Problem Set 6

*part I due by 10:00 p.m. on Monday, July 21*

*part II due by 10:00 p.m. on Wednesday, July 23*

**Preliminaries**

In your work on this assignment, make sure to abide by the policies on academic conduct described in the [syllabus](#).

If you have questions while working on this assignment, please come to office hours, post them on Piazza, or email libs111@fas.harvard.edu (removing the spaces).

**Important note:** In your work on this and subsequent problem sets, you should **not** use any of Java's built-in collection classes (e.g., `ArrayList`) or utility classes (e.g., `Arrays`), unless a problem explicitly states that you may do so.

**Part I: Short-Answer Problems (35-40 points total)**

*All of Part I is pair-optional. See the [syllabus](#) for a reminder of the rules for working with a partner.*

Because we are asking you to draw a diagram, you may submit either a plain-text file or a PDF file. Give it the name `ps6_partI.txt` or `ps6_partI.pdf`, and put your name and email address at the top of the file.

1. **Memory management and arrays** (5 points)

   Consider the following lines of Java code:
   ```java
   int[] a = {2, 4, 6, 8, 10, 12};
   int[] b = new int[6];
   int[] c = new int[6];
   b = a;
   for (int i = 0; i < a.length; i++)
     c[i] = a[i];
   a[2] = c[5];
   c[2]++;
   ```

   a) (3 pts.) Draw a single memory diagram that shows the **final** result of these lines. Include both the stack and the heap in your diagram. You may assume that these lines are part of the `main` method.

   b) (2 pts.) Indicate what will be printed by the final line of code shown above.
2. **Array practice** (5-10 points total; 5 points each part)

   In this problem, you will write one or two static methods that operate on arrays. These methods do *not* need to use recursion, and they do not need to be implemented as part of a class. Simply include the methods with your answers for the other Part I problems.

   a) Write a method with the header

   ```java
   public static boolean isSorted(int[] arr)
   ```

   that takes a reference to an array of integers and returns `true` if the array is sorted (i.e., if the elements are in increasing order), and `false` otherwise. If the parameter is `null`, the method should throw an `IllegalArgumentException`. If passed an array with length 0 or an array with one element, the method should return `true` – although you shouldn't need to include special code for these cases.

   b) *(required for grad-credit students; "partial" extra credit for others)*

   Write a method with the header

   ```java
   public static int mostFrequentValue(int[] arr)
   ```

   that takes a reference to a *sorted* array of integers and returns the value that occurs most frequently in the array. For example, consider this array:

   ```java
   int[] arr = {1, 2, 3, 3, 8, 8, 8, 8, 11, 11, 11, 14, 19, 19};
   ```

   The method call `mostFrequentValue(arr)` should return 8, because 8 is the most frequently occurring value in the array, appearing four times.

   If two or more values tie for the most occurrences, return the one that comes first. For example, if we added an extra 11 to the array shown above – giving that value four occurrences as well – the method should still return 8, because 8 comes before 11 in the array. If all of the values occur exactly once, the first element should be returned.

   The array is guaranteed to be in sorted order, and you may assume that it has at least one element. If there is only one element, it is the value that should be returned. *For full credit, you should not use another array as part of your solution, and you should not perform more than one scan through the array from left to right.*
3. **Recursion and the runtime stack** (10 points)

Consider the following recursive method:

```java
public static int mystery(int a, int b) {
    if (a < 0)
        return 1;
    else
        return 2 + mystery(a - b, b);
}
```

a) (3 pts.) Trace the execution of `mystery(20, 6)`. Use indentation to indicate which statements are performed by a given invocation of the method, following the approach used in the lecture notes to trace the `sum` method.

b) (2 pts.) What is the value returned by `mystery(20, 6)`?

c) (2 pts.) During the execution of `mystery(20, 6)`, method frames are added and then removed from the stack. How many method frames are on the stack when the base case is reached? You should assume that the initial call to `mystery(20, 6)` is made from within the `main` method, and you should include the stack frame for `main` in your count.

d) (3 pts.) Give an example of values of `a` and `b` that would produce infinite recursion, and explain why it would occur.

4. **Tracing a recursive palindrome tester** (5 points)

As we discussed in unit 4, a palindrome is a string that reads the same in both directions. For example, "dad", "radar", and "gohangasalamiimalasagnahog" are all palindromes. The following is pseudocode for a method that tests if a string is a palindrome:

```java
palindrome(String str) {
    if (str is the empty string or str is of length 1)
        return true;
    else if (str's first and last characters are the same)
        return palindrome(str minus its first and last characters);
    else
        return false
}
```

Trace the execution of this method on the string "racecar". Show the value of `str` in each invocation of the method, and indicate the return value as each call of the method completes.
5. **Rewriting a method** (10 points)
   Consider the following method, which uses iteration (a `for` loop) to search for an item in an array of integers. The method returns `true` if the item is found in the array, and `false` if it is not.

   ```java
   public static boolean search(int item, int[] arr) {
       for (int i = 0; i < arr.length; i++) {
           if (arr[i] == item) {
               return true;
           }
       }
       return false;
   }
   ```

   a. (4 points) Rewrite this method so that it searches for an item an array that can contain *any type of object*. Change the types of the parameters accordingly, and make whatever changes are needed to the body of the method.

   b. (6 points) Rewrite your answer to part a so that it uses recursion instead of iteration. You will need to add a third parameter (call it `i`) that keeps track of where you are in the array. More precisely, `i` will specify the position in the array where the search for `item` should begin. For example, `search("hello", arr, 0)` should search for "hello" in the full array (beginning at position 0), whereas `search("hello", arr, 2)` should search for "hello" in the subarray that begins at position 2 and goes to the end of the array.

---

**Part II: Programming Problems (65-70 points total)**

**Getting started:** Download the starter files for Part II from the assignments page of the course website.

1. **Adding ArrayBag methods** (25-30 pts total; 5 pts each part; `pair-optional`)
   In the file `ArrayBag.java`, add the methods described below to the `ArrayBag` class, and then add code to the `main()` method to test these methods. **In addition, you should update the Bag interface that we have given you in Bag.java to include these new methods. You should not add any new fields to the class.**

   a) `public int roomLeft()`
   This method should return the number of additional items that the `ArrayBag` has room to store. For example, if the maximum size of the bag is 10 and there are currently 7 items in the bag, this method should return 3. **Hint:** This method should only need one or two lines of code.
b) public boolean isEmpty()
   This method should return true if the ArrayBag is empty, and false otherwise.

c) public void increaseCapacity(int increment)
   This method should increase the maximum capacity of the bag by the specified amount. For example, if b has a maximum capacity of 10, then b.increaseCapacity(5) should give b a maximum capacity of 15. As part of your implementation, you will need to create a new array with room to support the new maximum capacity, copy any existing items into that array, and replace the original array with the new one by storing its reference in the called object. If increment is 0, the method should just return without making any changes to the called object. If the value of increment is negative, the method should throw an IllegalArgumentException. See our second ArrayBag constructor for an example of throwing an exception.

d) public boolean addItems(Bag other)
   This method should attempt to add to the called ArrayBag all of the items found in the parameter other. If there is room for all of the items to be added, the items are added and the method returns true. If there isn't enough room for all of the items to be added, none of them are added and the method returns false. The method should return true if the bag represented by other is empty. If the parameter is null, the method should throw an IllegalArgumentException. Note: that the parameter is of type Bag. As a result, your method should use method calls to access the internals of that bag. See our implementation of the containsAll() method for an example of this.

e) (required for grad-credit students; "partial" extra credit for others)
   public Bag intersectionWith(Bag other)
   This method should create and return an ArrayBag containing one occurrence of any item that is found in both the called object and the parameter other. For full credit, the resulting bag should not include any duplicates. For example, if b1 represents the bag {2, 2, 3, 5, 7, 7, 8} and b2 represents the bag {2, 3, 4, 5, 5, 6, 7}, then b1.intersectionWith(b2) should return an ArrayBag representing the bag {2, 3, 5, 7}. If there are no items that occur in both bags – including cases in which one or both of the bags are empty – the method should return an empty ArrayBag. If the parameter is null, the method should throw an IllegalArgumentException. Give the new ArrayBag a maximum size that is equal to the number of items in the bag with the smaller number of items. The note from part d also applies here. The return type is also Bag, but polymorphism allows you to just return the ArrayBag that you create, because ArrayBag implements Bag.
2. **Recursion and strings** (15 points total; 5 points each part; *individual-only*)

In a file named `StringRecursion.java`, implement the methods described below, and then create a `main()` method to test these methods. Your methods **must** be recursive; *no credit will be given for methods that employ iteration*. In addition, *global variables (variables declared outside of the method) are not allowed*. You may find it helpful to employ the `substring`, `charAt`, and `length` methods of the `String` class as part of your solutions.

a) `printReverse()` should take a `String` as a parameter and use recursion to print it in reverse order. The method should have the following header:

   ```java
   public static void printReverse(String str)
   ```

   For example, `printReverse("Harvard")` should print `dravraH`

   If the parameter is `null` or the empty string (`""`), the method should not print anything. This method should *not* call the `reverse` method that you write for part b.

b) `reverse()` should take a `String` as a parameter and use recursion to create and return a `String` that is the reverse of the original one. The method should have the following header:

   ```java
   public static String reverse(String str)
   ```

   This problem is similar to part a, but the method should *return* the reversed string rather than printing it. If the parameter is `null`, the method should return `null`. If the parameter is the empty string, the method should return the empty string.

c) `trim()` should take a `String` as a parameter and use recursion to create and return a `String` in which any leading and/or trailing spaces in the original string are removed. The method should have the following header:

   ```java
   public static String trim(String str)
   ```

   For example, `trim(" hello world ")` should return "hello world" and `trim("recursion ")` should return "recursion". If the parameter is `null`, the method should return `null`. If the parameter is the empty string, the method should return the empty string. The `String` class comes with a built-in `trim()` method that does the same thing as the method that we're asking you to write; you may *not* use that method in your solution!
3. **A Sudoku solver** (25 points; pair-optional)

In this problem, you will write a program that solves Sudoku puzzles using recursive backtracking.

A Sudoku puzzle consists of a 9x9 grid in which some of the cells have been filled with integers from the range 1-9. To solve the puzzle, you fill in the remaining cells with integers from the same range, such that each number appears exactly once in each row, column, and 3x3 subgrid. The left-hand image below shows an example of an initial puzzle, and the right-hand image shows its solution.

Most of the functionality of your program should go in a class called `Puzzle`, which you will use to represent an individual Sudoku puzzle. We have provided you with skeleton code for this class in the file `Puzzle.java`, including:

- a field called **values** that refers to a two-dimensional array of integers. This array is used to store the current contents of the cells of the puzzle, such that `values[r][c]` stores the current value in the cell at row r, column c of the puzzle. A value of 0 is used to indicate a blank cell.

- a field called **valIsFixed** that refers to a two-dimensional array of booleans. It is used to record whether the value in a given cell is fixed (i.e., part of the original puzzle). `valIsFixed[r][c]` is true if the value in the cell at row r, column c is fixed, and `valIsFixed[r][c]` is false if the value in that cell is not fixed. For example, in the original puzzle above, there is a fixed 4 in the cell at row 0, column 1 (the second cell in the top row), and thus `valIsFixed[0][1]` would be true for that puzzle.

- a field called **subgridHasValue** that refers to a three-dimensional array of booleans. This array allows us to determine if a given 3x3 subgrid of the puzzle already contains a given value. For example, `subgridHasValue[0][0][4]`
will be true if the 3x3 subgrid in the upper left-hand corner of the board (a
subgrid that we are identifying using the indices [0][0]) already has a 4 in
one of its cells. See the comments accompanying this field for more information
about the numbering of the subgrids.

• partial implementations of methods called placeValue() and
removeValue() that place a value in a given cell and remove a value from a
given cell by updating the fields mentioned above. You will need to add code to
these methods to update the other fields that you add.

• a full implementation of a method called readFrom() that takes a Scanner as
its only parameter and reads in a specification of a puzzle from that Scanner.
This method assumes that a puzzle specification consists of nine lines of text –
one for each row of the puzzle – and that each line contains nine digits
separated by whitespace. Here again, 0 is used to indicate a blank cell. For
example, the specification of the initial puzzle above would begin:

0 4 0 0 0 3 7 0 0
0 8 9 7 0 0 1 0 0
...

The method reads in the puzzle specification and makes the corresponding
changes to the fields mentioned above. You should not need to change this
method, because it calls placeValue(), and you are already modifying that
method as needed. However, we do recommend that you read this method over.

• the full implementation of a method called display() that prints out the
current state of the Puzzle object on which it is invoked.

• the skeleton of a method called solve() that you will implement. This is the
recursive-backtracking method, and it should return true if a solution to the
puzzle is found and false if no solution has been found (i.e., if the method is
backtracking). If the initial call to this method returns false, that means that
no solution can be found – i.e., that the initial puzzle is not valid. If there is
more than one solution (which can happen if the initial puzzle does not have
enough numbers specified), your code should stop after finding one of them.

Each invocation of the solve() method is responsible for finding the value of
a single cell of the puzzle. The method takes a parameter n, which is the
number of the cell that the current invocation of this method is responsible for.
We recommend that you consider the cells one row at a time, from top to
bottom and left to right, which means that they would be numbered as follows:

0 1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16 1
18 ...
• a partial implementation of a constructor. You will need to add code to initialize the fields that you add.

In addition to completing the methods mentioned above, you should also add to the Puzzle class whatever additional fields or methods that are needed to maintain the state of a Sudoku puzzle and to solve it using recursive backtracking. In particular, we recommend adding two fields: one to keep track of whether a given row already contains a given value, and one to keep track of whether a given column already contains a given value.

We have also given you a separate file (Sudoku.java) that contains the main() method for your Sudoku solver. You shouldn’t need to change this method (or any other code in this file), but we encourage you to review its use of the methods of the Puzzle class. In particular, note that it displays the puzzle after the solution is found, and thus it isn’t necessary for your recursive-backtracking method to display it. Finally, we have given you four sample puzzle files: puzzle1.txt, puzzle2.txt, no_solution.txt (which has no solution), and multi_sol.txt (which has multiple solutions).

Additional hints and suggestions:

• Take advantage of the second template in the notes for recursive backtracking – the one in which the method returns a boolean.

• As mentioned above, the recursive solve method takes a single parameter n that represents the number of the cell that the current invocation is responsible for. You will need to use n to compute the row and column indices of the cell, and you should be able to do so using simple arithmetic operations (+, −, *, /, and %).

• Make sure that you take advantage of the subgridHasValue field – along with the fields that you will add to keep track of the values in a given row and a given column of the puzzle – when deciding whether a particular value can be assigned to a particular cell. You should not need to scan through the puzzle to determine if a given value is valid. See the notes for n-Queens for another example of efficient constraint checking.

• Make sure that you use the addValue() and removeValue() methods when updating the state of the puzzle, and that you add code to these methods as needed to update the fields that you add to the Puzzle class.

• Make sure that you don’t attempt to assign a new number to cells that are filled in the original puzzle – i.e., cells whose values are fixed. Your solve() method will need to skip over these cells somehow.

A sample run of the program is shown below.
Please enter the name of puzzle file: puzzle2.txt

Here is the initial puzzle:

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th></th>
<th></th>
<th></th>
<th>3</th>
<th>7</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>7</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>5</td>
<td>1</td>
<td>8</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>1</td>
<td>9</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Here is the solution:

```
<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>2</th>
<th>9</th>
<th>8</th>
<th>3</th>
<th>7</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>8</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

Submitting Your Work
You should use the ps6 folder in the homework submissions dropbox to submit the following files:

- your ps6_partI.txt or ps6_partI.pdf file containing your part I answers
- your modified Bag.java file
- your modified ArrayBag.java file
- your StringRecursion.java file
- your modified Puzzle.java file.

Make sure that these classes have the correct names, paying attention to the use of uppercase and lowercase letters in the names. If any of your classes have incorrect names, fix them, and recompile your code until you have made all of the necessary changes.
Here are the steps:

- Go to the homework submissions dropbox (logging in as needed using the Login link in the upper-right corner, and entering your Harvard ID and PIN).
- Open the folder for ps6.
- Upload each of your files into this folder.
- If you worked on one or more pair-optional problems with a partner, you should click on the Comment link for the relevant files and include a comment that specifies that the name of your partner and the problems that you worked on together.
- In addition, you should click on the link for each file to view it so that you can ensure that you submitted the correct file. We will not accept any files after the fact, so please check your submission carefully.

Note: If you encounter problems submitting your files, close your browser and start again, or try again later if you still have time. If you are unable to submit and it is close to the deadline, email your homework before the deadline to libs111@fas.harvard.edu (with the spaces removed).