CSE 143, Winter 2012
Midterm Exam Key

1. `ArrayList` Mystery

List	Output
(a) `[5, 2, 5, 2]`	`[2, 2]`
(b) `[3, 5, 8, 9, 2]`	`[5, 9, 1, 3]`
(c) `[0, 1, 4, 3, 1, 3]`	`[1, 3, 3, 1]`

2. Recursive Tracing

Call	Result
a) `mystery(6, 3);`	`6 0 3`
b) `mystery(2, 3);`	`2 0 1 3`
c) `mystery(5, 8);`	`5 2 0 1 3 8`
d) `mystery(21, 12);`	`21 9 6 0 3 12`
e) `mystery(3, 10);`	`3 2 0 1 4 7 10`

3. Collections

```
// typical full-credit solution
public static Map<Integer, String> byAge(Map<String, Integer> ages, int min, int max) {
    Map<Integer, String> result = new HashMap<Integer, String>();   // TreeMap OK
    for (String name : ages.keySet()) {
        int age = ages.get(name);
        if (min <= age && age <= max) {
            if (result.containsKey(age)) {
                result.put(age, result.get(age) + " and " + name);
            } else {
                result.put(age, name);
            }
        }
    }

    return result;
}

// typical full-credit solution, plus variable-phobia and bad var names
public static Map<Integer, String> byAge(Map<String, Integer> m, int min, int max) {
    Map<Integer, String> m2 = new TreeMap<Integer, String>();   // HashMap OK
    for (String s : m.keySet()) {
        if (m.get(s) >= min && m.get(s) <= max) {
            if (m2.containsKey(m.get(s))) {
                m2.put(m.get(s), m2.get(m.get(s)) + " and " + s);
            } else {
                m2.put(m.get(s), s);
            }
        }
    }
    return m2;
}
```

```
// simpler if tests
public static Map<Integer, String> byAge(Map<String, Integer> ages, int min, int max) {
    Map<Integer, String> result = new HashMap<Integer, String>();   // TreeMap OK
    for (String name : ages.keySet()) {
        int age = ages.get(name);
        if (result.containsKey(age)) {
            result.put(age, result.get(age) + " and " + name);
        } else if (min <= age && age <= max) {
            result.put(age, name);
        }
    }

    return result;
}
```

4. Stacks and Queues

```
// using an auxiliary stack
public static boolean isSorted(Stack<Integer> s) {
    if (s.size() < 2) {
        return true;
    }

    boolean sorted = true;
    int prev = s.pop();
    Stack<Integer> backup = new Stack<Integer>();
    backup.push(prev);
    while (!s.isEmpty()) {
        int curr = s.pop();
        backup.push(curr);
        if (prev > curr) {
            sorted = false;
        }
        prev = curr;
    }

    while (!backup.isEmpty()) {   // restore s
        s.push(backup.pop());
    }

    return sorted;
}

// using an auxiliary stack, but put into backup first and then
// check sortedness on way back
public static boolean isSorted(Stack<Integer> s) {
    if (s.size() < 2) {
        return true;
    }

    Stack<Integer> backup = new Stack<Integer>();
    while (!s.isEmpty()) {
        backup.push(s.pop());
    }

    boolean sorted = true;
    int prev = backup.pop();
    backup.push(prev);
    while (!backup.isEmpty()) {
        int curr = backup.pop();
        s.push(curr);          // restore s
        if (curr > prev) {
            sorted = false;
        }
        prev = curr;
    }

    return sorted;
}
```

```
// using an auxiliary queue
public static boolean isSorted(Stack<Integer> s) {
    if (s.size() < 2) {
        return true;
    }
    boolean sorted = true;
    int prev = s.pop();
    Queue<Integer> backup = new LinkedList<Integer>();
    backup.add(prev);
    while (!s.isEmpty()) {
        int curr = s.pop();
        backup.add(curr);
        if (prev > curr) {
            sorted = false;
        }
        prev = curr;
    }

    q2s(backup, s);  // restore (and un-reverse) s
    s2q(s, backup);
    q2s(backup, s);
    return sorted;
}

// using an auxiliary stack; returns early but restores before doing so
public static boolean isSorted(Stack<Integer> s) {
    if (s.size() < 2) {
        return true;
    }
    int prev = s.pop();
    Stack<Integer> backup = new Stack<Integer>();
    backup.push(prev);
    while (!s.isEmpty()) {
        int curr = s.pop();
        backup.push(curr);
        if (prev > curr) {
            while (!backup.isEmpty()) {   // restore s
                s.push(backup.pop());
            }
            return false;
        }
        prev = curr;
    }

    while (!backup.isEmpty()) {   // restore s
        s.push(backup.pop());
    }
    return true;
}

// auxiliary stack, but I don't understand boolean so I use an int instead
public static boolean isSorted(Stack<Integer> s) {
    if (s.size() < 2) {
        return true;
    }
    int prev = s.pop();
    int sortedCount = 1;
    Stack<Integer> backup = new Stack<Integer>();
    backup.push(prev);
    while (!s.isEmpty()) {
        int curr = s.pop();
        backup.push(curr);
        if (prev <= curr) {
            sortedCount++;
        }
        prev = curr;
    }

    while (!backup.isEmpty()) {   // restore s
        s.push(backup.pop());
    }
    if (sortedCount == s.size()) {
        return true;
    } else {
        return false;
    }
}
```

```
// recursive!
public static boolean isSorted(Stack<Integer> s) {
    if (s.size() < 2) {
        return true;  // base case
    } else {
        int prev = s.pop();
        boolean sorted = prev <= s.peek() && isSorted(s);
        s.push(prev);
        return sorted;
    }
}
```

5. Linked Lists

```
// one-pass loop
public void minToFront() {
    if (front != null && front.next != null) {
        ListNode current = front;
        ListNode prev = front;
        while (current.next != null) {
            if (current.next.data < prev.next.data) {
                prev = current;
            }
            current = current.next;
        }

        if (prev.next != null && prev.next.data < front.data) {
            ListNode min = prev.next;
            prev.next = prev.next.next;
            min.next = front;
            front = min;
        }
    }
}

// two passes: one to find min value, one to find/move min node
public void minToFront() {
    if (front != null) {
        int min = front.data;   // figure out min value
        ListNode current = front;
        while (current != null) {
            min = Math.min(min, current.data);
            current = current.next;
        }

        if (min != front.data) {
            current = front;   // find min value node, move to front
            while (current.next.data != min) {
                current = current.next;
            }
            ListNode newFront = current.next;
            current.next = current.next.next;
            newFront.next = front;
            front = newFront;
        }
    }
}
```

6. Recursion

```
public static String moveToEnd(String s, char c) {
    if (s.length() == 0) {   // OK: s.isEmpty(),  s.equals(""),  s == ""
        return "";
    } else if (s.charAt(0) == c) {
        return moveToEnd(s.substring(1), c) + s.charAt(0);
    } else {
        return s.charAt(0) + moveToEnd(s.substring(1), c);
    }
}

public static String moveToEnd(String s, char c) {
    if (s.equals("")) {
        return s;
    }
    char first = s.charAt(0);
    s = s.substring(1);
    s = moveToEnd(s, c);
    if (first == c) {
        return s + c;
    } else {
        return first + s;
    }
}

// hackish semi-cheat solution using indexOf
public static String moveToEnd(String s, char c) {
    int i = s.indexOf(c);
    if (i < 0) {
        return s;
    } else {
        String rest = s.substring(0, i) + s.substring(i + 1);
        return moveToEnd(rest, c) + c;
    }
}

// hackish semi-cheat solution using contains and indexOf
public static String moveToEnd(String s, char c) {
    if (!s.contains("" + c)) {
        return s;
    } else {
        return s.substring(0, s.indexOf(c)) +
               moveToEnd(s.substring(s.indexOf(c) + 1), c) + c;
    }
}
```