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

```
// (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)

    ans = 1 // minimum LPS is of Length 1
    n = s.length()

    lmt = 1 << n; //2^n Possibilities

    for i = 0 to lmt-1

        string tmp
        for bit = n - 1 to 0
            // If that Bit is Set, add that character
            if (i & (1 << bit))
                tmp += s[bit]

        if (is_palindrome(tmp))
            if tmp.length() > ans
                ans = tmp.length()

    return ans
```

Time Complexity: $O(2^n)$

(b) Naive recursive solution

```
// (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))
```

Time Complexity: $O(2^n)$

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

Algorithm 1: Solve function that finds the longest palindromic subsequence

```
Data: dp: 2D array that stores the answer to states
      sequence: The sequence to calculate the answer for
      L: The left index of current range
      R: The right index of the current range
Result: Returns the longest palindromic subsequence
if L > R then
    return 0;
else if L = R then
    return 1;
else if dp[L][R] ≠ −1 then
    return dp[L][R];
else if sequence[L] = sequence[R] then
    dp[L][R] ← 2 + solve(dp, sequence, L + 1, R − 1);
else
    moveL ← solve(dp, sequence, L + 1, R);
    moveR ← solve(dp, sequence, L, R − 1);
    dp[L][R] ← maximum(moveL, moveR);
return dp[L][R];
```

Time Complexity: $O(n^2)$

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

Algorithm 2: Bottom-up approach to find the longest palindromic subsequence

```
Data: sequence: The sequence to calculate the answer for
      n: The length of the sequence
Result: Returns the length of the longest palindromic subsequence
for i ← 1 to n do
    dp[i][0] ← 0;
    dp[i][1] ← 1;
for len ← 2 to n do
    for L ← 1 to n − len + 1 do
        R ← L + len − 1;
        if sequence[L] = sequence[R] then
            dp[L][len] ← 2 + dp[L + 1][len − 2];
        else
            moveL ← dp[L + 1][len − 1];
            moveR ← dp[L][len − 1];
            dp[L][len] ← maximum(moveL, moveR);
return dp[1][n];
```

Time Complexity: $O(n^2)$

Code:

```cpp
#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
```

```cpp
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) << e
ndl;
    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**PCABZB**Q**CP**BA [Palindrome of Length 7 {PCBZBCP}]

```
LEPCABZBQCPBA
NAIVE ITERATIVE SOLUTION :
Length of Longest Palindromic Subsequence : 7
Time Taken : 0.016145 seconds.

NAIVE RECURSIVE SOLUTION :
Length of Longest Palindromic Subsequence : 7
Time Taken : 0.001417 seconds.

TOP DOWN DP (RECURSIVE + MEMOIZATION) SOLUTION :
Length of Longest Palindromic Subsequence : 7
Time Taken : 0.000965 seconds.

BOTTOM UP DP SOLUTION :
Length of Longest Palindromic Subsequence : 7
Time Taken : 0 seconds.
```

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

```
GAMXVFLCZQPO
NAIVE ITERATIVE SOLUTION :
Length of Longest Palindromic Subsequence : 1
Time Taken : 0.008129 seconds.

NAIVE RECURSIVE SOLUTION :
Length of Longest Palindromic Subsequence : 1
Time Taken : 0.001798 seconds.

TOP DOWN DP (RECURSIVE + MEMOIZATION) SOLUTION :
Length of Longest Palindromic Subsequence : 1
Time Taken : 0.001634 seconds.

BOTTOM UP DP SOLUTION :
Length of Longest Palindromic Subsequence : 1
Time Taken : 0.000998 seconds.
```

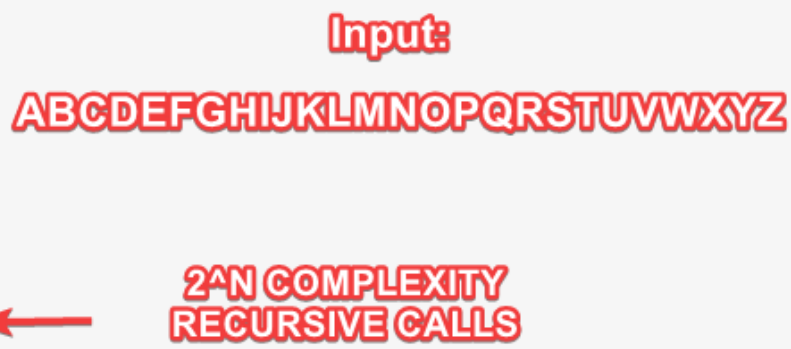
The dynamic programming approach is indeed `O(n^2)`. However, the recursive solution is exponential in `n`: any time two characters don't match, a subproblem of size `k` is converted into `2` subproblems of size `k-1` each. It's easy to see that, with all letters different, this produces a complexity of `O(2^n)`.

$$T(n) = \text{running time on input of length } n$$

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) & n > 1 \end{cases}$$

$$= 2^{n-1}$$

```
n = 1: answer = 1, count = 1
n = 2: answer = 1, count = 3
n = 3: answer = 1, count = 7
n = 4: answer = 1, count = 15
n = 5: answer = 1, count = 31
...
n = 24: answer = 1, count = 16777215
n = 25: answer = 1, count = 33554431
n = 26: answer = 1, count = 67108863
```



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

B.) Edit distance problem

Convert String1 to String2 Using Any Set of 6 Operations:

Copy a character from x to z by setting $z[j] = x[i]$ and then incrementing both i and j. This operation examines $x[i]$.

Replace a character from x by another character c, by setting $z[j] = c$, and then incrementing both i and j. This operation examines $x[i]$.

Delete a character from x by incrementing i but leaving j alone. This operation examines $x[i]$.

Insert the character c into z by setting $z[j] = c$ and then incrementing j, but leaving i alone. This operation examines no characters of x.

Twiddle (i.e., exchange) the next two characters by copying them from x to z but in the opposite order; we do so by setting $z[j] = x[i+1]$ and $z[j+1] = x[i]$ and then setting $i = i + 2$ and $j = j + 2$. This operation examines $x[i]$ and $x[i+1]$.

Kill the remainder of x by setting $i = m + 1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation.

(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:

```
Enter the Initial String :
not
Enter the Final Required String :
out
Enter the Cost for Following Operations :
COPY : 1
REPLACE : 2
DELETE : 3
INSERT : 4
TWIDDLE : 5
KILL : 6
NAIVE ITERATIVE SOLUTION :
Minimum Cost to Convert not to out : 5
Time Taken : 0.915986 seconds.

NAIVE RECURSIVE SOLUTION :
Minimum Cost to Convert not to out : 5
Time Taken : 0.000995 seconds.

TOP DOWN DP (RECURSIVE + MEMOIZATION) SOLUTION :
Minimum Cost to Convert not to out : 5
Time Taken : 0 seconds.

BOTTOM UP DP SOLUTION :
Minimum Cost to Convert not to out : 5
Time Taken : 0 seconds.
```

(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 = $O(N^2)$

(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:

```
Enter the Initial String :
algorithm
Enter the Final Required String :
altruistic
Enter the Cost for Following Operations :
COPY : 1
REPLACE : 2
DELETE : 3
INSERT : 4
TWIDDLE : 5
KILL : 6

NAIVE RECURSIVE SOLUTION :
Minimum Cost to Convert algorithm to altruistic : 18
Time Taken : 2.17842 seconds.

TOP DOWN DP (RECURSIVE + MEMOIZATION) SOLUTION :
Minimum Cost to Convert algorithm to altruistic : 18
Time Taken : 0 seconds.

BOTTOM UP DP SOLUTION :
Minimum Cost to Convert algorithm to altruistic : 18
Time Taken : 0 seconds.
```


SUBMITTED BY:

U19CS012

BHAGYA VINOD RANA