CL2001 **Data Structures Lab** [Subject]	**Lab 5** **Recursion and Backtracking**

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

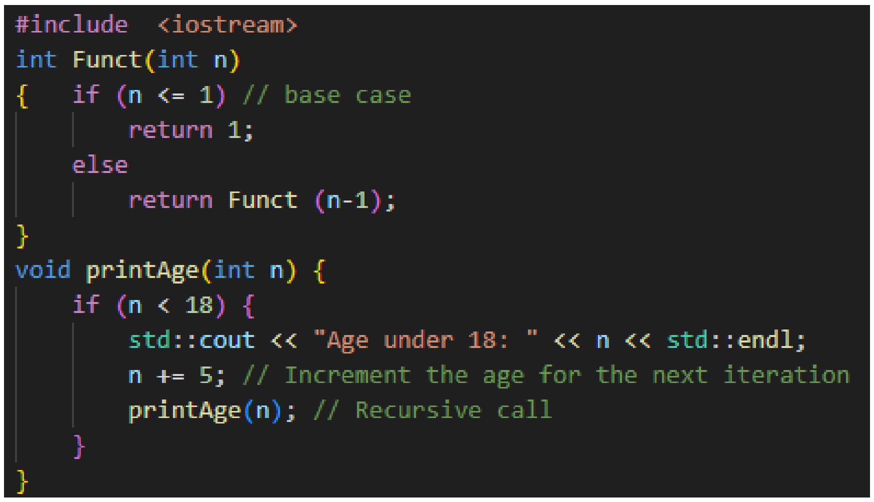
Fall 2025

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:



```
#include  <iostream>
int Funct(int n)
{   if (n <= 1) // base case
        return 1;
    else
        return Funct (n-1);
}
void printAge(int n) {
    if (n < 18) {
        std::cout << "Age under 18: " << n << std::endl;
        n += 5; // Increment the age for the next iteration
        printAge(n); // Recursive call
    }
}
```

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

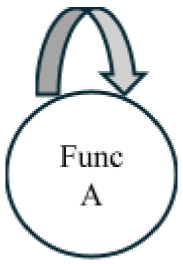
Direct and Indirect Recursions

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.

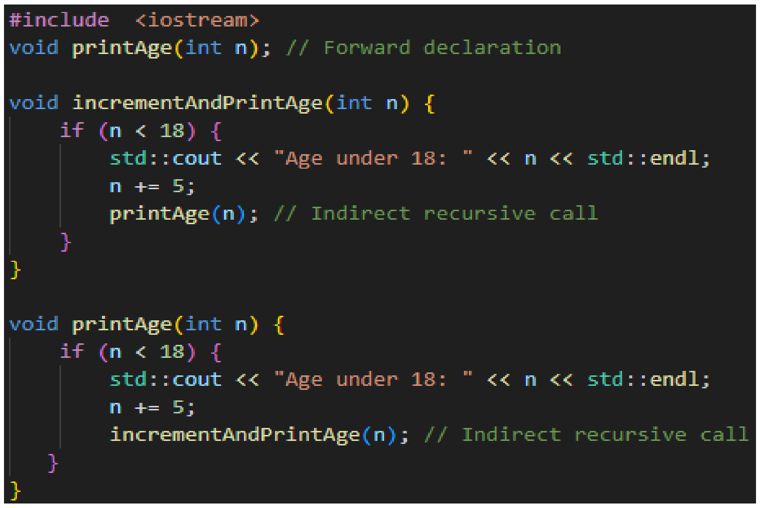
Sample Code (Direct Recursion)

```
void X()
{
   // Some code....
   X();
   // Some code...
}
```

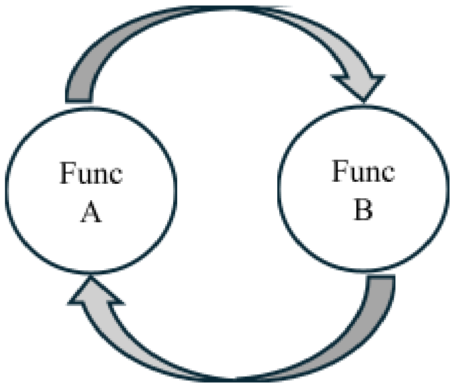
Here, X() calls itself directly within its own body. This is a classic example of direct recursion.



Sample Code (Indirect Recursion)



```
#include  <iostream>
void printAge(int n); // Forward declaration

void incrementAndPrintAge(int n) {
    if (n < 18) {
        std::cout << "Age under 18: " << n << std::endl;
        n += 5;
        printAge(n); // Indirect recursive call
    }
}

void printAge(int n) {
    if (n < 18) {
        std::cout << "Age under 18: " << n << std::endl;
        n += 5;
        incrementAndPrintAge(n); // Indirect recursive call
    }
}
```



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.

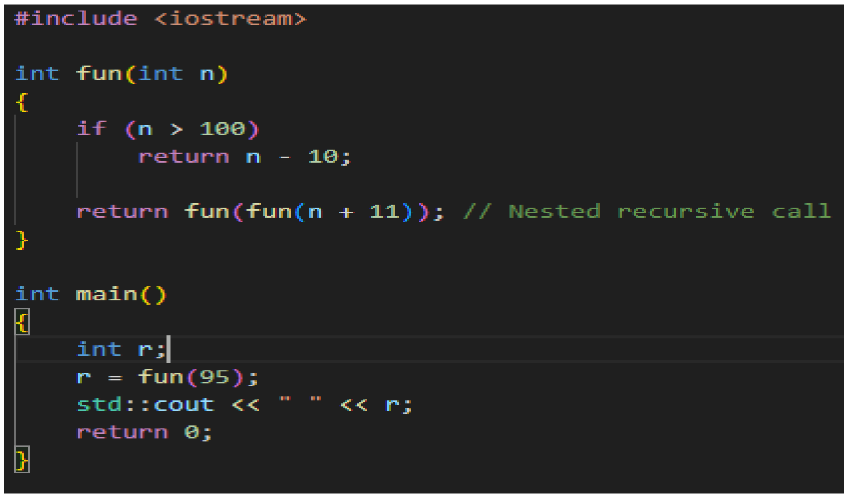
Sample Code (Non-Tail Recursion)

```
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



```
#include <iostream>

int fun(int n)
{
    if (n > 100)
        return n - 10;

    return fun(fun(n + 11)); // Nested recursive call
}

int main()
{
    int r;
    r = fun(95);
    std::cout << " " << r;
    return 0;
}
```

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.

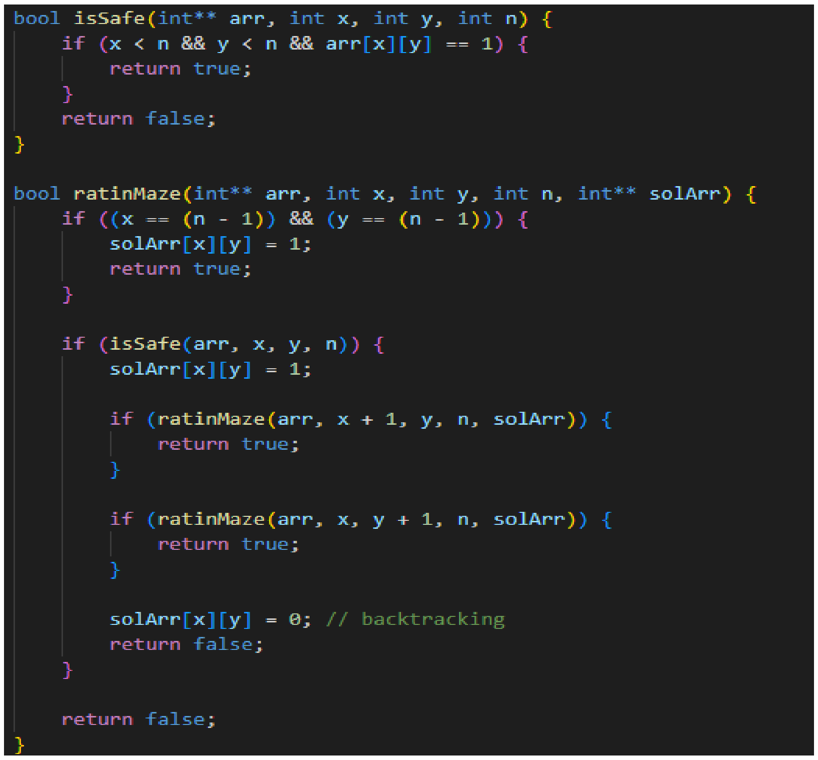
Sample Pseudocode

```
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.

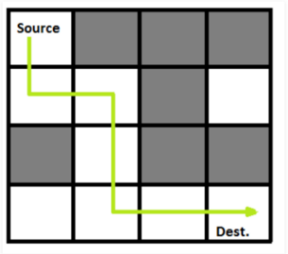
Maze (Rat in a Maze) - Code Example

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.



```cpp
bool isSafe(int** arr, int x, int y, int n) {
    if (x < n && y < n && arr[x][y] == 1) {
        return true;
    }
    return false;
}

bool ratinMaze(int** arr, int x, int y, int n, int** solArr) {
    if ((x == (n - 1)) && (y == (n - 1))) {
        solArr[x][y] = 1;
        return true;
    }

    if (isSafe(arr, x, y, n)) {
        solArr[x][y] = 1;

        if (ratinMaze(arr, x + 1, y, n, solArr)) {
            return true;
        }

        if (ratinMaze(arr, x, y + 1, n, solArr)) {
            return true;
        }

        solArr[x][y] = 0; // backtracking
        return false;
    }

    return false;
}
```

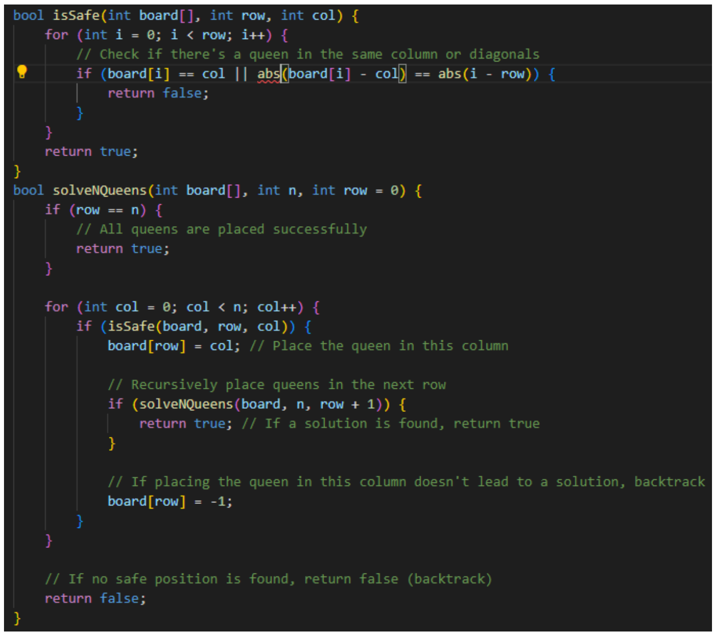
A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and must reach the destination. The rat can move only in two directions: forward and down.



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.



```cpp
bool isSafe(int board[], int row, int col) {
    for (int i = 0; i < row; i++) {
        // Check if there's a queen in the same column or diagonals
        if (board[i] == col || abs(board[i] - col) == abs(i - row)) {
            return false;
        }
    }
    return true;
}
bool solveNQueens(int board[], int n, int row = 0) {
    if (row == n) {
        // All queens are placed successfully
        return true;
    }

    for (int col = 0; col < n; col++) {
        if (isSafe(board, row, col)) {
            board[row] = col; // Place the queen in this column

            // Recursively place queens in the next row
            if (solveNQueens(board, n, row + 1)) {
                return true; // If a solution is found, return true
            }

            // If placing the queen in this column doesn't lead to a solution, backtrack
            board[row] = -1;
        }
    }

    // If no safe position is found, return false (backtrack)
    return false;
}
```

LAB TASKS

Task 1:

Write a recursive C++ function `calculateFactorial(int n)` that computes the factorial of a given integer `n` . The function should have a base condition to stop the recursion when `n` is 0 or 1, and should call itself with a smaller value of `n` (e.g., `n-1`) to solve the problem.

Task 2:

1. Write a simple C++ program that demonstrates direct recursion. Create a function `printNumbers(int n)` that prints numbers from `n` down to 1 by calling itself directly.
2. Next, demonstrate indirect recursion by creating two functions, `functionA(int n)` and `functionB(int n)`. `functionA` should call `functionB`, and `functionB` should call `functionA`, forming a cycle that prints numbers in a specific pattern. Ensure both programs have a base condition to prevent infinite loops.

Task 3:

1. Implement a C++ function `sumTail(int n, int total)` that calculates the sum of numbers from 1 to `n` using tail recursion. The recursive call should be the last operation in the function.
2. Implement another function `sumNonTail(int n)` that calculates the same sum using non-tail recursion. This function should perform an operation (e.g., addition) after the recursive call returns. Compare the two implementations and explain the difference in their call stacks.

Task 4:

Write a C++ program that implements the Ackermann function, which is a classic example of nested recursion. The function `ackermann(int m, int n)` is defined as:

- If m=0, return n+1.
- If m>0 and n=0, return `ackermann(m-1, 1)`.
- If m>0 and n>0, return `ackermann(m-1, ackermann(m, n-1))`.

This task requires you to trace the execution and understand how one recursive call is a parameter to another.

Task 5:

Implement a Sudoku solver using the backtracking technique. The program should take a partially filled 9x9 Sudoku grid and fill the empty cells (represented by 0) to solve the puzzle. Your algorithm should:

1. Find an empty cell.
2. Try placing a number from 1 to 9 in the empty cell.
3. Check if the number is valid in the current row, column, and 3x3 subgrid.
4. If valid, recursively call the function to solve the rest of the puzzle.
5. If the recursive call doesn't lead to a solution,
 backtrack by resetting the cell to 0 and trying the next number.

Task 6:

Modify the provided "Rat in a Maze" code to handle a more complex version of the problem. The rat should now be able to move in four directions (up, down, left, and right) instead of just two. Your updated code should still use backtracking to find a valid path from the source (0,0) to the destination (N-1, N-1) while avoiding dead ends.

Task 7:

Create a program that solves the N-Queens problem for any given N using backtracking. The program should print all possible solutions for placing N queens on an N x N chessboard so that no two queens threaten each other. This task requires you to use a recursive function to place queens row by row , and a validation function to check if a queen can be placed in a specific cell. If a placement leads to a conflict, the algorithm should backtrack to the previous row and try a different column.