Problem Set 7

part I due by 10:00 p.m. on Friday, July 25
part II due by 10:00 p.m. on Monday, July 28

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).

Part I: Short-Answer Problems (55-65 points total)
All of Part I is pair-optional. See the syllabus for a reminder of the rules for working with a partner.

Put your answers to this part of the assignment in a plain-text file called ps7_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:
   
   \begin{align*}
   14 & 7 & 27 & 13 & 24 & 20 & 10 & 33 \\
   \end{align*}

   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, 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 second 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 second call to the partition method?

   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 third call to the `merge()` method – the method that merges two subarrays? Note: the `merge` method is the helper method; it is not the recursive `mSort` method.

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

2. **Comparing two algorithms** (10 points)

The Fibonacci sequence begins as follows:

1, 1, 2, 3, 5, 8, 13, 21, ...

The first two elements in the sequence are both 1, and all other elements are the sum of the previous two elements. For example, 2 = 1 + 1, 3 = 1 + 2, and 5 = 2 + 3.

Below are two algorithms for printing the first n numbers in this sequence:

**Algorithm A:**

```java
public static void printFibA(int n) {
    for (int i = 0; i < n; i++) {
        int prev = 0;
        int curr = 1;
        for (int j = 0; j < i; j++) {
            int next = prev + curr;
            prev = curr;
            curr = next;
        }
        System.out.print(curr + " ");
    }
}
```

**Algorithm B:**

```java
public static void printFibB(int n) {
    int prev = 0;
    int curr = 1;
    for (int i = 0; i < n; i++) {
        System.out.print(curr + " ");
        int next = prev + curr;
        prev = curr;
        curr = next;
    }
}
```

What is the time efficiency of algorithm A as a function of the parameter n? What is the 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.
3. 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. selection sort
   b. Shell sort
   c. mergesort (Reminder: When an array has an odd length, the left subarray ends up with one more element than the right subarray.)

Explain each answer briefly.

4. Sum generator (10-20 points total; 5 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:

```java
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.

*Parts c and d are required for grad-credit students, and will be worth "partial" extra credit for others.*
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:

\[
\begin{array}{cccccc}
32 & 12a & 4 & 12b & 38 & 19
\end{array}
\]

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

\[
\begin{array}{cccccc}
4 & 12a & 12b & 19 & 32 & 39
\end{array}
\]

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:

\[
\begin{array}{cccccc}
4 & 12a & 12b & 19 & 32 & 39
\end{array}
\quad
\begin{array}{cccccc}
4 & 12b & 12a & 19 & 32 & 39
\end{array}
\]

Shell 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 Shell 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 first lecture for Unit 8.

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".

![Doubly Linked List Diagram](image)

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

```java
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.
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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x.ch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y.prev</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y.next.prev</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y.prev.next</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y.prev.next.next</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Part II: Programming Problems (45 points total)

1. **Removing duplicates** (15 points; pair-optional)

   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.

   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 notes:**
   - 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 \textit{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.
• You should limit yourself to \(O(1)\) additional memory – i.e., a small number of additional variables. You should not use a second array.

2. **A merge-like approach to finding the intersection of two arrays** (15 pts; individual-only)
   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. **Improving bubble sort** (15 points total; pair-optional)
   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.

   a. (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. 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. 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 `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 containing 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(logn), O(nlogn), O(n²), 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 `ps7_experiments.txt`.**
Submitting Your Work
You should use the ps7 folder in the homework submissions dropbox to submit the following files:

- your ps7_partI.txt file containing your part I answers
- your modified Sort.java file
- your Intersect.java file
- your modified SortCount.java file
- your ps7_experiments.txt file

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 ps7.
- 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).