Assignment 6

CS 514 – Algorithms

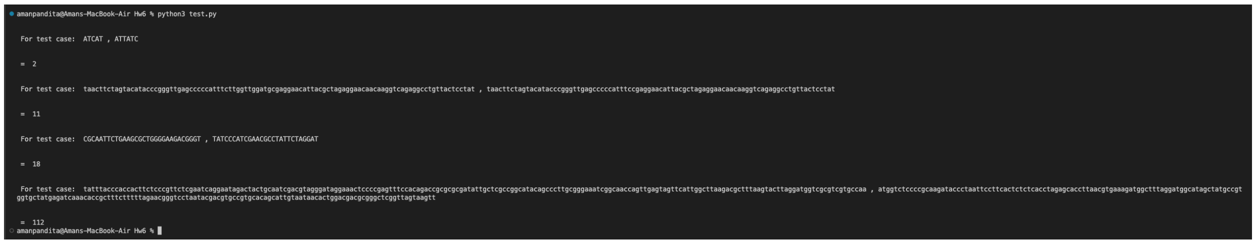
Submitted By: Aman Pandita

Onid: panditaa@oregonstate.edu

(a) (20 points) Make sure that your program works on this test data.

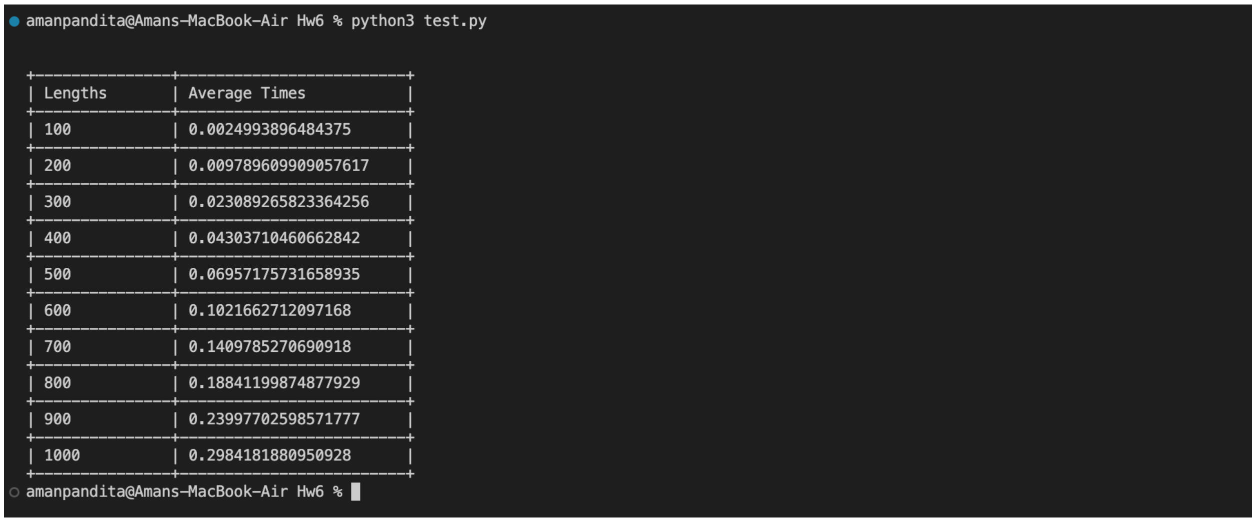
editDistance("ATCAT", "ATTATC") == 2
editDistance("taacttctagtacatacccgggttgagcccccatttcttggttggatgcgaggaacattacgctagaggaacaacaaggtcagaggcctgttactcctat",
"taacttctagtacatacccgggttgagcccccatttccgaggaacattacgctagaggaacaacaaggtcagaggcctgttactcctat")==11
editDistance("CGCAATTCTGAAGCGCTGGGGAAGACGGGT", "TATCCCATCGAACGCCTATTCTAGGAT") ==18
editDistance("tatttacccaccacttctcccgttctcgaatcaggaatagactactgcaatcgacgtagggataggaaactccccgagtttccacagaccgcgcgcgatattgctcgccggcatacagcccttgcgggaaatcggcaaccagttgagtagttcattggcttaagacgctttaagtacttaggatggtcgcgtcgtgccaa",
"atggtctccccgcaagataccctaattccttcactctctcacctagagcaccttaacgtgaaagatggctttaggatggcatagctatgccgtggtgctatgagatcaaacaccgctttctttttagaacgggtcctaatacgacgtgccgtgcacagcattgtaataacactggacgacgcgggctcggttagtaagtt")
==112

My output:

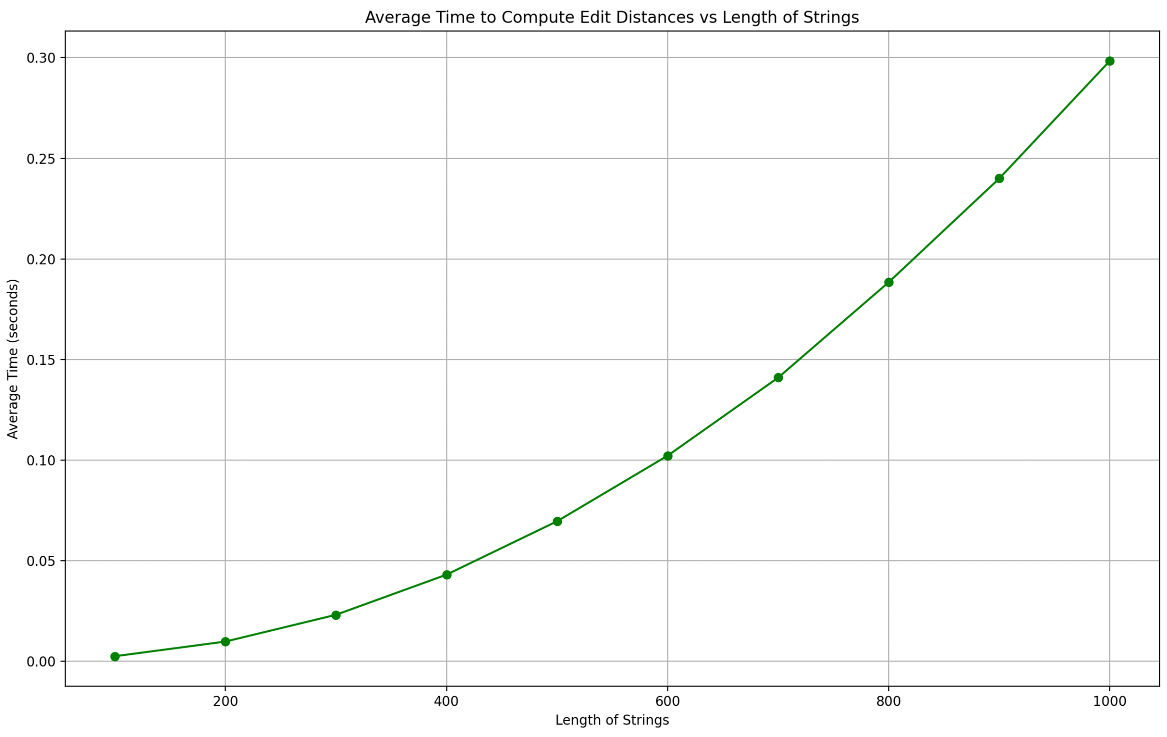


```
● amanpandita@Amans-MacBook-Air Hw6 % python3 test.py


  For test case:  ATCAT , ATTATC


  =  2


  For test case:  taacttctagtacatacccgggttgagcccccatttcttggttggatgcgaggaacattacgctagaggaacaacaaggtcagaggcctgttactcctat , taacttctagtacatacccgggttgagcccccatttccgaggaacattacgctagaggaacaacaaggtcagaggcctgttactcctat


  =  11


  For test case:  CGCAATTCTGAAGCGCTGGGGAAGACGGGT , TATCCCATCGAACGCCTATTCTAGGAT


  =  18


  For test case:  tatttacccaccacttctcccgttctcgaatcaggaatagactactgcaatcgacgtagggataggaaactccccgagtttccacagaccgcgcgcgatattgctcgccggcatacagcccttgcgggaaatcggcaaccagttgagtagttcattggcttaagacgctttaagtacttaggatggtcgcgtcgtgccaa , atggtctccccgcaagataccctaattccttcactctctcacctagagcaccttaacgtgaaagatggctttaggatggcatagctatgccgt
 ggtgctatgagatcaaacaccgctttctttttagaacgggtcctaatacgacgtgccgtgcacagcattgtaataacactggacgacgcgggctcggttagtaagtt


  =  112
○ amanpandita@Amans-MacBook-Air Hw6 %
```

(b) (20 points) Generate 10 random pairs of edit strings of length 100, 200, ...,1000, and find their edit distances. Plot the average time taken to compute their edit distances as a function of the length of strings. Comment on the performance of your algorithm.



```
● amanpandita@Amans-MacBook-Air Hw6 % python3 test.py


  +----------------+---------------------------+
  | Lengths        | Average Times             |
  +----------------+---------------------------+
  | 100            | 0.0024993896484375        |
  +----------------+---------------------------+
  | 200            | 0.009789609909057617      |
  +----------------+---------------------------+
  | 300            | 0.023089265823364256      |
  +----------------+---------------------------+
  | 400            | 0.04303710460662842       |
  +----------------+---------------------------+
  | 500            | 0.06957175731658935       |
  +----------------+---------------------------+
  | 600            | 0.1021662712097168        |
  +----------------+---------------------------+
  | 700            | 0.1409785270690918        |
  +----------------+---------------------------+
  | 800            | 0.18841199874877929       |
  +----------------+---------------------------+
  | 900            | 0.23997702598571777       |
  +----------------+---------------------------+
  | 1000           | 0.2984181880950928        |
  +----------------+---------------------------+
○ amanpandita@Amans-MacBook-Air Hw6 %
```

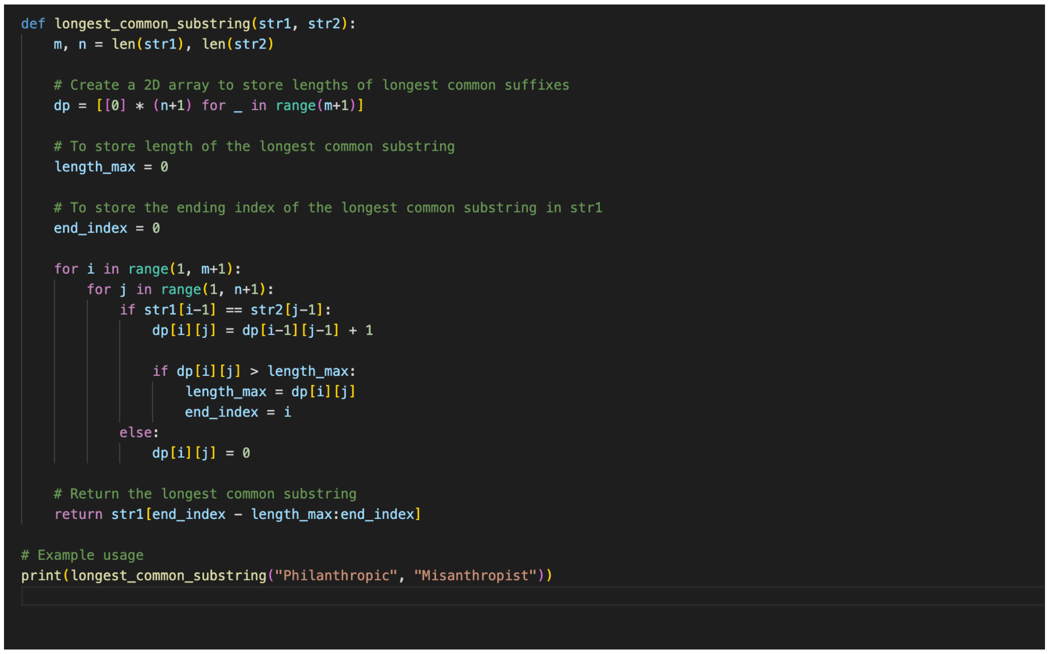


We get the following information from the results above:

- The average time increases as the length of the string’s increases.
- This confirms the behavior of the algorithm, as the algorithm has a ***time complexity of O(m*n)***, where m and n are the lengths of the two strings.
- For strings of length 100 to 1000, the average time increases from approximately ***0.0024*** seconds to ***0.2984 seconds***. The trend appears to be more than linear, suggesting the quadratic nature of the algorithm's complexity.

This proves that algorithm is efficient for short strings but becomes significantly slower as the string length increases. This indeed confirms the quadratic nature of the algorithm and suggests us to use more efficient algorithms in case of larger strings

2. (10 points) Give an *O(mn)* algorithm for finding the longest common substring of two input strings of length *m* and *n*. For example, if the two inputs are 'Philanthropic" and "Misanthropist," the output should be "anthropi."



```python
def longest_common_substring(str1, str2):
    m, n = len(str1), len(str2)

    # Create a 2D array to store lengths of longest common suffixes
    dp = [[0] * (n+1) for _ in range(m+1)]

    # To store length of the longest common substring
    length_max = 0

    # To store the ending index of the longest common substring in str1
    end_index = 0

    for i in range(1, m+1):
        for j in range(1, n+1):
            if str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1

                if dp[i][j] > length_max:
                    length_max = dp[i][j]
                    end_index = i
            else:
                dp[i][j] = 0

    # Return the longest common substring
    return str1[end_index - length_max:end_index]

# Example usage
print(longest_common_substring("Philanthropic", "Misanthropist"))
```

1. ***Initialization:***

   A 2D array ***"dp"*** of size **(m+1) x (n+1)** is created, where **m** and **n** are the lengths of the two input strings, **str1** and **str2**. This array will hold the lengths of the longest common suffixes of substrings ending at each position in **str1** and **str2**. The array is initialized with 0.

2. ***Filling the 'dp' Array:***

   It iterates through each character of **str1** and **str2**.
   For each pair of characters **(i, j)**, if **str1[i]** = **str2[j]**,
   *set **dp[i][j]** = **dp[i-1][j-1]** + 1,*
   which is the length of the longest common suffix ended at **str1[i]** and **str2[j]**, extended by the matching characters.
   If the characters do not match, set **dp[i][j]** = 0, indicating no common suffix.

3. ***Tracking the Maximum Length:***

   As ***"dp"*** array fills, we keep track of the maximum length of any common substring found so far and the position where it ends in **str1**.

4. ***Extracting the Longest Common Substring:***

   Using the maximum length and the ending position, the extraction of the longest common substring from **str1**.

3. (30 points) BigBucks wants to open a set of coffee shops in the I-5 corridor. The possible locations are at miles $d_1, \ldots, d_n$ in a straight line to the south of their Headquarters. The potential profits are given by $p_1 \ldots p_n$. The only constraint is that the distance between any two shops must be at least k (a positive integer).

- Construct a counterexample to show that a greedy algorithm that chooses in the order of profits could miss the optimal (most profitable) solution.
- Give an efficient dynamic programming-based algorithm to maximize the profit.

Counter Example:

Let's assume:

coffee shop locations at distances-→ ***[d1, d2, d3, d4]***
Corresponding profits → ***[p1, p2, p3, p4]***
Let's set the minimum distance ***k*** between any two shops.

Suppose:

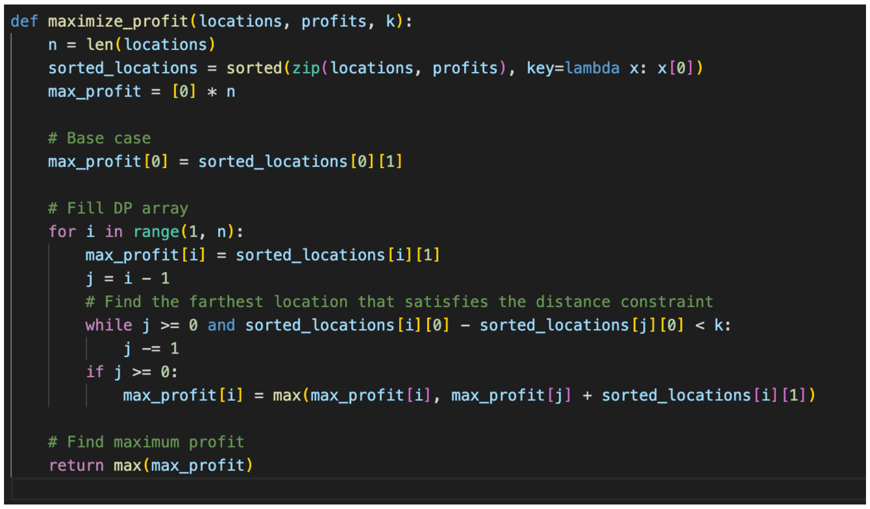
- ***d1 = 0, d2 = k, d3 = 2k, d4 = 3k*** (distances are multiples of ***k***, ensuring they meet the minimum distance constraint).
- ***p1 = 100, p2 = 300, p3 = 200, p4 = 1000*** (profits at each location).

A greedy algorithm that chooses based on the highest profit would select ***d4*** first (for the highest profit of ***p4 = 1000***). However, selecting **d4** precludes the selection of ***d2*** and ***d3*** due to the minimum distance constraint.

The total profit with the greedy choice would be = ***1000*** (only ***d4***).

An optimal solution would select ***d2*** and ***d4***, giving a total profit of ***300 + 1000 = 1300***. This is higher than the greedy solution and **shows that the greedy approach can miss the optimal solution**.

Dynamic Programming algorithm to maximize the profit:



```python
def maximize_profit(locations, profits, k):
    n = len(locations)
    sorted_locations = sorted(zip(locations, profits), key=lambda x: x[0])
    max_profit = [0] * n

    # Base case
    max_profit[0] = sorted_locations[0][1]

    # Fill DP array
    for i in range(1, n):
        max_profit[i] = sorted_locations[i][1]
        j = i - 1
        # Find the farthest location that satisfies the distance constraint
        while j >= 0 and sorted_locations[i][0] - sorted_locations[j][0] < k:
            j -= 1
        if j >= 0:
            max_profit[i] = max(max_profit[i], max_profit[j] + sorted_locations[i][1])

    # Find maximum profit
    return max(max_profit)
```

About the algorithm:

Sort Locations: Sort the locations in increasing order of distances.

Initialize DP Array: Create a DP array **max_profit** of length **n** (number of locations), where **max_profit[i]** will store the maximum profit that can be obtained considering up to the **i**$^{\text{th}}$ location.

Base Case: Set ***max_profit[0] = p1*** as the initial condition.

Fill DP Array: For each location ***i*** from 1 to ***n-1***:

- Initialize ***max_profit[i]*** as the profit of the current location ***pi***.
- Find the previous location ***j (where j < i)*** that is the farthest yet satisfies the distance constraint (i.e., distance from ***i*** is at least ***k***). This can be done more efficiently using a binary search or keeping a pointer that moves only forward.
- Update ***max_profit[i]*** as the maximum of its current value and ***max_profit[j] + pi.***

Find Maximum Profit: The final answer will be the maximum value in the ***max_profit*** array.

Time Complexity:

The sorting complexity remains ***O(nlogn)***. After that comes the step for filling the dp array, for each location ***i***, we don't iterate through all previous locations ***j***. Instead, we quickly find the farthest valid location that satisfies the distance constraint, significantly reducing the number of operations. This part's complexity is closer to ***O(n)***, as each location ***i*** requires at most a single pass backward through the list.

The ***overall time complexity*** of the algorithm is *O(nlogn+n)*, which simplifies to ***O(nlogn)*** *(as nlogn dominates n for large n).*

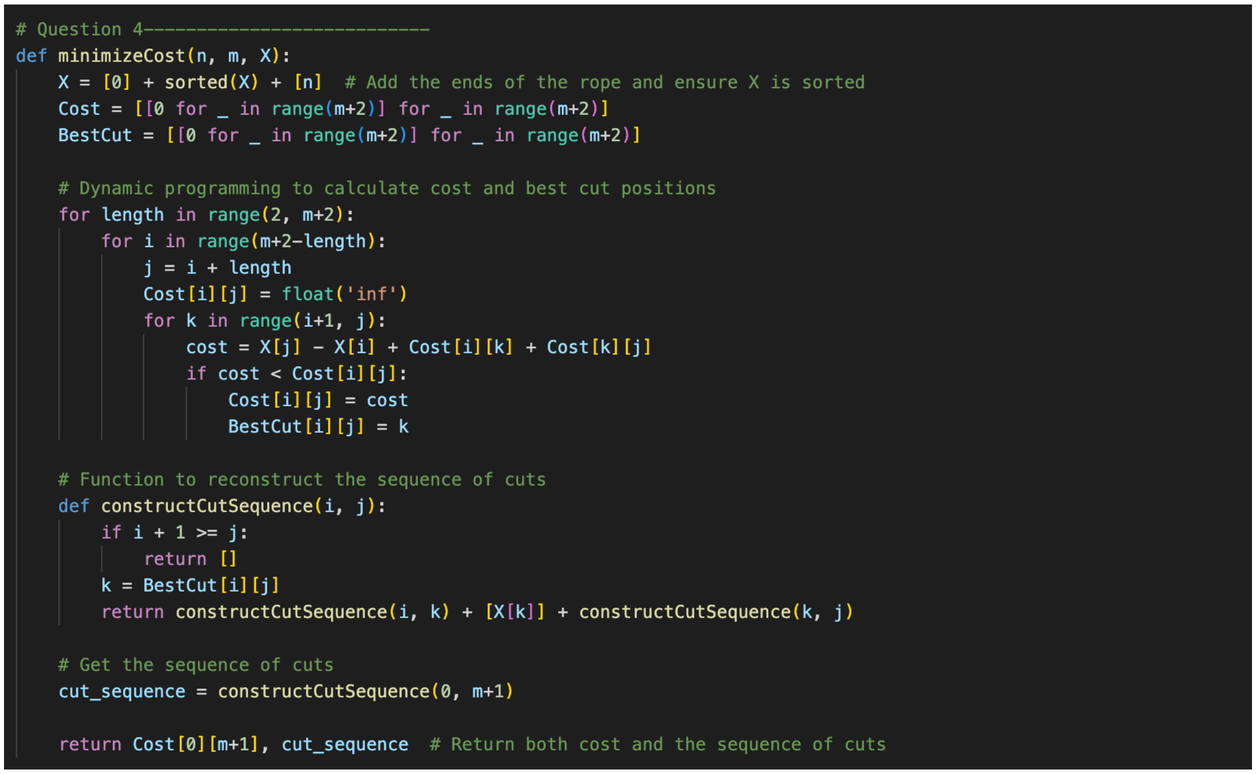
4. (20 points) In a rope cutting problem, cutting a rope of length *n* into two pieces costs *n* time units, regardless of the location of the cut. You are given m desired locations of the cuts, X_1, …., X_m. Give a dynamic programming-based algorithm to find the optimal sequence of cuts to cut the rope into m+1 pieces to minimize the total cost.

Hint: Let X_0 and $X_{[m+1]}$ be the two ends of the rope. Write a Bellman equation for Cost[X_i , X_j] which represents the minimum cost of cutting the part of the rope from location X_i to location X_j into j-i pieces in between.

Given:

1. Rope of length ***n***
2. ***m*** desired locations of the cuts: ***X1, X2, …, Xm***
3. The cost of cutting the rope of length ***n*** into two pieces is ***n*** time units.

Let's define the two ends of the rope as ***X0 = 0*** and ***X[m+1] = n***. We need to find the minimum cost to cut the rope into ***m+1*** pieces.



```python
# Question 4---------------------------
def minimizeCost(n, m, X):
    X = [0] + sorted(X) + [n]  # Add the ends of the rope and ensure X is sorted
    Cost = [[0 for _ in range(m+2)] for _ in range(m+2)]
    BestCut = [[0 for _ in range(m+2)] for _ in range(m+2)]

    # Dynamic programming to calculate cost and best cut positions
    for length in range(2, m+2):
        for i in range(m+2-length):
            j = i + length
            Cost[i][j] = float('inf')
            for k in range(i+1, j):
                cost = X[j] - X[i] + Cost[i][k] + Cost[k][j]
                if cost < Cost[i][j]:
                    Cost[i][j] = cost
                    BestCut[i][j] = k

    # Function to reconstruct the sequence of cuts
    def constructCutSequence(i, j):
        if i + 1 >= j:
            return []
        k = BestCut[i][j]
        return constructCutSequence(i, k) + [X[k]] + constructCutSequence(k, j)

    # Get the sequence of cuts
    cut_sequence = constructCutSequence(0, m+1)

    return Cost[0][m+1], cut_sequence  # Return both cost and the sequence of cuts
```

Dynamic Programming Approach:

1. ***State Definition:***

   Cost[Xi, Xj] is the minimum cost of cutting the part of the rope from location ***Xi*** to location ***Xj***. Additionally, we maintain a separate table, ***BestCut[Xi, Xj]***, to store the position of the best cut between ***Xi*** and ***Xj***.

2. ***Base Case:***

   For any two adjacent locations ***Xi*** and ***Xi+1***, ***Cost [Xi, Xi+1] = 0*** and ***BestCut[Xi, Xi+1]*** is undefined.

3. ***Optimized Recursive Relation:***

   The main optimization is in reducing the range of possible cuts that need to be considered for each subproblem.
   This can be done by making sure optimal cut for a segment ***[Xi, Xj]*** lies between the optimal cuts for the segments ***[Xi, Xk]*** and ***[Xk, Xj]***, where ***Xk*** is between ***Xi*** and ***Xj***.

   For segment ***[Xi, Xj]***, we need to find ***k*** such that ***Xi < k < Xj*** and it minimizes ***Cost[Xi, Xj]*** as per the formula:

   $$\underline{Cost[Xi, Xj] = (Xj - Xi) + \min_{BestCut[Xi, Xk] <= k <= BestCut[Xk, Xj]}(Cost[Xi, Xk] + Cost[Xk, Xj]))]}$$

4. ***Implementation***:

   Implement this using two 2D arrays: one for *Cost* and another for *BestCut*. The optimal cut locations are updated based on the solution of smaller subproblems.

This approach can significantly reduce the number of potential cuts that need to be considered, especially for larger instances.

The exact time complexity depends on the distribution of the cut points but is ***generally better than O(m^3)***.

References:

1. Geeks for Geeks, https://www.geeksforgeeks.org/

2. Leetcode, https://leetcode.com/