Data Structure Algorithm & Application (CT-159) Lab 04

Recursion, Backtracking, Stack

Objectives

The objective of the lab is to get students familiar with recursion and stack, and their usage in backtracking Algorithm.

Tools Required

Dev C++ IDE

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Recursion - Introduction

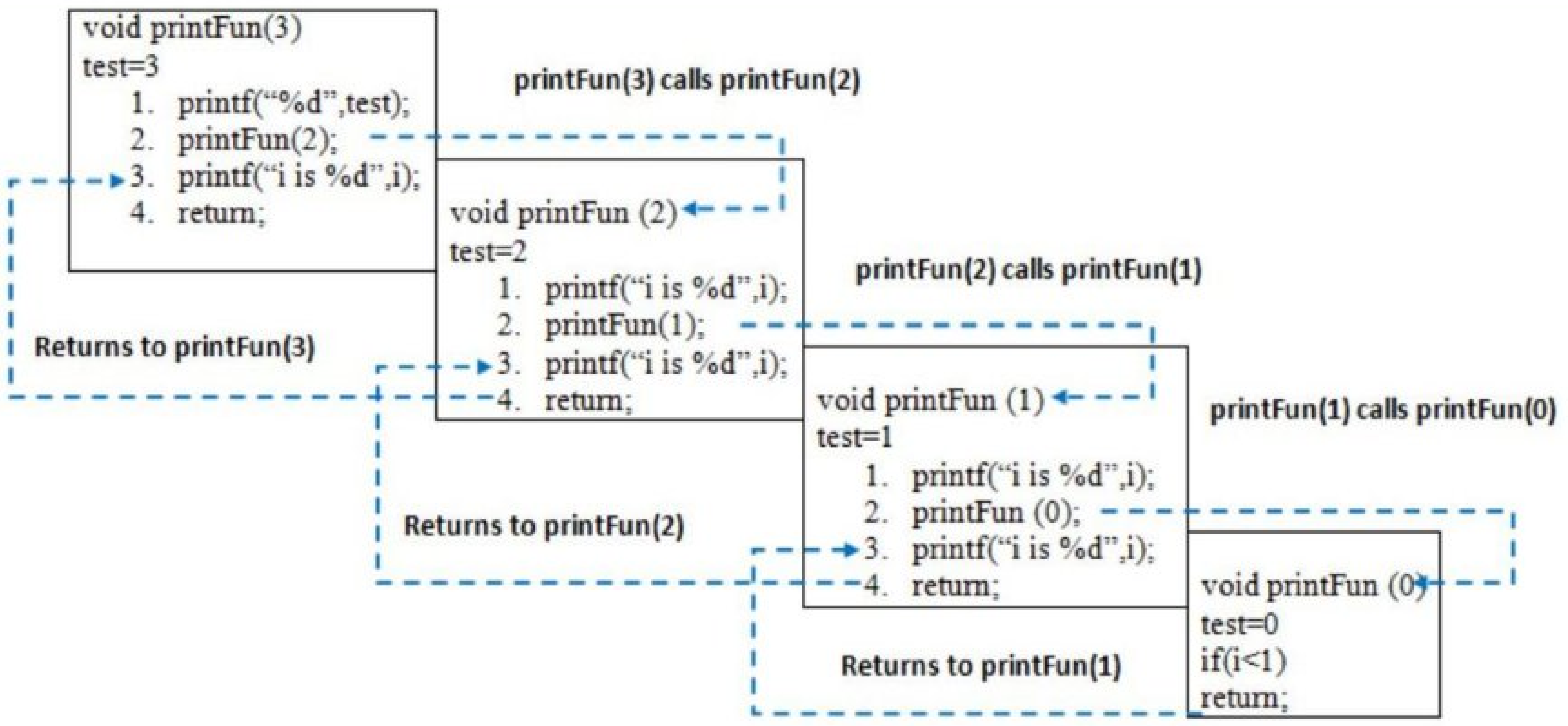
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc. A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.

Properties of Recursion:

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion otherwise infinite loop will occur.

Example 01:

```
// A C++ program to demonstrate working of recursion
#include <iostream>
using namespace std;

void printFun(int test){
        if (test < 1)
                return;
        else {
                cout << test << " ";
                printFun(test - 1); // statement 2
                cout << test << " ";
                return;
        }
}
// Driver Code
int main(){
        int test = 3;
        printFun(test);
}
```



Tail and Non-Tail Recursion

Tail recursion is a specific type of recursion where the recursive call is the last operation in the function. This means that after the recursive call, the function does nothing else; it returns the result of the recursive call directly.

Example 02: Find the greatest common divisor (GCD) using tail recursion.

```
#include <iostream>
using namespace std;
// Tail-recursive GCD function
int gcdTailRecursive(int a, int b) {
  if (b == 0) {
        return a;  // Base case
    } else {
        return gcdTailRecursive(b, a % b);  // Tail-recursive call
    }
}
int main() {
    int a = 56;
    int b = 98;
    int result = gcdTailRecursive(a, b);
    cout << "GCD of " << a << " and " << b << " is: " << result << endl;
    return 0;
}
```

Non-tail recursion refers to a recursive function where the recursive call is **not** the last operation performed in the function. In other words, after the recursive call, the function still has to perform some additional computations or operations before returning a result. Because the recursive call is not the final action, the function cannot immediately return the result of the recursive call. Instead, it has to keep the current function's state on the stack until all recursive calls complete. This often leads to increased memory usage and can potentially result in stack overflow for deep recursion.

Example 03: Find Fibonacci series till term n.

```
int fibonacci(int n) {
    if (n <= 1) return n; // Base case
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}
```

```
                             fibonacci(4)
                            /            \
                   fibonacci(3)          fibonacci(2)
                   /          \            /      \
        fibonacci(2)    fibonacci(1)  fibonacci(1)  fibonacci(0)
           /  \
fibonacci(1) fibonacci(0)
```

Backtracking – Introduction

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ensure that such a return is possible and that all paths can be tried. This technique is called *backtracking*, and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem. This technique is used in artificial intelligence, and one of the problems in which backtracking is very useful is the eight queens problem.

How many Subsets are possible for an Array of size 'N' ?

Number of Subsets of an array of size N = 2^N

Proof: For each element of the array we have 2 choices:

- **Choice 1:** Include it into the subset.
- **Choice 2:** Exclude it from the subset.

Since, each element has 2 choice to contribute into the subset and we have total N elements, therefore total subsets = 2^N. Backtracking algorithm can allow us to explore all possible choices one by one recursively. It uses non-tail recursion.

Backtracking Algorithm:

1. **Initialize**: Start with an empty subset and an empty list to store all subsets.
2. **Recursive Function**: Define a recursive function that takes the current index and the current subset as parameters.
    - **Base Case**: If the current index equals the size of the input set, add the current subset to the list of all subsets.
    - **Recursive Case**:

- **Include the element**: Add the current element to the subset.
- **Explore the rest of the elements:** by recursively calling the function for the next index.
- **Exclude the element (Backtrack)**: After exploring all possibilities for the current element, backtrack by removing the last added element from the current subset to explore other combinations.

Example 04: Find all possible subsets of a set with N elements.

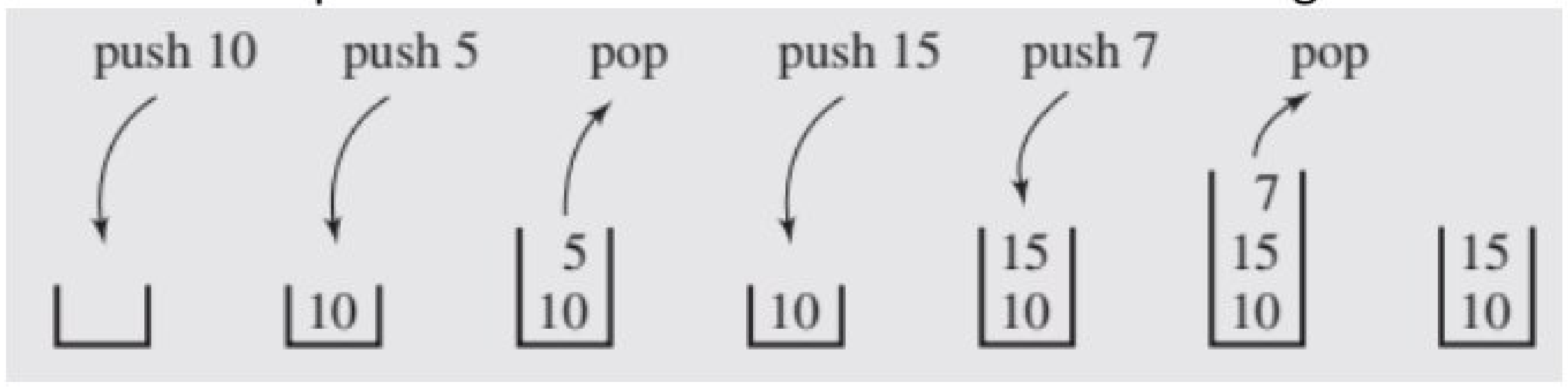
```
#include <iostream>
#include <vector>
using namespace std;

void calcSubset(vector<int>& A, vector<vector<int> >& res, vector<int>& subset, int index){
  // Add the current subset to the result list
  res.push_back(subset);
  // Generate subsets by recursively including and excluding elements
  for (int i = index; i < A.size(); i++) {
    // Include the current element in the subset
    subset.push_back(A[i]);

    // Recursively generate subsets with the current element included
    calcSubset(A, res, subset, i + 1);

    // Exclude the current element from the subset (backtracking)
    subset.pop_back();
  }
}

vector<vector<int> > subsets(vector<int>& A) {
  vector<int> subset;
  vector<vector<int> > res;
  int index = 0;
  calcSubset(A, res, subset, index);
  return res;
}

// Driver code
int main(){
  vector<int> array = { 1, 2, 3 };
  vector<vector<int> > res = subsets(array);

  // Print the generated subsets
  for (int i = 0; i < res.size(); i++) {
    for (int j = 0; j < res[i].size(); j++)
      cout << res[i][j] << " ";
    cout << endl;
  }

  return 0;
}
```

Stack - Introduction

A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data. Such a stack resembles a stack of trays in a cafeteria: new trays are put on the top of the stack and taken off the top. The last tray put on the stack is the first tray removed from the stack. For this reason, a stack is called an LIFO structure: last in/first out.

A stack is defined in terms of operations that change its status and operations that check this status. The operations are as follows:

- ✓ clear()—Clear the stack.
- ✓ isEmpty()—Check to see if the stack is empty.
- ✓ push(el)—Put the element el on the top of the stack.
- ✓ pop()—Take the topmost element from the stack.
- ✓ topEl()—Return the topmost element in the stack without removing it.



Example 03: Implementation of Stack using array.

```
#include <iostream>
using namespace std;
class Stack {
private:
   int* arr;       // Pointer to the array that holds the stack elements
   int capacity;    // Maximum size of the stack
   int topIndex;     // Index of the top element in the stack
public:
   // Constructor to initialize the stack with a given capacity
   Stack(int size) {
      capacity = size;
      arr = new int[capacity];
      topIndex = -1;  // Indicates an empty stack
   }
   // Destructor to clean up the allocated memory
   ~Stack() {
      delete[] arr;
   }

   // Push an element onto the stack
   void push(int element) {
      if (topIndex == capacity - 1) {
         cout << "Stack overflow. Unable to push " << element << endl;
         return;
      }
      arr[++topIndex] = element;
   }

   // Pop an element from the stack
   int pop() {
      if (topIndex == -1) {
         cout << "Stack underflow. Unable to pop." << endl;
         return -1;  // Returning -1 to indicate an error (assuming only positive numbers are used)
      }
      return arr[topIndex--];
   }
   // Peek at the top element of the stack
   int peek() const {
      if (topIndex == -1) {
         cout << "Stack is empty." << endl;
         return -1;  // Returning -1 to indicate an error
      }
      return arr[topIndex];
   }
```

```
    // Check if the stack is empty
    bool isEmpty() const {
        return topIndex == -1;
    }
    // Get the current size of the stack
    int size() const {
        return topIndex + 1;
    }
};
int main() {
    Stack stack(5);  // Create a stack with a capacity of 5
    stack.push(10);
    stack.push(20);
    stack.push(30);
    cout << "Top element is: " << stack.peek() << endl;
    cout << "Stack size is: " << stack.size() << endl;
    stack.pop();
    cout << "Top element after pop is: " << stack.peek() << endl;
    cout << "Stack size after pop is: " << stack.size() << endl;
    return 0;
}
```

Applications

1. Delimiter Matching

One application of the stack is in matching delimiters in a program. This is an important example because delimiter matching is part of any compiler: No program is considered correct if the delimiters are mismatched. In C++ programs, we have the following delimiters: parentheses "(" and ")", square brackets "[" and "]", curly brackets "{" and "}", and comment delimiters "/*" and " */".

The delimiter matching algorithm reads a character from a C++ program and stores it on a stack if it is an opening delimiter. If a closing delimiter is found, the delimiter is compared to a delimiter popped off the stack. If they match, processing continues; if not, processing discontinues by signaling an error. The processing of the C++ program ends successfully after the end of the program is reached and the stack is empty.

Pseudocode

```
function isValidDelimiterString(s):
  stack = empty stack
  for each character in s:
    if character is one of '(', '{', '[':
      push character onto stack
    else if character is one of ')', '}', ']':
      if stack is empty:
        return False
      top = pop from stack
      if character does not match the type of top:
        return False
  return stack is empty
```

2. Postfix Expression

A postfix expression is an expression in which each operator follows its operands. The advantage of this form is that there is no need to group sub-expressions in parentheses or to consider operator precedence. Stack can be used to evaluate a postfix (or prefix) expression.

Pseudocode

```
function evaluatePostfix(expression):
  stack = empty stack

  for each token in expression:
    if token is an operand:
      Push token onto the stack
    else if token is an operator:
      Operand2 = Pop from stack
      Operand1 = Pop from stack
      Result = Apply operator on Operand1 and Operand2
      Push Result onto the stack

  // After processing all tokens, the result is the only element left in the stack
  return Pop from stack
```

3. Tower of Hanoi

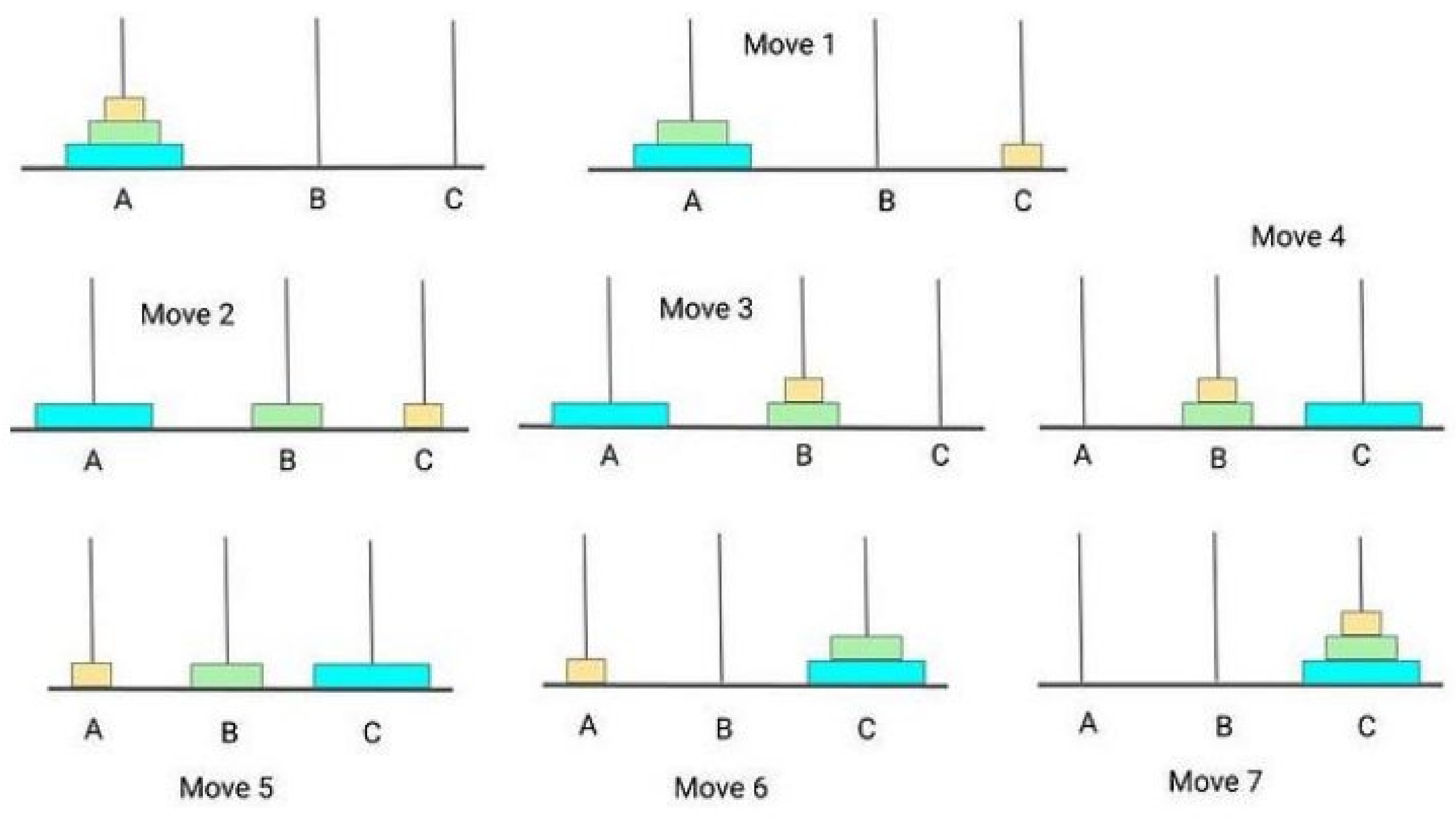
The **Tower of Hanoi** is a classic problem in computer science and mathematics that involves moving a set of disks from one peg (or rod) to another, following specific rules. It is an excellent example for teaching the concept of recursion and backtracking due to its simple rules and recursive solution.

Problem Statement

You have three rods (usually named A, B, and C) and a number of disks of different sizes. Initially, all disks are stacked in increasing size on one rod, with the largest disk at the bottom and the smallest disk at the top.
The goal is to move the entire stack of disks from the initial rod (let's say rod A) to another rod (let's say rod C), using a third rod (rod B) as an auxiliary.

Rules

I. Only one disk can be moved at a time.
II. A disk can only be placed on top of a larger disk or on an empty rod.
III. All disks must be moved from the source rod to the destination rod.



Example: Consider 3 disks labelled 1, 2, and 3 in increasing order of size, with 1 being the smallest and 3 being the largest:

- **Initial State**: All disks are on rod A.
- **Goal**: Move all disks to rod C, using rod B as an auxiliary rod.

Recursive Algorithm for Tower of Hanoi

The problem can be solved recursively by breaking it down into smaller subproblems:

I. **Base Case**: If there is only one disk, move it directly from the source rod ‘A’ to the destination rod ‘C’.
II. **Recursive Case**: If there are n disks:

- **Move `n-1` disks** from rod `A` to rod `B` using rod `C` as an auxiliary.
- **Move the nth disk** (largest disk) from rod `A` to rod `C`.
- **Move `n-1` disks** from rod `B` to rod `C` using rod `A` as an auxiliary.

```
void towerOfHanoi(int n, char sourceRod, char destinationRod, char auxiliaryRod) {
  if (n == 1) {
    // Base case: only one disk, move it from source to destination
    cout << "Move disk 1 from " << sourceRod << " to " << destinationRod << endl;
    return;
  }

  // Step 1: Move n-1 disks from source to auxiliary rod using destination as auxiliary
  towerOfHanoi(n - 1, sourceRod, auxiliaryRod, destinationRod);

  // Step 2: Move the nth disk from source to destination
  cout << "Move disk " << n << " from " << sourceRod << " to " << destinationRod <<endl;

  // Step 3: Move n-1 disks from auxiliary to destination rod using source as auxiliary
  towerOfHanoi(n - 1, auxiliaryRod, destinationRod, sourceRod);
}
```

4. Finding a path in a Maze

Maze solving using backtracking is another classic problem where we determine if there is a path from a starting point to an end point in a maze. The maze is typically represented by a 2D grid where some cells are open and others are blocked.

Problem Statement

Given a maze represented as a 2D grid of cells, where:

- 1 represents an open path.
- 0 represents a blocked path.

The task is to determine if there is a path from a starting cell (usually the top-left corner (0, 0)) to an ending cell (usually the bottom-right corner (n-1, m-1)) using only valid moves (typically up, down, left, or right).

Algorithm for Maze Solving Using Backtracking

The maze-solving problem can be solved using a backtracking approach. The basic idea is to start from the starting cell and explore all possible paths using depth-first search (DFS). If a path leads to the end, the maze is solvable; otherwise, we backtrack and try a different path.

Algorithm Steps

I. **Base Case**:
   - If the current cell is the destination (end point), return true.
   - If the current cell is blocked or out of bounds, return false.

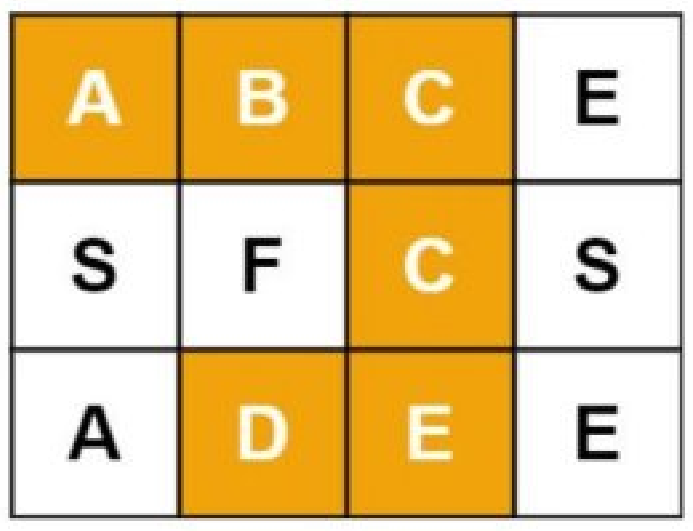
II. **Recursive Case**:
   - Mark the current cell as part of the path (visited).
   - Recursively try moving in all four possible directions (up, down, left, right).
   - If moving in any direction returns true, the path is found.
   - If all directions return false, backtrack by unmarking the current cell and return false.

Pseudocode

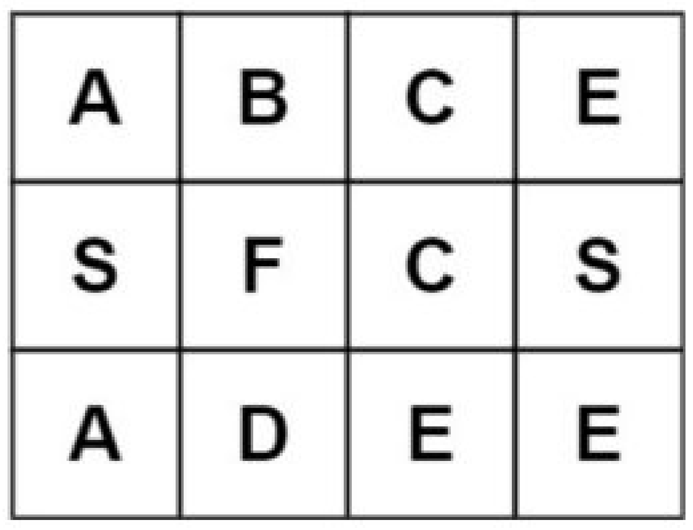
```
function solveMaze(maze, x, y, solution):
  if (x, y) is the destination:
    mark solution[x][y] as part of the path
    return True

  if (x, y) is a valid cell and not visited:
    mark solution[x][y] as part of the path
    // Move right
    if solveMaze(maze, x, y + 1, solution):
      return True

    // Move down
    if solveMaze(maze, x + 1, y, solution):
      return True

    // Move left
    if solveMaze(maze, x, y - 1, solution):
      return True

    // Move up
    if solveMaze(maze, x - 1, y, solution):
      return True

    // Backtrack: unmark (x, y) as part of the path
    unmark solution[x][y]
    return False

  return False
```

Exercise

1. A palindrome is a word, phrase, number, or another sequence of characters that reads the same backward and forwards. Can you determine if a given string, s, is a palindrome? Write a Program using stack for checking whether a string is palindrome or not.
2. Given two strings s and t, return true if they are equal when both are typed into empty text editors. '#' means a backspace character. Note that after backspacing an empty text, the text will continue empty.
   Example 1: Input: s = "ab#c", t = "ad#c", Output: true, Explanation: Both s and t become "ac".
   Example 2: Input: s = "a#c", t = "b", Output: false, Explanation: s becomes "c" while t becomes "b".
3. Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.
   Example 1: Input: nums = [1,2,3], Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
   Example 2: Input: nums = [0,1], Output: [[0,1],[1,0]]
4. Given an m x n grid of characters board and a string word, return true *if* word *exists in the grid*. The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.
   Example 01: Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED", **Output:** true



   Example 02: Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB", **Output:** false



Lab 01 Evaluation		
Student Name:	**Student ID:**	**Date:**
Rubric	**Marks (25)**	**Remarks by teacher in accordance with the rubrics**
R1		
R2		
R3		
R4		
R5		