Tutorial 1

Prakhar Bindal 17CS10036

21-07-2019

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 $\sum\limits_{i=1}^{m} \big|A[i] - C[i]\big|$ is minimized.

2 Recurrences

Let us define $dp[i][j]$ where $i \leq m$ and $j \leq n$ as the minimum value of the sum of absolute differences between $A[1..i]$ and a subsequence of $B[1..j]$ of size i . The recurrences for this problem are as follows

$dp[i][j] = min(dp[i][j-1], dp[i-1][j-1] + abs(A[i] - B[j])$
where $1 \leq i \leq m$ and $i+1 \leq j \leq n$

$dp[i][i] = dp[i-1][i-1] + abs(A[i] - B[i]$

Base Cases: $dp[i][0] = 0$ where $0 \leq i \leq m$

3 Algorithm

As mentioned above let us define $dp[i][j]$ where $i \leq m$ and $j \leq n$ as the minimum value of the sum of absolute differences between $A[1..i]$ and a subsequence of $B[1..j]$ of size i. Now as we can observe that when we will be iterating over B We can either take the current element in our subsequence and pair it with the current element of A or we can just leave this element as it is and we will pair it with some next element of A. This leads to above mentioned recurrence $dp[i][j] = min(dp[i][j-1], dp[i-1][j-1] + abs(A[i] - B[j])$ where $dp[i-1][j-1] + abs(A[i] - B[j])$ denotes the value of the sum when we are pairing the current B element with the current A element and using the previous minimum cost of elements upto i-1 and j-1 . $dp[i][j-1]$ denotes the case when we are not pairing the current B value with the current A element and just taking the minimum csum of elements upto index j-1. As we need to minimize the total sum we can just update our current state with the minimum of the sum of the

two previous states and just proceed in this way. Base cases will obviously be $dp[i][0] = 0$ because when we don't have anything to take from B then the minimum sum will obviously be zero. Also $dp[i][i] = dp[i-1][i-1] + abs(A[i] - B[i])$ as the size of both the arrays is same at this point of time and hence we cannot skip any element of j and we have to pair all elements of A with all the corresponding elements of B.
Let's come to the part of restoring the subsequence with minimum sum. Now after we have executed the above algorithm then we will have hour dp table ready with all the values of minimum sum. As it is clear from above the answer will be stored in dp[n][m] which will be simply the answer to our task . But now to restore our sequence we will simply backtrack our dp table starting from dp[n][m] and see which elements are included in our subsequence which has the minimum sum of absolute differences with A. As it is clear from our recurrence relations we add elements in our subsequence only when we pair the current element of A with the current element of B i.e. $dp[i-1][j-1] + abs(A[i] - B[j])$ .This represents the case when we our included B[j] in our minimum sum subsequence. So for restoring our subsequence we will just start from $dp[m][m]$ and see that from which previous state did this value of $dp[n][m]$ came . Whenever the value of $dp[i][j]$ came from the value of $dp[i-1][j-1]$ we can just add that $B[j]$ to our subsequence and similarly we can go until the length of our subsequence doesn't become equal to m and keep on updating the values of i and j.

Pseudo Code

```
int $dp[m+1][n+1]$
for i in [1,m]
    $dp[i][0] = 0$
for i in [1,m]:
    $dp[i][i] = dp[i-1][i-1] + abs(A[i] - B[i]$
    for j in [i+1,n]:
        $dp[i][j] = min(dp[i][j-1], dp[i-1][j-1] + abs(A[i] - B[j])$
```

Restoring the subsequence

```
int res[m+1]
int indexa=m
int indexb=n
int currentsize=0
while(currentsize!=m):
    if dp[indexa][indexb]==dp[indexa-1][indexb-1]+abs(A[indexa]-B[indexb]):
            res[++currentsize]=b[indexb]
            indexa=indexa-1
            indexb=indexb-1
            continue
    indexb=indexb-1
```

```
for i in [n,1]:
    print(res[i])
```

4 Demonstration

Let us consider the sample input given to us first. Here we can see that m=3 and A=[2,7,2] , n=4 and B=[5,3,6,8]. After Running the above algorithm we can obtain the following dp table

i	j	$dp[i][j]$
1	1	3
1	2	1
1	3	1
1	4	1
2	2	7
2	3	2
2	4	2
3	3	11
3	4	8

As we can from the value of $dp[n][m]$ i.e. $dp[3][4]$ the minimum value of the sum of absolute differences is 8. Now for restoring the subsequence which leads to this minimum possible sum we can run the backtracking algorithm as described above and on running the algorithm we can get the following values

ia	ib	res
3	4	garbage
2	3	8
1	2	6
0	1	3

Here ia and ib denote the current indices of A and B which we are looking at in our backtracking algorithm. res denotes the denotes the subsequence which will be printed in reverse order. So as it can be seen here that our subsequence is 3,6,8 which will give the minimum sum of absolute differences (which is equal to 8 as is obtained from the dp table) the given Array A. This result matches with the answer given in the Assignment Question
Here is another example A=[4 2 1 5 6 4 23 123 9 21] (m=10)
B=[15 14 13 12 11 10 9 8 7 6 5 4 3 2 1] (n=15)
Running the above algorithm yields the answer as 189 as the minimum value of the sum and Subsequence of B which gives this value is given as C=[10 9 8 7 6 5 4 3 2 1] which also matches with the brute force solution of complexity of $O(m * 2^m)$ which basically generates all possible subsets of B of size m and finds the minimum value of the sum of absolute differences.
The Brute force code can be found at https://ideone.com/D8ZjMi
The Dynamic programming implementation can be found at http://ideone.com/Uf8LlD
Both of these codes are private and can be accessed only via these links

5 Time and space complexities

As it can be seen from the above Pseudo Code we are running two nested loops one going upto n and other going upto m twice. So the net time complexity can be written as $O(2 * m * n + m)$ which is equivalent to $O(m * n)$. So the time complexity of this algorithm is $O(m * n)$
Again regarding space complexity as we can see from the above pseudo code we are just using a dp table of size n*m in our extra space . So we can say that the space complexity of our algorithm is $O(m * n)$.