| **ESO207A: Data Structures and Algorithms** | *July-Nov 2019* |
|---|---|
| **Mid-Semester Exam** | *18th September, 2019* |

**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.** You are given an array $A = (a_1, \ldots, a_n)$ of numbers. Define $\mu_\infty$ to be the value of $\mu$ that minimizes the function

$$\max_i |a_i - \mu|$$

Design an algorithm to compute the statistic $\mu_\infty$ in time $O(n)$. You may assume that simple arithmetic and comparison operations on the $a_i$'s are computed each in $O(1)$ time. (10)

***Solution.*** Let $m = \min_i a_i$ and $M = \max_i a_i$. For any value of $\mu$,

$$\max_i |a_i - \mu| = \max(|\mu - m|, |M - \mu|) \geq (M - m)/2 \ .$$

This value is attained for $\mu = (M + m)/2$. The algorithm is to find the minimum and maximum elements in the given array and return their average. This can be computed in time $O(n)$.

**Problem 2.** You are given a very large array $A[\cdot]$ in which the first $n$ elements contain integers in sorted order and the rest of the cells are filled with $\infty$. But you are *not* given the value of $n$. Describe an algorithm that takes an integer $x$ as input and finds a position in the array containing $x$, if such a position exists in $O(\log n)$ time. (*Note:* You may use binary search as a routine by specifying it without writing pseudo-code.) (15)

***Solution.*** The algorithm starts with array segment $A[1 \ldots 1]$, and if $x$ is larger than the current interval range, it keeps doubling it, that is, $[1 \ldots 1], [2 \ldots 3], [4 \ldots 7], [8 \ldots 15], \ldots, [2^k, \ldots, 2^{k+1} - 1]$ etc., in sequence, until it either finds the rightmost segment item $A[2^{k+1} - 1] \geq x$ or $A[2^{k+1} - 1] = \infty$. Now a binary search for $x$ in the interval $A[2^k, \ldots, 2^{k+1} - 1]$ is done . The routine $BinSearch(A, s, r, x)$ does a binary search for $x$ in the array segment $A[s \ldots r]$ and either returns *not found* or returns an index $t$ such that $A[t] = x$ and $s \leq t \leq r$.

*Search*$(A, x)$
1. $k = 1$, $r = 1$, $s = 1$ // $r = 2^{k-1}$, $s = 2^k - 1$
2. **while** $A[s] < x$
3. $\quad k = k + 1$
4. $\quad r = 2(r + 1) - 1$, $s = 2s$
5. $\quad$ **return** *BinSearch*$(A, s, r, x)$.

Complexity: The outer while loop runs at most $k = \lceil \log(2N) \rceil$ times. The binary search is done over an array segment of size at most $2^k$ and requires $O(k)$ time, where can be at most $\log N$. Total time is therefore $O(\log(N)$.

**Problem 3.** Given a binary tree on $n$ nodes, outline an $O(n)$ time procedure that prints a leaf node that is furthest away from the root node (in terms of number of number of edges in a simple downward path from the root). Argue correctness of the algorithm and its time complexity. (15)

***Solution.***

Keep a queue $Q$. Enter the root node $r$ of $T$ into an empty queue and set $r.d = 0$. For any vertex $u$, $u.d$ is the shortest distance from $r$ to $u$ in the tree $T$. At each iteration until the queue is empty, dequeue a node $u$ from the front of the queue. Enqueue all non-NIL children $v$ of $u$ and set $v.d = u.d + 1$. Return the last node whose dequeue makes the queue empty.

Time complexity is $O(n)$, as all edges and vertices are visited once. For correctness, note that the following invariants are maintained: $u.d$ is the length of the downward simple path from $r$ to $u$. Because of the use of a queue, $u$'s parent is enqueued first and then $u$ is enqueued. $u$ is enqueued when $u.p$ is dequeued and is under processing. Hence, vertices are enqueued and dequeued respectively in order of increasing distance from the root.

Note that the $d$ attributes are unnecessary, since, vertices are inserted as per the $.d$ values and therefore are dequeued as per the $d$ values. Thus, the vertex dequeued at the end is one of the furthest vertices.

```
FURTHEST_LEAF(T)
1.   Q = φ
2.   r = T.root
3.   r.d = 0
4.   Q.ENQUEUE(r)
5.   whilenot ISEMPTY(Q)
6.           u = Q.DEQUEUE()
7.           if u.left ≠ NIL
8.                   Q.ENQUEUE(u.left)
9.                   u.left.d = u.d + 1
10.          if u.right ≠ NIL
11.                  Q.ENQUEUE(u.right)
12.                  u.right.d = u.d + 1
13.  return u
```

**Problem 4.** Let $A, B$ and $C$ be $n \times n$ matrices. In this problem, we consider a randomized algorithm to test if $AB = C$ in expected $O(n^2)$ time.

1. Let $v$ be an $n$-dimensional vector whose entries are randomly and independently chosen to be 0 or 1 with equal probability 1/2. Prove that if $M$ is a non-zero $n \times n$ matrix, then, $\Pr[Mv = 0] \leq 1/2$. (10)
2. Show that $\Pr[ABv = Cv] \leq 1/2$ if $AB \neq C$. Give an $O(n^2)$ expected time algorithm for checking whether $AB = C$. (10)

***Solution.***

1. Let $M$ be a non-zero matrix. Then, $Mv = 0$ iff $v$ is in the null space of $M$, which can be at most $n-1$ dimensional. By construction, $v$ is one of the $2^n$ end points of a unit cube anchored at the origin. So at least half of the cube vertices (or more) lie outside of any $n-1$ dimensional plane passing through the origin. Hence, $\Pr[Mv = 0] \leq 1/2$, assuming $M$ is non zero.

2. Note that for any $n \times n$ matrix $M$, and any vector $u$, $Mu$ can be calculated in the standard way using $O(n^2)$ operations. Thus, $ABv = Cv$ is equivalent to $(AB - C)v = 0$ can be calculated as follows. Compute $Cv$ in $O(n^2)$ time; compute $Bv$ in $O(n^2)$ time and $A(Bv)$ in another $O(n^2)$ time. Thus, $ABv = Cv$ can be verified in $O(n^2)$ time.

   If $AB = C$ then $ABv = Cv$ for all $v$. Otherwise, $\Pr[ABv = Cv] \leq 1/2$. So, we choose independently different random values of $v = v_0, v_1, \ldots, v_k, \ldots,$. The first value for which $ABv_j \neq Cv_j$ we can conclude that $AB \neq C$.

   *Analysis.* Suppose $AB \neq C$. The probability of getting a false positive for each of the $k$ trials, by independence of trials, is at most $1/2^k$. So probability of a false diagnosis (i.e., $AB \neq C$ but the tests are unable to diagnose inequality) is $\leq 2^{-k}$. The expected number of trials $j$ before we get $ABv_j \neq Cv_j$ is less or equal to $1/(1/2) = 2$.

**Problem 5. Stack** *Algorithm and Correctness: 16 Complexity 4*

You are given a sequence $S$ consisting of the characters left parenthesis '(' and right parenthesis ')' only and of ***even*** length. Find the smallest number of parenthesis reversals that are needed to make the sequence a balanced parenthesis expression. Examples

$S_1 =$ ) ( Output = 2, $S_2 =$ ) ( ) ( Output = 2, $S_3 =$ ) ( ( ( Output = 3

1. Write $S_1 = )_1(_2$ where the subscripts are used to label the parenthesis characters. Reverse it to $(_1)_2$ to balance.

2. Let $S_2 = )_1(_2)_3(_4$. Mininum reversals to give balanced expression is $(_1(_2)_3)_4$.

3. Write $S_3 = )_1(_2(_3(_4$. Reverse $)_1$ to $(_1$ to give $(_1(_2(_3(_4$. Then, $(_3$ and $(_4$ are each reversed to give the sequence $(_1(_2)_3)_4$ which is balanced. Also $(_2$ and $(_4$ can be reversed to give the sequence $(_1)_2(_3)_4$. Either way requires 3 reversals.

(*Notes*: The subscript numbering is shown only for convenience.)

***Solution.*** *Outline.* The algorithm is as follows. Set reversals counter to 0. Scan input from left to right one character at a time.

1. If we see a $(_i$, push $i$ onto stack.

2. If next character is a $)_j$, then, pop stack, match with whichever $i$ was on top of stack and pair them away. However, if the stack is empty, reverse to $(_j$, push it onto stack, and increment reversals counter by one.

3. Finally, when the input is finished, the stack contains a sequence of (s. If this is odd, then no amount of reversals can suffice. If it is even, then change half of the stack from the top to ) and add this to reversals counter.

Return *reversal_count* + (stack height)/2.

The correctness of the algorithm is based on the invariant that at any point of the input the minimum number of reversals is the current reversal count + (1/2) stack height of unpaired left (s, if the stack height is even.

**Problem 6.** *Range queries over Red-Black Trees.* (20)

A *right range-query* RIGHTRANGEQ$(T, a)$ takes a red-black tree $T$ and a key $a$ and returns *all* the nodes $x$ of $T$ such that $a \leq x.key$. Note that $a$ may or may not be present in the tree. The output order of the nodes should be as per the ordering of the inorder traversal of $T$. Assume $T$ has $n$ nodes in total and $m$ is the number of nodes in the output of RIGHTRANGEQ$(T, a)$. Give an $O(m + \log n)$ algorithm for RIGHTRANGEQ$(T, a, b)$. (*Algorithm + Correctness*: 13, *Complexity*: 7)

***Solution.*** Define the function CGE$(T, a)$ to be the closest greater than or equal to node $x$ in the inorder ordering of the nodes of $T$ such that $x.key \geq a$. In particular, if $a$ occurs in $T$, then $x$ is the earliest occurrence of key value $a$ in the inordering. If $a$ does not occur in $T$, then $x.key$ is the smallest value greater than $a$ that occurs in the tree.

Once $x =$CGE$(T, a)$ is found, we can repeatedly call the SUCCessor function to print all nodes of $T$ starting from $x$ in the inordering.

How do we compute CGE$(T, a)$? First we SEARCH$(T)$ for the key $a$. If the answer is Yes and a node $y$ is returned such that $y.key = a$, then, we repeatedly call PRED of $y$, till we obtain the first node with key $< a$. The following node gives the starting point.

If the answer to SEARCH$(T)$ for $a$ was No, let $y$ be the leaf node of $T$ from where the search for $a$ dropped off to NIL. Note that if $a > y.key$, then, our starting point is SUCC$(y)$ and otherwise if $a < y.key$, then the starting point is $y$ itself. (Why?)

```
CGE(T, a)
1.  y = NIL, x = T      // first search for a
2.  while x! = NIL and  x.key ≠ a
3.        y = x
4.        if  x.key < a  x = x.left
5.        else  x = x. right
6.  if x ≠ NIL // if a is found, find the earliest a
7.        z = x; x = pred(x);
8.        while x! = NIL and  x.key == a
9.              z = x; x = pred(x);
10.       return z;
11. elseif  a < y.key
12.       return y
13. else return  succ(y);
```

```
RIGHTRANGEQ(T, a)
1.  x =CGE(T, a)
2.  while  x ≠ NIL
```

3. **print** $x$
4. $x = \text{SUCC}(x)$

Complexity Analysis: For simplicity, first assume that $a$ is a key that appears uniquely (i.e., only once) in the tree $T$ and the node is $s$. Then, RIGHTRANGEQ locates $s$ along a downward path from the root and follows it by repeated calls to SUCC. Let $S$ be the sequence of edges traversed from the root to $s$ followed by the edges traversed in the repeated calls to SUCC.

Every edge $\{u, v\}$ is such that $u = v.p$ is traversed first in the downward direction $(u, v)$ and some time later in the reverse direction $(v, u)$ in $S$. This is because of the nature of the query: after $s$, all successors are printed until the next call to succ returns NIL. In this sense, every edge $\{u, v\}$ in $S$ is traversed in both directions, downwards and later upwards.

Say that $\{u, v\}$ is traversed with $u = v.p$ bi-directionally. Then, $v$ may or may not be printed. When is $v$ not printed? When $v$ is not $s$ and is also not a successor to $s$. Then, $v$ lies on the path from root to $s$ and is a left child of its parent. The number of such $v$'s is at most the number of nodes on the unique path from root to $s$, which is bounded above by height of $s$. Every other edge pair $(u, v)$ and $(v, u)$ appears twice in $S$ and can be charged to the vertex $v$ that gets printed in the output. So the total number of edges traversed is at most $2h + 2m = O(\log n + m)$, since $h = O(\log n)$ for RB-trees.