Last name: **First name:** **NetID**:

Question 0.	_________	(out of 02)
Question 1.	_________	(out of 13)
Question 2.	_________	(out of 18)
Question 3.	_________	(out of 12)
Question 4.	_________	(out of 20)
Question 5.	_________	(out of 10)
Question 6.	_________	(out of 10)
Question 7.	_________	(out of 15)
Total	___________	(out of 100)

Submit all regrade requests by 8PM TONIGHT. Use the CMS where possible. Regrades for prelims will not be considered.

Grades for the final will be posted on the CMS early next week. Course grades will be posted several days later. You can look at your final only when you return in the Spring. HAVE A GREAT HOLYDAY BREAK!

You have 2.5 hours to complete the questions in this exam, which are numbered 0..7, worth 100 points total. Please glance through the whole exam before starting.

Question 0 (2 points). Print your name and Cornell **NetID** at the top of each page. Please make them legible.

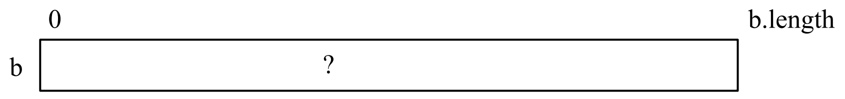
Question 1 (13 points). Algorithms. Write procedure `selectionSort` (see below). Work as follows:

(1) We give the pre- and post-conditions as diagrams. You draw the loop invariant in the place indicated below, as a diagram. Everything you write after that should depend on the invariant, so if the invariant is wrong, few points can be given.

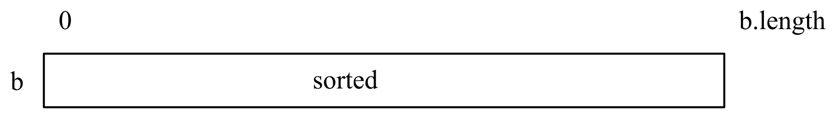
(2) Develop the loop using the four loopy questions and the precondition, postcondition, and invariant. (You *don't* have to write the four loopy questions themselves.)

(3) We expect the loop repetend to be written with high-level but precise statements written in English and math notation, saying *what* the repetend has to accomplish, not *how* to do it, as done in lectures and the text.

precondition:



postcondition:



invariant:

/** Sort b using selection sort, using the conventional lexical (alphabetical) order of Strings. */
public static void selectionSort(String[] b) {

}

Question 2 (18 points) This question is designed to test your knowledge of two-dimensional arrays, catching exceptions, throwing exceptions, and loops.

Complete the body of function `sumEntries`, given below. Do not use recursion. In writing the body, you will write a loop that processes a range of integers. The loop must have a suitable loop invariant.

You may wish to make use of function `Integer.parseInt(String s)`, which parses `s` as a signed decimal integer and throws a `NumberFormatException` if `s` does not contain a parsable integer.

```
/** = the sum of the ints represented by the String elements of row b[j].
    ("14" represents an int; neither "a" nor "fifty" do.
    Throw an IllegalArgumentException with detail message "an element of b[j]
    is not an int" if any of b[j]'s elements do not represent ints.
    Precondition: b is not null, none of its elements are null, and 0 <= j < b.length.*/
public static int sumEntries(String[][] b, int j) {









}
```

Last name: **First name:** **NetID**:

Question 3 (12 points). Recursion, loops, ragged arrays, real and apparent types

Write the recursive function specified below to count the number of **`null`** values in its parameter `b`. `b` is an object of class `Object`, or class `Object[]` (1-dimensional array), or class `Object[][]` (2-dimensional array), or class `Object[][][]` (3-dimensional array), etc. This is neat!
Example: for the following object, the answer is 3:

```
new Object[]{null, 3, new Object[]{null, new Object[]{true, null}}}
```

Here are some ideas.

1. You can use operation **`instanceof`** to determine whether some base case holds. But be careful. *Every* object is an instance of class `Object`.
2. If parameter `b` is an array (of any number of dimensions), you can cast it to type `Object[]` and then use it as an array, meaning you can write a for-loop that processes its elements. How will you process its elements? (Writing an invariant is suggested but not required.)

```
/** = Number of nulls in b, that is:
        if b is null, the answer is 1.
        if b is not null and is not an array, the answer is 0.
        if b is an array, the answer is the sum of the numbers of nulls in b's elements.  */
public static int nulls(Object b) {



}
```

Question 4 (20 points). Classes and subclasses. This question and the next are designed to test your knowledge of and ability to write classes, to deal with issues of inheritance, overriding, casting, constructors, and so forth, and to read class invariants and specifications of methods carefully.

This question concerns maintaining information about a class at Cornell (which, to avoid confusion with the word "class" in Java, we call a "course meeting"). An object of class `CourseMtg` (see below) maintains a course name, the instructors involved with it, and the list of registered students, sometimes called the *roster*. Note carefully that the instructors are stored in an array, but the students are stored in the `Vector` that is the superclass of `CourseMtg`. Whether extending Vector in this manner is good design is a matter of debate, but it does help us test your Java and programming fluency.

Methods in a Vector object v	
v.size()	= number of elements in v
v.add(ob);	append object ob to v
v.get(i)	= element v[i] of v
v.contains(ob)	= "v contains object ob"
v.toString()	= comma-separated list of elements of v, delimited by [ ]

(a) State here the purpose of a constructor. Then complete the body of the constructor of `CourseMtg` below. Your constructor can use other methods of `CourseMtg`.

(b) Complete the body of function `add` of `CourseMtg` (on the next page).

(c) Complete the body of function `equals` of `CourseMtg` (on the next page). If you write a loop to process a range of integers, you must give a suitable loop invariant.

```
/** An instance is a course meeting. It maintains the name of the course, the roster (a list of netIDs
    of students registered for it), and a list of netIDs of instructors teaching it.*/
public abstract class CourseMtg extends Vector<String>{

   /* This class extends Vector. The Vector contains the roster —netIDs of all students in the course. */

   private String name; // Name of course meeting, e.g. "CS 1110". It is not null.

   private String[] instructors; // netIDs of instructors (at least one). It is not null, and has no null entries.
                                 // The netIDs are in alphabetical order.

   /** A new instance with name name, instructor list b, and no students.
      Precondition: name, b, and b's elements are non-null, b's elements are netIDs.
                    b is not necessarily in alphabetical order.  */
   public CourseMtg(String name, String[] b) {










   } // (class continued on the next page)
```

(d) Complete the body of function `toString` of `CourseMtg` (below). You may use function `nameAndInstr`, which appears just above it, and any useful methods of class `Vector`. You may not use a loop.

```
/** If student with netID n is not in this course's roster, add this student to the roster.
      Precondition: n is a valid netID. */
  public void add(String n) {

  }

   /** = "ob is a CourseMtg with the same name and same set of instructors as this one." */
  public boolean equals(Object ob) {

        // you may not use nameAndInstr here.

  }
  /** Sort b, using selection sort. */
  public static void selectionSort(String[] b) {
     // You do not have to write this procedure. You did it in question 1.
  }
  /** =  a String that gives the name and list of instructors in this course */
  public String nameAndInstr() {
    // You do not have to write this function.
  }
  /** = a String that contains the name, list of instructors, and list of students (as netIDs)*/
  public String toString() {

  }
}
```

Question 5 (10 points). Subclasses. A course, or instance of `CourseMtg`, always has a lecture, and it may have a set of recitation or lab sections, as does CS 1110. Students register in the lecture and in a section (if there are sections). To maintain this kind of information, we have two other classes, `Lecture`, declared to the right, and `Section`, given below. We show only components that are of interest for this question.

Class `Lecture` maintains a list of `Section`s associated with it, while class `Section` contains a field for the `Lecture` with which it is associated. Things get intertwined.

We have completed the only relevant method in class `Lecture`, the constructor. Field `secList` is public in order to simplify your work.

```
/** An instance is a lecture, with associated list
    of sections */
public class Lecture extends CourseMtg {

  /** List of sections associated with lecture.
      May not be null, can be empty */
  public Vector<Section> secList=
                     new Vector<Section>();

  /** Constructor: a Lecture with name n,
      instructor list ls, no sections, no students.
      Precondition: no arguments are null and
      ls contains a nonempty list of netIDs. */
  public Lecture(String n, String[] ls)
     {  super(n, ls);  }
}
```

Your task is to complete the constructor and method `add` in class `Section`. Be careful. You would be wise to draw an object of each relevant class so that you know what fields of each have to be assigned to to keep class invariants true.

```
/** An instance is a section associated with a lecture */
public class Section extends CourseMtg {
  // The lecture with which this section is associated. It is not null
  private Lecture mainLecture;

  /** Constructor: a Section with name n, instructor list ls, no students, and connected with lec.
      Precondition: no arguments are null and ls contains a nonempty list of netIDs */
  public Section(String n, String[] ls, Lecture lec) {









  }

  /** Add student n (a netID) to this section's roster, if not in (and to the roster of the associated
     mainLecture, if not there)
     Precondition: n is not null and is a netID.*/
  public void add(String n) {









  }
}
```

Last name: **First name:** **NetID**:

Question 6 (10 points) Executing Java statements. To the right is a class `Homework`.

(a) Fill in the 3 blanks in class `Homework` so that its class invariant is true.

(b) Below is a sequence of statements. First, draw all variables declared in this sequence and the static variables of class `Homework`. Then, execute the sequence, using the variables you drew, and making sure to draw all objects that are created. You need not draw the `Object` partition.

```
Homework d= new Homework("A1", 4);
Homework f= d;
d.expand();
Homework e= new Homework("A2", 2);
d= e;
e.expand();
```

```
public class Homework {
  private String title; // Homework title
  private int probs; // no. of problems in Homework
  /* no. of Homeworks created */
  private static int numH= 0;
  /* no. of problems in all Homeworks created */
  private static int numP= 0;
  /** Constructor: new Homework with title t
     and n problems.   Precondition: n > 0 */
  public Homework(String t, int n) {
     title= t;   probs= n;
     numH= ______________________;
     numP= _______________________;
  }
  /** = no. of problems in this Homework */
  public int getSize() {  return probs;  }
  /** = average no. of problems per Homework */
  public static double getAvgProbs() {
     return (double)numP / numH;
  }
  /** Add a problem to this Homework */
  public void expand()  {
     probs= probs + 1;
     numP= _______________________;
  }
}
```

(c) Suppose these two statements are executed:

```
Homework c1=  new Homework("A3", 5);
Homework c2=  new Homework("A3", 5);
```

To the right of each expression, write its value:

```
c1 == c2

c1.equals(c2)
```

Question 7 (15 points) Miscellaneous topics

(a) What three things must be done to be able to listen to an event in a GUI?

(b) Draw a template for the frame for a method call. (We gave you a template in the lecture in which we first introduced frames for calls, and you used the template to draw actual frames. Drawing an example instead of a template will not receive much credit.)

(c) Define the terms *parameter*, *argument*, *local variable*, and *inside-out rule*.