Homework #2
Introduction to Algorithms/Algorithms 1
600.463
Spring 2016

Submitted By: Sindhuula Selvaraju
JHED ID:sselvar4

February 12, 2016

NOTE:I have assumed that all arrays begin with index 1.

1 Problem 1 (10 points)

Given two unsorted integer arrays, A and B, of size n, 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 $O(n \log n)$ time with correctness proof and running time analysis.
ANSWER:
Given two unsorted integer arrays, A and B, of size n, where A has no repeated elements and B has no repeated elements, the algorithm that finds k-th smallest entry of their intersection, $A \cap B$ is:

1. Sort the two arrays using merge-sort
2. Search for each element of A in B.
3. If an element is found add it to $A \cap B$
4. Return the value of $A \cap B[k]$ i.e. the k^{th} element of the array if $k < sizeof A \cap B$
5. Else return error

Pseudo-Code: Assume mergeSort and BinarySearch functions have already been implemented.

Algorithm 1: Finding k^{th} smallest element in $A \cap B$

```
   Input  : Two unsorted arrays A and B
   Output: k^th smallest element in A ∩ B
1  function smallestK (A[], B[], k)
2  { if k≤ length of A then
3      A = mergeSort(A, p, r);
4      B = mergeSort(B, p, r);
5      i = 1;
6      while i ≤ n do
7          if binarySearch(A[i], B)==true then
8              Add A[i] to A ∩ B;
9          else
10             continue;
11         end
12         i = i + 1;
13     end
14     if k ≤ size of(A ∩ B) then
15         return A ∩ B[k];
16     else
17         return error;
18     end
19 else
20     return error;
21 end
22 }
```

Correctness Proof:

Initialization :
We start with the unsorted arrays A and B and before any other operation we check that k does not exceed the length of the arrays and return error if it does.

Maintenance :
The two arrays are sorted using mergeSort which does not return any direct result for the k^{th} element. it simply sorts the arrays.
The while loop takes care of finding $A \cap B$ by using binarySearch to find all elements of A in B.
The k^{th} element is then returned from the $A \cap B$, so if k is greater than the intersection size it will give an error.

Termination :
This proves that our algorithm will work for all correct inputs of A, B and k and will return the k^{th} smallest element if it exists.

Running Time Analysis:

1. Sorting Array A = O($n \log n$)
2. Sorting Array B = O($n \log n$)
3. Searching for 1 element of A in B = O($\log n$)
4. Searching for n elements of A in B = O($n \log n$)
5. Adding the elements to $A \cap B$=O(n) (if both arrays are identical)
6. Returning the k^{th} element = O(1)
7. $\implies Total = O(\text{nlog}\, n)$ + O($n \log n$) + O($n \log n$) +O(n)+O(1)
8. $\implies Total = 3O(\text{nlog}\, n)$+O($n$)+O(1)
9. $\implies Total = O(\text{nlog}\, n)$

Hence, the total time taken = O($n \log n$)+ O(1) = O($n \log n$)

2 Problem 2 (15 points)

Given two sorted integer arrays, A and B, of size n, give an efficient algorithm that finds k-th smallest entry of their union, $A \cup B$. For full credit, you need to provide an algorithm that runs in $O(\log n)$ time with correctness proof and running time analysis.
ANSWER:
Given two sorted integer arrays, A and B, of size n, an efficient algorithm that finds k-th smallest entry of their union, $A \cup B$ is: The first k element in $A \cup B$ will have first i elements of A and first j elements of B where $i + j = k$

1. Let us first start with $i = j = \lfloor \frac{k}{2} \rfloor$. In this case $i + j = k$.
2. If $A[i] < B[j]$ there might be a $A[i + 1] < B[j]$ and so on.
3. So we consider the right half of A and left half of B for further calculations. i.e. A is reduced to range from $A[i + 1]$ to $A[n]$ and B from $B[1]$ to $B[j]$
4. Else if $A[i] > B[j]$ there might be a $A[i] > B[j + 1]$ and so on.
5. So we consider the right half of B and left half of A for further calculations. i.e. A is reduced to range from $A[1]$ to $A[i]$ and B from $B[j + 1]$ to $B[n]$
6. NOTE: We have now reduced the size of our total search space from $2n$ to n
7. Repeat steps 1 to 5 with $k = k/2$ till the size of A and B is reduced to 1
8. Return the smaller number from what is left in A and B. i.e.
   if $A[1] > B[1]$ return $B[1]$
   else return $A[1]$

Pseudo-Code: Assume that A and B are sorted arrays.

Algorithm 2: Finding k^{th} smallest element in $A \cup B$

Input : Two sorted arrays A and B
Output: k^{th} smallest element in $A \cup B$

```
1  smallestK(A[], B[], k)
2  {
3  if k≤ sum length of A and B then
4  |  checkForK(A[], B[], k)
5  else
6  |  return error;
7  end
8  }
9  checkForK(A[], B[], k)
10 {
11 if (sizeof(A) == 1)(sizeof(B) == 1) then
12 |  if A[1] < B[1] then
13 |  |  return A;
14 |  else
15 |  |  return B;
16 |  end
17 else
18 |  i = ⌊k/2⌋;
19 |  j = ⌊k/2⌋;
20 |  if A[i] < B[j] then
21 |  |  A = A[i + 1] to A[sizeof(A)];
22 |  |  B = B[1] to B[j];
23 |  else
24 |  |  B = B[j + 1] to B[sizeof(B)];
25 |  |  A = A[1] to A[i];
26 |  end
27 |  checkForK(A, B, k/2)
28 end
29 }
```

Correctness Proof:

Initialization :
We start with the sorted arrays A and B. Before any other operation we check if k$\leq$ sum length of A and B. If it is not true then we return an error and terminate.

Maintenance :
We check if both arrays are of length 1 and return the smaller of the 2 if they are of length 1.
We take i and j as index for A and B such that they are $\lfloor \frac{k}{2} \rfloor$
We then take the parts of each array which we think maybe the k smallest element based on the condition.
We keep reducing the array till each array has only 1 element left.
Since both arrays have equal length and the size of the total search space keeps reducing from $2n$ to n to $\frac{n}{2}$... it will definitely reach a point where it will have a size of 2. In this case we return the element that is smaller of the two.

Termination :
This proves that our algorithm will work for all correct inputs of A, B and k and will return the k^{th} smallest element if it exists.

Running Time Analysis: As after every iteration the size of the array reduces by half:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\frac{n}{2}) + O(1) & \text{if } n > 1 \end{cases}$$

So we get:

T(n) = T($\frac{n}{2}$)+1
T(n) = T($\frac{n}{4}$)+2
T(n) = T($\frac{n}{8}$)+3
.
.
.
T(n) = T($\frac{n}{2^k}$)+k
Setting $\frac{n}{2^k}$ = 1 gives T(n) = $\log n + 1$ = O($\log n$).

3 Problem 3 (10 points)

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

$$r = \max_{1 \le i,j \le n} |a_i - a_j|$$

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). Correctness proof and running time analysis are required in both cases.
ANSWER:
To find

$$r = \max_{1 \le i,j \le n} |a_i - a_j|$$

in an unsorted integer array A using at most $\frac{3}{2}n$ comparisons on the worst case input we can:

1. If there are odd number of elements in the array take the first element as the min and max
2. If there are even number of elements in the array compare the first 2 elements and take the lower valued element as min and the other as max
3. For each subsequent pair of elements:
   (a) Compare the elements with each other
   (b) Compare the smaller element with the value of min and if it is smaller then make min = element
   (c) Compare the larger element with the value of mas and if it is larger then make max = element
4. After all pairs have been compared, we now have the value of the minimum and maximum element in the array.
5. Subtract the two values to get

$$r = \max_{1 \le i,j \le n} |a_i - a_j|$$

.

Pseudo-code:

Algorithm 3: Finding $r = \max_{1 \leq i,j \leq n} |a_i - a_j|$ for unsorted array A

```
   Input  : Unsorted array A
   Output: r = max_{1≤i,j≤n} |a_i − a_j|
1  findMaxR(A[]) { if (sizeof(A) ≤ 1) then
2  |  return error;
3  else
4  |  if (sizeof(A)%2 == 0) then
5  |  |  if A[1] > A[2] then
6  |  |  |  min = A[2];
7  |  |  |  max = A[1];
8  |  |  else
9  |  |  |  min = A[1];
10 |  |  |  max = A[2];
11 |  |  end
12 |  |  i = 3;
13 |  else
14 |  |  min = A[1];
15 |  |  max = A[1];
16 |  |  i = 2;
17 |  end
18 |  while i < sizeof(A) do
19 |  |  if A[i] < A[i + 1] then
20 |  |  |  if A[i] < min then
21 |  |  |  |  min = A[i];
22 |  |  |  else
23 |  |  |  |  if A[i + 1] > max then
24 |  |  |  |  |  max = A[i + 1];
25 |  |  |  |  else
26 |  |  |  end
27 |  |  else
28 |  |  |  if A[i + 1] < min then
29 |  |  |  |  min = A[i + 1];
30 |  |  |  else
31 |  |  |  if A[i] > max then
32 |  |  |  |  max = A[i];
33 |  |  |  else
34 |  |  end
35 |  |  i = i + 2
36 |  end
37 |  return max − min;
38 end
39 }
```

Correctness Proof:

Initialization :
We start with the unsorted array A. Before any other operation we check if the length of A is less than 1. If it is then we return an error and terminate.

Maintenance :
Since we have checked that the array has minimum 2 elements we now start with the main algorithm.
For an odd sized array initializing min and max with the first element ensures that there are even number of elements left. So now we can unify our algorithm for odd and even sized arrays.
Comparing each pair of elements with each other and then with the values of min and max reduces the number of comparisons required. For example if we had checked each array element against min and max separately we would have made $2n$ comparisons.
When checking pairwise we first find the local minimum and maximum values and compare them with the global maximum and minimum values
The final values of min and max are the values of the minimum and maximum values in the array.
Taking the difference between the 2 will ensure that it is the maximum difference between any 2 elements.

Termination :
This proves that our algorithm will work for all correct inputs of A and will return the maximum difference between 2 elements of the array.

Running Time Analysis: For finding initial values of min and max = 1 + 1 = 2
Comparing each pair of elements = $\begin{cases} \frac{n}{2} - 1 & \text{if } n \text{ is odd} \\ \frac{n}{2} - 2 & \text{if } n \text{ is even} \end{cases}$
Comparing the lesser valued element of each pair with min for $\frac{n}{2} - 1$ elements = $\frac{n}{2} - 1$
Comparing the greater valued element of each pair with min for $\frac{n}{2} - 1$ elements= $\frac{n}{2} - 1$

Total number of comparisons= $\begin{cases} (\frac{n}{2} - 1) + (\frac{n}{2} - 1) + (\frac{n}{2} - 1) + 2 = (\frac{3n}{2} - 1) & \text{if } n \text{ is odd} \\ (\frac{n}{2} - 2) + (\frac{n}{2} - 1) + (\frac{n}{2} - 1) + 2 = (\frac{3n}{2}) & \text{if } n \text{ is even} \end{cases}$

The running time = O(n)

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.
   ANSWER:
   Insertion Sort algorithm typically compares each element A[i] of the array A with each one of it's predecessors. If the current element A[i] is less than the predecessor A[i-1] it is then compared with A[i-2] and so on till it finds a place where the predecessor is less than A[i]. A[i] is then inserted here and the rest of the elements are moved up. The best case scenario for Insertion sort would be if the the array of length n is already sorted. But even in this case each element is compared with the previous element. Hence, a minimum of n operations are done. In our case we also know that at most m inversions will be made.
   $\implies$ Totally there are at most $n + m$ operations that are done
   $\implies$ It will take at most $O(n + m)$ steps
   Proof by Induction:
   Base case: Let array A have only 2 elements. It will have atmost $m = 1$ as atmost only 1 inversion is possible.
   $\implies$ Time Complexity = O(m).(1) Induction Hypothesis:
   Assume that the time complexity to sort an array with length $n - 1$ with j inversions is $O(n - 1 + j)$. (2)
   Induction Step: Consider an array of n terms. If the number added is lesser than all numbers preceding it, which is the worst case scenario there will be $(n-1)$ inversions for it. So the added number has atmost $(n-1)$ inversions. Now we know there will be atmost a total of $(n - 1) + j$ inversions considering the j inversions for the first $n - 1$ numbers of the array.
   $\implies$ The Time Complexity is $O(n - 1 + j + n - 1)$ = $O(2n + j - 2)$ = $O(n + j)$ (after ignoring constant terms). Substituting j with m(as we know there are m inversions in our case) we get:
   The Time Complexity = $O(n + m)$
   Hence Proved.

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

The maximum possible inversions will be for a case where the array is in descending order and we need to sort it using Insertion Sort.
For Example: A = [10,9,8,7,6,5,4,3,2,1]
To bring the last element to it's correct position it will take $n-1$ inversions.
For example: 1 will have to be compared with 2,3,4,5,6,7,8,9 and finally 10 and each case is an inversion = 9 inversions
Hence, the total number of inversions = $(n-1)+(n-2)+...+1$
$\implies$ Number of Inversions = $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
$\implies$ Maximum Possible Inversions = $\frac{n(n-1)}{2}$