CS 6363.003 Spring 2021 — Author. Yaokun Wu (yxw200031)

Homework 2 Problem 1

(a) 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 pseudo code is shown below.

```
MEDIANELEMENT(A[1 .. n], B[1 .. n]):
  x ← A[⌈n/2⌉]
  y ← B[⌈n/2⌉]
  if len(A) = 2 and len(B) = 2
      return min(max(A[1], B[1]), min(A[2], B[2]))
  if len(A) is odd and len(B) is odd
      if x < y
          median ← MEDIANELEMENT(A[⌈n/2⌉ .. n], B[0 .. ⌈n/2⌉])
      else
          median ← MEDIANELEMENT(A[0 .. ⌈n/2⌉], B[⌈n/2⌉ .. n])
  else
      if x < y
          median ← MEDIANELEMENT(A[⌈n/2⌉ + 1 .. n], B[0 .. ⌈n/2⌉])
      else
          median ← MEDIANELEMENT(A[0 .. ⌈n/2⌉], B[⌈n/2⌉ + 1 .. n])
  return median
```

Analysis:

If length of array A and B is 2, the median is the second element of sorted A and B, which is given by $min(max(A[1], B[1]), min(A[2], B[2]))$. If length of array A and B is odd, and the middle term in A is less than middle term in B, this suggests that the median element will either in the right half of A or the left half of B, otherwise the median element will be either in left half of A and right half of B. And if length of array A and B is even, and the middle term in A is less than middle term in B, this suggests that the median element will either in the right half of A (not including the middle element in A) or the left half of B, otherwise the median element will be either in left half of A and right half of B (not including the middle element in B).

Time complexity:

O(logn) because the search space is cut in half and the number of calls is logn, each of the call requires O(1) for processing. ■

(b) Describe an algorithm to find the kth smallest element in $A \cup B$ in $O(\log(m+n))$ time.

Solution: The pseudo code is shown below.

```
KTHSMALLEST(A[1 .. m], B[1 .. n], k):
  if (m > n)
      return SEARCH(0, m, k, A[1 .. m], B[1 .. n])
  else
      return SEARCH(0, n, k, B[1 .. n], A[1 .. m])
```

```
kthSmallest(left, right, k, X[1 .. max(m,n)], Y[1 .. min(m,n)]):
  i ← ⌈(left + right)/2⌉
  j ← k − i
  if left = right and j <= 0
      return X[left]
  if i >= k
      return kthSmallest(left, i, k)
  else
      if j <= len(Y)
          if Y[j] > X[i]
              if i = len(X) or Y[j] < X[i+1]
                  return Y[j]
              else
                  return Search(i+1, right, k)
          else
              if j = len(Y) or Y[j+1] > X[i]
                  return X[i]
              else
                  return Search(left, i−1, k)
      else
          return Search(i+1, right, k)
```

Analysis:

In this problem, we only consider the search space in the longer array.

Basecase:

When the left index is equal to right index meaning that the kth element is still not found in previous searching and the current element is the kth element, we return X[left].

When the middle element index i is greater than k, it means that we only need to search for the left half of X. Otherwise, if the jth element is not exist in Y, it means that we should search for the right half in X. if the jth element in Y does exist, we shall compare the Y[j] with X[i], if X[i] is the rightmost element or its next element is larger than Y[j], the kth element must be Y[j]. Otherwise, we should search for the right half of array X. On the contrary, if Y[j] is the rightmost element or its next element is larger than X[i], the kth element must be X[i]. Otherwise, we should search for the left half of array X. The algorithm either stop at the searching stage and find and return the element or at the basecase where we only need to compare four elements.

Time complexity: O(log(max(m,n))) because the search space is the 1 to max(m,n) and is cut in half, where the number of calls is logn, each of the call requires O(1) for processing.

■

CS 6363.003 Spring 2021	Author. Yaokun Wu (yxw200031)
Homework 2 Problem 2	

Describe and analyze an $O(n \log n)$ time algorithm to output the set of Pareto optimal members of P. (Any reasonable output describing these points is fine; for example, you could output an array $Z[1 .. h]$ where each element of Z is the index i of a Pareto optimal point p_i.)

Solution: The pseudo code is shown below.

```
ParetoOptimal(X[1 .. n], Y[1 .. n]):
  (Assume points come pre-sorted by x index)
  if n = 1
      return Z[1], Y[1]
  else
      XL[1 .. ⌈n/2⌉] and YL[1 .. ⌈n/2⌉] ← leftMost⌈(n/2⌉points
      ZL[1 .. h1] and MaxYL ← ParetoOptimal(XL[1 .. ⌈n/2⌉], YL[1 .. ⌈n/2⌉])
      XR[1 .. ⌈n/2⌉] and YR[1 .. ⌈n/2⌉] ← rightMost⌈n/2⌉ points
      ZR[1 .. h2] and MaxYR ← ParetoOptimal(XR[1 .. ⌈n/2⌉], YR[1 .. ⌈n/2⌉])
      MAX ← MaxYR
      for i ← 0 to h1
          if Y[ZL[i]] > MaxYR
              Z append i
              MAX ← max(MAX, Y[ZL[i]])
      return Z[1 .. h3], MAX
```

Prove that at each level, the "merging process" is correct:

Suppose the Pareto optimal function have correctly returned the Pareto optimal for the left half and the right half. And we want to select the Pareto optimal by comparing the $ZR[1 .. h2]$ and $ZL[1 .. h1]$. Every points in ZR will be a Pareto optimal in $Z[1 .. h3]$ because all the points in ZR are to the right. And we only need to loop through the ZL to find any points that are above $MaxYR$ and add them to $Z[1 .. h3]$, at the same time we need to keep a new MAX of the y coordinate. And the result will be $Z[1 .. h]$.

Suppose the points are given in sorted order by x index. In the base case when n equals to 1, we can direct return the point Z[1]. And suppose Pareto optimal function return Z correctly for k from 1 to $n-1$. Then Pareto optimal function will give the Pareto optimal points for the first half and second half. And since the "merging process" is correct. The overall algorithm is correct.

Time complexity:

O(nlogn) because at each level, it requires O(n) time to process and there are logn levels of resursive call. ■

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)

Homework 2 Problem 3

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

Solution:

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

■

(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: The running time is O(n) because we will compute the maxSum at each position exactly once, and it takes constant time for each operation. ■

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

Solution: The pseudo code is shown below. Here, I use "DP" as the array name instead of "maxSum" used in (a) and (b)

```
MAXSUM(A[1 .. n]):
  Create an array of DP[1 .. n]
  DP[1] ← A[1]
  MAX ← 0
  for i ← 2 to n
      if DP[i−1] > 0
          DP[i] ← DP[i−1] + A[i]
      else
          DP[i] ← A[i]
      MAX ← max(MAX, DP[i])
return MAX
```

Proof:

Suppose that DP[i], i from 1 to n - 1 contains the largest sum of elements in a contiguous subarray of A ended at index i. Then for i equals to n, there are two cases. If DP[i-1] is greater than 0, the DP[i] will be assigned to DP[i-1] + A[i]. Otherwise, DP[i] is assigned to A[i]. In either case, DP[i] contains the largest sum of elements. And the result is the largest one through DP[1..n], So the algorithm is correct.

Time complexity:

O(n) because each index i is calculated only once.

■

(d) 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: Let DP[i][j] represent the largest sum of elements in a contiguous subarray of size k whose last member is A[j].

```
maxSumX(A[1 .. n], X):
  Create an array of DP[1 .. n][1 .. X]
  DP[1][1] ← A[1]
  MAX ← 0
  for i ← 2 to n
      for j ← 1 to X
          if j = 1
              DP[i][j] ← A[i]
          else
              DP[i][j] ← DP[i−1][j−1] + A[i]
          MAX ← max(MAX, DP[i][j])
  return MAX
```

Proof:

Basecase:

DP[i][1] is A[i] because there is only one element. Suppose the for i from 1 to n - 1, for j from 1 to X - 1, DP[i][j] are calculated correctly. Then for DP[n][X], DP[n][X] is assigned to DP[n-1][X-1] + A[n], which gives the correct result for DP[n][X]. And the final result is the maximum value from DP[i][j] for i from 1 to n and for j from 1 to X.

Time complexity:

O(nX) because there are two for loops, one goes from 1 to n, the other goes from 1 to X. ■

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)
Homework 2 Problem 4

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

Solution: let $DP(i, j)$ equal whether $X[1 .. i-1]$ is a subsequence of $Y[1 .. j-1]$.

```
SUBSEQUENCE(X[1 .. k], Y[1 .. n]):
  Create an array DP[1 .. k+1][1 .. n+1]
  for i ← 1 to n+1
      DP[1][i] ← true
  for i ← 2 to k+1
      for j ← 1 to n+1
          if j = 1
              DP[i][1] ← false
          else
              if X[i−1] = Y[j−1]
                  DP[i][j] ←(DP[i][j−1] or DP[i−1][j−1])
              else
                  DP[i][j] ← DP[i][j−1]
  return DP[k+1][n+1]
```

Proof:

Basecase:

Dp[1][i] = true because an empty string is a subsequence of any string. Also, DP[i][1], i > 1 is false because any non empty string is not a subsequence of an empty string.

Suppose that DP[i][j], 1 < i <= k and 1 < j <= n, are calculated correctly. Then, for i = k + 1 and for j = n + 1, if DP[i][j-1] is true, DP[i][j] is obviously true. and if X[i-1] == Y[j-1] and DP[i-1][j-1] is true, then DP[i][j] is also true. This gives the correct result for DP[k+1][n+1].

Time complexity:

O(nk) because the there are two for loops in the algorithm and one goes from 1 to k+1 and one goes from 1 to n+1. ■

(b) Describe and analyze an algorithm to compute the minimum cost of any occurrence of X as a subsequence of Y.

Solution: Let DP[i][j] represent the minimum total cost of subsequence X[1 .. i-1] in Y[1 .. j-1].

$$DP(i,j) = \begin{cases} DP(i,j-1), & \text{if } X[i] != Y[j] \\ min(DP(i,j-1), DP(i-1,j-1)+C[j]), & \text{otherwise} \end{cases}$$

```
minCost(X[1 .. k], Y[1 .. n], C[1 .. n]):
  Create an array DP[1 .. k+1][1 .. n+1]
  for i ← 1 to n+1
      DP[1][i] ← 0
  for i ← 2 to k+1
      DP[i][1] ← +∞
  for i ← 2 to k+1
      for j ← 2 to n+1
          DP[i][j] ← DP[i][j−1]
          if X[i−1] = Y[j−1]
              DP[i][j] ← min(DP[i][j], DP[i−1][j−1] + C[j−1])
return DP[k+1][n+1]
```

Proof:

Basecase:

DP[1][i] is 0 because there is no cost for the subsequence. Dp[i][1], i > 1 is positive inf because there is no way for the conversion.

Suppose that DP[i][j],1 <= i <= k,1 <= j<= n are calculated correctly, I want to proof that DP[k+1][n+1] is also correct. When X[k] is not equal to Y[n], DP[k+1][n+1] should be DP[k+1][n], and when X[k] is equal to Y[n], then two cases are considered, DP[k+1][n] and DP[k][n] + C[n], the minimum of the two will give the smallest cost for DP[k+1][n+1].

Time complexity:

O(nk) because there are two nested loops in the algorithm one goes from 1 to k+1 and the other goes from 1 to n+1. ■

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

Solution: Let DP[i][j] represent the total number of occurrence of subsequence X[1 .. i-1] in Y[1 .. j-1].

$$DP(i,j) = \begin{cases} DP(i,j-1), & \text{if } X[i]! = Y[j] \\ DP(i,j-1) + DP(i-1,j-1), & \text{otherwise} \end{cases}$$

```
minCost(X[1 .. k], Y[1 .. n]):
  Create an array DP[1 .. k+1][1 .. n+1]
  for i ← 1 to n+1
      DP[1][i] ← 1
  for i ← 2 to k+1
      DP[i][1] ← 0
  for i ← 2 to k+1
      for j ← 2 to n+1
          DP[i][j] ← DP[i][j−1]
          if X[i−1] = Y[j−1]
              DP[i][j] ← DP[i][j] + DP[i−1][j−1])
return DP[k+1][n+1]
```

Proof:

Basecase:

DP[1][i] is 1 because an empty string is a subsequence of an non empty string.

DP[i][1], i > 1 is 0 because there is no subsequence existed in an empty string. Suppose that DP[i][j], 1 <= i <= k, 1 <= j<= n is calculated correctly, I want to proof that DP[k+1][n+1] is also correct. When X[k] is not equal to Y[n], DP[k+1][n+1] should be DP[k+1][n], and when X[k] is equal to Y[n], there are two possible cases and should be added together because there is no overlap between the two cases, the sum of the two will give the total number of occurrence for DP[k+1][n+1].

Time complexity:

O(nk) because there are two nested loops in the algorithm one goes from 1 to k+1 and the other goes from 1 to n+1.

■