Homework 4: Sorting -- SOLUTIONS

You will submit your solution to this assignment to the Curator System (as HW4). Your solution must be either a plain text file (e.g., NotePad) or a typed MS Word document; submissions in other formats will not be graded. Credit will only be given if you show relevant work.

1.  [15 points] Here is an array which has just been partitioned by the first step of Quicksort:

   3, 0, 2, 4, 5, 8, 7, 6, 9

   Which of these elements could have been the pivot? (if there are more than one possibility, list them all)

   Answer:
   4, 5, 9

   Explanation: everything to the left of x must be < x, and everything to the right of x must be > x.

2.  [25 points] Put the following list of values into the order that would produce the worst possible case when input into Quicksort. Assume Quicksort is using the partitioning algorithm as shown in the course notes, including the pivot selection algorithm that selects the middle-most element (rounded down in the case of lists of even size) as in \text{Pivot = List[(Lo + Hi)/2]}. Then make a sequence showing the state of the array after each recursive level of the Quicksort algorithm, marking the chosen pivots and the recursive subdivision of the array.

   1, 2, 3, 4, 5, 6, 7, 8, 9

   Answer:
   
   2 6 9 5 1 3 4 7 8
   2 6 9 5 1* 3 4 7 8
   1|6 9 5 2* 3 4 7 8
   1 2|9 6 3* 4 7 8
   1 2 3|5 6 9 4* 7 8
   1 2 3 4|6 9 5* 7 8
   1 2 3 4 5|9 6* 7 8
   1 2 3 4 5 6|9 7* 8
   1 2 3 4 5 6 7|9* 8
   1 2 3 4 5 6 7 8 9

   There are other possible valid orders. The important thing is that the Quicksort is maximized to 8 iterations. Each iteration should pick a pivot that is either the min or max of the remaining elements, thus causing all elements to partition to one side of the pivot. Thus each partitioning reduces the partition size by only 1, resulting in a total of n-1 iterations.

   It is easiest to generate a solution by starting from the end and working backwards.
3. [30 points] Prove by strong induction that sorting \( n \) elements with MergeSort takes \( T(n) = n \cdot \log_2 n \) steps, for all \( n \) that are powers of 2. Assume the merge algorithm within the MergeSort algorithm takes exactly \( m \) steps to merge a total of \( m \) elements from two sorted sub-lists, and that all other atomic operations take 0 steps.

Hint: you will need to define a recursive equation for \( T(n) \) based on the logic of MergeSort.

Answer: (note all logs are base 2)

Induct on \( n \) (technically, we should induct on some integer variable \( i \) and use \( n = 2^i \), but this is a little more readable.)

Base case: \( n = 1 = 2^0 \)

Mergesort does nothing but check the list size to see that its only length 1 and returns.

According to the problem statement, that takes 0 steps.

\( T(n) = T(1) = 0 = 1 \cdot 0 = 1 \cdot \log 1 = n \log n \)

Induction step:

Assume theorem is true for all \( i \) where \( i < n \); now prove its true for \( n \) (where \( n \) is a power of 2).

We can apply the inductive hypothesis for sorting the halves, because their lengths are \( < n \).

E.g. \( T(n/2) = n/2 \log n/2 \)

So we only need to show the theorem for the initial Mergesort call.

According to the problem statement, it takes \( n \) steps to merge \( n \) elements.

\( T(n) = \text{steps to sort each half} + n \text{ steps to merge the halves} \)

\( T(n) = T(n/2) + T(n/2) + n = 2 \cdot T(n/2) + n \)

\( = 2 \cdot (n/2 \log n/2) + n \)

\( = n \log n - \log 2 + n \)

\( = n \log n - n + n \)

\( = n \log n \)

4. [30 points] Write an algorithm that takes an unsorted list of \( n \) integers whose values are in the range of 0 to \( n \), and outputs a list of all duplicate integers. There must be no duplicates in the output list, which can also be unsorted. The algorithm must run in \( O(n) \) time. Show your analysis of the running time.

Answer:

General comparison sorting is too slow, Omega(nlogn).

Thus, use modified Bin sort: Create a table (array) of size \( n \). Each entry will serve as a counter. Initialize all entries to zero. For each of the \( n \) elements, increment the counter in the table using the element as the index position. Then traverse the table and output all indices that have counter > 1. Note this produces sorted output.

**FindDuplicates(array a, size n)**  //assume n means 0-n

```plaintext
counter = new int array(n);  // O(1)
for i = 0 to n  // O(n)
    counter[i] = 0;  // O(1)
for i = 0 to n  // O(n)
    counter[a[i]]++;  // O(1)
for i = 0 to n  // O(n)
    if(counter[i] > 1)  // O(1)
        print i;  // O(1)
```

Analysis: \( O(1 + n + n + n) = O(n) \)