Problem Set 2

Due before the start of lecture on October 10, 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

55 points total

Submit your answers to part I in a plain-text file called `ps2_partI.txt`, and put your name and email address at the top of the file.

Important

When big-*O* expressions are called for, please use them to specify tight bounds, as explained in the lecture notes.

Problem 1: Sorting practice

14 points; 2 points for each part

Given the following array:

```
{10, 18, 4, 24, 33, 40, 8, 3, 12}
```

1. If the array were sorted using selection sort, what would the array look like after the *second* pass of the algorithm (i.e., after the second time that the algorithm performs a pass or partial pass through the elements of the array)?
2. If the array were sorted using insertion sort, what would the array look like after the *fourth* iteration of the outer loop of the algorithm?
3. If the array were sorted using Shell sort, what would the array look like after the initial phase of the algorithm, if you assume that it uses an increment of 3? (The method presented in lecture would start with an increment of 7, but you should assume that it uses an increment of 3 instead.)
4. If the array were sorted using bubble sort, what would the array look like after the *third* pass of the algorithm?
5. If the array were sorted using the version of quicksort presented in lecture, what would the array look like after the initial partitioning phase?
6. If the array were sorted using radix sort, what would the array look like after the initial pass of the algorithm?
7. If the array were sorted using the version of mergesort presented in lecture, what would the array look like after the completion of the *fourth* call to the `merge()` method—the method that merges two subarrays? Note: the `merge` method is the helper method; is *not* the recursive `mSort` method.

Important

There will be no partial credit on the above questions, so please *check your answers carefully*!

Problem 2: Counting comparisons

6 points total; 2 points each part

Given an *already sorted* array of 5 elements, how many comparisons of array elements would each of the following algorithms perform?

1. insertion sort
2. bubble sort
3. quicksort

Explain each answer briefly.

Problem 3: Comparing two algorithms

8 points

Suppose that you want to count the number of duplicates in an unsorted array of *n* elements. A duplicate is an element that appears multiple times; if a given element appears *x* times, *x* - 1 of them are considered duplicates. For example, consider the following array:

```
{10, 6, 2, 5, 6, 6, 8, 10, 5}
```

It includes four duplicates: one extra 10, two extra 6s, and one extra 5.

Below are two algorithms for counting duplicates in an array of integers:

Algorithm A:

```
public static int numDuplicatesA(int[] arr) {
    int numDups = 0;
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] == arr[i]) {
                numDups++;
                break;
            }
        }
    }
    return numDups;
}
```

Algorithm B:

```
public static int numDuplicatesB(int[] arr) {
    Sort.mergesort(arr);
    int numDups = 0;
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] == arr[i - 1]) {
            numDups++;
        }
    }
    return numDups;
}
```

What is the worst-case time efficiency of algorithm A in terms of the length *n* of the array? What is the worst-case time efficiency of algorithm B? Make use of big-O notation, and explain briefly how you came up with the big-O expressions that you use.

Problem 4: Sum generator

12 points total; 3 points each part

Let's say that you want to implement a method `generateSums(n)` that takes an integer `n` and generates and prints the following series of sums:

```
1
1 + 2
1 + 2 + 3
...
1 + 2 + ... + n.
```

For example, `generateSums(4)` should print the following:

```
1
3
6
10
```

One possible implementation of this method is:

```
public static void generateSums(int n) {
    for (int i = 1; i <= n; i++) {
        int sum = 0;
        for (int j = 1; j <= i; j++) {
            sum = sum + j; // how many times is this executed?
        }
        System.out.println(sum);
    }
}
```

1. Derive an exact formula for the number of times that the line that increases the sum is executed, as a function of the parameter `n`.
2. What is the time efficiency of the method shown above as a function of the parameter `n`? Use big-O notation, and explain your answer briefly.
3. Create an alternative, non-recursive implementation of this method that has a better time efficiency.
4. What is the time efficiency of your alternative implementation as a function of the parameter `n`? Use big-O notation, and explain your answer briefly.

Problem 5: Stable and unstable sorting

5 points

Sorting algorithms can be applied to *records* of data, where each record consists of multiple fields. In such contexts, the sorting algorithm orders the records according to one of the fields, and the value of that field is referred to as the *key* of the record.

A sorting algorithm is *stable* if it preserves the order of elements with equal keys. For example, given the following array:

```
{32, 12a, 4, 12b, 39, 19}
```

where `12a` and `12b` represent records that both have a key of `12`, a stable sorting algorithm would produce the following sorted array:

```
{4, 12a, 12b, 19, 32, 39}
```

Note that `12a` comes before `12b`, just as it did in the original, unsorted array. Insertion sort is an example of a stable sorting algorithm.

Stability can be useful if you want to sort on the basis of two different fields—for example, if you want records sorted by last name and then, within a given last name, by first name. You could accomplish this in two steps: (1) use any sorting algorithm to sort the records by first name, and (2) use a stable sorting algorithm to sort the records by last name. Because the second algorithm is stable, it would retain the order of records with the same last name, and thus those records would remain sorted by first name.

By contrast, an *unstable* sorting algorithm may end up reversing the order of elements with equal keys. For example, given the same starting array shown above, an unstable sorting algorithm could produce either of the following arrays:

```
{4, 12a, 12b, 19, 32, 39}

{4, 12b, 12a, 19, 32, 39}
```

Selection sort is an example of an unstable sorting algorithm. Construct an example of an input array containing two elements with equal keys whose order is reversed by selection sort. Show the effect of the algorithm, step by step, on this array, labeling the elements with equal keys as we did in our example in order to keep them straight.

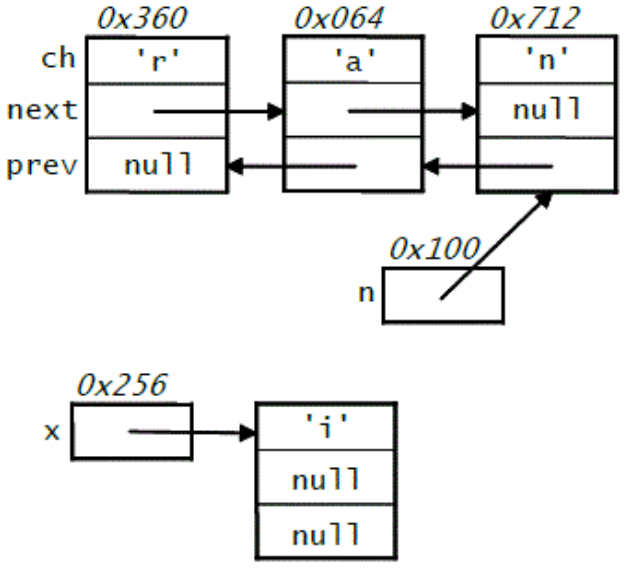
Problem 6: Practice with references

10 points total

Note: We will cover the material needed for this problem in the lecture on October 3, so you may want to wait until after that lecture to complete it.

As discussed in lecture, a *doubly linked list* consists of nodes that include two references: one called `next` to the next node in the linked list, and one called `prev` to the previous node in the linked list. The first node in such a list has a `prev` field whose value is `null`, and the last node has a `next` field whose value is `null`.

The top portion of the diagram below shows a doubly linked list of characters that could be used to represent the string `"ran"`.



Each of the nodes shown is an instance of the following class:

```
public class DNode {
    private char ch;
    private DNode next;
    private DNode prev;
}
```

(In the diagram, we have labeled the individual fields of the `DNode` object that contains the character `'r'`.)

In addition to the list representing `"ran"`, the diagram shows an extra node containing the character `'i'`, and two reference variables: `n`, which holds a reference to the third node in the list (the `'n'` node); and `x`, which holds a reference to the `'i'` node. The diagram also shows memory addresses of the start of the variables and objects. For example, the `'r'` node begins at address `0x360`.

1. (6 points) Complete the table below, filling in the address and value of each expression from the left-hand column. You should assume the following: the address of the `ch` field of a `DNode` is the same as the address of the `DNode` itself, the address of the `next` field of a `DNode` is 2 more than the address of the `DNode` itself, and the address of the `prev` field of a `DNode` is 6 more than the address of the `DNode` itself.

Expression	Address	Value
n		
n.ch		
n.prev		
n.prev.prev		
n.prev.next.next		
n.prev.prev.next		

2. (4 points) Write a Java code fragment that inserts the `'i'` node between the `'a'` node and the `'n'` node, producing a linked list that represents the string `"rain"`. Your code fragment should consist of a series of assignment statements. You should not make any method calls, and you should not use any variables other than the ones provided in the diagram. Make sure that the resulting doubly linked list has correct values for the `next` and `prev` fields in all nodes.

Part II

45-55 points total

Problem 7: Searching an array for pairs that sum to k

15-25 points total

Suppose you are given an array of `n` integers, and you need to find all pairs of values in the array (if any) that sum to a given integer `k`. In a class named `Problem7`, write code that performs this task for you and outputs all of the pairs that it finds. For example, if `k` is 12 and the array is `{10, 4, 7, 7, 8, 5, 15}`, your code should output something like the following:

```
4 + 8 = 12
7 + 5 = 12
7 + 5 = 12
```

Note that we get two `7 + 5` sums because `7` appears twice in the array. However, while the method or methods that you write *may* print a given pair of values more than once in such cases, it is *not* necessary to do so. In addition, the order in which the sums (and the terms within each sum) are printed does *not* matter.

1. (15 points) Implement a static method named `pairSums()` that requires $O(n^2)$ steps to solve this problem. The method should have the following header:

   ```
   public static void pairSums(int k, int[] arr)
   ```

In the comments that accompany the method, include a brief argument showing that your algorithm is O(n^2). In addition, you should add test code for it to a `main` method.

2. (10 points; *required of grad-credit students; "partial" extra credit for others*) Implement a static method named `pairSumsImproved()` that takes the same parameters as `pairSums`, but that requires only *O*(nlogn) steps in the average case to solve this problem. (*Hint*: you should begin by sorting the array using one of the methods from our `Sort` class. Once you have done so, only *O*(n) additional steps are needed to find the pairs.) Here again, you should include a brief argument showing that your algorithm is *O*(nlogn), and add test code to the `main` method.

Problem 8: A merge-like approach to finding the intersection of two arrays

15 points

In a class named `Problem8`, implement a static method named `findIntersect()` that takes two arrays of integers as parameters and uses an approach based on merging to find and return the intersection of the two arrays.

More specifically, you should begin by creating a new array for the intersection, giving it the length of the smaller of the two arrays. Next, you should use one of the more efficient sorting algorithms from `Sort.java` to sort both of the arrays. Finally, you should find the intersection of the two arrays by employing an approach that is similar to the one that we used to merge two sorted subarrays (i.e., the approach taken by the `merge` method in `Sort.java`). Your method should **not** actually merge the two arrays, but it should take a similar approach—using indices to "walk down" the two arrays, and making use of the fact that the arrays are sorted. As the elements of the intersection are found, put them in the array that you created at the start of the method. At the end of the method, return a reference to the array containing the intersection.

For full credit, the intersection that you create should not have any duplicates, and ***your algorithm should be as efficient as possible.*** In particular, you should perform at most one complete pass through each of the arrays. Add test code for your method to the main method. You can use the `Arrays.toString()` method to convert an array to a string; import the `java.util` package to gain access to the `Arrays` class.

Here are some example calls from the Interactions Pane in DrJava:

```
> int[] a1 = {10, 5, 7, 5, 9, 4};
> int[] a2 = {7, 5, 15, 7, 7, 9, 10};
> int[] result = Problem8.findIntersect(a1, a2);
> result
{ 5, 7, 9, 10, 0, 0 }
> int[] a3 = {0, 2, -4, 6, 10, 8};
> int[] a4 = {12, 0, -4, 8};
> int[] result = Problem8.findIntersect(a3, a4);
> result
{-4, 0, 8, 0}
```

Notes:

- In both tests, the array containing the intersection has the same length as the shorter of the original arrays.
- When the number of values in the intersection (call it *n*) is smaller than the length of the shorter array, we end up with extra 0s at the end of the array containing the intersection. This happens because the array of integers that we create is initially filled with 0s, and we only use the first *n* positions of the array.
- If 0 is one of the values in the intersection, it will also show up earlier in the results, as it does in our second test above.

Problem 9: Improving bubble sort

15 points total

In the version of bubble sort presented in lecture, the method always performs `n - 1` passes when sorting an array of `n` elements, regardless of the contents of the array. We can improve bubble sort by having it stop once the array is fully sorted. This will require adding code that allows the algorithm to determine when no further passes are needed.

1. (7 points) Implement an improved version of bubble sort, adding it to the file `SortCount.java`. Your method should be as efficient as possible. In particular, it should not perform any unnecessary passes over the elements of the array, and you should not consider pairs of elements unnecessarily. In fact, your modified method should be able to detect that the array is sorted without performing any additional comparisons beyond those that are already performed in the course of a given pass of the algorithm.

   Call the new method `bubbleSort2()`. Its only parameter should be a reference to an array of integers. Like the other methods in this file, your `bubbleSort2()` method must make use of the `compare()`, `move()`, and `swap()` helper methods so that you can keep track of the total number of comparisons and moves that it performs. In particular:

   - If you need to compare two array elements, you should use the `compare` method. For example, instead of writing

     ```
     if (arr[i] < arr[j])
     ```

     you should instead write

     ```
     if (compare(arr[i] < arr[j]))
     ```

     This method will return the result of the comparison that is passed in (either `true` or `false`), and it will increment the count of comparisons that the class maintains.

   - If you need to move an element of the array, you should use the `move` method. For example, instead of writing

     ```
     arr[i] = arr[j];
     ```

     you should instead write

     ```
     move(arr, i, j);
     ```

     This method will move element `j` of `arr` into position `i`, and it will increment the count of moves that the class maintains.

2. (8 points) Determine the big-O time efficiency of `bubbleSort2()` when it is applied to two types of arrays: arrays that are fully sorted, and arrays that are randomly ordered. You should determine the big-O expressions *by experiment*, rather than by analytical argument.

   To do so, run the algorithm on arrays of different sizes (for example, n = 1000, 2000, 4000, 8000 and 16000). Modify the test code in the main() method so that it runs `bubbleSort2()` on the arrays that are generated, and use this test code to gather the data needed to make your comparisons. (Note: you can also test the correctness of your method by running it on arrays of 10 or fewer items; the sorted array will be printed in such cases.)

   For each type of array, you should perform at least ten runs for each value of n and compute the average numbers of comparisons and moves for each set of runs. Based on these results, determine the big-O efficiency class to which `bubbleSort2()` belongs for each type of array ($O(n)$, $O(\log n)$, $O(n \log n)$, $O(n^2)$, etc.). Explain clearly how you arrived at your conclusions. If the algorithm does not appear to fall neatly into one of the standard efficiency classes, explain your reasons for that conclusion, and state the two efficiency classes that it appears to fall between. See the section

notes for more information about how to analyze the results. ***Put the results of your experiments, and your accompanying analysis and conclusions, in a plain-text file called*** `ps2_experiments.txt`.

Submitting Your Work

You should use Canvas to submit the following files:

- **for part I:** your `ps2_partI.txt` file. Make sure that this file is a plain-text file.
- **for part II:**
  - your `Problem7.java` file
  - your `Problem8.java` file
  - your modified `SortCount.java` file
  - your `ps2_experiments.txt` file, which should be a plain-text 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.

1. Go to the page for submitting assignments (logging in as needed)
2. Click on the appropriate link: either `ps2, part I` or `ps2, 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`.