ESO207: Data Structures and Algorithms	*Time : 2 hrs*
Mid-Semester Exam: 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 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. Given an array $A[1, \ldots, n]$, and given $1 \leq k \leq l \leq n$, give an expected $O(n)$-time algorithm that outputs $l - k + 1$ numbers that are the kth smallest to the lth smallest in the array A, *in any order*. (15)

Soln. . Suppose we first call $a = \textsc{Select}(A, 1, n, k)$ to return the k th smallest element in the array A. On expectation this takes $O(n)$ time. Now let us partition the array $A[1 \ldots n]$ around a, to get two indices m, p such that $A[1 \ldots m] < a$, and $A[m+1, \ldots, p] = a$ and $A[p+1 \ldots n] > a$. Clearly, $m + 1 \leq k \leq p$. Consider $A[k, \ldots n]$. Call $b = \textsc{Select}(A, k, n, l - k + 1)$ to return the $l - k + 1$th smallest element in the new sub-array $A[k, \ldots n]$. This takes on expectation $O(n - k + 1) = O(n)$. Partition $A[k \ldots n]$ around b. Now return $A[k \ldots l]$.

procedure $\textsc{Selectktol}(A, 1, n, k, l)$
  $a = \textsc{Select}(A, 1, n, k)$
  $(m, p) = \textsc{Partition}(A, 1, n, a)$
  $b = \textsc{Select}(A, k, n, l - k + 1)$
  $\textsc{Partition}(A, k, n, b)$
  return $A[k, \ldots, l]$

Problem 2. An array $A[1 \ldots n]$ is said to have a majority element if more than half of its entries are the same. Given the array, you have to design an efficient algorithm to tell whether the array has a majority element, and if so, find that elements. Note that the elements are not necessarily from some ordered domain like the integers, so there *cannot* be any comparisons of the form "is $A[i] > A[j]$?". However, you *can* answer questions of the form "is $A[i] = A[j]$?". (*Note:* Design an $O(n)$ worst-case time algorithm. An $O(n \log n)$-worst case time algorithm will fetch only 3/4th of the marks.) (20)

Soln. One way to solve the problem is to consider pairs $A[1, 2]$, $A[3, 4], \ldots$, and check if they are the same. If the elements of the pair are different, we drop the pair, otherwise, both elements are the same, and we promote this element to the next level. So at the next level, we create a list of elements. The key observation is that if the original list had a majority element, then this sub-list has the same majority element. (Otherwise, no guarantee is made). At each step, the size of the list is reduced at least by 1/2. So we get the recurrence relation $T(n) < T(n/2) + O(n)$, whose solution is $T(n) = O(n)$. Finally, when

only one element remains, this element can be tested to check if it is the majority by making one pass over the array. If yes, it is the majority element, and if no, there is no majority element.

Argument: Divide the set of all pairs as we have formed them (i.e., $(1,2),(3,4),\ldots$)) into two sets, one in which the pair elements are distinct, and one in which the pair elements are the same. Now suppose there is a majority element. Hence among the pairs where the elements are the same, this element must be in majority. This proves the observation for the divide and conquer algorithm above.

Solution 2. Keep an integer counter c and variable x that can store an element. Initialize $c = 0$. x has a meaning only when $c > 0$. Make a pass over the array elements, one at a time in order. Suppose we are currently looking at $A[i]$. First check if $c = 0$? If so, then, set c to 1 and x to $A[i]$ and move forward. If $c > 1$, then, check if $x = A[i]$? If so, then, increment c by 1. If not, then, decrement c.

If there is a majority element, then, it must be in x. To verify, make an additional pass over $A[1 \ldots n]$.

Argument The decrement of c by the algorithm can be seen as "pairing and throwing" away a pair of distinct elements. The increment happens when we see a copy of the same element. Suppose there is a majority element m. Suppose every occurrence of a non-majority is paired with m and thrown away. But since m is in majority, that still leaves at least one occurrence of m at the end.

Problem 3. Given an array $A[1 \ldots n]$ of numbers, let $l(i)$ be the largest index smaller than i such that $A[l(i)] > A[i]$ and let $r(i)$ be the smallest index larger than i such that $A[i] < A[r(i)]$. If $l(i)$ is undefined as per the above definition (for e.g., for $i = 1$), then $l(i)$ is set to 0 and similarly, if $r(i)$ is undefined, then, $r(i)$ is set to $n + 1$. The dominating span of an index $i \in \{1, 2, \ldots, n\}$ is defined as $h(i) = r(i) - l(i) - 1$, that is the indices $l(i) + 1, \ldots, r(i) - 1$ is the largest contiguous segment of indices where $A[j] \leq A[i]$. Design a linear time algorithm to compute the dominating span array $h[1, \ldots, n]$. Write in pseudo-code. (20)

Soln. . We will compute the array $l[1 \ldots n]$ and $r[1 \ldots n]$ separately and then for each i, compute $h(i) = r(i) - l(i) - 1$.

Computing $l(i)$. As a convention, let $A[0] = \infty$. We keep a stack S of indices: say $i_k, i_{k-1}, \ldots, i_2, i_1$, from the top to the bottom. The loop invariant is the following.

1. $i_k > i_{k-1} > \ldots > i_1$.
2. For every index i_j, $A[i_j] \geq A[t]$, for every $i_j - 1 \leq t \leq i_{j-1}$.
3. $A[i_k] < A[i_{k-1}] < \ldots < A[i_1]$.

When we see $A[j]$ for the first time, we check the top of the stack i_k if the value at that index is at most $A[j]$. If so, we pop the stack and continue in this manner, until the top of stack $i_{k'}$ satisfies $A[i_{k'}] > A[j]$. Now we set $l(j) = i_{k'}$ and push j onto stack. Initially, we push 0 on stack.

```
ComputeL(A, n)
1.  Make S to be empty stack
2.  Push(S, 0)    // Recall A[0] = ∞
3.  for  j = 1 to n
4.       while  A[Top (S)] ≤ A[j]
5.            Pop(S)
6.       l[j] =Top(S)
7.       Push(S, j)
```

Every element is pushed exactly once and popped exactly once. In doing so, a constant number of operations are executed. Hence, this is $\Theta(n)$.

The computation of $r[i]$ is symmetric. Here we let $A[n+1] = \infty$ for ease of termination condition. Keep a stack T, at the point just before scanning index j, say the stack contents from top to bottom are, $i_k, i_{k-1}, \ldots, i_2, i_1$. While the $A[$ top of the stack index $]$ is at most $A[j]$, we set $r($ top of the stack $)$ to j, and pop the stack. When the condition fails, then we push j onto the stack.

```
COMPUTER(A, n)
1.  Make T to be empty stack
2.  for  j = 1 to n + 1
3.       while not ISEMPTY(T) and A[TOP(T)] ≤ A[j]
4.            r[TOP(T)] = j
5.            POP(T)
6.       PUSH(j)
```

Time taken is $O(n)$ as earlier. Now compute $h[i] = r[i] - l[i] - 1$, for each $i = 1, 2, \ldots, n$, in $\Theta(n)$ time.

Problem 4. You are given a binary search tree T and a (non-NIL) node s of the tree. The SPLIT(T, s) operation returns a binary search tree T' in which s is the root of T' and the tree T' has exactly the same non-NIL keys as T. The split operation may destroy the tree T and transform it to T'. Design an $O(h)$-worst case time algorithm for this problem, where, h is the height of T. (20)

Soln. . The procedure is to keep rotating about the parent of the node s until s becomes the root. In one step, we do the following. If s is the left child of its parent y, then we call RIGHT-ROTATE(T, y) to make s the parent of y. Otherwise, if s is the right child of its parent y, then we call LEFT-ROTATE(T, x). By doing this one call, the distance of s from the root of the tree diminishes by 1. This is repeated until s becomes the root of the tree. Since the rotations take time $O(1)$, the total time is bounded by $O(h)$.

Problem 5.

1. Let M be an $n \times n$ matrix. Let v be an n-dimensional vector, each of whose entries are chosen randomly and independently to be 0 or 1, each with probability 1/2. Show that if M is a non-zero matrix, then, $\Pr[Mv = 0] \leq 1/2$. (10)
2. Using the above, give a randomized algorithm that takes as input A, B and C, runs in expected $O(n^2)$ time, and tests whether $AB = C$ in the following sense. If $AB = C$, then the algorithm outputs True, otherwise, in expected $O(n^2)$ time, it outputs False. (15)

Soln. Suppose $Mv = 0$ and M is non-zero. Then v is in the null space of M. The null space has dimension k, where, $k = 0, 1, \ldots n-1$ (not n because M is assumed non-zero). The highest probability comes when the dimension of the null space is $n-1$, which is defined by the plane passing through n points, including one the origin. The plane will not pass through the remaining $2^n - n$ points of the cube. So the probability that $Mv = 0$ is at most $\dfrac{n}{2^n} \leq \dfrac{1}{2}$.

In order to test whether $AB = C$, we perform the above random test $(AB - C)v = 0$, where, v is chosen randomly on the unit cube as above. If $AB = C$, then, the above is always 0. If $AB \neq C$, then,

$AB - C \neq 0$, and with probability at most $1/2$ the test $(AB - C)v$ returns 0. The expected number of times before random test $(AB - C)v$ returns a non-zero is at most 2 (recall fair coin tossing; this coin is unfair and has greater probability of turning heads). Each test of $(AB - C)v$ takes time $O(n^2)$, since, ABv can be computed as $A(Bv)$, which takes $O(n^2 + n^2) = O(n^2)$. Hence, if $AB \neq C$, then in expected $O(n^2)$ time, the above method will find it by finding a $v \in \{0, 1\}^n$ such that $(AB - C)v \neq 0$.