Solutions
Homework #2
Introduction to Algorithms/Algorithms 1
600.433/633
Spring 2018

Due on: Thursday, February 15th, 5.00pm
Late submissions: will NOT be accepted
Format: Please start each problem on a new page.
Where to submit: Gradescope.

Please type your answers; handwritten assignments will not be accepted.
To get full credit, your answers must be explained clearly,
with enough details and rigorous proofs.

February 21, 2018

1 Problem 1 (10 points)

You are given one unsorted integer array A of size n. You know that A is almost sorted, that is it contains at most m pairs of indices (i, j) such that $i < j$ and $A[i] > A[j]$. To sort array A you applied algorithm Insertion Sort. Prove that it will take at most $O(n + m)$ steps.

Solution:
To prove this fact, consider the pseudocode for insertion sort.

for $j := 2$ *to length[A]* **do**
$key := A[j]$
$i := j - 1$
while $i > 0$ *and* $A[i] > A[i+1]$ **do**
$A[i+1] := A[i]$
end
$A[i+1] := key$
end

Algorithm 1: Insertion-Sort

Everything except the while loop requires $\Theta(n)$ time. We now observe that every iteration of the while loop can be thought of as swapping an adjacent pair of outof-order elements $A[i]$ and $A[i+1]$. Such a swap decreases the number of inversions (pairs of indices (i, j) such that $i < j$ and $A[i] > A[j]$) in A by exactly one since $(i, i+1)$ will no longer be an inversion and the other inversions are not affected. Since there is no other means of increasing or decreasing the number of inversions of A, we see that the total number of iterations of the while loop over the entire course of the algorithm must be equal to m.

2 Problem 2 (10 points)

Given two unsorted integers arrays of size n, A and B, where A has no repeated elements and B has no repeated elements, give an algorithm that finds k-th smallest entry of their intersection $A \cap B$. For full credit, you need to provide an algorithm that runs in $O(nlogn)$ time with correctness proof and running time analysis.

Solution:

Let's first sort each array. Using merge sort we can do it in $O(n \log n)$ time. Now when both A and B are sorted we will find k-th smallest element of their intersection using algorithm 1 (see below).

We will maintain two indexes i and j like in merge procedure, and one counter s which will count number of elements from the intersection $A \cap B$ seen so far. When s changes from $k-1$ to k we output current item.

$s := 0$ — number of elements in $A[1:i] \cap B[1:j]$;
$i, j := 1$ — indixes like in merge procedure;
while $i \le n$ *and* $j \le n$ **do**
  if $A[i] < B[j]$ **then**
    $i := i + 1$;
  end
  if $A[i] > B[j]$ **then**
    $j := j + 1$;
  end
  if $A[i] = B[j]$ **then**
    $s := s + 1$;
    if $s = k$ **then**
      return $A[i]$;
    end
    $i := i + 1$;
    $j := j + 1$;
  end
end
return NULL — intersection of A and B has less than k elements;

Algorithm 2:

Correctness proof:

Loop invariant:

$$|A[1:i-1] \cap B[1:j-1]| = s \tag{1}$$

$$\begin{matrix} \forall c_1 \in A[1:i-1] \cup B[1:j-1] \\ \forall c_2 \in A[i:n] \cup B[j:n] \end{matrix} \quad : \quad c_1 < c_2. \tag{2}$$

Initialization:

Before loop starts we have $i = 1$ and $j = 1$ both statements for the loop invariant hold because $A[1:i-1] = B[1:j-1] = \emptyset$

Maintenance:

Induction hypothesis (IH): suppose both statements for the loop invariant hold after iteratation number l.
Induction step (IS): let's prove it will still hold after iteration number $l+1$. we

have three options in our pseudocode.

1. $A[i] < B[j]$.
   If this is true, then we can conclude that $\forall j' > j: \; A[i] < B[j] < B[j']$ and $\forall i' > i: \; A[i] < A[i']$. second statement of the loop invariant holds.
   As long as $\forall j' > j-1: \; A[i] < B[j']$ (look above) and $\forall j' \le j-1: \; A[i] > B[j']$ (accoring to IH) we can conclude that $A[i] \notin B$, thus $|A[1:i] \cap B[1:j-1]| = |A[1:i-1] \cap B[1:j-1]| = s$. Therefore first statement of the loop invariant holds as well.

2. $A[i] > B[j]$.
   Similar proof (look above).

3. $A[i] = B[j]$.
   If this is true, then we can conclude that $\forall j' > j: \; A[i] = B[j] < B[j']$ (array B is sorted) and $\forall i' > i: \; B[j] = A[i] < A[i']$ (array A is sorted), thus second statement of the loop invariant holds.
   As long as we add to both both sets the same element, we can conclude, that $|A[1:i] \cap B[1:j]| = |A[1:i-1] \cap B[1:j-1]| + 1 = s+1$. Therefore first statement of the loop invariant holds.

Termination:

The algorithm stops when s changes from $k-1$ to k and outputs last considered item, we will call it x. First of all x belongs to both A and B, as long as we change s only when meet such i, j that $A[i] = B[j]$. According to loop invariant: $A[1:i] \cap B[1:j]$ contains all k smallest items of intersection $A \cap B$. x is the largest according to the second statement of the loop invarian, thus x is the k-th smallest item in $A \cap B$.

Running time analysis:

To sort both arrays we used $O(n \log n)$ time, to find k-th smallest element we will make at most $2n$ iterations of the loop, like in merge procedure. Thus total time complexity of the algorithm is $O(n \log n) + O(n) = O(n \log n)$

3 Problem 3 (10 points)

An array of numbers A is almost sorted if for every $1 \le i \le \sqrt{n} \le j$, we have $A[i] \le A[j]$ and for every $\sqrt{n} \le j \le k \le n$ we have $A[j] \le A[k]$. Give an algorithm that takes as input an almost sorted array A and sorts A in $o(n)$ time.

Solution:

It suffices to only sort the first $\sqrt{n}$ entries of A, which can be done in $O(\sqrt{n}\log\sqrt{n})$ time using mergesort. This runtime is $o(n)$. The correctness of the algorithm follows from the fact that by assumption A is sorted except for its first $\sqrt{n}$ elements, so sorting the first $\sqrt{n}$ elements suffices to make A fully sorted.

4 Problem 4 (10 points)

A sequence $a_1, a_2, \ldots, a_n$ has a dominant element if more than half of the elements in the sequence are the same. For example, 3 is a dominant element in the sequence $7, 3, 3, 3, 1, 3, 3, 4, 5, 3$. On the other hand, the sequence $5, 4, 1, 1, 2, 3, 2, 3, 6$ has no dominant element. Give a divide and conquer algorithm that runs in time $O(n \log n)$ and returns a dominant element in a sequence of n numbers or returns $None$ if no such element exists. Prove the correctness of your algorithm and prove that its runtime is $O(n \log n)$. (Note: there exists an $O(n)$ algorithm to solve this problem that doesnt make use of divide and conquer if you figure it out, you may prove its correctness and runtime instead.)

Solution:

The intuitive solution is to apply divide and conquer as we have seen before split the list (call it A) into two halves, say L and R. If the list had a dominant element, then it must also be the dominant element in at least one of these two lists. Assuming we can check each list for a dominant element, we can perform the merge step of the divide and conquer algorithm by checking if the dominant element of L (assuming it exists) appears enough times in R to be a dominant element of A and checking if the dominant element of R appears enough times in L to be a dominant element of A (note that the pigeonhole principle implies that at most one of these two conditions can hold true). This checking operation requires $O(n)$ time. A divide and conquer approach would then have a recurrence pattern essentially the same as mergesort, and would this require time $O(nlogn)$. We can do better using Moores voting algorithm, in which we make a single linear pass through the list counting how many times two consecutive elements have the same value to find a candidate element, then makes a second pass to verify whether or not the candidate element is actually a dominant element.

5 Problem 5 (13 points)

We will say that pivot provides $x|n-x$ separation if x elements in array are smaller than the pivot, and $n-x$ elements are larger than the pivot.

Suppose Bob knows the secret way to find a good pivot with $\frac{n}{3}|\frac{2n}{3}$ separation in constant time. But at the same time Alice knows her own secret technique, which provides separation $\frac{n}{4}|\frac{3n}{4}$, her technique also works in constant time.

Alice and Bob applied their secret techniques as subroutine in QuickSort algorithm. Whose algorithm works **asymptotically** faster? Prove your statement.

Solution:

To reorder the elements around the pivot at each step takes $O(n)$, which gives us the relation $T_B(n) = T_B(\frac{n}{3}) + T_B(\frac{2n}{3}) + O(n)$ for Bob and $T_A(n) = T_A(\frac{n}{3}) + T_A(\frac{2n}{3}) + O(n)$ for Alice.
We know that in case of separation $\frac{n}{2}|\frac{n}{2}$ we have recurrence: $T_C(n) = 2T_C(\frac{n}{2}) + O(n)$, from merge sort procedure we know that $T_C(n) = \Theta(n \log n)$. We will use it as initial guess for $T_A(n)$ and $T_B(n)$ and will prove it by substitution.

1. (a) For large enough C_B, n_0 and $\forall n \geq n_0$, $T_B(n) \leq C_B n \log n$.
   BC: Trivial.
   IH: $\forall k < n: \quad T_B(k) \leq C_B k \log k$.
   IS: $T_B(n) = T_B(\frac{n}{3}) + T_B(\frac{2n}{3}) + Cn \leq C_B \frac{n}{3}\log(\frac{n}{3}) + C_B\frac{2n}{3}\log(\frac{2n}{3}) + Cn \leq C_B n \log n - (C_B\frac{\log 3}{3} + C_B\frac{2\log\frac{3}{2}}{3} - C)n \leq C_B n \log n$. Where last inequality holds for $C_B \geq \frac{3C}{\log 3 + \log\frac{3}{2}}$.
   Therefore $T_B(n) = O(n \log n)$.

   (b) For small enough positive C_B and large enought n_0 and $\forall n \geq n_0$, $T_B(n) \geq C_B n \log n$. Using same idea as below. We will show only induction step. $T_B(n) = T_B(\frac{n}{3}) + T_B(\frac{2n}{3}) + Cn \geq C_B\frac{n}{3}\log(\frac{n}{3}) + C_B\frac{2n}{3}\log(\frac{2n}{3}) + Cn = C_B n \log n + (C - C_B\frac{\log 3}{3} - C_B\frac{2\log\frac{3}{2}}{3})n \geq C_B n \log n$, where last inequality holds for $C \geq \frac{3C}{\log 3 + \log\frac{3}{2}}$. Therefore $T_B(n) = \Omega(n \log n)$.

2. (a) For large enough C_A, n_0 and $\forall n \geq n_0$, $T_A(n) \leq C_A n \log n$. Same idea as for $T_B(n)$ we will just show induction step.$T_A(n) = T_A(\frac{n}{4}) + T_A(\frac{3n}{4}) + Cn \leq C_A\frac{n}{4}\log(\frac{n}{4}) + C_A\frac{3n}{4}\log(\frac{3n}{4}) + Cn \leq C_A n \log n - (\frac{1}{4}C_A \log 4 + \frac{3}{4}C_A \log\frac{4}{3} - C)n \leq C_A n \log n$, where last inequality holds when $C_A \geq \frac{4C}{\log 4 + 3\log\frac{4}{3}}$. Therefore $T_B(n) = O(n \log n)$.

   (b) For small enough positive C_A and large enough n_0 and $\forall n \geq n_0$, $T_A(n) \geq C_A n \log n$. Same idea as for $T_B(n)$ we will just show induction step. $T_A(n) = T_A(\frac{n}{4}) + T_A(\frac{3n}{4}) + Cn \geq C_A\frac{n}{4}\log(\frac{n}{4}) +$

$C_A \frac{3n}{4} \log(\frac{3n}{4}) + Cn = C_A n \log n + (C - \frac{1}{4} C_A \log 4 - \frac{3}{4} C_A \log \frac{4}{3}) n$, where last inequality holds when $C_A \leq \frac{4C}{\log 4 + 3 \log \frac{4}{3}}$. Therefore $T_B(n) = \Omega(n \log n)$.

Therefore $T_B(n) = \Theta(n \log n) = T_A(n)$. Asymptotically the solutions are the same.