# Solution to CSE143X Section/Contest #11 Problems

Below is a list of style problems with the bad ArrayIntList:

class comment: Don't include Stuart's name, include just your name. It would also be helpful to include the date and your section or TA's name. The class comment is meaningless. Make some kind of attempt to describe what the class is used for, as in "Class ArrayIntList can be used to store a list of integers."

fields: Fields should be declared private. The field comments are useless because they are repeating the names of the fields. Only include comments if you can provide something beyond the field name. The field called "capacity" is redundant because it is equal to elementData.length.

class constant: It is improperly declared because it is missing "final". It is also improperly named because the convention for constants is to use all uppercase letters and underscore characters, as in DEFAULT_CAPACITY.

first constructor: It should have a comment and it should be using the "this(...)" notation to call the other constructor. It does not use the class constant as it should.

second constructor: Comment doesn't mention the fact that it throws an IllegalArgumentException when capacity is negative and the comment should describe more about what it does. The use of if/else is not appropriate. The convention in Java is to throw exceptions with an if and then to have the standard code follow without being inside an else.

size method: It has no comment.

get method: Comment doesn't mention what kind of exception is thrown and doesn't describe what it does. The for loop is not needed and makes the method extremely inefficient, basically negating the benefit of using an array (the random-access aspect of the array).

toString method: No comment. Spacing is terrible. Introduce spaces to make your code more readable. The indentation is also off on many lines.

contains method: The comment has implementation details, discussing the use of a for loop and the fact that it is searching an array. This method is also highly redundant. It should call indexOf.

isEmpty method: No comment and it violates boolean zen.

indexOf method: The comment has implementation details, talking about the array and the size fields. It also does not describe significant behavior: the fact that the first occurrence is returned and that a -1 is returned if not found. The implementation is horrible. It uses a variable called index that is not needed and then there is a redundant test after the loop. Even if you are going to use this index variable, then initialize it to -1 and return it after the loop instead of having the same test both inside and outside the loop.

first add method: This is a good method. You can eliminate some redundancy by having it call the other add method, but that is a subtle point that we wouldn't expect students to notice.

second add method: The exception comments are good, but the description of what it does is incomplete. It doesn't say what happens to the old value at the given index. It shifts subsequent values to the right, which should be described in the comment so that the client knows what it does.

capacity method: You are not allowed to add extra public functions to a class that weren't part of the specification. You can add private methods that are part of the implementation, but not public methods.

remove method: The comment shouldn't discuss the implementation detail of decreasing the size and it should mention what happens to the list when the value is removed. The subsequent values are shifted to the left.

overall: The lines of code to check for illegal indexes in get and remove are redundant. It would have been good to introduce a private method to eliminate the redundancy.

1. One possible solution appears below.

```
public String acronymFor(List<String> words) {
    String acronym = "";
    for (String next : words) {
        acronym += next.charAt(0);
    }
    return acronym.toUpperCase();
}
```

2. One possible solution appears below.

```
public void switchPairs(List<String> list) {
    int i = 0;
    while (i < list.size() - 1) {
        String first = list.get(i);
        list.set(i, list.get(i + 1));
        list.set(i + 1, first);
        i += 2;
    }
}
```

3. One possible solution appears below.

```
public void stutter(List<Integer> list) {
    for (int i = 0; i < list.size(); i += 2) {
        list.add(i, list.get(i));
    }
}
```

4. Two possible solutions appear below.

```
public void reverse3(List<Integer> list) {
    for (int i = 0; i < list.size() - 2; i += 3) {
        int n1 = list.get(i);
        int n3 = list.get(i + 2);
        list.set(i, n3);
        list.set(i + 2, n1);
    }
}
```

```
public void reverse3(List<Integer> list) {
    for (int i = 0; i < list.size() - 2; i += 3) {
        list.add(i, list.remove(i + 2));
        list.add(i + 2, list.remove(i + 1));
    }
}
```

5. One possible solution appears below.

```
public boolean hasOdd(Set<Integer> set) {
    for (int value : set) {
        if (value % 2 != 0) {
            return true;
        }
    }
    return false;
}
```

6. One possible solution appears below.

```
public Set<Integer> removeEvens(Set<Integer> s) {
    Set<Integer> result = new TreeSet<Integer>();
    Iterator<Integer> i = s.iterator();
    while (i.hasNext()) {
        int n = i.next();
        if (n % 2 == 0) {
            result.add(n);
            i.remove();
        }
    }
    return result;
}
```

7. One possible solution appears below.

```
public boolean containsAll(Set<Integer> s1, Set<Integer> s2) {
    Iterator<Integer> i = s2.iterator();
    while (i.hasNext()) {
        if (!s1.contains(i.next())) {
            return false;
        }
    }
    return true;
}
```

8. One possible solution appears below.

```
public boolean equals(Set<Integer> s1, Set<Integer> s2) {
    if (s1.size() != s2.size()) {
        return false;
    }
    for (int n : s1) {
        if (!s2.contains(n)) {
            return false;
        }
    }
    return true;
}
```

9. One possible solution appears below.

```
public void retainAll(Set<Integer> s1, Set<Integer> s2) {
    Iterator<Integer> i = s1.iterator();
    while (i.hasNext()) {
        if (!s2.contains(i.next())) {
            i.remove();
        }
    }
}
```

10. map: {baz=c, mumble=d, foo=a, bar=b}
    set returned: [a, b, c, d]

    map: {f=z, d=x, e=y, b=y, c=z, a=x}
    set returned: [x, y, z]

    map: {f=2, g=10, d=20, e=1, b=10, c=2, a=1, h=20}
    set returned: [1, 10, 2, 20]

11. One possible solution appears below.

```
    public Map<Integer, Integer> counts(List<Integer> list, Set<Integer> set) {
        Map<Integer, Integer> counts = new TreeMap<Integer, Integer>();
        for (int value : list) {
            if (set.contains(value)) {
                if (counts.containsKey(value)) {
                    counts.put(value, counts.get(value) + 1);
                } else {
                    counts.put(value, 1);
                }
            }
        }
        return counts;
    }
```

12. One possible solution appears below.

```
  public Map<Integer, Set<String>> split(Set<String> words) {
      Map<Integer, Set<String>> buckets = new TreeMap<Integer, Set<String>>();
      for (String word : words) {
          int n = word.length();
          if (!buckets.containsKey(n)) {
              buckets.put(n, new TreeSet<String>());
          }
          buckets.get(n).add(word);
      }
      return buckets;
  }
```

13. One possible solution appears below.

```
    public Map<String, Integer> reverse(Map<Integer, String> map) {
        Map<String, Integer> result = new TreeMap<String, Integer>();
        for (int key : map.keySet()) {
            String value = map.get(key);
            result.put(value, key);
        }
        return result;
    }
```

14. One possible solution appears below.

```
public int maxOccurrences(List<Integer> list) {
    Map<Integer, Integer> counts = new TreeMap<Integer, Integer>();
    for (int value : list) {
        if (counts.containsKey(value)) {
            counts.put(value, counts.get(value) + 1);
        } else {
            counts.put(value, 1);
        }
    }
    int max = 0;
    for (int count : counts.values()) {
        max = Math.max(max, count);
    }
    return max;
}
```

15. One possible solution appears below.

```
public Map<String, Set<String>> convert(Set<String> numbers) {
    Map<String, Set<String>> result = new TreeMap<String, Set<String>>();
    for (String s : numbers) {
        String exchange = s.substring(0, 3);
        String digits = s.substring(4);
        if (!result.containsKey(exchange)) {
            result.put(exchange, new TreeSet<String>());
        }
        result.get(exchange).add(digits);
    }
    return result;
}
```

16. One possible solution appears below.

```
public Map<String, Set<List<String>>> acronyms(Set<List<String>> lists) {
    Map<String, Set<List<String>>> result =
        new TreeMap<String, Set<List<String>>>();
    for (List<String> words : lists) {
        String acronym = acronymFor(words);
        if (!result.containsKey(acronym)) {
            result.put(acronym, new HashSet<List<String>>());
        }
        result.get(acronym).add(words);
    }
    return result;
}
```

17. One possible solution appears below.

```
public Set<Point> extractEqual(Set<Point> data) {
    Set<Point> result = new HashSet<Point>();
    for (Point p : data) {
        if (p.getX() == p.getY()) {
            result.add(p);
        }
    }
    return result;
}
```