Lab 6 Solutions

Problem 1. Show that any comparison-based algorithm to sort 4 elements requires at least 5 comparisons.

Solution. We showed in class that a binary tree T having L leaves must have node of depth at least $\lceil \log L \rceil$. In particular, if T is the decision tree for a sorting algorithm, sorting n distinct values, there must be a branch in T having length at least $\lceil \log n! \rceil$; this means that the sorting algorithm requires, in the worst case, at least $\lceil \log n! \rceil$ comparisons.

When $n = 4$, therefore, at least $\lceil \log n! \rceil$ comparisons are necessary. But

$$\lceil \log 4! \rceil = \lceil \log 24 \rceil = 5.$$

Therefore, at least 5 comparisons are needed.

Problem 2:

Use MergeSort to give ordinary sorting of input array A.

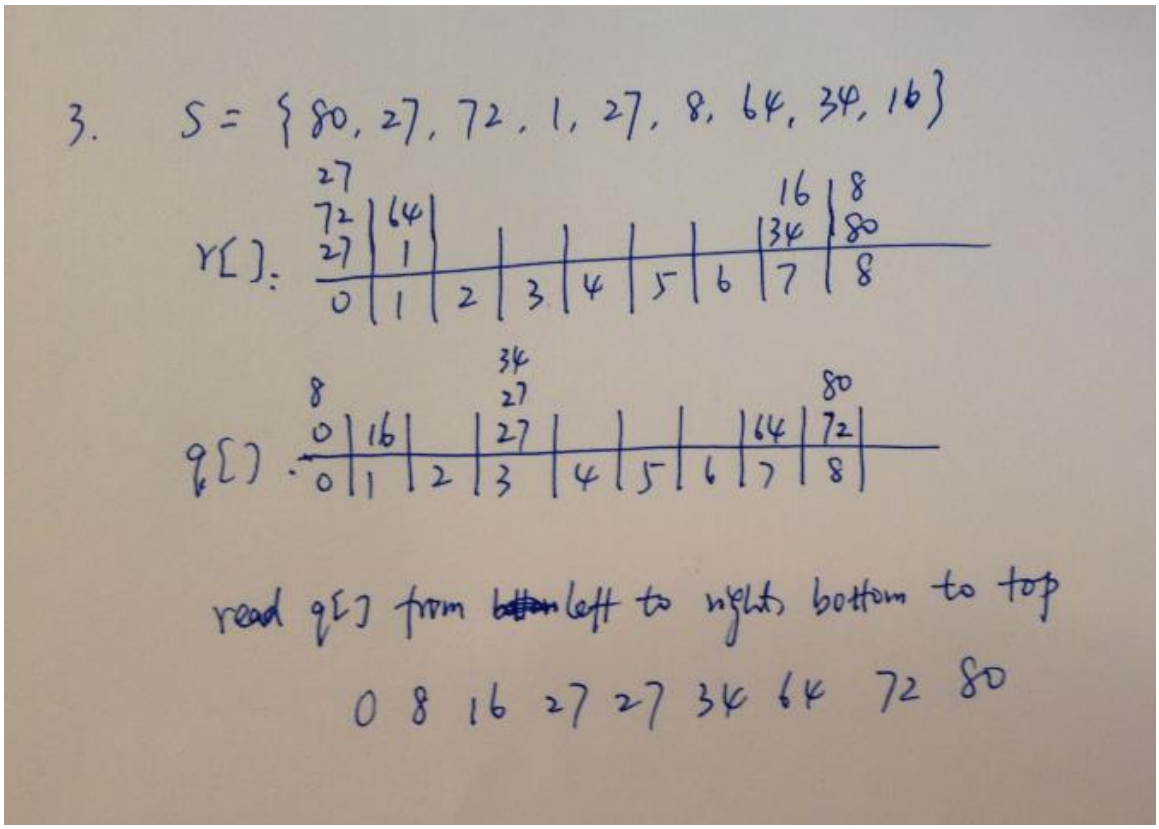
Now reorganize A using two pointers left and right. Left starts at A[0] and right starts at A[A.length-1]. Create new output array B of length n.

Insert A[0] into B[0], increment left pointer.
Insert A[A.length -1] into B[1], decrement right pointer.
Insert A[left] into B[2], increment left pointer.
Insert A[right] into B[3], decrement right pointer.

Continue until left == right.

Fastest possible is Θ(n log n). Suppose some algorithm does it in f(n) time and f(n) is o(n log n). Then we can do O(n) further processing to put the output array into normal sorted order (using the method described above in reverse), and this requires O(n) additional steps. This gives us a running time of f(n) + O(n) in o(n log n) to sort an array based on comparisons, violating the known lower bound for running times of such algorithms.

Problem 3:



Problem 4. Describe an O(n) algorithm (you can use pseudo-code, Java, or a sufficiently detailed English description) that does the following: Given an input array of *n* integers lying in the range 0 .. 3n - 1, the algorithm outputs the first integer that occurs in the array only once. (You may assume that each input array contains at least one number that has no duplicates in the array.) Explain why your algorithm has an O(n) running time.

Example: If the input array is [1, 2, 4, 9, 3, 2, 1, 4, 5], then the return value is 9 since 9 is the first integer that occurs in the array only once.

Solution: Let *A* denote the initial input array of integers. We use an auxiliary array T as a tracking array

```
T ← new array(3n)  //initialized with 0s
for i ← 0 to n − 1 do //scan A, increment T at A[i] each i
      x ← A[i]
      T[x] ← T[x] + 1
for i ← 0 to n-1 do
      x ← A[i]
      if T[x] = 1 then
            return x
```

Running Time: Two O(n) loops -> total running time is O(n)