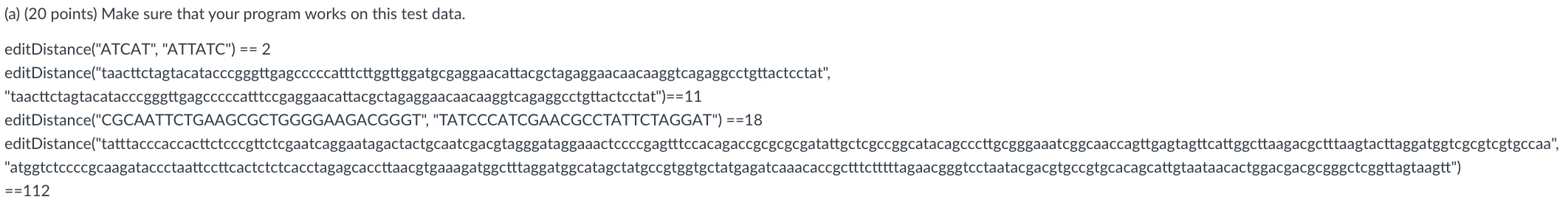
Assignment 6

CS 514 – Algorithms

Submitted By: Aman Pandita

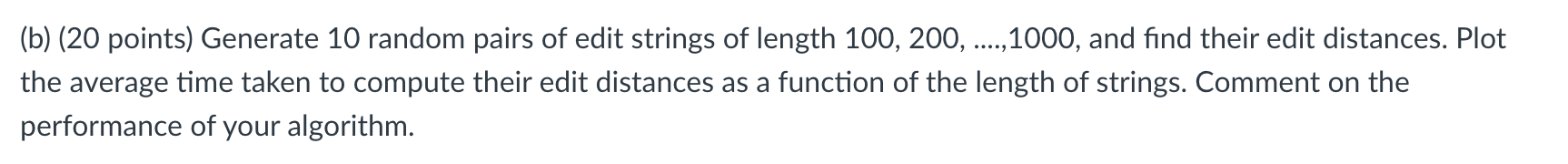
Onid: [panditaa@oregonstate.edu](mailto:panditaa@oregonstate.edu)



*My output:*

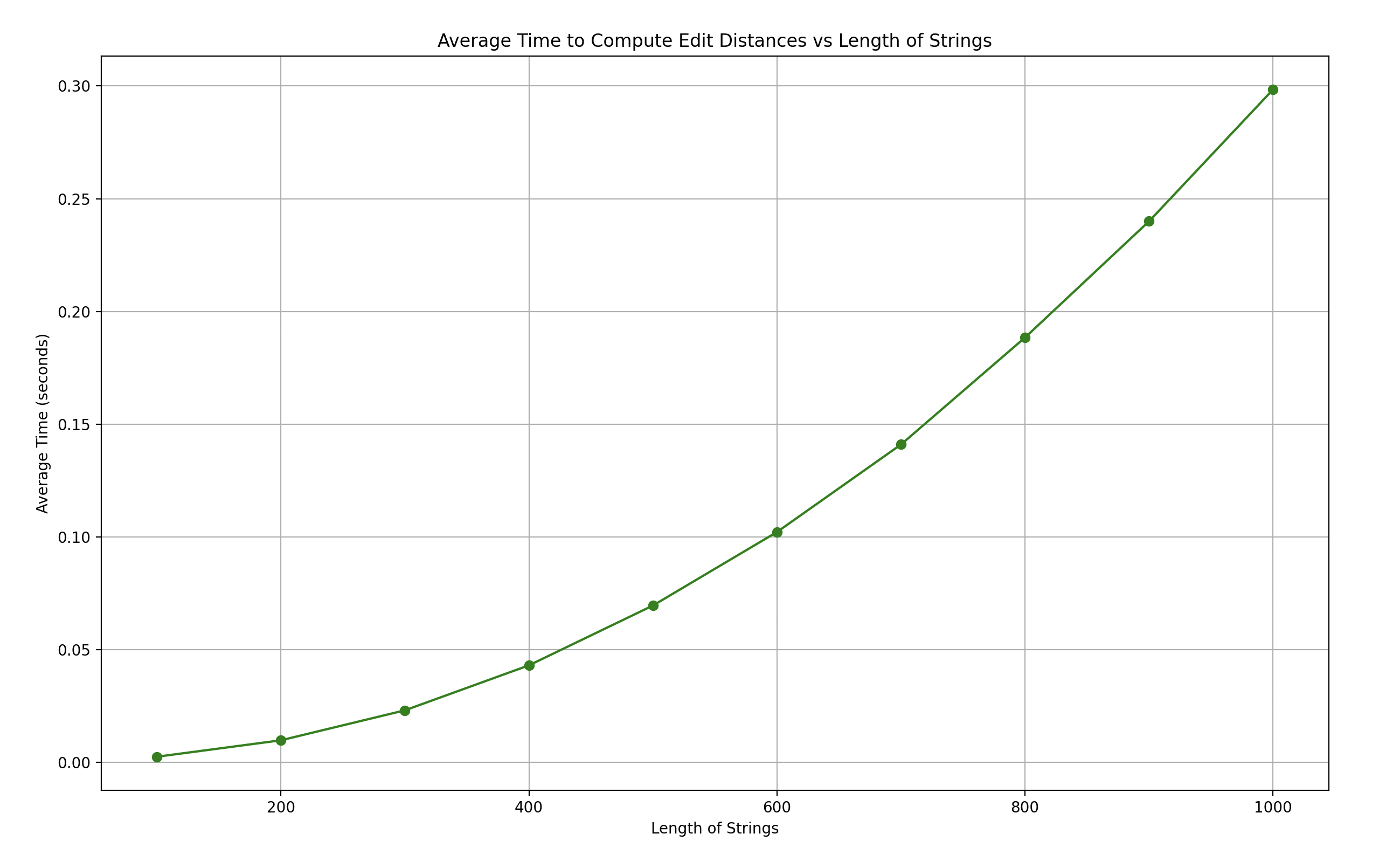
A black and white rectangular object

Description automatically generated



A screenshot of a computer

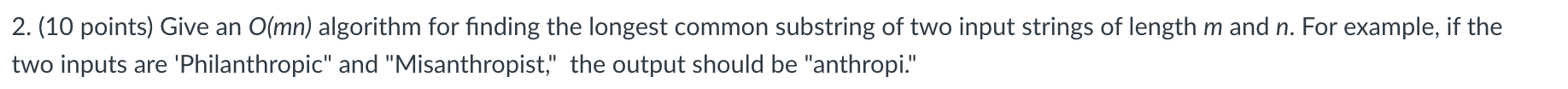
Description automatically generated



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



A computer screen shot of a program

Description automatically generated

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.

1. ***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**.

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

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

A white background with black text

Description automatically generated

***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:***

A screen shot of a computer program

Description automatically generated

***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 **ith** 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).*

A white background with black text

Description automatically generated

*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.

A computer screen shot of a program code

Description automatically generated

***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***.

1. ***Base Case:***

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

1. ***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:

*Cost[Xi, Xj] = (Xj - Xi) + minBestCut[Xi, Xk] <= k <= BestCut[Xk, Xj](Cost[Xi, Xk] + Cost[Xk, Xj])]*

1. ***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(m3)***.

**References:**  
  
1. Geeks for Geeks, <https://www.geeksforgeeks.org/>

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