Problem Set 1

Due before the start of lecture on September 26, 2018.
See below for a summary of the policies regarding late submissions.

Preliminaries

Homework is due prior to the start of lecture. If it is submitted more than 10 minutes after the start of lecture, it will be considered a full day late. There will be a 10% deduction for late submissions that are made by 11:59 p.m. on the Sunday after the deadline, and a 20% deduction for submissions that are made after that Sunday and before the start of the next lecture. ***We will not accept any homework that is more than 7 days late.*** Plan your time carefully, and don't wait until the last minute to begin an assignment. Starting early will give you ample time to ask questions and obtain assistance.

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 attend office hours, post them on Piazza, or email `cscie22@fas.harvard.edu`.

Part I: Short-answer problems

30-35 points total

If you haven't already done so, you should complete Problem Set 0 before beginning this assignment.

Answers to Part I problems should typically be submitted as plain-text files, with no formatting of any kind. However, 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 `ps1_partI.txt` or `ps1_partI.pdf`.

Java built-in classes

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.

Problem 1: Memory management and arrays

5 points

Make sure that you read the notes above before you continue.

Consider the following lines of Java code:

```
int[] a = {2, 4, 6, 8, 10, 12};
int[] b = new int[6];
for (int i = 0; i < b.length; i++) {
    b[i] = a[i];
}
int[] c = b;

c[4] = a[5];
a[5] = b[4];
b[5] = a[4];

System.out.println(a[5] + " " + b[5] + " " + c[5]);
```

1. (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.
2. (2 pts.) Indicate what will be printed by the final line of code shown above.

Problem 2: Array practice

5-10 points total; 5 points each part

1. Write a method with the header

   ```
   public static void swapPairs(int[] arr)
   ```

   that takes a reference to an array of integers `arr` and swaps adjacent pairs of elements: `arr[0]` with `arr[1]`, `arr[2]` with `arr[3]`, etc. For example, consider this array:

   ```
   int[] values = {0, 2, 4, 6, 8, 10};
   ```

   After calling `swapPairs(values)`, the contents of the `values` array should be `{2, 0, 6, 4, 10, 8}`. In an odd-length array, the last element should not be moved. If the parameter is `null`, the method should throw an `IllegalArgumentException`.

This method does *not* need to use recursion, and it does not need to be implemented as part of a class. Simply include your implementation with your other answers for Part I.

2. *(required for grad-credit students; "partial" extra credit for others)*
   Write a method with the header

   ```
   public static int longestSorted(int[] arr)
   ```

   that takes a reference to an array of integers and returns the length of the longest sorted sequence of integers in the array. For example, consider the following array:

   ```
   int[] arr = {3, 8, 6, 14, -3, 0, 14, 207, 98, 12};
   ```

   The method call `longestSorted(arr)` should return `4`, because the longest sorted sequence in the array has four values in it (the sequence `-3, 0, 14, 207`). Note that the sorted sequence could contain duplicates. For example:

   ```
   int[] arr2 = {17, 42, 3, 5, 5, 5, 8, 4, 6, 1, 19};
   ```

   The method call `longestSorted(arr2)` should return `5` for the length of the sequence `3, 5, 5, 5, 8`. If the parameter is `null`, the method should throw an `IllegalArgumentException`. Your method should return `0` if passed an empty array (i.e., an array of length 0), and it should return `1` if passed an array with one element or an array in which all of the elements are in decreasing order.

   This method does *not* need to use recursion, and it does not need to be part of a class. Simply include it with your other answers for Part I.

Problem 3: Recursion and the runtime stack

10 points

Consider the following recursive method:

```
public static int mystery(int a, int b) {
    if (a <= b) {
        return a;
    } else {
        return b + mystery(a - 3, b + 1);
    }
}
```

1. (3 pts.) Trace the execution of `mystery(10, 1)`. 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.

2. (2 pts.) What is the value returned by `mystery(10, 1)`?

3. (2 pts.) During the execution of `mystery(10, 1)`, 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(10, 1)` is made from within the `main()` method, and you should include the stack frame for main in your count.

4. (3 pts.) Are there any initial values of the parameters `a` and `b` that would produce infinite recursion? Explain briefly.

Problem 4: Using recursion to print an array in reverse order

10 points

Write a Java method called `printReverse()` that takes an array of type `Object` and uses recursion to print the contents of the array in reverse order. The array itself should *not* be reversed; it must be left in its original form.

Your method should have the following header:

```
public static void printReverse(Object[] arr, int i)
```

where `arr` is a reference to the array, and `i` is an integer parameter that you may use as you see fit. You do *not* need to code up this method as part of a class — simply include your implementation with your other answers for Part I.

For full credit, your `printReverse()` method should consider the first element of the array first — i.e., the first call to `printReverse()` should be the one that prints the first element, the second call should be the one that prints the second element, and so on. Half credit will be given for methods that consider the last element first. In either case, your method *must* be recursive; no credit will be given for methods that employ iteration.

Part II: Programming problems

70-75 points total

Problem 5: Adding methods to the `ArrayBag` class

30 points total; 5 points each part

Begin by downloading the following files:

- `Bag.java`
- `ArrayBag.java`

Use the File→Save As (or equivalent) option in your browser to put these files in the folder that you're using for your work on this assignment.

In `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.** These methods should be publicly accessible. **You should *not* add any new fields to the class.**

1. `public int capacity()`

This method should return the maximum number of items that the `ArrayBag` is able to hold. This value does not depend on the number of items that are currently in the `ArrayBag` — it is the same as the maximum size specified when the `ArrayBag` was created.

2. `public boolean isEmpty()`

   This method should return `true` if the `ArrayBag` is empty, and `false` otherwise.

3. `public int numOccur(Object item)`

   This method should return the number of times that the parameter `item` occurs in the called `ArrayBag`. For example, if `b1` represents the bag `{7, 5, 3, 5, 7, 2, 7}`, then `b1.numOccur(2)` should return `1`, `b1.numOccur(7)` should return `3`, and `b1.numOccur(20)` should return `0`.

4. `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 currently room for all of the items to be added, the items should be added and the method should return `true`. If there isn't enough room for all of the items to be added, *none* of them should be added and the method should return `false`. 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.

5. `public boolean equals(Bag other)`

   This method should determine if the called `ArrayBag` is equal to the parameter `other`. Two bags are equal if they contain the same items. The location of the items in the bags does *not* matter (e.g., `{5, 6, 6, 7}` is equal to `{7, 6, 5, 6}`), but bags that are equal must have the same number of occurrences of each item (e.g., `{5, 6, 6, 7}` is *not* equal to `{5, 6, 7}`). The method should return `true` if the two bags are equal, and `false` otherwise (including cases in which the parameter is `null`). 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. ***We strongly encourage you to have your `equals` method take advantage of one of the other methods that you are implementing for this problem.***

6. `public Bag unionWith(Bag other)`

   This method should create and return an `ArrayBag` containing *one occurrence* of any item that is found in either the called object or 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, 7, 8}` and `b2` represents the bag `{2, 3, 4, 5, 5, 6, 7}`, then `b1.unionWith(b2)` should return an `ArrayBag` representing the bag `{2, 3, 4, 5, 6, 7, 8}`. Give the new `ArrayBag` a maximum size that is the sum of the two bag's maximum sizes. If one of the bags is empty, the method should return an `ArrayBag` containing one occurrence of each item in the non-empty bag. If both of the bags are empty, the method should return an empty `ArrayBag`. 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. The return type is also `Bag`, but polymorphism allows you to just return the `ArrayBag` that you create, because `ArrayBag` implements `Bag`.

Problem 6: Recursion and strings

10-15 points total; 5 points each part

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.

1. `public static void printWithSpaces(String str)`

   This method should use recursion to print the individual characters in the string `str`, separated by spaces. For example, `printWithSpaces("space")` should print

   ```
   s p a c e
   ```

   where there is a single space after the `e`. The method should **not** return a value.

   Special cases: If the value `null` or the empty string (`""`) are passed in as the parameter, the method should just print a newline (by executing the statement `System.out.println();`) and return.

2. `public static String weave(String str1, String str2)`

   This method should use recursion to ***return*** the string that is formed by "weaving" together the characters in the strings `str1` and `str2` to create a single string. For example:

   - `weave("aaaa", "bbbb")` should return the string `"abababab"`
   - `weave("hello", "world")` should return the string `"hweolrllod"`

   If one of the strings is longer than the other, its "extra" characters — the ones with no counterparts in the shorter string — should appear immediately after the "woven" characters (if any) in the returned string. For example, `weave("recurse", "NOW")` should return the string `"rNeOcWurse"`, in which the extra characters from the first string — the characters in `"urse"` — come after the characters that have been woven together.

   This method should *not* do any printing; it should simply return the resulting string. If `null` is passed in for either parameter, the method should throw an `IllegalArgumentException`. If the empty string (`""`) is passed in for either string, the method should return the other string. For example, `weave("hello", "")` should return `"hello"` and `weave("", "")` should return `""`.

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

   `public static int indexOf(char ch, String str)`

   This method should use recursion to find and return the index of the first occurrence of the character `ch` in the string `str`, or -1 if `ch` does not occur in `str`. For example:

   - `indexOf('b', "Rabbit")` should return 2
   - `indexOf('P', "Rabbit")` should return -1

The method should return -1 if the empty string (`""`) or the value `null` is passed in as the second parameter. The `String` class comes with a built-in `indexOf()` method; you may *not* use that method in your solution!

Problem 7: A Sudoku solver

30 points

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.

Here is an example of an initial puzzle:

	4				3	7		
	8	9	7			1		
					4	2		
								1
6			5	1	8			9
2								
		5	3					
		6			1	9	4	
		1	2				6	

and here is its solution:

1	4	2	9	8	3	7	5	6
5	8	9	7	2	6	1	3	4
7	6	3	1	5	4	2	9	8
9	5	8	4	3	2	6	7	1
6	3	7	5	1	8	4	2	9
2	1	4	6	9	7	5	8	3
4	7	5	3	6	9	8	1	2
3	2	6	8	7	1	9	4	5
8	9	1	2	4	5	3	6	7

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`, which you should download and modify. The provided code includes:

- 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 allow 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 private method called `solveRB()` 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 `solveRB()` 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 17
 ...
```

- the full implementation of a public "wrapper" method called `solve()` that makes the initial call to `solveRB()`, and that returns the same value that `solveRB()` returns.
- 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 efficiently 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. You must avoid iterating over the sudoku board unnecessarily using these additional fields and methods.**

We are also providing:

- a separate `Sudoku` class in the file `Sudoku.java`. It 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.
- four sample puzzle files:
  - `puzzle1.txt`
  - `puzzle2.txt`
  - `no_solution.txt`, which has no solution
  - `multi_sol.txt`, which has multiple solutions

Additional hints and suggestions:

- Take advantage of the first template in the notes for recursive backtracking — the one in which only one solution is found.
- As mentioned above, the recursive `solveRB()` 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 `solveRB()` 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: puzzle1.txt

Here is the initial puzzle:
-----------------------------------
|   | 4 |   |   |   | 3 | 7 |   |   |
-----------------------------------
|   | 8 | 9 | 7 |   |   | 1 |   |   |
-----------------------------------
|   |   |   |   |   |   | 4 | 2 |   |   |
-----------------------------------
|   |   |   |   |   |   |   |   |   | 1 |
-----------------------------------
| 6 |   |   | 5 | 1 | 8 |   |   |   | 9 |
-----------------------------------
| 2 |   |   |   |   |   |   |   |   |   |
-----------------------------------
|   |   |   | 5 | 3 |   |   |   |   |   |
-----------------------------------
|   |   |   | 6 |   |   |   | 1 | 9 | 4 |   |
-----------------------------------
|   |   |   | 1 | 2 |   |   |   |   | 6 |   |
-----------------------------------

Here is the solution:

-----------------------------------
| 1 | 4 | 2 | 9 | 8 | 3 | 7 | 5 | 6 |
-----------------------------------
| 5 | 8 | 9 | 7 | 2 | 6 | 1 | 3 | 4 |
-----------------------------------
| 7 | 6 | 3 | 1 | 5 | 4 | 2 | 9 | 8 |
-----------------------------------
| 9 | 5 | 8 | 4 | 3 | 2 | 6 | 7 | 1 |
-----------------------------------
| 6 | 3 | 7 | 5 | 1 | 8 | 4 | 2 | 9 |
-----------------------------------
| 2 | 1 | 4 | 6 | 9 | 7 | 5 | 8 | 3 |
-----------------------------------
| 4 | 7 | 5 | 3 | 6 | 9 | 8 | 1 | 2 |
-----------------------------------
| 3 | 2 | 6 | 8 | 7 | 1 | 9 | 4 | 5 |
-----------------------------------
| 8 | 9 | 1 | 2 | 4 | 5 | 3 | 6 | 7 |
-----------------------------------
```

Submitting Your Work

You should use Canvas to submit the following files:

- **for part I:** your `ps1_partI.txt` or `ps1_partI.pdf` file.
- **for part II:**
  - your modified `Bag.java` file
  - your modified `ArrayBag.java` file
  - your `StringRecursion.java` file
  - your modified `Puzzle.java` file

Make sure to use these exact file names for your files. If you need to change the name of a Java file so that it corresponds to the name we have specified, make sure to also change the name of your class and check that it still compiles.

Here are the steps you should take to submit your work:

1. Go to the page for submitting assignments (logging in as needed)
2. Click on the appropriate link: either `ps1, part I` or `ps1, part II`.
3. Click on the *Submit Assignment* link near the upper-right corner of the screen. (If you have already submitted something for this assignment, click on *Re-submit Assignment* instead.)
4. Use the *Choose File* button to select a file to be submitted. If you have multiple files to submit, click *Add Another File* as needed, and repeat the process for each file.
5. Once you have chosen all of the files that you need to submit, click on the *Submit Assignment* button.
6. After submitting the assignment, you should check your submission carefully. In particular, you should:
   - Check to make sure that you have a green checkmark symbol labeled *Turned In!* in the upper-right corner of the submission page (where the *Submit Assignment* link used to be), along with the names of all of the files from the part of the assignment that you are submitting.
   - **Click on the link for each file to download it, and view the downloaded file so that you can ensure that you submitted the correct file.**

Warning

We will not accept any files after the fact, so please check your submission carefully following the instructions in Step 6.

Important

- You must submit all of the files for a given part of the assignment (Part I or Part II) at the same time. If you need to resubmit a file for some reason, you should also resubmit any other files from that part of the assignment.

- If you re-submit a file in Canvas, it will append a version number to the file name. You do *not* need to worry about this. Our grading scripts will remove the version number before we attempt to grade your work.
- There is a *Comments* box that accompanies each submission, but we do **not** read anything that you write in that space. If you need to inform us of something about your submission, please email `cscie22@fas.harvard.edu`.
- 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 `cscie22@fas.harvard.edu`.