ESO207: Data Structures and Algorithms *Time : 90min.*

Quiz: 1 *Max. Marks: 100*

Instructions.

a. The exam is closed-book and closed-notes. No collaboration is permitted. You may not have your cell phone on your person.

b. You may use algorithms done in the class as subroutines and cite their properties, unless explicitly asked to prove properties about them.

c. Describe your algorithms completely and precisely in English, preferably using pseudo-code.

d. Grading will be based not only on the correctness of the answer but also on the justification given and on the clarity of your arguments. Always argue correctness of your algorithms.

Problem 1.

a. Derive the solution to the recurrence relation $T(n) = 2T(2n/3) + \Theta(n)$. Assume $T(1) = 1$. You may assume that n is a power of 3. (3)

Solution. Unrolling the recurrence relation,

$$\begin{aligned} T(n) &= 2T(2n/3) + cn \\ &= cn + c(2)(2n/3) + 2^2T((2/3)^2n) \\ &= cn + c(2)(2n/3) + 2^2((2/3)^2n) + \ldots + 2^LT((2/3)^Ln) \quad L = \log_{3/2} n \\ &= \Theta(n + (4/3)n + (4/3)^2n + \ldots + (4/3)^Ln + 2^L) \end{aligned}$$

As this is a geometrically increasing series with common ratio (4/3), the sum is Theta of the last term. Now,

$$2^L = 2^{\log_{3/2}(n)} = 2^{\frac{\log_2 n}{\log_2(3/2)}} = n^{\frac{1}{\log_2(3/2)}} \; .$$

Just to verify, $(4/3)^Ln = 2^L(2/3)^Ln = 2^L$ as above. Thus, $T(n) = n^{1/\log_2(3/2)}$ (Answer).

b. Can there exist a comparison sort for sorting 7 integers using 12 comparisons. (Hint: $7! = 5040$). (3)

Solution. As proved in the class, a comparison sort must make at least $\log_2(n!)$ comparisons in the worst case. For $n = 7$, $\log_2(7!) = \log_2(5040) > 12$ (since, $2^{12} = 4096$) and hence it is impossible.

c. What is the time complexity taken by Heapify(A, i) as a function of the height of node i? In a heap A consisting of n nodes, there are at most $\lceil n/2^{h+1} \rceil$ nodes at height h. Using this result, show that the routine Build-Heap(A, n) takes linear time. (1 + 3 = 4)

```
Build-Heap$(A, n)$
1.   for $i = \lfloor n/2 \rfloor$ downto 1 { Heapify$(A, i)$ }
```

Solution. The time complexity of $\textsc{Heapify}(A,i)$ is $O(h)$, where, h is the height of the node i in the heap (viewed) as a tree. The complexity of $\textsc{Build-Heap}$ may be upper bounded as follows. Let n_h be the number of nodes at height h. For each such node i, $\textsc{Heapify}$ (i) takes time $O(h)$. Let H be the height of the tree, where, $H = \Theta(\log n)$. Thus, the time taken by $\textsc{Build-Heap}$ is

$$\begin{aligned}\sum_{h=1}^{H} h \cdot n_h &\le \sum_{h=1}^{H} h \cdot \lceil \frac{n}{2^{h+1}} \rceil \\ &\le \sum_{h=1}^{H} 2h \cdot \frac{n}{2^{h+1}} \\ &= n \sum_{h=1}^{\infty} \frac{h}{2^h} \\ &= 2n\end{aligned}$$

The for loop runs for $\Omega(n)$ steps. Hence the complexity is $\Theta(n)$.

d. Let ω_n be the primitive nth root of unity. Let $a = (a_0, \ldots, a_{n-1})$ be an n-dimensional vector. Let $y_j = \sum_{k=0}^{n-1} a_k \omega_n^{kj}$, $j = 0, 1, \ldots, n-1$. Simplify the expression to output in terms of the a_j's. (15)

$$b_m = \sum_{l=0}^{n-1} y_l \omega_n^{lm}, \quad m = 0, 1, \ldots, n-1 \ .$$

Solution

$$\begin{aligned}b_m &= \sum_{l=0}^{n-1} \omega_n^{lm} y_l \\ &= \sum_{l=0}^{n-1} \omega_n^{lm} \sum_{k=0}^{n-1} a_k \omega_n^{kl} \\ &= \sum_{k=0}^{n-1} a_k \sum_{l=0}^{n-1} \omega_n^{l(m+k)}, \text{ by changing order of summation}\end{aligned}$$

Fix $k \in \{0, \ldots, n-1\}$. If $m + k = 0 \mod n$, that is, either (i) $m = 0$ and $k = 0$, or, (ii) $m > 0$ and $k = n - m$, then, $\omega_n^{m+k} = 1$ and hence, $\omega_n^{l(m+k)} = 1$, for each l. In this case, the inner summation $\sum_{l=0}^{n-1} \omega_n^{l(m+k)} = n$. Suppose $m+k \neq 0 \mod n$, that is the above case does not occur. Then, $\displaystyle\sum_{l=0}^{n-1} \omega_n^{l(m+k)} = \frac{1 - \omega_n^{n(m+k)}}{1 - \omega_n^{m+k}} = \frac{1-1}{1 - \omega_n^{m+k}} = 0$, since, the denominator is not 0.

Hence, $b_k = n \cdot a_m$, where, $m + k = 0 \mod n$. In other words, $b_0 = n \cdot a_0$ and for $m = 1, \ldots, n-1$, $b_m = n \cdot a_{n-m}$.

Problem 2. Consider a generalized version of the selection problem where, given an array $A[1 \ldots n]$ of numbers and a number k, where, $k = \Theta(1)$, we wish to return the first to the kth smallest elements in *some* order, not necessarily in increasing order. That is, we want to return the smallest element, the second smallest element, ..., the kth smallest element, and these elements only, in some order. Give an efficient $o(n \log n)$ time algorithm for this problem, that is, without fully sorting the array. Analyze your algorithm for worst-case or expected case, as appropriate. (25 points)

Solution 1. One possible solution is to use Min-Heaps.

```
SelectFirstK($A, k$) // returns the $k$ smallest elements in $A$
1.  Let $S[1 \ldots k]$ be an array
2.  Build-Heap($A, n$) // Transform $A$ into a heap
3.  for  $i = 1$ to  $k$
4.      $S[i]$ =ExtractMin($A$)
```

Time Complexity. Note that the time taken by Build-Heap is $O(n)$, followed by k calls to $ExtractMin$ which takes $O(\log n)$ each. So total time is $O(n + k \log n)$, which for small values of k is $o(n \log n)$.

Solution 2. Another possible solution is to use the Select procedure that calls Randomized_Partition as done in the class and has expected time complexity $O(n)$. The procedure below makes two calls to Select, taking expected $O(n)$ time, and one call to Partition, taking $\Theta(n)$ time, for a total expected time $O(n)$.

```
SelectFirstK($A, k$)
1.  $j$ = Select($A, k$)
2.  $(m, q)$ = Partition($A, 1, n, A[j]$) // Partition $A[1 \ldots n]$ around pivot $A[l]$.
3.  Print $A[1 \ldots k]$
```

First, the algorithm finds the index j of the kth smallest element (line 1). Now call Partition on $A[1 \ldots n]$ using this element as the pivot. Since it is the kth smallest element, after this call, the first k elements are the k smallest elements. (Note that this is true even if there are repeated elements who are equally kth smallest).

Problem 3. Suppose you have k sorted arrays each with n elements and we wish to merge them into a single sorted array of kn elements. Design an $O(nk \log k)$-time algorithm for this problem . (*Hint*: Use a single min-heap or use Divide and Conquer). (20 points)

Solution1. We first consider a heap-based solution. Every element of the heap is a pair (v, j) where, v is the *value* field, namely the value of an element in an array, and j is the index of the array from which this value is taken, that is, $j \in \{1, 2, \ldots, k\}$. We do the following.

1. Remove the first and smallest element from each array $j = 1, 2, \ldots, k$. Let v_j be the value of the first element obtained from array j. Each such element is represented as the pair (v_j, j). Take these k elements and build it into a heap. This takes $O(k)$ time.

2. Initialize a final sorted list L as empty implemented as an array.

3. Repeat the following until the heap is empty.
    (a) Call EXTRACT-MIN operation on this heap. Suppose the item (v, j) is returned. Insert v at the end of L.
    (b) Get the next element from the jth list, say with value v'. Call INSERT $((v', j))$ in the heap.

In step 3b, if the list j from which the minimum element is returned by EXTRACT-MIN has become empty, then ignore this step.

Time Complexity. The heap is always of size at most k, and contains the current minimum element from each of the residual k arrays, unless an array has finished. The EXTRACT-MIN operation takes time $O(\log k)$. INSERT also takes time $O(\log k)$. There are nk elements, so total time is $O(nk \log k)$.

Solution 2. The second solution is to do this using Divide and Conquer, as follows. We model the problem by representing it as a two dimensional matrix $A[n \times k]$, that is, $A[1 \ldots n, 1 \ldots k]$. In other words there are k columns each representing a column array that is sorted. The procedure is given below. It uses the classical MERGE procedure for merging two sorted arrays done in the class.

```
MERGESORTEDARRAYS($A, p, r$) // Merge the sorted columns $p \ldots r$ and output a sorted column
1.  if $p == r$ { Copy and return $p$th column of $A$}
2.  else // Divide and Conquer
3.      $m = \lfloor (p+r)/2 \rfloor$
4.      $B$ = MERGESORTEDARRAYS($A, p, m$)
5.      $C$ = MERGESORTEDARRAYS($A, m+1, r$)
6.      return MERGE($B, C$)
```

Time Complexity. The recurrence relation is $T(k) = 2T(k/2) + \Theta(nk)$. The solution to this is $T(k) = \Theta(nk \log k)$.

Problem 4. Consider the task of searching a sorted array $A[1 \ldots n]$ for a given element x by an algorithm that accesses the array only via comparisons, that is, by asking questions of the form "is $A[i] \le y$?". Show that such an algorithm must take $\Omega(\log n)$ steps. (25 points).

Solution. There are n elements which are distinct outcomes for searching, and the union of the gaps $< a[1]$, $a[1] < v < a[2], \ldots, a[n-1] < v < a[n], v > a[n]$. The collection of these gaps gives us $n+1$ elements that correspond to "not found". Each of these values is distinguishable using some test of the form $A[i] \le y$ or $y < A[i]$, for y belonging to these $2n+1$ outcomes and for some $i = 1, 2 \ldots, n$. Since the comparison results in a binary decision, there are at least $2n+1$ leaves and so the height of any binary tree with at least $2n+1$ leaves satisfies, $2^h \ge 2n+1$, or, $h \ge \log_2(2n+1) = \Omega(\log n)$. Here, h is the number of comparisons.