ESO207: Data Structures and Algorithms	*Time : 90min.*
Quiz: 1	*Max. Marks: 100*

Instructions.

a. The exam is closed-book and closed-notes. 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.

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

d. Grading will be based also on the justification given and on the clarity of your arguments. Always argue correctness of your algorithms and analyze them.

Problem 1.

a. Solve the recurrence relation $T(n) = T(n-1) + n^c$, where, $c > 1$ is a constant. (3)

$T(n) = n^c + (n-1)^c + \ldots + 1^c = \Theta(n^{c+1})$.

b. Solve the recurrence relation $T(n) = T(\sqrt{n}) + 1$. Assume $T(1) = 1, T(2) = \Theta(1)$. (3)

$$T(n) = T(n^{1/2}) + 1 = T(n^{1/2^2}) + 2 = \cdots = T(n^{1/2^k}) + k$$

Setting, $k = \log(\log(n))$, $T(n^{\frac{1}{2^k}}) = T(n^{\frac{1}{\log n}}) = T(2^{\frac{\log n}{\log n}}) = T(2) = \Theta(1)$. Hence, $T(n) = \log\log n + \Theta(1)$. (All logs are to base 2).

c. Suppose A is an array with n elements that forms a min-heap, that is, $A[\textsc{Parent}(i)] \leq A[i]$, for all $i = 2, 3, \ldots, n$. Give an $O(n)$ time procedure to convert A to a max-heap. (3)

Just use Build-Max-Heap(A, n) which runs in time $\Theta(n)$.

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

$$b_m = \sum_{l=0}^{n-1} y_l \omega_n^{l(m+c)}, \quad m = 0, 1, \ldots, n-1$$

where, c is a fixed constant in $\{0, 1, \ldots, n-1\}$. (16)

$$\begin{aligned} b_m &= \sum_{l=0}^{n-1} y_l \omega_n^{l(m+c)} \\ &= \sum_{l=0}^{n-1} \sum_{k=0}^{n-1} a_k \omega_n^{kl} \omega_n^{l(m+c)} \\ &= \sum_{k=0}^{n-1} a_k \sum_{l=0}^{n-1} \omega_n^{l(k+(m+c))} \end{aligned}$$

Consider the inner summation, for a fixed value of k. If $k = -(m+c)$, the inner summation is n. If $k \neq -(m+c)$, then, $k+m+c \in \{-(n-1),\ldots,(n-1)\}$ except 0. Hence, the inner summation is 0. Thus, we have,

$$b_m = na_{-(m+c) \mod n} = na_{n-m-c \mod n}, \qquad \text{for } m = 0,\ldots,n-1\ .$$

Problem 2. Design a variant *New_Partition* of the *Partition*(A,p,r) procedure that *runs in time* $O(r-p+1)$ and divides the input array $A[p\ldots r]$ into three subarrays $A[p,\ldots,q-1]$, $A[q,\ldots,s-1]$ and $A[s,\ldots,r]$. Let *key* denote the value $A[r]$ in the original array A. After *New_Partition*(A,p,r), each element of $A[p,\ldots,q-1]$ is strictly less than *key*, the elements $A[q,\ldots,s-1]$ are identical in value to each other and to *key*, and each element of $A[s,\ldots,r]$ is strictly greater than *key*. *New_Partition* returns the pair (q,s), where, $p \le q < s \le r$.

a. Describe clearly the invariant that your loop satisfies (use a figure if it helps), and show how you will process the next element of the array so that the loop invariant is maintained. (10)

b. Based on the loop invariant, write pseudo-code for *New_Partition*(A,p,r). (10)

c Write pseudo-code for Quicksort using *New_Partition*. (5)

Solution Outline. Let j be the running counter from p to $r-1$. Invariant: Let $A[p\ldots i]$ be each $< A[r]$, $A[i+1,\ldots,k]$ be each equal to $A[r]$ and $A[k+1,\ldots,j-1]$ be each $> A[r]$. The region $A[j\ldots r-1]$ is unrestricted.

Initially, to maintain the invariant, let $i = p-1$, $k = p-1$. Loop counter j runs from $j = p\ldots r-1$ as a for loop.

1. $A[j] >$ *key*. We do nothing, and j will be incremented in the for loop. Invariant is preserved.

2. $A[j] =$ *key*: exchange $A[k+1]$ by $A[j]$. Increment k–as follows:
   exchange $A[k+1]$ with $A[j]$; $k = k+1$;

3. $A[j] <$ *key*. Now $A[j]$ should be copied to position $A[i+1]$. To make room at position $i+1$, $A[i+1]$ should be copied to position $A[k+1]$, $A[k+1]$ should be copied to $A[j]$, all simultaneously. Increment i and k. We can do this with the sequence
   key $= A[j]; A[j] = A[k+1]; A[k+1] = A[i+1]; A[i+1] =$ *key*; $i = i+1, k = k+1$;

   Equivalently, we can exchange $A[j]$ with $A[k+1]$, then exchange $A[k+1]$ with $A[i+1]$; now increment i and k.

```
New_Partition(A, p, r)
    pivot = A[r]
    i = p − 1; k = p − 1
    for j = p to r − 1
        if A[j] == pivot {
            exchange A[k + 1] with A[j]; k = k + 1 }
        elseif A[j] < pivot {
            exchange A[j] with A[k + 1]
            exchange A[k + 1] with A[i + 1]
            k = k + 1; i = i + 1 }
    }
    exchange A[r] with A[k + 1] ; k = k + 1
    return (i + 1, k + 1)
```

Time complexity is obviously $\Theta(r - p + 1)$. For each index $j = p \ldots r$, $\Theta(1)$ number of comparisons and operations are done. The quicksort pseudo code is as follows. Top-level call is *Quicksort*$(A, 1, n)$.

```
QuickSort(A, p, r) // Sort A[p . . . r]
    if p < r {
        (q, s) = New_Partition(A, p, r)
        // A[q, . . . , s − 1] are all equal to pivot and in correct sorted position in A.
        QuickSort(A, p, q − 1)
        Quicksort(A, s, r)
    }
```

Problem 3. Given an array $A[1, \ldots, n]$ of integer numbers, give an algorithm to sort A in time $O(n+M)$ where,

$$M = (\max_i A[i]) - (\min_i A[i]) \ .$$

Give an outline of the algorithm and then argue its time complexity. (18 + 7)

Solution outline.

1. Make a single pass over $A[1 \ldots n]$ and find $k = \min_{i=1}^{n} A[i]$ and $K = \max_{i=1}^{n} A[i]$. This takes $O(n)$ time
2. Create a new array $C[k, \ldots, K]$, each array has two fields $C[l].f$ (frequency) and $C[l].s$ (cumulative frequency). We want $C[l].f$ to be the number of occurrences of l in $A[1 \ldots n]$. $C[l].s$ is cumulative frequency $= x \sum_{r=k}^{l} C[r].f$. Takes time $O(M + 1)$.
3. Initialize $C[l].f$ to all 0s. Time is $O(M)$.
4. Count the number of times l occurs in $A[1 \ldots n]$, for $k \leq l \leq K$, as follows. This is frequency count. This takes time $O(n)$ time.

```
for i = 1 to n {
    C[A[i]].f = C[A[i]].f + 1 }
```

5. Make a pass over the array $C[k \dots K]$ to get cumulative frequencies $C[l].s = \sum_{r=k}^{l} C[r].f$. Takes time $O(M+1)$.

   $sum = 0$
   for $l = k$ to K {
   $\quad C[l].s = sum + C[l].f$
   $\quad sum = sum + C[l].s$ }

6. Make an empty copy of $A[1 \dots n]$ in $B[1 \dots n]$. Time $O(n)$.
7. Make a backwards pass over A, that is, for $i = n$ down to 1. Index of $A[i]$ is $C[A[i]].cumul$, and decrement $C[A[i]].cumul$ by 1. Time: $O(n)$

   for $i = n$ **downto** 1 {
   $\quad posn = C[A[i]].s$
   $\quad B[posn] = A[i]$
   }
   $\quad C[A[i]].s = C[A[i]].s - 1$ // reduce cumulative frequency of $A[i]$ by 1

   Total time is $O(n + M)$.

Problem 4. You are given k sorted arrays with a total number of n elements across all the arrays. We wish to merge them into a single sorted array of kn elements. Describe an $O(n \log k)$-time algorithm for this problem. (25 points)

Solution outline. Keep a min-heap or min-priority queue H. Each entry is a pair (v, i), where, v is the key value and i is array number between 1 and k from which this element has been picked. The ith array is $A[i, 1] \dots A[i, N[i] + 1]$. Assume $N[i]$ is the number of items in the ith array and $A[N[i] + 1] = \infty$.

1. Initialize Heap.

   1. **for** $i = 1$ to k
   2. $\quad$ INSERT$(H, (A[i, 1], i))$
   3. create empty array $B[1 \dots n]$
   4. **for** $i = 1$ to k
   5. $\quad N[i] = 2$ // current index of next smallest element in ith array

   The time taken is $O(k \log k)$ for the for loop in lines 1-2.

2. Extract the smallest element from the heap. Let $(v, i) =$ EXTRACT-MIN(H). Place v at the current end j of union array B. Insert next item from array i into Heap H.

   for $j = 1$ to n {
   $\quad (v, i) =$ EXTRACT-MIN(H)
   $\quad B[j] = v$
   $\quad$ INSERT$(H, A[i, N[i]])$
   $\quad N[i] = N[i] + 1$
   }

   The heap has k elements, the current min from each of the k arrays. The operation Extract-Min takes time $O(\log k)$. Insert also takes time $O(\log k)$. Total time of the loop is $O(n \log k)$.