CCS545-01, Spring 21
Prof. Karpenko

Project 1: Sorting[1]

Due: 02/26/2021, 11:59pm

For project1, you need to implement the following sorting methods:

- Insertion sort
- Cocktail Shaker sort
- Randomized Quick Sort
- Hybrid sort (that uses both quicksort and insertion sort)
- Bucket sort
- Radix sort
- External sort
- "A", "B", "C" sort - a linear time algorithm (other requirements apply, see the description below) for sorting an array that contains "A", "B", "C" characters and determining the character that occurs most frequently.
- One more sorting algorithm of your choice that we do not cover in class, that you need to research, understand, implement and describe in a Readme file.

You *may* use the sorting code that I posted on github (assuming you understand the code completely and can explain it), but you may **not** copy any code from any other source (even partially). You may **not** use any built-in sorting methods (or in-built classes from the Collections framework such as ArrayList, LinkedList, HashMap etc.) for this assignment.

Implementation Details

The starter code has been provided to you. Fill in code in class `SortingImplementation` that implements the following `SortingInterface` (it's important that you do **not** modify the signatures of any methods):

```
public interface SortingInterface {

        void insertionSort(Comparable[] array, int lowindex, int highindex,
        boolean reversed);

         void shakerSort(Comparable[] array, int lowindex, int highindex,
        boolean reversed);


        void randomizedQuickSort(Comparable[] array, int lowindex, int  highindex);

        void hybridSort(Comparable[] array, int lowindex, int highindex);

        void bucketSort(Elem[] array, int lowindex, int highindex, boolean reversed);
```

[1] Some *parts* of this assignment are modified from the assignment developed by Professor Galles.

```
        void radixSort(int[] array, int lowindex, int highindex, boolean reversed);

        void externalSort(String inputFile, String outputFile, int k, int m);

        String sortAndFindWinner (String[] votes);

        // Add one more sorting method here - find something interesting online,
        // research and implement

}
```

You should **not** use any instance variables for this assignment apart from constants; use only local method variables. You may write private helper methods.

We describe each of the sorting algorithms below:

1. Insertion Sort

```
public void insertionSort(Comparable[] array, int lowindex, int   highindex,
boolean reversed);
```

Modify the code of the insertion sort we discussed in class (posted on github) so that:

- It sorts all elements in the array with indices in the range from `lowindex` to `highindex` (inclusive). You should not change elements outside the range `lowindex` to `highindex`.
- If reversed is false, the list should be sorted in ascending order. If the reversed flag is true, the list should be sorted in descending order.
- Can sort the array of any Comparable objects, not just integers.

2. Cocktail Shaker Sort

```
public void shakerSort(Comparable[] array, int lowindex, int highindex, boolean
reversed);
```

This is a variation of the bubble sort that sorts in both directions (bubbles the largest element to the end, and bubbles the smallest element to the front on each pass). It has the same asymptotic running time as bubble sort.

The first pass consists of two parts: we first iterate over the array from left to right and bubble the largest element to the end of the list. Then we take the leftward pass (iterate from the (last-1) element to the first element) and bubble the smallest element to the front of the array.
The second pass will first bubble the second largest element to the end of the list (to index last-1) and then shift the second smallest element to the correct position at index 1.
...
After each pass, we reduce the size of the list that needs to be sorted by two elements.
Example (assume we want to sort the list in ascending order, and lowindex =0, and highindex = array.length-1):
4, 10, 6, 9, 2, 3, 8, 4
After the first part of pass 1 we get: 4, 6, 9, 2, 3, 8, 4, 10 (the largest element is in the last position).
After the second part of pass 2 (iterating from right to left): 2, 4, 6, 9, 3, 4, 8, 10. The smallest element is in the first position. Now we need to sort the list from index 1 to index last -1.

After the first part of pass 2, the list will look like this: 2, 4, 6, 3, 4, 8, 9, 10.
After the second part of pass 2 we get: 2, 3, 4, 6, 4, 8, 9, 10

We will now sort from index 2 to index 5 (inclusive):

After the first part of pass 3 we get: 2, 3, 4, 4, 6, 8, 9, 10. The list won't change after the second part of pass 3. It would now need to sort the list from index 3 to index 4. The list won't change and we are done.
Note : your code should work for any lowindex, highindex and in both ascending and descending order.

3. Randomized Quick Sort

```
public void randomizedQuickSort(Comparable[] array, int lowindex, int
highindex);
```

Change the code of the quick sort we discussed in class (that is posted on github) so that:

- It sorts the sub-list of the original list (from lowindex to highindex)
- At each pass, it picks **three random elements** of the sub-list, chooses the **median** of these three elements and uses it as a pivot. How do you compute a median of three values? If we "sort" three elements, the median is the element in the middle (for instance, if the three elements are 5, 2, 19, the median is 5); note that it is different from *mean*! Such algorithm is called a *randomized quick-sort.* The expected running time of a randomized quick sort is O(n log n).

Example: Consider the following array 5, 2, 9, 12, 6, 8, 3, 7 and assume we want to sort the whole array (so lowindex = 0, highindex = 7). We first generate three random integers from 0 to 7 (assuming a uniform distribution) and we get indices 1, 7, 4. These indices correspond to elements 2, 7, 6 of the array. The median of (2, 7, 6) is a 6 (because 6 is in-between 2 and 7). So our pivot for the first pass is a 6. We then run quicksort as usual:
5, 2, 9, 12, 7, 8, 3, 6 (swap the pivot with the last element)
Move i until it points at 9. j points at 3. Swap them:
5, 2, 3, 12, 7, 8, 9, 6
i now points at 12, j moves until it crosses i and points at 3 (because elements 8, 7, 12 are all larger than the pivot 6). We swap the pivot with the element at i and get:
5, 2, 3, 6, 7, 8, 9, 12
So the first pass split the list into elements < 6, 6 and elements > 6. We now need to recursively run randomized quicksort on sublists 5, 2, 3 and 7, 8, 9, 12. For each sublist, we would again pick three random elements of the sublists and choose a median as a pivot. If the sublist contains only two elements, randomly pick one of the two as the pivot. Finish this example before you start coding randomized quick sort.

4. Hybrid Sort

```
public void hybridSort(Comparable[] array, int lowindex, int highindex);
```

For large lists, quicksort tends to be the fastest general-purpose (comparison) sorting algorithm in practice. However, it runs slower than some of the $\Theta(n^2)$ algorithms on small lists. Since it's a recursive divide and conquer algorithm, it needs to sort many small sublists. You need to design a hybrid sorting algorithm that **combines quicksort with insertion sort** to make quicksort faster. Hint: You can run quicksort as usual until the sublists become small (say, when a number of elements in a sublist is less than a certain threshold), and then use insertion sort to sort the small lists. If you want to implement an alternative hybrid sort, discuss it with the instructor to confirm it's a good alternative to the one proposed above.

For this algorithm, you also need to **design tests and run them on both the randomized quick sort and on hybrid sort** to see if your hybrid sort is faster. Make sure your hybrid sort is fast on all kinds of lists, including random, sorted and inverse sorted lists.
Submit the java file with the tests and a README that describes your hybrid algorithm and the results of the tests.

5. Bucket Sort

```
public void bucketSort(Elem[] array, int lowindex, int highindex, boolean
reversed);
```

You need to implement a bucket sort that we discussed in class. It should sort an array of elements (also referred to as "records"), where each `element` contains an integer key and data of type `Object`:

```
public class Elem {
	private int key;
	private Object data;

	// Constructor, getters, etc.
}
```

The array should be sorted based on the **key**. The number of buckets should be the number of elements to be sorted divided by two. Please note that the number of buckets is **not** `array.length/2`, but `(highindex - lowindex + 1)/2`. First, iterate over the list to compute the maximum value stored in the list. Then you can assume the range of elements is from 0 to the maximum (we assume here that the keys are >=0). The size of all buckets should be the same.
Note that you are not sorting the list of Comparable-s here (but a list of records with integer keys): remember that the **bucket sort is *not* a comparison-based algorithm.**
If `reversed` is false, the list should be sorted in ascending order, otherwise in descending order.
For the bucket sort, you may use either your own class LinkedList or Java's built-in LinkedList class. But you need to make sure the linked lists are maintained in sorted order - so when you insert a new key, make sure you insert it in the right place into the sorted LinkedList.

6. Radix Sort

```
public void radixSort(int[] array, int lowindex, int highindex, boolean
reversed);
```

When you implement a radix sort, you need to first sort by the least important digit. Use can base 10 for your implementation of the radix sort, or pick some other base.

7. External Sort

```
public void externalSort(String inputFile, String outputFile, int k, int m);
```

What if we need to sort a *very large list* that does not fit into memory all at once? Then you need to use external sort that stores partial results in files on the disk. Assume we can only fit `k` integers into memory at a time, and we have a text file that contains **`N`** integers (one per line, to keep it simple). We can read k integers from the file at a time, store them in a list and sort the list using some existing **efficient** sorting algorithm such as quicksort. Then we can write the result to a temporary file. We can repeat this process for another chunk of the original file. We would need to do it `m (number of chunks) = ceiling(N/k) times`, until all

the partial results are stored in temporary files (the provided test expects you to call them "temp0.txt", "temp1.txt", ..., "temp99.txt"). In this method, `k` (how many integers can fit in an array) and `m` (the number of chunks) are passed as parameters to the method. Please note that N is *not* passed as a parameter (but it can be determined from the number of lines in a file).

We need to merge the sorted sublists stored in the temp files into a single sorted "list" stored in the output file. You can use the algorithm similar to the merge step of the mergesort, except that you would read data from the temp files (all of them need to be open at the same time, you may use an array of BufferedReaders for that) and keep writing the numbers to another file as your algorithm proceeds with the merge. You will read the first number from each temporary file, find the minimum of those numbers, and write it to the output file. Then read another number *from the file that contained the minimum* and again find the minimum of the currently read set of numbers and write it to the output file. For this problem, the list in the output file should be sorted ***in ascending order***.
Consider the following **example** (to keep the example short, let's assume the input file has 6 numbers, and our tiny memory is only able to fit k = 3 integers at a time):
8
4
10
3
7
5

The external sort algorithm will read m = 2 chunks from the input file: [8, 4, 10] and [3, 7, 5], sort them with quicksort and save the **sorted** sublists in two temporary files:
"temp0.txt"
4
8
10

"temp1.txt"
3
5
7

It will then open both temp files and merge them as following: it will first read 4 and 3 (the first numbers in each temp file), save them into the temporary array, find the minimum (3) and write it to the output file (assume it is called "output"):
"output"
3

Then it would read another number from "temp1" since it's the file that contained the minimum element. Now the array of elements is [4 , 5], the minimum is a 4 and we write it to the output file"
"output"
3
4

We read another number from "temp0", 8, and the array is [5, 8]. The minimum is a 5, we

write it to the output file:
"output"
3
4
5

We read another number from "temp1", a 7, the array is now [8, 7], the minimum is 7 and the output file is:
"output"
3
4
5
7

We continue as before until we write all the elements to the output file. The temporary files can then be deleted (although you are not required to delete them in your program).

To test your external sort, create a large file of integers (also done in the test file, provided by the instructor).

8. "A", "B", "C" Sort

```
public String sortAndFindWinner (String[] votes);
```

Suppose some city has **n** people, and these people need to vote to select a mayor of the city. There are three candidates for a mayor: "A", "B" and "C". We are given an array of **n** Strings where each element represents a vote for either candidate "A" or candidate "B", or candidate "C". For the purpose of this problem, let's assume there is a clear winner (so one candidate has more votes than the other two).

Design and implement an in-place algorithm **for sorting** this array and determining **who wins** the election, "A", "B" or "C".
Example: if we are given the following array that represents votes of 11 people:
["A", "B", "A", "C", "A", "A", "A", "B", "C", "A", "B"],
your method should return "A" and change the array so that it is sorted:
["A", "A", "A", "A", "A", "A", "B", "B", "B", "C", "C"]

The algorithm needs to satisfy the following requirements:

- Use the variation of the partition method of quicksort (should build on top of the algorithm you wrote in code camp 1 for sorting "A" and "B")
- Should run in linear time
- Use **no** extra memory (except for two integer indices and a `tmp` variable for swapping).
- Run in two passes

Do NOT just iterate over the array and count the number of "A"s, "B"s and "C"s - such solutions will get 0 points. Do NOT use counting sort or its variations. The solution needs to use a variation of partition method of quicksort. No other solutions will get credit.

9. One more sorting algorithm that we did not cover in class, and that is not yet mentioned above.

Implement one more sorting algorithm of your choice, that you need to research, understand, implement and describe in a Readme file.

Testing

The instructor provided some basic tests for testing several methods of the project, but they simply check if the list is sorted after running your algorithm. Passing the tests does *not* guarantee that your code is correct; you should also do your own testing and are encouraged to write your own tests.

Submission

Submit project 1 to your private github repo by the deadline. Only github submissions are accepted. You need to have at least 5 meaningful commits before the deadline.

Code Style

Your code needs to adhere to the code style described in the *StyleGuidelines.pdf* document on Canvas. You may receive a deduction if your code does not follow these guideilines.

Interactive Code Reviews:

I may invite several students for an interactive code review for this project. Please come prepared to answer any questions about your code and being able to reproduce parts of it during the code review. If you are not able to explain your code or reproduce parts of it, you may **not** get **any** credit for it.

Please do **not** copy any code from the web or other people, and do **not** discuss implementation details with anybody apart from the instructor, the TAs and the CS tutors. Sorting-related questions often come up during job interviews; this project is a great chance for you to practice writing these algorithms.