CS 125 Practice Final Exam Solutions

—SOLUTION SET—

Practice exam questions are intended to represent the types and difficulty level of questions you should expect to encounter on the real final exam. However, since one of the main goals of the practice exam is to give you practice writing code using pen and paper, questions that require code writing may be overrepresented.

Answer each question as **clearly** and **succinctly** as possible. Draw pictures or diagrams if they help. The point value assigned to each question may help you allocate your time. **No aids of any kind are permitted.**

1. (10 points) Modeling an Airport

Write a Java class to represent an airport. You should model the aspects of airports that are important to planes as they land, drop off and pick up new passengers, and take off again.

Specifically, your airport class should model the following airport *properties* (2 points):

- a unique identifier: "CMI" for the University of Illinois Willard Airport[1]
- location: CMI is located at 40°02′21″N 88°16′41″W
- some number of runways: CMI has three runways
- some number of gates: CMI has three gates

You can assume that the number of runways and gates does not change once the airport is created.

You should also model the following airport *operations* (6 points):

- **Landing:** a plane should be able to determine if it can land at a specific airport, and on which runway. Note that a plane should only be able to land if there is both a runway and gate available.
- **Parking:** once a plane lands it should be directed to a gate. You should ensure that two planes do not attempt to park at the same gate—this tends to annoy passengers and pilots.
- **Departing:** at some point a plane will request to leave its gate and take off via a runway. You should not allocate the same runway for a simultaneous takeoff and landing! Once plane has successfully cleared the airport it should notify your airport that it has departed and the runway is now free.

You can assume that a `Plane` class exists and has implemented `equals` correctly, allowing you to uniquely identify each plane that uses your airport.

Finally, your class should provide a way to look up an airport by its identifier (2 points). As you provide this feature, you can assume that no two airports are ever created with the same identifier, and that no more than some number of airports will ever be created.

[1] Fun fact: the University of Illinois is the only university that operates an airport.

Solution for Question 1

Solution:

The entire solution is long, so let's break it into parts.

Here's a snippet showing instance variables and a complete constructor:

```
1  /**
2   * A class that models an airport.
3   */
4  public class Airport {
5      /** Our identifier. */
6      private String identifier;
7
8      /** Our location latitude. */
9      private double latitude;
10
11     /** Our location longitude. */
12     private double longitude;
13
14     /** How many runways our airport has. */
15     private int numRunways;
16
17     /** Runway usage. */
18     private Plane[] runwayUsage;
19
20     /** How many gates our airport has. */
21     private int numGates;
22
23     /** Gate usage. */
24     private Plane[] gateUsage;
25
26     /**
27      * Create a new airport with the specified properties.
28      *
29      * @param setIdentifier the string code for the new airport
30      * @param setLatitude the latitude of the new airport
31      * @param setLongitude the longitude of the new airport
32      * @param setNumRunways the number of runways the new airport has
33      * @param setNumGates the number of gates the new airport has
34      */
35     public Airport(final String setIdentifier, final double setLatitude,
36             final double setLongitude, final int setNumRunways, final int setNumGates) {
37         super();
38         identifier = setIdentifier;
39         latitude = setLatitude;
40         longitude = setLongitude;
41         numRunways = setNumRunways;
42         runwayUsage = new Plane[numRunways];
43         numGates = setNumGates;
44         gateUsage = new Plane[numGates];
45         for (int i = 0; i < MAX_AIRPORTS; i++) {
46             if (allAirports[i] == null) {
47                 allAirports[i] = this;
48             }
49         }
50     }
51 }
```

Here's a snippet showing the `land` instance method. This one is probably the most complicated, since it needs to use both the runway and gate arrays.

```
1  /**
2   * Attempt to land at the given airport.
3   *
4   * @param plane the plane that is attempting to land
5   * @return the runway to use on success, OPERATION_FAILED if all runways are busy.
6   */
7  public int land(final Plane plane) {
8      /*
9       * Make sure we can allocate both a runway and a gate before setting the plane in our arrays
10      * and returning success.
11      */
12     int runwayToUse = OPERATION_FAILED;
13     for (int i = 0; i < runwayUsage.length; i++) {
14         if (runwayUsage[i] == null) {
15             runwayToUse = i;
16             break;
17         }
18     }
19     if (runwayToUse == OPERATION_FAILED) {
20         return OPERATION_FAILED;
21     }
22     int gateToUse = OPERATION_FAILED;
23     for (int i = 0; i < gateUsage.length; i++) {
24         if (gateUsage[i] == null) {
25             gateToUse = i;
26             break;
27         }
28     }
29     if (gateToUse == OPERATION_FAILED) {
30         return OPERATION_FAILED;
31     }
32     /*
33      * At this point we have a runway and a gate, so assign them and return the runway.
34      */
35     runwayUsage[runwayToUse] = plane;
36     gateUsage[gateToUse] = plane;
37     return runwayToUse;
38 }
```

Here's a snippet showing the `park` instance method.

```
1  /**
2   * Park at the gate reserved for this plane.
3   *
4   * Also mark the runway that was used as available.
5   *
6   * @param plane the plane that is attempting to park
7   * @return the gate to park at, or OPERATION_FAILED if this plane has not already successfully
8   *         requested to land
9   */
10 public int park(final Plane plane) {
11     boolean previouslyLanded = false;
12     for (int i = 0; i < runwayUsage.length; i++) {
13         if (runwayUsage[i] != null && runwayUsage[i].equals(plane)) {
14             /*
15              * Deallocate this runway.
16              */
17             previouslyLanded = true;
18             runwayUsage[i] = null;
19             break;
20         }
21     }
22     /*
23      * This plan must not have previously landed. Fail the operation.
24      */
25     if (!previouslyLanded) {
26         return OPERATION_FAILED;
27     }
28
29     for (int i = 0; i < gateUsage.length; i++) {
30         if (gateUsage[i] != null && gateUsage[i].equals(plane)) {
31             return i;
32         }
33     }
34     /* Should not get here. */
35     return OPERATION_FAILED;
36 }
```

Here are the rest of the instance methods:

```
1  /**
2   * Leave the gate and prepare to take off.
3   *
4   * Also mark the runway that was used as available.
5   *
6   * @param plane the plane that is attempting to leave the gate
7   * @return the runway to take off from, or OPERATION_FAILED if a runway is not currently
8   *         available
9   */
10 public int leaveGate(final Plane plane) {
11     int runwayToUse = OPERATION_FAILED;
12     for (int i = 0; i < runwayUsage.length; i++) {
13         if (runwayUsage[i] == null) {
14             runwayToUse = i;
15             runwayUsage[i] = plane;
16             break;
17         }
18     }
19     if (runwayToUse == OPERATION_FAILED) {
20         return OPERATION_FAILED;
21     }
22     for (int i = 0; i < gateUsage.length; i++) {
23         if (gateUsage[i] != null && gateUsage[i].equals(plane)) {
24             gateUsage[i] = null;
25         }
26     }
27     return runwayToUse;
28 }
29
30 /**
31  * Leave airport.
32  *
33  * Marks the runway that was in use for departure as available.
34  *
35  * @param plane the plane that is attempting to take off
36  * @return true on success, or false if this plane does not currently have a runway
37  */
38 public boolean leaveAirport(final Plane plane) {
39     for (int i = 0; i < runwayUsage.length; i++) {
40         if (runwayUsage[i] != null && runwayUsage[i].equals(plane)) {
41             /*
42              * Deallocate this runway.
43              */
44             runwayUsage[i] = null;
45             return true;
46         }
47     }
48     return false;
49 }
```

And finally, here are the static variables and method:

```
1  /**
2   * A class that models an airport.
3   */
4  public class Airport {
5      /** Maximum number of airports in our system. */
6      public static final int MAX_AIRPORTS = 1024;
7
8      /** Operation failure code. */
9      public static final int OPERATION_FAILED = -1;
10
11     /** Array holding all created airports. */
12     private static Airport[] allAirports = new Airport[MAX_AIRPORTS];
13
14     /**
15      * Lookup an airport by its identifier.
16      *
17      * @param identifier the identifier to search for.
18      * @return the airport with this identifier, or null on failure
19      */
20     public static Airport lookupAirport(final String identifier) {
21         for (int i = 0; i < MAX_AIRPORTS; i++) {
22             if (allAirports[i] != null && allAirports[i].identifier.equals(identifier)) {
23                 return allAirports[i];
24             }
25         }
26         return null;
27     }
28 }
```

Rubric:

As embedded in the question. This question would require a great deal of time to answer. If a question like this appears on the final exam, expect it to be worth more points and to have more time to complete it.

2. (10 points) Debugging

The following function contains bugs. Identify the bugs (5 points) using the existing code snippet, and then rewrite the code so that it is correct (5 points).

```
1  /**
2   * Count duplicate strings in an array.
3   *
4   * A string is a duplicate if it equals any other string in the array.
5   * For example:
6   *   countDuplicates({ "a", "b" }) should return 0
7   *   countDuplicates({ "a" }) should return 0
8   *   countDuplicates({ "a", "a" }) should return 2
9   *   countDuplicates({ "a", "b", "a", "a" }) should return 3
10  */
11 public long countDuplicates(final String[] strings) {
12   int count = 0;
13   for (int i = 0; i < strings.length; i++) {
14     for (int j = 0; j < strings.length; j++) {
15       if (strings[i] == strings[j]) {
16         count++;
17       }
18     }
19   }
20   return count;
21 }
```

Solution:

Bugs are marked in the code snippet below.

```
1  // Syntax error: method is declared to return long but returns int
2  public long countDuplicates(final String[] strings) {
3    // Logic error: strings is not checked for null
4    int count = 0;
5    for (int i = 0; i < strings.length; i++) {
6      for (int j = 0; j < strings.length; j++) {
7        // Logic error: items are compared against themselves
8        // Logic error: should use .equals, not ==
9        if (strings[i] == strings[j]) {
10         // Logic error: all matches are counted, rather than just the first
11         count++;
12       }
13     }
14   }
15   return count;
16 }
```

Here is a corrected snippet:

```
1  public int countDuplicates(final String[] strings) {
2    if (strings == null) {
3      return 0;
4    }
5    int count = 0;
6    for (int i = 0; i < strings.length; i++) {
7      for (int j = 0; j < strings.length; j++) {
8        if (i != j && strings[i].equals(strings[j])) {
9          count++;
10         break;
11       }
12     }
13   }
14   return count;
15 }
```

Rubric:

As embedded in the question: 1 point per bug, and 5 points for a correct solution.

3. (10 points) Tree Recursion

Recall the binary tree structure that we used on MP6:

```
1  public class Tree {
2
3      /* Current node's parent. May be null if I'm the root of the tree. */
4      private Tree parent;
5
6      /* Current node's left child, or null if none. */
7      private Tree left;
8
9      /* Current node's left child, or null if none. */
10     private Tree right;
11
12     /* Current node's value. */
13     private int value;
14 }
```

And recall that when we insert a new element into the tree, we put it:

- *to the left* if the new value is less than the node we are comparing against, and
- *to the right* if the new value is greater than the node that we are comparing against.

Previously we did not consider how to handle duplicate elements[2]. First, determine how to extend our binary tree to handle duplicate elements (4 points) You will either need to modify the tree node definition, or the insertion algorithm.

Next, write a recursive algorithm that, given the root of the tree, counts the total number of duplicate elements (6 points)—similar to the previous question. This algorithm must work on the tree that can handle duplicates that you designed above.

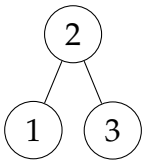
[2] I realize that duplicates are becoming a theme of this practice exam.

Solution:

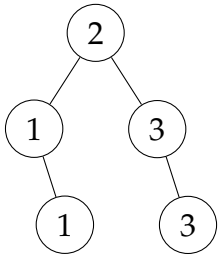
There were (at least) two ways to approach this problem: either modify the insertion algorithm to handle duplicates, or change the tree object structure itself. We'll cover both, although modifying the tree object is probably the better solution as it leads to a simpler recursive algorithm.

To modify the insertion algorithm to handle duplicates, have it simply put them to the right. So if the value of the item being inserted is the same as the current item, either (a) at it as the right child of the current node if it does not have a right child, or (b) traverse the right child and try again.

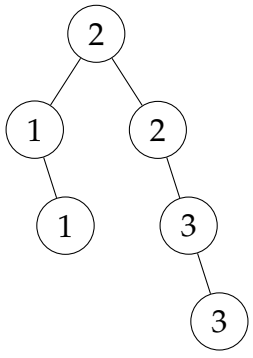
So, for example, if we start with this tree:



and then insert 1 and then 3, we get the following tree:



This is simple enough—as long as we are adding duplicates of the *leaves*. But what about if we wanted to add another 2? The resulting tree *should* look like this:



But that complicates our insertion algorithm further, since it now needs to potentially move an entire portion of the tree to accommodate non-leaf duplicates.

Here is Java code for the modified insertion algorithm. Note that it would require our *Tree* object to have setters for its *parent* and *right* instance variable. We would have accepted pseudocode for this question, or just a written description of the insertion algorithm similar to what we provided for MP6.

```
1  /**
2   * Insert a new value into the binary tree.
3   * <p>
4   * Insertion <i>should not</i> fail if the value already exists in the tree.
5   *
6   * @param newValue the new value to insert
7   */
8  public void insert(final int newValue) {
9      if (newValue == value) {
10         Tree toInsert = new Tree(newValue, this);
11         if (right != null) {
12           right.setParent(toInsert);
13           toInsert.setRight(right);
14           right = toInsert;
15         } else {
16           right = toInsert;
17         }
18     } else if (newValue > value) {
19         if (right != null) {
20             return right.insert(newValue);
21         } else {
22             right = new Tree(newValue, this);
23         }
24     } else {
25         if (left != null) {
26             return left.insert(newValue);
27         } else {
28             left = new Tree(newValue, this);
29         }
30     }
31 }
```

With this strategy, the following recursive summation algorithm will return a count of the number of duplicate values in the tree:

```
1  /**
2   * Count the number of duplicate values in a tree.
3   */
4  public int countDuplicates() {
5      int duplicateCount = 0;
6      if (right != null && value == right.value) {
7        if (parent != null && value == parent.value) {
8          /* We were already counted by our parent, so only count
9           * the duplicate below us.
10          */
11         duplicateCount = 1;
12       } else {
13         /* We've counted two new duplicates: us, and the item
14          * below us. */
15         duplicateCount = 2;
16       }
17     }
18     if (this.left != null) {
19       duplicateCount += left.countDuplicates();
20     }
21     if (this.right != null) {
22       duplicateCount += right.countDuplicates();
23     }
24     return duplicateCount;
25 }
```

This is a bit subtle, since we need to avoid double counting. So we need to check both our parent's value and our right child's value. If both are equals to our own, then we're in the middle of a run and should only count our child. If our right child's value is equal to ours, but our parent's is not, then we're at the start of a run and need to count both us and our child.

Now let's consider the other approach. We simply modify our `Tree` object as follows:

```
1  public class Tree {
2      /* Current node's parent. May be null if I'm the root of the tree. */
3      private Tree parent;
4
5      /* ... rest omitted ... */
6
7      /* Count of insertions with this value. Initialized to 1. */
8      private int count;
9  }
```

This requires no change to our tree structure and only a minor modification to our insertion algorithm:

```
1  /**
2   * Insert a new value into the binary tree.
3   * <p>
4   * Insertion <i>should not</i> fail if the value already exists.
5   *
6   * @param newValue the new value to insert
7   */
8  public void insert(final int newValue) {
9      if (newValue == value) {
10         this.count++;
11     } else if (newValue > value) {
12         if (right != null) {
13             return right.insert(newValue);
14         } else {
15             right = new Tree(newValue, this);
16         }
17     } else {
18         if (left != null) {
19             return left.insert(newValue);
20         } else {
21             left = new Tree(newValue, this);
22         }
23     }
24 }
```

The recursive summation is also simpler:

```
1  /**
2   * Count the number of duplicate values in a tree.
3   */
4  public int countDuplicates() {
5      int duplicateCount = this.count - 1;
6      if (this.left != null) {
7        duplicateCount += left.countDuplicates();
8      }
9      if (this.right != null) {
10       duplicateCount += right.countDuplicates();
11     }
12     return duplicateCount;
13 }
```

Between the two options, this was definitely the way to go.

Rubric:

As embedded in the question. We would accept pseudocode or written descriptions in lieu of actual code for this question. On the exam we will make it *extremely clear* when we expect near-correct Java code and when a written description or pseudocode will be sufficient. However, you are always welcome to write actual code—sometimes is the fastest way to get your point across clearly.

4. (10 points) Quicksort Pivoting

First, provide a brief high-level overview of the quicksort algorithm (2 points). Feel free to use pseudocode if this is easier for you.

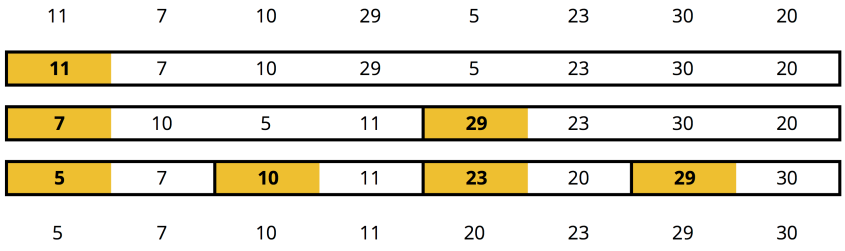
Second, explain why the choice of pivot value is so important to quicksort. Using a diagram and a small example, describe what occurs in the best-case selection of pivot value (4 points), and also in the worst case (4 points). Make sure to describe the $\mathcal{O}(n)$ runtime in both cases.

Solution:

Quicksort is a divide-and-conquer sorting algorithm. It proceeds by repeatedly partitioning the array into smaller and smaller subarrays, each of which is then partitioned further. Partitions are created by choosing a *pivot value*, with values smaller than the pivot being put into one partition and items greater into another. Once the subarray is of size 1 it is sorted by definition, and Quicksort will also sort an array of size 2 with further partitioning regardless of which pivot value is chosen. Because of how partitions are created, two sorted partitions combine immediately to form one sorted partition.

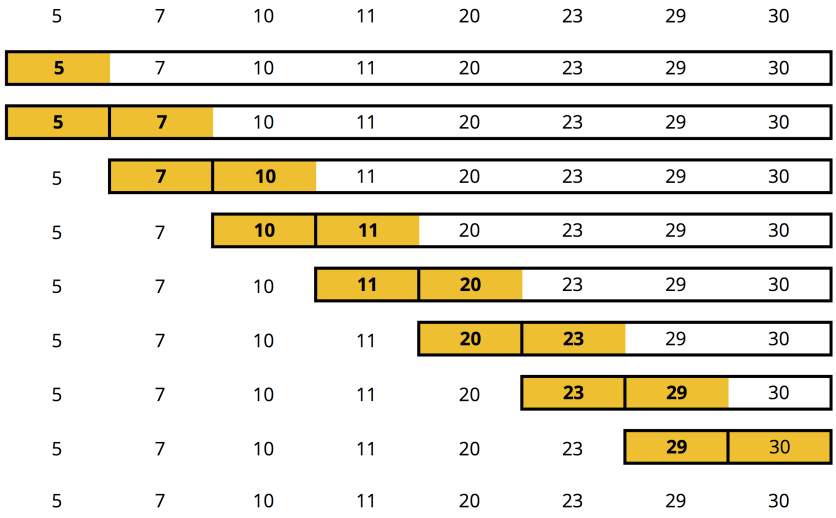
Quicksort's performance depends heavily on the choice of pivot value. A good pivot value will divide an array of size n into two arrays of size roughly $\frac{n}{2}$, allowing Quicksort to achieve $\mathcal{O}(n\ log\ n)$ performance. In the best case it performs n comparisons on $log\ n$ levels, hence $\mathcal{O}(n\ log\ n)$. However, in the worst case a pivot value divides an array of size N into one array of size 1 and a second of size $n-1$. In the worst case it performs n comparisons on n levels, thus achieving $\mathcal{O}(n^2)$—equivalent to much worse sorting algorithms like selection sort and bubble sort.

Quicksort performance depends on both the contents of the input array and the choice of pivot value. Here's an example of best-case performance achieved on random data and by selecting the first item in the sub-array as the pivot value. Pivot values are highlighted in each step, while individual steps in each partitioning are not shown.

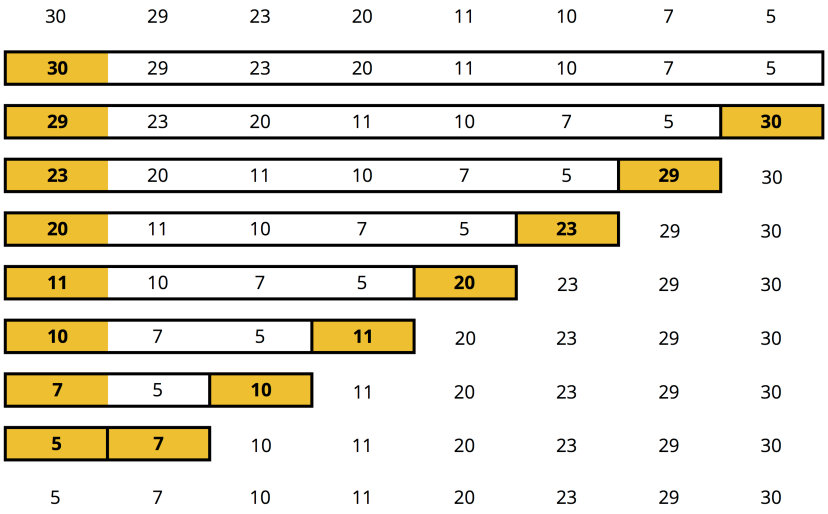


As is shown, sorting an array of size 8 takes three levels—$3 = log_2 8$.

In contrast, here's an example of worst-case performance achieved by sorting a random data and again selecting the first item as the pivot:



This can also happen if the array was sorted in reverse order:



In both these pathological cases, Quicksort is only sorting one item per iteration. As a result, the depth will be n, each level requiring n comparisons, resulting in $\mathcal{O}(n^2)$ performance. Yuck.

Rubric:

As embedded in the question.