Problem Set 2

Due before lecture on Wednesday, October 6

Getting Started

To get the files that you will need for this problem set, log into `nice.harvard.edu` and enter the following command:

```
gethw 2
```

This will create a directory named `ps2` in your Harvard home directory containing the files that you need for this assignment. To change into this directory, the command is:

```
cd ~/ps2
```

The final versions of your files for Problem Set 2 should ultimately be put in this directory and submitted from there, following the directions at the end of the assignment.

Submitting Your Work

Once you are ready to submit, make sure that all of your files are in your `ps2` directory on `nice.harvard.edu`.

To check that your code compiles, enter the following command from within the `ps2` directory: `make`

To submit your assignment, enter the following command: `make submit`

To check which files were submitted, enter the following command: `make check`

Important note: the submit commands only work if you are logged onto *`nice`*`.harvard.edu`.

I. Written Problems (55 points total)

Once again, you may submit your solutions to these problems either in written form or electronically. See Problem Set 1 for details.

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:

   ```
   10  18  4  24  33  40  8  3  12
   ```

   a. 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)?
   b. 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?
   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 *third* 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. **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?
   a. insertion sort
   b. bubble sort
   c. quicksort
   Explain each answer briefly.

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, the array {10, 6, 2, 5, 6, 6, 8, 10, 5} 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.

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);
    }
}
```

a. 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.
b. 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.
c. Create an alternative, non-recursive implementation of this method that has a better time efficiency.
d. 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.

5. **Stable and unstable sorting** (5 points)
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  38  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 keys – 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 sorted arrays:

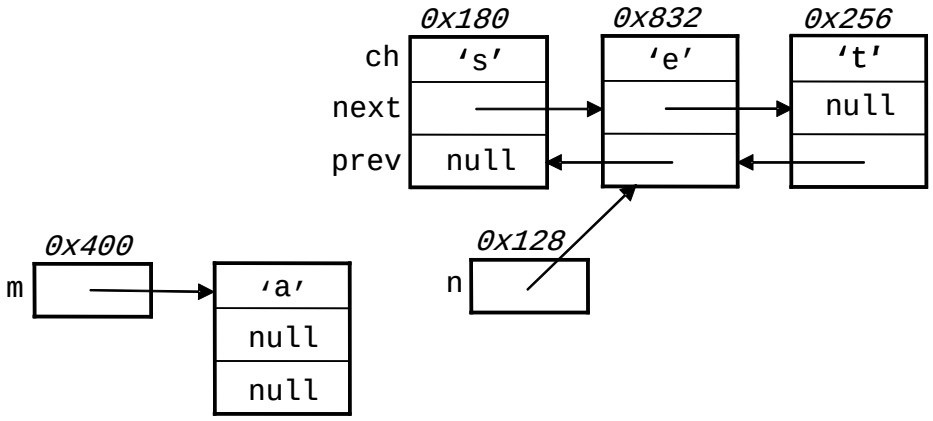
```
4  12a  12b  19  32  39
4  12b  12a  19  32  39
```

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.

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 "set".



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 's'.)

In addition to the list representing "set", the diagram shows an extra node containing the character 'a', and two reference variables: `n`, which holds a reference to the second node in the list (the 'e' node); and `m`, which holds a reference to the 'a' node. The diagram also shows memory addresses of the start of the variables and objects. For example, the 's' node begins at address 0x180.

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**
`n`		
`n.ch`		
`n.prev`		
`n.prev.next`		
`n.next.prev`		
`n.next.prev.prev`		

b) (4 points) Write a Java code fragment that inserts the 'a' node between the 'e' node and the 't' node, producing a linked list that represents the string "seat". 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.

II. Programming Problems (45-55 points total)

1. **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 `PairSums` (implemented in a file named PairSums.java), 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.

a. (15 points) Implement a static method named `findPairs()` that requires $O(n^2)$ steps to solve this problem. The method should take two parameters: the value `k` and a reference to the array. 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 the `main` method. You may find it helpful to call the `randomArray` method from our `SortCount` class to generate test arrays.

b. (10 pts.; *required of grad-credit students; "partial" extra credit for others*) Implement a static method named `findPairsFaster()` that requires $O(n \log n)$ 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 $O(n \log n)$, and add test code to the `main` method.

2. **A merge-like approach to finding the intersection of two arrays** (15 pts)
   In a file named `Intersect.java`, implement a static method named `intersect` 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 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 may find it helpful to call the `randomArray` method from our `SortCount` class to generate test arrays.

3. **Combining bubble and Shell** (15 points total)
   One of the problems with bubble sort is that it only swaps adjacent elements. In the same way that Shell sort improves on insertion sort by allowing for elements to make larger jumps, we can improve bubble sort by modifying it to operate on pairs of elements that are separated by some increment. Like Shell sort, the algorithm would begin a large increment, and successive stages of the algorithm would use smaller and smaller increments, with a final increment of 1.

   In order for this approach to produce efficiency gains, we also need to modify bubble sort so that it keeps track of how many swaps that it performs during a given pass through the array. If the algorithm performs no swaps for an entire pass, or if there are no more passes to perform at that increment, the algorithm should move on to the next smaller increment. If the algorithm performs no swaps for an entire pass when the increment is 1, the algorithm terminates.
   For example, consider the following array:

```
20   7  14  18   2  30   6  23  11   5
```

Assume that we begin with an increment of 3, which effectively divides the array into three subarrays: {20, 18, 6, 5}, {7, 2, 23}, {14, 30, 11}. At the start of its first pass at that increment, the algorithm compares 20 and 18 and swaps them because they are out of order with respect to each other:

```
18   7  14  20   2  30   6  23  11   5
```

It next compares 7 and 2 and swaps them:

```
18   2  14  20   7  30   6  23  11   5
```

Then it compares 14 and 30 and does *not* swap them, because they are already in order with respect to each other. It continues comparing and swapping as needed until it reaches the end of the array:

```
18   2  14   6   7  30  20  23  11   5
18   2  14   6   7  30  20  23  11   5    // no swap needed
18   2  14   6   7  11  20  23  30   5
18   2  14   6   7  11   5  23  30  20
```

At this point, we know that the largest element of each subarray has bubbled to the end of its subarray. Thus, the next pass does not need to reconsider those elements – the final 3 elements of the array – on the next pass at this increment. This is similar to the way that regular bubble sort does not consider the last element on its second pass.

Here is a summary of the second pass with an increment of 3, with italics showing the elements that should not be reconsidered at this stage:

```
6   2  14  18   7  11   5  23  30  20
6   2  14  18   7  11   5  23  30  20    // no swap needed
6   2  11  18   7  14   5  23  30  20
6   2  11   5   7  14  18  23  30  20
```

At this point, we know that the largest *two* elements of each subarray have bubbled to the end of their respective subarrays. Thus, the next pass does not need to reconsider those elements – the final 6 elements of the array – on the next pass at this increment. As a result, there is only one comparison/swap to perform on the third pass:

```
5   2  11   6   7  14  18  23  30  20
```

At this point, because the final nine elements are in the appropriate places in their subarrays, there is nothing left to do at an increment of 3, and the algorithm moves to an increment of 1, which is equivalent to regular bubble sort. However, it stops as soon as an entire pass performs no swaps.

Here's the first pass at increment 1:

```
2   5   11   6   7   14   18   23   30   20
2   5   11   6   7   14   18   23   30   20   // no swap needed
2   5    6  11   7   14   18   23   30   20
2   5    6   7  11   14   18   23   30   20
2   5    6   7  11   14   18   23   30   20   // no swap needed
2   5    6   7  11   14   18   23   30   20   // no swap needed
2   5    6   7  11   14   18   23   30   20   // no swap needed
2   5    6   7  11   14   18   23   30   20   // no swap needed
2   5    6   7  11   14   18   23   20   30
```

The second pass at that increment compares the same pairs of elements – except the final pair – but only performs one swap:

```
2   5   6   7  11   14   18   20   23   30
```

The third pass at that increment compares the same pairs of elements – except the final two pairs – but performs *no* swaps. As a result, the algorithm terminates.

a. (7 points) Implement this combination of Shell sort and 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.

   Call the new method `shellBubbleSort`. Its only parameter should be a reference to an array of integers. Like the other methods in this file, your `shellBubbleSort` 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 `shellBubbleSort` 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 `shellBubbleSort` 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 `shellBubbleSort` 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 the same document as your written problems.***