Tutorial 1

Shivam Kumar Jha (17CS30033)

20-07-2019

1 Problem Statement

$A[1..m]$ and $B[1..n]$ are two 1D arrays containing m and n integers respectively, where $m \leq n$. We need to construct a sub-array $C[1..m]$ of B such that $\sum_{i=1}^{m} \big|A[i] - C[i]\big|$ is minimized.

2 Recurrences

For the given two 1D arrays, $A[1...m]$ and $B[1...n]$ with m and n elements respectively, we can write the following recursive function. The recursive function takes as arguments the last index we have reached in both the arrays traversing from right to left (denoted as i and j).

There are three cases for the recursion function to take into consideration:

1. $i = 0$: This is one of the two base cases which tells that we have reached the starting index of source array A. For this we need to put no new element in result array C. This has been given a zero weight.

2. $j < i$: This is the second base case and, if encountered, should terminate the current branch of recursion since the remaining length of source array A is greater than remaining length of B and hence no sub-sequence is possible. Similarly it has been provided a very high weight for decision making of third case.

3. The last case is when we go into a recursion and check if absolute difference of current elements and recursion with $i - 1$ and $j - 1$ or recursion with i and $j - 1$ has lower weight. Depending on the result; if former is lesser we append $B[j]$ to array C.

Below is an equation representing the above described function:

$$recurFunc(i,j) = \begin{cases} 0, & \text{if } i = 0 \\ \infty, & \text{if } j < i \\ \min\left(recurFunc(i, j-1), |A[i] - B[j]| + recurFunc(i-1, j-1)\right), & \text{otherwise} \end{cases}$$

3 Algorithm

We use Dynamic Programming and it's Bottom-Up approach in our algorithm using the above defined recursive function. We use two 2D arrays corresponding to each $A[1...i]$ and $B[1...j]$ by index $[i][j]$: $WGHTS$ to store value of the weight and $LAST$ store the last index of B present in C.

Let us say, we have the result array $C[1...i]$, a sub-sequence from $B[1...j]$ for source array $A[1...i]$. Now, we can say for sure that there are only two case from here on:

1. $C[i] = B[j]$: In this case, when we go into the recursive function, we are looking for the solution with arrays $A[1...i-1]$, $B[1...j-1]$ and $C[1...i-1]$ and, for usage in cases when this repeats, we store the weight in $WGHTS[i][j]$ and j in $LAST[i][j]$. This is basically handled by the case when the minimum is given by second argument of our min function in the above given $recurFunc$.

2. $C[i]! = B[j]$: We arrive at this case when min is obtained from the first argument. Here we ignore the current last element of B and as a repeating solution for $A[1...i-1]$, $B[1...j-1]$ and $C[1...i-1]$, store $LAST[i-1][j]$ in $LAST[i][j]$ and corresponding weight in $WGHTS[i][j]$.

NOTE: $WGHTS$ should be initialized with 0 for $i <= j$ and with ∞ otherwise.

Pseudo-Code:

Algorithm 1 BestSubArray

```
1: procedure SetWeights(A[0..m], B[0..n])
2:     for i ← 1 to m do
3:         for j ← i to n do
4:             WGHTS[i][j] ← min(|A[i] − B[j]| + WGHTS[i − 1, j − 1], WGHTS[i, j − 1])
```

4 Time and space complexities

Space Complexity: We are using arrays of max size mxn and hence the space complexity of our algorithm is $O(mn)$.

Time Complexity: We are using two for loops to traverse the arrays; one loop per array, traversing each element. We also need to traverse all mxn cells of $WGHTS$ and $LAST$. Hence the time complexity of the algorithm is $O(mn)$.