Data Structures and Algorithms

LECTURE 03: RECURSION AND BACKTRACKING

UNIVERSITY of GREENWICH
Alliance with FPT Education
Pearson BTEC

Contents

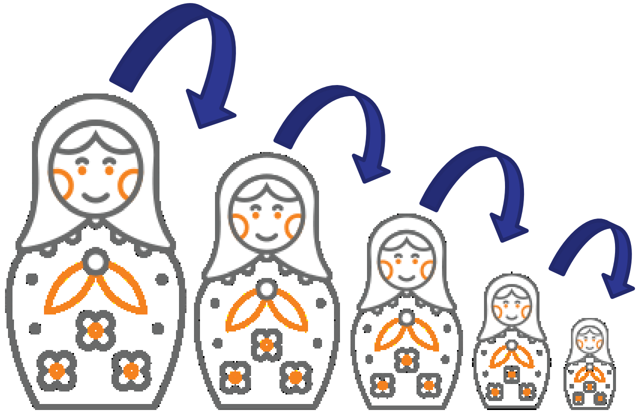
- Recursion
- Generating Simple Combinations
- Backtracking
  - – The 8 Queens Problem
  - – Finding All Paths in a Labyrinth Recursively
- Recursion or Iteration?
  - – Harmful Recursion and Optimizing Bad Recursion

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**.

Recursion

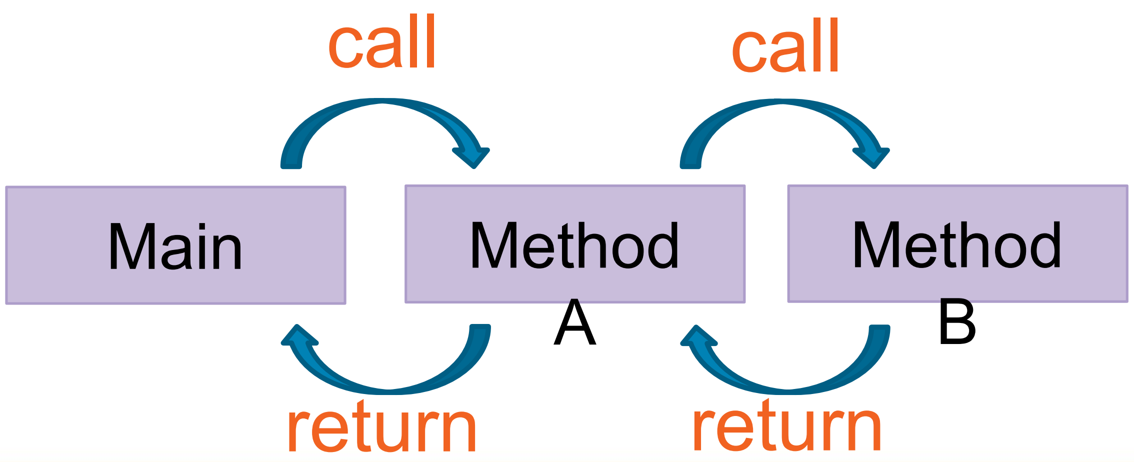
- What is Recursion?
  - – A function or a method that **calls itself one or more** times until a specified **condition** is **met**
  - – After the recursive call the rest code is processed **from** the **last** one called **to** the **first**.



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**

Call Stack



call

call

Main

Method A

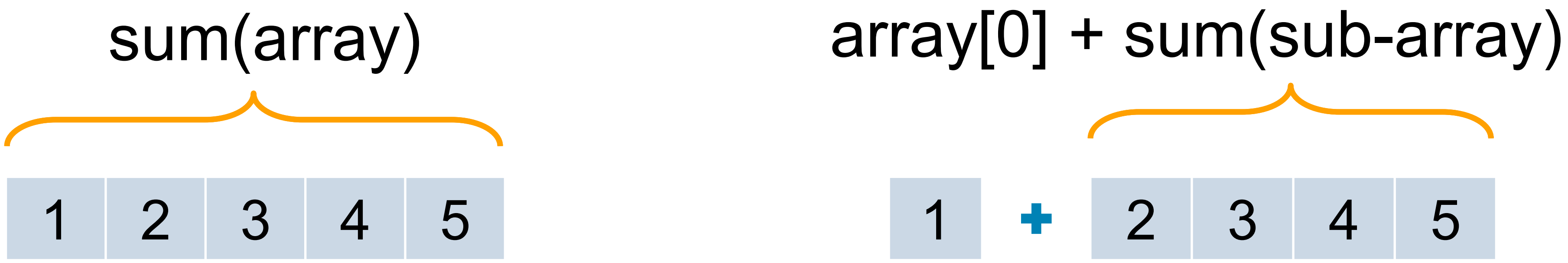
Method B

return

return

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**



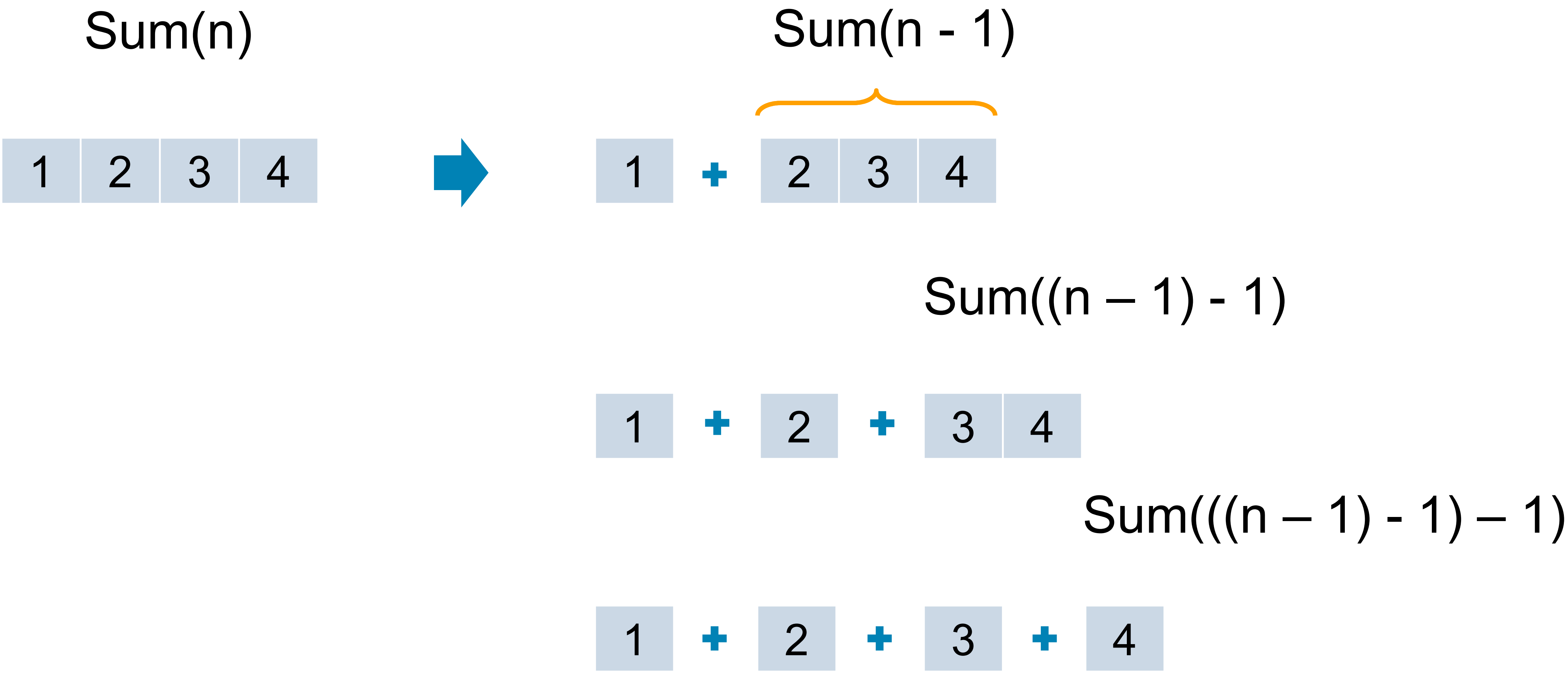
sum(array)

1 2 3 4 5

array[0] + sum(sub-array)

1 + 2 3 4 5

Array Sum – Example



Sum(n)

| 1 | 2 | 3 | 4 |

Sum(n - 1)

1 + | 2 | 3 | 4 |

Sum((n – 1) - 1)

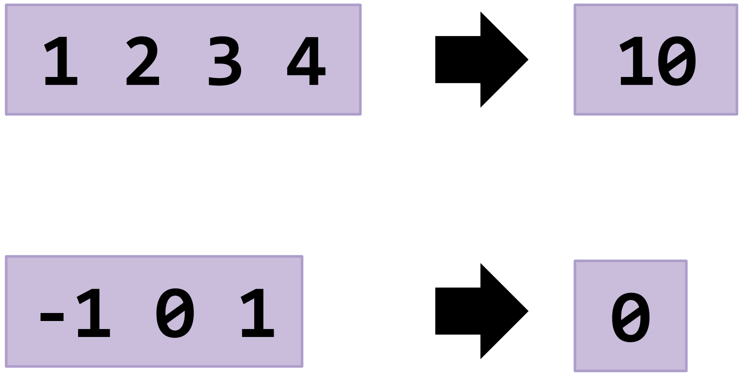
1 + 2 + | 3 | 4 |

Sum(((n – 1) - 1) – 1)

1 + 2 + 3 + 4

Problem: Array Sum

- Create a **recursive method** that
  - – Finds the sum of all numbers stored in an
  - – Read numbers from the console



1 2 3 4 → 10

-1 0 1 → 0

Solution: Array Sum

- Sample source code

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


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

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

- Sample source code

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

    return num * factorial(num - 1);
}
```

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)

```
static void 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

- Sample source code

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

Recursive Algorithm

- Generating Simple Combinations

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**

$$0\ \underbrace{\ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots}_{n\ -\ 1} \qquad 1\ \underbrace{\ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots}_{n\ -\ 1}$$

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

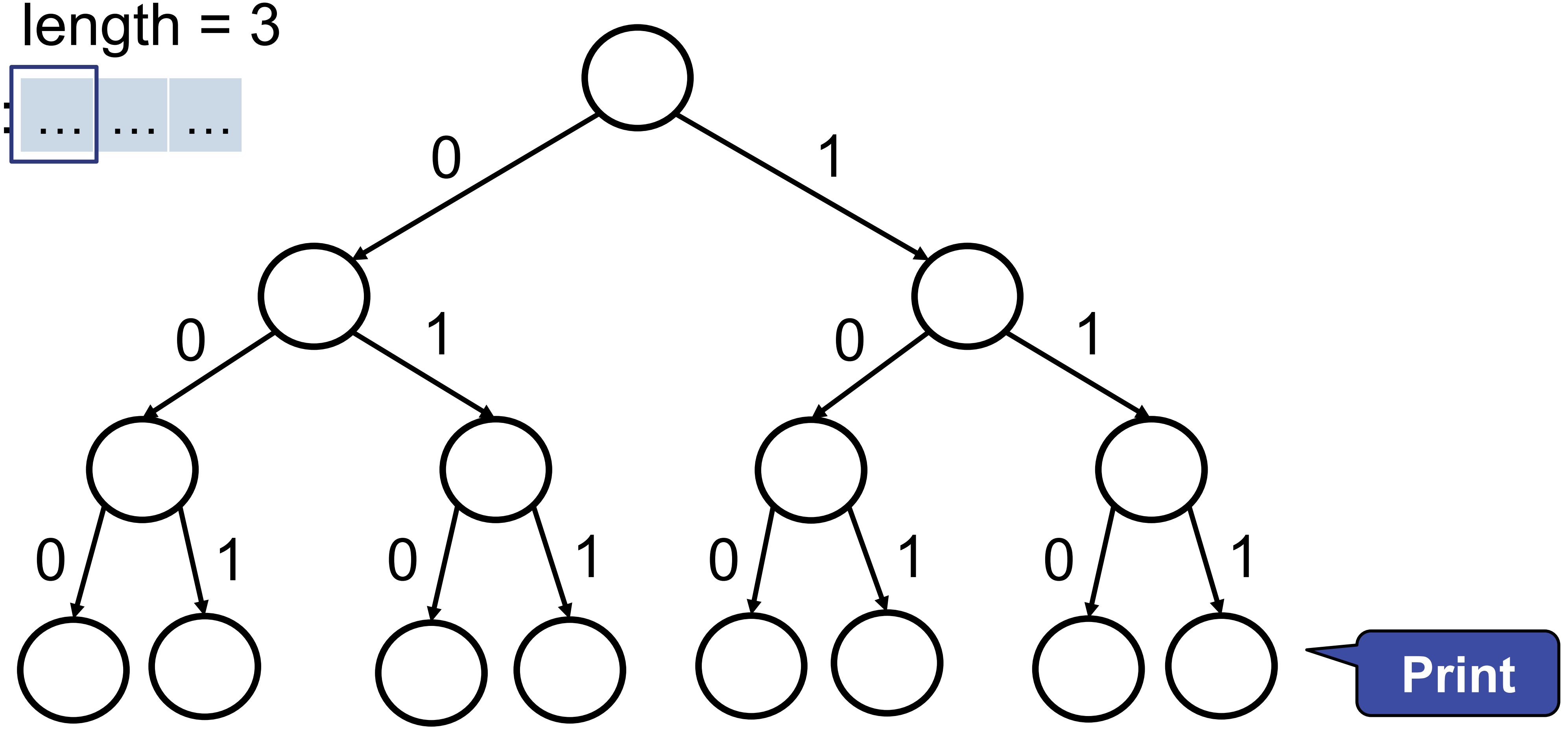
Solution: Generate n-bit Vectors

```
static void gen01(int index, int[] vector) {
   if (index >= vector.length()) {
      print(vector);
   } else {
      for (int i = 0; i <= 1; i++) {
          vector[index] = i;
          gen01(index + 1, vector);
      }
   }
}
```

Generating 3-bit Vectors Recursion Tree

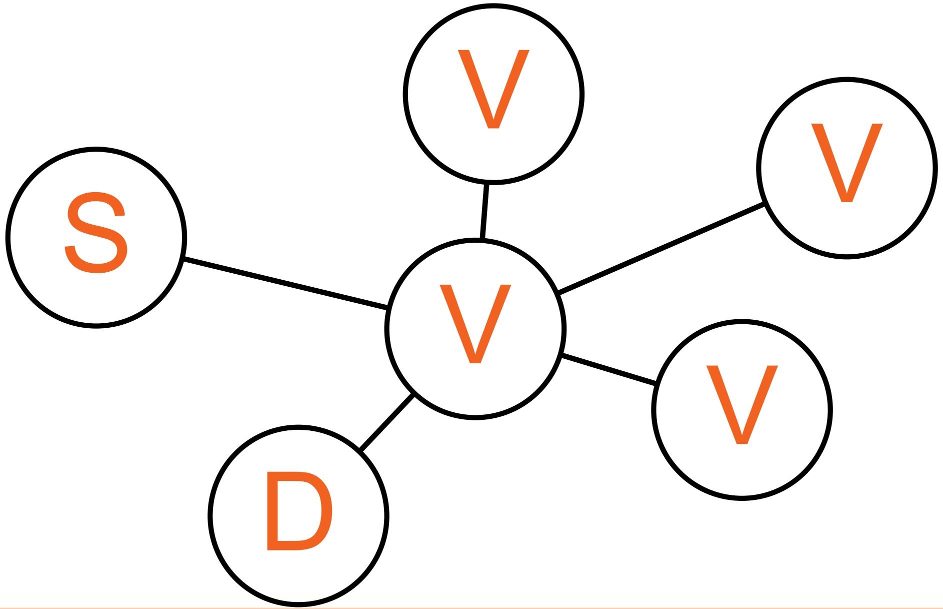
length = 3

vector: … … …



Backtracking

- What is **backtracking**?
  - Class of algorithms for **finding all solutions**
    - 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**

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

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: Algorithm (2)

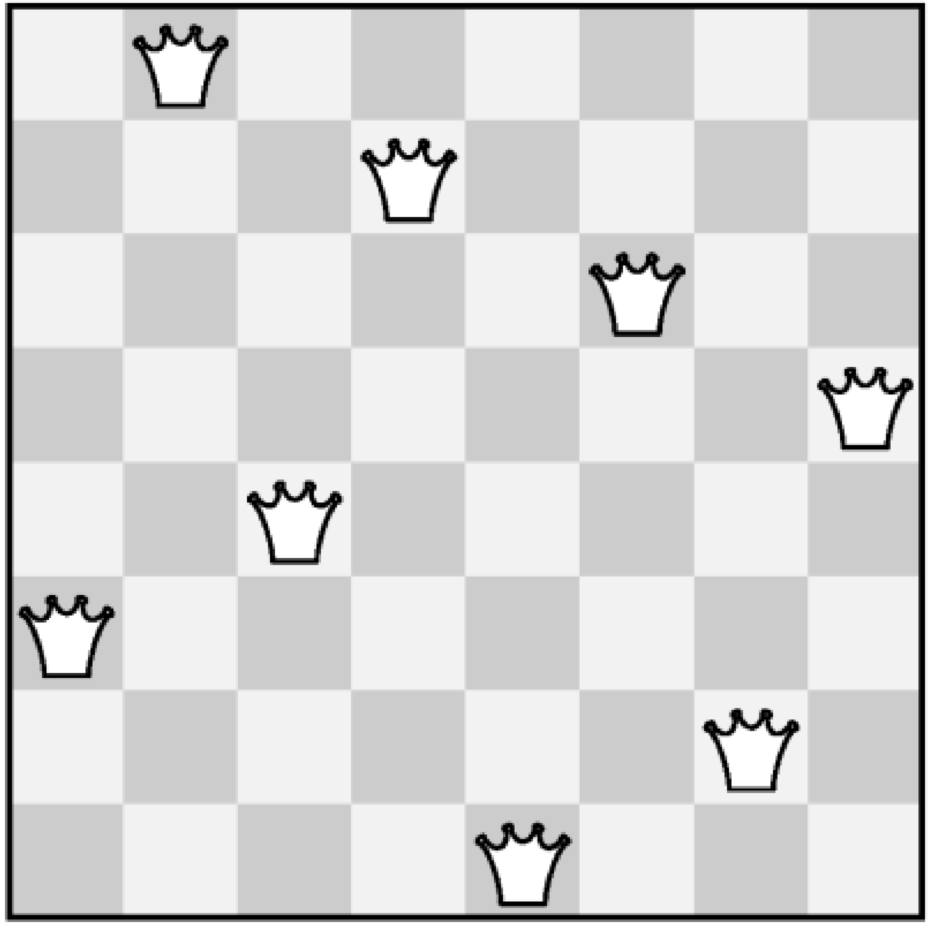
```
static void findPath(int row, int col) {
  if (!isInBounds(row, col)) return;
  if (isExit(row, col)) printPath();
  else if (!isVisited(row, col) && isPassable(row, col)) {
    mark(row, col);
    findPath(row, col + 1); // Right
    findPath(row + 1, col); // Down
    findPath(row, col - 1); // Left
    findPath(row - 1, col); // Up
    unmark(row, col);
  }
}
```

Find All Paths and Print Them

- Create a **`List<Character>`** that will store the path
- Pass a direction at each recursive call (**`L`**, **`R`**, **`U`** or **`D`**)
- At the start of each recursive call
  - **`Add direction`**
- At the end of each recursive call
  - **`Remove last direction`**

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);
      }
```

Performance: Recursion vs. Iteration

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

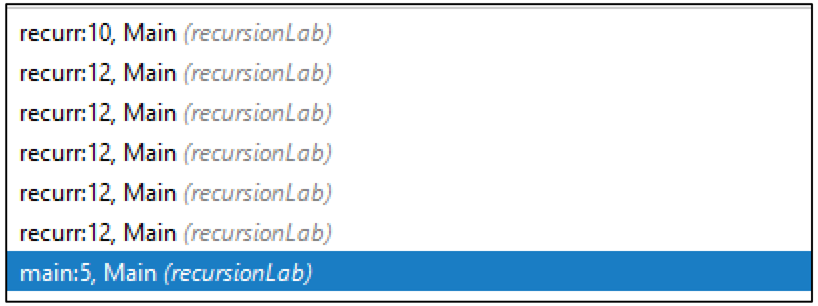
```
static long recurFact(int n) {
  if (n == 0)
    return 1;

  return n * Fact(n - 1);
}
```

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

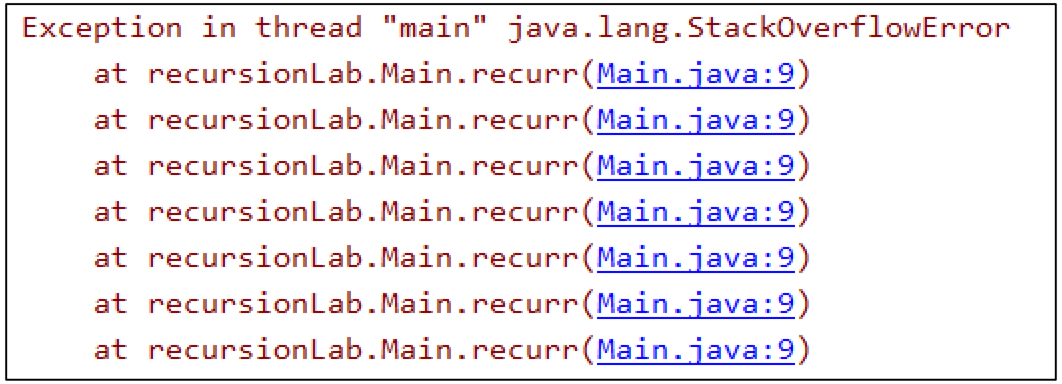
```
static long iterFact(int num) {
  long result = 1;
  for (int i = 1; i <= n; i++)
    result *= i;
  return result;
}
```

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



```
recurr:10, Main (recursionLab)
recurr:12, Main (recursionLab)
recurr:12, Main (recursionLab)
recurr:12, Main (recursionLab)
recurr:12, Main (recursionLab)
recurr:12, Main (recursionLab)
main:5, Main (recursionLab)
```



```
Exception in thread "main" java.lang.StackOverflowError
	at recursionLab.Main.recurr(Main.java:9)
	at recursionLab.Main.recurr(Main.java:9)
	at recursionLab.Main.recurr(Main.java:9)
	at recursionLab.Main.recurr(Main.java:9)
	at recursionLab.Main.recurr(Main.java:9)
	at recursionLab.Main.recurr(Main.java:9)
	at recursionLab.Main.recurr(Main.java:9)
```

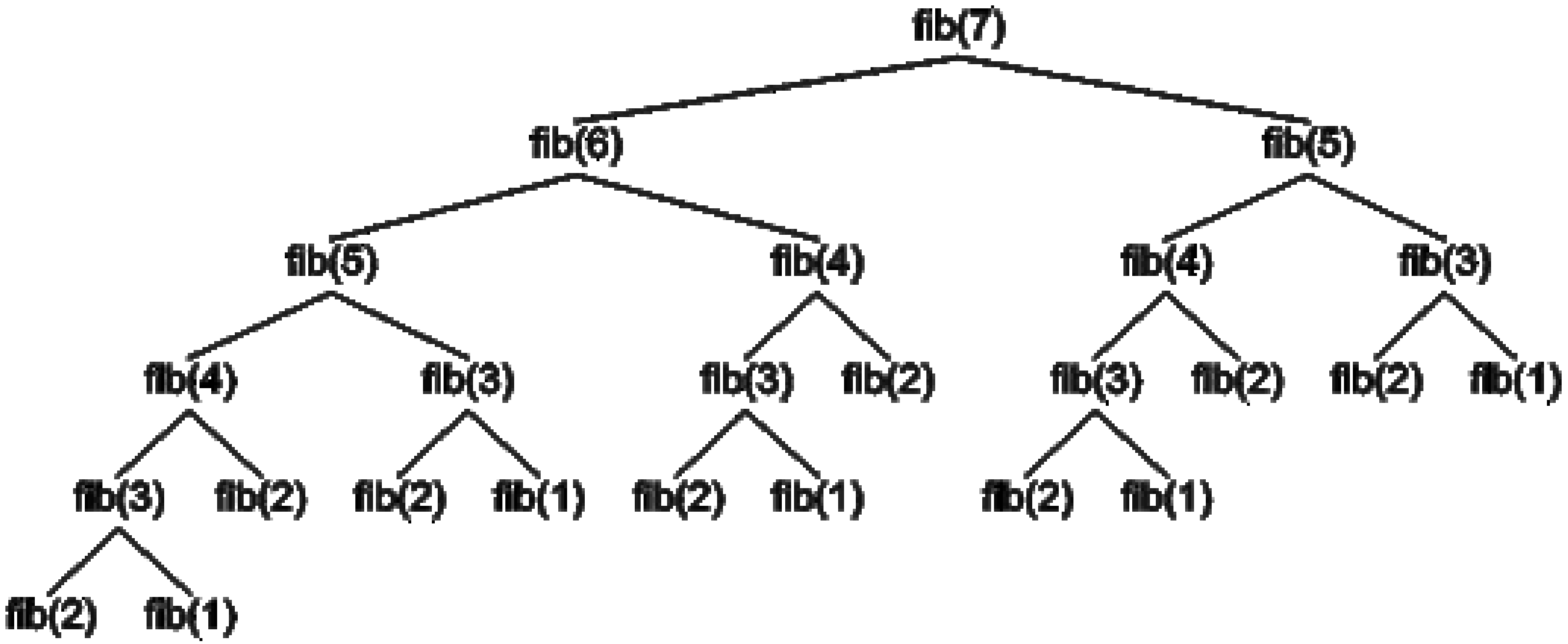
Recursion Can be Harmful!

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

```
static long fibonacci(int n) {
    if (n <= 1)
        return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

public static void main(String[] args) {
    System.out.println(fibonacci(10)); // 89
    System.out.println(fibonacci(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, i.e. **branched** recursive algorithms

Summary

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