|  |  |
| --- | --- |
| **CL2001**  **Data Structures Lab** | **Lab 5**  **Recursion and Backtracking** |

**National University of Computer and Emerging Sciences**

**Fall 2025**

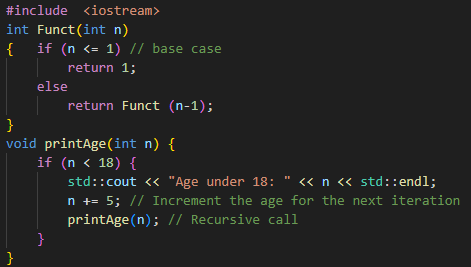
# **Lab Content**

1. Recursion and its base condition
2. Types of Recursions
3. Issues in Recursion
4. Backtracking

# **Recursion and its base condition**

A **recursive function** solves a problem by solving smaller versions of itself. Each recursive call reduces the problem's size, moving it closer to a simple, solvable case.

The **base condition** is a critical part of recursion. It tells the function when to stop calling itself. Without a base condition, the function would call itself infinitely, leading to a stack overflow error. An example is shown below:



**Key Points**: In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

# **Types of Recursions**

Recursion can be categorized into three main types based on how the function calls are made. Each type has its own structure and use case in problem-solving. These include**:**

1. Direct and Indirect Recursion
2. Tailed and non-tailed recursion
3. Nested Recursion

# **Direct and Indirect Recursions**

Recursion occurs when a function calls itself. If a function directly calls itself, it is **direct recursion**. When function A calls function B, and B calls A, it becomes **indirect recursion**. Both must include a base condition to avoid infinite loops.

**Sample Code (Direct Recursion)**

void X()

{

// Some code....

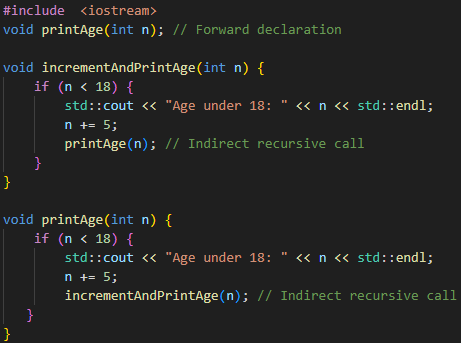
X();

// Some code...

}

Here, X() calls itself directly within its own body. This is a classic example of direct recursion.

**Sample Code (Indirect Recursion)**



**Key Points**: In this example, incrementAndPrintAge() calls printAge(), and printAge() calls back incrementAndPrintAge(). This forms an **indirect recursion cycle** between two functions.

# **Tailed and Non-Tailed Recursions**

Recursion can be classified based on the position of the recursive call. If the recursive call is the **last operation** in the function, it's called **tail recursion**. If there's more work to do **after** the recursive call returns, it's known as **non-tail recursion**.

**Sample Code (Tail Recursion)**

void Funct (int a)

{

if (a < 1) return; // base case

cout << a;

// recursive call

Funct(a/2);

}

In this case, the recursive call is the **last operation** performed, which makes it **tail recursion**, potentially more efficient due to compiler optimizations.

**Sample Code (Non-Tail Recursion)**

void Funct (int a)

{

if (a < 1) return; // base case

// recursive call

return Funct(a/2);

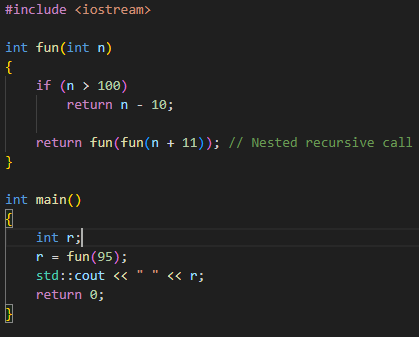
}

Here, the recursive call is not the final action — the function waits for the result, making it **non-tail recursion**.

# **Nested Recursions**

Nested recursion occurs when a function's **parameter itself is a recursive call**, meaning recursion happens **inside another recursive call**. This type of recursion is more complex and harder to trace compared to direct or tail recursion.

**Sample Code**



# **Backtracking**

Backtracking is a recursive problem-solving technique that tries out all possible solutions and **"backs up"** as soon as it determines that a path won't lead to a valid solution. It is particularly useful in constraint-based problems like mazes, puzzles, and the N-Queens problem.

**Sample Pseudocode**

void findSolutions(n, other params) {

if (found a solution) {

solutionsFound = solutionsFound + 1;

displaySolution()

}

if (solutionsFound >= solutionTarget) {

System.exit(0);

return

}

for (val = first to last) {

if (isValid(val, n)) {

applyValue(val, n);

findSolutions(n+1, other params);

removeValue(val, n);

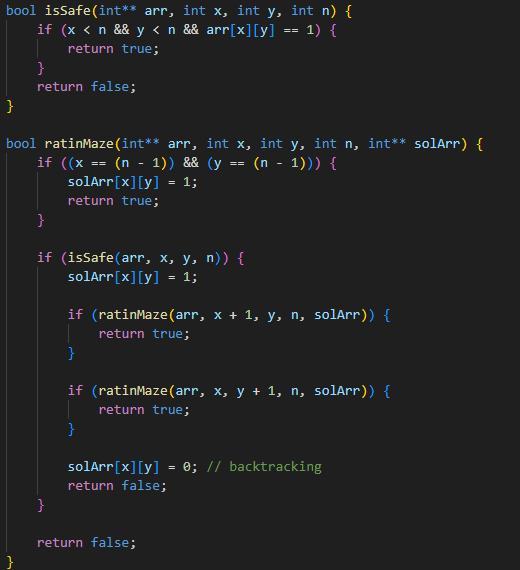
}

}

This pseudocode outlines the core structure of a backtracking algorithm: check for solution, try a possible value, explore deeper recursively, and undo the move if it fails.

**Maze (Rat in a Maze) - Code Example**

This simplified version of the maze problem demonstrates how backtracking helps explore all possible paths from the source to destination and stops once a valid path is found by avoiding obstacles.



**A Maze is given as N\*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and must reach the destination. The rat can move only in two directions: forward and down.**

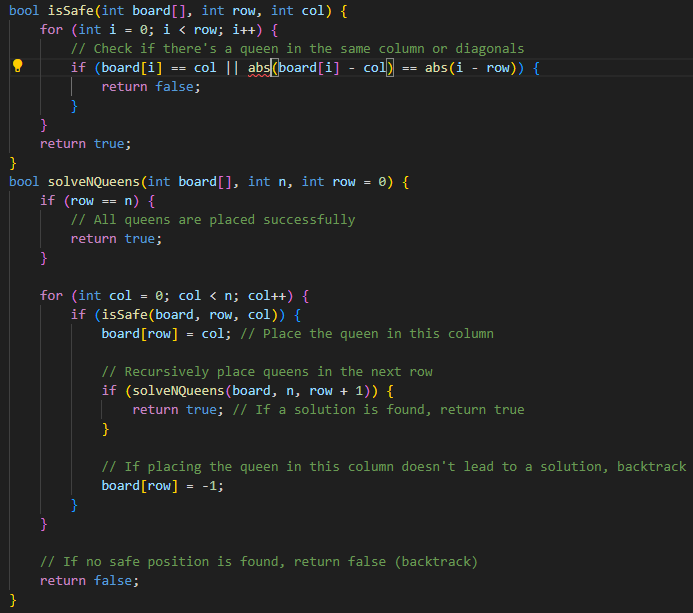
**A grid with green lines and arrows

AI-generated content may be incorrect.**

**In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions, and a more complex version can be with a limited number of moves.**

**N Queen Problem - Code Example**

The N-Queens problem is a classic example of backtracking where the goal is to place queens on a chessboard without threatening each other. The backtracking algorithm places queens row-by-row, and backtracks if it encounters a conflict.



# **LAB TASKS**