CS325: Analysis of Algorithms, Fall 2022

Practice Assignment 2 Solution

Problem 1.

Algorithm Description: Let $p[i].x$ and $p[i].y$ denote X and Y coordinates of $p[i]$. The algorithm outputs S, the set of all maximal points. Initially, S is empty.

(a) Sort all points based on their X coordinates. Let $p[1], p[2], ..., p[n]$ be the order of the points obtained, that is $p[1].x \leq p[2].x \leq ... \leq p[n].x$

(b) Add $p[n]$ to the set S. Let *cur* be the index of the last point added to S by the algorithm. Set, $cur = n$. Note that, this point has the largest Y-coordinate among all maximal points in S.

(c) The algorithm considers all $p[i]$'s iteratively, from $p[n-1]$ to $p[1]$.

(d) When $p[i]$ is considered, if its Y-coordinate is larger than *cur*, the algorithm adds it to S and updates *cur* to i, otherwise, the algorithm disregards $p[i]$.

Can you come up with a recursive formulation of this iterative algorithm? (it is simpler)

Pseudocode: Let $p[i].x$ and $p[i].y$ denote the value of x and y coordinate of the ith point, where $i = 1, 2, ..., n$.

```
STAIRCASE
    sort list p by their x coordinate value. (use merge sort)
    S ← ∅ as empty
    S.insert(p[n])
    cur ← n
    for i = n − 1 to 1
        if p[i].y ≥ p[cur].y
            S.insert(p[i])
            cur ← i
        end if
    end for
return S
```

Proof. We prove that after the ith iteration of the for loop, S contains the maximal points of $\{p[n-i], p[n-i+1], \ldots, p[n]\}$, which, in particular, implies that S contains all maximal points in the end. We use induction on i. The **base case** is for $i = 0$. S contains $p[n]$ the only maximal point of $\{p[n]\}$ before any iteration of the for loop.

Induction Hypothesis ensures that for any $j < i$ we have the desired property, that is after the jth iteration S contains maximal points of $\{p[n-j], p[n-j+1], \dots, p[n]\}$.

Induction step is to prove the statement for i, that is after the ith iteration S contains maximal points of $\{p[n-i], p[n-i+1], \dots, p[n]\}$. By induction hypothesis, after $i-1$ iterations S contains maximal points of $\{p[n-i+1], p[n-i+2], \dots, p[n]\}$. Note that these are also maximal points of $\{p[n-i], p[n-i+1], \dots, p[n]\}$, as $p[]$ is sorted by X-coordinates. Also, $p[i]$ is a maximal point if and only if its Y-coordinate is larger than all maximal points of $\{p[n-i+1], p[n-i+2], \dots, p[n]\}$ (Why?). This condition is checked by the algorithm. $\square$

Running time: The algorithm spends $O(n \log n)$ time to sort the points (merge sort). After that, it spends $O(1)$ time per iteration. So the total running time is $O(n + n \log n)$, which is $O(n \log n)$.

Problem 2. Let $I(i)$ be the length of the maximum increasing subsequence that ends with $X[i]$, and let $D[i]$ be the maximum decreasing subsequence that *starts* with $X[i]$. Finally, let $B(i)$ be the maximum bitonic subsequence with its maximum at $X[i]$, and let B be the maximum bitonic subsequence that we are looking for. We have:

$$B(i) = I(i) + D(i) - 1,$$

and,

$$B = \max_{1 \leq i \leq n} (B(i)).$$

(Why?) Our algorithm computes $I(i)$ and $D(i)$ for all $1 \leq i \leq n$. Then, it computes B in $O(n)$ time.

We have seen how to compute $I(i)$ (for all $1 \leq i \leq n$) in $O(n^2)$ time. Computing $D(i)$ is a symmetric problem that can be done in $O(n^2)$ time similarly. (How?) Therefore, the total running time of the algorithm is $O(n^2)$.

Problem 3. We solve this problem by dynamic programming. Lets come up with the recursive relation first. Let $S[i,j]$ be true if and only if $C[1..(i+j)]$ is a shuffle of $A[1..i]$ and $B[1..j]$. We have:

$$S(i,j) = \begin{cases} \text{False}, & \text{if } i < 0 \text{ or } j < 0 \\ \text{True}, & \text{if } i = 0 \text{ and } j = 0 \\ (S(i-1,j) \wedge (A[i] = C[i+j])) \vee & \\ (S(i,j-1) \wedge (B[j] = C[i+j])), & \text{otherwise} \end{cases}$$

Our dynamic programming will fill the boolean table S, row by row, to ensure that the required information for each i, j is available when we need them. (Can you write a pseudocode for this DP?)

Proof:

1. Base Cases: If $i = j = 0$ then the statement is trivially true. If $i < 0$ or $j < 0$ we say that $S(i,j)$ is false.

2. Inductive Hypothesis: The algorithm correctly decides if $C[1, \dots, k+\ell]$ is a shuffle of $A[1, \dots, k]$ and $B[1, \dots, \ell]$ if $k + \ell < i + j$.

3. Inductive Step: The algorithm correctly decides if $C[1, \ldots, i+j]$ is a shuffle of $A[1, \ldots, i]$ and $B[1, \ldots, j]$. There are two ways for $C[i+j]$ to be a shuffle of $A[i]$ and $B[j]$: (i) $B[j] = C[i+j]$ and $C[1, \ldots, i+j-1]$ is a shuffle of $A[1, \ldots, i]$ and $B[1, \ldots, j-1]$, or (ii) $A[i] = C[i+j]$ and $C[1, \ldots, i+j-1]$ is a shuffle of $A[1, \ldots, i-1]$ and $B[1, \ldots, j]$. The algorithm checks these two cases.

Running Time Analysis: The algorithm fills each cell of the dynamic programming table in $O(1)$ time, and there are mn cells. Thus, the running time is $O(mn)$.