11253444

Abstract

This report contains the designed algorithms and aims to analyse each algorithm by its CPU time, number of comparisons, swaps and operations.

algorithm & data structures

Coursework 2



Contents

[introduction 4](#_Toc131513248)

[Bubble sort 4](#_Toc131513249)

[Pseudocode 4](#_Toc131513250)

[Code 5](#_Toc131513251)

[Output 6](#_Toc131513252)

[CPU time 7](#_Toc131513253)

[Number of swaps 7](#_Toc131513254)

[Analysis 8](#_Toc131513255)

[Graph 9](#_Toc131513256)

[Insertion sort 9](#_Toc131513257)

[Pseudocode 9](#_Toc131513258)

[Code 10](#_Toc131513259)

[Output 11](#_Toc131513260)

[CPU time 11](#_Toc131513261)

[Number of swaps 12](#_Toc131513262)

[Analysis 12](#_Toc131513263)

[Graph 13](#_Toc131513264)

[Merge sort 13](#_Toc131513265)

[Pseudocode 14](#_Toc131513266)

[Code 15](#_Toc131513267)

[Output 17](#_Toc131513268)

[CPU time 18](#_Toc131513269)

[Number of Comparison 18](#_Toc131513270)

[Analysis 18](#_Toc131513271)

[Graph 19](#_Toc131513272)

[Selection sort 19](#_Toc131513273)

[Pseudocode 20](#_Toc131513274)

[Code 20](#_Toc131513275)

[Output 22](#_Toc131513276)

[CPU time 22](#_Toc131513277)

[Number of swaps 23](#_Toc131513278)

[Analysis 23](#_Toc131513279)

[Graph 24](#_Toc131513280)

[Quick sort 24](#_Toc131513281)

[Pseudocode 24](#_Toc131513282)

[Code 25](#_Toc131513283)

[Output 27](#_Toc131513284)

[CPU time 27](#_Toc131513285)

[Number of swaps 28](#_Toc131513286)

[Analysis 28](#_Toc131513287)

[Graph 29](#_Toc131513288)

[Binary Search 29](#_Toc131513289)

[Pseudocode 29](#_Toc131513290)

[Code 30](#_Toc131513291)

[Output 32](#_Toc131513292)

[CPU time 32](#_Toc131513293)

[Analysis 33](#_Toc131513294)

[Graph 34](#_Toc131513295)

[Linear search 34](#_Toc131513296)

[Pseudocode 34](#_Toc131513297)

[Code 35](#_Toc131513298)

[Output 36](#_Toc131513299)

[CPU time 37](#_Toc131513300)

[Analysis 37](#_Toc131513301)

[Graph 38](#_Toc131513302)

[CPU Time of all algorithms 39](#_Toc131513303)

[Average Case 40](#_Toc131513304)

[Best Case 40](#_Toc131513305)

[Worst Case 41](#_Toc131513306)

[Binary vs. Linear 41](#_Toc131513307)

[Average case 41](#_Toc131513308)

[Best case 42](#_Toc131513309)

[Worst case 42](#_Toc131513310)

[Reference 43](#_Toc131513311)

[Appendix 45](#_Toc131513312)

[Pseudocode walkthrough 45](#_Toc131513313)

[Sum of first(n-1) positive integers 46](#_Toc131513314)

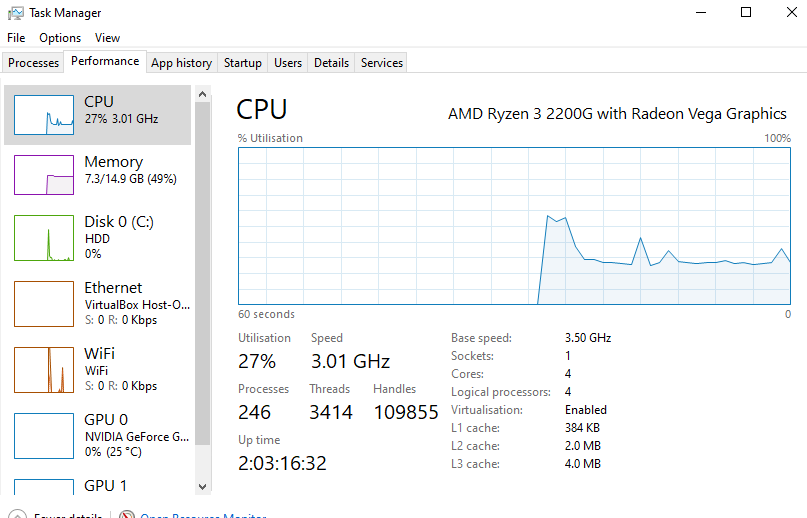
# introduction

This report contains the designed algorithms using the object-oriented programming language, Python. In addition, it aims to analyse each algorithm by its CPU time, number of comparisons, swaps and operations using big O notation.

To provide consistency in this report, I have used my personal computer in running all the algorithms. This will standardise the results based on the limits of my machine and limit any hardware or software biases.

In addition, asymptotic analysis will also be provided to measure the efficiency of the algorithms as this is not machine-specific and does not require the algorithm to be executed (Types of asymptotic notations, n.d.). Also, pseudocode explanations will be provided in the appendix.

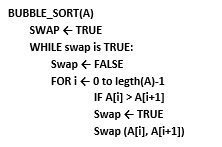
Below is my computers hardware setup. AMD Ryzen 3 2200G uses a processor with 4 cores with 4 threads. It has a 3.5 GHz base clock (AMD Ryzen, 2022).



## Bubble sort

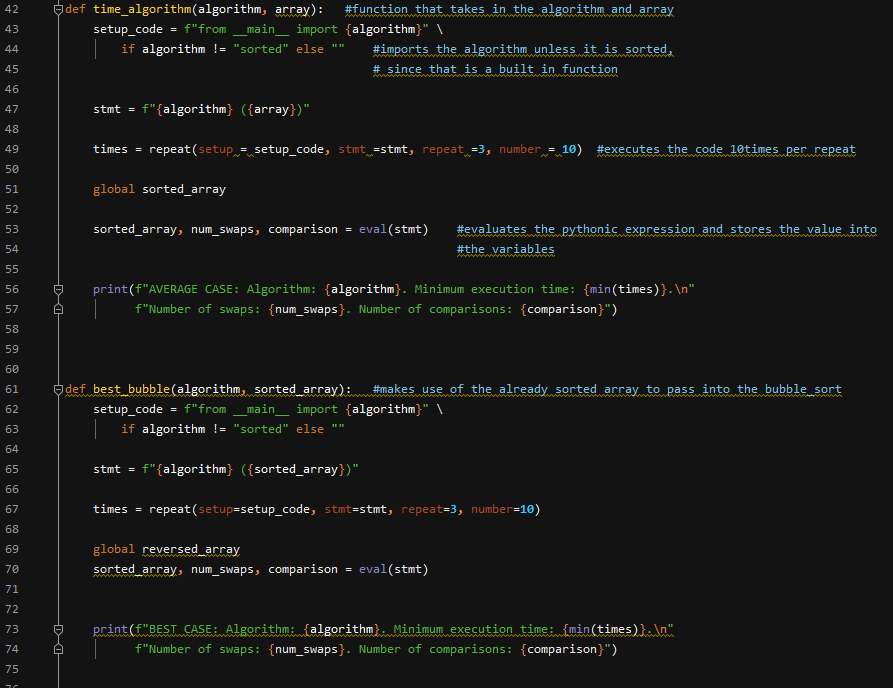
Bubble sort is a simple algorithm that makes use of 2 nested loops. Although it is considered inefficient, it is generally taught in computer science courses to build intuition on sorting (Moore & Khim, n.d.).

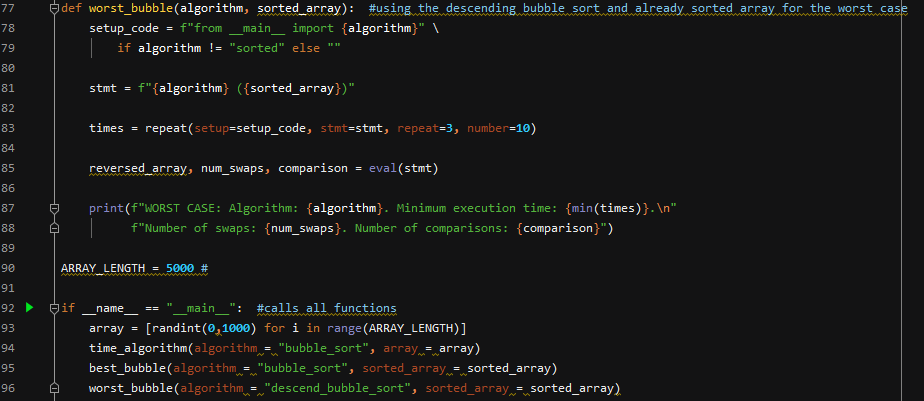
### Pseudocode



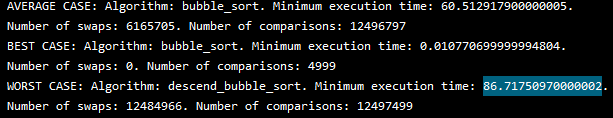
### Code





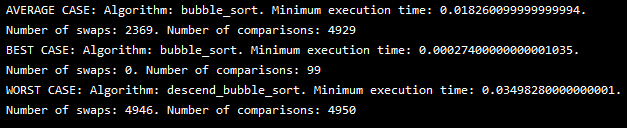


### Output

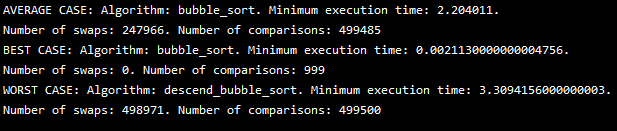


### CPU time

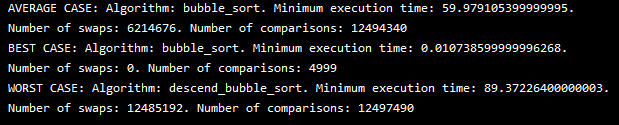
#### Range of 100



#### Range of 1000



#### Range of 5000



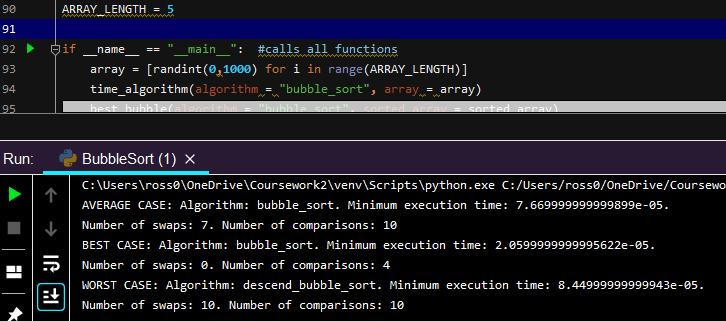
### Number of swaps

To determine the number of swaps done, I initialised the variable swap as 0 and increment it by 1 every time the condition **if array[j]** > **array[j+1]** is met. The number of swaps will be relative to how unsorted the array is.

Moreover, as there are n number of elements in an array and it takes n-1 swaps to move the elements into its final index, it will equate to n(n-1)/2. This is due to the number of swaps being the sum of 1 to n-1.

### Analysis

To determine the number of comparisons in the bubble sort, I initialised a variable called comparison. This variable will be incremented by 1 in every iteration of the inner loop.



In bubble sort, the algorithm will perform O(n) comparisons as an array contains n number of elements or O(n) number of elements. Since bubble sort performs O(n) operations on O(n) number of elements, this will lead to O(n2) total running time (Moore & Khim, n.d.). Discussions of the sum of the first (n-1) positive integers will be at the appendix.

In the worst-case scenario, the smallest element would be located at the end of the list and will have to be swapped each time to reach the first element of the list. It will only reach the front of the list when all n iterations have finished. As the outer loop runs O(n) times and the inner loop runs O(n) times, it results in O(n\*n) = O(n2).

The best-case scenario would be for the array to be already sorted when it is assigned to the bubble sort algorithm. However, it first has to be modified so that it terminated after one pass without doing any swaps. This will result in O(n) times.

The average case scenario would require (n/2) passes and O(n) comparisons for each pass. This will result in O(n/2 \* n) = O(n2).

Even though, bubble sort is easy to implement it is deemed inefficient due to the average and worst-case running time being O(n2).

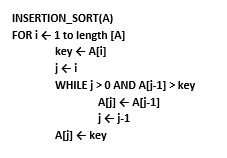
It can be noted that the worst case and average case exponentially increase in running time as the number of arrays are increased. Whereas the best-case time remained low, as there were no elements to be swapped.

### Graph

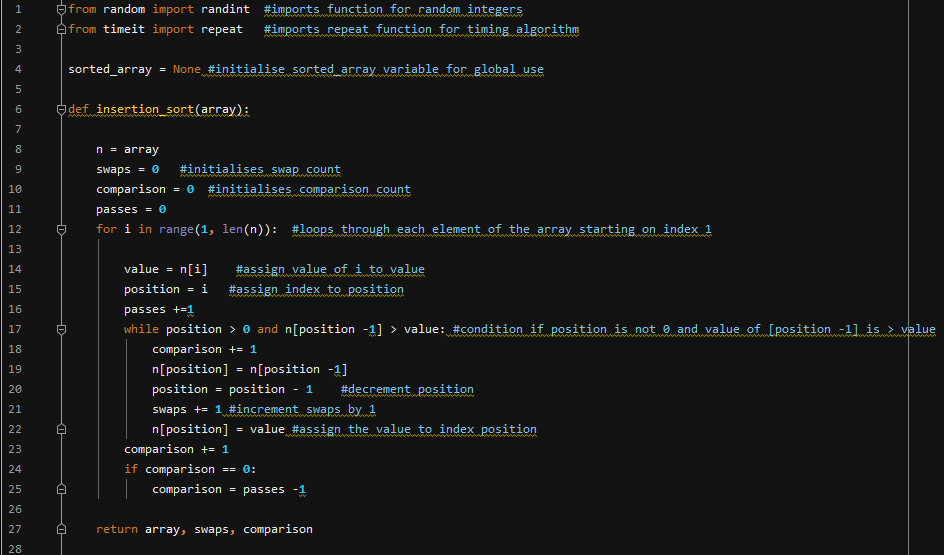
## Insertion sort

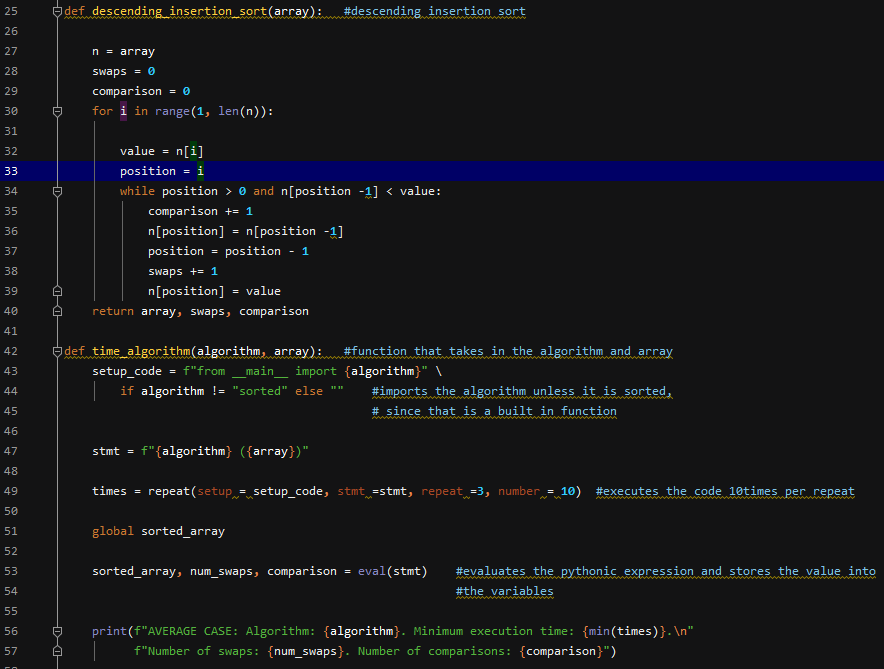
Insertion sort is a straight forward algorithm that is seen as more efficient than bubble sort algorithm. It is based in the concept of playing cards where it is sorted by order. It is an in-place and stable algorithm that works best in few and or nearly sorted elements. Insertion sort is said to be an in-place algorithm as it does not need to make another copy of the array to modify it. In addition, it is said to be stable as it retains the order of equal objects in the initial array (Insertion sort, n.d.).

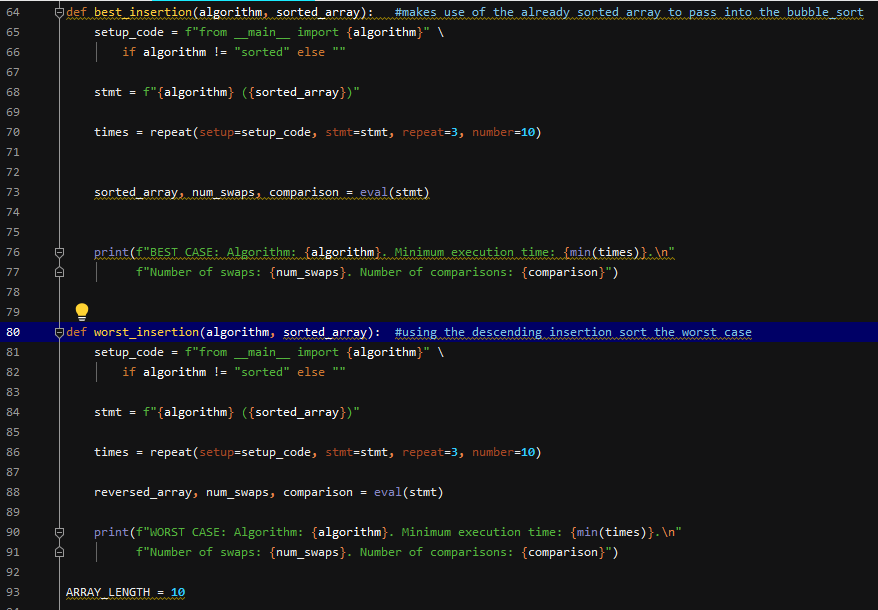
### Pseudocode

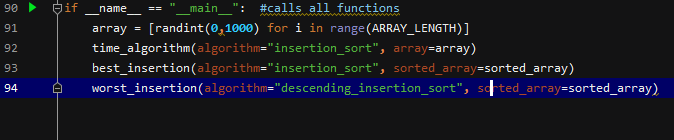


### Code

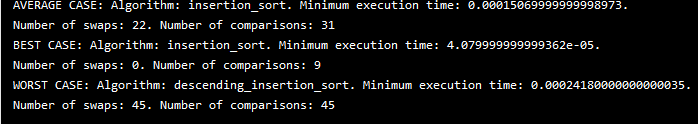






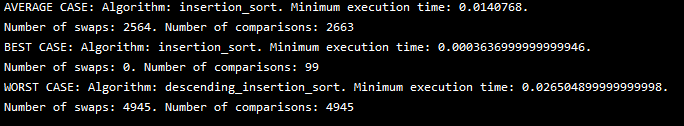


### Output

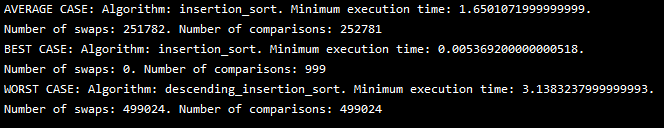


### CPU time

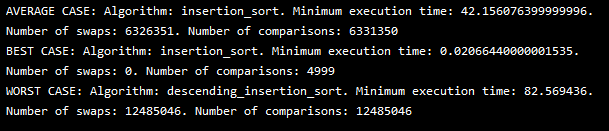
#### Range of 100



#### Range of 1000



#### Range of 5000



### Number of swaps

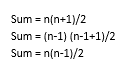
In determining the swaps in insertion sort, I created a variable named passes which would count the number of loops. I also made a variable named comparison and placed it inside the while loop. This would count all the comparisons in the average and worst scenario and give out the comparisons in the best case where it would depend on the number of passes.

### Analysis

It can be noted that the number of comparisons in an insertion sort is dependent on the N number of items and the order of which the items are originally sorted. Insertion sort require N-1 passes in a sorted list. This is then used in the unsorted list as it starts with N-1 items as the first item is already in the sorted list. In addition, there is no general formula for all cases. However, there are separate formulas for the average, best and worst case.

The best-case scenario for the insertion sort would be to have an already sorted list. The last item in the sorted list would be less than or equal to the first item in the unsorted list. This will lead to 1 comparison on each pass which results to N-1 comparison which is O(n) or linear in time complexity. Compared to selection sort, insertion sort is a faster algorithm if the number of comparisons is smaller (Insertion sort, n.d.).

In the worst-case, the order of the list would be in reverse. In a list with N number of elements, the comparison would be 1+2+3+…+(N-1). Using the formula for sum of natural integers:



A list with 5 items would result in 10 comparisons. Therefore, the maximum number of comparisons would be n(n-1)/2 or O(n2) in time complexity.

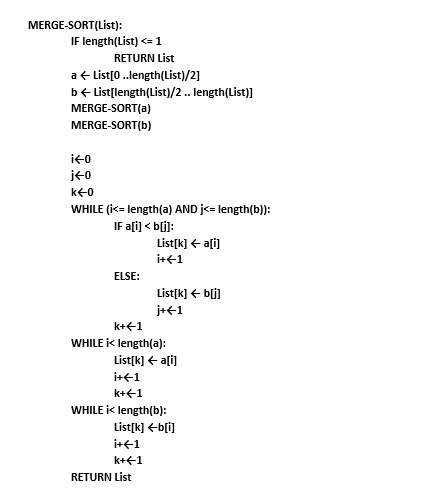
In the average case, the worse cases and best cases will cancel each other out as there is equal probability for each to occur. In this list the numbers are jumbled. This will lead to ½ of the sorted list being examined (Insertion sort, n.d.). This is then multiplied to: n(n-1)/2 which will yield n(n-1)/4. Therefore, the average case would be n(n-1)/4 which is O(n2) in time complexity.

### Graph

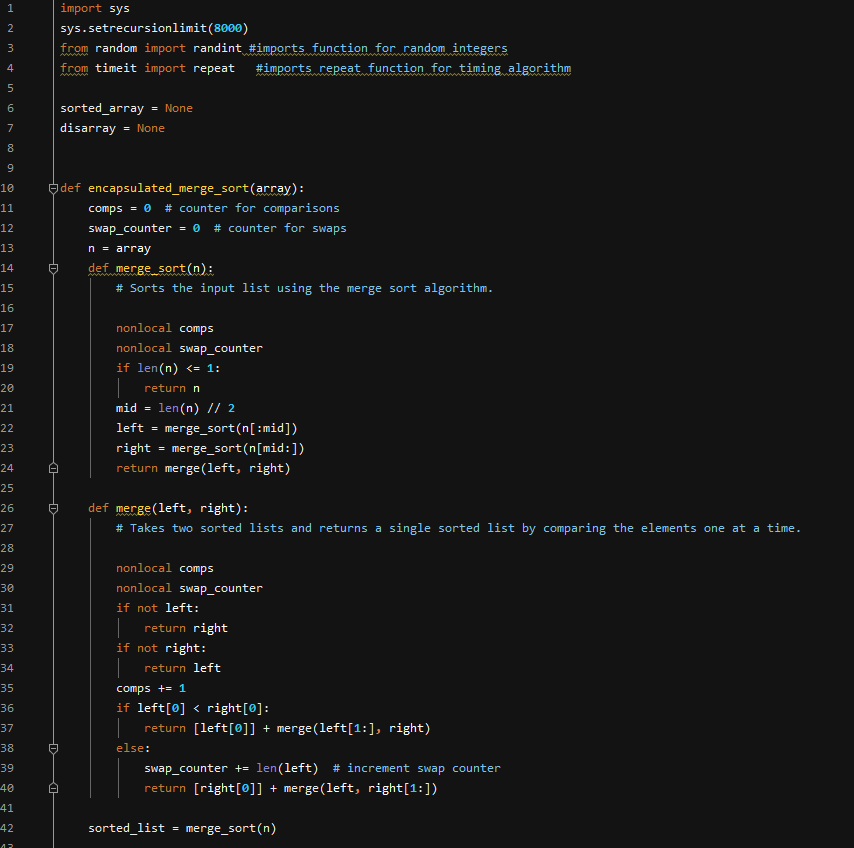
## Merge sort

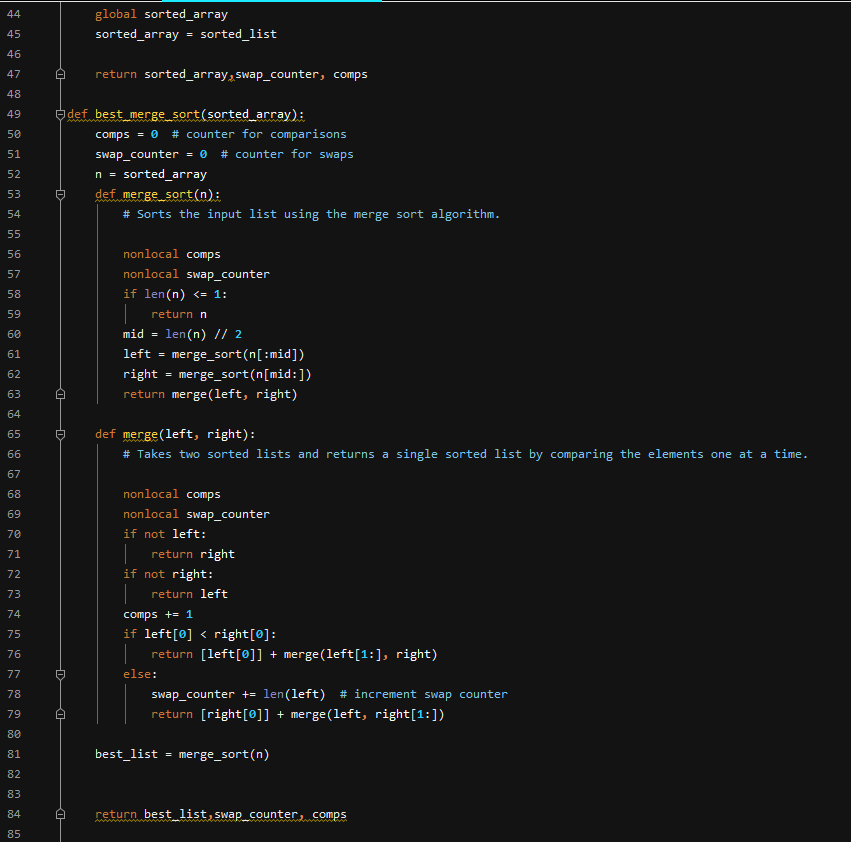
Merge sort is an algorithm that works in the concept of divide and conquer. This involves dividing the problem into smaller sub-problems. After each one is solved, they are amalgamated into a final solution (Janghu, n.d.). It has its advantages such as it is seen as a faster and more efficient sorting algorithm as well as more consistent in running time. However, it takes up more memory since it is not an in-place sorting and is more complex to program (Sorting algorithms, n.d.).

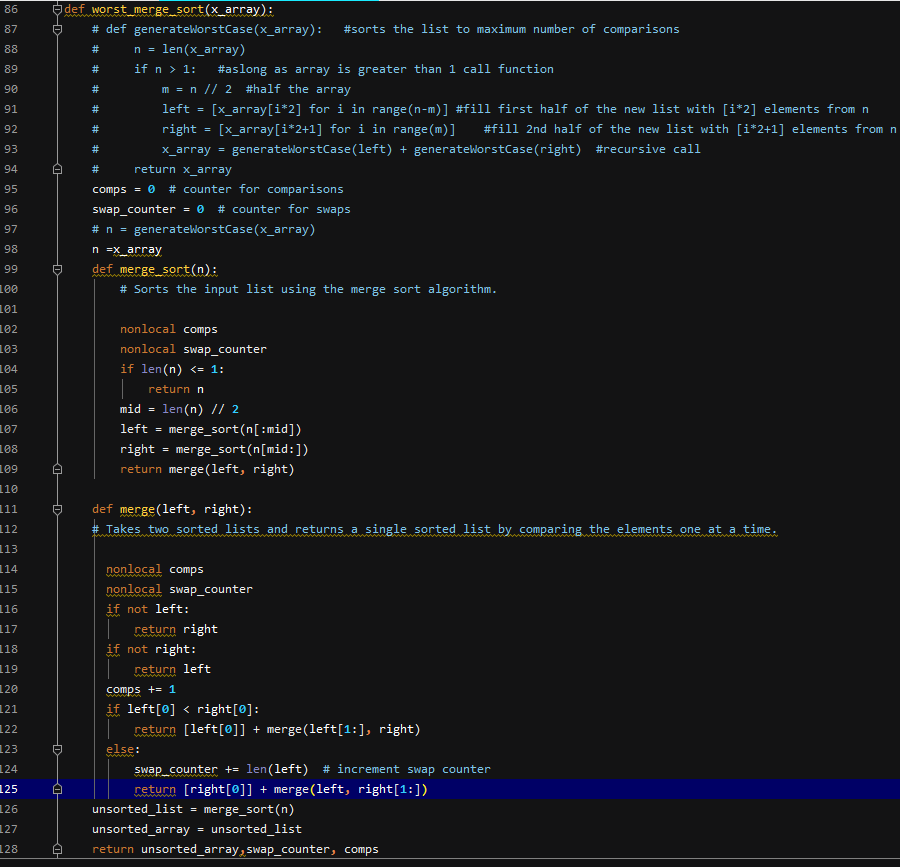
### Pseudocode

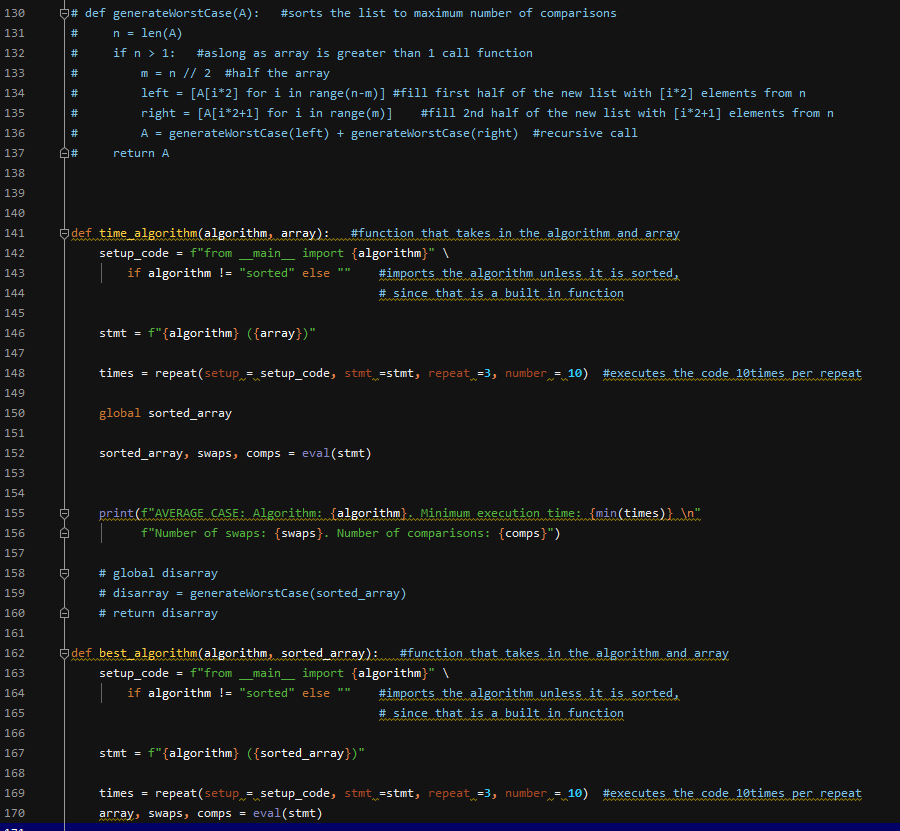
****

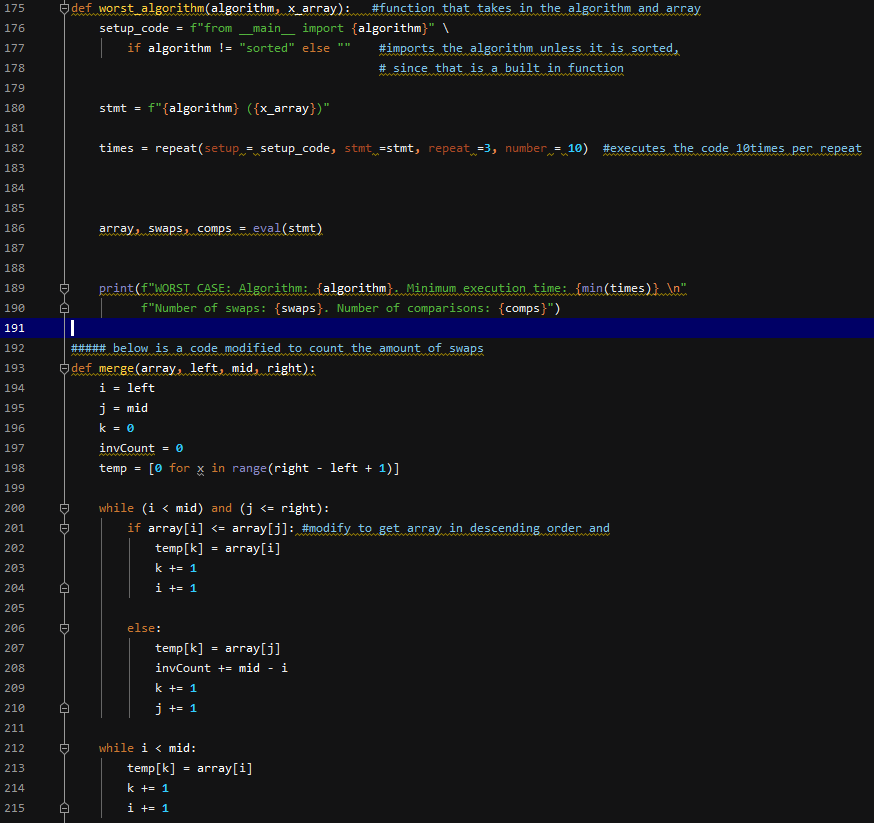
### Code

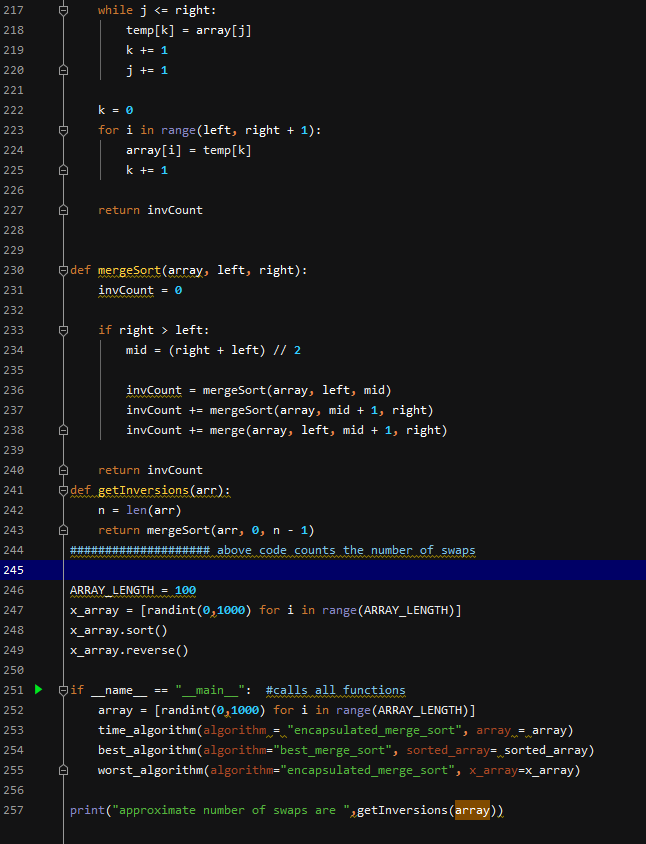




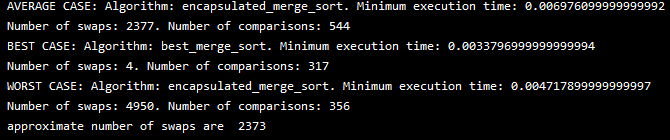






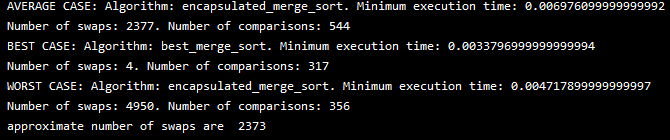


### Output

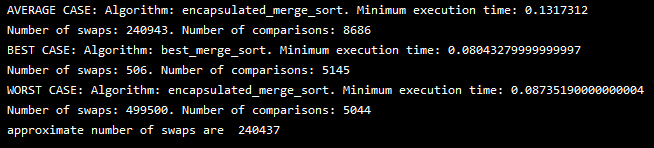


### CPU time

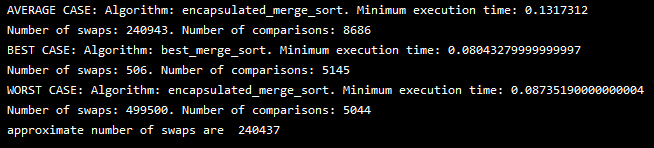
#### Range of 100



#### Range of 1000



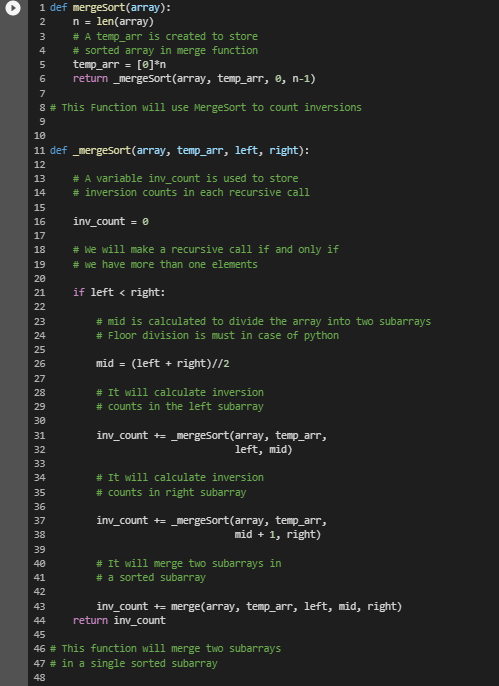
#### Range of 5000

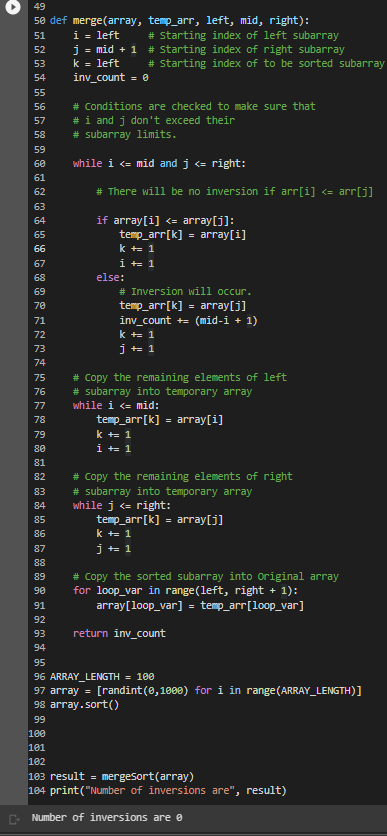


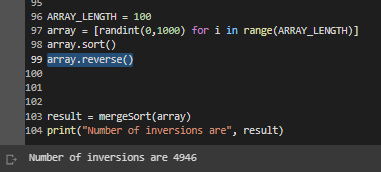
### Number of Comparison

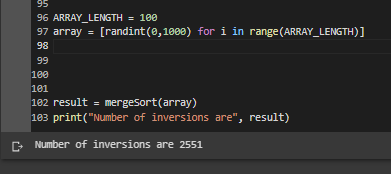
In the program comparisons count would be incremented when the program enters the while loop. It will increment as long as there are elements on both lists. The swap count is also triggered every time a swap is done. However, even in the best case where the list is already sorted it can be observed that the swap count still increments. This is due to the merging of the sub-lists which still compares and/or swaps the order of the lists.

In addition, there is an additional program that provides the number of swaps or inversions done in the algorithm. As in the images below, there are 0 swaps made in the best case, 4946 swaps in the worst case and 2551 swaps in the average. The worst-case output is significantly close to the 4950 output when using the formula (n\*(n-1)/2). This formula was utilised since a reversed array’s total number of inversion is the sum of integers from 1 to n-1.









### Analysis

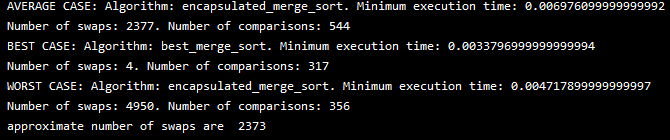
To count the number of comparisons in a merge sort, the formula Nlog2N is used. N is derived from the number of comparisons as merging of N item at most is N-1. Therefore, the number of comparisons will not exceed N. In addition, Log2N is derived from the number of iterations. As the number of lists are reduced by half in each iteration it results in N/2^k (Merge sort, n.d.). This is further rearranged to 2^k = N. Therefore, k = Log2N. The number if merge comparison can then be computed by multiplying the number of comparisons needed to merge all pairs of lists with the number of merge processes that must be performed:



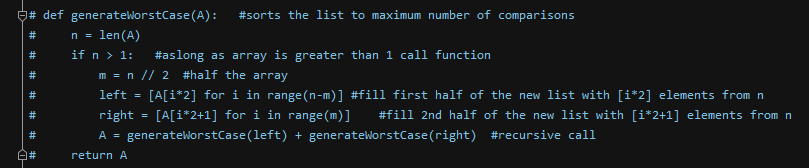
However, the formula is only an approximation as it assumes an even number of lists.

In addition, the time complexity for merge sort in the best, worst and average case scenario is O(n\*log n). This is because the algorithm will always divide the array in 2 halves and this will take linear time to merge (Pankaj, 2022). The average case scenario will have the elements jumbled while the best-case scenario would be to have an already sorted list. The worst-case scenario will have the reversed list.

Furthermore, how the merge sort was implemented in code can affect the output of swaps and comparisons significantly. An example is when sorting an array of 100 elements. Using the formula Nlog2N, the maximum number of comparisons in the worst-case should be 664. However, in my program output, the average-case is closer (544) compared to the worst-case which is only 356. A possible reason for this is that the algorithm takes advantage of the array since it is already sorted in descending order. This causes the program to do lesser comparisons.



Moreover, there is also this code to generate the worst-case scenario. This does not reverse the list in descending order but places the values in the indexes in a way that causes the maximum amount of coparisons.

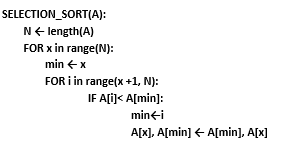


### Graph

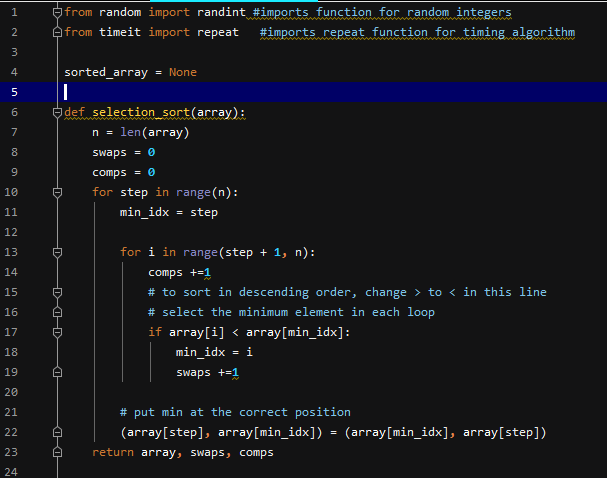
## Selection sort

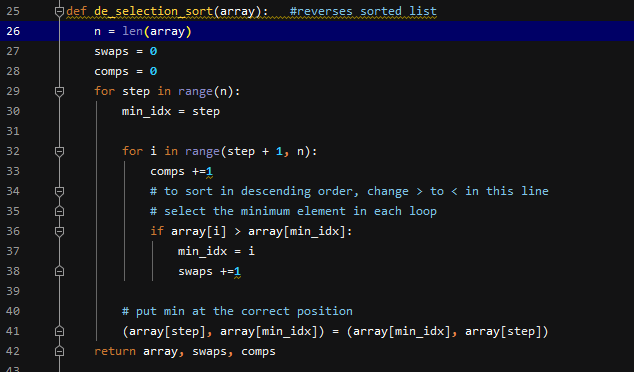
Selection sort is a sorting algorithm that loops through a list and swaps the elements according to its ordering/ It is advantageous to use this algorithm in occasions where the list is small. In addition, it does not require additional storage as it is an in-place sorting algorithm. However, its performance is very poor in large sets of lists and is affected by how the lists are pre ordered (Wandy, n.d.).

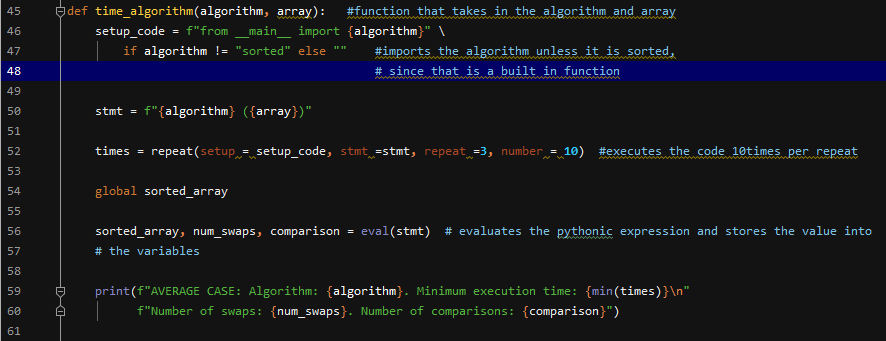
### Pseudocode

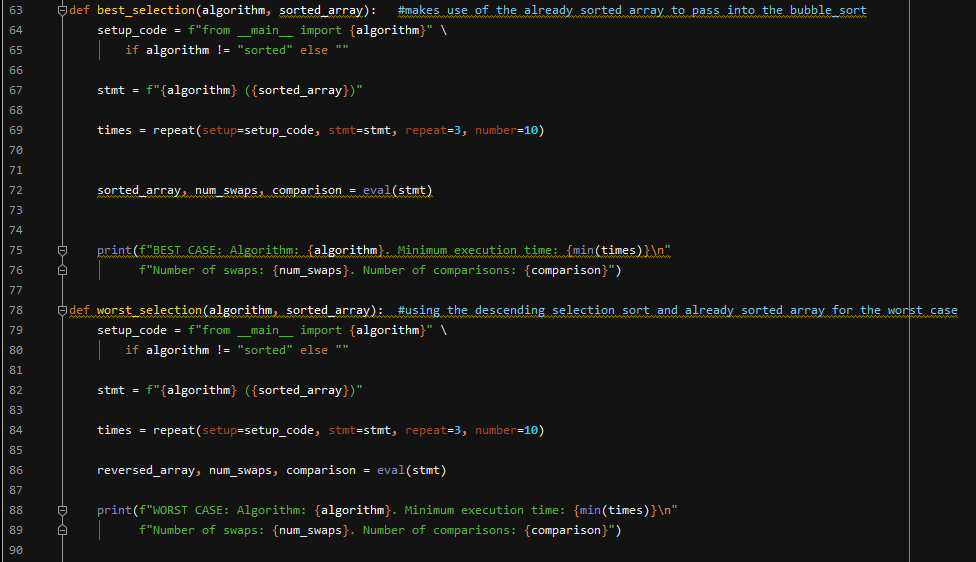
****

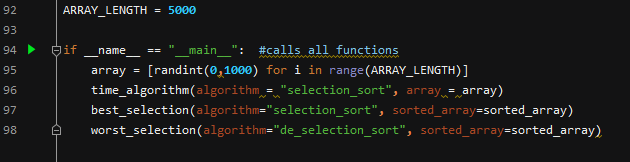
### Code



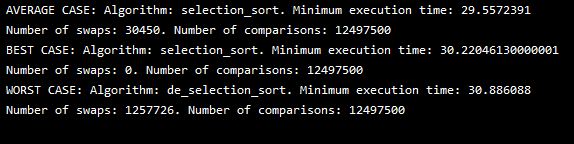






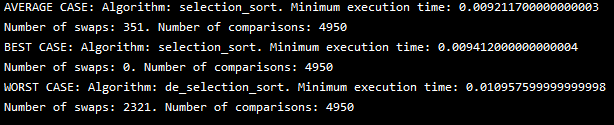


### Output

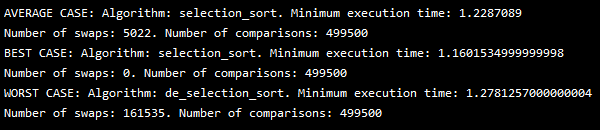


### CPU time

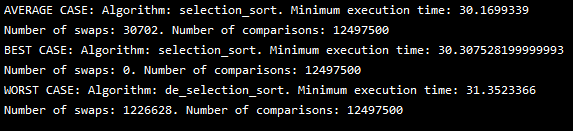
#### Range of 100



#### Range of 1000



#### Range of 5000



### Number of swaps

In the code, I have placed the swap increment inside the if statement. It will increment every time a swap has been made. Since this algorithm only swaps once every iteration of the inner loop, the number of swaps will be: N-1 or O(N). In the best-case scenario, the swap count will be 0.

### Analysis

Similar to the bubble sort algorithm, the selection sort uses the formula:



This counts the total number of comparisons in the algorithm. This is because it each element in the list must be compared to all remaining elements which results in (n-1)+(n-2)+…+1. And by using the sum of the first n-1 integers, the formula above is derived.

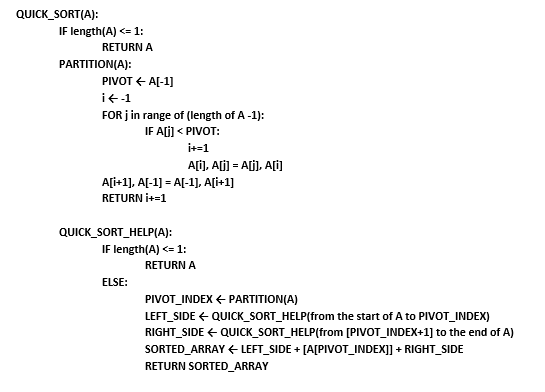
In the worst-case scenario where the list is reversed, the time complexity would be O(N2). Similarly, the average case is O(N2) as well. Furthermore, in the best-case scenario where the list is already sorted, the time complexity will be O(N2). This is because the number of comparisons has not changed even though there were no swaps performed.

### Graph

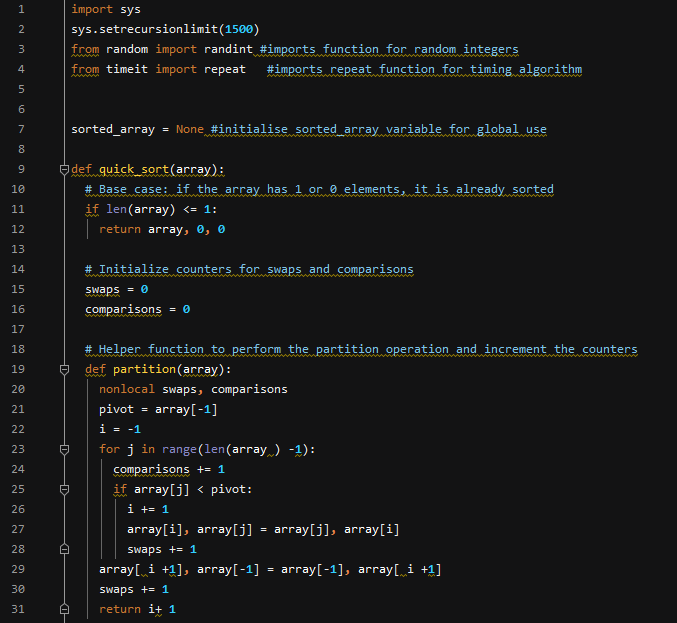
## Quick sort

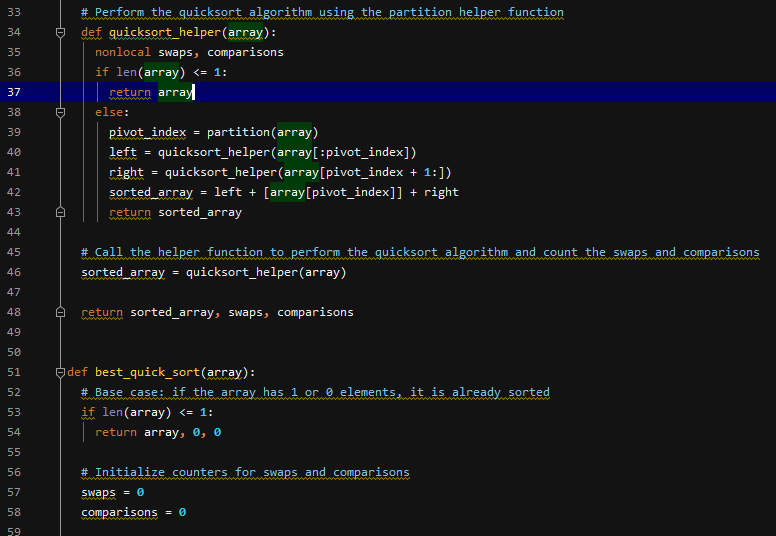
Quick sort is an efficient algorithm that uses the divide and conquer approach. It has the best time complexity compared to other sorting algorithms. However, due to its recursive element it is difficult to implement. Also, if the pivot elements are the largest, smallest or all are the same size, its performance is substantially affected (Patel, 2022).

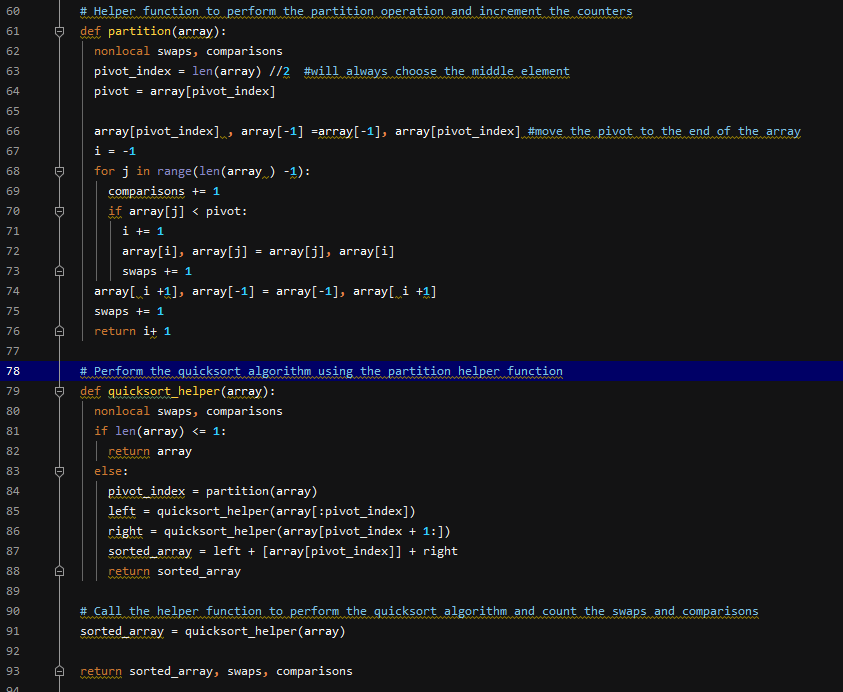
### Pseudocode

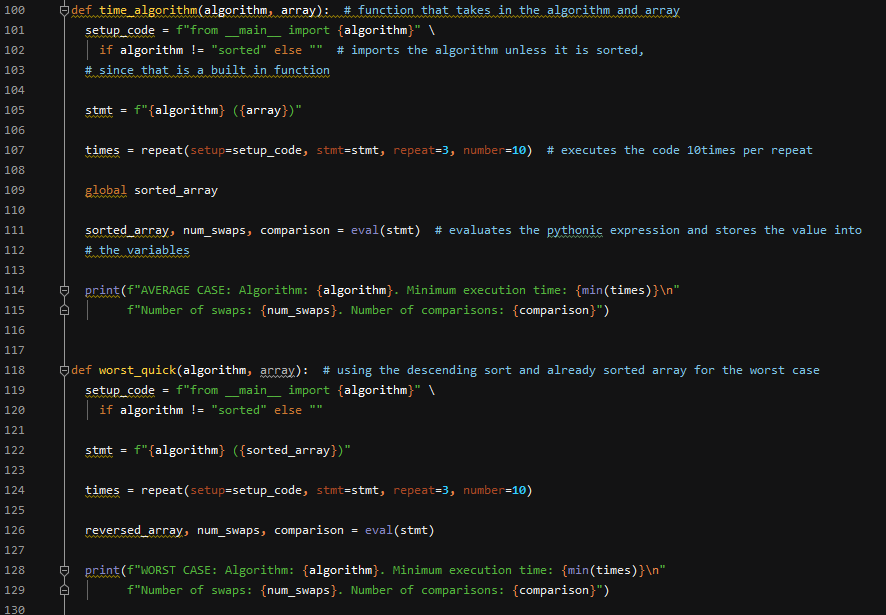


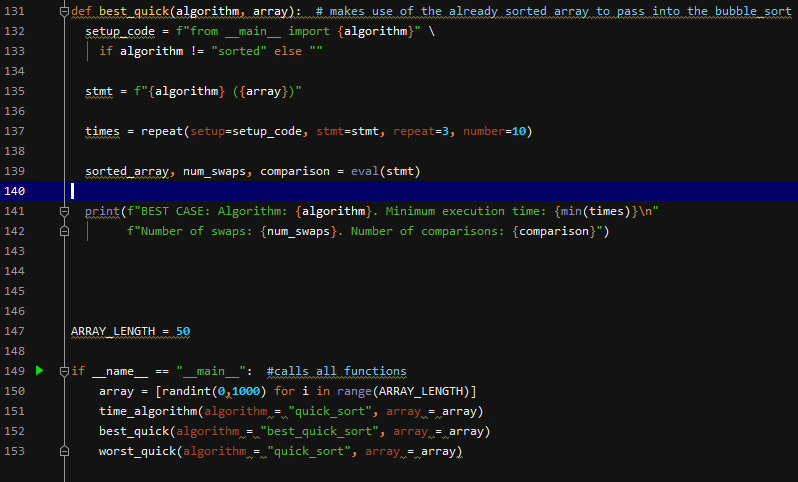
### Code



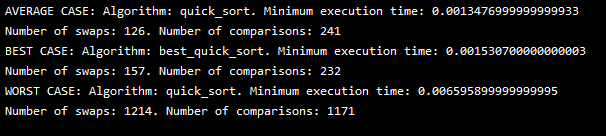






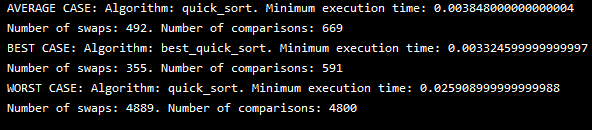


### Output

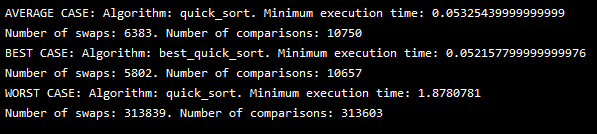


### CPU time

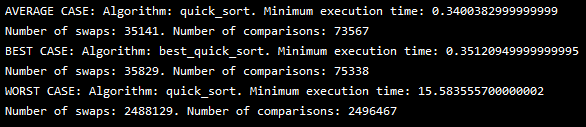
#### Range of 100



#### Range of 1000



#### Range of 5000



### Number of swaps

In the code above, the swaps count is incremented inside the partition function. There will be 2 increments. The first will be inside a for loop when it satisfies the condition and the other after it comes out of the for loop. The formula for the number of swaps will be:



n-1 because the pivot is not counted as part of the total number of swaps.

In the worst-case scenario where the pivot element is chosen as either the smallest or largest in the array and the given array is already sorted in reversed order, the total number of swaps will be O(n2). It is due to the recurring n-1 recursive calls (n-1 + n-2 + …+2+1), which results to n(n-1)/2.

However, in the best and average case, the number of swaps will be O(*n*log n). It is because, partitioning the array takes O(n) times and the array is being divided in to 2, recursively (N/2). This can be seen as N/2^k which is k = Log2N or O(logN).

### Analysis

To count the number of comparisons, I placed it within the for loop of the partition function. It can be seen in the output image that the worst case is significantly higher than the average and best case. This is due to the fact that the number of comparisons in average and best case is O(*n*LogN) and the O(n2) for the worst case (Quick sort algorithm, n.d.).

The average case occurs when the elements are jumbled and not in ascending or descending order. This results in the array being divided into 2 parts recursively, leading to O(*nLogN*).

Similar to the average case, the best-case scenario will have O(*n*LogN). It is because the case will occur if the pivot element is in the middle or near the middle element.

In the worst-case, the pivot element is either the largest or smallest element. This is similar to the worst-case scenario in swapping where the partition will have unbalanced sub arrays, one with 1 element (first element) and the other sub arrays with n-1 element or all other elements. This leads to O(n2).

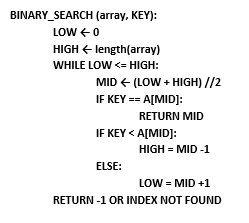
As seen below, the worst-case line starts to exponentially rise at 500 elements while the average and best case stay low.

### Graph

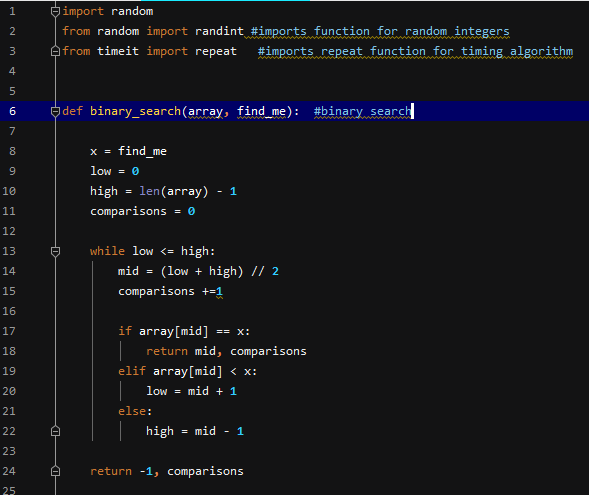
## Binary Search

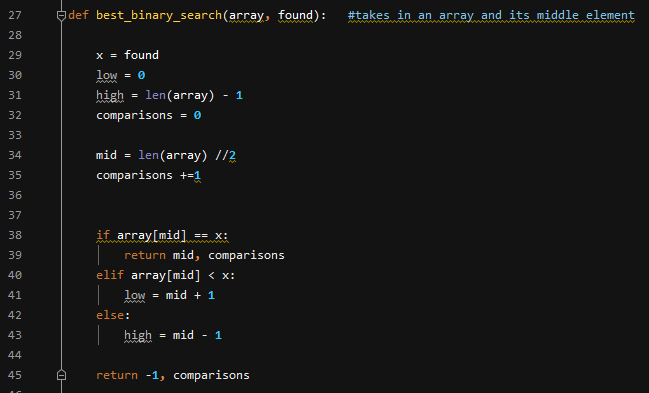
Binary search is a searching algorithm that makes use of the divide and conquer approach. It can be done recursively or iteratively. In the code that I have done, I made use of a while loop to iteratively divide the list until the key element is found.

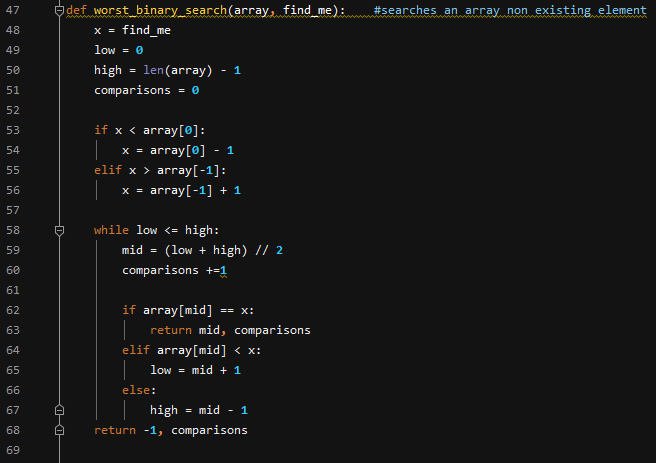
### Pseudocode

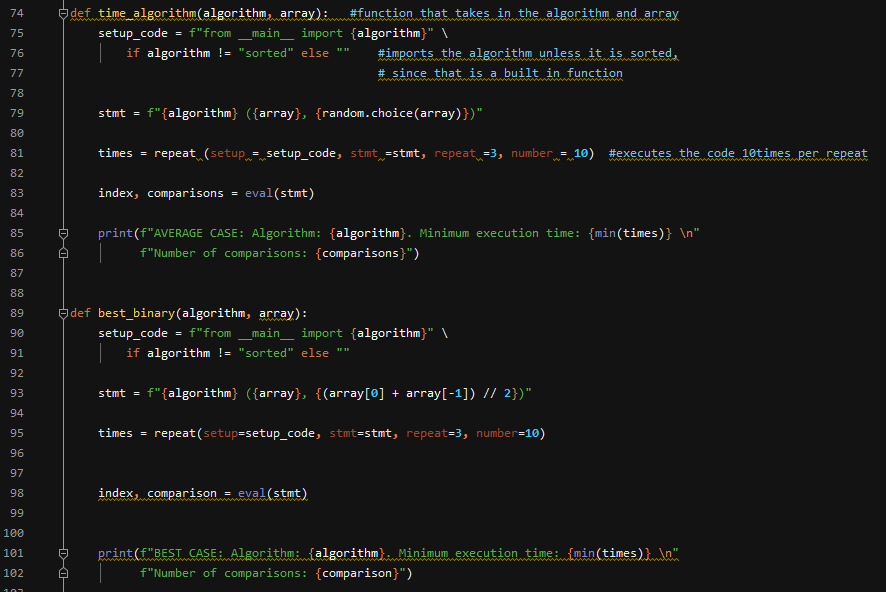


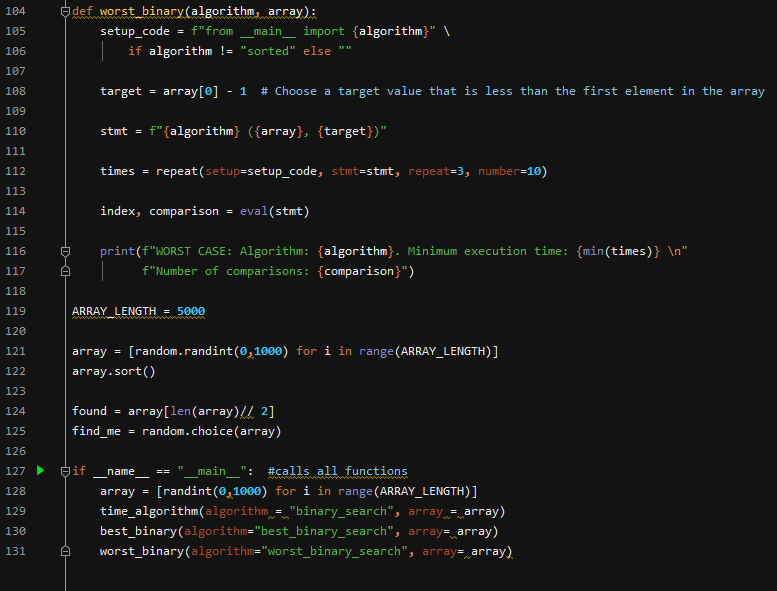
### Code



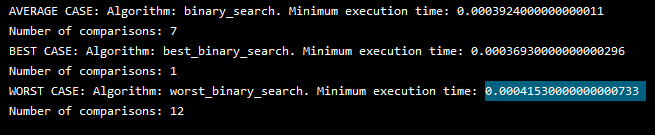






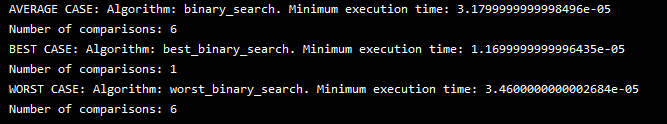


### Output

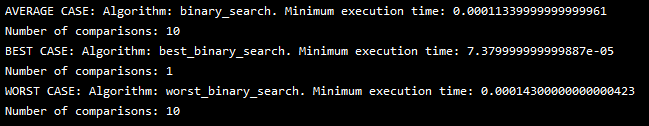


### CPU time

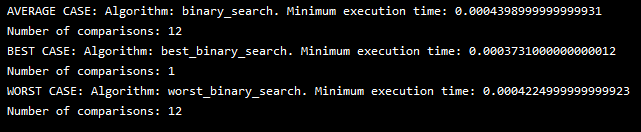
#### Range of 100



#### Range of 1000



#### Range of 5000



### Analysis

As this is a searching algorithm, there are no swaps to count. However, the number of comparisons can be solved depending on the case. For the best-case scenario where the key element can be found at the middle of a list, it will take 1 comparison with a time complexity of O(1) independent of the size of a list. The average case and worst-case time complexity is O(logN). The number of comparisons depend on the height of the tree. In addition, the worst-case scenario is when the key element is at the start or end of the list. This will ensure the maximum comparisons (Vaish, 2022). In an average case, it assumes that the key element is equally likely to found at any part of the list (4.4.1 Average Case Analysis of BST Operations, n.d.).

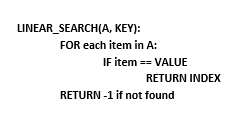
Binary search is faster than linear search when it comes to larger arrays. It is because the linear search increases linearly with the size of the array. Moreover, binary search is simpler to implement compared to other algorithms. However, it does require the array to be pre-sorted which can add an additional O(*n*LogN) time complexity for the sorting step (Binary search, 2023).

### Graph

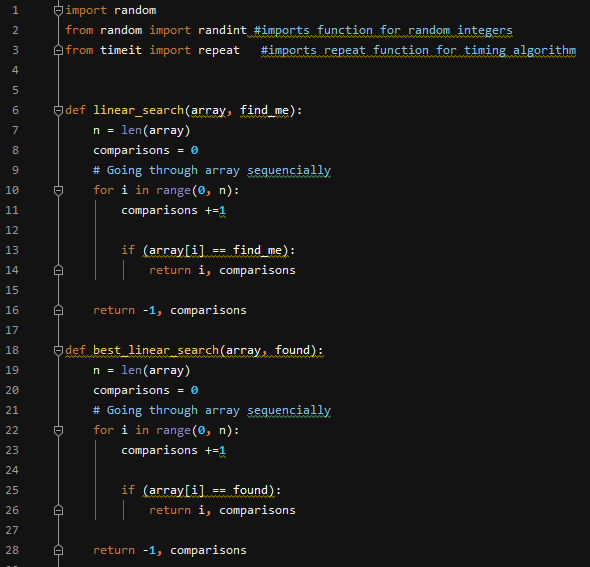
## Linear search

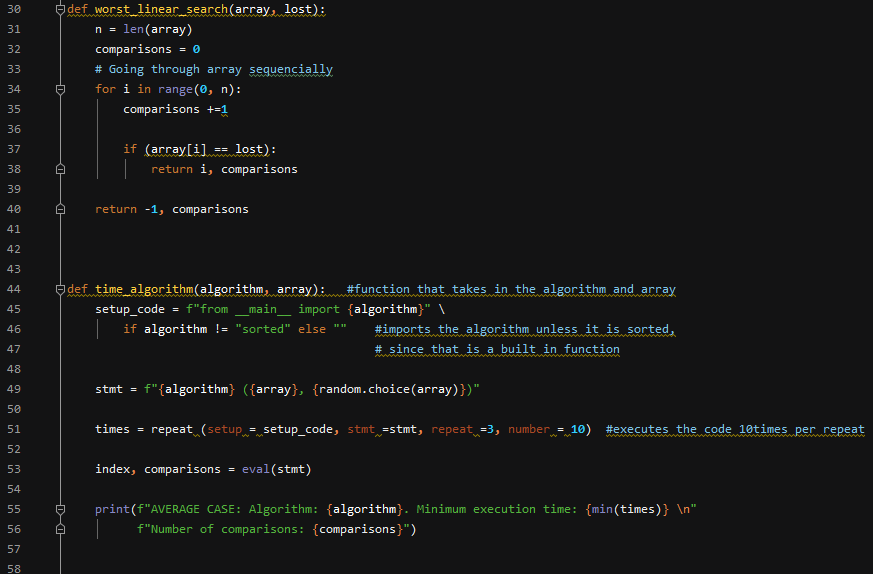
Linear or Sequential search is the simplest searching algorithm. It traverses the list completely and matches each element of the list with the key element searched for. It then returns the index of the found element. However, if the key element is not found, it returns null (Linear search algorithm, n.d.).

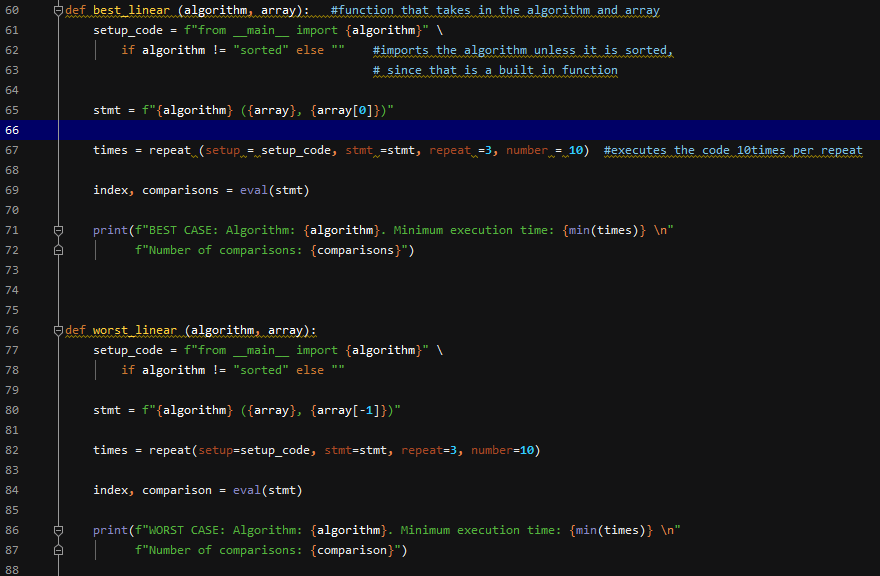
### Pseudocode

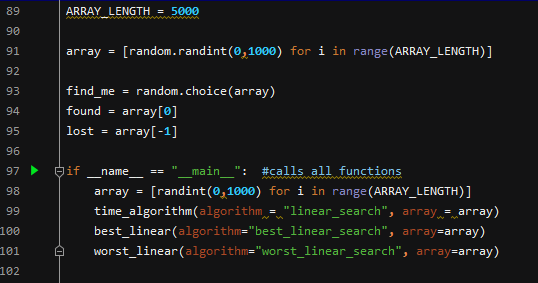


### Code

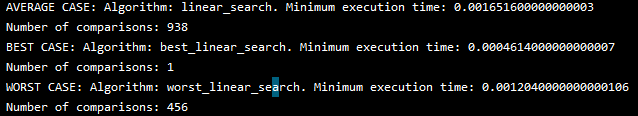






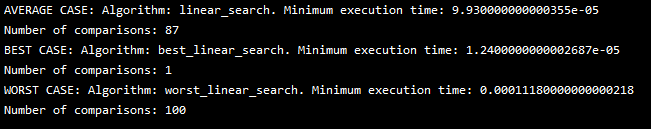


### Output

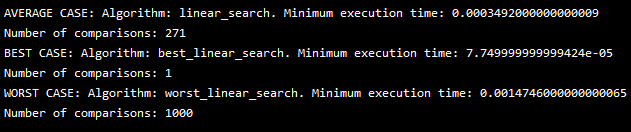


### CPU time

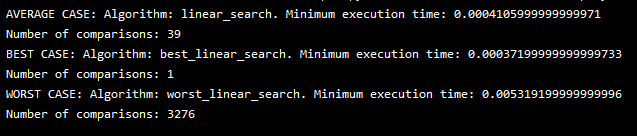
#### Range of 100



#### Range of 1000



#### Range of 5000



### Analysis

The number of comparisons in a linear search depends on the case. In an average case where the key item is either in the list (N\* (N+1)/2) or not in the list (N), there is N+1 distinct cases to consider in total. This will result in (N\* ((N+1)/2+1))/(N+1) = N/2 + N/(N+1). Therefore, the average number of comparisons would be N/2 with a time complexity of O(N) (Kiao,n.d.).

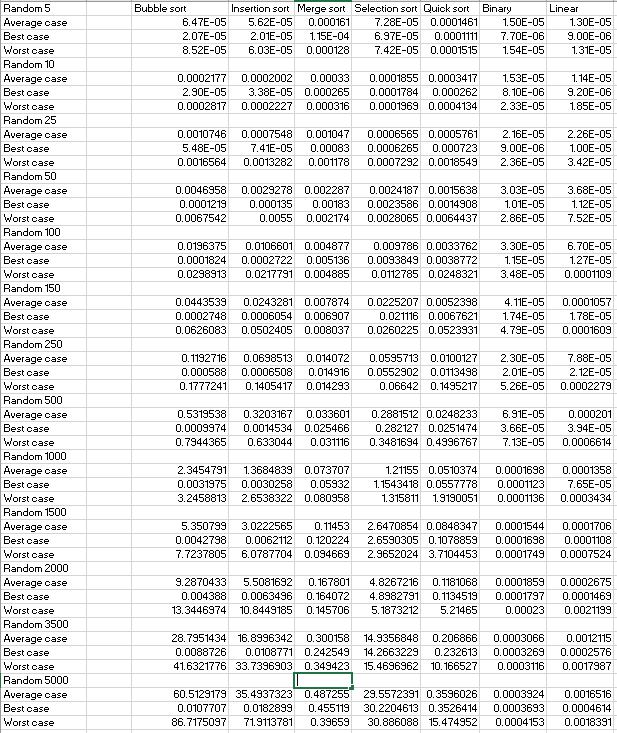
In addition, the worst-case would be similar to the average case in time complexity O(N). However, they would differ in how it would happen. The worst-case will happen if the element is at the last index of the list or the element is not in found in the list. This will result in to the maximum number of comparisons (Kiao,n.d.).

For the best-case scenario, the key element would be found at the first index of a list. This would result to 1 comparison with a time complexity of O(1) (Kiao,n.d.).

The main advantages of a linear search are its simplicity and does not require additional memory. Unlike the binary tree, it does not need the list to be sorted and can be used on arrays of any data type. However, due to its time complexity (O(N)), it is not suitable for large data sets or arrays (Linear search, n.d.).

### Graph

## CPU Time of all algorithms



Further discussions on comparing the different algorithms will be in the appendix.

## Average Case

## Best Case

## Worst Case

## Binary vs. Linear

### Average case

### Best case

### Worst case

# Reference

AMD Ryzen 3 2200G Drivers & Support. (2022). AMD. Retrieved March 29, 2023 from: [https://www.amd.com/en/support/apu/amd-ryzen-processors/amd-ryzen-3-desktop- processors-radeon-vega-graphics/amd-ryzen-3-0](https://www.amd.com/en/support/apu/amd-ryzen-processors/amd-ryzen-3-desktop-%09processors-radeon-vega-graphics/amd-ryzen-3-0)

Asymptotic notation. (n.d.). Codecademy. Retrieved March 29, 2023 from: [https://www.codecademy.com/learn/cspath-asymptotic-notation/modules/cspath- asymptotic- notation/cheatsheet](https://www.codecademy.com/learn/cspath-asymptotic-notation/modules/cspath-%09asymptotic-%09notation/cheatsheet)

Binary search. (2023). Geekforgeeks. Retrieved March 29, 2023 from: <https://www.geeksforgeeks.org/binary-> search/

Huang, S. (2020). *What is big O notation explained: Space and time complexity*. Freecodecamp. Retrieved March 29, 2023 from: [https://www.freecodecamp.org/news/big-o- notation- why-it-matters-and-why- it-doesnt-1674cfa8a23c/](https://www.freecodecamp.org/news/big-o-%09notation-%09why-it-matters-and-why-%09it-doesnt-1674cfa8a23c/)

Insertion sort. (n.d.). Watson. Retrieved March 29, 2023 from: <http://watson.latech.edu/book/algorithms/algorithmsSorting2.html>

Insertion sort in python. (n.d.). Javatpoint. Retrieved March 29, 2023 from: <https://www.javatpoint.com/insertion-sort-in-python>

Janghu, R. (n.d.). *Merge sort in python.* Scaler topics. Retrieved March 31, 2023 from: https://www.scaler.com/topics/merge-sort-in-python/

Kiao, U. (n.d.). *Time & space complexity of linear search [Math analysis].* IQ.opengenus.org. Retrieved March 31, 2023 from: <https://iq.opengenus.org/time-complexity-of-linear-search/>

Linear search algorithm. (n.d.). Javatpoint. Retrieved March 31, 2023 from: <https://www.javatpoint.com/linear-search>

Merge sort. (n.d.). Watson. Retrieved March 29, 2023 from: <http://watson.latech.edu/book/algorithms/algorithmsSorting2.html>

Moore, L. & Khim, J. (n.d.). *Bubble sort.* Brilliant. Retrieved March 29, 2023 from: <https://brilliant.org/wiki/bubble-sort/#:~:text=1%20return%20array- ,Complexity%20of%20Bubble%20Sort,%2D1%20n%E2%88%921%20comparisons>.

Pankaj. (2022). *Merge sort algorithm = java, c, and python implementation.* Digitalocean.Retrieved March 29, 2023 from: [https://www.digitalocean.com/community/tutorials/merge- sort-algorithm-java-c-python](https://www.digitalocean.com/community/tutorials/merge-%09sort-algorithm-java-c-python)

Quick sort algorithm. (n.d.). javatpoint. Retrieved March 29, 2023 from: <https://www.javatpoint.com/quick-> sort

Sorting algorithms. (n.d.). Bournetocode. Retrieved March 29, 2023 from: [https://bournetocode.com/projects/GCSE\_Computing\_Fundamentals/pages/3-1-4- sort\_alg.html#:~:text=Merge%20Sort%20%2D%20Is%20a%20'Divide,longer%20to%20sort%2 0shorter %20lists](https://bournetocode.com/projects/GCSE_Computing_Fundamentals/pages/3-1-4-%09sort_alg.html#:~:text=Merge%20Sort%20%2D%20Is%20a%20'Divide,longer%20to%20sort%2 0shorter %20lists).

The bubble sort algorithm. (n.d.). Sparknotes. Retrieved March 29, 2023 from: <https://www.sparknotes.com/cs/sorting/bubble/section1/#:~:text=The%20total%20number %20of%20comparisons,since%20no%20swaps%20were%20made>.

The department of mathematics and computer science. (n.d.). Emory oxford college. Retrieved March 29, 2023 from: https://mathcenter.oxford.emory.edu/site/cs171/quickSortAnalysis/

Types of asymptotic notations in complexity analysis of algorithms. (2023). Geeksforgeeks. Retrieved March 29, 2023 from: [https://www.geeksforgeeks.org/types-of-asymptotic-notations-in- complexity-analysis-of-algorithms/](https://www.geeksforgeeks.org/types-of-asymptotic-notations-in-%09complexity-analysis-of-algorithms/)

Upadhyay, S. (2023). *Bubble sort algorithm: Overview, time complexity, pseudocode and more.* Simplilearn. Retrieved March 29, 2023 from: [https://www.simplilearn.com/tutorials/data- structure-tutorial/bubble-sort- algorithm#:~:text=Bubble%20sort%20may%20require%20(n,)%20%3D%20O%20(n2)](https://www.simplilearn.com/tutorials/data-%09structure-tutorial/bubble-sort-%09algorithm#:~:text=Bubble%20sort%20may%20require%20(n,)%20%3D%20O%20(n2)).

Vaish, P. (2022). *Time complexity of binary search algorithm.* Byju’s exam prep.Retrieved March 29, 2023 from: https://byjusexamprep.com/time-complexity-of-binary-search-i

Wandy, J. (n.d.). *The advantage & disadvantages of sorting algorithms.* eHow. Retrieved March 29, 2023 from: <https://www.ehow.co.uk/info_8446142_advantages-disadvantages-> sorting-algorithms.html

4.4.1 Average Case Analysis of BST Operations. (n.d.). <https://gtl.csa.iisc.ac.in/dsa/node92.html>

# Appendix

## Pseudocode walkthrough

Bubble sort

It can be observed that the outer loop iterates once per element in a data with n size until the second to the last index. This is due to the last element being in the right position after the swap. The inner loop iterates through each data set of n until the second to the last number, while the i variable indicates the current position in the array that is being compared. The swap happens when the first element’s value is larger than the second element. The loop will continue until the last unsorted element. In addition, a Boolean condition has been added. If flag is 0/ True, it will break out of the loop, avoiding redundant passes.

Insertion sort

In the pseudocode, I have virtually created 2 parts of the array, the sorted and unsorted part. The sorted part will be the first element (index 0) and the unsorted part would be the rest of the element. The algorithm will insert the elements by moving each element if the left side is greater than the right side. This will loop until all elements are inserted in the correct place.

Merge sort

The pseudocode shows an if statement the triggers if the list has 1 element. If not then it will proceed to the next line to divide the list into left(a) and right(b). The function will then recursively call itself until the elements are less than or equal to 1. After that, i,j and k are initialised at 0. A while loop is put in place to run and finish the list on both a and b. The elements are then compared and the lesser element is added to the k-list. Another while loop is placed at the bottom to catch any remaining elements in j or k, if there are no more elements to compare it with.

Selection sort

In the pseudocode, it accepts a list as a parameter and uses its length to loop through it. Each time the loop iterates an element would be assigned to the variable min. The inner loop will start at the next index after min. An if statement will have a condition which will trigger if the i-th element is lesser than the min and will assign the i as min if the condition is true. The inner loop will continue till the end of the list and will swap the least value with the value of x in the outer loop.

Quick sort

The pseudocode has three steps, divide, conquer and combine. In dividing, it first picks a pivot then rearranges the array into 2 subarrays. The left array would have elements that are lesser than the pivot and the right array would have elements that are larger that the pivot. In the conquer step, it recursively calls itself to sort the 2 subarrays. And in the combine step, it combines the already sorted arrays as seen in the pseudocode below (Quick sort, n.d.).

Binary search

As seen in the pseudocode, the function accepts 2 parameters, the array and the key element. It then initialises low as 0 and high as the length of the list. To iterate through the list, a while loop is placed to run as long as lower index is less than the higher index. The middle element is then determined by adding the two together and dividing by 2. If the key element is at the middle, it will return the element and if it is lower than the middle index, high will be equal to middle index minus 1. An else clause is then placed to assign low as middle index plus 1 if the key element is higher than the middle element. If the element is not found, the function will return a value of -1.

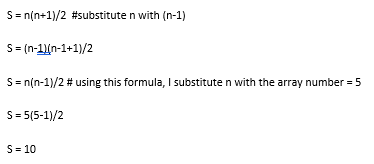
Linear search

The pseudocode accepts 2 parameters, the array and the key element. The for loop will iterate through the elements in an array and an if statement will catch the value sought after. It will then return the index of the key element. However, if the key element is not found in the list, the function will come out of the for loop and return not found or -1.

## Sum of first(n-1) positive integers

This can be proven using the sum of the first (n-1) positive integer ((n-1) + (n-2)… (2) + (1)). As the first pass compares the first element of the array to the (n-1)th element, it requires (n-1) comparisons. The second pass will run until the (n-2)th element and does a comparison with the (n-1)th element. This will be (n-2) comparison. This will continue until the last pass in the first element which requires 1 comparison.

Therefore, the total number of comparisons in a bubble sort is: (n-1) + (n-2)… (2) + (1). I then use the sum of natural integers formula to determine the number of comparisons.



Furthermore, as I have determined the number of comparisons in the algorithm, I can determine the time complexity. Determining the time complexity if an algorithm requires knowledge of Big O notation. Big O notation is a mathematical notation that defines the limiting behaviour of a function, if the argument is inclined to an infinity or a particular value. It uses algebraic notations to describe the complexity of a code. Also, it is used to describe the upper bound of a complexity (Huang, 2020).

## Evaluating Sorting algorithms

The average case graph shows the performance of each sorting algorithm when given a specific size of data. It can be observed that each algorithm performed at the same rate from a data size of 100 to 1000 with minimal differences in the time it took to sort the array. However, the disparity is significant after the 1000 random elements mark. It can be noted that the bubble sort performed worst amongst all other algorithms while quick sort and merge sort remained relatively linear. This is due to the O(N2) time complexity of the bubble sort. This algorithm is ideal in small data sets but mostly teaching environments with regards to sorting algorithms.

However, in the best-case scenario, it can be noted that the bubble sort and insertion sort performed the best. This is due to the implementation of the code where it can be modified to terminate early of the array is already sorted. It can be noted that the selection sort fared the worst as it still needs to go through each element and compare them even though there are no swaps.

The worst-case scenario graph shows how the algorithms started to vary in time at 1000 elements. In a reversed array, the bubble sort time significantly rose the highest in the 5000 elements mark followed by insertion sort while merge sort stayed linear all through out. It can be said that merge sort performs best in large quantity of data however, it uses more memory and is a bit slower in smaller data sets.

A combination of 2 algorithms maybe used to make sorting more efficient. Insertion sort is an in-place algorithm that can be used with smaller data sets. Combined with merge sort, they can complement each other in terms of efficiency.

### Binary search vs linear search

The graphs show the performance of the two searching algorithms. In an average scenario it can be noted that the linear search time significantly increased at the 1000 elements mark while the binary search maintained a slight linear increment until the 5000th element. This is due to the time complexity difference where the binary uses the divide and conquer method and uses O(logN) while the linear search uses O(N) and has to start from the beginning of the list until it finds the target element.

In addition, it can be seen that both algorithms performed close to equal in the best-case scenario. However, the reasons behind are different. Binary search best case occurs when the target element is at the middle of the list while linear best case occurs when the target element is at the beginning of the list.

Furthermore, in the worst-case scenario it can be observed that there is a gross difference in the performance of both algorithms. The binary search vastly out performs linear search past 100 elements. However, binary search has a drawback as it requires the list to be pre-sorted before running where this does not apply to linear search.

It can be concluded that linear search can be a viable searching algorithm in data sets up to 1000. Larger data sets will require the binary search however this would require the set to be pre-sorted. Depending on the users available resources in terms of memory, quicksort or merge sort can be used to sort the list.