Advanced Programming and Data Structure
Course 2022-2023

Project 1 of the first semestre – Recursive Sorting

ZaballosZapatery

G03

Students	Login	Name
	adrianjorge.sanchez	Adrián Sánchez López

Date	26/11/2023

Table of Contents

Programming language choice

Primarily I have chosen Java over other programming languages to keep practicing in Java, as this year I do not use Java in any subject, so to remain used to it, I preferred to use Java over C. Although this is the main reason it is also a plus that Java, specially in IntelliJ, lets you take shortcuts and has a lot of prebuilt methods that you can use to speed up the time it takes you to program.

It is true however that compilation in C can be optimized to be faster than Java since it doesn't need to be interpreted and gets translated to machine code straight after compiling. Although this seems like an advantage when one of the aspects we want to test with this project is time efficiency, we are only analyzing the efficiency of each algorithm, so this doesn't really matter as long as you can appreciate how effective one is compared to another and how it varies with the input. C is also more flexible with memory usage however, once more, this doesn't contribute to the objective of the investigation.

Moreover, Java makes it easier to work with lists of objects. In my case, I knew I was going to deal with a large amount of data; and Java lets you add objects to a list, swap them, and remove them in an easy, intuitive, and fast way.

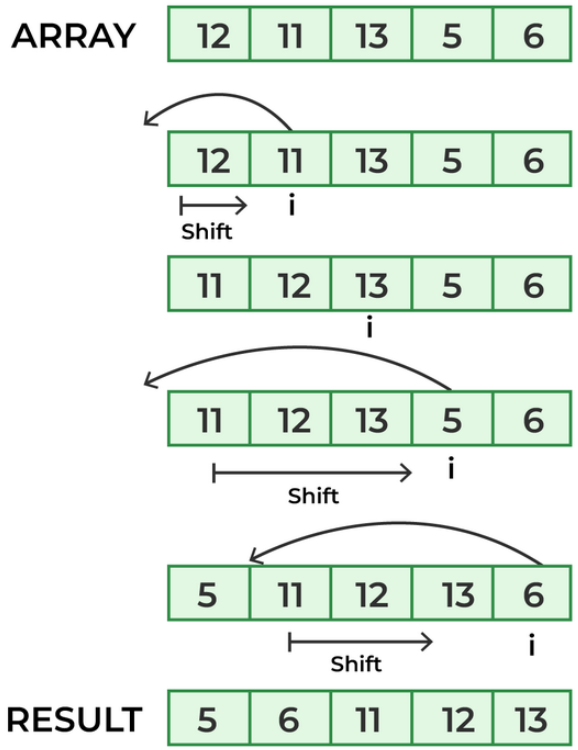
Algorithms:

Sort by name:

The insertion sort and selection sort algorithms have been implemented to sort the shoes by name. To be able to sort by name I have created the function strcmp, which does the same as the strcmp function of the string.h in c. This function receives two strings and loops until the end of the smallest string. During the loop it compares character by character both strings, if the character is different it returns the difference between both characters in ASCII(character of string 1 - character of string 2). If the loop ends, meaning all characters were equal, it checks if both strings were equal in length, if they aren't, it returns the length of the first minus the second. If they are then it returns 0. Therefore, with this function if it returns 0 it means they are equal, if it returns a positive value it means that the second string comes before the first alphabetically. If it returns a negative value, then the first string comes first.

Insertion Sort

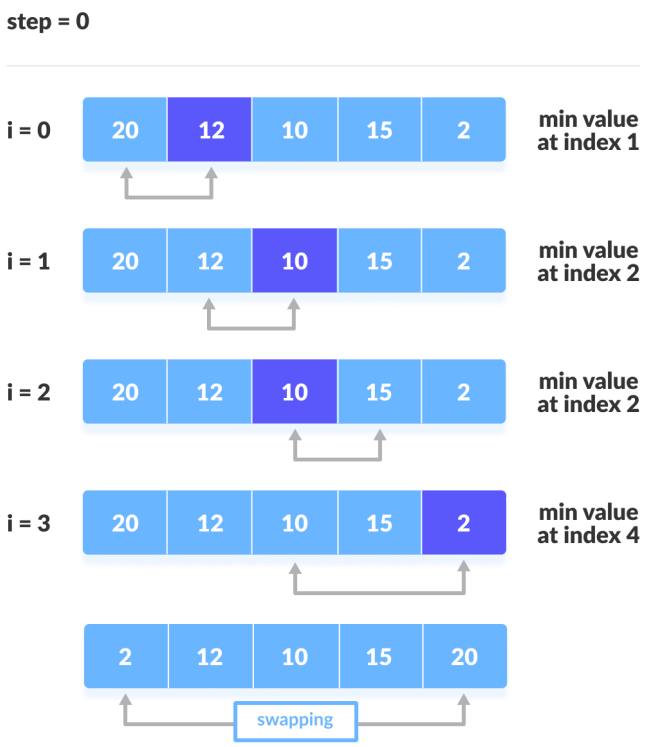
Insertion sort is an algorithm that sorts an array similar to how a person will sort a deck of cards. For example, given an array, it will go through the array comparing each element to its predecessor, if it comes before then it will swap them and compare the element with its new predecessor and will do so until its predecessor comes before. It has an average cost of O(N^2). It remains the same for the worst case. But in the best case it has a cost of O(N), as it will go through the array one time, as each time it compares an element the one before will come first, so it will pass to compare the next element which will be the same case. The advantage of insertion sort is that it is pretty simple to implement and works well for small datasets or partially sorted datasets as it will skip a lot of iterations.



To implement this algorithm only 1 function called insertionSort has been implemented as it is pretty easy to implement. This function receives an array and loops through the array starting in the second position and then for each element it has a while that loops while the index of its predecessor is bigger or equal to 0 and strcmp of the predecessor and the element returns 0 or bigger. Inside the while it puts the predecessor in the next position, and when the loop ends, it puts the element in the last position where the element came before the predecessor.

Selection Sort

Selection sort is an algorithm that sorts an array by comparing each element to all the elements that come after it and replacing the element with the smallest or biggest element found(depending on how you are sorting the array). It has a cost of O(N^2) for all cases, as no matter if it is the best case or the worst case it will always go through each element and compare it to all the other elements. The advantage of it, is that it is easy to implement and works well for small datasets.



To implement it, only the function selectionSort has been implemented, as it is pretty easy to implement. SelectionSort receives the array to be sorted and has a for loop that goes through all the elements inside it. For each element it stores it as the smallest element found and it has another for loop that goes from the next element to that one until the end of the array and has an if that checks the value returned of the strcmp passing the next element name and the smallest element name. If it returns a negative number, then the element compared is stored as the smallest element. Finally, after this for loop, it swaps the smallest element with the element of the first loop.

Sort by quality:

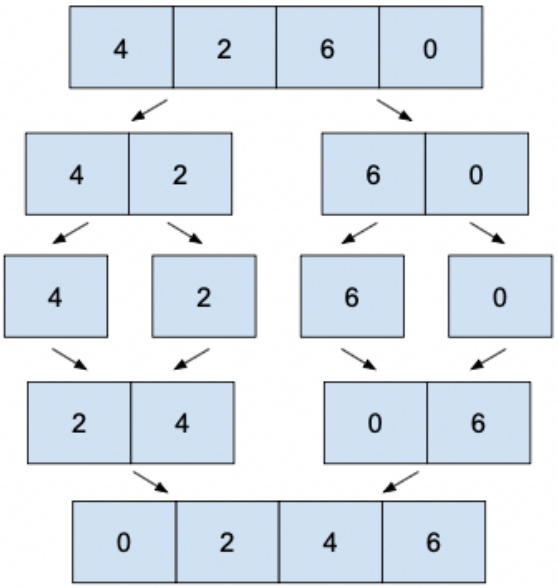
The quicksort and mergesort algorithms have been implemented to sort the shoes by quality, which is an attribute I have created by using all of the other attributes except for the name, being the bigger the quality number, the greater the shoe is. The formula used to create the quality attribute is the following:

$$\frac{Score \cdot 0.7 + 0.3 \cdot (size_{max} - size_{min})}{prize \cdot \frac{weight}{1000}} \cdot 300$$

The quality is made by multiplying the score time 0.7 as I consider the score given by the customers highly important. Then the range of sizes(maxSize - minSize) is multiplied by 0.3 as I consider the availability of the shoe another important factor, but not as much as the score. Finally, both attributes are added. Then, the result is divided by the prize times the weight in kilograms, as I consider a shoe to be better if it weighs less and if it costs less. After the division the result is multiplied by 300 to show it in numbers higher than 0 so it is shown in a more understandable way.

Mergesort

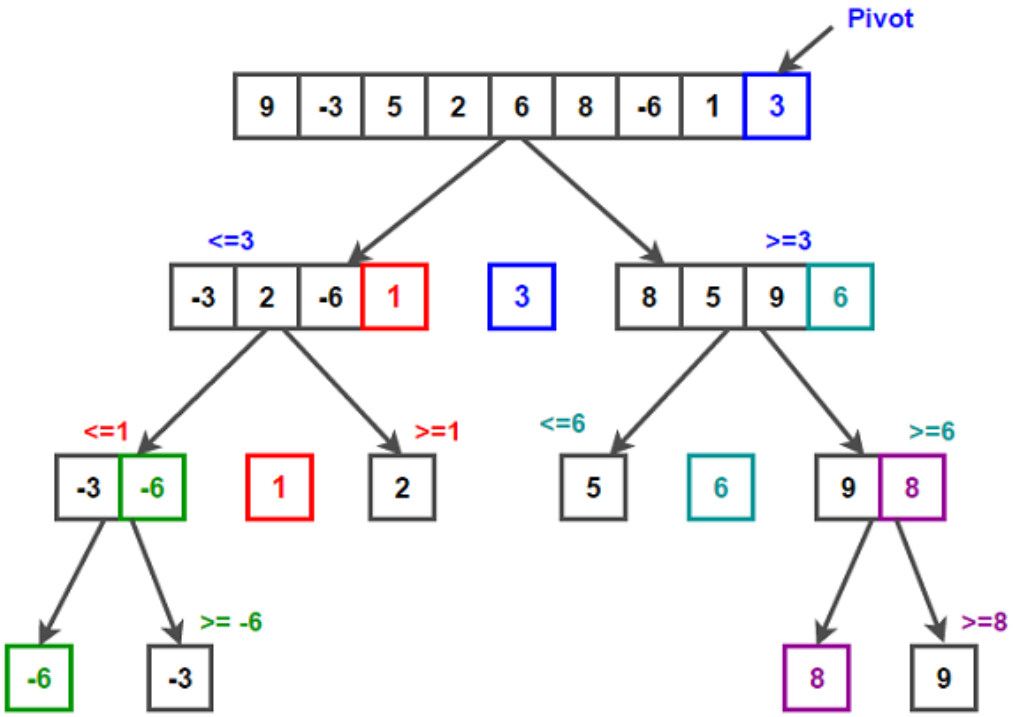
Mergesort is an algorithm that, given an array, splits it into 2 subarrays recursively until it cannot split it more. Once it cannot be splitted more, then it starts merging them, putting the smaller elements on the left and the bigger on the right. It has a cost of O(N*log(N)). The advantage of merge sort is that it always has the same cost, no matter if it is the worst or best case, as it always divides the array into two parts and it takes linear time to merge two parts. The drawback is that it wastes a lot of memory as it has to store in auxiliary memory all the subarrays it creates.



This algorithm has been implemented to sort the shoes by quality, as it has been previously explained. To implement it two functions have been created: mergesort and merge. Mergesort is a recursive function that given an array, a left index and a right index, while the left index is smaller than the right index. It calculates the middle index and calls itself recursively passing the left index as left and the middle index as right. After that it calls itself recursively again but passing as left the middle index plus 1 and the right index as right. Doing that, the function will constantly call itself 2 times, giving one time as the array the left part of the array given to that function and the other time the right part. Once it has called itself recursively for both subarrays, then it calls the function merge and finishes. The function merge receives an array and a left, a middle and a right index. Firstly it creates two temporary arrays where it adds the left part of the given array to one, and the second part to the other array. Then it starts a while loop until one of the temporary arrays has reached its end. Inside that loop it has an if that checks if the quality of the next shoe of the left array is bigger or equal to the next shoe of the right array. If it does, then it copies the shoe of the left array to the given array. If it doesn't, then it copies the next shoe of the right array into the given array. Once the loop finishes it adds to the given array all the remaining elements of either the first subarray or the second. As to get out of the first while only 1 array needs to be completely copied. The other one, in the worst case, will still have all its elements to be copied.

Quicksort

Quicksort is an algorithm similar to mergesort. But in this case we use an element of the array as a pivot, in my case the leftmost element. After taking this element, all the other elements are compared, if they are greater than the pivot they are put to the right, if they are smaller, then to the left. And that is done recursively to sort the whole array. This algorithm has an average cost of O(N*log(N)), in the best case it remains the same O(N*log(N)). But in the worst case its cost is O(N^2). Its advantages is that it's pretty fast on average and it requires a small amount of memory. In contrast, if the pivot is chosen poorly, it can lead to the worst case, taking much time to sort the array.



To implement it 3 functions have been created: quickSort, partition and swap. Swap is a function created to swap 2 given elements of the array, by copying the first one in a temporary element, then copying the second one in the first one and the temporary in the second one. Partition is a function that takes the last element as pivot and loops through the whole array placing the bigger elements to its left and the smaller to its right, updating the pivot according to the swaps it is making. To do so the previously explained function swap is used, at the end it returns the pivot updated after all the swaps. Finally quickSort is a recursive function that given an array a left index and a right index, while the left index is smaller than the right one it will call the function partition and then it will call itself recursively 2 times(1 recursive call passing left index as left index and passing pivot - 1 as right index, and the other passing pivot + 1 as left index and right index as the right index).

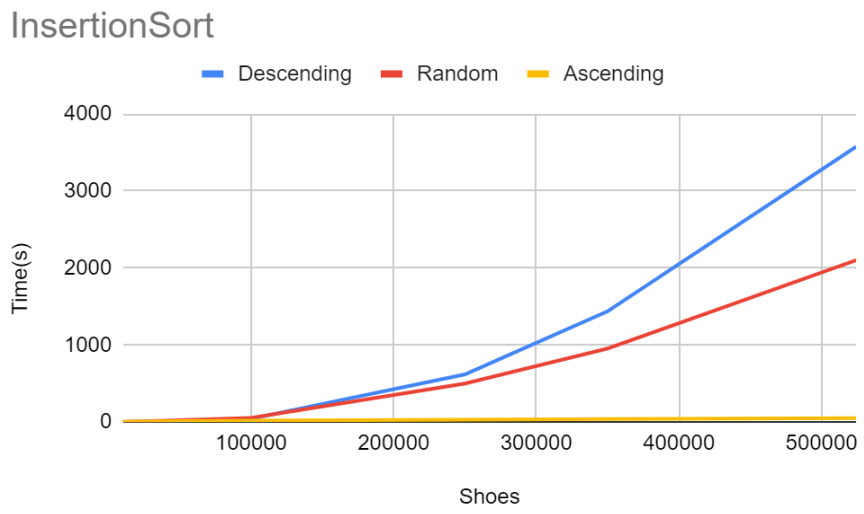
Result analysis:

I measured the average time that it took for my program to execute each type of algorithm for each of the required datasets for different numbers of shoes. Each measurement was made 3 times and then the average was taken. This is what I observed:

Insertion sort

After taking three measurements for each dataset and for different ranges of shoes, this were the results obtained:

Datasets:	Ascending	Descending	Random
Shoes	AvgTime(ms)	AvgTime(s)	AvgTime(s)
10000	4,333333333	0,5483333333	0,2343333333
50000	13,33333333	25,73533333	12,99133333
100000	19	53,09533333	41,05833333
250000	30	499,5836667	618,4326667
350000	38	953,7906667	1438,401333
524288	50,66666667	2.103	3575,663

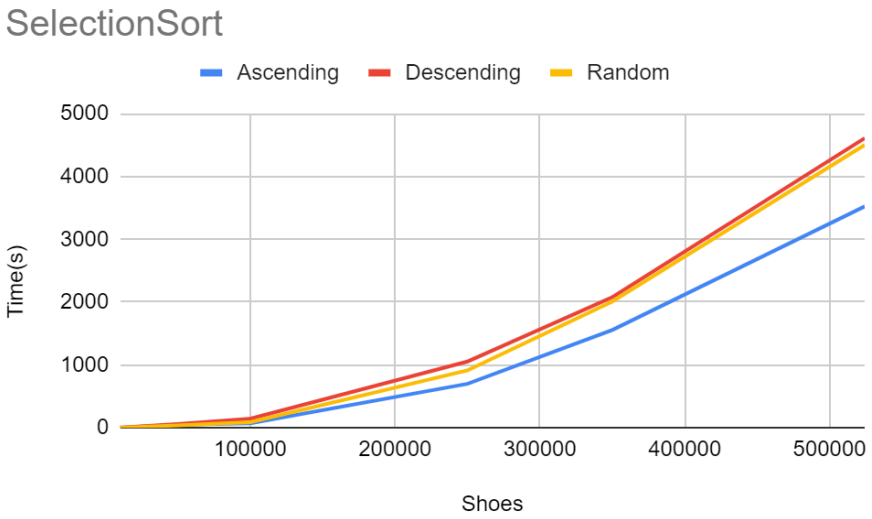


As it can bee seen in the table or in the graphic, the ascending dataset did not take more that 50ms to go through more than 500K shoes, whereas for the other one it took more than 30 minutes. This happened, as explained before, because insertion sort has a cost O(N) in the best case, as it never enters the while loop inside the first for loop because the element always comes before. So the algorithm just goes through all the elements 1 time and does nothing. Also it can be observed how it works well with small amounts of data, in all datasets it sorts the data pretty fast until there are 150K elements or so, when the time it takes to sort them starts growing more exponentially. Moreover, it can be seen how it takes more time to sort the descending dataset than the random, as it is the worst case. So, even though they have more or less the same growth relation, the descending grows at a higher ratio.

Selection sort

After taking three measurements for each dataset and for different ranges of shoes, this were the results obtained:

Datasets:	Ascending	Descending	Random
Shoes	AvgTime(s)	AvgTime(s)	AvgTime(s)
10000	0,638	1,215333333	0,4783333333
50000	47,19866667	58,07366667	36,26
100000	71,684	143,4296667	88,874
250000	699,6396667	1053,877	913,4596667
350000	1555,089667	2078,900667	2009,979667
524288	3524,027	4610,180333	4503,172

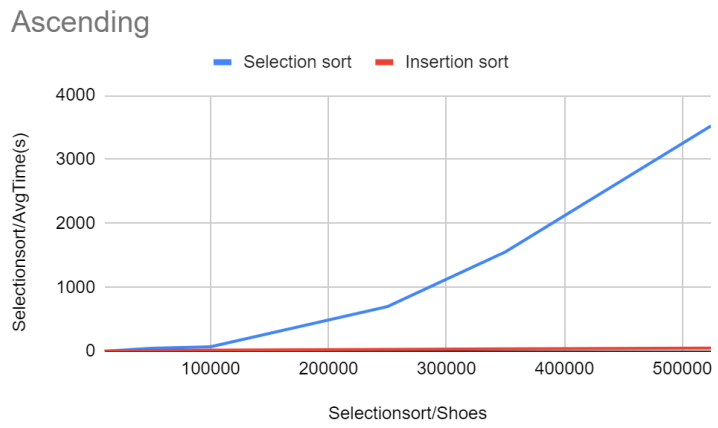


As it can be seen, in contrast to insertion sort, and as explained before, the selection sort in ascending took less than the others, but almost the same. Because, as said before, the cost is always O(N^2). As even though the dataset is already ordered, the algorithm will traverse all the next elements until the end for each element in the array, the only thing is that it will not swap any element as they are already ordered. Another thing to notice, is that, as with insertion, it works fine with small datasets, as can be seen how until the 150K or so, the algorithm works relatively faster in contrast to how it works past the 150K elements. Another thing to notice is how the selection is almost the same in both random and descending, being a little bit faster than the random one. Whereas insertion was pretty much faster in the random than in the descending. That's because insertion benefits a lot more from the dataset being ordered, as it will not even iterate when an element is already ordered in contrast to its next element.

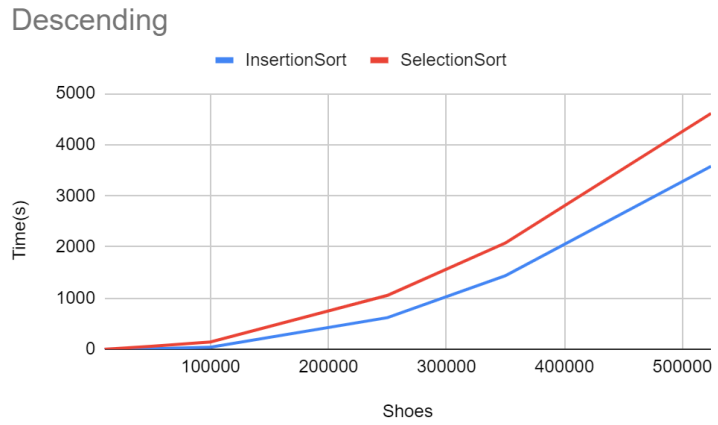
But selection no, it will always iterate until the end even if it is already ordered. The only thing it may benefit from an array being already or partially ordered is that it will not waste time swapping elements and will not waste time changing the smallest index. But those things are done pretty fast, so it does not affect much overall.

Insertion & selection sort

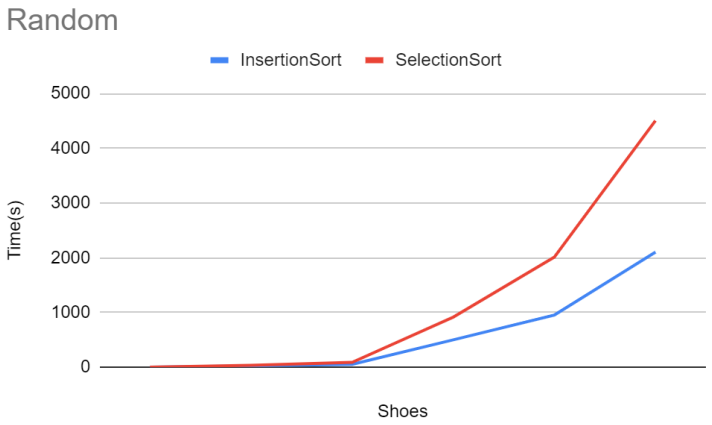
If I compare both algorithms for each dataset we will get the next charts:



With the ascending we can see what I already explained about insertion sort benefiting a lot from ordered datasets, as it will not iterate not even 1 time, than selection sort, which always iterates until the end no matter what.



In the descending one, where both should have the exact same cost as it is the worst case, I can appreciate how the insertion algorithm is still faster, even though both have the same number of iterations, insertion sort is still faster, around 25% faster. That is due to insertion sort being implemented with a for and a while inside, while selection sort is implemented with 2 for loops. The thing here is that insertion sort makes the strcmp check as the while condition. Whereas the selection sort has an if where it makes the strcmp check. Although this might seem meaningless, it makes insertion sort to just check the strcmp in each iteration while the selection sort for each iteration checks if the for loop is still in range and then it checks the if with the strcmp. So selection sort is checking 2 things while insertion 1.



In the random dataset it can be observed how insertion sort is twice as fast as selection sort. Apart from what I have just explained that makes insertion sort a little bit faster always, here rather than that, what is most affecting the insertion sort to be way faster is that the dataset has some parts in order, some not. Therefore, for those parts the selection sort will iterate until the end while the insertion sort will not iterate. Therefore, it is skipping a lot of iterations.

Quicksort & Mergesort

After taking three measurements for each different range of shoes, this were the results obtained:

Shoes	Quicksort(ms)	Mergesort(ms)
10000	5,666666667	10
50000	21,33333333	41
100000	42,33333333	55,33333333
250000	86	142,3333333
350000	138,3333333	228,6666667
524288	199	285

It can be appreciated in their growth that both have an average cost of O(Nlog(N)), but quick sort grows at a smaller ratio than merge sort, in other words, it's faster. That basically means that each recursive call of the quicksort is faster than a recursive call of the merge sort. That is because the merge sort copies both halves to 2 temporary arrays and then it compares those 2 arrays and based on the result it stores each element in the final array. The procedure of loading all the data into 2 temporary arrays takes some extra time that quicksort does not take. Moreover, merge sort wastes a lot of memory resources. So, overall I would say quicksort is better than mergesort, as it has proven to be faster and wasting far less memory resources. Moreover, quicksort can be optimized in various ways to perform better, such as choosing a good pivot to avoid the worst-case scenarios or part of them. Even

though, for how I've seen merge sort works as well as for how it grows in the graphic, I believe merge sort to be a better algorithm to sort even more massive amounts of data. It can be seen how as it grows, merge sort is reducing the gap in time between quicksort and him. Moreover, it can be parallelized to take advantage of multiple processors, which would make merge sort even faster.

Observed Problems

The only problem I found, if it may be called a problem, is the vast amount of time it took me to run insertion sort and selection sort, as the data grew more and more the time grew even more. Which caused me to have to wait a lot of time to collect all the data. I could have run both algorithms until the 250K or 100K shoes and make more subsets inside this range. But as I already had 528K shoes I preferred to not stop at 100K/250K to see how it continues behaving as the amount of shoes grows more.

Total dedication

Making the report and developing and testing the code I have taken around 34 hours approximately. Which can be divided in the following sections:

Research: Between 1 and 2 hours.
Design & Implementation: Around 2 and a half hours
Result analysis: 25,5 hours
Report: 5 hours

Note that result analysis took me 25,5 hours due to the massive amounts of times it takes insertion and selection sort to sort datasets of more than 100K elements. Moreover, to run them 3 times per each subset they had to be run for 3 different datasets. So those 25,5 hours are hours that took me to collect all the data, but were mostly hours in which the laptop was running the algorithms while I took advantage to do other things.

Conclusion

This project has helped me settle the knowledge about sorting algorithms. It has made me realize the importance of sorting algorithms and choosing wisely. As, even though all algorithms can be used in all the sortings of this project, all algorithms have its unique traits that make them quite faster or slower for one problem or another and more easy to implement for one project or another. Therefore, analyzing the problem and thinking about which algorithm to use is the most important part, as it will not only save you time while implementing it, but it will increase the performance of the sorting incredibly. The difference in time you can get from choosing the appropriate algorithm in contrast with using a random algorithm is astonishing.

Moreover, I've been able to see the difference between the algorithms and where to use each one. For example, insertion sort has proven to be an easy algorithm to understand and implement in cases where you have small amounts of data and don't want to waste too much time in designing the best sorting algorithm as it is not going to be used a lot or is not critical. Quicksort and mergesort on the other hand have proved to be quite versatile and fast, as they can be used in most of the problems and don't have that many restrictions.

Bibliography

Source I consulted to develop the project:

Pol Muñoz: *S1_Notes*. September 2022

- S1 Notes.pdf

Geeksforgeeks: Insertion Sort - Data Structure and Algorithm Tutorials. 31th may 2023

- https://www.geeksforgeeks.org/insertion-sort/

Geeksforgeeks: Selection Sort - Data Structure and Algorithm Tutorials. 26th May 2023

- https://www.geeksforgeeks.org/selection-sort/

Geeksforgeeks: Quicksort - Data Structure and Algorithm Tutorials. 16th october 2023

- https://www.geeksforgeeks.org/quick-sort/

Geeksforgeeks: Mergesort - Data Structure and Algorithm Tutorials. 06th July 2023

- https://www.geeksforgeeks.org/merge-sort/