Higher Diploma in Computer Science - Data Analytics

Computer Thinking with Algorithms | 46887

Noa Pereira Prada Schnor | G00364704

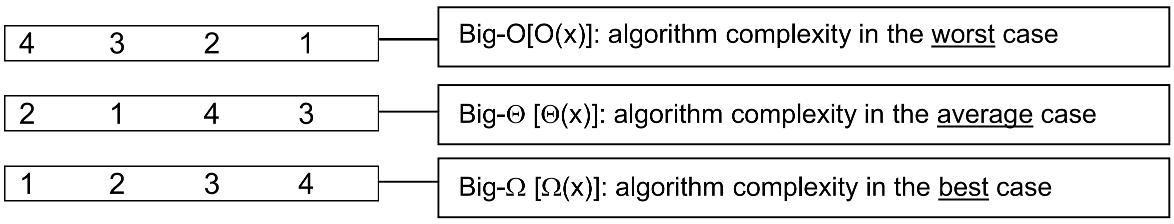
Contents

1. INTRODUCTION

Sorting Algorithms reorganize elements of a list into a specific order. It can be a numerical or lexicographical order [1,2]. As sorting can reduce the complexity of a problem, sorting is an important algorithm [3]. It is estimated that one quarter to a half of all computing time is spent on sorting lists [4].
There are several ways to sort elements in an array, and every way has its pros and cons. Therefore, there is no sorting algorithm that it is the best for sorting all type of lists. To find the best sorting algorithm to solve a specific problem, an algorithm analysis (of the complexity) is needed [4].
In this report, I've introduced and discussed relevant concepts, like time and space complexity and five sorting algorithms that were chosen to be used in the application that benchmarked them. Finally, I've described the implementation of the application and discuss the results of the benchmark.

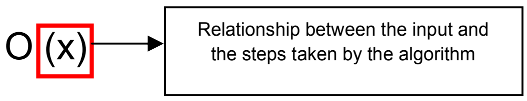
1.1. Asymptotic notation

Asymptotic notation, also known as an algorithm's growth rate, is the computing the running time of an algorithm, identifying its behaviour as the input size grows. There are three different forms of asymptotic notation: Big-O notation, Big-Θ notation, and Big-Ω notation [5].
Big-O is an asymptotic notation for the worst case, Omega (Big-Θ) for the best case, and Theta (Big-Ω) for the average case. In the present project, Big-O was chosen to analyse the performance of five sorting algorithms: insertion sort, merge sort, counting sort, quicksort and Timsort [5].


4 3 2 1
Big-O[O(x)]: algorithm complexity in the worst case
2 1 4 3
Big-Θ [Θ(x)]: algorithm complexity in the average case
1 2 3 4
Big-Ω [Ω(x)]: algorithm complexity in the best case

1.2. Time complexity (Big-O)

Big-O notation is the standard metric that computes the complexity of an algorithm. In other words, Big-O notation is used to measure the order of growth of an algorithm, the number of steps that the algorithm takes as the input size grows [6,7].


O(x)
Relationship between the input and the steps taken by the algorithm

Common types of time complexities in Big O Notation.

Name	Big-O
Constant	O(1)
Linear	O(n)
Quadratic	O(n^2)
Cubic	O(n^3)
Exponential	O(2^n)
Logarithmic	O(log(n))
Linearithmic	O(n log(n))

1.2.1. Constant [O(1)]

The complexity of an algorithm is said to be constant when no matter the size of the input, the steps to execute the algorithm remain constant. So the running will always be the same. The constant complexity is denoted by O(1) [6,8].

```
def compare(a, b):
    if a > b:
      return True

    else:
      return False
```

1.2.2. Linear [O(n)]

When the running time of an algorithm increases linearly with the size of the input data is said that its time complexity is linear. Linear time is the best possible time complexity when all the elements in the input data must be examined. The linear complexity is denoted by O(n) [6,9]. For example, a function that prints all elements of a list requires time proportional to the number of elements of the list [6,9].

```
def print_element(array):
    for element in array:
        print(element)
```

1.2.3. Quadratic [O(n^2)]

An algorithm that performs a linear time operation for each value in the input data has quadratic time complexity. In other words, the algorithm performs proportionally to the squared size of the input data. The function below is an example of an algorithm that has O(n^2) time complexity as it iterates twice over the array [6].

```
def add2_elements(array):
    for a in array:
       for b in array:
          print(a+b)
```

1.2.4. Cubic [O(n^3)]

An algorithm runs in cubic time if the running time of the three loops is proportional to the cube of n. So, if the input size doubles, the number of steps that the algorithm will take is multiplied by 8 [7,10,11].

```
def add3_elements(array):
    for a in array:
       for b in array:
          for c in array:
            print(a + b + c)
```

1.2.5. Exponential [O(2^n)]

An algorithm that has exponential time complexity when the growth doubles with each calculation to the input data set. An example of an algorithm that has O(2^n) time complexity is a recursive function that calculates the Fibonacci numbers [6].

```
def fib(number):
    if number  == 0 or number == 1:
       return number

    else:
          return fib(number-1) + fib(n-2)
```

1.2.6. Logarithmic [O(log(n))]

An algorithm has a logarithmic time complexity when the size of its input data reduces in each step. For this reason, a logarithmic algorithm is considered highly efficient [6,9].

```
def binary_search(array, left, right, num):

  if right >= 1:

      mid = (left + (right -1))/2

      if array[mid] == num:

          return mid

      if array[mid] > num:

         return binary_search(array, left, mid - 1, num)

     return binary_search(array, left, mid + 1, num)

  return - 1
```

1.2.7. Linearithmic | Log-linear | Quasilinear [O(n log(n))]

An algorithm has linearithmic | log-linear | quasilinear time complexity when every single operation in the input data have a logarithm time complexity [6].
A linearithmic time complexity algorithm performs worse than linear time complexity (O(n)) algorithm, but better than a quadratic time complexity (O(n^2)) algorithm [6,12]. The linearithmic time complexity floats around linear time complexity (O(n)) until the input reaches advanced size [6,12].

1.3. Space complexity

An algorithm memory may be used for variables, instructions and execution. Space complexity is the memory used by the algorithm to run, including the space used for inputs, and instructions [13].
Space complexity is quite important when the data volume has a similar or larger magnitude than the memory available. So, algorithms with high space complexity could perform worse than the suggested by Big-O as it probably would have to swap memory constantly [13].

Value of memory used by different types of datatype variables

Datatype	**Memory used**
bool, char, _int8	1 byte
_int16	2 bytes
float, _int32	4 bytes
_int64	8 bytes

```
def sum_all(a, b, c):
    n = a + b + c

    return n
```

The example above has 4 variables all integers. Each integer will use 4 bytes of memory, so the total memory required is 20 bytes [(4(4) + 4)], including the memory for the return value (4 bytes).

1.4. Performance

The main objective of measuring the performance of an algorithm is to check how the increase in input data affects the rate of growth of operations required. Time complexity and space complexity can be used to measure an algorithm's performance, and both are impacted by the length of the input data [5].

1.5. In-place sorting algorithms and not-in-place sorting algorithms

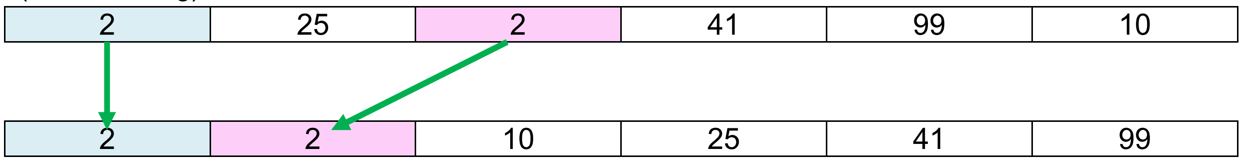
In-place sorting algorithms do not require extra space for comparison as the sort happens within the array itself. On the other hand, sorting algorithms that requires space for comparison and temporary storage of elements of the list to be sorted are called not-in-place sorting [14,15].

In-place sorting algorithms	Not-in-place sorting algorithms
Bubble sort	Merge sort
Insertion sort	Counting sort
Selection sort	

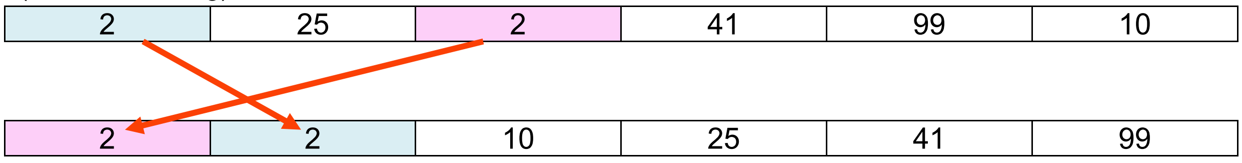
1.6. Sorting stability

A sorting algorithm is stable if objects with equal key appear in the output array in the same order as they appear in the input array [14,16]. In other words, the object that appears first in the input array should appear first in the output array [14].

(Stable sorting)



(Unstable sorting)



Stable sorting	Non-stable sorting
Insertion sort	Quicksort
Merge sort	
Count sort	
Timsort	

1.7. Comparison-based and non-comparison-based sorting algorithms

Sorting algorithms can be classified into two categories: comparison-based, and non-comparison based. The difference between them is the comparison decision [17].

1.7.1. Comparison-based sorting algorithms

In comparison-based sorting algorithms, a comparator is required to compare numbers or items. This comparator defines the ordering e.g. numerical order, lexicographical order, also known as dictionary order to arrange elements [17].

1.7.2. Non-comparison-based sorting algorithms

Non-comparison based sorting algorithms perform sorting not using comparison, but by assuming the data to be sorted [17].

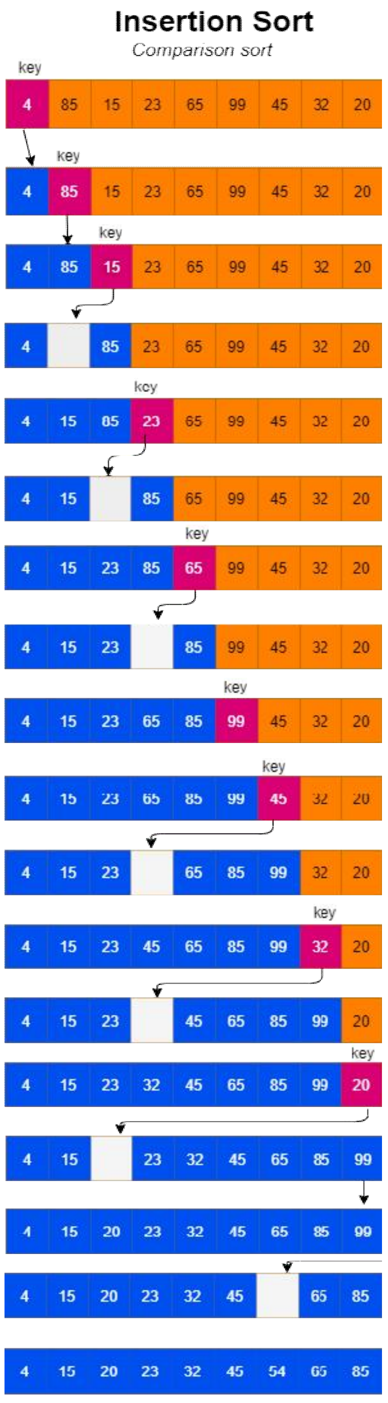
1.7.3. Comparison-based versus non-comparison-based sorting algorithms

Usually, non-comparison based sorting algorithms are faster than comparison-based sorting algorithms due to the working difference between them [17].

	Comparison-based	**Non-comparison-based**
Speed	Limit of speed is O(n log(n)).	Faster as they do not do a comparison. The limit of speed is O(n).
Comparator	Require comparator (<, <=, >, >=, ==) to sort elements.	Do not require a comparator.
Usage	Can use to sort any object.	Cannot sort anything other than integers.
Memory	The best case is O(1): possible to sort an array in-place. e.g. Quicksort.	Always O(n).
Examples	Quicksort Merge sort Insertion sort	Counting sort Bucket sort Radix sort

Usually, comparison-based sorting algorithms are used in real-world data. However, non-comparison-based sorting algorithms are a good option if the data meets the special conditions required to use them [17].

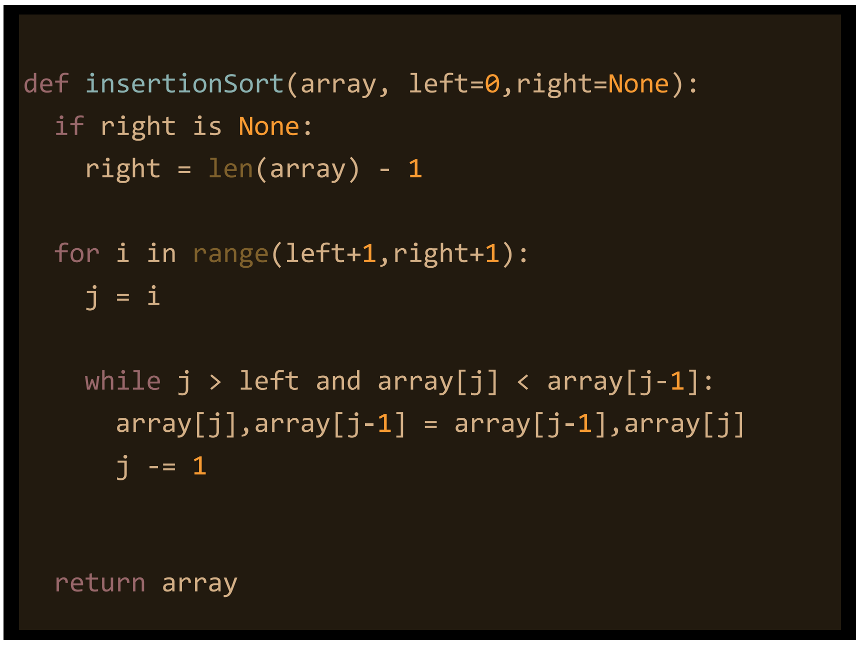
2. SORTING ALGORITHMS



2.1. Insertion Sort

It is an in-place comparison sort algorithm of choice when the data is nearly sorted and/or the array size is small[18,21,22]. It works similarly to the way a player sorts the cards: a subarray is kept sorted and an element of the unsorted part of the array is picked and inserted at the correct position in the sorted subarray. This sequence happens until all the elements of the array are in the correct position[18,21].
Insertion sort has the following steps [19,20]:

1: Iterate from the first to the last element of the array.
2: Compare the element in question (key) to its adjacent.
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.
4.Repeat until the array is sorted.



```
def insertionSort(array, left=0,right=None):
  if right is None:
    right = len(array) - 1

  for i in range(left+1,right+1):
    j = i

    while j > left and array[j] < array[j-1]:
      array[j],array[j-1] = array[j-1],array[j]
      j -= 1


  return array
```

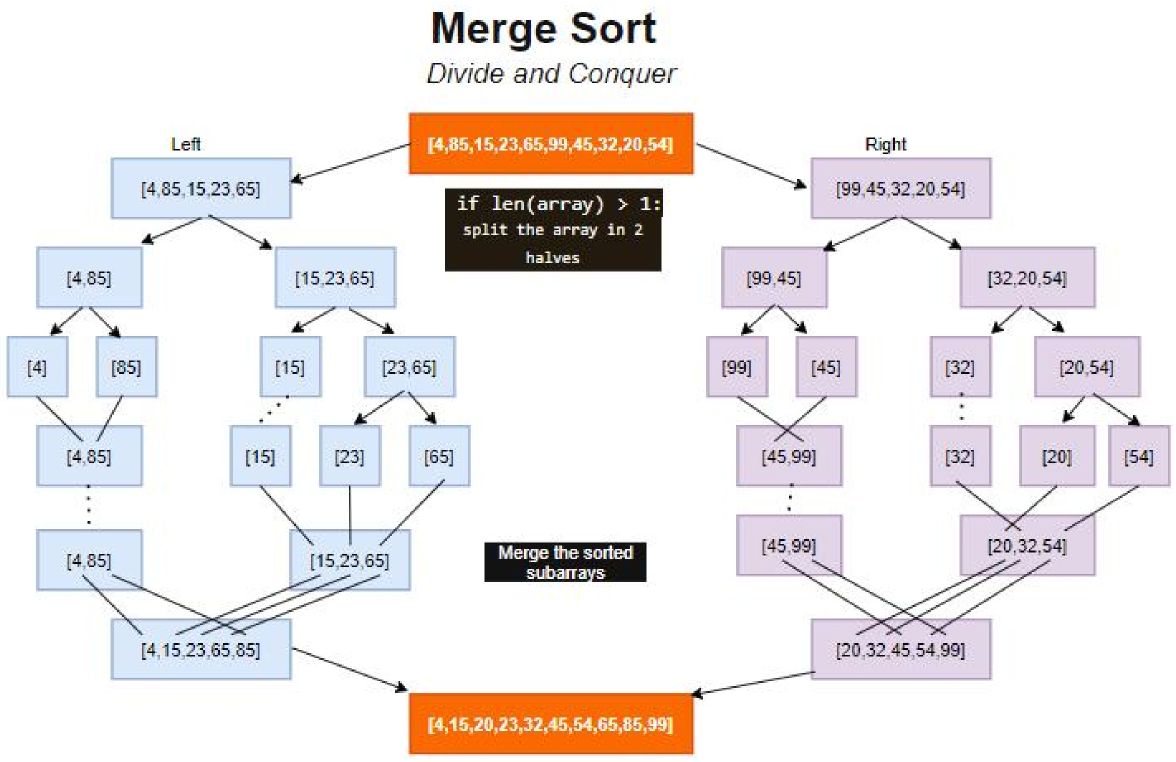
	Complexity[18,22]
Worst case	$O(n^2)$
Best case	$\Omega(n)$
Average case	$\Theta\ (n^2)$
Space	$O(1)$
Stable	*Yes*
Class	*Comparison sort*

2.2. Merge Sort

Developed by J. von Neumann in 1945, merge sort is a recursive algorithm that uses a divide and conquer strategy. Merge sort involves 3 steps[23,24]:

1. Continually splits an array in half to divide the problem into small chunks;
2. Solve the subproblems;
3. Merge the sorted halves/chunks resulting in a single sorted array.

In the following example, the length of the array is 10, so the left array contains the first 05 elements from the left side of the original array and the right array contains the rest of the elements from the original array[23,24].
It is a recursive function that repeats the splitting process in the halves until only the separate element remains. Then, merges the lists resulting in a final sorted array[23.,24].



```python
def mergeSort(array, left=None, mid = None, right=None):

  if len(array) > 1:
    if left is None and mid is None and right is None:
      mid = len(array)//2
      left = array[:mid]
      right = array[mid:]

    mergeSort(left)
    mergeSort(right)

    i, j, k = 0, 0, 0

    while i < len(left) and j < len(right):
      if left[i] < right[j]:
        array[k] = left[i]
        i += 1
      else:
        array[k] = right[j]
        j += 1
      k +=1

    while i < len(left):
      array[k] = left[i]
      i += 1
      k += 1

    while j < len(right):
      array[k] = right[j]
      j += 1
      k += 1

  return array
```

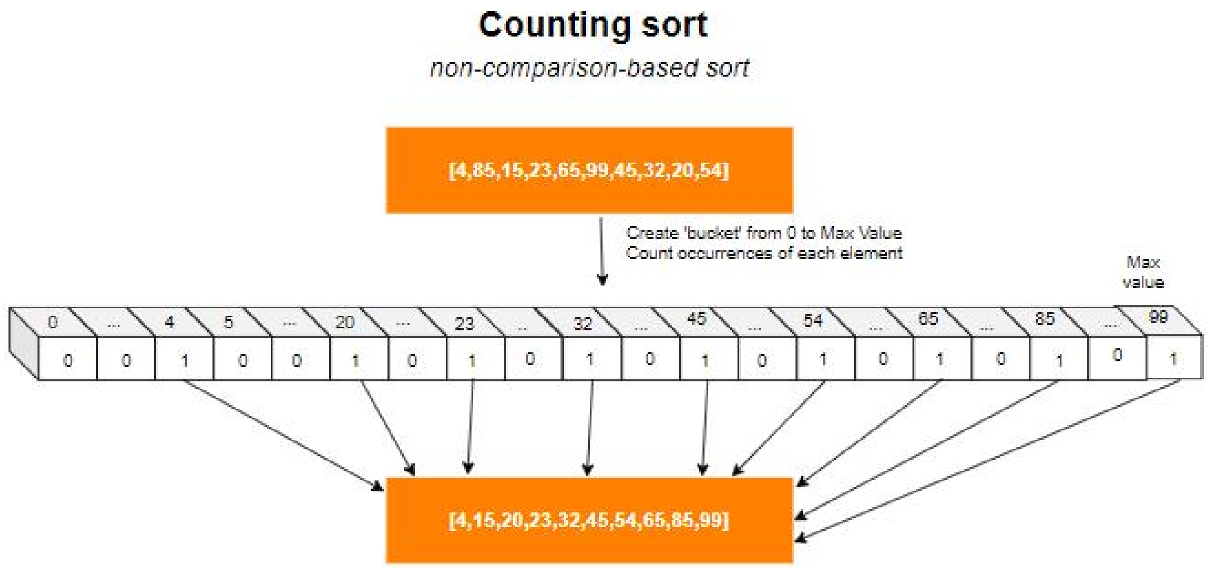
	Complexity
Worst case	*O(n log(n))*
Best case	*Ω(n log(n))*
Average case	*Θ (n log(n))*
Space	*O(n)*
Stable	*Yes*
Class	*Comparison sort*

2.3. Counting sort

Developed by Harold H. Seward in 1954, counting sort is a non-comparison sorting algorithm used when linear complexity is the need. It sorts an array by counting the number of occurrences of each distinct element in the array, then calculates the position of each element in the array resulting in a sorted array [25,26].

There special conditions required to use counting sort as follows [27]:
1. Values to be sorted must be integers;
2. The input should lie in the range of 0 to k.

Note: if the value k is too high, the running time of the counting sort will increase [27].

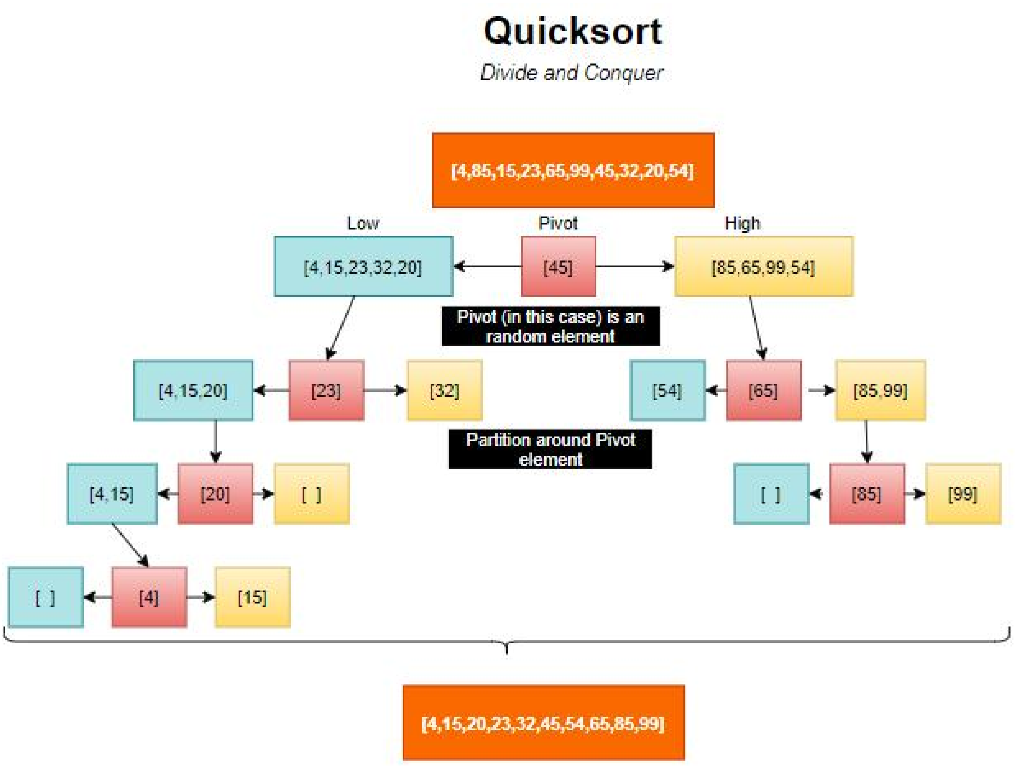


```
def countingSort(array):

  counter = [0] * (100)

  for i in array:
    counter[i] += 1

  i=0
  for j in range(100):
    for _ in range(counter[j]):
      array[i] = j
      i += 1

  return array
```

	Complexity
Worst case	*$O(n+k)$*
Best case	*$\Omega(n+k)$*
Average case	*$\Theta(n+k)$*
Space	*$O(k)$*
Stable	*Yes*
Class	*Non-comparison sort*

2.4. Quicksort

In-place, not stable sorting algorithm developed by Tony Hoare in 1959. Quicksort uses the divide and conquer strategy to sort an array dividing it recursively into 3 segments: left(low), middle(pivot) and right(high). The elements in the left segment are smaller than the 'key'(pivot), and the right segment contains elements bigger than the key. So, each segment can be sorted independently and would not require merging following the sorting of segments [28].



The pivot element can be chosen in different ways[29,30]:

- Random element
- First element
- Last element
- Element in the median position

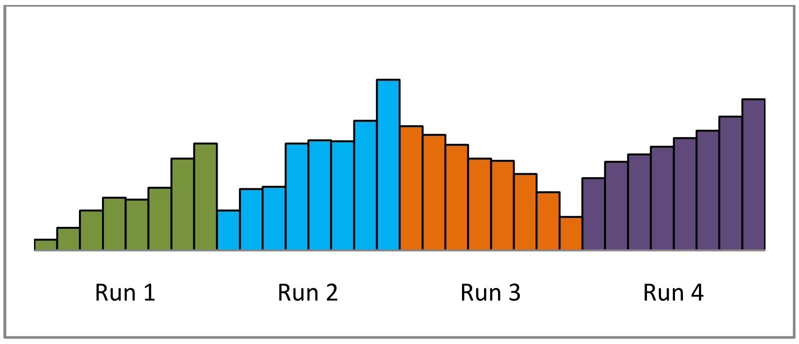
The choice of the pivot determines the performance of quicksort. Quicksort has the worst-case time complexity if the pivot is the first element of the array. However, if a good pivot is chosen, then quicksort has the time complexity of $O(n \log(n))$ exceeding the performance of merge sort[29,30].

```
def quickSort(array):
  if len(array) < 2:
    return array

  low, same, high = [],[],[]

  pivot = array[np.random.randint(0,len(array)-1)]

  for item in array:
    if item < pivot:
      low.append(item)
    elif item == pivot:
      same.append(item)
    elif item > pivot:
      high.append(item)

  return quickSort(low) + same + quickSort(high)
```

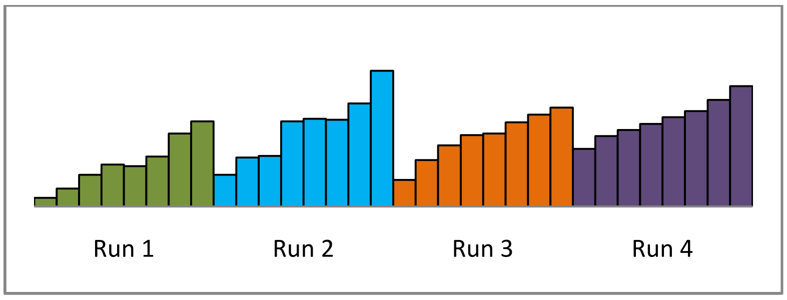
	Complexity[29,30]
Worst case	*O(n^2)*
Best case	*Ω(n log(n))*
Average case	*Θ (n log(n))*
Space	*O(n log(n))*
Stable	*No*
Class	*Comparison sort*

2.5. Timsort

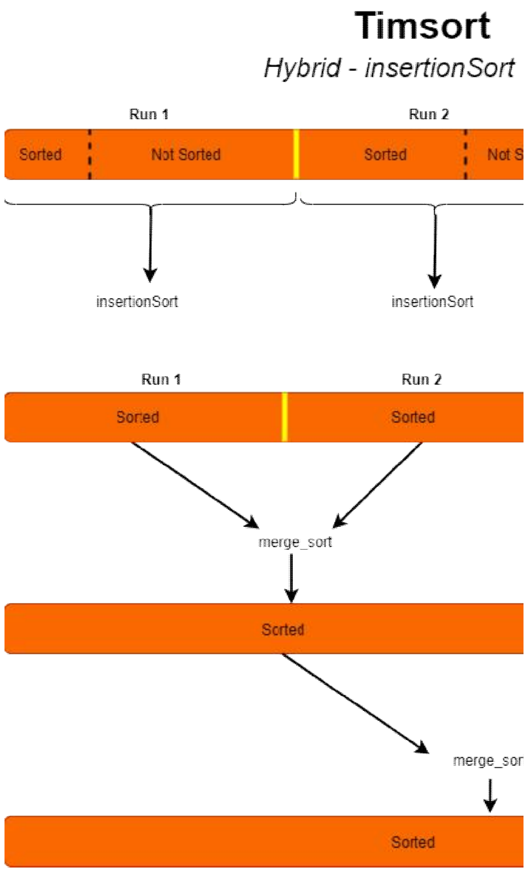
It is a hybrid sort algorithm designed by Tim Peters in 2002 to use already-ordered elements that usually exist in real-world data, also called 'natural runs' [31,32]. Timsort is implemented in Java (Arrays.sort()) and in Python(sorted() and sort()).
As any hybrid sorting algorithm, Timsort blends different sorting techniques into one algorithm to result in a good real-world performance. Tim chose to blend insertion sort and merge sort. However, Timsort also identifies 'natural runs' in the input data simply comparing the next element to the previous one. In this step, to make Timsort a stable sorting algorithm, equal values are not included in the same run [31].
As shown in the above graph, there are existing parts of the array that can be already sorted. Not always all the elements of a run will be exactly in the right position. The data represented in the graph below has 4 runs, run 3 is a descending run and the rest are ascending runs.



After the runs are identified, Timsort reverses the descending runs



Timsort works with the parameter 'minrun', the minimum size that a run should have before merging runs. Minrun should be the number that when the size of the input data is divided by it is equal to a power of two, so the merge procedure works faster. Besides that, Minrun also should be small enough for insertion sort to perform faster and [33].
Besides that, Timsort works with a stack that keeps track of the position of merged runs in the order they are in the array and new runs are added to the top of it [32].



Summary of Timsort steps [32]:
1. Find 'natural runs'
2.Reverse the runs
3.Add input elements in a run by insertion sort until the minimum size is reached
4.Merge runs

```
min_num = 32

def minrun(n):
  r = 0
  while n >= min_num:
    r |= n & 1
    n >>= 1

  return n + r
```

```
def timSort(array):
  n = len(array)
  min_run = minrun(n)

  for s in range(0,n,min_run):
    e = min(s + min_run - 1, n-1)
    insertionSort(array,s,e)

  size = min_run

  while size < n:

    for left in range(0,n,2*size):
      mid = min(n-1,left+size-1)
      right = min((left+2*size-1),(n-1))
      merge_sort(array,left,mid,right)

    size = 2*size

  return array
```

	Complexity[32]
Worst case	*O(n log(n))*
Best case	*Ω(n)*
Average case	*Θ (n log(n))*
Space	*O(n)*
Stable	*Yes*
Class	*(Hybrid) based on 2 comparison sort*

3. IMPLEMENTATION AND BENCHMARKING

Microbenchmark was performed to compare the performance of the chosen sorting algorithms.

3.1.Testing environment

HP Pavilion 15 laptop with Intel Core i3-5010u processor, that has a maximum 2.10GHz frequency. The operating system used was Windows 8.1 x64.

3.2.Sorting algorithms

As I was interested to analyse Timsort, as it is widely used on real-world data, I also decided to include in the project insertion sort and merge sort. I chose counting sort as it has a good performance with an array of non-negative integers. Finally, quicksort was chosen because is considered an efficient sorting algorithm.

Insertion sort insertionSort(array, left, right)	Adapted. Original code developed by Tarcisio Marinho [A]
Merge sort (Benchmark) mergeSort(array,left,mid,right)	Adapted. Original code developed by Panjak [B]
Merge sort (Timsort) merge_sort(array,lt,mid,rt)	Adapted. Original code available at Python Pool [C]
Counting sort countingSort(array)	Adapted. Original code available at Codez Up [D]
Quicksort quickSort(array)	Adapted. Original code developed by Santiago Valdarrama [E]
Timsort timSort(array)	Adapted from Tarcisio Marinho [A] and DrYshio [F]
Minrun (Timsort) minrun(n)	Code developed by Karthik Desingu [G]

A. https://github.com/tarcisio-marinho/sorting-algorithms/blob/master/timsort.py
B. https://journaldev.com/31541/merge-sort-algorithm-java-c-python/
C. https://www.pythonpool.com/python-timsort/
D. https://codezup.com/implementation-of-counting-sort-algorithm-in-python/
E. https://realpython.com/sorting-algorithms-python/
F. https://github.com/DrYshio/Timsort/blob/main/main.py
G. https://www.codespeedy.com/timsort-algorithm-implementation-in-python/

3.3.Data generation

Data generation is an important part of the performance analysis of sorting algorithms. The function *random_array* was created to generate an array containing 10 arrays of the same size. I've used *numpy.random.randint()* function, that returns a random sampling with specified shape (in this case, positive integers from *0 to 99*).
I've chosen to generate an array with 10 arrays of the same size, so I could pass a different array to be sorted every time that the sorting function run to measure its running time.
Arrays of 15 different sizes (*20, 50,100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750,10000*) were generated and stored in a list called *list_arr*.
As the created arrays were used as an argument for the different sorting functions, I've made a copy of the arrays to keep the original arrays unsorted. To do that, I've used the *copy.deepcopy()* from the copy module.

```
random_array(n)
```

```
list_arr = [arr20, arr50, arr100, arr250, arr500, arr750, arr1000,arr1250,
arr2500, arr3750, arr5000, arr6250, arr7500, arr8750, arr10000]
```

3.4.Run time measurement

The *time* module was used to measure the running time of each sorting algorithm. The *time.time()* function returns a floating-point number in seconds. To convert it to milliseconds, the value was multiplied by 1000.
The function *timerRun* returns the average of 10 runs and it is called by another function named *sortRun*, which returns an array of the average time of 10 runs for the 15 different input sizes (*list_arr*).

```
timerRun(function, array)

sortRun(sort_function)
```

3.5.Console output

To show the running time of the five chosen sorting functions using different input sizes, I've written a function called *dataframeResults*, which returns a dataframe of the results using *pandas.Dataframe()*. Before doing that, I've used *numpy* to convert the results in a *numpy.array()* and *numpy.vstack()* to stack the arrays returned by the sortRun function row-wise.

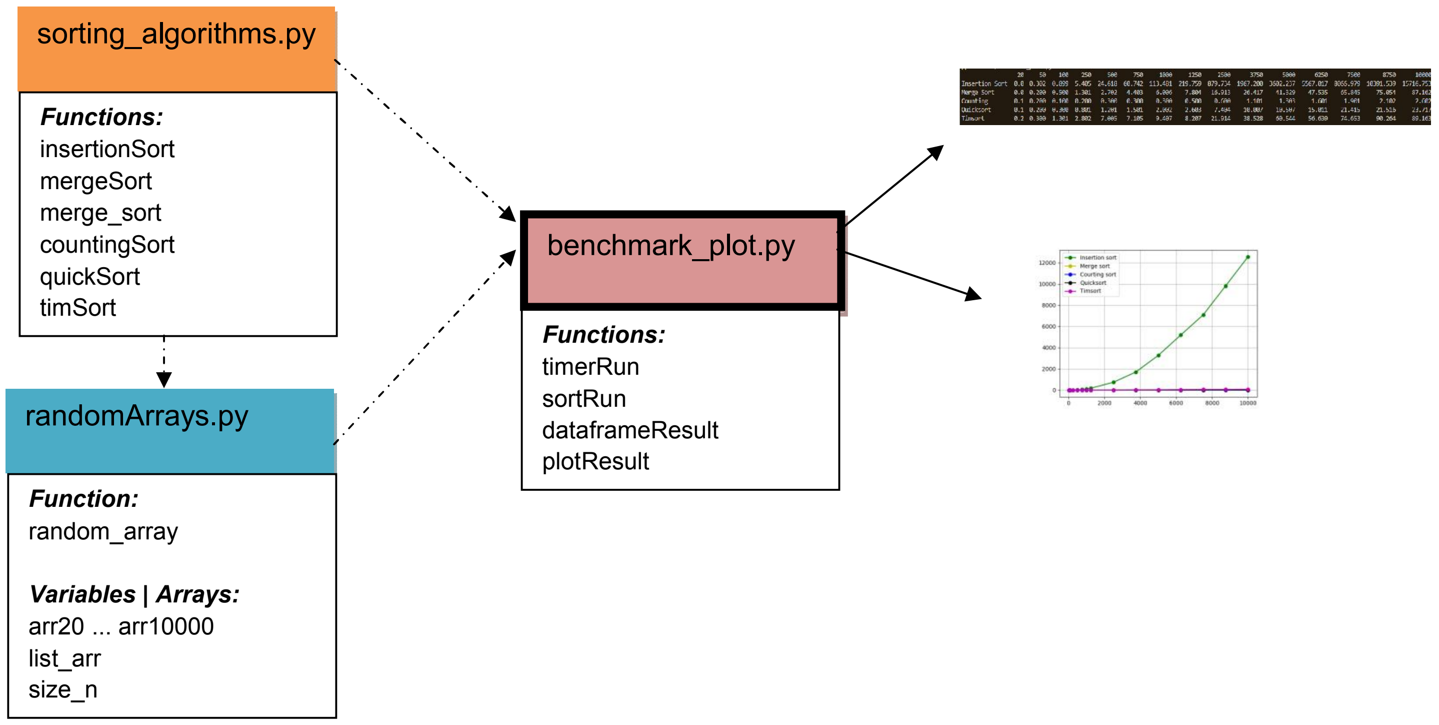
```
dataframeResult()
```

3.6.Plot

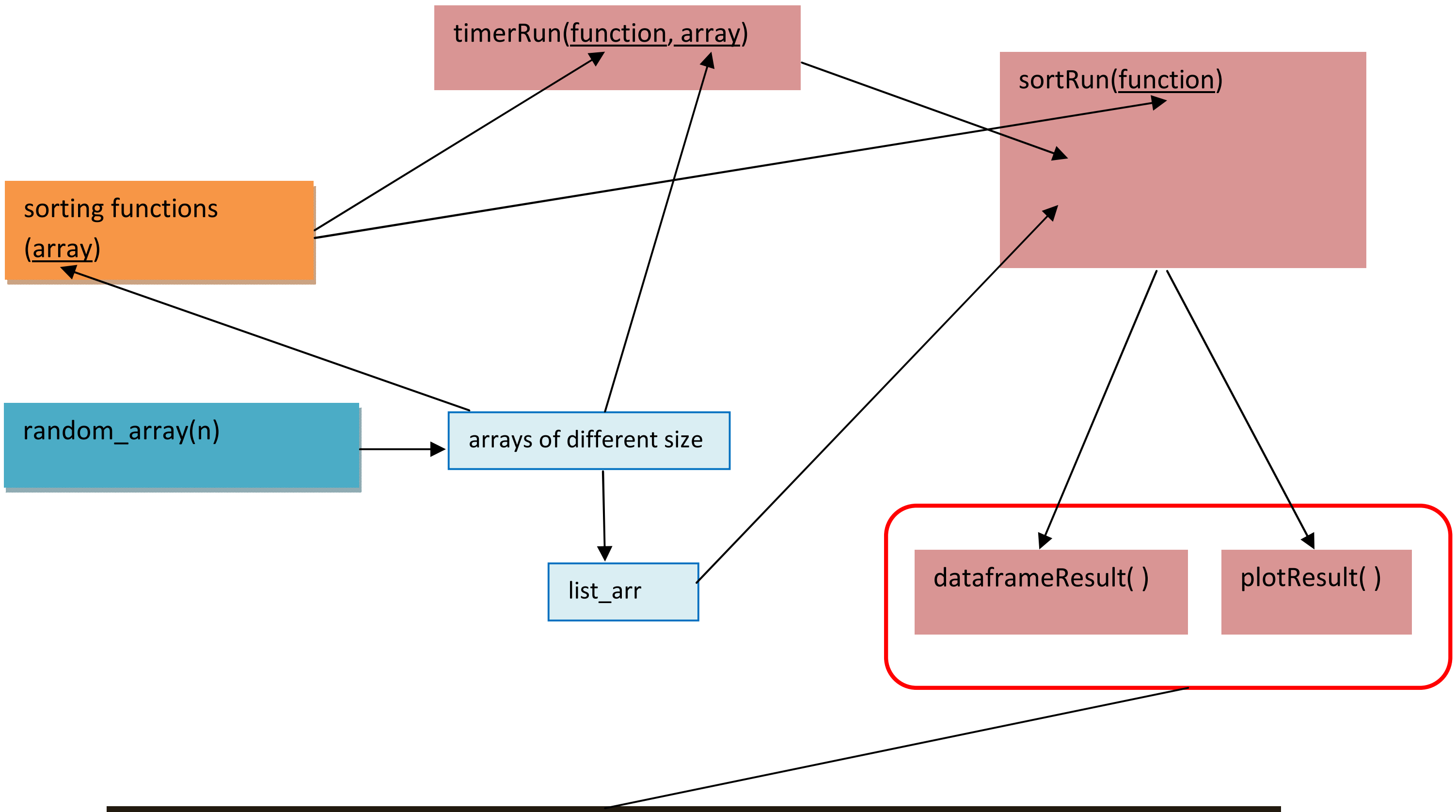
To plot the performance I've used *matplotlib*. The developed function named *plotResult* produces a line plot of the running time (in milliseconds) and the input size of all functions in one single graph.

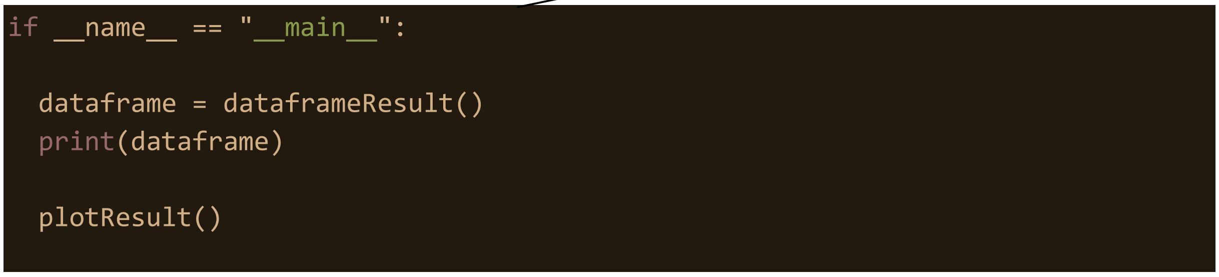
```
plotResult()
```

Note about the files:
There are 3 files with all the code created for this project. In the file *'randomArrays.py'* it is possible to find the *random_array* function, and the arrays of different input size created. All the sorting functions are in the *'sorting_algorithms.py'*. While the *'benchmark_plot.py'* file runs the __main__ , which calls *dataframeResult* and *plotResult* functions. Other 2 functions are in *'benchmark_plot.py'*: *timerRun* and *sortRun*. These functions return the running time that will be used by *dataframeResult* and *plotResult* functions.



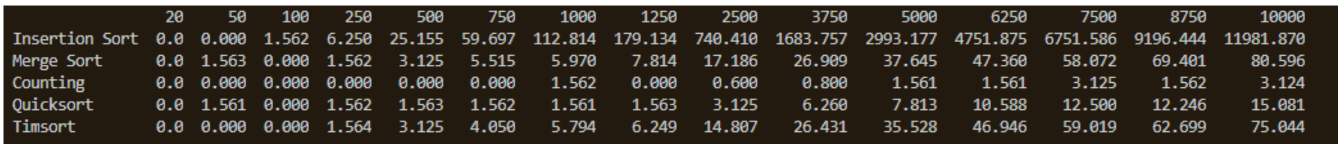
```
                  20    50    100    250     500     750     1000     1250     2500      3750      5000      6250      7500       8750      10000
Insertion Sort   0.0 0.302 0.099  5.405  24.610  60.742  113.401  219.759  879.734  1967.200  3602.237  5567.017  8066.979  10391.539  15716.753
Merge Sort       0.0 0.200 0.500  1.301   2.702   4.403    6.006    7.804   16.913    26.417    41.329    47.535    65.845     75.054     87.162
Counting         0.1 0.200 0.100  0.200   0.300   0.300    0.300    0.500    0.600     1.101     1.303     1.601     1.901      2.102      2.602
Quicksort        0.1 0.200 0.300  0.801   1.201   1.501    2.002    2.603    7.404    10.007    10.507    15.011    21.415     21.516     23.71
Timsort          0.2 0.300 1.301  2.802   7.005   7.105    9.407    8.207   21.914    38.528    60.544    56.639    74.653     90.264     89.163
```



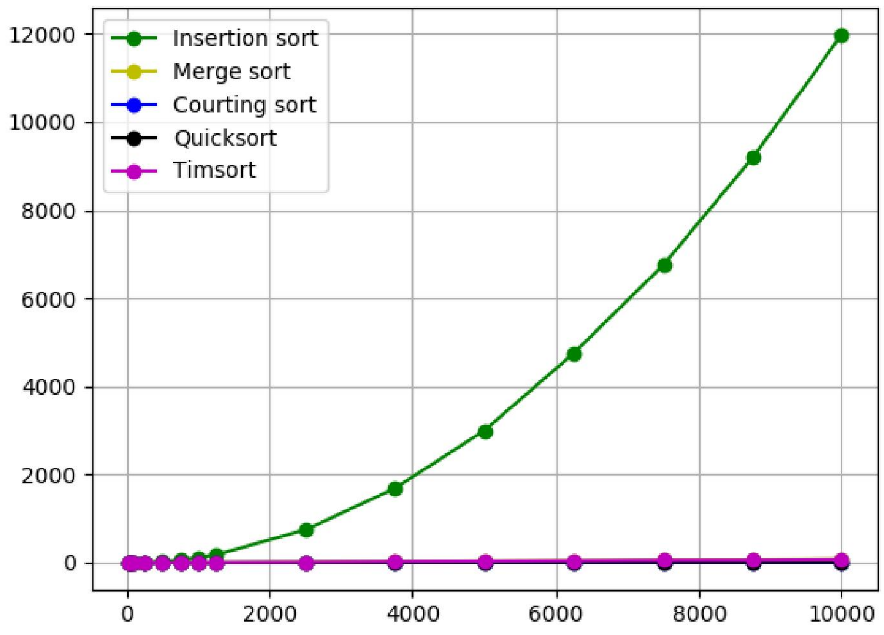


```
if __name__ == "__main__":

  dataframe = dataframeResult()
  print(dataframe)

  plotResult()
```

3.7.Results



```
                20     50    100    250     500     750     1000     1250     2500      3750      5000      6250      7500      8750      10000
Insertion Sort  0.0  0.000  1.562  6.250  25.155  59.697  112.814  179.134  740.410  1683.757  2993.177  4751.875  6751.586  9196.444  11981.870
Merge Sort      0.0  1.563  0.000  1.562   3.125   5.515    5.970    7.814   17.186    26.909    37.645    47.360    58.072    69.401     80.596
Counting        0.0  0.000  0.000  0.000   0.000   0.000    1.562    0.000    0.600     0.800     1.561     1.561     3.125     1.562      3.124
Quicksort       0.0  1.561  0.000  1.562   1.563   1.562    1.561    1.563    3.125     6.260     7.813    10.588    12.500    12.246     15.081
Timsort         0.0  0.000  0.000  1.564   3.125   4.050    5.794    6.249   14.807    26.431    35.528    46.946    59.019    62.699     75.044
```



Summary of expected time and space complexity of the chosen sorting algorithms

Time Complexity	**Insertion sort**	**Merge sort**	**Counting sort**	**Quicksort**	**Timsort**
Best case	Ω(n)	Ω(n log(n))	Ω(n+ k)	Ω(n log(n))	Ω(n)
Average case	Θ(n^2)	Θ(n log(n))	Θ(n+ k)	Θ(n log(n))	Θ(n log(n))
Worst case	O(n^2)	O(n log(n))	O(n+ k)	O(n^2)	O(n log(n))
Space Complexity	O(1)	O(n)	O(k)	O(log(n))	O(n)

Counting sort was the fastest function, followed by quicksort, Timsort and Merge sort. The slowest function, as expected was Insertion sort. Insertion sort works better with small lists. The running time jumps from 250 and it soars from 500.
Usually, counting sort has a linear time complexity, running faster than comparison-based sorting algorithms. The random arrays created for this project and that were passed to the chosen sorting algorithm to measure the running time, have a known range of elements. For that reason, counting sort worked faster. However, in the case of a big range of potential values in the array to be sort, counting sort would require extra space.
Quicksort was the second fastest sorting algorithm tested in this project. It is considered a fast algorithm with good asymptotic notation. However, it is not stable and its running time depends on the input data, and its worst-case performance (O(n^2)) is quite similar to the average case of insertion sort.
If the input data was real-world data (in other words, data that is not too randomly distributed), I assume Timsort would work so much faster compared to this project's results.
In the project, Timsort showed good running time for input size from 20 to 1250. However, the running time jumped from the input size 2500. The results also showed that for the input size equal to or higher than 3750, Timsort and merge sort had similar running time.

For Input size 10000, counting sort run 26 times faster than merge sort, 25 times faster than Timsort, 5 times faster than quicksort, and nearly 4000 times faster than insertion sort.

3.8.Final considerations

An ideal sorting algorithm would be stable, in-place, adaptive and would have O(n log(n)) for comparisons and O(n) for swaps. However, no sorting algorithm has all those properties. That way, to choose a sorting algorithm some things should be considered for applications, such as input size, amount of time required to move the data, and how much memory is available [34,35].

Some sorting algorithms like merge sort need space to sort, while algorithms, such as insertion sort, does not require extra space to sort, but is not the fastest option available.

Therefore, the requirements of the system and its limitations should be determined to decide the best algorithm to use [34,35].

4. REFERENCES

1.https://en.wikipedia.org/wiki/Sorting_algorithm

2. https://medium.com/@tssovi/comparison-of-sorting-algorithms-298fdf037c8f
3.https://www.freecodecamp.org/news/sorting-algorithms-explained/#:~:text=Why%20Sorting%20Algorithms%20are%20Important,structure%20algorithms%2C%20and%20many%20more.
4. http://faculty.tamuc.edu/dcreider/csci520/Note520/Note%206.htm
5. https://www.integralist.co.uk/posts/algorithmic-complexity-in-python/
6.https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7
7.https://inst.eecs.berkeley.edu/~cs10/fa15/discussion/04/Algorithmic%20Complexity%20Slides.pdf
8.https://stackabuse.com/big-o-notation-and-algorithm-analysis-with-python-examples/#:~:text=Algorithm%20Analysis%20with%20Big%2DO,by%20opening%20and%20closing%20parenthesis.
9.https://en.wikipedia.org/wiki/Time_complexity
10. https://www.kaggle.com/ozkanozturk/big-o-notation-code-complexity-python-basics
11. https://www.linkedin.com/pulse/big-o-notation-time-complexity-algorithm-vikas-kumar/
12.https://levelup.gitconnected.com/differentiating-logarithmic-and-linearithmic-time-complexity-976cd49c351b#:~:text=Linearithmic%20time%20(%20O(n%20log%20n)%20)%20is%20the,until%20input%20reaches%20advanced%20size.
13. https://people.duke.edu/~ccc14/sta-663/AlgorithmicComplexity.html
14.https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm
15. https://www.geeksforgeeks.org/sorting-terminology/
16. https://www.geeksforgeeks.org/stability-in-sorting-algorithms/
17.https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html?m=1
18. Khan Academy. Insertion sort. Available on https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort
19..https://www.geeksforgeeks.org/insertion-sort/
20. https://www.interviewbit.com/tutorial/insertion-sort-algorithm/
21. https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm
22. https://www.toptal.com/developers/sorting-algorithms/insertion-sort
23.https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html
24. https://www.journaldev.com/31541/merge-sort-algorithm-java-c-python#merge-sort-algorithm-flow

25. https://www.programiz.com/dsa/counting-sort

26. https://www.geeksforgeeks.org/counting-sort/

27. https://www.codesdope.com/course/algorithms-count-and-sort/

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

29.https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

30. https://www.educative.io/edpresso/nlogn-sorting-algorithms

31.https://www.chrislaux.com/timsort.html

32. https://en.wikipedia.org/wiki/Timsort#Minimum_size_.28minrun.29

33. https://hackernoon.com/timsort-the-fastest-sorting-algorithm-youve-never-heard-of-36b28417f399

33**.** https://www.interviewbit.com/tutorial/sorting-algorithms/#sorting-algorithms

34. https://www.toptal.com/developers/sorting-algorithms