Problem Set 2

Due before lecture on Wednesday, October 7

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 `cscie22@fas.harvard.edu`.

Part I: Short-Answer ("Written") Problems (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.

1. **Sorting practice** (14 points; 2 points for each part)
   Given the following array:

   ```
   24  3  27  13  34  2  50  12
   ```

   a. If the array were sorted using selection sort, what would the array look like after the *third* pass of the algorithm (i.e., after the third time that the algorithm performs a pass or partial pass through the elements of the array)?
   b. If the array were sorted using insertion sort, on how many iterations of the outer loop would the inner `do...while` loop be skipped?
   c. 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.)
   d. If the array were sorted using bubble sort, what would the array look like after the *fourth* pass of the algorithm?
   e. If the array were sorted using the version of quicksort presented in lecture, what would the array look like after the initial partitioning phase?
   f. If the array were sorted using radix sort, what would the array look like after the initial pass of the algorithm?
   g. 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.

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

2. **Comparing two algorithms** (5 points)
   Suppose you want to find the largest element in an unsorted array of `n` elements. Algorithm A searches the entire array sequentially and keeps track of the largest element seen thus far. Algorithm B sorts the array using one of the most efficient comparison-based sorting algorithms, and then reports the last element as the largest. *Compare* the time efficiency of these algorithms, making use of big-O notation. You should *not* implement the methods. Explain your answers briefly.

3. **Counting comparisons** (6 points total; 2 points each part)
   Given an *already sorted* array of 6 elements, how many comparisons of array elements would each of the following algorithms perform?
   a. selection sort
   b. insertion sort
   c. mergesort

   Explain each answer briefly.

4. **Swap sort** (10 points total; 5 points each part)
   Swap sort is another sorting algorithm based on exchanges. Like bubble sort, swap sort compares pairs of elements and swaps them when they are out of order, but swap sort looks at different pairs of elements than bubble sort does. On pass i of swap sort, the element in position i of the array is compared with the elements in all positions to the right of position i, and pairs of elements are swapped as needed. At the end of pass i, the element in position i is where it belongs in the sorted array.

   For example, let's say that we have the following initial array:

   ```
   15   8   20   5   12
   ```

   On pass 0, swap sort compares the element in position 0 with the elements in positions 1 through n – 1, where n is the number of elements in the array. First, 15 is compared to 8, and because 8 is less than 15, the two elements are swapped:

   ```
   8   15   20   5   12
   ```

   Next, 8 (which is now in position 0) is compared to 20, and no swap occurs because 8 is already smaller than 20. Then 8 is compared with 5, and the two elements are swapped:

   ```
   5   15   20   8   12
   ```

   Finally, 5 is compared to 12, and no swap occurs because 5 is already smaller than 12. At this point, pass 0 is complete, and the element in position 0 (the 5) is where it belongs in the sorted array.

   On pass 1, swap sort compares the element in position 1 with the elements in positions 2 through (n – 1) and performs swaps when needed. The process continues for passes 2 through (n – 2), at which point the array is fully sorted.

a. What is the best case for this algorithm? In this best case, what is the big-O expression for the number of comparisons that it performs? For the number of moves? What is its overall time efficiency? Explain your answers briefly.

b. What is the worst case for this algorithm? In this worst case, what is the big-O expression for the number of comparisons? For the number of moves? What is its overall time efficiency? Explain your answers briefly.

5. **Mode finder** (10-20 points total; 5 points each part)
Let's say that you want to implement a method `findMode(arr)` that takes an array of integers and returns the *mode* of the values in the array – i.e., the value that appears most often in the array. For example, the mode of the array {10, 8, 12, 8, 10, 5, 8} is 8.

If two or more values are tied for the mode, the method should return the smallest of the most frequently occurring values. For example, in the array {7, 5, 3, 5, 7, 11, 11}, three values (5, 7, and 11) each appear twice, and no value appears more than twice; the method should return 5 because it is the smallest of the three values that appear twice.

One possible implementation of this method is:

```
public static int findMode(int[] arr) {
    int mode = -1;
    int modeFreq = 0;    // number of times that the mode appears

    for (int i = 0; i < arr.length; i++) {
        int freq = 1;    // number of times that arr[i] appears from
                         // position i to the end of the array
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] == arr[i])  // how many times is this executed?
                freq++;
        }
        if (freq > modeFreq || (freq == modeFreq && arr[i] < mode)) {
            mode = arr[i];
            modeFreq = freq;
        }
    }

    return mode;
}
```

a. Derive an exact formula for the number of times that the line comparing `arr[j]` to `arr[i]` is executed, as a function of the length of the array.

b. What is the time efficiency of the method shown above as a function of the length of the array? Use big-O notation, and explain your answer briefly.

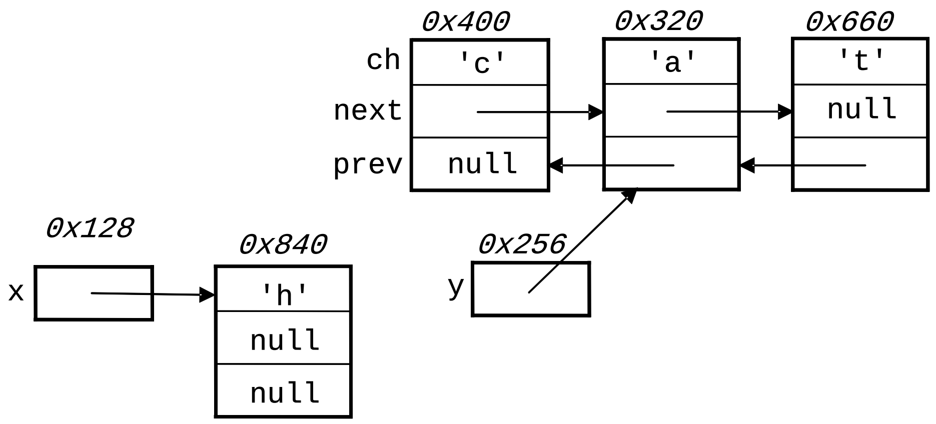
Parts c and d on the next page are required for grad-credit students, and will be worth "partial" extra credit for others.

c. Create an alternative implementation of this method that has a better time efficiency than the method above. *Hint:* Begin by sorting the array, which you can do by calling an algorithm from our `Sort` class. Make your implementation as efficient as possible. Put your code in the same document as your solutions to the other written problems.

d. What is the time efficiency of your alternative implementation as a function of the length of the array? Use big-O notation, and explain briefly.

6. **Practice with references** (10 points total)
Note: We will cover the material needed for this problem in the fifth lecture, 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 "cat".



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 'c'.)

In addition to the list representing "cat", the diagram shows an extra node containing the character 'h', and two reference variables: `y`, which holds a reference to the second node in the list (the 'a' node); and `x`, which holds a reference to the 'h' node. The diagram also shows memory addresses of the start of the variables and objects. For example, the 'c' node begins at address 0x400.

a. (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**
`x`		
`x.ch`		
`y.prev`		
`y.next.prev`		
`y.prev.next`		
`y.prev.next.next`		

b. (4 points) Write a Java code fragment that inserts the 'h' node between the 'c' node and the 'a' node, producing a linked list that represents the string "chat". 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: Programming Problems (45 points total)

1. **Removing duplicates** (15 points)
Suppose you are given an already sorted array that may contain duplicate items – i.e., items that appear more than once. Add a method to the `Sort` class called `removeDups()` that takes a sorted array of integers and removes whatever elements are necessary to ensure that no item appears more than once. Make your method as efficient as possible in the number of operations that it performs. In addition, your method should use O(1) additional memory – i.e., it should *not* create and use a second array.

The remaining elements should still be sorted, and they should occupy the leftmost positions of the array. The array locations that are unused after the duplicates are removed should be filled with zeroes. For example, if `arr` is the array `{2, 5, 5, 5, 10, 12, 12}`, after the call `removeDups(arr)`, the array should be `{2, 5, 10, 12, 0, 0, 0}`.

The method should return an `int` that specifies the number of unique values in the array. For example, when called on the original array above, the method should return a value of 4. Add code to the `main` method to test your new method.

Important note: One inefficient approach would be to scan from left to right, and, whenever you encounter a duplicate, to shift all of the remaining elements left by one. The problem with this approach is that elements can end up being shifted multiple times, and thus the algorithm has a worst-case running time that is $O(n^2)$. Your method should move each element *at most once*. This will give it a running time that is $O(n)$. Only half credit will be given for methods that move elements more than once.

2. **Finding the median** (15 points)
The median of an array of integers is the value that would belong in the middle position of the array if the array were sorted. For example, consider the following array:

```
int[] arr = {4, 18, 12, 34, 7, 42, 15, 22, 5};
```

The median of this array is 15, because that value would end up in the middle position if the array were sorted:

```
int[] arr = {4, 5, 7, 12, 15, 18, 22, 34, 42};
```

If an array has an even number of elements, the median is the average of the elements that would belong in the middle two positions of the array if the array were sorted.

In the file `Median.java`, implement a program that finds the median of an array of integers by using the partitioning approach taken by quicksort. *However, for the sake of efficiency, you should not sort a subarray if it could not contain the median.*

In `Median.java`, we have given you the following:

- a copy of the `partition()` method from our `Sort` class. The recursive method that you will write should call this method to process the necessary portions of the array (see below). You should not need to change it.
- the skeleton of a method named `findMedian()` that will serve as a "wrapper" method around your recursive median-finding method, just as the `quickSort()` method in our `Sort` class serves as a wrapper around the recursive `qSort()` method. To implement this wrapper method, all you need to do is add an appropriate initial call to your recursive method. You should *not* change the header of this method in any way.
- a `main()` method that you should fill with code that tests your median-finding algorithm. We have included sample definitions of odd- and even-length arrays that you can use as you see fit. Make sure that your `main()` method calls the wrapper method, rather than calling the recursive method directly.

The recursive method that you write will be similar to the recursive `qSort()` method in our `Sort` class. However, you will need to modify the algorithm so that it allows you to put the median value or values in the appropriate positions *without* actually sorting the entire array. You are expected to only sort as much of the array as is necessary to determine the median value, and for full credit you *must* avoid calling `partition()` on subarrays that could not possibly contain the median value (see below).

Your recursive method only needs to succeed in getting the median value (or values, in the case of an even array) into the middle position (or positions, in the case of an even array). It does *not* need to return the median value; it may simply be a `void` method. Your `main()` method can then print the median value by looking at the value(s) in the middle position(s) of the array after your algorithm has finished.

Hints:

- Your recursive method will be similar to our `qSort()` method. However, after you call `partition()`, you need to determine whether to make recursive call(s) to process just the left subarray, just the right subarray, or both subarrays, depending on whether each subarray could contain the median value (or, in the case of an even-length array, at least one of the two median values). When making this decision, remember that after a call to `partition()`, everything in the left subarray is less than or equal to everything in the right subarray. Given this fact, and given the location(s) where the median value(s) must ultimately end up, you should be able to determine whether you need to make a recursive call on a particular subarray. Tracing through some concrete examples may help you to discover the logic.

  Note: Rather than checking to see whether a given recursive call is needed, it's also possible to defer this checking to the start of the next invocation of the method. In other words, the recursive method could *begin* by checking to see if the current subarray could contain one or more of the median values, and, if it could not, the method could return without doing anything.

- *You should be able to accomplish this task with only 4-10 lines worth of changes/additions to the standard quicksort algorithm.* If your modifications to the code in `qSort()` are substantially longer than this, you are likely on the wrong track.

3. **Implementation and experimental analysis of swap sort** (15 points total)
   a. (7 points) In the file `SortCount.java`, add a method called `swapSort()` that implements the swap sort algorithm described in written problem 4. Its only parameter should be a reference to an array of integers. Like the other methods in this file, your `swapSort` 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. If you need to compare

two array elements, you should make the method call `compare(`*comparison*`)`; for example, `compare(arr[0] < arr[1])`. This method will return the result of the comparison (`true` or `false`), and it will increment the count of comparisons that the class maintains. If you need to move element `j` of `arr` to position `i`, instead of writing `arr[i] = arr[j]`, you should write `move(arr, i, j)`.

b. (8 points) Determine the big-O time efficiency of `swapSort` when it is applied to two types of arrays: arrays that are almost 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 `swapSort` 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 `swapSort` belongs for each type of array (O(n), O(logn), O(nlogn), 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:

- your `ps2_partI.txt` file.
- your modified `Sort.java` file
- your modified `Median.java` file
- your modified `SortCount.java` file
- your `ps2_experiments.txt` 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:

- Go to the page for submitting assignments (logging in as needed using the Login link in the upper-right corner, and entering your Harvard ID and PIN).
- Click on the link for **Problem Set 2**.
- 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.)
- Use the *Choose File* button to select a file to be submitted. Continue selecting files by clicking *Add Another File*, and repeat the process for each file. ***Important:*** You must submit all of the files for a given assignment at the same time. If you need to resubmit a file for some reason, you should also resubmit the other files from that assignment.
- Once you have chosen all of the files that you need to submit, click on the *Submit Assignment* button.
- 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 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.

  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 `cscie22@fas.harvard.edu`