Solutions
Homework #2
Introduction to Algorithms/Algorithms 1
600.363/463
Spring 2016

Due on: Thursday, February 11th, 5pm
Late submissions: will NOT be accepted
Format: Please start each problem on a new page.
Where to submit: On blackboard, under student assessment
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 13, 2016

1 Problem 1 (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, your algorithm must run in $O(n \log n)$

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] ∩ B[1 : j];
i, j := 1 — indixes like in merge procedure;
while i ≤ n and j ≤ 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 1:

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 $\forall j' \leq j - 1 : \ A[i] > B[j']$ (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 $O(n \log n) + O(n) = O(n \log n)$

2 Problem 2 (15 points)

Given two sorted integers arrays of size n, A and B, give an efficient algorithm that finds k-th smallest entry of their union $A \cup B$. For full credit, your algorithm must run in $O(\log n)$.

Solution:
We will use recursive algorithm which is based on the idea of binary search.
Our algorithm will use next property of median for arrays (not sets!):

$$\min\left(med(A), med(B)\right) \leq med(A \cup B) \leq \max\left(med(A), med(B)\right)$$

Consider more general problem: we are given two sorted arrays A and B of sizes l_A and l_B correspondingly, and we need to find k-th smallest element of their union. If $k > \frac{l_A+l_B}{2}$, than target element is larger than the median of the union, thus we can safely discard any elements which are less than $\min\left(med(A), med(B)\right)$. Suppose $med(A) < med(B)$, than we can discard left half of the array $A[1 : l_A/2]$. Similarly for $k < \frac{l_A+l_B}{2}$, we can safely discard any elements which are larger than $\max\left(med(A), med(B)\right)$.
After discarding one half of one of the arrays, we feed the result to the same algorithm recursivly. The only thing we shouldn't forget is to update k, when discarding elements which are smaller than k.

Base case is when one of the arrays has only one element. Without loss of generality let $l_A = 1$, then we need to median of array of size three $[A[1], B[k-1], B[k]]$, it can be done in constant time.

```
Procedure algo(A, B, k)
1   | l_A := length(A)
2   | l_B := length(B)
3   | if l_A = 1 then
4   |   | return basecase(B, A[1], k)
    | end
5   | if l_B = 1 then
6   |   | return basecase(A, B[1], k)
    | end
7   | if k > (l_A + l_B)/2 then
8   |   | if A[l_A/2] > B[l_B/2] then
9   |   |   | k := k − l_B/2
10  |   |   | return algo(A, B[l_B/2 : l_B], k)
    |   | end
11  |   | if A[l_A/2] < B[l_B/2] then
12  |   |   | k := k − l_A/2
13  |   |   | return algo(A[l_A/2 : l_A], B, k)
    |   | end
    | end
14  | if k < (l_A + l_B)/2 then
15  |   | if A[l_A/2] > B[l_B/2] then
16  |   |   | return algo(A[1 : l_A/2], B, k)
    |   | end
17  |   | if A[l_A/2] < B[l_B/2] then
18  |   |   | return algo(A, B[1 : l_B/2], k)
    |   | end
    | end
Procedure basecase(C, d, k)
19  | if C[k] < d then
20  |   | return C[k];
    | else
21  |   | if C[k − 1] < d then
22  |   |   | return d;
    |   | else
23  |   |   | return C[k − 1]
    |   | end
    | end
```

Algorithm 2:

Correctness proof
BC: It's easy to check that basecase procedure is correct. Thus algorithm outputs correct answers for basecases $(l_A, 1)$ and $(1, l_B)$.
IH: Suppose our algorithm returns correct answer for input arrays' lengths $l'_A/2$ and l_B and for input arrays' lengts l'_A and $l_B/2$.
IS: We want to prove that it will also output correct answer for input arrays of lengths l_A and l_B.

If $k > \frac{l_A+l_B}{2}$, then $(A \cup B)[k] > \min(med(A), med(B))$, thus as it was written in the beginning of the solution we can discard left half of array with smaller median. If $med(A) < med(B)$ then k-th element of $A \cup B$ is $(k - l_A/2)$-th element of $A[l_A/2 : l_A] \cup B$, otherwise k-th element of $A \cup B$ is $(k - l_B/2)$-th element of $A \cup B[l_B/2 : l_B]$. But in both cases we call algo routine with input arrays sizes either $(l_A/2, l_B)$ or $(l_A, l_B/2)$, and according to IH in both of them return correct answer.

If $k < \frac{l_A+l_B}{2}$, then $(A \cup B)[k] < \max(med(A), med(B))$, thus we can discard right half of array with larger median. If $med(A) > med(B)$ then k-th element of $A \cup B$ is k-th element of $A[1 : l_A/2] \cup B$, otherwise k-th element of $A \cup B$ is k-th element of $A \cup B[1 : l_B/2]$. But in both cases we call algo routine with input arrays sizes either $(l_A/2, l_B)$ or $(l_A, l_B/2)$, and according to IH in both of them return correct answer.

Thus by induction our algorithm return correct answer.

Running time analysis
Now let's prove that time complexity of our algorithm is $O(n \log n)$. On each step we get rid of at half of one array, and spend $O(1)$ time during each step, thus our recurrence is:

$$T(l_A, l_B) = \begin{cases} T(l_A/2, l_B) \\ T(l_A, l_B/2) \end{cases} + O(1)$$

Base case is also constant: $T(1, l_B) = T(l_A, 1) = O(1)$. In total we can make at most $2 \log n$ steps. Thus our complexity is $O(\log n)$.

3 Problem 3 (10 points)

You are given one unsorted integers array of $A = \{a_i\}_{i=1}^n$ of size n. Provide an algorithm that finds

$$r = \max_{i,j \le n} |a_i - a_j|$$

using at most $O(n)$ comparisons on the worst case input (5 points) or an algorithm which uses at most $\frac{3}{2}n$ comparisons on the worst case input (10 points).

Solution:
To find $r = \max_{i,j \leq n} |a_i - a_j| = \max_{i \leq n} a_i - \min_{i \leq n} a_i = a_{max} - a_{min}$ we need to find maximum a_{max} and minimum a_{min} values of array A. We will give an algorithm which finds them using less than $\frac{3}{2}n$ comparisons. For simplicity let n be even, for odd n same idea works.
Initialize two arrays m and M of size $n/2$. For all $0 \leq i < n/2$ do the next: compare a_{2i} and a_{2i+1} and store $\max(a_{2i}, a_{2i+1})$ as $M[i]$ and $\min(a_{2i}, a_{2i+1})$ as $m[i]$. To do this we need to spend $n/2$ comparisons. Now we can find a_{min} in array m using $n/2$ copmarisons, and a_{max} in array M using again $n/2$ comparisons. So totally we made $\frac{3}{2}n$ comparisons.

Correctness follows from the fact that maximum among "local" maximums is global maximum and the same fact for minimums.

4 Problem 4 (15 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 inversions, where inversion is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$.

1. 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 3: 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 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. What is a maximum possible number of inversions in the integer array of size n?

   Solution:
   Maximum number of inversion is equal to $\frac{n(n-1)}{2}$. It happens if every pair of items is an inversion, in total we have $\frac{n(n-1)}{2}$ pairs. It happens when we array is inverse sorted.