NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
COURSE: DSA
COURSE CODE: CT-159
ASSIGNMENT PREPARED BY:
TAMIA NAEEM
ROLL NO. AI-004
SUBMITTED TO:
MISS NASR KAMAL

Exercise

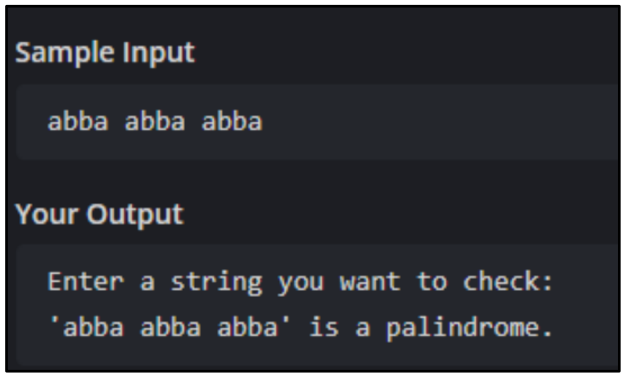
- A palindrome is a word, phrase, number, or another sequence of characters that reads the same backward and forwards. Can you determine if a given string, s, is a palindrome? Write a Program using stack for checking whether a string is palindrome or not.

SOURCE CODE

```
#include<iostream>
#include<stack>
#include<string>
using namespace std;
bool Palindrome(const string&sentence) {
        stack<char> Stack;
        int l= sentence.length();
        for(int i = 0; i < l; i++) {
                Stack.push(sentence[i]);
        }
        for(int i = 0; i < l; i++) {
                if(sentence[i]!= Stack.top()) {
                        return false;
                }
          Stack.pop();
    }
    return true;
}
int main(){
        string sentence;
```

```
        cout<< "Enter a string you want to check:" << endl;
        getline(cin, sentence);
        if(Palindrome(sentence)) {
                cout << "'" << sentence << "'" << " is a palindrome." << endl;
        }
    else{
        cout << "'" << sentence << "'" << " is not a palindrome." << endl;
        }
return 0;
}
```

OUTPUT



- 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".

SOURCE CODE

```
#include<iostream>
#include<stack>
#include<string>
using namespace std;

string compare (const string&sen) {
    string temp = "";
        int l= sen.length();
        for(int i = 0; i < l; i++) {
                char c = sen[i];
                if(c == '#'){
                    if(!temp.empty()){
                        temp.pop_back();
                    }
                }
```

```
                else temp.push_back(c);
            }
            return temp;
        }
        bool CompareAndEqual (const string&sen, const string&ten){
            return compare(sen) == compare(ten);
        }

        int main() {
            string s;
            string t;
            cout << "Enter a    word containing #: " << endl;
            cin >> s;
            cout<< "Enter another word containing #: " << endl;
            cin >> t;

            string removingS = compare(s);
            string removingT = compare(t);

            cout << "Processed first word: " << removingS << endl;
            cout << "Processed second word: " << removingT << endl;


            if(CompareAndEqual(s,t)) {
                cout << "The words are equal." << endl;
            }
            else {
                cout << "The words are not equal." << endl;
            }
                return 0;
        }
```

OUTPUT

Sample Input

```
acb#a
acd#a
```

Your Output

```
Enter a  word containing #:
Enter another word containing #:
Processed first word: aca
Processed second word: aca
The words are equal.
```

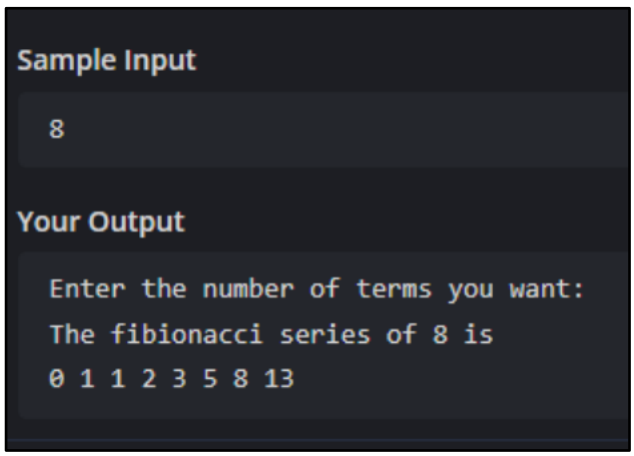
- Write a C++ program that generates the Fibonacci series up to a specified number of terms using a recursive function.

    - Background: The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. The series goes: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc.

SOURCE CODE

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

int fibionacci(int n){
    if(n <= 1) return n;
    else return fibionacci(n-1) + fibionacci (n-2);
}
int main() {
    int num;
       cout << "Enter the number of terms you want: " << endl;
       cin >> num;
    cout << "The fibionacci series of " << num << " is " << endl;
    for(int i = 0;i < num;i++){
          cout << fibionacci(i) << " ";
    }
                return 0;
}
```

OUTPUT



- 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]]

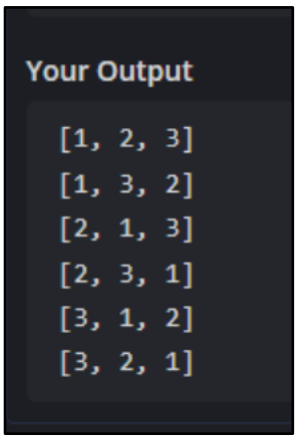
SOURCE CODE

```
#include <iostream>
```

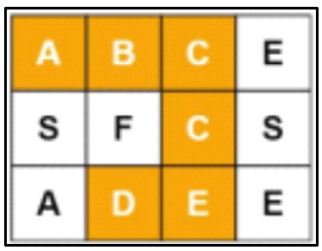
```
#include <vector>
using namespace std;

// Function to generate permutations using backtracking
void backtrack(vector<int>& arr, vector<vector<int>>& ans, vector<int>& exist,
vector<bool>& checked) {
    // If the current permutation is complete, add it to the result
    if (exist.size() == arr.size()) {
        ans.push_back(exist);
        return;
    }

    // Iterate through each element in the array
    for (int i = 0; i < arr.size(); ++i) {
        // Skip the element if it is already included in the current permutation
        if (checked[i]) continue;

        // Include the element in the current permutation
        checked[i] = true;
        exist.push_back(arr[i]);

        // Recursively generate permutations with the current element
included
        backtrack(arr, ans, exist, checked);

        // Backtrack: remove the element and mark it as not included
        checked[i] = false;
        exist.pop_back();
    }
}

// Function to initialize the permutation generation process
vector<vector<int>> permute(vector<int>& arr) {
    vector<vector<int>> ans; // To store all permutations
    vector<int> exist; // To store the current permutation
    vector<bool> checked(arr.size(), false); // To keep track of included
elements
    backtrack(arr, ans, exist, checked); // Start the backtracking process
    return ans; // Return the generated permutations
}

int main() {
    vector<int> arr = {1, 2, 3}; // Input array
    vector<vector<int>> permutations = permute(arr); // Generate
permutations

    // Print all generated permutations
```

```
        for (const auto& perm : permutations) {
            cout << "[";
            for (int i = 0; i < perm.size(); ++i) {
                cout << perm[i];
                if (i < perm.size() - 1) {
                    cout << ", ";
                }
            }
            cout << "]" << endl;
        }

        return 0;
    }
```

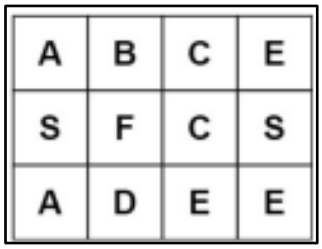
OUTPUT



- 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



SOURCE CODE

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

// Depth-First Search function to check if the word exists in the board
bool searchWord(vector<vector<char>>& board, string& word, int row, int col,
int index) {
    // If all characters of the word are matched
    if (index == word.size()) return true;

    // Check boundaries and character match
    if (row < 0 || row >= board.size() || col < 0 || col >= board[0].size() ||
board[row][col] != word[index]) return false;

    // Save the current character and mark the cell as visited
    char temp = board[row][col];
    board[row][col] = '#';

    // Explore all possible directions: down, up, right, left
bool found = searchWord(board, word, row + 1, col, index + 1) || // Move down
searchWord(board, word, row - 1, col, index + 1) || // Move up
searchWord(board, word, row, col + 1, index + 1) || // Move right
searchWord(board, word, row, col - 1, index + 1);      // Move left

    // Restore the original character in the board
    board[row][col] = temp;
    return found;
}

// Function to check if the word exists in the board
bool wordExists(vector<vector<char>>& board, string word) {
    // Iterate through each cell in the board
    for (int row = 0; row < board.size(); ++row) {
        for (int col = 0; col < board[0].size(); ++col) {
            // Start a DFS search from the current cell
            if (searchWord(board, word, row, col, 0)) return true;
        }
    }
    return false;
}

int main() {
    // Define the board
    vector<vector<char>> board = {
```

```
        {A', 'B', 'C', 'E'},
        {S', 'F', 'C', 'S'},
        {A', 'D', 'E', 'E'}
    };

    // Define the word to search for
    string word = "ABCCED";

    // Check if the word exists in the board and print the result
    if (wordExists(board, word)) {
        cout << "Word exists in the grid" << endl;
    } else {
        cout << "Word does not exist in the grid" << endl;
    }

    return 0;
}
```

OUTPUT