

READ THIS PAGE FIRST. *Please do not discuss this exam with people who haven't taken it.* Your exam should contain 7 problems on 18 pages. Officially, it is worth 50 points.

This is an open-book test. You have three hours to complete it. You may consult any books, notes, or other inanimate objects available to you. You may use any program text supplied in lectures, problem sets, or solutions. Please write your answers in the spaces provided in the test. Make sure to put your name, login, and lab section in the space provided below. Put your login and initials *clearly* on each page of this test and on any additional sheets of paper you use for your answers.

Be warned: my tests are known to cause panic. Fortunately, this reputation is entirely unjustified. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious. Should you feel an attack of anxiety coming on, feel free to jump up and run around the building once or twice.

Your name: _____

Login: _____

1. _____/8

2. _____/6

Login of person to your Left: _____ Right: _____

3. _____/7

Discussion section number or time: _____

4. _____/10

Discussion TA: _____

5. _____/12

Lab section number or time: _____

6. _____/0

7. _____/7

Lab TA: _____

TOT _____/50

1. [8 points] Answer “true” or “false” for each of the following statements about Java, and give a short explanation (≤ 20 words) for each answer. *IMPORTANT*: you *must* give an explanation for each to get credit!

- a. If a function that executes the following fragment returns a negative `int` value, then `x` must have been larger than 15.

```
x = g (y) >>> 1;
return x << 27;
```

- b. Because of its catch clauses, the program fragment below can go into an infinite loop if `readData` throws `IOExceptions`.

```
try {
    result = null;
    result = readData (0);
} catch (IOException e) {
    reportProblem (result.message);
} catch (NullPointerException e) {
    /* Try alternate channel. */
    result = readData (1);
}
```

- c. If the following code fragment prints “<0” then `x` must be an instance variable or a static (class) variable.

```
x = 1;
f (x);
if (x < 0)
    System.err.println (“<0”);
```

- d. Given the following (partial) class definition:

```
package things;
public class Widget {
    protected int x;
    public void f (int y, util.Zapper z) {
        z.execute (this, y);
    }
}
```

If every line of Java code that is executed during the call to `z.execute` is contained in the class `util.Zapper` (`util` is a package), then the call to `execute` might modify `this.x` if `Zapper` is a subtype of `Widget`.

- e. Given the following (partial) class definition:

```
package things;
public class Widget extends util.Zapper {
    protected int x;
    public void f (int y, util.Zapper z) {
        z.execute (this, y);
    }
}
```

If every line of Java code that is executed during the call to `z.execute` is contained in the class `util.Zapper` (`util` is a package), then the call to `execute` might modify `this.x`.

- f. The program fragment below prints "0":

```
String[] items = new String[3];
System.out.println (items[0].length ());
```

- g. The second method below is a non-destructive version of the first method.

```
GenList limit (GenList L, int lim) {  
    for (GenList p = L; p != null; p = p.tail)  
        p.head = Math.max (p.head, lim);  
    return L;  
}
```

```
GenList limit (GenList L0, int lim) { // Non-destructive ?  
    GenList L = L0;  
    for (GenList p = L; p != null; p = p.tail)  
        p.head = Math.max (p.head, lim);  
    return L;  
}
```

- h. Suppose that P and Q are different packages. If a class P.Secret is not public and contains a method F, then in a class Q.User, there is no way to call F directly (that is, in Q.User, a statement of the form

something.F (...)

cannot be calling P.Secret.F).

2. [6 points] The following questions concern an alternative to Dijkstra's algorithm. Assume that there are N vertices numbered $1 \dots N$. Assume also that `edgeLength(x, y)` returns the length of the edge between vertices x and y in constant time, with `edgeLength(x, x) = 0` and `edgeLength(x, y) = ∞` if there is no edge between x and y .

- a. For the definition of `shortestPathLength` below, demonstrate that the worst-case time for computing `shortestPathLength(1,2)` is $\Omega(2^N)$. (Hints: this is a loose bound; you can consider just part of the execution of SPL to get it. The time required to compute SPL depends only on k , whose initial value comes from `shortestPathLength`.)

```
/** The length of the shortest path between vertex V0 and vertex V1. */
int shortestPathLength (int v0, int v1) {
    return SPL (v0, v1, N);
}

/** The length of the shortest path between vertex V0 and V1 that uses
 * only vertices numbered <= K (except possibly for the two endpoints V0 and
 * V1). */
int SPL (int v0, int v1, int k) {
    if (v0 == v1)
        return 0;
    int len;
    len = edgeLength (v0, v1);
    for (int i = 1; i <= k; i += 1)
        len = Math.min (len, SPL (v0, i, i-1) + SPL (i, v1, i-1));
    return len;
}
```

- b. Here is most of an alternative, memoized algorithm for the same problem. Fill in the blanks to make it work. In addition to the assumptions at the beginning of the problem, you may also assume that $\text{edgeLength}(x, y) > 0$ if $x \neq y$.

```

_____ memo = new _____;

static int SPL (int v0, int v1, int k) {

    if (_____) {
        int len;
        len = edgeLength (v0, v1);
        for (int i = 1; i <= k; i += 1)
            len = Math.min (len, SPL (v0, i, i-1) + SPL (i, v1, i-1));
        _____ = len;
    }

    return _____;
}

```

- c. Given the definition of SPL in part (b) above, give a bound on the time to compute the following loop (as a function of N , the number of vertices):

```

for (int v0 = 1; v0 <= N; v0 += 1)
    for (int v1 = 1; v1 <= N; v1 += 1)
        shortestDist[v0][v1] = shortestPathLength (v0, v1);

```

3. [7 points, inspired by a posting from A. Podgornaia] In this problem, we ask you to translate an English description of an idea into Java. There's a good deal of reading here, but you shouldn't need to write very much in response.

For a standard Java `HashSet<T>` to work, it is necessary that the items of type `T` currently stored in it never change the values that `.hashCode` or `.equals` return for them. In this problem, we look at a general technique (or *design pattern*) for getting around that. To be concrete, consider the following class:

```
class StringHolder {
    private String val;

    StringHolder (String initial) {
        val = initial;
    }

    public boolean equals (Object x) {
        return val.equals (x);
    }

    public int hashCode () {
        return val.hashCode ();
    }

    void put (String val) {
        this.val = val;
    }

    String get () {
        return val;
    }
}
```

If we create some of these and insert them into a `HashSet` like this:

```
HashSet<StringHolder> S = new HashSet<StringHolder> ();
StringHolder[] builders = new StringHolder[N];
for (int i = 0; i < N; i += 1) {
    builders[i] = new StringHolder (A[i]);
    S.add (builders[i]);
}
```

where `A` is an array of Strings, then `S.contains(builders[i])` will be true for $0 \leq i < N$.

a. Explain why `S.contains(builders[i])` may no longer be true after executing

```
for (int i = 0; i < N; i += 1) {
    builders[i].put (B[i]);
}
```

even though `S` will still contain each of the objects `builders[i]`.

Continued on next page

Now let's fix the problem raised in part (a) by defining a new kind of `HashSet` that requires its elements to implement an interface `Observable`. We'll call this new type `HashMutableSet`. The idea is that `Observable` objects define a method that allows another object (in our case, a `HashMutableSet`) to "register" itself as an *observer* of the `Observable` object. At any given time, an `Observable` may be observed by any number of objects (e.g., a `StringHolder` might belong to any number of `HashMutableSets` simultaneously).

To function as an observer, a class (in our case, `HashMutableSet`) must implement another interface, which we'll call `Observer`. An `Observer` provides two methods by which an `Observable` object that it has registered with can inform the `Observer` of changes to that `Observable` object. The `Observable` object calls one method on all of its `Observers` just *before* it makes some change to itself, and it calls the other method just *after* it makes some change to itself. For both methods, the `Observable` passes itself (the object that is changing) as an argument.

So the idea is that when a `HashMutableSet` .`add`s an object, *A*, to itself, it first registers itself to observe *A*. If a method of *A* wants to change *A*'s value, then it first tells all of *A*'s registered observers (including our `HashMutableSet`) that *A* is about to change, and after carrying out the change, it tells the same observers that *A*'s value has changed.

The rest of this problem involves implementing this idea.

- b. Define the interfaces `Observable` and `Observer`. These interfaces must *not* be specific in any way to `HashMutableSet` or `StringHolder`.

```
public interface Observable {
```

```
}
```

```
public interface Observer {
```

```
}
```

Continued on next page

- c. Now implement an `Observable` extension of `StringHolder`. This must work with *any* kind of `Observer`, not just `HashMutableSets`.

```
class ObservableStringHolder extends StringHolder implements Observable {  
  
    ObservableStringHolder (String initial) {  
        super (initial);  
    }  
  
    // FILL IN AS NEEDED (our solution uses about 11 non-blank lines)
```

```
}
```

Continued on next page

- d. Finally, implement `HashSet` as an extension of `HashSet`. Beyond the new methods you must implement, just worry about the `.add` method; we won't worry about removing objects. Again, this must work with *any* kind of `Observable`, not just `ObservableStringHolder`.

```
class HashSet<T _____>
    extends HashSet<T> implements Observer
{

    /** Add OBJ this THIS. Returns true iff OBJ is not already
     * in THIS. */
    public boolean add (T obj) { // Our solution has 5 non-blank lines

}

// FILL IN AS NEEDED (our solution is 6 non-blank lines)

}
```

4. [10 points] The following questions involve sorting. Warning: do not assume that the algorithms illustrated always conform exactly to those presented in the reader and lecture notes. We are interested in whether you understand the major ideas behind the algorithms. Where the question asks for a reason, you *must* provide an explanation to get credit.

- a. Could these be major steps from a run of quicksort? Why or why not?

```
dze ccf bkw hwy pjk xce aux qtr xpa atm
atm ccf bkw hwy pjk xce aux qtr xpa dze
atm aux bkw hwy pjk xce ccf qtr xpa dze
atm aux bkw ccf pjk xce hwy qtr xpa dze
atm aux bkw ccf dze xce hwy qtr xpa pjk
atm aux bkw ccf dze hwy xce qtr xpa pjk
atm aux bkw ccf dze hwy pjk qtr xpa xce
atm aux bkw ccf dze hwy pjk qtr xce xpa
```

- b. Nunne T. Wisely wrote a program that he thought performed an LSD radix sort. Here is an illustration at major steps of the algorithm, starting with the input. As you can see it gets the wrong answer in this case. Describe as succinctly as possible how he's apparently messed up the algorithm.

```
dze ccf bkw cwy pjk xce dzx ctr xpa pjm
xpa xce dze ccf pjk pjm ctr bkw dzx cwy
ccf xce pjm pjk bkw xpa ctr cwy dzx dze
bkw cwy ctr ccf dze dzx pjk pjm xpa xce
```

- c. What sorting algorithm does the following illustrate? The first line represents the input and the remainder are contents of the output array at points in the algorithm where it changes ('--' means "not yet assigned to").

```
14 13 10 17 23 26 07 24 29 04
-- -- -- -- 14 -- -- -- -- --
-- -- -- 13 14 -- -- -- -- --
-- -- 10 13 14 -- -- -- -- --
-- -- 10 13 14 17 -- -- -- --
-- -- 10 13 14 17 23 -- -- --
-- -- 10 13 14 17 23 -- 26 --
-- 07 10 13 14 17 23 -- 26 --
-- 07 10 13 14 17 23 24 26 --
-- 07 10 13 14 17 23 24 26 29
04 07 10 13 14 17 23 24 26 29
04 07 10 13 14 17 23 24 26 29
```

d. What sorting algorithm does the following illustrate?

```
dze ccf hwy pjk bkw xce aux qtr xpa atm
ccf dze hwy bkw pjk aux xce qtr atm xpa
ccf dze bkw hwy pjk aux xce atm qtr xpa
bkw ccf dze hwy pjk atm aux qtr xce xpa
atm aux bkw ccf dze hwy pjk qtr xce xpa
```

e. What sorting algorithm does the following illustrate?

```
dze ccf bkw hwy pjk xce aux qtr xpa atm
dze ccf bkw hwy pjk xce aux qtr atm xpa
dze ccf bkw hwy pjk xce aux atm qtr xpa
dze ccf bkw hwy pjk xce atm aux qtr xpa
dze ccf bkw hwy pjk atm aux qtr xce xpa
dze ccf bkw hwy atm aux pjk qtr xce xpa
dze ccf atm aux bkw hwy pjk qtr xce xpa
dze atm aux bkw ccf hwy pjk qtr xce xpa
atm aux bkw ccf dze hwy pjk qtr xce xpa
```

5. [12 points] Answer each of the following *briefly*. Where a question asks for a yes/no answer, give a brief reason for the answer (or counter-example, if appropriate).

a. Suppose that $f_1(x), f_2(x), \dots$ are all (*possibly different*) functions, each of which is in $O(1)$. Define $g(n) = \sum_{1 \leq i \leq n} f_i(n)$. Show how it is possible for $g(n)$ to be in $\Omega(N^2)$.

b. We represented the heap data structure (used for priority queues) with an array. But a heap is a form of binary tree. Why don't we use arrays for all binary trees?

c. We can explore the nodes of a binary tree one level at a time using either breadth-first search, in which we visit each node once, or by iterative deepening, where we traverse nodes by depth-first search down to a maximum depth multiple times, increasing the maximum depth by one each time. In the best case for iterative deepening, if we visit N nodes by breadth-first search, how many nodes do we visit using iterative deepening?

- d. If we have space to store M nodes (in addition to the space already occupied by our tree), how deep a tree can we explore using breadth-first search (see problem c)?
- e. Suppose we have a heap in which the top value is the largest. At what depths in the tree (distances from the root) can the fourth-largest value occur? Give examples in which it occurs at the minimal possible depth and at the maximal possible depth.
- f. If a 2-4 tree has a height of h (that is, if the lowest nodes that contain keys are at a distance of h edges from the root), what is the maximum possible height of a corresponding red-black tree? Give your reasoning.

6. [1 point] In what state was the first 911 call made?
7. [7 points] In the following (partial) implementation of a trie data structure, there are several errors at *some* of the points indicated by “// ERROR?”. When “// ERROR?” occurs alone on a line, it means that there *might* be a missing statement. Correct these *as succinctly as possible*. Do *not* correct things that don’t need to be corrected (not all indicated points are erroneous).

```
/** A set of Strings. TrieSets are initially empty. They may be added to,
 * but Strings cannot be removed. */
public class TrieSet {

    /** Set THIS to the union of THIS with { X }. Return true iff
     * THIS changes as a result (i.e., X was not previously present).
     * If y is a String in THIS that is not equal to X, then X is assumed
     * not to begin with y and y is assumed not to begin with X. */
    public boolean add (String x) {
        int size0 = root.size ();
        root = root.insert (x, 0);
        return size0 != root.size ();
    }

    /** True iff THIS contains X */
    public boolean contains (String x) {
        return root.contains (x, 0);
    }

    /** Number of (distinct) Strings in THIS. */
    public int size () {
        return root.size ();
    }

    private Node root = new Empty ();
}
```

Continued on next page

```

/* Representation: The set is represented as a Trie.  A Node that is
 * k levels below the root is either a Leaf nodes (containing a String),
 * or an Inner node, containing one or more children, one for each
 * different value of character #k of the Strings stored in the subtrees
 * under it.  An Inner node that has N children contains two arrays, one,
 * kids, containing the children, and one, keys containing the characters
 * that lead to the corresponding child.  So a node, X, with two children
 * Y and Z that look like this:

```

```

 *
 *
 *           X
 *          / \
 *         'b' / \ 'q'
 *          /   \
 *         Y     Z
 *
 * contains the arrays
 *
 * keys: { 'b', 'q' }
 * kids: { Y, Z }

```

```

 *
 * The root node ("0 levels below the root") may also be an Empty node,
 * containing nothing.  Inner nodes do not contain their level.  Instead,
 * they are told their level whenever they are traversed.
 */

```

```

private static abstract class Node {
    /** Assuming THIS is K levels below the root, return the result of
     * inserting X if it is not present. */
    abstract Node insert (String x, int k);
    /** True IFF X is stored in THIS, assuming that THIS is K levels below
     * the root. */
    abstract boolean contains (String x, int k);
    /** The number of strings in THIS subtree. */
    abstract int size ();
}

private static class Empty extends Node {
    Node insert (String x, int k) {
        return this;
        // ERROR?
    }
    boolean contains (String x, int k) {
        return false;
    }
    int size () {
        return 0;
    }
}

```

Continued on next page

```
private static class Leaf extends Node {
    Leaf (String x) {
        val = x;
    }

    Node insert (String x, int k) {

                                                // ERROR?

        Node result = new Inner ();
        result.insert (val, k);
        result.insert (x, k);
        return result;
    }

    boolean contains (String x, int k) {
        return val == x;
                                                // ERROR?
    }

    int size () {
        return 1;
                                                // ERROR?
    }

    private String val;
}
```

Continued on next page

```
private static class Inner extends Node {
    private char[] keys = null;           // ERROR?
    private Node[] kids = null;          // ERROR?
    private int size = 0;

    int size () {
        return size;
    }

    boolean contains (String x, int k) {
        for (int i = 0; i < kids.length; i += 1)
            if (keys[i].equals (x.charAt (k))) // ERROR?
                return kids[i].contains (x, k); // ERROR?
        return false;
    }

    Node insert (String x, int k) {
        for (int i = 0; i < keys.length; i += 1) {
            if (keys[i] == x.charAt (k)) { // ERROR?
                int size0 = kids[i].size ();
                kids[i] = kids[i].insert (x, k+1);
                size += kids[i].size (); // ERROR?
                return this;
            }
        }
        Node[] newKids = new Node[kids.length+1];
        char[] newKeys = new char[keys.length+1];
        System.arraycopy (kids, 0, newKids, 0, kids.length);
        System.arraycopy (keys, 0, newKeys, 0, kids.length);
        newKids[kids.length] = new Leaf (x);
        newKeys[kids.length] = x.charAt (k);

        // ERROR?

        size += 1; // ERROR?
        return this;
    }
}

}
```