CCS245-03, Spring 18
Prof. Karpenko

Project 3, Part 2: Sorting algorithms II[1]

Due: 04/09/2018, 11:59pm

For part 2 of project 3, you need to
1. Implement the following sorting methods:
    - Bucket Sort
    - Radix Sort
    - External Merge Sort (for sorting a very large list stored in a file that does not fit into memory all at once).

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, HashMap etc.) for this assignment.

Implementation Details

You will need to write a class called `SortingAlgorithms` that implements the following `SortInterface` (it's important that you do **not** modify the signatures of any methods):

```
public interface SortInterface {

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

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

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

}
```

Your `SortingAlgorithms` class must implement the interface above. You should **not** use any instance variables for this assignment apart from constants, only local method variables. You may write private helper methods (you would need them for several sorts such as a heap sort).

All sorting algorithms you will write should sort a list *in ascending order* (from smallest to largest).

We describe each of the sorting algorithms below:

- **Bucket Sort**

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

You need to implement a bucket sort that we discussed in class. It should sort an array of

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

Elem-s ("records"), where each `Elem` contains an integer key and some data:

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

      // constructor, getters etc.
 }
```

The array should be sorted based on the **key in ascending order**. 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 (of type Elem) with integer keys: it's because the bucket sort is *not* a comparison-based algorithm.

- **Radix Sort**

`public void radixSort(int[] array, int lowindex, int highindex)`
You need to first sort by the least important digit. You would need to modify the code that was posted on Canvas so that it is able to sort elements in the range from lowindex to highindex (inclusive). Use base 10 in your implementation of the radix sort.

- **External Sort**

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

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 the 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 Theta(n log n) 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 `numChunks = ceiling(N/K) times`, until all the partial results are stored in temporary files.

We then need to merge `the` sorted sublists stored in the temp files into a single sorted list. 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 3 integers at a time):

4
10
3
7
5

The external sort algorithm will first read 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:
"temp1"
4
8
10

"temp2"
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 "temp2" 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 "temp1", 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 "temp2", 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.

Submission

Submit project3 part 2 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

Starting this project, your code needs to adhere to the code style described in the *CodeStyle.pdf* document on Canvas.

Interactive Code Reviews:

I will invite several students for an interactive code review for this project. Please come prepared to answer any questions about your code. If you are not able to explain your code (even one method), you may **not** get **any** credit for the whole project.

Please do **not** copy any code from the web, and do **not** discuss low-level 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.