Homework 5. May 10th: 11:59pm. 100 points.

This project has three parts: theory, coding, C part and an optional extra credit.

Point distribution:

Theoretical (45 points)

- Problem 1: 5 points
- Problem 2: 6 points
- Problem 3: 4 points
- Problem 4: 5 points
- Problem 5: 10 points
- Problem 6: 15 points

Practical: Counting vs Merge (45 points)

- Counting Sort
- Merge Sort
- Comparison (Code and Chart)
- Conclusions

C problem (10 points)

- Bubble sort

Extra credit problems (8 points)

- Find the smallest k values in an array in a sorted manner

Files to submit:

- HW5_1.pdf
- HW5_2.pdf
- hw5_C.c
- ExtraCredit.pdf (optional Extra credit part)
- QuickSort.java (optional Extra credit part)

Part 1

Create **HW5_1.pdf** file and write all your answers there

Problem 1. For a given array A = (9, 5, 11, 3, 2, 8, 4, 6) and a partition method **below**, show the resulting array after making a call to Partition(A, 0, 7). Show each iteration step in detail in a table of the following format:

j	Action	Array contents

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Problem 2. You need to sort numbers {4, 6, 2, 5, 3, 12, 13} in DECREASING order. Show the resulting array after each step of the algorithm (each outer loop) in a neat table format using:

a. Selection sort
b. Insertion sort
c. Bubble sort

Problem 3. Sorting question

Give an **efficient** O(n) algorithm to rearrange an array of *n* keys so that all the negative keys precede all the nonnegative keys. Your algorithm must be in-place, meaning you cannot allocate another array to temporarily hold the items.

Problem 4. Counting Sort

This algorithm does not compare elements among themselves, it counts them. The algorithm can sort numbers in O(n+k) steps, where k is the maximum element.

A few links for you to read:

https://en.wikipedia.org/wiki/Counting_sort

http://www.geeksforgeeks.org/counting-sort/

https://www.cs.usfca.edu/~galles/visualization/CountingSort.html

After the algorithm makes sense, I'd like you to sort numbers {4, 9, 2, 3, 7, 8, 1}, using it. Show the resulting array after each step of the algorithm in a neat table format just like in earlier cases.

Problem 5. Find Duplicates

For this problem you are given an array of size n containing non-negative integers. Each integer is from the range [0, n-1]. Create an algorithm (English or pseudocode) that finds all duplicates in a given array and then outputs them.

Requirements:

- No duplicate values in the output array
- It is ok if the output is not sorted
- Your algorithm should work in O(n) time (tight bound).
- You can't use hashing (in case you were going to).
- You must explain the complexity of your algorithm

Problem 6. Let's put it all together

Finally, compare the efficiency of sorting algorithms

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Counting Sort

under worst case and best case. If the efficiencies using Big Oh notation are different for worst and best cases, please explain what array condition or sorting strategy will cause the difference. You can have a large table with different algorithms and complexities.

Part 2. Counting Sort vs Merge Sort

Create **HW5_2.pdf** file and write all your answers there. By the way, I always wanted to run this experiment but never got a chance, so think of this exercise as a research problem. Once you have everything set up, plug in different values of k, and step size to see the changes.

In this part you need to compare the efficiency of merge sort and counting sort in the following manner:

1) In a file named Sorting.java you need to create an array object of size n = 2000.
2) Populate the array with numbers in the range [0 - k] in a random order. Ensure that your array always contains the numbers 0 and k. Here k is the maximum number in the array. It is fine to have duplicates. Start with k = 100 and see how it goes.
3) Implement merge sort using a method mergeSort (you may create additional methods) and counting sort using a method countingSort. Write the best possible algorithms for each of these. If you use outside sources, please cite them.
4) Then you will run these methods on various k (max value) and calculate the average for these runs (say, 100 runs) for every k for more accurate estimates (just like HW3). Try different step sizes for increasing k and mention it clearly in comments. I have a feeling that increasing k by 1 will take a long time to see a difference.
5) Plot the average time against k for both the sorting methods.
6) You will observe that as you increase the maximum value k in your array at one point counting sort will start to become less efficient. Report this point at which merge sort beats counting sort.

Use the following APIs as in HW3:

Returns current time in milliseconds:
    static long System.currentTimeMillis()
Returns current time in nanoseconds:
    static long System.nanoTime()

Part 3. C problem: Implementing Bubble Sort.

Suggested reading (you can use any other source).

- Scanf, printf: http://www.cprogramming.com/tutorial/c/lesson1.html
- arrays in C: http://www.cprogramming.com/tutorial/c/lesson8.html
- loops: http://www.cprogramming.com/tutorial/c/lesson3.html

Your code should read an array from the user and sort it using **Bubble Sort** in a non-decreasing order**. Your algorithm should take O(n) in the best case.**
Sample run is shown below:

```
$ ./a.out
Enter the number of elements that will be in the array
4
Enter 4 integers
2
5
1
6
The sorted array is:
1
2
5
6
$ ./a.out
Enter the number of elements that will be in the array
7
Enter 7 integers
3
3
2
1
0
8
5
The sorted array is:
0
1
2
3
3
5
8
$ ./a.out
Enter the number of elements that will be in the array
5
Enter 5 integers
2
3
4
5
8
The sorted array is:
2
3
4
5
8
```

Do not change any print statement or other statements in the starter code except for filling up the TODO markers (This section is auto-graded). You can assume an intelligent user:

- No negative number as the length of the array.
- Number of elements are always integers.
- The array elements are integers.

What to submit:

- hw5_C.c

Submission

Files to submit:

- HW5_1.pdf
- HW5_2.pdf
- hw5_C.c
- ExtraCredit.pdf (optional Extra credit part)
- QuickSort.java (optional Extra credit part)

Remember:

- All relevant files must have the same file names as in “Files to submit”
- Make sure you hit **submit** button to submit your code
- **Do not forget to check submission.txt report for correctness**
- If submission.txt says you will get a 0 and you don’t change your files, you *will* get a 0 and cannot ask for a regrade

Extra Credit Problem

Modify Quicksort to find the smallest k values in an array of records. Your output should be the array modified so that the k smallest values are sorted in the first k positions of the array. Your algorithm should do the minimum amount of work necessary, that is, no more of the array than necessary should be sorted.

First, describe the algorithm in English or pseudocode in a file ExtraCredit.pdf
Second, implement it in QuickSort.java (For the implementation, start by creating an array of size n =100 and initialize it with random integers. Take the value of k as 30.)