Assignment 6

Dhruv Kulkarni
U21CS036

April 20, 2023

1 Longest Palindromic subsequence

The following is the solution to the longest palindromic subsequence problem using a naive iterative approach.

```
def is_palindrome(s):
    return s == s[::-1]

def longest_palindromic_subsequence(s):
    n = len(s)
    longest = ""
    for i in range(n):
        for j in range(i, n):
            if is_palindrome(s[i:j+1]):
                if len(s[i:j+1]) > len(longest):
                    longest = s[i:j+1]
    return longest
```

Intuition:Checking the substring after reversing it you basically check if the revrsed substring is the same as the original substring. And you do this iteratively to find the longest such combination. Time complexity $O(2^n)$

2 Longest Palindromic subsequence-Naive recursive

The following is the solution to the longest palindromic subsequence problem using a naive recursive approach.

```
def is_palindrome(s):
    return s == s[::-1]

def longest_palindromic_subsequence_helper(s, i, j):
    if i == j:
        return s[i]
    if is_palindrome(s[i:j+1]):
        return s[i:j+1]
    left = longest_palindromic_subsequence_helper(s, i, j-1)
    right = longest_palindromic_subsequence_helper(s, i+1, j)
    if len(left) > len(right):
        return left
    else:
        return right

def longest_palindromic_subsequence(s):
    return longest_palindromic_subsequence_helper(s, 0, len(s)-1)
```

Intuition:Divide the problem into two parts, and find the solution for each part, and then combine them.Here the helper() function is doing that. Time complexity O(2^n)

3 Longest Palindromic subsequence-Top Down DP

The following is the solution to the longest palindromic subsequence problem using a code Top Down DP approach.

```
def longest_palindromic_subsequence_helper(s, i, j, memo):
    if i == j:
        return s[i]
    if (i, j) in memo:
        return memo[(i, j)]
    if s[i] == s[j]:
        memo[(i, j)] = s[i] + longest_palindromic_subsequence_helper(s,
    else:
        left = longest_palindromic_subsequence_helper(s, i, j-1, memo)
        right = longest_palindromic_subsequence_helper(s, i+1, j, memo)
        if len(left) > len(right):
            memo[(i, j)] = left
        else:
            memo[(i, j)] = right
    return memo[(i, j)]

def longest_palindromic_subsequence(s):
    memo = {}
    return longest_palindromic_subsequence_helper(s, 0, len(s)-1, memo)
```

Intuition:We implement the process of memoization here. This involves breaking the problem into interlinked subproblems and solving them to find the answer.Use memoization to avoid redundant calculations.TC:O(n^2)

4 Longest Palindromic subsequence-Bottom Up DP

The following is the solution to the longest palindromic subsequence problem using a bottom up DP approach.

```
def longest_palindromic_subsequence(s):
    n = len(s)

    L = [[0] * n for _ in range(n)]

    for i in range(n):
        L[i][i] = 1

    for cl in range(2, n+1):
        for i in range(n-cl+1):
            j = i+cl-1
            if s[i] == s[j] and cl == 2:
                L[i][j] = 2
            elif s[i] == s[j]:
                L[i][j] = L[i+1][j-1] + 2
            else:
                L[i][j] = max(L[i+1][j], L[i][j-1])

    longest = ""
    i, j = 0, n-1
    while i <= j:
        if s[i] == s[j]:
            longest += s[i]
            i += 1
            j -= 1
        elif L[i+1][j] > L[i][j-1]:
            i += 1
        else:
            j -= 1
    return longest
```

Intuition:Using a master matrix with find the longest palindromic subsequence present in the string.

5 Edit Distance-Naive iterative

The following is the solution to the edit distance problem using a iterative approach.

```
def edit_distance(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n+1) for _ in range(m+1)]
    for i in range(m+1):
        for j in range(n+1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]
    return dp[m][n]
```

Intuition:Using a master matrix with find the minimum no. of operations to convert one string to another string.

6 Edit Distance-Naive Recursive

The following is the solution to the edit distance problem using a recursive approach.

```
def edit_distance(s1, s2):
    def helper(i, j):
        if i == 0:
            return j
        elif j == 0:
            return i
        elif s1[i-1] == s2[j-1]:
            return helper(i-1, j-1)
        else:
            return 1 + min(helper(i-1, j), helper(i, j-1), helper(i-1, j
    return helper(len(s1), len(s2))
```

Intuition:Do the same as above but recursively.

7 Edit Distance-Top Down DP

The following is the solution to the edit distance problem using a Top Down DP approach.

```
def edit_distance(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[-1] * (n+1) for _ in range(m+1)]
    def helper(i, j):
        if dp[i][j] != -1:
            return dp[i][j]
        if i == 0:
            dp[i][j] = j
        elif j == 0:
            dp[i][j] = i
        elif s1[i-1] == s2[j-1]:
            dp[i][j] = helper(i-1, j-1)
        else:
            dp[i][j] = 1 + min(helper(i-1, j), helper(i, j-1), helper(i
        return dp[i][j]
    return helper(m, n)
```

Intuition:This is a Top Down DP approach.Here too we implement memoization.Storing results of interlinked subproblems to avoid redundant calculations.

8 Edit Distance-Bottom Up DP

The following is the solution to the edit distance problem using a Bottom DP approach.

```
def edit_distance(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n+1) for _ in range(m+1)]
    for i in range(m+1):
        dp[i][0] = i
    for j in range(n+1):
        dp[0][j] = j
    for i in range(1, m+1):
        for j in range(1, n+1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]
    return dp[m][n]
```

Intuition:This is a Bottom Up DP approach.