Analysis of Algorithms: Assignment 2

Kaiviswanathan Srikant Iyer (5222-2519)

19th Nov 2021

1 Problem 1

1.1 Algorithm

Dynamic programming is a problem solving approach, where a problem is broken down into smaller subproblems. Using this approach, we avoid recomputations of repeated subproblems. A two dimensional array is used to implement the dynamic approach for the weighted approximate common substring algorithm. Weights of the alphabets are stored in a hashmap. The weights are based on the frequency of the letters usage in the English language.

Two for loops are used to traverse through the lengths of both given input strings; when two letters from both input strings match, the weight of the matched letter is added to the previous diagonal element. This value is used to fill the current cell in the two dimensional array. When a mismatch is come upon, the current cell value is filled with the previous diagonal elements value minus the penalty cost. The penalty is a random intermediate value between the smallest and highest weights of the alphabets considered.

The final recurrence equation is

$$Match[i,j] = \begin{cases} \max(Match(\text{i}-1,\text{j}-1))+\text{occurance(match) if } Str1[i] = Str2[j] \\ \text{Match}(i\text{-}1,\ j\text{-}1)-\text{penalty if } Str1[i] \neq Str2[j] \end{cases}$$

1.2 Algorithm Correctness

The proof of the substring being largest available using dynamic is shown below.

Algorithm 1: Substring with Penalty

Input: Two Input strings Str1,Str2 not necessarily of same length

Output: Largest sub string with penalties

Let *occurance* **be a hash-map with values assigned to each alphabet in accordance with their frequency in the English.**

Let *penalty* **be a random intermediate value between the lowest and highest weight of the letters**

$m \leftarrow$ *length*($Str1$)

$n \leftarrow$ *length*($Str2$)

for $i \leftarrow$ *0 to* m **do**

  for $j \leftarrow$ *0 to* n **do**

    if $i == 0$ || $j == 0$ **then**

      $Match[i][j] = 0$;

    else if *Str1[i]=Str2[i]* **then**

      Match[i][j]=Weight[Str1]+Match[i-1][j-1];

    else

      $Match[i][j]$= $Match[i-1][j-1]$ - *penalty*; ;

    end

  end

end

1.2.1 Proof by Induction

Proof: The ideal case is when we can set Match[i,j]=weight(match). For the Induction Hypothesis, we can assume all recurrence is similar $i \leq m$ and for $j \leq n$. When we get Match[k+1,j]:

- If $Str1_{k+1}$=$Str2_j$, then Match(k+1,j) is the same as Match[k,j-1] plus matched alphabet weight
- If $Str1_{k+1}$≠Str2$_j$, then Match(k+1,j) is the same as Match[k,j-1] minus the penalty set.

$$Match[i,j] = \begin{cases} \max(Match(\text{i}-1,\text{j}-1))+\text{occurance(match) if } Str1[i] = Str2[j] \\ \text{Match}(i\text{-}1,\ j\text{-}1)-\text{penalty if } Str1[i] \neq Str2[j] \end{cases}$$

1.2.2 Proof of Termination

Case 1: Substrings are found

Case 2: No substring exists

Claim: Algorithm terminates with $m * n$ iterations where m is the length of string $Str1$ and n is the length of string $Str2$

```
Length of longest substring with penalties is: 21
8.12 8.05 9.54 12.25 20.37 28.49 28.42 28.35 29.84 29.77 31.26 31.19 35.51 35.44 35.37 35
AABCAABBBDBCDHODSBBDB
ABBCAACCBBBBDBCBBBBBB
The highest weight of common substing with penalty of 0.07 per mismatch is: 39.0
```

Figure 1: Result of Substring with penalty 0.07.

```
Length of longest substring with penalties is: 4
1.49 4.2 12.32 20.44
BCAA
BCAA
The highest weight of common substing with penalty of 10.57 per mismatch is: 20.0
```

Figure 2: Result of Substring with penalty 10.57.

Proof: In the algorithm we traverse through the characters of the given input strings and make a two dimensional array. If a match is found or not, traversal through the lengths of both input strings will bring us to a stop.

1.3 Time complexity and space

The time complexity for finding the weighted approximate common sub-string is $O(m * n)$, where m and n are the lengths of the two input strings. Two loops are used traversing the lengths of both input strings, resulting in a time complexity of $O(m * n)$. The average, best and worst case time complexity is $O(m * n)$. The space complexity is $O(m * n)$ as we construct a two dimensional array having m rows and n columns to implement the dynamic programming approach.The worst case time complexity will also be $O(m * n)$, as even in the worst case there is no string match, we will traverse the lengths of both strings.

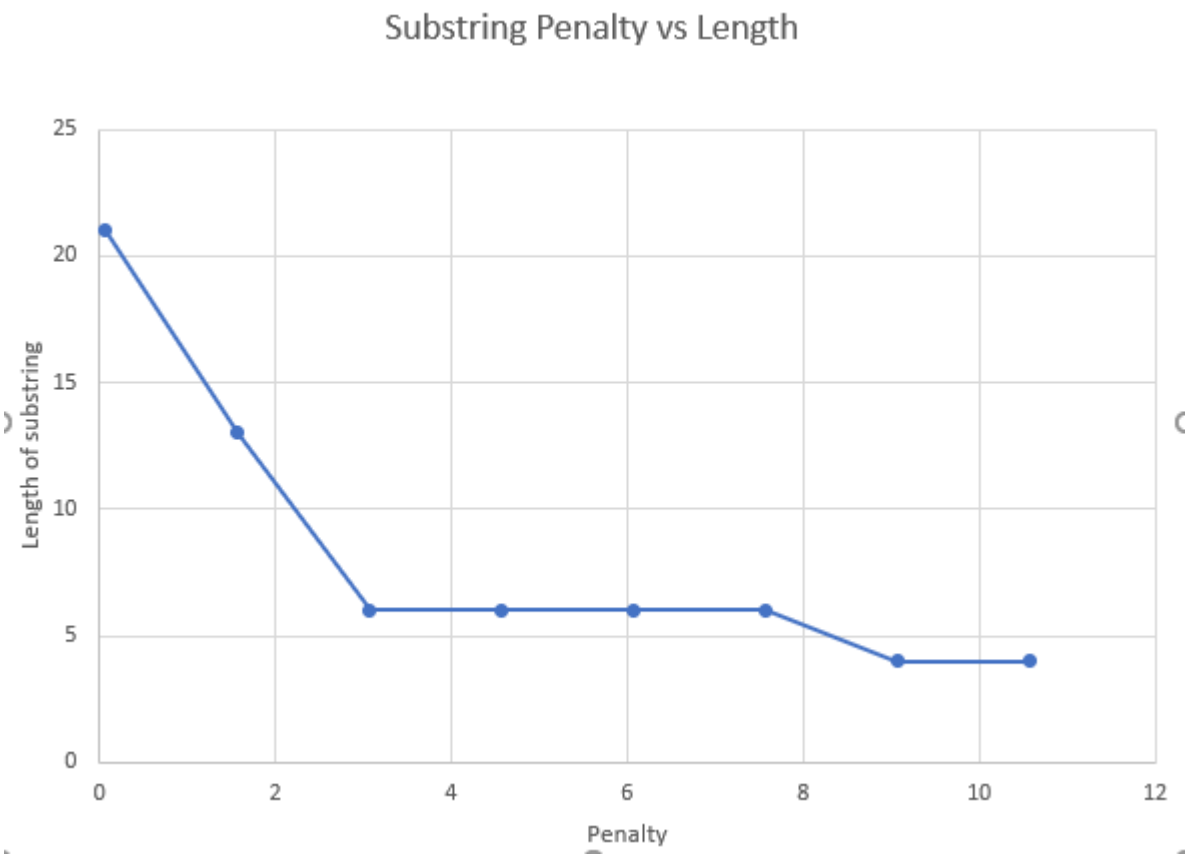

Figure 3: Graph 1: Penalty vs String Length

For the figure Fig 3: we can clearly observe as the penalty increases for the pair of same same string the number of substring reduces. this is due to the presence of different values in between the common substring which during smaller penalties was worth including as the payoff of including the common characters was more. As penalty increased, including the substring separated by tbe different characters started decreasing the overall value, so it was discarded thereby making the common substring smaller.

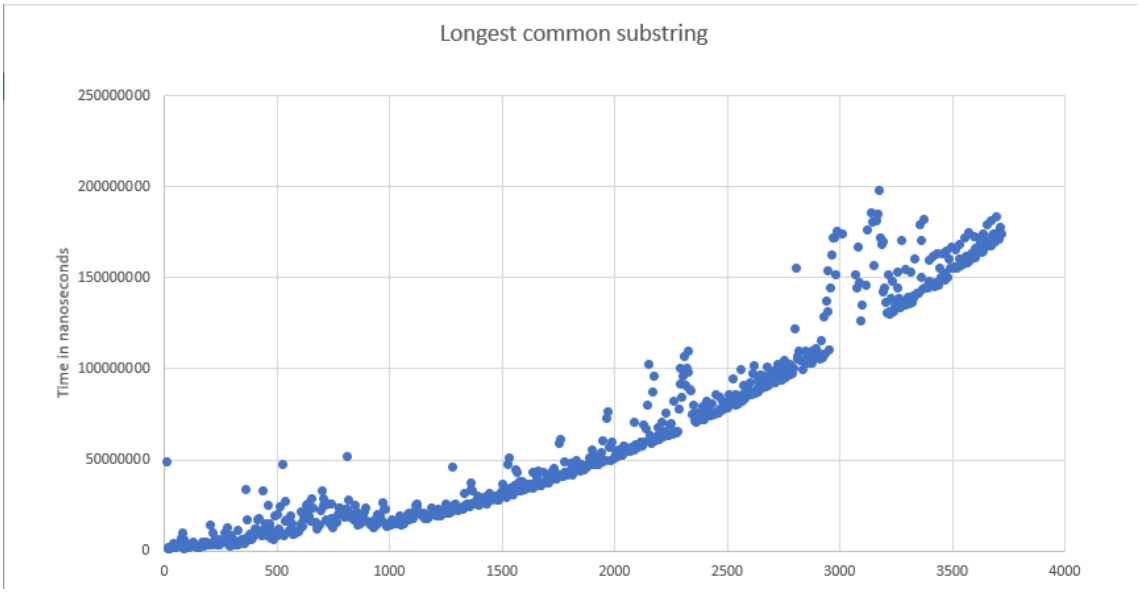

Figure 4: Grapg 2:Length of String vs time

1.4 Results

The figure shows the relation between the length of substring and the running time (in nanoseconds) of the substrng with penalty algorithm. The graph should resemble a O[n*m]curve. The best weighted approximate common substring algorithm has been designed and it's pseudo-code, running time analysis, and proof of correctness have been explained.

2 Problem 2

As part of this problem, an efficient algorithm was designed and implemented to find the most efficient number of segments in which average of y will be minimum.

2.1 Algorithm

The input to the interval based constant best approximation is a set of N points. It is required to partition the N points such the the function $f(x)$ is minimized.

$$f(x) = E + cL \tag{1}$$

In the function given above, L represents the number of line segments the N points are partitioned into, c represents the constant cost of each line segment ($c > 0$), and E represents the error function of each line segment. According to the problem statement, the error of each line segment is to be calculated based on the average value of y of each segment.

There exists a point j and a point i before j, that both fit on the same line segment and give the minimum cost. The points existing before i lie on different line segments. To find the point i before j that gives us the minimum cost, it is required to iterate through the values of i and find the minimum cost.

The recurrence equation used in this dynamic programming approach is given by

$$\text{DynamnicOptimisation[j]} = \begin{cases} 0 \text{ if j} = 0 \\ min(\text{e}_{ij} + Penalty + DynamnicOptimisation[i-1]) \text{ if } j > 0 \end{cases}$$

$DynamnicOptimisation[i-1]$ is the cost of modelling points before i to fit on line segments.

Algorithm 2: Interval based constant Approximation Algorithm

Input: Value of N number of points tobe generated

Output: Minimum error after adding partations penalties

```
1  IntervalApproximation ()
2  {
3  Point[n][2] be matrix with randonly generated x and y coordinates. for j ← 1 to n do
4      for i ← 1 to j do
5          Min (e_ij)  Where e_ij = Σ_{n=i}^{J} n^2 − Σ_{n=i}^{J} n^2/m
6          Between (point_i, ..., point_j)
7      end
8  end
9  for j ← 1 to n do
10     for i ← 1 to j do
11         DynamicOptimization[i] = min (e_ij + Penalty + DynamicOptimization[i − 1])
12     end
13 end
14 }
15 return DynamicOptimization[n]
```

2.2 Algorithm Correctness

To prove the correctness of the algorithm, proof by contradiction and proof by termination is explained in this section.

2.2.1 Proof by Contradiction

Claim: Using the below equation we are to get the most optimal segmentation with which our points can be separated taking into consideration the penalty value.

$$\text{DynamnicOptimisation[j]}= \begin{cases} 0 \text{ if j} = 0 \\ min(\text{e}_{ij} + Penalty + DynamnicOptimisation[i-1]) \text{ if } j > 0 \end{cases}$$

Proof: Let us assume K is the most optimal solution. We know from our algorithm that the possible combinations of errors is computed using $Min(e_ij)$. As we have all the possible combinations and we know the fixed penalty, so by adding the pervious error square value with penalty each time we add a segment to all the pervious points optimal value we will get a new optimal value if it is

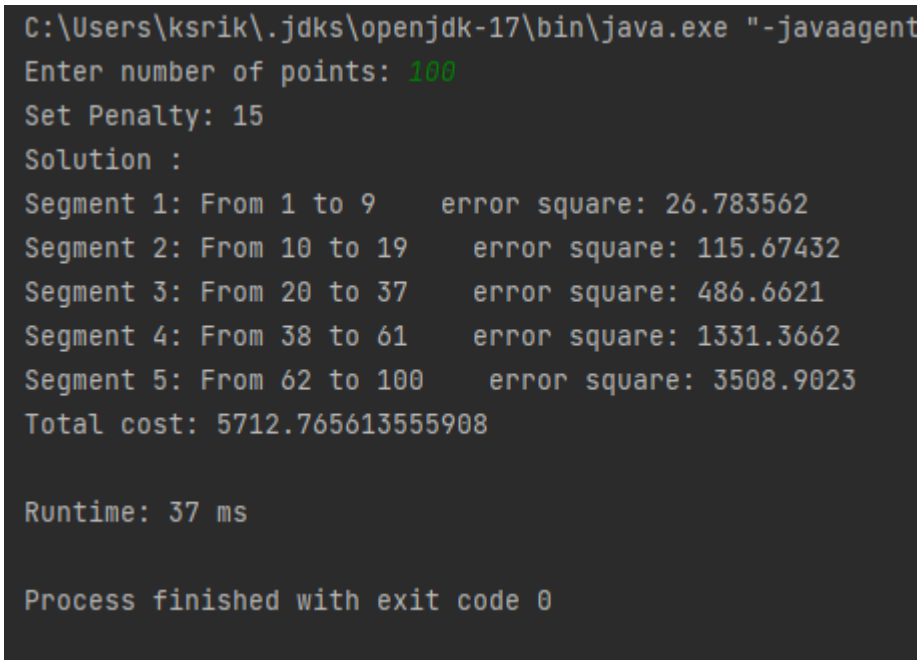

Figure 5: Output for 100 points with penalty 15

smaller. As all the points are considered to get the samllest, any other value like K cannot be smaller than the value received at $DynamicOptimisation[n]$. Thereby the K is a false value.

2.2.2 Proof of Termination

Claim: The algorithm iterates to at most $n^3 + n^2$ iterations and terminates .
Proof: The algorithm, iterates the value of i and j in a loop entering it into a matrix till j reaches n and i reaches j. Similarly the penalty is added to the optimisation problem maximum of n+1 times. Thereby, for a finite number n, the program will terminate once n+1 is reached by the optimisation function after iterating every one of the elements.

2.3 Algorithm Running Time

The errors are computed $e_{i,j}$ for each line segment in $O(n)$ time complexity per pair. To fill the array $DynamicOptimisation$ used in the dynamic programming approach, we require $O(n^2)$ time. Thus, if the errors are not precomputed our algorithm requires O(n^3 time complexity. The bottleneck of the algorithm lies in computing the errors of each line segment, if the errors were precomputed, the time complexity of the algorithm would be $O(n^2)$. The algorithm also requires space complexity of $O(n^2)$.

The figure shows the relation between the number of points and the running time of algorithm in nanoseconds. The graph resembles the running time of algorithm, $O|n^3|$. The points generated are random and produce the expected output curve.

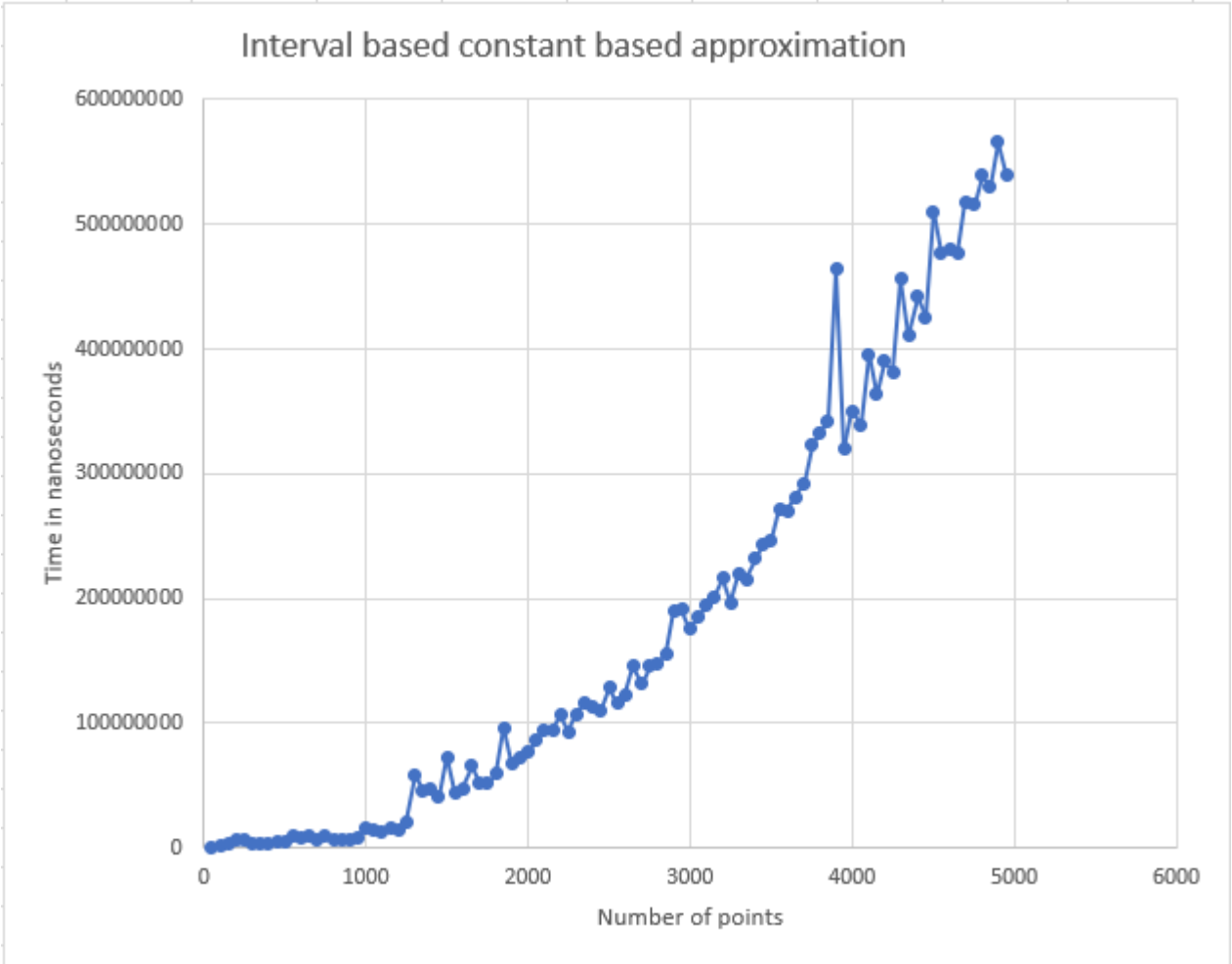

Figure 6: Graph 3: Number of points vs time.

2.4 Array vs HashMap

Experimenting with different data structures, it is found that the HashMap takes up more memory usage than the array data structure. This is reasoned from the fact that to use a one-dimensional array as a HashMap we now need to save the index as well. Also, it should be noted that in Java, HashMaps are objects and therefore obviously require more memory usage.

2.5 Result

The graph shows that the code ran as expected giving us a curve which is in accordance with what we calculated.

3 Running the Code, Testing and Evaluation

To run the code.

- Install java
- Run "q1.java"
- Run "q2.java"

The following tests were done on the code to verify its correctness:

- For question 1 the substrings were tested for test strings "ABCAABCAABBBDBCDHODSBBDB" and "ABBCAACCBBBBDBCBBBBBBBB". where result came as "13-7 substrings found as per the penalty".
- The Substring was tested for larger String and produced the same result.
- The graph showed the expected output.
- In Question 2 the we tested the output by printing out the values and plotting a graph.
- The error square calculated for each point can be printed along with the dynamicOptimal array.