CS 6363.003 Spring 2021 Official Solutions

Homework 2 Problem 1

(a) Suppose we are given two sorted arrays $A[1 .. n]$ and $B[1 .. n]$. Describe an algorithm to find the median element (the element of rank n) in the union of A and B in $O(\log n)$ time.

Solution: The procedure FINDMEDIANSORTED$(A[1 .. n], B[1 .. n])$ finds the median element of $A[1 .. n] \cup B[1 .. n]$ assuming the arrays are sorted and contain no duplicate elements.

```
FINDMEDIANSORTED(A[1 .. n], B[1 .. n]):
  if n = 1
      return min {A[1], B[1]}
  else if A[⌊n/2⌋] < B[⌊n/2⌋]
      return FINDMEDIANSORTED(A[⌊n/2⌋ + 1 .. n], B[1 .. ⌈n/2⌉])
  else
      return FINDMEDIANSORTED(A[1 .. ⌈n/2⌉], B[⌊n/2⌋ + 1 .. n])
```

If $n = 1$, then there are two elements total, and the algorithm is correct to return the smaller of them as the median. Otherwise, suppose $A[\lfloor n/2 \rfloor] < B[\lfloor n/2 \rfloor]$. Consider any element $A[i]$ where $i \leq \lfloor n/2 \rfloor$. All $\lceil n/2 \rceil$ elements from $A[\lfloor n/2 \rfloor + 1 .. n]$ and all $\lceil n/2 \rceil + 1$ elements from $B[\lfloor n/2 \rfloor .. n]$ are greater than $A[i]$, so $A[i]$ has rank at most $n - 1$ and it cannot be the median. Likewise, consider any element $B[i]$ where $i \geq \lceil n/2 \rceil + 1$. Now, all $\lfloor n/2 \rfloor$ elements of $A[1 .. \lfloor n/2 \rfloor]$ and all $\lceil n/2 \rceil$ elements of $B[1 .. \lceil n/2 \rceil]$ are less than $B[i]$, meaning it has rank strictly greater than n and cannot be the median. The median must belong to one of $A[\lfloor n/2 \rfloor + 1 .. n]$ or $B[1 .. \lceil n/2 \rceil]$. Further, because those subarrays are missing exactly $\lfloor n/2 \rfloor$ elements smaller than the median of $A[1 .. n] \cup B[1 .. n]$, it must have rank $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$, making it the median of $A[\lfloor n/2 \rfloor + 1 .. n] \cup B[1 .. \lceil n/2 \rceil]$. The recursive call FINDMEDIANSORTED$(A[\lfloor n/2 \rfloor + 1 .. n], B[1 .. \lceil n/2 \rceil])$ finds this element by induction on n and returns it. The final case of $A[\lfloor n/2 \rfloor > B[\lfloor n/2 \rfloor]$ is symmetric.

The algorithm running time follows the recurrence $T(n) = T(n/2) + 1$ which we know from binary search or recursion trees has a solution of $\mathbf{\Theta(\log n)}$.

■

Rubric: 5 points total: 2.5 points for the algorithm; 1.5 points for the justification; 1 point for running time analysis.

(b) Now suppose we are given two sorted arrays $A[1 .. m]$ and $B[1 .. n]$ with no duplicate elements and an integer k where $1 \leq k \leq m + n$. Describe an algorithm to find the kth smallest element in $A \cup B$ in $O(\log(m + n))$ time.

Solution: The procedure SELECTSORTED$(A[1 .. m], B[1 .. n], k)$ finds the element of rank k in $A[1 .. m] \cup B[1 .. n]$ assuming the arrays are sorted and contain no duplicate elements.

```
SelectSorted(A[1 .. m], B[1 .. n], k):
  if m = 1
      if k ≠ 1 and A[1] < B[k − 1]
          return B[k − 1]
      else if k > n or A[1] < B[k]
          return A[1]
      else
          return B[k]
  else if n = 1
      if k ≠ 1 and B[1] < A[k − 1]
          return A[k − 1]
      else if k > m or B[1] < A[k]
          return B[1]
      else
          return A[k]
  else if A[⌊m/2⌋] < B[⌊n/2⌋]
      if k ≤ ⌊m/2⌋ + ⌈n/2⌉
          return SelectSorted(A[1 .. m], B[1 .. ⌈n/2⌉], k)
      else
          return SelectSorted(A[⌊m/2⌋ + 1 .. m], B[1 .. n], k − ⌊m/2⌋)
  else
      if k ≤ ⌈m/2⌉ + ⌊n/2⌋
          return SelectSorted(A[1 .. ⌈m/2⌉], B[1 .. n], k)
      else
          return SelectSorted(A[1 .. m], B[⌊n/2⌋ + 1 .. n], k − ⌊n/2⌋)
```

The base case of $m = 1$ has the algorithm figure out $A[1]$'s rank in relation to $B[k-1]$ (if it exists) and $B[k]$ (if it exists) to know which of the three to return. The base case of $n = 1$ is similar. We'll assume we aren't in a base case for the rest of the argument.

Suppose $A[\lfloor m/2 \rfloor] < B[\lfloor n/2 \rfloor]$. Further suppose $k \leq \lfloor m/2 \rfloor + \lceil n/2 \rceil$. Consider any element $B[i]$ where $i \geq \lceil n/2 \rceil + 1$. All $\lfloor m/2 \rfloor$ elements of $A[1 \,..\, \lfloor m/2 \rfloor]$ and all $\lceil n/2 \rceil$ elements of $B[1 \,..\, \lceil n/2 \rceil]$ are less than $B[i]$, meaning it has rank strictly greater than $\lfloor m/2 \rfloor + \lceil n/2 \rceil$. In particular, element $B[i]$ has rank strictly greater than k, and the element of rank k appears somewhere in $A[1 \,..\, m]$ or $B[1 \,..\, \lceil n/2 \rceil]$. All elements of rank at most k appear in those arrays as well, so we want the element of rank k from those subarrays, which is returned inductively by the recursive call SelectSorted$(A[1 \,..\, m], B[1 \,..\, \lceil n/2 \rceil], k)$. Now, suppose instead that $k > \lfloor m/2 \rfloor + \lceil n/2 \rceil$. Consider any element $A[i]$ where $i \leq \lfloor m/2 \rfloor$. All $\lceil m/2 \rceil$ elements of $A[\lfloor m/2 \rfloor + 1 \,..\, m]$ and $\lceil n/2 \rceil + 1$ elements of $B[\lfloor n/2 \rfloor \,..\, n]$ are greater than $A[i]$, meaning it has rank at most $m + n - \lceil m/2 \rceil - \lceil n/2 \rceil - 1 \leq \lfloor m/2 \rfloor + \lceil n/2 \rceil$. In particular, element $A[i]$ has rank strictly less than k, and the element of rank k appears somewhere in $A[\lfloor m/2 \rfloor + 1 \,..\, m]$ or $B[1 \,..\, n]$. There are $\lfloor m/2 \rfloor$ elements of rank less than k not appearing in those subarrays, so we want the element of rank $k - \lfloor m/2 \rfloor$ relative to the subarrays, which is returned inductively by the recursive call SelectSorted$(A[\lfloor m/2 \rfloor + 1 \,..\, m], B[1 \,..\, n], k - \lfloor m/2 \rfloor)$. The cases where $A[\lfloor m/2 \rfloor > B[\lfloor n/2 \rfloor]$ is symmetric.

For running time, we observe that we spend constant time in each recursive call. One of m or n is being halved in each recursive call, so we can only recurse $\Theta(\log(m) + \log(n))$ times before we hit a base case. Finally, we observe $\log(m) + \log(n) = \log(mn) \leq \log((m+n)^2) = \Theta(\log(m+n))$. ***The running time is $\Theta(\log(m+n))$.*** ■

Rubric: 5 points total: 2.5 points for the algorithm; 1.5 points for the justification; 1 point for running time analysis.

CS 6363.003 Spring 2021 Official Solutions

Homework 2 Problem 2

Suppose you are given a set $P = \{p_1, \ldots, p_n\}$ of n points in the plane, represented by two arrays $X[1 \,..\, n]$ and $Y[1 \,..\, n]$. Specifically, point p_i has coordinates $(X[i], Y[i])$.

Describe and analyze an $O(n \log n)$ time algorithm to output the set of Pareto optimal members of P.

Solution: Per the hint, we'll use a divide-and-conquer algorithm. Similar to how we handled the closest pair problem, we'll start by dividing the input points into two equal sized sets by x-coordinate, and then compute the Pareto optimal points on the two sides. As we prove below, (a) all Pareto optimal points of the original input set P are Pareto optimal for one of these two subsets, and (b) we can easily figure out which Pareto optimal points from the subsets are also optimal for P as well in only $O(n)$ time.

The procedure ParetoOptimal($X[1 \,..\, n], Y[1 \,..\, n]$) takes as input the point set P represented with the coordinate arrays X and Y as described in the question. It outputs the array $Z[1 \,..\, k]$ containing the indices of the Pareto optimal points of P. For simplicity, we assume the points $\langle p_1, \ldots, p_n \rangle$ are sorted from left-to-right, no two points are identical, and $n \geq 1$. We can guarantee the points are sorted by running any $O(n \log n)$ time sorting algorithm before calling ParetoOptimal. We'll also guarantee the output array Z is sorted from left-to-right.

```
ParetoOptimal(X[1 .. n], Y[1 .. n]):
  if n = 1
      return ⟨1⟩
  else
      m ← ⌊n/2⌋
      Z_ℓ[1 .. k_ℓ] ← ParetoOptimal(X[1 .. m], Y[1 .. m])
      Z_r[1 .. k_r] ← ParetoOptimal(X[m+1 .. n], Y[m+1 .. n]
      ⟨⟨Z_r is 1-indexed based on the right recursive call.⟩⟩
      k ← 0

      ⟨⟨Scan for optimal points on from the left.⟩⟩
      for i ← 1 to k_ℓ
          if Y[Z_ℓ[i]] > Y[m + Z_r[1]]
              k ← k + 1
              Z[k] ← Z_ℓ[i]

      ⟨⟨Make sure left-most point on right is optimal for P.⟩⟩
      if k = 0 or X[Z_r[1]] > X[Z[k]]
          k ← k + 1
          Z[k] ← m + Z_r[1]
      ⟨⟨Remaining points on right must be optimal for P.⟩⟩
      for i ← 2 to k_r
          k ← k + 1
          Z[k] ← m + Z_r[i]

      return Z[1 .. k]
```

If $n = 1$, then the only point of P must be optimal. Now, suppose $n > 1$. Let $P_\ell = \{p_1, \ldots, p_m\}$ and $P_r = \{p_{m+1}, \ldots, p_n\}$. By induction, we find the Pareto optimal points of P_ℓ and P_r, storing their indices (1-indexed relative to just their respective subsets) in Z_ℓ and Z_r, respectively.

We claim each Pareto optimal point p_i of P must be Pareto optimal for its respective subset P_ℓ or P_r. Indeed, for any subset $P' \subseteq P$, if $p_i \in P'$ is not Pareto optimal for P', then there exists a point $p_j \in P'$, and therefore in P as well, above and to the right of p_i. Therefore, it suffices to search Z_ℓ and Z_r to compute Z.

Let $p^* = p_{m+Z_r[1]}$ be the leftmost Pareto optimal point of P_r. Observe how p^* is also a highest point of P_r; indeed, p^* acts as evidence that every point of P_r to its left is not Pareto optimal. Suppose a Pareto optimal point p_i of P_ℓ is not optimal for P. Then, there must exist a point $p_j \in P_r$ above to the right of p_i. Point p^* would also be above and to the right of p_i in that case. Therefore, every Pareto optimal point of P_ℓ not below and to the left of p^* must be optimal for P. The first for loop adds the indices of these points to Z in left-to-right order.

The next if statement checks that the right-most Pareto optimal point of P_ℓ is not above and to the right of p^* (a situation that may occur if multiple points have the median x-coordinate). If not, the index of p^* is added to Z.

Finally, every other Pareto optimal point of P_r must be Pareto optimal for P, because every point of P_ℓ lies strictly to their left. The second for loop adds the indices of these points to Z in left-to-right order before we return the entire array Z.

The procedure ParetoOptimal$(X[1 \,..\, n], Y[1 \,..\, n])$ performs two recursive calls on arrays of size approximately $n/2$, and it spends $O(n)$ time outside the recursive calls. The running time follows the common recurrence $T(n) = 2T(n/2) + O(n)$, which we know solves to $O(n \log n)$. Even accounting for sorting by x-coordinate, ***the whole algorithm takes $O(n)$ time***. ■

Rubric: 10 points total: 5 points for the algorithm; 3 points for the justification; 2 points for running time analysis. No penalty for (implicitly) assuming points have distinct x or y-coordinates.

CS 6363.003 Spring 2021 Official Solutions

Homework 2 Problem 3

Suppose we are given an array $A[1 \,..\, n]$ of numbers, which may be positive, negative, or zero, and which are ***not*** necessarily integers. We are going to design a dynamic programming algorithm that finds the largest sum of elements in a contiguous subarray $A[i \,..\, j]$.

(a) Let $maxSum(j)$ equal the largest sum of elements in a contiguous subarray of $A[1 \,..\, j]$ *whose last member is* $A[j]$.

Give a recursive definition for $maxSum(j)$.

Solution: There are two cases to consider: First, it could be the case that $A[j]$ is the *only* member of the maximum sum contiguous subarray whose last member is $A[j]$. However, assuming $j \geq 2$, it could be the case that $A[j]$ is not the only member in that subarray. In this latter case, $A[j]$ is proceeded by a contiguous subarray ending at $A[j-1]$. This prefix subarray should have the largest sum of elements of any such subarray, and by definition, that sum is $maxSum(j-1)$. We conclude,

$$maxSum(j) = \begin{cases} A[j] & \text{if } j = 1 \\ A[j] + \max\{0, maxSum(A[j-1])\} & \text{otherwise} \end{cases}.$$

■

Rubric: 3 points total: 2 points for the recurrence; 1 point for some justification.

(b) What would be the running time of a dynamic programming algorithm that computes $maxSum(j)$ for all j from 1 to n using your recursive definition?

Solution: Each subproblem depends upon at most one other, and it has a smaller parameter, so we can solve each of them in increasing parameter order. The recurrence takes only constant time to evaluate given its dependencies have been evaluated, so the running time would be $n \cdot O(1) = O(n)$. ■

Rubric: 1 point total.

(c) Describe and analyze an efficient algorithm that finds the largest sum of elements in a contiguous subarray of $A[1 \,..\, n]$.

Solution: Per the hint, we'll fill a table $maxSum[1 \,..\, n]$ with solutions to each subproblem $maxSum(j)$. Either the largest sum contiguous subarray is empty (and we should return 0) or it is the largest sum contiguous subarray ending with some element $A[j]$, and we can look at the table entries to find that largest sum.

```
LargestSum(A[1 .. n]):
  for j ← 1 to n
      if j = 1
          maxSum[j] ← A[j]
      else
          maxSum[j] ← A[j] + max{0, maxSum[j − 1]}

  largest ← 0
  for j ← 1 to n
      if maxSum[j] > largest
          largest ← maxSum[j]
  return largest
```

The algorithm just adds an $O(n)$ time for loop on top of the time to fill the memoization table, so ***the total running time is $O(n)$***. ■

Rubric: 2 points total: 1 point for filling the table, 0.5 points for returning the solution, 0.5 points for running time analysis.

(d) Now suppose in addition to $A[1 .. n]$, you are given an additional integer $X \geq 0$. Describe and analyze an algorithm that finds the largest sum of elements in a contiguous subarray of A *whose length is at most* X.

Solution: We would like to use the same recursive strategy. Unfortunately, we need a way to limit the length of the consecutive subarrays that we consider, and this limit has to change when we make a recursive call to find the best prefix subarray.

Let $maxSum2(j, x)$ equal the largest sum of elements in a contiguous subarray of length at most x whose last member is $A[j]$. As before, $A[j]$ may be the only member of the best subarray. If not, the subarray consists of one of length at most $x-1$ ending with $A[j-1]$ followed by the element $A[j]$ itself. We conclude,

$$maxSum2(j,x) = \begin{cases} A[j] & \text{if } j = 1 \text{ or } x = 1 \\ A[j] + \max\{0, maxSum2(A[j-1], x-1)\} & \text{otherwise} \end{cases}.$$

We need evaluate each $maxSum2(j, x)$ where $1 \leq j \leq n$ and $1 \leq x \leq X$, so we'll store the solutions in an array $maxSum2[1 .. n, 1 .. X]$. Entries depend only on those with strictly smaller j and x parameters, so we can fill the array from low j index to high and from low x index to high. If $X = 0$, we must return 0. Otherwise, we should return the greater of 0 and the largest sum of a subarray of length at most X ending at the best element $A[j]$, whatever that element happens to be. Here's the code:

```
LargestSumShort(A[1 .. n], X):
  for j ← 1 to n
      for x ← 1 to X
          if j = 1 or x = 1
              maxSum[j, x] ← A[j]
          else
              maxSum[j, x] ← A[j] + max{0, maxSum[j − 1, x − 1]}

  largest ← 0
  if X ≥ 1
      for j ← 1 to n
          if maxSum[j, X] > largest
              largest ← maxSum[j]
  return largest
```

We spend constant time evaluating each subproblem of which there are $O(nX)$. ***The running time is $O(nX)$.*** ■

Rubric: 4 points total: 2 points total for the recurrence; −0.5 for no justification, −0.5 for missing the base case; 1.5 points for filling the table (0 if the recurrence is very wrong); 0.5 points for the running time analysis.

+3 extra credit points for a correct *and justified* linear time algorithm that does not rely on outside sources.

CS 6363.003 Spring 2021 Official Solutions
Homework 2 Problem 4

For each of the following problems, the input consists of two arrays $X[1..k]$ and $Y[1..n]$ where $k \le n$. Describe and analyze an algorithm to find a central vertex in an arbitrary given binary tree.

(a) Describe and analyze an algorithm to decide whether X is a subsequence of Y.

Solution: We rely on the following observations: If X is a subsequence of Y, then either the last character of X is the last character of Y and the proceeding characters of X appear earlier in Y, or all characters of X appear earlier in Y. Also, a non-empty X cannot be a subsequence of an empty Y, but an empty X is a subsequence of an empty Y.

Let $isSubsequence(i, j)$ be true if and only if $X[1..i]$ is a subsequence of $Y[1..j]$. From the above observations, we see

$$isSubsequence(i,j) = \begin{cases} \textsc{True} & \text{if } i = 0 \text{ and } j = 0 \\ \textsc{False} & \text{if } i > 0 \text{ and } j = 0 \\ isSubsequence(i, j-1) & \text{if } X[i] \neq Y[j] \\ isSubsequence(i-1, j-1) \vee isSubsequence(i, j-1) & \text{otherwise} \end{cases}.$$

We need evaluate each $isSubsequence(i, j)$ where $0 \le i \le k$ and $0 \le j \le n$, so we'll store the solutions in an array $isSubsequence[1..k, 1..n]$. Entries depend only on those with a smaller j parameter, so we can fill the array from low j index to high and from low i index to high. We want to know $isSubsequence(k, n)$. Here's the code:

```
IsSubsequence(X[1..k], Y[1..n]):
  for j ← 0 to n
      for i ← 0 to k
          if i = 0 and j = 0
              isSubsequence[i, j] ← True
          else if i > 0 and j = 0
              isSubsequence[i, j] ← False
          else if X[i] ≠ Y[j]
              isSubsequence[i, j] ← isSubsequence[i, j − 1]
          else
              isSubsequence[i, j] ← isSubsequence[i − 1, j − 1] ∨ isSubsequence[i, j − 1]
return isSubsequence[k, n]
```

We spend constant time evaluating each subproblem of which there are $O(kn)$. ***The running time is $O(kn)$.*** ■

Rubric: 3 points total: 1.5 points total for the recurrence; −0.5 for no justification, −0.5 for missing the base cases; 1 point for filling the table (0 if the recurrence is very wrong); 0.5 points for the running time analysis.

+2 extra credit points for a correct *and justified* linear time algorithm that does not rely on outside sources.

(b) Suppose the input also includes a third array $C[1 .. n]$ of numbers, which may be positive, negative, or zero, where $C[i]$ is the ***cost*** of $Y[i]$. Describe and analyze an algorithm to compute the minimum cost of any occurrence of X as a subsequence of Y.

Solution: We slightly modify the previous solution. Let $minCost(i, j)$ denote the minimum cost of any occurrence of $X[1 .. i]$ as a subsequence of $Y[1 .. j]$ or ∞ if no occurrence exists. We modify the base cases of the above recursive definition of $isSubsequence$ in the obvious way. Instead of an $\vee$, we use a min when we have the option of using $Y[j]$, taking the better of the total costs for both options.

$$minCost(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ \infty & \text{if } i > 0 \text{ and } j = 0 \\ minCost(i, j-1) & \text{if } X[i] \neq Y[j] \\ \min\{C[j] + minCost(i-1, j-1), minCost(i, j-1)\} & \text{otherwise} \end{cases}.$$

The set of subproblems and their dependency structure is exactly the same as above. We want to know $minCost(k, n)$. Here's the code:

```
MinCostOccurence(X[1 .. k], Y[1 .. n]):
  for j ← 0 to n
      for i ← 0 to k
          if i = 0 and j = 0
              minCost[i, j] ← 0
          else if i > 0 and j = 0
              minCost[i, j] ← ∞
          else if X[i] ≠ Y[j]
              minCost[i, j] ← minCost[i, j − 1]
          else
          minCost[i, j] ← min{C[j] + minCost[i − 1, j − 1], minCost[i, j − 1]}
return minCost[k, n]
```

We spend constant time evaluating each subproblem of which there are $O(kn)$. ***The running time is $O(kn)$.*** ■

Rubric: 4 points total: 2 points total for the recurrence; −0.5 for no justification, −0.5 for missing the base cases; 1.5 points for filling the table (0 if the recurrence is very wrong); 0.5 points for the running time analysis.

(c) Describe and analyze an algorithm to compute the total number of (possibly overlapping) occurrences of X as a subsequence of Y.

Solution: We again modify the solution to part (a). Let $numOccurrences(i, j)$ denote the total number of occurrences of $X[1 .. i]$ as a subsequence of $Y[1 .. j]$. The empty sequence occurs once as a subsequence of the empty sequence. However, any other sequence occurs 0 times as a subsequence of the empty sequence. Finally, instead of an $\vee$, we use a $(+)$ when we have the option of using $Y[j]$, letting us count the total number of occurrences

using one or the other option.

$$numOccurrences(i,j) = \begin{cases} 1 & \text{if } i = 0 \text{ and } j = 0 \\ 0 & \text{if } i > 0 \text{ and } j = 0 \\ numOccurrences(i, j-1) & \text{if } X[i] \neq Y[j] \\ numOccurrences(i-1, j-1) + numOccurrences(i, j-1)\} & \text{otherwise} \end{cases}.$$

The set of subproblems and their dependency structure is exactly the same as before. We want to know $numOccurrences(k,n)$. Here's the code:

```
TotalOccurrences(X[1 .. k], Y[1 .. n]):
  for j ← 0 to n
      for i ← 0 to k
          if i = 0 and j = 0
              numOccurrences[i, j] ← 1
          else if i > 0 and j = 0
              numOccurrences[i, j] ← 0
          else if X[i] ≠ Y[j]
              numOccurrences[i, j] ← numOccurrences[i, j − 1]
          else
          numOccurrences[i, j] ← numOccurrences[i − 1, j − 1] + numOccurrences[i, j − 1]
return numOccurrences[k, n]
```

We spend constant time evaluating each subproblem of which there are $O(kn)$. ***The running time is $O(kn)$.*** ■

Rubric: 3 points total: 1.5 points total for the recurrence; −0.5 for no justification, −0.5 for missing the base cases; 1 point for filling the table (0 if the recurrence is very wrong); 0.5 points for the running time analysis.