Design and Analysis of Algorithms (CS206)

Assignment - 6

**U19CS012**

**A.) Longest palindrome subsequence problem**

**Problem Statement:**

Give an efficient algorithm to find the length of longest palindrome that is a subsequence of a given input string.

A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. E.g.: civic, racecar, and aibohphobia (fear of palindromes).

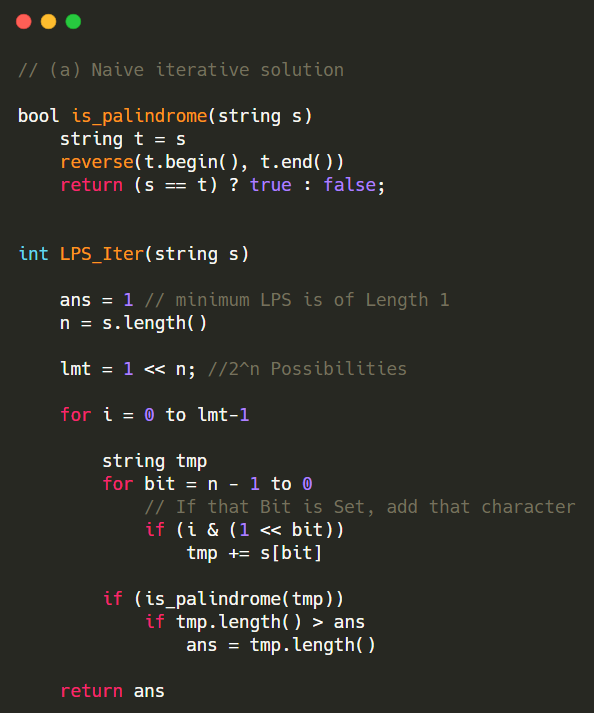
A ***subsequence*** is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements

If the given sequence is “**BBABCBCAB**”, then the output should be **7** as

“BABCBAB” is the longest palindromic subsequence in it.

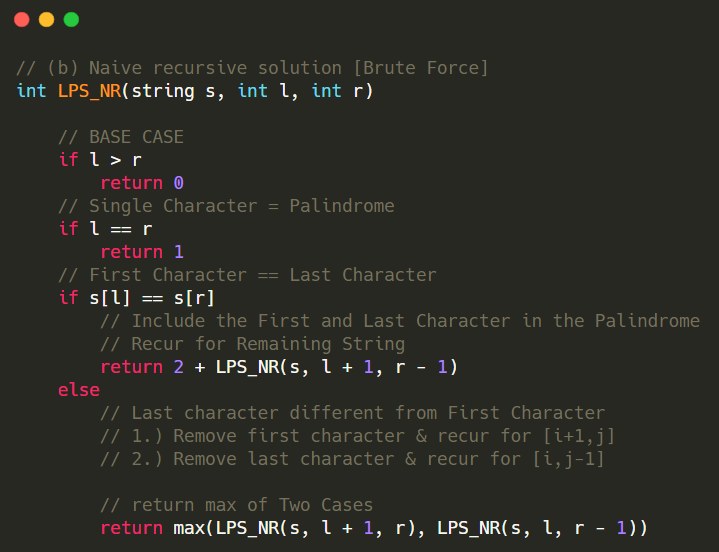
“BBBBB” and “BBCBB” are also palindromic subsequences of the given sequence, but not the longest ones.

**(a) Naive iterative solution**



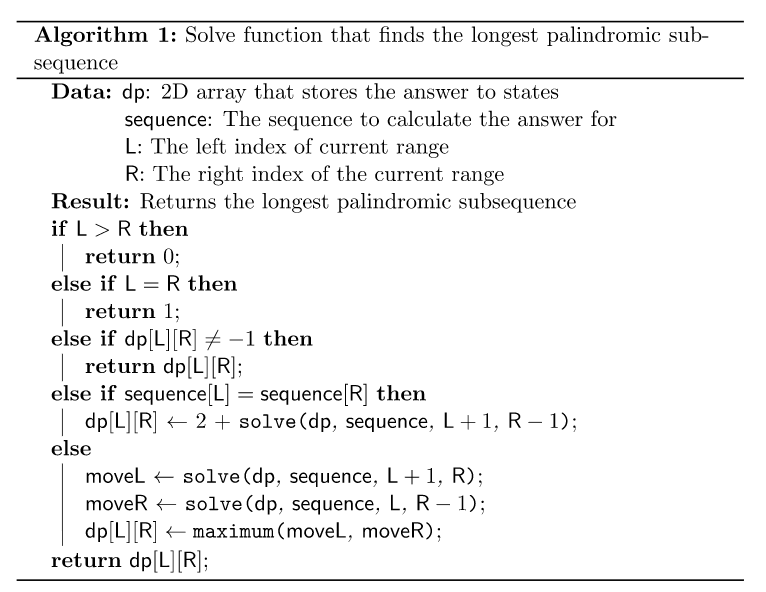
Time Complexity: O(2n)

**(b) Naive recursive solution**



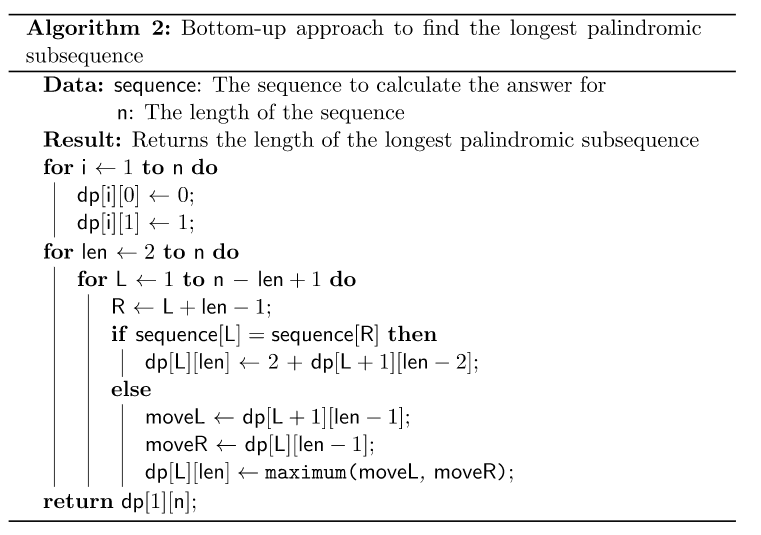
Time Complexity: O(2n)

(c) **Top-down dynamic programming** solution [Add Memory Element to Naive Rec.]



Time Complexity: O(n2)

(d) **Bottom-up dynamic programming** solution



Time Complexity: O(n2)

**Code:**

*#include* <bits/stdc++.h>

*// For Time Calculation*

*#include* <chrono>

using namespace std;

using namespace std::chrono;

*// (a) Naive iterative solution*

bool is\_palindrome(string s)

{

    string t = s;

    reverse(t.begin(), t.end());

*return* (s == t) ? true : false;

}

int LPS\_Iter(string s)

{

    int ans = 1;

    int n = s.length();

    long long int lmt = 1 << n;*//2^n Possibilities*

*for* (int i = 0; i < lmt; i++)

    {

        string tmp;

*for* (int bit = n - 1; bit >= 0; bit--)

        {

*// If that Bit is Set*

*if* (i & (1 << bit))

            {

                tmp += s[bit];

            }

        }

*if* (is\_palindrome(tmp))

        {

*if* (tmp.length() > ans)

                ans = tmp.length();

        }

    }

*return* ans;

}

*// (b) Naive recursive solution [Brute Force]*

int LPS\_NR(string s, int l, int r)

{

*// BASE CASE*

*if* (l > r)

*return* 0;

*// Single Character = Palindrome*

*if* (l == r)

*return* 1;

*// First Character == Last Character*

*if* (s[l] == s[r])

    {

*// Include the First and Last Character in the Palindrome*

*// Recur for Remaining String*

*return* 2 + LPS\_NR(s, l + 1, r - 1);

    }

*else*

    {

*// Last character different from First Character*

*// 1.) Remove first character & recur for [i+1,j]*

*// 2.) Remove last character & recur for [i,j-1]*

*// return max of Two Cases*

*return* max(LPS\_NR(s, l + 1, r), LPS\_NR(s, l, r - 1));

    }

}

*// (c) Top-down dynamic programming solution*

int LPS\_TDDP(string s, int l, int r, vector<vector<int>> &mem)

{

*// BASE CASE*

*if* (l > r)

*return* 0;

*// Single Character = Palindrome*

*if* (l == r)

*return* 1;

*// Check if Answer is Already Computed [Memoization]*

*if* (mem[l][r])

*return* mem[l][r];

*// First Character == Last Character*

*if* (s[l] == s[r])

    {

*// Include the First and Last Character in the Palindrome*

*// Recur for Remaining String*

*return* 2 + LPS\_TDDP(s, l + 1, r - 1, mem);

    }

*else*

    {

*// Last character different from First Character*

*// 1.) Remove first character & recur for [i+1,j]*

*// 2.) Remove last character & recur for [i,j-1]*

*// return max of Two Cases*

*return* max(LPS\_TDDP(s, l + 1, r, mem), LPS\_TDDP(s, l, r - 1, mem));

    }

}

*// (d) Bottom-up dynamic programming solution*

int LPS\_BUDP(string s)

{

    int n = s.length();

*// Create a table to store results of subproblems*

    int table[n][n];

    int i, j, strlen;

*// String of Length 1 = Palindromic*

*for* (int i = 0; i < n; i++)

    {

        table[i][i] = 1;

    }

*for* (strlen = 2; strlen <= n; strlen++)

    {

*for* (i = 0; i < n - strlen + 1; i++)

        {

            j = i + strlen - 1;

*// i -> Left Pointer*

*// j -> Right Pointer*

*// Same Character Palindrome of Length 2*

*if* (s[i] == s[j] && strlen == 2)

            {

                table[i][j] = 2;

            }

*else* *if* (s[i] == s[j])

            {

                table[i][j] = 2 + table[i + 1][j - 1];

            }

*else*

            {

                table[i][j] = max(table[i + 1][j], table[i][j - 1]);

            }

        }

    }

*// LPS in Range 0 to n-1*

*return* table[0][n - 1];

}

int main()

{

    string s;

    cin >> s;

    int n = s.length();

    auto start = high\_resolution\_clock::now();

    auto end = high\_resolution\_clock::now();

    auto time\_taken = duration\_cast<nanoseconds>(end - start);

    double ans = 0;

*// (a) Naive iterative solution*

    cout << "NAIVE ITERATIVE SOLUTION : " << endl;

    start = high\_resolution\_clock::now();

    cout << "Length of Longest Palindromic Subsequence : " << LPS\_Iter(s) << endl;

    end = high\_resolution\_clock::now();

    time\_taken = duration\_cast<nanoseconds>(end - start);

    ans = (double)time\_taken.count() / (double)(1e9);

    cout << "Time Taken : " << ans << " seconds." << endl;

*// (b) Naive recursive solution*

    cout << "\nNAIVE RECURSIVE SOLUTION : " << endl;

    start = high\_resolution\_clock::now();

    cout << "Length of Longest Palindromic Subsequence : " << LPS\_NR(s, 0, n - 1) << endl;

    end = high\_resolution\_clock::now();

    time\_taken = duration\_cast<nanoseconds>(end - start);

    ans = (double)time\_taken.count() / (double)(1e9);

    cout << "Time Taken : " << ans << " seconds." << endl;

*// (c) Top-down dynamic programming solution*

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

    cout << "\nTOP DOWN DP (RECURSIVE + MEMOIZATION) SOLUTION : " << endl;

    start = high\_resolution\_clock::now();

    cout << "Length of Longest Palindromic Subsequence : " << LPS\_TDDP(s, 0, n - 1, mem) << endl;

    end = high\_resolution\_clock::now();

    time\_taken = duration\_cast<nanoseconds>(end - start);

    ans = (double)time\_taken.count() / (double)(1e9);

    cout << "Time Taken : " << ans << " seconds." << endl;

*// (d) Bottom-up dynamic programming solution*

    cout << "\nBOTTOM UP DP SOLUTION : " << endl;

    start = high\_resolution\_clock::now();

    cout << "Length of Longest Palindromic Subsequence : " << LPS\_BUDP(s) << endl;

    end = high\_resolution\_clock::now();

    time\_taken = duration\_cast<nanoseconds>(end - start);

    ans = (double)time\_taken.count() / (double)(1e9);

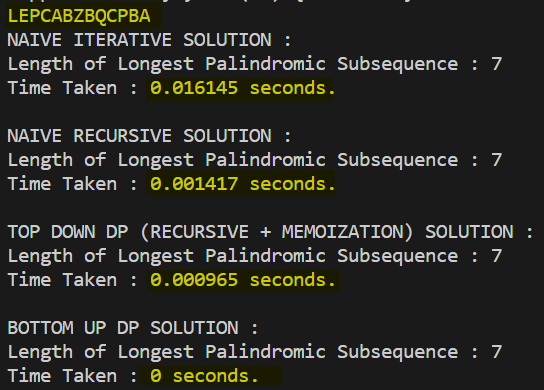
    cout << "Time Taken : " << ans << " seconds." << endl;

*return* 0;

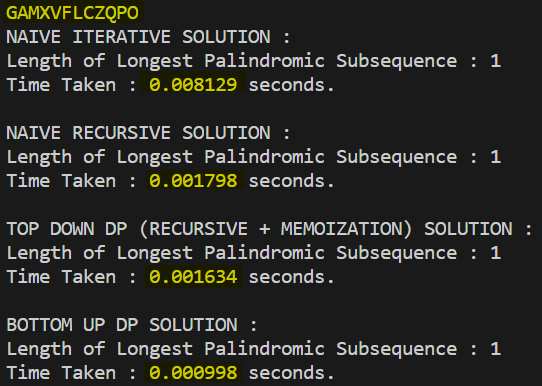
}

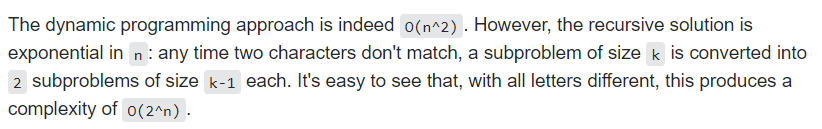
**Test Cases:**

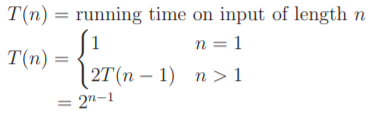
(A) LE**PC**A**BZB**Q**CP**BA [Palindrome of Length 7 {PCBZBCP}]

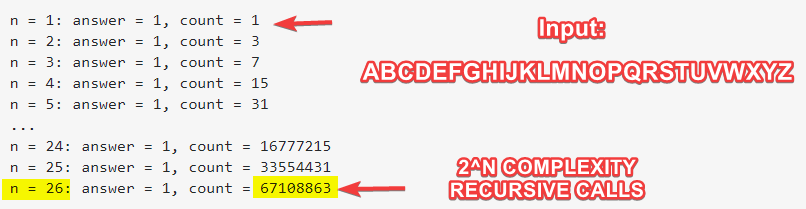


(B) GAMXVFLCZQPO [Palindrome of Length 1 {G/A/..Any Character in Str}]





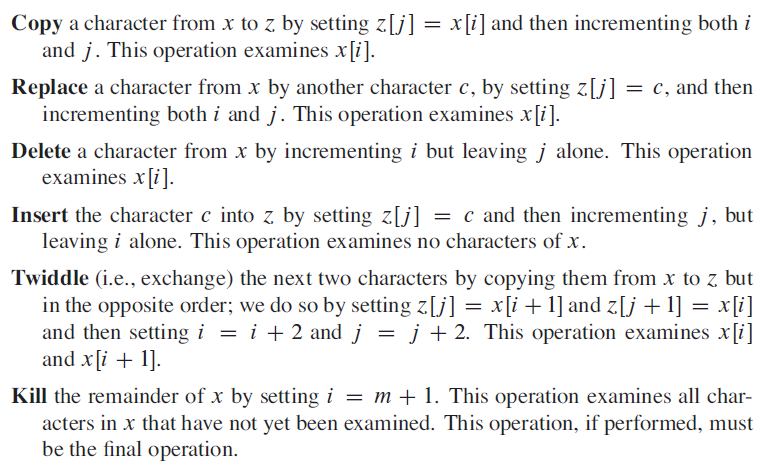




<https://ideone.com/vrLD4W> [Above Code Implementation Link]

**B.) Edit distance problem**

Convert String1 to String2 Using Any Set of 6 Operations:



(a)Naive iterative solution

**Code:**

bool check(string s1, string s2, int n, int m, string tmp)

{

    int ptr1 = 0, ptr2 = 0;

    char ch;

*for* (int i = 0; i < tmp.length() && ptr1 < n && ptr2 < m; i++)

    {

        ch = tmp[i];

*if* (ch == 'C')

        {

*if* (s1[ptr1] == s2[ptr2])

            {

                ptr1++;

                ptr2++;

            }

        }

*else* *if* (ch == 'R')

        {

            ptr1++;

            ptr2++;

        }

*else* *if* (ch == 'D')

        {

            ptr1++;

        }

*else* *if* (ch == 'I')

        {

            ptr2++;

        }

*else* *if* (ch == 'T')

        {

*if* (ptr1 <= n - 2 && ptr2 <= m - 2)

            {

*if* (s1[ptr1] == s2[ptr2 + 1] && s1[ptr2] == s2[ptr1 + 1])

                {

                    ptr1 += 2;

                    ptr2 += 2;

                }

            }

        }

*else*

        {

*if* (ptr2 == m)

            {

                ptr1 = n;

*break*;

            }

*// if (ptr1 == n)*

*// {*

*//     ptr2 = m;*

*//     break;*

*// }*

        }

    }

*if* (ptr1 == n && ptr2 == m)

*return* true;

*else*

*return* false;

}

*// Checks all Possible Operations*

int Naive(string s1, string s2)

{

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

*// Since there are 6 Operations*

    long int lmt = pow(7, m + n) - 1;

    int res = MAX\_VAL;

*// Finding 7-ary Equivalent to Decimal Number*

*for* (int i = 0; i <= lmt; i++)

    {

        string tmp;

        int nums = i;

*for* (int j = 0; j < max(m,n); j++)

        {

            tmp = to\_string(nums % 7) + tmp;

            nums /= 7;

        }

*// cout << tmp << "\n";*

        bool fl = false;

*for* (int k = 0; k < tmp.length(); k++)

        {

*if* (tmp[k] == '0')

            {

                fl = true;

*break*;

            }

        }

*// Skip Number with 0 -> No Operation*

*if* (fl != false)

*continue*;

        string tmp2;

*// 1 -> Copy, 2 -> Replace, 3 -> Delete, 4 -> Insert, 5 -> Twiddle 6 -> Kill*

*for* (int k = 0; k < tmp.length(); k++)

        {

*if* (tmp[k] == '1')

                tmp2 += 'C';

*else* *if* (tmp[k] == '2')

                tmp2 += 'R';

*else* *if* (tmp[k] == '3')

                tmp2 += 'D';

*else* *if* (tmp[k] == '4')

                tmp2 += 'I';

*else* *if* (tmp[k] == '5')

                tmp2 += 'T';

*else*

                tmp2 += 'K';

        }

        bool is\_possible = check(s1, s2, n, m, tmp2);

*if* (is\_possible)

        {

            int cost = 0;

*for* (int k = 0; k < tmp2.length(); k++)

            {

*if* (tmp2[k] == 'C')

                    cost += copy\_cost;

*else* *if* (tmp2[k] == 'R')

                    cost += rep\_cost;

*else* *if* (tmp2[k] == 'I')

                    cost += ins\_cost;

*else* *if* (tmp2[k] == 'T')

                    cost += twiddle\_cost;

*else* *if* (tmp2[k] == 'D')

                    cost += del\_cost;

*else*

                    cost += kill\_cost;

            }

*if* (cost < res)

            {

*// cout << tmp2 << endl;*

                res = cost;

            }

        }

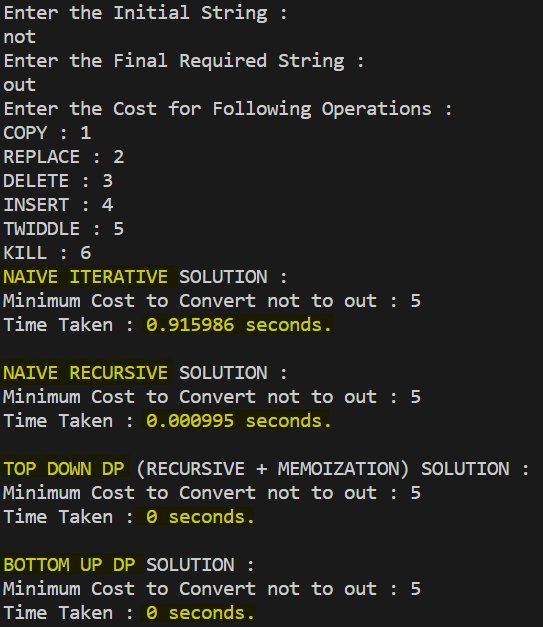
    }

*return* res;

}

Time Complexity = O(6(m+n))

Test-Case:



(b)Naive recursive solution

**Code:**

*// (b) Naive recursive solution [Brute Force]*

int Edit\_Dist\_Recur(string s1, string s2, int i, int j)

{

*if* (i == -1 && j == -1)

    {

*return* 0;

    }

*// If the First String is Empty,*

*// Only Option is to Insert all Characters in Second String*

*if* (i == -1)

    {

*return* (j + 1) \* ins\_cost;

    }

*// If the Second String is Empty,*

*// Only Option is to Remove all Characters from First String*

*if* (j == -1)

    {

*return* min(kill\_cost, (i + 1) \* del\_cost);

    }

    int ans = MAX\_VAL;

*if* (s1[i] == s2[j])

    {

        ans = min(ans, copy\_cost + Edit\_Dist\_Recur(s1, s2, i - 1, j - 1));

    }

*// Twiddle*

*if* (i >= 1 && j >= 1)

    {

*if* (s1[i] == s2[j - 1] && s1[i - 1] == s2[j])

        {

            ans = min(ans, twiddle\_cost + Edit\_Dist\_Recur(s1, s2, i - 2, j - 2));

        }

    }

    ans = min(ans, ins\_cost + Edit\_Dist\_Recur(s1, s2, i, j - 1));*// Insert*

    ans = min(ans, del\_cost + Edit\_Dist\_Recur(s1, s2, i - 1, j));*// Remove*

    ans = min(ans, rep\_cost + Edit\_Dist\_Recur(s1, s2, i - 1, j - 1));*// Replace*

*return* ans;

}

Time Complexity = O(6(m+n))

(c)Top-down dynamic programming solution

**Code:**

*// (c) Top-down dynamic programming solution*

int Edit\_Dist\_TD(string s1, string s2, int i, int j, vector<vector<int>> &mem)

{

*if* (i < 0 || j < 0)

*return* 0;

*// If Sub-Problem has Already been Solved*

*if* (mem[i][j] != -1)

    {

*return* mem[i][j];

    }

*// If any string is empty,*

*// return the remaining characters of other string*

*if* (i == 0)

    {

        mem[i][j] = j \* ins\_cost;

*return* mem[i][j];

    }

*if* (j == 0)

    {

        mem[i][j] = min(i \* del\_cost, kill\_cost);

*return* mem[i][j];

    }

    int ans = MAX\_VAL;

*// If last characters are equal,*

*// recur for n-1, m-1*

*if* (s1[i - 1] == s2[j - 1])

    {

        ans = copy\_cost + Edit\_Dist\_TD(s1, s2, i - 1, j - 1, mem);

    }

*// Twiddle Case*

*if* (i >= 2 && j >= 2)

    {

*if* (s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[j - 1])

        {

            ans = min(ans, twiddle\_cost + Edit\_Dist\_TD(s1, s2, i - 2, j - 2, mem));

        }

    }

*// If characters are nt equal, we need to*

*// find the minimum cost out of all 3 operations.*

    ans = min(ans, ins\_cost + Edit\_Dist\_TD(s1, s2, i, j - 1, mem));*// Insert*

    ans = min(ans, del\_cost + Edit\_Dist\_TD(s1, s2, i - 1, j, mem));*// Remove*

    ans = min(ans, rep\_cost + Edit\_Dist\_TD(s1, s2, i - 1, j - 1, mem));*// Replace*

*return* mem[i][j] = ans;

}

Time Complexity = O(N2)

(d)Bottom-up dynamic programming solution

**Code:**

*// (d) Bottom-up dynamic programming solution*

int Edit\_Dist\_BU(string s1, string s2)

{

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

*// Table to Store the Results of Sub-Problems*

    int dp[m + 1][n + 1];

*// If the Second String is Empty,*

*// Only Option is to Remove all Characters from First String*

*for* (int i = 0; i < m + 1; i++)

    {

        dp[i][0] = min(i \* del\_cost, kill\_cost);

    }

*// If the First String is Empty,*

*// Only Option is to Insert all Characters in Second String*

*for* (int j = 0; j < n + 1; j++)

    {

        dp[0][j] = j \* ins\_cost;

    }

*// FILL dp[][] in Bottom Up Manner*

*for* (int i = 1; i <= m; i++)

    {

*for* (int j = 1; j <= n; j++)

        {

            dp[i][j] = MAX\_VAL;

*// If Last Character is Same,*

*// Ignore the Last Character and Recur for Remiaining String*

*if* (s1[i - 1] == s2[j - 1])

                dp[i][j] = dp[i - 1][j - 1] + copy\_cost;

*// Twiddle Case*

*if* (i >= 2 && j >= 2)

            {

*if* (s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[j - 1])

                {

                    dp[i][j] = min(dp[i][j], twiddle\_cost + dp[i - 2][j - 2]);

                }

            }

            dp[i][j] = min(dp[i][j], ins\_cost + dp[i][j - 1]);*// Insert*

            dp[i][j] = min(dp[i][j], del\_cost + dp[i - 1][j]);*// Remove*

            dp[i][j] = min(dp[i][j], rep\_cost + dp[i - 1][j - 1]);*// Replace*

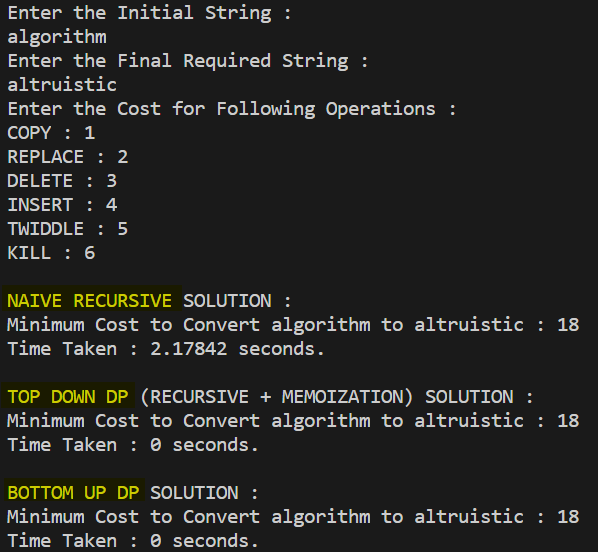
        }

    }

*return* dp[m][n];

}

**Test Case:**



**SUBMITTED BY:**

**U19CS012**

**BHAGYA VINOD RANA**