CMPS 256 — ALGORITHMS AND DATA STRUCTURES
Summer 2011 – 2012
Midterm Exam
Friday July 13, 10:00 – 11:00 a.m.

Instructions.
This midterm is scored out of 100.
This midterm is open book and open notes: you can use the textbook, notes you have taken in class, and your homework solutions. Show your work, as partial credit will be given.
You may use any algorithm that was covered in class. Just give the name of the algorithm and the page number in the book where the algorithm is given.

Please draw a horizontal line to separate each answer from the next.
Best of luck!

Problem 1 (20 points)

(1.1) (4 points each) Give tilde approximations for the following:

1. $N^2 + 3N \lg N$
2. $N2^N + 2^N$
3. $1 + 1/(\lg N)$

(1.2) (4 points each) State whether the following are true or false. Give a brief justification for your answer.

1. $n^3 = \Omega(n^{2.99})$
2. $(\lg n)^{10} = O(n)$

Problem 2. (40 points) You are given an array A of size N, indexed as usual from 0 to N. You are also given that A contains exactly 4 distinct values v_0, v_1, v_2, v_3. Formally:

(for all i from 0 to $N-1$ inclusive: $A[i] = v_0$ or $A[i] = v_1$ or $A[i] = v_2$ or $A[i] = v_3$)
and
$v_0 < v_1 < v_2 < v_3$

You are also given what the actual values are.

(2.1) (20 points) Give a comparison-based sorting algorithm
`lsort(int[] A, int v0, int v1, int v2, int v3)`
which sorts A, where `v0, v1, v2, v3` are the values occurring in A, as discussed above.

Your algorithm must have **worst case** running time in $O(N)$. Remember that worst case is a limit on the running time for **all** inputs, unlike average case running time. You may give pseudocode or actual Java code. **Do not make any assumptions about the relative frequencies of occurrence of v0, v1, v2, v3**.

(2.2) (20 points) Prove that your algorithm has worst case running time in $O(N)$.

No credit for an $O(N \lg N)$ running time algorithm.

Problem 3. (40 points) We wish to add to a regular BST a field `min` for each node `x` which records the minimum value in the subtree roooted at `x`.

(3.1) (20 points) Give a modified `put` procedure (see text, p. 399) which maintains the `min` field.

(3.2) (20 points) Prove that your modified put has worst case running time in $O(h)$, where h is the height of the BST. You do not have to handle deletion.

No credit for a put procedure with running time asymptotically larger than $O(h)$.

Sample Solutions.

Problem 1 (20 points)

(1.1) (4 points each) Give tilde approximations for the following:

1. $N^2 + 3N \lg N$. **Sample solution.** N^2
2. $N2^N + 2^N$. **Sample solution.** $N2^N$
3. $1 + 1/(\lg N)$. **Sample solution.** 1

(1.2) (4 points each) State whether the following are true or false. Give a brief justification for your answer.

1. $n^3 = \Omega(n^{2.99})$. **Sample solution.** True, since $n^3 \geq cn^{2.99}$ for $c = 1$.
2. $(\lg n)^{10} = O(n)$. **Sample solution.** True, since any polylog is asymptotically smaller than any polynomial.

Problem 2. (40 points) You are given an array A of size N, indexed as usual from 0 to N. You are also given that A contains exactly 4 distinct values v_0, v_1, v_2, v_3. Formally:

(for all i from 0 to $N - 1$ inclusive: $A[i] = v_0$ or $A[i] = v_1$ or $A[i] = v_2$ or $A[i] = v_3$)
and
$v_0 < v_1 < v_2 < v_3$

You are also given what the actual values are.

(2.1) (20 points) Give a comparison-based sorting algorithm
`lsort(int[] A, int v0, int v1, int v2, int v3)`
which sorts A, where `v0, v1, v2, v3` are the values occurring in A, as discussed above.

Your algorithm must have **worst case** running time in $O(N)$. Remember that worst case is a limit on the running time for **all** inputs, unlike average case running time. You may give pseudocode or actual Java code. **Do not make any assumptions about the relative frequencies of occurrence of v0, v1, v2, v3**.

(2.2) (20 points) Prove that your algorithm has worst case running time in $O(N)$.

No credit for an $O(N \lg N)$ running time algorithm.

Sample solution.

(2.1) First, modify the partition procedure of Quicksort (text, page 291) so that (1) a pivot can be given as an argument, and (2) elements in the left partition are strictly less than the pivot. Note that I removed the auto increment/decrement operators, since the needed modification doesn't work correctly in their presence.

```
    // partition the subarray a[lo .. hi] by returning an index j
    // so that a[lo .. j-1] <= a[j] <= a[j+1 .. hi]
    private static int partition(Comparable[] a, int lo, int hi, Comparable v) {
        int i = lo - 1;     //NOW MUST INCLIDE a[lo] IN THE PARTITION
        int j = hi + 1;
        //       Comparable v = a[lo];   REMOVED
        while (true) {
            {invariant: a[lo..i] < v /\ a[j..hi] >= v}
            // find item on lo to swap
            i += 1;
            while (less(a[i], v)) {
                i += 1;
                if (i == hi) break;
            }
            // a[lo ... i-1] < v /\ a[i] >= v

            // find item on hi to swap
            j -= 1;
            while (less(v, a[j]) || equal(v, a[j])) {   //CHANGED
                j -= 1;
                if (j == lo) break;
            }
            // a[j+1..hi] >= v /\ a[j] < v
                //NOTE a[j] strctly less than pivot

            // check if pointers cross
            if (i = j+1) break;          //CHANGED TERMINATION TEST
            //   a[lo..i-1] < v /\ a[j+1..hi] >= v

            exch(a, i, j);
            //after swap, a[i] < v /\ a[j] >= v, hence
            //   a[lo..i] < v /\ a[j..hi] >= v
        }

        // put v = a[j] into position
        //exch(a, lo, j);              REMOVED

        //   a[lo..i-1] < v /\ a[j+1..hi] >= v /\ i = j+1
        //   a[lo..j] < v /\ a[j+1..hi] >= v /\ i = j+1
        return j;
    }


// is v = w ?
    private static boolean equal(Comparable v, Comparable w) {
        return (v.compareTo(w) == 0);
    }
```

Next, call this modified partition in succession using the pivots v_1, v_2, v_3. Will result in a sorted array.

```
private static lsort(int[] a, int v0, int v1, int v2, int v3) {
     int i1 = partition(a, 0, a.length-1, v1);
     //a[0..i1] = v0 /\ a[i1+1...N] = {v1,v2,v3}
     int i2 = partition(a, i1+1, a.length-1, v2);
     //a[0..i1] = v0 /\ a[i1+1..i2] = v1 /\ a[i2+1...N] = {v2,v3}
     int i3 = partition(a, i2+1, a.length-1, v3);
     //a[0..i1] = v0 /\ a[i1+1..i2] = v1 /\ a[i2+1...i3] = v2 /\ a[i3+1...N] = v3
}
```

(2.2) `partition` still runs in linear time, since the same argument continues to apply: each element of the array section being partitioned is acessed at most 3 times (once for comparison with the pivot, and twice for swapping).

`lsort` calls `partition` 3 times, and hence also runs in linear time.

Problem 3. (40 points) We wish to add to a regular BST a field `min` for each node `x` which records the minimum value in the subtree roooted at `x`.

(3.1) (20 points) Give a modified `put` procedure (see text, p. 399) which maintains the `min` field.

(3.2) (20 points) Prove that your modified put has worst case running time in $O(h)$, where h is the height of the BST. You do not have to handle deletion.

No credit for a put procedure with running time asymptotically larger than $O(h)$.

Sample Solution.

(3.1). Add a field `m` to the `Node` class, which maintains for each node x the minimum key that occurs in the subtree rooted at x. Upon insertion, update the `m` field for each node from the root down to the point of insertion by comparing with the key `key` to be inserted: if `key` is smaller, then set the `m` field to `key`, otherwise leave it unchanged.

```
/***********************************************************************
    *  Insert key-value pair into BST
    *  If key already exists, update with new value
    ***********************************************************************/

// add a field m to the Node class to store the min value

 private class Node {
        private Key key;           // sorted by key
        private Value val;         // associated data
        private Node left, right;  // left and right subtrees
        private int N;             // number of nodes in subtree
        private Key m;         // minimum key in subtree rooted at Node

        public Node(Key key, Value val, int N, Key m) {
            this.key = key;
            this.val = val;
            this.N = N;
            this.m = m;  //minimum key in subtree rooted at this
        }


    private Node put(Node x, Key key, Value val) {
        if (x == null) return new Node(key, val, 1, val);
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x.left  = put(x.left,  key, val);
        else if (cmp > 0) x.right = put(x.right, key, val);
        else              x.val   = val;
        x.N = 1 + size(x.left) + size(x.right);
        if (key.compareTo(x.m) < 0) x.m = key;   //UPDATE min
       return x;
    }
```

(3.2). The modification just adds an if-statement at each node on the way down. The modified `put` still traverses the BST from top to bottom, doing constant work at each node. Hence running time is still $O(h)$.