CSE 143, Winter 2010
Midterm Exam Key

1. `ArrayList` Mystery

List	Output
(a) `[10, 20, 10, 5]`	`[0, 10, 20, 10]`
(b) `[8, 2, 9, 7, -1, 55]`	`[0, 0, 8, 9, -1, 55]`
(c) `[0, 16, 9, 1, 64, 25, 25, 14, 0]`	`[0, 0, 0, 0, 16, 9, 64, 25, 0]`

2. Recursive Tracing

Call	Value Returned
a) `mystery(7, 1)`	7
b) `mystery(4, 2)`	6
c) `mystery(4, 3)`	4
d) `mystery(5, 3)`	10
e) `mystery(5, 4)`	5

Side note: This method is actually computing *N*-choose-*k*, the number of unique combinations of *k* items that can be taken from a set of *N* choices.

3. `Comparable`

```
public class Dieter implements Comparable<Dieter> {
    ...

    public int compareTo(Dieter other) {
        if (getBMI() < other.getBMI()) {
            return -1;
        } else if (getBMI() > other.getBMI()) {
            return 1;
        } else if (height < other.getHeight()) {
            return -1;
        } else if (height > other.getHeight()) {
            return 1;
        } else if (name.compareTo(other.getName()) < 0) {
            return -1;
        } else if (name.compareTo(other.getName()) > 0) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

```
public class Dieter implements Comparable<Dieter> {
    ...

    public int compareTo(Dieter other) {
        int cmp = (int) Math.signum(getBMI() - other.getBMI());
        if (cmp != 0) {
            return cmp;
        } else if (height != other.getHeight()) {
            return height - other.getHeight();
        } else {
            return name.compareTo(other.getName());
        }
    }
}
```

4. Stacks and Queues

```
// using a temporary stack
public static void expunge(Stack<Integer> s) {
    if (!s.isEmpty()) {
        // copy sorted contents into temp stack s2
        Stack<Integer> s2 = new Stack<Integer>();
        int prev;
        while (!s.isEmpty()) {
            prev = s.pop();
            while (!s.isEmpty() && s.peek() < prev) {
                s.pop();
            }
            s2.push(prev);
        }

        // transfer s2 back into s
        while (!s2.isEmpty()) {
            s.push(s2.pop());
        }
    }
}

// using a temporary queue and s2q/q2s methods
public static void expunge(Stack<Integer> s) {
    if (s.size() > 1) {
        // transfer sorted contents into temp queue q
        Queue<Integer> q = new LinkedList<Integer>();
        int max = s.pop();
        q.add(max);
        while (!s.isEmpty()) {
            int n = s.pop();
            if (n >= max) {
                max = n;
                q.add(n);
            }
        }

        // transfer q back into s (need to do it 2x to fix ordering)
        q2s(q, s);
        s2q(s, q);
        q2s(q, s);
    }
}

// using a temporary stack, no temp variables
public static void expunge(Stack<Integer> s) {
    if (s.isEmpty() || s.size() == 1) { return; }

    // copy sorted contents into temp stack s2
    Stack<Integer> s2 = new Stack<Integer>();
    s2.push(s.pop());   // so s2 will never be empty

    while (!s.isEmpty()) {
        int n = s.pop();
        if (n > s2.peek()) {
            s2.push(n);
        }
    }

    // transfer s2 back into s
    while (!s2.isEmpty()) {
        s.push(s2.pop());
    }
}
```

5. Collections

```
// Map + separate Set solution
public static Set<String> commonFirstLetters(List<String> list) {
    // count the first letters into a map
    Map<String, Integer> m = new TreeMap<String, Integer>();
    for (int i = 0; i < list.size(); i++) {
        String first = list.get(i).toLowerCase().substring(0, 1);
        if (m.containsKey(first)) {
            m.put(first, m.get(first) + 1);
        } else {
            m.put(first, 1);
        }
    }

    // put the desired members of the map into a set
    Set<String> set = new TreeSet<String>();
    for (String s : m.keySet()) {
        if (m.get(s) >= 2) {
            set.add(s);
        }
    }
    return set;
}

// map-only solution; remove from keySet
public static Set<String> commonFirstLetters(List<String> list) {
    Map<String, Integer> m = new TreeMap<String, Integer>();
    for (String s : list) {
        String ch = s.substring(0, 1).toLowerCase();
        if (!m.containsKey(ch)) {
            m.put(ch, 0);
        }
        m.put(ch, m.get(ch) + 1);
    }

    // evict undesirable members from the map, then we'll return its key set
    // (have to use Iterator rather than just removing from Map/Set directly)
    Set<String> keySet = m.keySet();
    Iterator<String> itr = keySet.iterator();
    while (itr.hasNext()) {
        if (m.get(itr.next()) < 2) {
            itr.remove();
        }
    }
    return keySet;
}

// two-sets solution
public static Set<String> commonFirstLetters(List<String> list) {
    Set<String> set1 = new HashSet<String>();
    Set<String> set2 = new HashSet<String>();
    for (String word : list) {
        String first = word.substring(0, 1).toLowerCase();
        if (set1.contains(first)) {
            set2.add(first);     // occurs >= 2 times
        }
        set1.add(first);
    }
    return set2;
}

// "sort an array or list, then look for duplicates" solution
public static Set<String> commonFirstLetters(List<String> list) {
    String[] letters = new String[list.size()];
    for (int i = 0; i < list.size(); i++) {
        letters[i] = list.get(i).toLowerCase().substring(0, 1);
    }
    Arrays.sort(letters);
    Set<String> set = new HashSet<String>();

    for (int i = 0; i < letters.length - 1; i++) {
        if (letters[i].equals(letters[i + 1])) {
            set.add(letters[i]);
        }
    }
    return set;
}
```

6. Linked Lists

```
// two-pass intuitive solution
public void removeMin() {
    if (front == null) {
        throw new NoSuchElementException();
    }

    // walk to the end, looking for the min value
    int min = front.data;
    ListNode current = front;
    while (current != null) {
        if (current.data < min) {
            min = current.data;
        }
        current = current.next;
    }

    // remove the minimum value
    if (min == front.data) {
        front = front.next;
    } else {
        // make a second pass to find node before min value
        current = front;
        boolean found = false;
        while (!found && current.next != null) {
            if (current.next.data == min) {
                current.next = current.next.next;
                found = true;   // important; must stop to avoid removing 2x
            }
            current = current.next;
        }
    }
}


// two-pass, store min node only (not min value or prev node) solution
public void removeMin() {
    if (front == null) {
        throw new NoSuchElementException();
    }

    ListNode min = front;           // find/store min node
    ListNode current = front;
    while (current != null) {
        if (current.data < min.data) {
            min = current;
        }
        current = current.next;
    }

    if (min == front) {             // min == front case
        front = front.next;
    } else {                        // find prev node before min
        current = front;
        while (current != min && current.next != min) {
            current = current.next;
        }
        current.next = min.next;    // remove min node
    }
}
```

```
// one-pass elegant solution
public void removeMin() {
    if (front == null) {
        throw new NoSuchElementException();
    }

    // walk to end, look for min value and point to node before it
    ListNode minNode = front;
    int minValue = front.data;

    ListNode current = front;
    while (current.next != null) {
        if (current.next.data < minValue) {
            minNode = current;
            minValue = current.next.data;
        }
        current = current.next;
    }

    // remove min node through our pointer
    if (minNode == front) {
        front = front.next;
    } else {
        minNode.next = minNode.next.next;
    }
}
```