Day 8

find the minimum number of operation insert delete and replace from one string to another

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int minDistance(string s1, string s2) {
    int m = s1.length();
    int n = s2.length();

    // Create a 2D DP table
    vector<vector<int>> dp(m + 1, vector<int>(n + 1));

    // Initialize base cases
    for (int i = 0; i <= m; i++) {
        dp[i][0] = i; // Deleting all characters from s1
    }

    for (int j = 0; j <= n; j++) {
        dp[0][j] = j; // Inserting all characters into s1
    }

    // Fill the DP table
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1]; // No operation needed
            } else {
                dp[i][j] = 1 + min({dp[i - 1][j],    // Delete
                                    dp[i][j - 1],    // Insert
                                    dp[i - 1][j - 1]}); // Replace
            }
        }
    }

    // The result is in the bottom-right corner of the DP table
    return dp[m][n];
}

int main() {
    string s1, s2;
    cout << "Enter the first string: ";
    cin >> s1;
    cout << "Enter the second string: ";
    cin >> s2;

    cout << "Minimum number of operations required: " << minDistance(s1, s2) << endl;

    return 0;
}
```

Output

```
Enter the first string: word1
Enter the second string: word2
Minimum number of operations required: 1


=== Code Execution Successful ===
```

solve travelling salesman problem by using dp approach

```
#include <iostream>
#include <vector>
#include <climits>
#include <cmath>

using namespace std;

// Function to solve the Traveling Salesman Problem using DP (Held-Karp algorithm)
int tsp(vector<vector<int>>& dist, int n) {
    // DP table: dp[mask][i] - minimum cost to visit all cities in "mask" and end at city i
    vector<vector<int>> dp(1 << n, vector<int>(n, INT_MAX));

    // Base case: start at city 0
    dp[1][0] = 0;  // Only city 0 is visited, cost is 0

    // Iterate through all subsets of cities (represented by "mask")
    for (int mask = 1; mask < (1 << n); ++mask) {
        for (int i = 0; i < n; ++i) {
            // Skip if city i is not in the current mask
            if ((mask & (1 << i)) == 0) continue;

            // Try coming to city i from every possible city j
            for (int j = 0; j < n; ++j) {
                if (i != j && (mask & (1 << j)) != 0) {
                    dp[mask][i] = min(dp[mask][i], dp[mask ^ (1 << i)][j] + dist[j][i]);
                }
            }
```

```
        }
    }

    // The final answer is the minimum cost of visiting all cities and returning to
city 0
    int ans = INT_MAX;
    for (int i = 1; i < n; ++i) {
        ans = min(ans, dp[(1 << n) - 1][i] + dist[i][0]);
    }

    return ans;
}

int main() {
    int n;
    cout << "Enter the number of cities: ";
    cin >> n;

    vector<vector<int>> dist(n, vector<int>(n));

    cout << "Enter the distance matrix (nxn):\n";
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> dist[i][j];
        }
    }

    // Solve TSP
    int result = tsp(dist, n);

    cout << "The minimum cost of the tour is: " << result << endl;

    return 0;
}
```

```
Output
Enter the number of cities: 1
Enter the distance matrix (nxn):
3
The minimum cost of the tour is: 2147483647


=== Code Execution Successful ===
```

construct a bst with a minimum search cost

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

// Function to construct the optimal binary search tree and return the
minimum search cost
int optimalBST(const vector<int>& keys, const vector<int>& freq, int n) {
   // dp[i][j] will store the minimum cost of a BST for keys i to j
   vector<vector<int>> dp(n, vector<int>(n, 0));

   // sum[i][j] will store the sum of frequencies from keys i to j
   vector<vector<int>> sum(n, vector<int>(n, 0));

   // Initialize sum of frequencies
   for (int i = 0; i < n; ++i) {
      sum[i][i] = freq[i];
      for (int j = i + 1; j < n; ++j) {
         sum[i][j] = sum[i][j - 1] + freq[j];
      }
   }

   // Fill the dp table
   // dp[i][j] will be the minimum cost of constructing BST from keys[i] to
keys[j]
   for (int len = 1; len <= n; ++len) {
      for (int i = 0; i <= n - len; ++i) {
         int j = i + len - 1;
         if (len == 1) {
            dp[i][j] = freq[i];
```

```
            } else {
                dp[i][j] = INT_MAX;
                for (int r = i; r <= j; ++r) {
                    int leftCost = (r > i) ? dp[i][r - 1] : 0;
                    int rightCost = (r < j) ? dp[r + 1][j] : 0;
                    int cost = leftCost + rightCost + sum[i][j];
                    dp[i][j] = min(dp[i][j], cost);
                }
            }
        }
    }

    return dp[0][n - 1];  // The minimum cost for the whole range
}

int main() {
    int n;
    cout << "Enter the number of keys: ";
    cin >> n;

    vector<int> keys(n), freq(n);

    cout << "Enter the keys: ";
    for (int i = 0; i < n; ++i) {
        cin >> keys[i];
    }

    cout << "Enter the frequencies of the keys: ";
    for (int i = 0; i < n; ++i) {
        cin >> freq[i];
    }

    // Solve the problem
    int result = optimalBST(keys, freq, n);

    cout << "The minimum search cost for the optimal BST is: " << result << 
endl;

    return 0;
}
```

Output

```
Enter the number of elements in the array: 5
Enter the elements of the array: 6
```

find the maximum sum of continuous sum array by using dynamic programming

```
#include <iostream>
#include <vector>
#include <climits>  // For INT_MIN

using namespace std;

// Function to find the maximum sum of a contiguous subarray using
Kadane's Algorithm
int maxSubArraySum(const vector<int>& arr) {
   int n = arr.size();

   // Initialize variables for the current subarray sum and the maximum sum
found so far
   int current_sum = arr[0];
   int max_sum = arr[0];

   // Traverse the array starting from the second element
   for (int i = 1; i < n; ++i) {
      // Update the current sum: either start a new subarray from arr[i] or
continue adding to the current subarray
      current_sum = max(arr[i], current_sum + arr[i]);

      // Update the maximum sum found so far
      max_sum = max(max_sum, current_sum);
   }

   return max_sum;
}

int main() {
   int n;
   cout << "Enter the number of elements in the array: ";
```

```
    cin >> n;

    vector<int> arr(n);

    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    // Find and output the maximum sum of the contiguous subarray
    int result = maxSubArraySum(arr);
    cout << "The maximum sum of the contiguous subarray is: " << result <<
endl;

    return 0;
}
```

Output

```
Enter the number of elements in the array: 5
Enter the elements of the array: 6
```

find the length of palindromic subseqeuence in the string

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int longestPalindromicSubseq(const string& s) {
    int n = s.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) {
        dp[i][i] = 1;
    }

    for (int gap = 1; gap < n; ++gap) {
```

```
        for (int i = 0; i < n - gap; ++i) {

            int j = i + gap;
            if (s[i] == s[j]) {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            } else {
                dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[0][n - 1];
}
int main() {
    string s;
    cout << "Enter the string: ";
    cin >> s;
}
```

Output

```
Enter the string: state


=== Code Execution Successful ===
```

N-th Tribonacci Number

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

long long tribonacci(int n) {
    // Base cases
    if (n == 0) return 0;
    if (n == 1 || n == 2) return 1;

    long long t0 = 0, t1 = 1, t2 = 1, t3;
    for (int i = 3; i <= n; ++i) {
        t3 = t0 + t1 + t2; // Current Tribonacci number
        t0 = t1;  // Move to next
        t1 = t2;
        t2 = t3;
    }
    return t3;
}
```

```
int main() {
    int n;
    cout << "Enter the value of n: ";
    cin >> n;

    cout << "The " << n << "-th Tribonacci number is: " << tribonacci(n) << endl;
    return 0;
}
```

Output

```
Enter the value of n: 2
The 2-th Tribonacci number is: 1


=== Code Execution Successful ===
```

Maximum Repeating Substring

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

string maxRepeatingSubstring(const string& s) {
    int n = s.size();
    if (n == 0) return "";  // Empty string

    unordered_map<string, int> substringCount;
    int maxCount = 0;
    string maxSubstring = "";

    // Try all substrings of different lengths
    for (int len = 1; len <= n / 2; ++len) {
        for (int i = 0; i <= n - len; ++i) {
            string substr = s.substr(i, len);
            substringCount[substr]++;

            // Track the substring with the maximum count
            if (substringCount[substr] > maxCount) {
                maxCount = substringCount[substr];
                maxSubstring = substr;
```

```
            }
        }
    }

    return maxSubstring;
}

int main() {
    string s;
    cout << "Enter the string: ";
    cin >> s;

    string result = maxRepeatingSubstring(s);
    if (result.empty()) {
        cout << "No repeating substring found!" << endl;
    } else {
        cout << "The maximum repeating substring is: " << result << endl;
    }

    return 0;
}
```

```
Output
Enter the string: 1
No repeating substring found!


=== Code Execution Successful ===
```

Pascal's Triangle II

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

vector<int> getRow(int rowIndex) {
    vector<int> row(rowIndex + 1, 1);  // Initialize the row with all elements as
1

    // Compute the values in the row using the formula for binomial coefficients
    for (int i = 1; i <= rowIndex / 2; ++i) {
```

```
        // Calculate the element at index i based on the previous element
        row[i] = (long long)row[i - 1] * (rowIndex - i + 1) / i;
        // Since Pascal's triangle is symmetric, the element at position rowIndex -
i is the same
        row[rowIndex - i] = row[i];
    }

    return row;
}

int main() {
    int rowIndex;
    cout << "Enter the row index: ";
    cin >> rowIndex;

    vector<int> result = getRow(rowIndex);

    // Print the k-th row
    cout << "Row " << rowIndex << " of Pascal's Triangle: ";
    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Output

```
Enter the row index: 2
Row 2 of Pascal's Triangle: 1 2 1


=== Code Execution Successful ===
```

Longest Palindromic Substring

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

```cpp
// Function to expand around the center and find the longest palindrome
string expandAroundCenter(const string& s, int left, int right) {
    while (left >= 0 && right < s.size() && s[left] == s[right]) {
        left--;
        right++;
    }
    return s.substr(left + 1, right - left - 1);
}

// Function to find the longest palindromic substring
string longestPalindrome(string s) {
    if (s.empty()) return "";

    string longest = "";
    for (int i = 0; i < s.size(); ++i) {
        // Odd length palindrome
        string oddPalindrome = expandAroundCenter(s, i, i);
        // Even length palindrome
        string evenPalindrome = expandAroundCenter(s, i, i + 1);

        // Update longest palindrome
        if (oddPalindrome.size() > longest.size()) {
            longest = oddPalindrome;
        }
        if (evenPalindrome.size() > longest.size()) {
            longest = evenPalindrome;
        }
    }

    return longest;
}

int main() {
    string s;
    cout << "Enter the string: ";
    cin >> s;

    string result = longestPalindrome(s);
    cout << "Longest palindromic substring: " << result << endl;

    return 0;
}
```

Output

```
Enter the string: 2
Longest palindromic substring: 2


=== Code Execution Successful ===
```

Longest Increasing Path in a Matrix

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

class Solution {
public:
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) return 0;

        int rows = matrix.size();
        int cols = matrix[0].size();
        vector<vector<int>> memo(rows, vector<int>(cols, -1));  // Memoization
table

        int longestPath = 0;

        // Directions: up, down, left, right
        vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        // Regular DFS function
        int dfs(int row, int col) {
            if (memo[row][col] != -1) {
                return memo[row][col];  // Return already computed result
            }

            int maxLength = 1;  // The current cell itself is part of the path

            for (auto& dir : directions) {
                int newRow = row + dir.first;
```

```
            int newCol = col + dir.second;

            // Check if the new position is within bounds and the value is greater
than the current one
            if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol
< cols && matrix[newRow][newCol] > matrix[row][col]) {
                maxLength = max(maxLength, 1 + dfs(newRow, newCol));  //
Explore the next cell
            }
        }

        memo[row][col] = maxLength;  // Memoize the result
        return maxLength;
    }

    // Try starting DFS from every cell
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            longestPath = max(longestPath, dfs(i, j));  // Update the longest path
        }
    }

    return longestPath;
  }
};

int main() {
  Solution sol;
  vector<vector<int>> matrix = {
    {9, 9, 4},
    {6, 6, 8},
    {2, 1, 1}
  };

  int result = sol.longestIncreasingPath(matrix);
  cout << "Longest Increasing Path Length: " << result << endl;

  return 0;
}
```