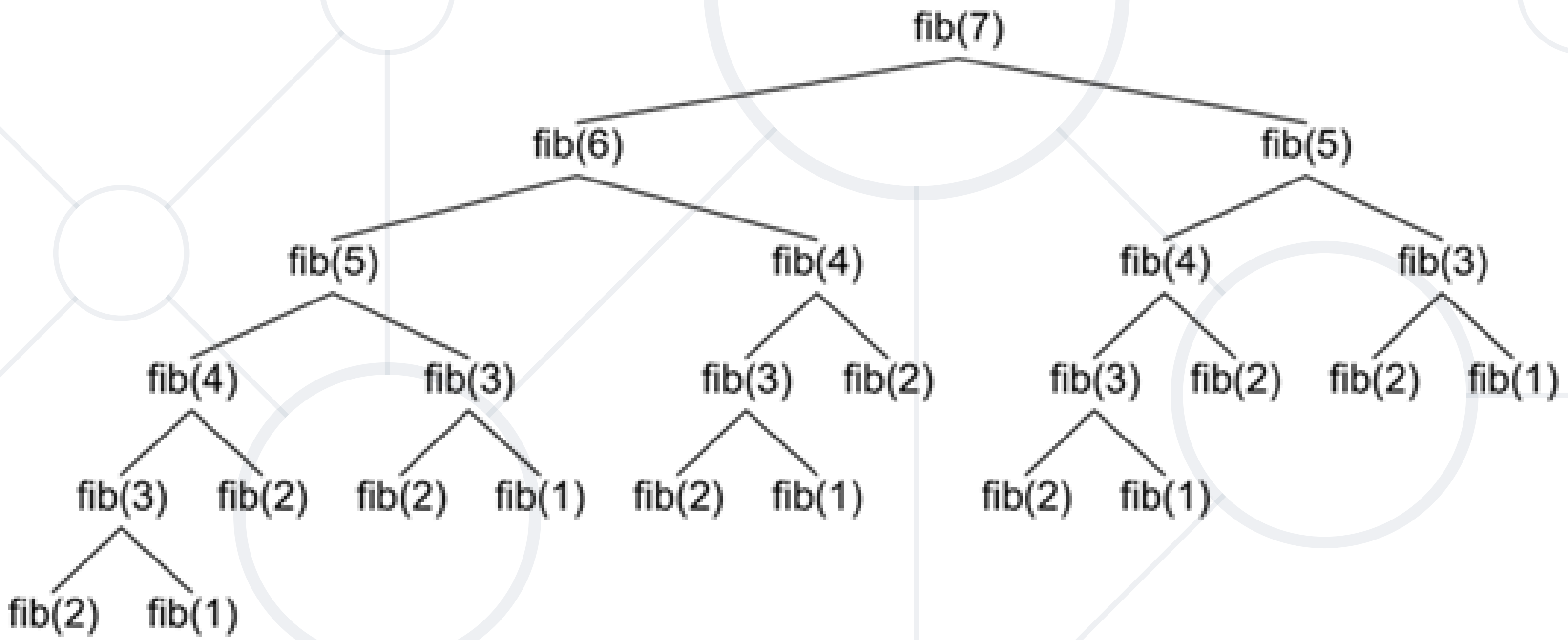
- When used incorrectly recursion could take too much memory and computing power

```python
def calc_fib(number):
    if number <= 1:
        return 1
    return calc_fib(number - 1) + calc_fib(number - 2)


print(calc_fib(10)) # 89
print(calc_fib(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 where:
  - At each step you need to **recursively** explore more than one possible continuation
    - **Branched** recursive algorithms

Summary

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

Questions?

SoftUni

Software University | SoftUni Creative | SoftUni Digital | SoftUni Foundation | SoftUni Kids | СофтУни БУДИТЕЛ | Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

THE CROWN IS YOURS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

PHAR VISION

Trainings @ Software University (SoftUni)

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

License

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