Project 1 - LSManga

Recursive Sorting

Group 16

Arnau Castellà (arnau.castella)

Index

Contenido

Justification of programming language

I decided to use Java to develop this project for many reasons. First of all, it is the language that we have been using in class, so it is the one that I currently work most comfortable with.

Another important reason that influenced my decision, is how easy it is to work with JSON files with Java. Using the Gson library, it is extremely easy to parse the information to a Java object.

Moreover, Java has a lot of available packages that are very useful. For example, the methods included in the Arrays and Lists packages facilitate a lot of work that you would need to develop in languages such as C.

Finally, a very important reason why I chose Java is the IDE that I use to work with it, IntelliJ IDEA. The debugger in this IDE is extremely useful when it comes to solving problems in your code.

Sorting algorithms

In this project, I have developed three different sorting algorithms: Quicksort, Merge sort and Bucket sort.

Quicksort

This is a recursive sorting algorithm that works by moving elements from an array around a certain position called pivot. The inputs of this function are the array to be sorted and the starting and ending positions that we want to sort. In my case, I have used the average score as the key factor to sort the array using this sorting algorithm.

```
public static void quickSort(Serie[] arr, int start, int end) {
    int index = partition(arr, start, end);
    if (start < index - 1) quickSort(arr, start,  end: index-1);
    if (index < end) quickSort(arr, index, end);
}
```

The partition function gets the value in the last position of the array and arranges it in that same array, returning the final position where that value ended. We now know that this particular position is sorted so we recall the function two separate times, one for the lower part of the array from the pivot position, and one to sort the upper part of the array.

```java
//Sorts the value at the middle position and returns its sorted position (pivot)
private static int partition(Serie[] arr, int start, int end) {
    int i = start, j = end;
    int pivot = arr[(start+end)/2].getAverageScore();
    while (i <= j) {
        while (arr[i].getAverageScore() < pivot) i++;
        while (arr[j].getAverageScore() > pivot) j--;
        if (i <= j) {
            swap(arr, i, j);
            i++;
            j--;
        }
    }
    return i;
}
```

The value that I use as pivot is the one in the middle of the given array. The pivot could be placed in any position of the array, however this one is useful to reduce the cost in case that the array is already sorted.

Merge sort

This recursive sorting algorithm is based on the “Divide and conquer” principle. Its functioning is to divide the array in half and once we have the individual positions of the array, start merging them in the desired order.

```java
public static Serie[] mergeSort (Serie[] arr) {
    if (arr.length == 1) return arr;

    Serie[] arrOne = Arrays.copyOfRange(arr, from: 0, to: arr.length/2);
    Serie[] arrTwo = Arrays.copyOfRange(arr, from: arr.length/2, arr.length);

    arrOne = mergeSort(arrOne);
    arrTwo = mergeSort(arrTwo);

    return merge(arrOne, arrTwo);
}
```

The merge function gets two arrays and transforms them to lists. This step is necessary because the merge function will always join the arrays in order, so the first position of the array will always be the smallest one. By using lists, we are able to remove the first position and the entire list will shift to the left.

To sort using this algorithm, I have used the start date of each series and I sort them from old to new.

```java
//Merge sort by date (old - new)
private static Serie[] merge (Serie[] arrOne, Serie[] arrTwo) {
    Serie[] merged = new Serie[arrOne.length + arrTwo.length];
    List<Serie> listOne = new ArrayList<>(Arrays.asList(arrOne));
    List<Serie> listTwo = new ArrayList<>(Arrays.asList(arrTwo));
    int i = 0;
    while (listOne.size() > 0 && listTwo.size() > 0) {
        if (listOne.get(0).getStartDate().after(listTwo.get(0).getStartDate())) {
            merged[i++] = listTwo.get(0);
            listTwo.remove( index: 0);
        } else {
            merged[i++] = listOne.get(0);
            listOne.remove( index: 0);
        }
    }
    while (listOne.size() > 0) {
        merged[i++] = listOne.get(0);
        listOne.remove( index: 0);
    }
    while (listTwo.size() > 0) {
        merged[i++] = listTwo.get(0);
        listTwo.remove( index: 0);
    }
    return merged;
}
```

Bucket sort

This sorting algorithm takes each element of the array and stores it in a separate array, also known as “bucket”, depending on the value of the element. There is no specification when it comes to the number of buckets, so I have decided to always create sqrt(n) buckets, where n is the size of the array.

Once all the values have been introduced to their respective buckets, I use quicksort to sort them based on the previously calculated priorities, so the cost of the algorithm is O(n + k), where n is the cost of inserting the values into buckets and k is the cost of sorting them.

```
public static void bucketSort (Serie[] arr, int numBuckets) {
    //Average score ranges between 0 and 100, so that is the min and max values used
    int MAX_VAL = 100, MIN_VAL = 0;

    double range = (double)(MAX_VAL - MIN_VAL) / numBuckets;
    Serie[][] buckets = new Serie[numBuckets][0];

    //Insert values into buckets
    for (Serie serie : arr) {
        int index = (int)(serie.getAverageScore()/range);
        buckets[index] = putInBucket(buckets[index], serie);
    }
    //Sort each bucket
    for (Serie[] bucket : buckets) {
        bsQuickSort(bucket, start: 0, end: bucket.length-1);
    }
    //Merge every bucket into arr in descending order
    int j = 0;
    for (int i = buckets.length-1; i >= 0; i--) {
        for (Serie serie : buckets[i]) {
            arr[j++] = serie;
        }
    }
}
```

In order to assign these priorities to each position of the array, I call a function at the beginning of my main procedure once we have the data from the JSON file.

The way I have implemented this calculation is by giving a score out of ten to each characteristic (popularity, average score and favouritism), dividing it by three and adding them up. In the case of popularity and favouritism, first of all I search the maximum value in the array and then I calculate each score based on that value, so only the one that has the most popularity will have a 10/10 in that particular score, and the same for favouritism. The average score is already a grade out of 100 so I just divide that value.

```
private static void calculatePriorities(Serie[] arr) {
    float maxPopularity = 0;
    float maxFavourites = 0;
    //Find max popularity and favourites
    for (Serie serie : arr) {
        if (serie.getPopularity() > maxPopularity) maxPopularity = serie.getPopularity();
        if (serie.getFavourites() > maxFavourites) maxFavourites = serie.getFavourites();
    }
    for (Serie serie : arr) {
        serie.setCombinedPriorities((serie.getPopularity() / maxPopularity * 10 / 3) +
                (serie.getFavourites() / maxFavourites * 10 / 3) +
                (float) serie.getAverageScore() / 100 * 10 / 3);
    }
}
```

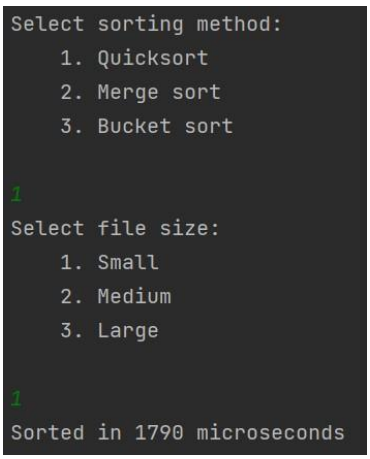
Results analysis

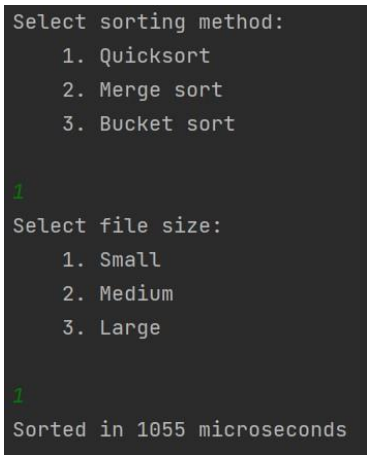
To compare the different algorithms, I am going to execute them three times for each dataset and measure the time it takes. I am going to calculate the average time and compare each algorithm.

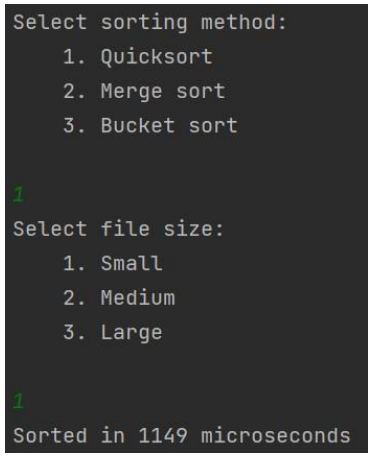
To do so, I call the System.nanoTime function before and after calling the algorithm and I these values to get the runtime of the algorithm.

```
//Bucket sort
long startTime = System.nanoTime();
Serie[] sorted = Sorter.bucketSort(arr,  digit: 100);
long stopTime = System.nanoTime();
//for (Serie serie : sorted) {
//    System.out.println(serie.getTitle().getEnglish() + ' ' + serie.getAverageScore());
//}
System.out.println("Sorted in " + (stopTime - startTime)/1000 + " microseconds.");
```

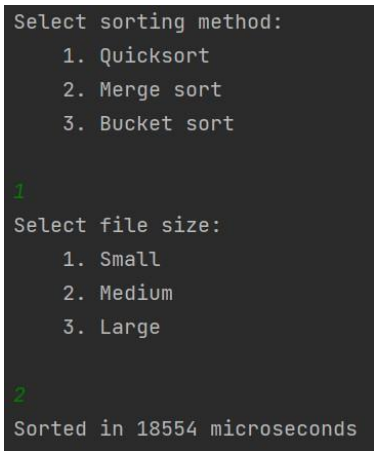
Quicksort

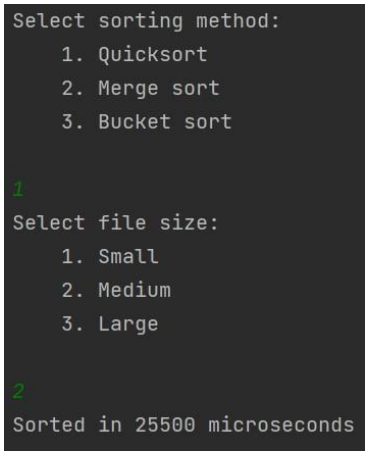


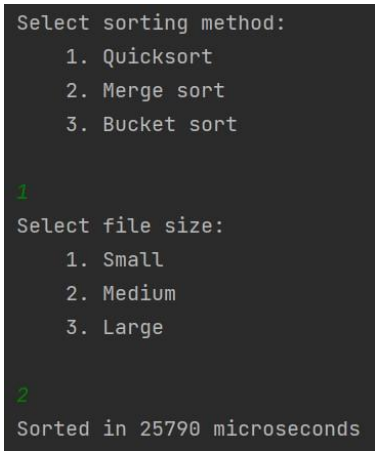




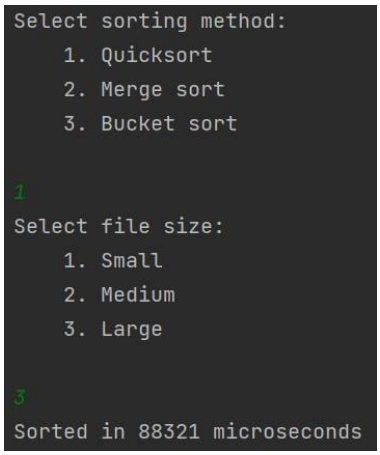
Quicksort with small file average time = 1331.3 us

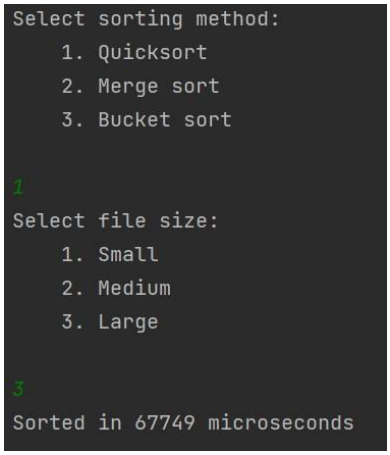


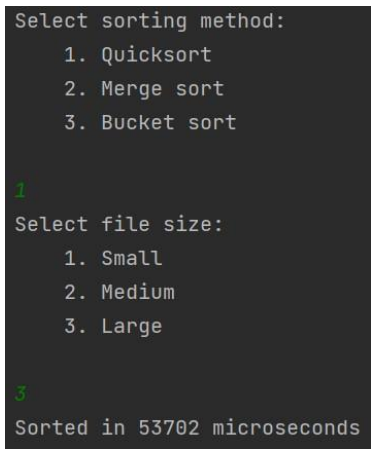




Quicksort with medium file average time = 23,281.3 us

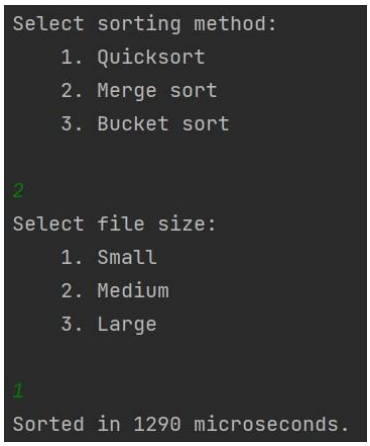


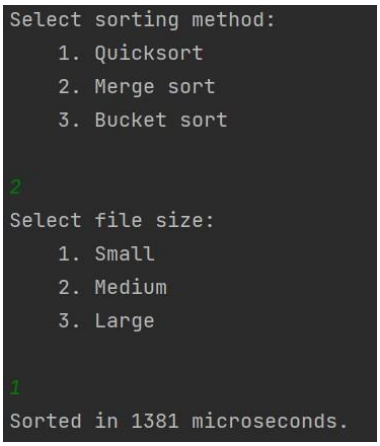


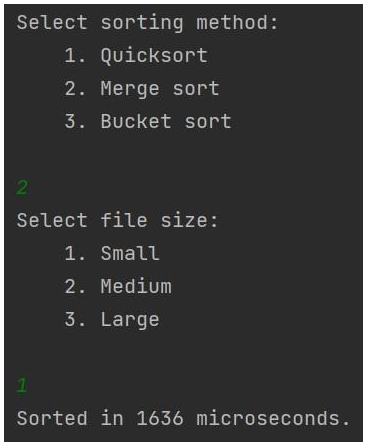


Quicksort with large file average time = 69,924 us

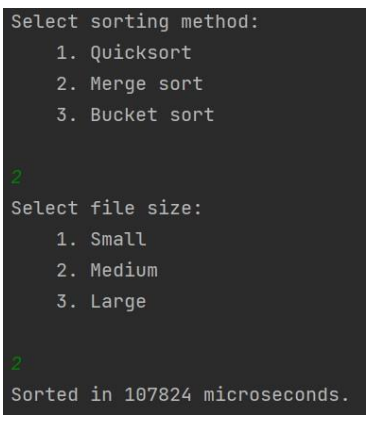
Merge sort

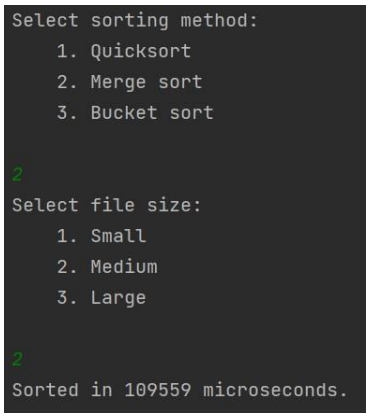


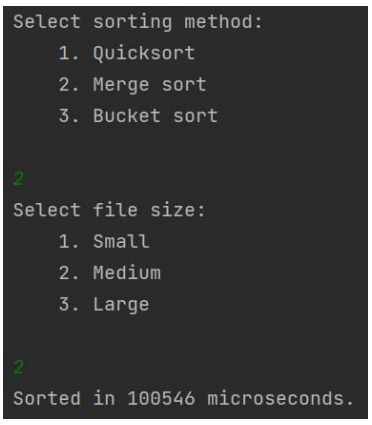




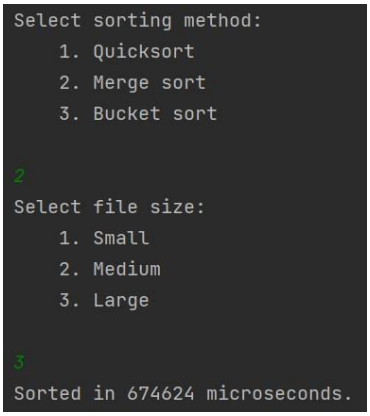
Merge sort with small file average time = 1,435.6 us

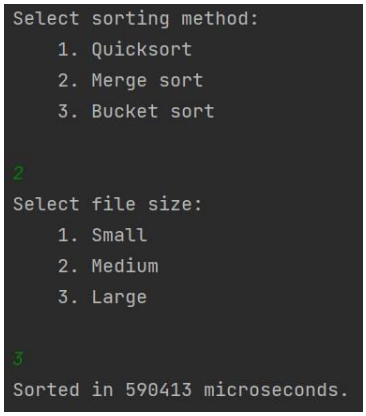


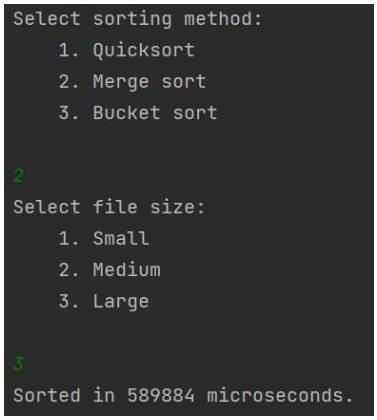




Merge sort with medium file average time = 105,976.3 us

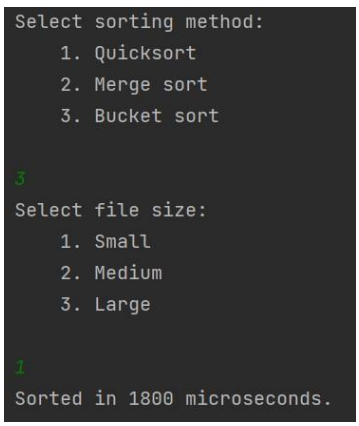


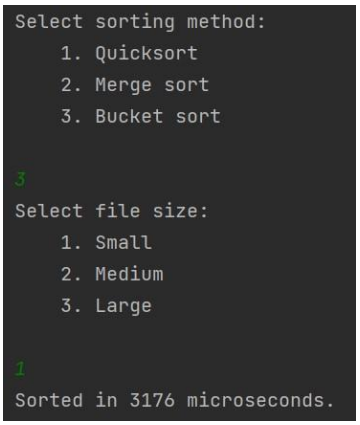


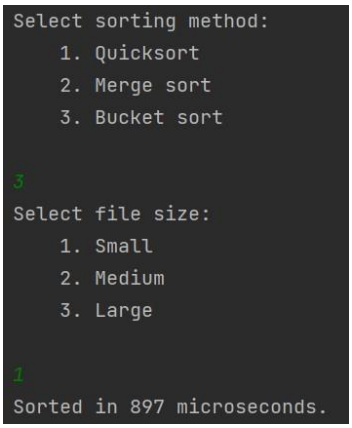


Merge sort with large file average time = 618,307 us

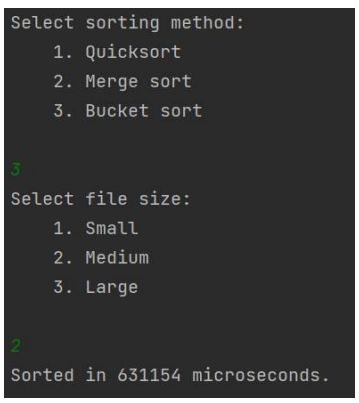
Bucket sort

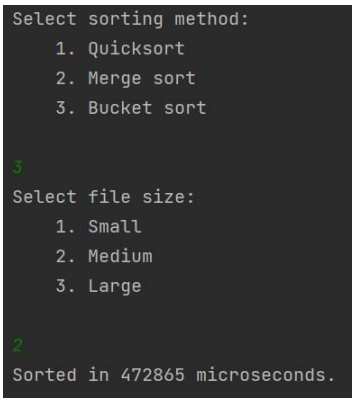


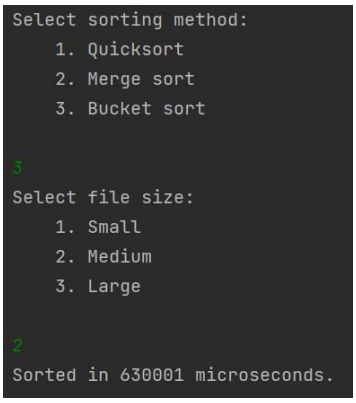




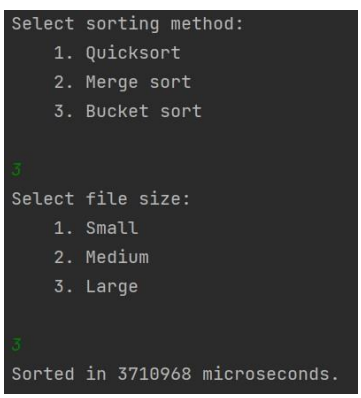
Bucket sort with small file average time = 1,957.6 us

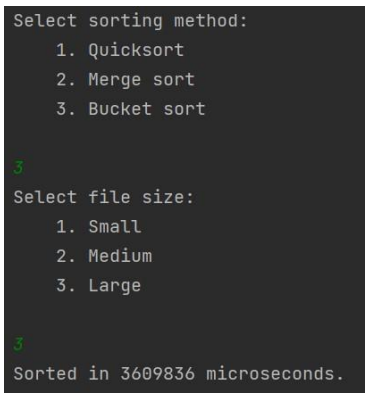


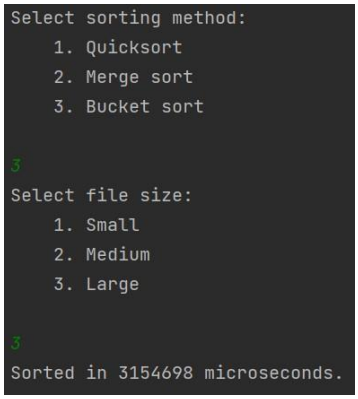




Bucket sort with medium file average time = 578,006.6 us







Bucket sort with large file average time = 3,491,834 us

	Small file	**Medium file**	**Large file**
Quicksort O(nlogn)	1,331 us	23,281 us	69,924 us
Merge sort O(nlogn)	1,435 us	105,976 us	618,307 us
Bucket sort O(n + k)	1,957 us	578,006 us	3,491,834 us

As we can see, the quicksort is the most efficient algorithm in all cases. Merge sort is very fast with small inputs, but as the input scales, its cost increments a lot. Bucket sort has the same problem, but its cost scales even faster.

The theoretical cost of quicksort and merge sort algorithms is the same, so it is surprising to see how different the results found are. For small data sets they behave similarly, however merge sort increases its cost much faster.

Observed problems

I did not have any major problem during the development of this project. The most notorious one was that, at the beginning of the implementation, I was using the average score to sort with Quicksort. This caused the program to crash only when I tried running it with the large file with a stackoverflow error.

However, I decided that it was better to use the average score to sort with bucket sort in order to reduce the number of digits to analyse, since I was sorting it by priorities which was a float. When I changed quicksort to sort by the given priorities, the error never happened again.

Conclusions

Even though I was not able to finish this project on time, I had a fun time exploring the different sorting algorithms and experimenting with them. I have definitely gained a deeper understanding of recursion and sorting algorithms thanks to the development of this practice.

Quicksort, although it can be trickier to implement sometimes, has proved to be the most efficient sorting algorithm of the three, so that is the main point that I take from this practice.

Bibliography

GeeksforGeeks. (2021). Bucket Sort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/bucket-sort-2/ [Accessed 6 Jan. 2022].

GeeksforGeeks. (2021). Merge Sort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/merge-sort/ [Accessed 5 Jan. 2022].

GeeksforGeeks. (2021). QuickSort - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/quick-sort/ [Accessed 18 Nov. 2021].

Youtube. (2020). Algoritmos - Ordenamiento rápido - Quick Sort – Lupo Montero. [online] Available at: https://www.youtube.com/watch?v=DYmTpUfcyT8&ab_channel=LupoMontero [Accessed 18 Nov. 2021]

Youtube. (2018). Merge sort in 3 minutes – Michael Sambol. [online] Available at: https://www.youtube.com/watch?v=4VqmGXwpLqc&ab_channel=MichaelSambol [Accessed 5 Jan. 2022]

TutorialsPoint. (2021). Data Structure and Algorithms – Quicksort – TutorialsPoint. [online] Available at: https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm [Accessed 18 Nov. 2021]