CSCI 3104-Spring 2016: Assignment #4.

Assigned date: Tuesday, 2/16/2016,
Due date: Tuesday, 2/23/2016, before class
Maximum Points: 50 points (includes 5 points for legibility).

Note: This assignment *must be turned in on paper, before class.* Please do not email: it is very hard for us to keep track of email submissions. Further instructions are on the moodle page.

P1 (40 points) Provide efficient algorithms for each of the problems below. You should describe the main idea in a few sentences and provide pseudo code. Your pseudocode can use any previously taught algorithm as a function call. Just specify what the call does and its time complexity.

Also, write down the worst case complexities for your algorithms. Assume all arrays below are integer arrays.

(A, 5 points) Given *sorted arrays* `a` and `b`, each of size n, do they have an element in common?

Solution. The solution is to scan both arrays `a` and `b` just as in the `merge` algorithm. But instead of merging, we simply look for a common element.

```
def haveElementsInCommon(a,b):
    # Assumption: a, b are sorted
    n = len(a)
    m = len(b) # We can assume m = n, if needed.
    i = 0 # pointer into a
    j = 0 # pointer into b
    while (i < n and j < m):
        if (a[i] == b[j]):
            return True
        elif (a[i] < b[j]):
            i = i +1
        else: # a[i] > b[j]
            j = j +1
    return False
```

The time complexity is $\Theta(n)$ in the worst case.

(B, 5 points) Given a *sorted array* `a` of size m and an unsorted array `b` of size n, do they have an element in common? Your solution must not sort the array `b`.

Solution. Assume that a function `binarySearch(a,e)` searches efficiently for `e` in array `a` in time $\Theta(\log(m))$.

We simply iterate for each element of `b` and search if the corresponding element is in `a` using binary search.

```
def haveElementsInCommon(a,b):
    # Assumption: a is sorted, but b is not
    m = len(a)
    n = len(b)
    for i in range(0,n): # Iterate through b
       if (binarySearch(a,b[i])):
          return True  # Found!

    return False # Not Found
```

The complexity is $\Theta(m \log(n))$ in the worst case.

(C, 10 points) Given *unsorted arrays* `a` and `b` of size n, do they have an element in common? Your solution must avoid sorting either array.

Solution. A simple solution that runs in $\Theta(n^2)$ worst case simply compares each element of `a` to that of `b`. This is too simplistic. We can obviously do better.

A better solution is to use a hash table with a universal hash function. We will build a hash table of size n and insert all elements of `a` into the table. Next, for each element of `b`, find if the element exists in the hash table. In the worst case, all elements will collide and the complexity is still $\Theta(mn)$. However, in the average case, the complexity becomes $\Theta(m + n)$, assuming a universal hash function.

```
def haveElementsInCommon(a,b):
    # Assumption: a and b are not sorted
    m = len(a)
    n = len(b)
    h = hashTable(m)
    for i in range(0,m): # Iterate through a
       insertHashTable(h,a[i])
    for j in range(0,n):
       if (findHashTable(h,b[j]) == True):
          return True
    return False
```

(D, 10 points) The rank of a given number e in an array a is the number of elements in a that are strictly less than e. Also, given array a, let $\mathsf{minimum}(a)$ represent the minimum element of a.

Given *unsorted arrays* `a` and `b` of size n, find the rank of $\mathsf{minimum}(a)$ in the array `b`. Your solution must run in time $\Theta(n)$.

Solution.

1. Find the minimum of `a` in $\Theta(n)$ time. Let m be the minimum found.
2. Simply scan `b` and count how many elements are less than m.

Worst case complexity is $\Theta(n)$.

(E, 5 points) Given *min heaps* `a` of size m and `b` of size n, merge them into a single *min heap* of size $m+n$. Assume that $m \leq \sqrt{n}$.

Solution.

A simple solution would be to combine `a` and `b` into a single array. Just heapify this array. The overall complexity is $\Theta(m+n) = \Theta(n)$ since n dominates $m = \sqrt{n}$.

However, the solution should use the given fact that $m \leq \sqrt{n}$. A better strategy is to insert each element of the array `b` into the heap `a`.

The worst case complexity is therefore

$$\sum_{j=1}^{m} \log(n+j) \leq m \log(n+m) \ \in O(\sqrt{n}\log(n))$$

This is superior to the original algorithm since $\sqrt{n}\log(n) \in \Omega(n)$.

(F, 5 points) Given two unsorted arrays `a` and `b`, each of size n, output "YES" if every element of `a` is strictly less than every element of `b`, and "NO" otherwise.

Solution. The solution is as follows:

1. Compute the maximum of `a`, call it e.
2. Compute the minimum of `b`, call it f.
3. Output "YES" if $e < f$ and "NO", otherwise.

Time complexity is $\Theta(n)$.

P2 (10 points) A company constantly receives (integer) price quotes for its product over the internet, and at any point, we are asked to store the k smallest price quotes, where k is fixed by the manager in advance.

We consider three possible data structures for the task of maintaining the k smallest quotes so far.

1. A min heap data structure with k elements in it.
2. A balanced binary search tree data structure.

3. A sorted array of k elements.

For each of the structures, write down how you would update the data structure when a new quote comes in and the worst case running time.

Solution.

1. The heap data structure.

When each quote comes in, we insert the quote into the heap ($\log_2(k)$ operations). This grows the heap to $k+1$ elements. Therefore, we delete the largest element.

Overall complexity is $\Theta(k)$ since finding and deleting the largest element is requires us to scan the heap.

2. The balanced binary search tree data structure.

When each quote comes in, we insert the quote into the tree. This requires $\log_2(k)$ operations. We then delete the largest element in the tree. This also requires $\log_2(k)$ operations.

Overall complexity using a balanced binary tree is $\Theta(\log_2(k))$ operations.

3. Sorted array.

When each quote comes in, we find out where the quote is to be inserted in $\log_2(k)$ operations using a binary search. Deleting the largest quote is $\Theta(1)$. However, we have to now move the elements to the right of the insert position. This can be $\Theta(k)$ in the worst case.

As a result, balanced binary search tree is $\Theta(k)$.