

GMIT Higher Diploma

in

Data Analytics

Computational Thinking

with Algorithms

Project 2021

Laura Brogan

G00000329

Table of Contents

1. Project Introduction	2
1.2 Sorting	2
1.3 Complexity	2
1.4 Performance.....	3
1.5 In Place Sorting	3
1.6 Stable Sorting Algorithm	4
1.7 Comparison based or non-comparison based sorting	5
2. Sorting Algorithms	5
2.1 Selection Sort	5
2.2 Quick Sort.....	6
2.3 Counting Sort	8
2.4 Bubble Sort.....	9
2.5 Merge Sort	10
3. Implementation and Benchmarking	11
3.1 Selection Sort	12
3.2 Quick Sort.....	12
3.3 Counting Sort	13
3.4 Bubble Sort.....	13
3.5 Merge Sort	14
3.6 Results	15
4. Conclusion	17
5. References	19

Table of Figures

Figure 1. <i>Example of Selection Sort Algorithm process</i>	5
Figure 2. <i>Example of Quick Sort Algorithm process</i>	7
Figure 3. <i>Example of Counting Algorithm process</i>	8
Figure 4. <i>Example of Bubble Sort Algorithm process</i>	9
Figure 5. <i>Example of Merge Sort Algorithm process</i>	10
Figure 6. <i>Benchmarking Results</i>	16
Figure 7. <i>Average Case Time Running Algorithms</i>	16
Figure 8. <i>Detailed Average Times</i>	17
Figure 9. <i>Expected Benchmarking Results</i>	18
Figure 10. <i>Order of Performance from Benchmarking</i>	18

1. Project Introduction

This project will introduce five different sorting algorithms and will look at the results of the benchmarking process on the five different sorting algorithms. The five algorithms chosen for this project are: Selection Sort, Quick Sort, Counting Sort, Bubble sort and Merge Sort.

My submission contains this report and six Python files one for each of the sorting algorithms, and the benchmarking code. It contains six .PNG files four of these are diagrams to aid the explanation of the sorting algorithms, the other two are the results of benchmarking. There are two Excel files, one Excel file contains my rough work for the creation of the sorting algorithm diagrams. The second Excel file "Benchmarking" contains the benchmarking results in which I create a graph and a chart which are included as figure 8 and 9 in the project below.

1.2 Sorting

Sorting arranges a collection of items according to some pre-defined ordering rules.

A collection of items is deemed to be "sorted" if each item in the collection is less than or equal to its successor. (Mannion, 2017a)

Sorting algorithms reorganise large numbers of items into order such as alphabetical, shortest to longest, highest to lowest. The algorithms perform specific operations on an array of items to give an ordered array as output. (Wigmore, 2021)

Sorting is used for everyday tasks such as bank account statements by transaction number, tax details, phone book entries, library books.

Algorithms can be classified by comparison and non-comparison algorithms but there are other ways to classify algorithms. Good runtime efficiency in the best average and worst case. Preserving the order of already sorted input therefore looking at the stability of the sorting algorithm. Also looking at the memory used to run the sorting algorithm and if it is an in-place sorting algorithm. Suitability the strengths and weaknesses of the algorithm. (Mannion, 2017a)

Different algorithms suit different purposes but there is a way to analyze their performance looking at the complexity and performance of algorithms.

1.3 Complexity

To measure the efficiency of an algorithm we look at two parameters: Space and Time.

Space complexity: is the total memory space required for the execution by the program.

Time Complexity: is the number of times a particular instruction set is executed rather than the total time taken. As the total time taken will depend on external factors. (Geeksforgeeks.org, 2020)

Space and time complexity can define the effectiveness of an algorithm knowing how the algorithm works effectively can help us choose the way we program. (GreatLearning.com, 2020)

1.4 Performance

The performance of an algorithm can be affected by the type of computer used, the compiler used to run the code and the efficiency of the code itself. Performance is the practical measure of an algorithm's efficiency. Best performance occurs when the array is already sorted. Searching algorithms do not have the same speed on the same import but may have the same efficiency. performance is based on average case best case and worst-case efficiency.

Space is used when judging on algorithms performance does the algorithm require scratch space or can the array be sorted in place with no need for additional memory some algorithms such as keep sort never require extra space.

Stability is another criterion for measuring performance does the sort preserve the order of items with equal value most simple sorts do this but some sorts such as heapsort do not.

The chart below compares some sorting algorithms based on the different criteria. The algorithms with higher continuous terms appear first, though this is clearly an implementation-dependent concept and should only be taken as a rough model when picking among sorts of the same big-O efficiency. (Allain, 2019)

Sort	Time					
	Average	Best	Worst	Space	Stability	Remarks
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant	Stable	Always use a modified bubble sort
Modified Bubble sort	$O(n^2)$	$O(n)$	$O(n^2)$	Constant	Stable	Stops after reaching a sorted array
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$	Constant	Stable	In the best case (already sorted), every insert requires constant time
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Constant	Instable	By using input array as storage for the heap, it is possible to achieve constant space
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Depends	Stable	On arrays, merge sort requires $O(n)$ space; on linked lists, merge sort requires constant space
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Constant	Stable	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.

(Allain, 2019)

1.5 In Place Sorting

In place sorting algorithm changes the import without using any extra memory as the algorithm is run the input is usually overwritten by the output and therefore no additional space is needed for the operation.

In place algorithm may require a small amount of extra memory for its operation however the amount of memory required must not be dependent on the input size and should be constant.

In place algorithms are usually used in embedded systems that run in limited memory. They reduced the space requirements hugely and the algorithm time complexity increases in some cases. (Techiedelight.com, 2021)

1.6 Stable Sorting Algorithm

In a stable sorting algorithm objects with the same sort key appear in the output array in the same order as they do in the input array. Stable sort breaks ties between two objects with equal sort key by placing the first in the output array whichever element appears first in the input array.

Repeated elements in a stable sort algorithm are sorted in the same order that they appear in the input. With some kinds of data only part of the data is examined when determining the sort order this allows multiple different correctly sorted versions of the original array. If two items match as equal the relative order will be maintained so that if one came before the other in the input, it would also come before the other in the output. (Cormen, 2013)

If two elements ai and aj in the original unordered array are equal, it may be important to maintain their relative ordering in the sorted set. Example, if $i < j$, then the final location for $A[i]$ must be to the left of the final location for $A[j]$. Sorting algorithms that guarantee this property are stable. Unstable sorting algorithms do not preserve this property. (Mannion, 2017a)

Stable sort of flight information

- All flights which have the same destination city are also sorted by their scheduled departure time; thus, the sort algorithm exhibited stability on this collection.
- An unstable algorithm pays no attention to the relationships between element locations in the original collection (it might maintain relative ordering, but it also might not).

Destination	Airline	Flight	Departure Time (Ascending)	→	Destination (Ascending)	Airline	Flight	Departure Time
Buffalo	Air Trans	549	10:42 AM		Albany	Southwest	482	1:20 PM
Atlanta	Delta	1097	11:00 AM		Atlanta	Delta	1097	11:00 AM
Baltimore	Southwest	836	11:05 AM		Atlanta	Air Trans	872	11:15 AM
Atlanta	Air Trans	872	11:15 AM		Atlanta	Delta	28	12:00 PM
Atlanta	Delta	28	12:00 PM		Atlanta	Al Italia	3429	1:50 PM
Boston	Delta	1056	12:05 PM		Austin	Southwest	1045	1:05 PM
Baltimore	Southwest	216	12:20 PM		Baltimore	Southwest	836	11:05 AM
Austin	Southwest	1045	1:05 PM		Baltimore	Southwest	216	12:20 PM
Albany	Southwest	482	1:20 PM		Baltimore	Southwest	272	1:40 PM
Boston	Air Trans	515	1:21 PM		Boston	Delta	1056	12:05 PM
Baltimore	Southwest	272	1:40 PM		Boston	Air Trans	515	1:21 PM
Atlanta	Al Italia	3429	1:50 PM		Buffalo	Air Trans	549	10:42 AM

Reference: Pollice G., Selkow S. and Heineman G. (2016). Algorithms in a Nutshell, 2nd Edition. O' Reilly.

(Mannion, 2017a)

1.7 Comparison based or non-comparison based sorting

A comparison sort is a class of sorting algorithm which uses comparison procedures only to determine which of two elements should appear first in a sorted list. A sorting algorithm is comparison-based if the only way to obtain information about the overall order is by comparing a pair of elements at a time via the order \leq .

Bubble Sort, Insertion Sort and Selection Sort are simple sorting algorithms which fall into this category. A fundamental result in algorithm analysis is that no algorithm that sorts by comparing elements can do better than $n \log n$ performance in the average or worst cases. Non-comparison sorting algorithms (e.g., Bucket Sort, Counting Sort, Radix Sort) can have better worst-case times. (Mannion, 2017b)

2. Sorting Algorithms

2.1 Selection Sort

This is a simple comparison-based sort. Selection sort starts with the first element of the array and places this as the minimum value until it scans the entire array to find the minimum value and then swaps this to be the first element.

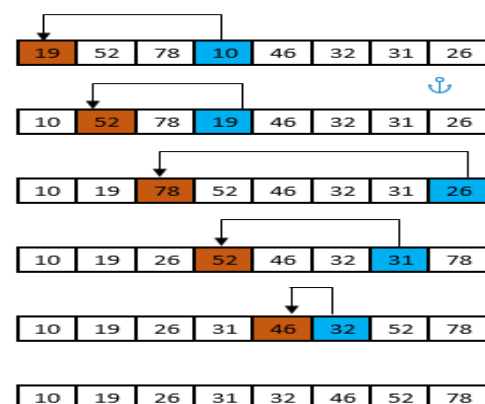
The time complexity of selection sort is $O(n^2)$. This is the total number of items in the list. Time complexity measures the number of iterations required to sort the list, the list is divided into two partitions the first is the sorted items and the second is the unsorted items.

In Figure 1. below we can see that the first item in the unsorted list is compared with all the values to the right-hand side to check if it is the minimum value if it is not the minimum value then its position is swapped with the minimum value as in the example below 10 is swapped with 19 as 10 is the minimum value in the array.

52 is taken next as the first item of the unsorted list: this is compared with all other items in the unsorted list 19 is found to be the minimum value and it therefore replaces 52.

This method is sustained until all items in the array are sorted.

Figure 1. Example of Selection Sort Algorithm process¹



¹ The Chart is by Laura Brogan adapted from W3resource.com, 2020. *Searching and Sorting Algorithm Exercise 11*. [Online] Available at: <https://www.w3resource.com/csharp-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-11.php> [Accessed May 2021].

TIME & SPACE COMPLEXITIES SELECTION SORT:

Worst case this is where the list providers is in descending order the algorithm performs the maximum number of iterations which is expressed as **[Big-O]: $O(n^2)$**

Best case this occurs when the provided list is already sorted the algorithm performs the minimum number of iterations which is expressed as **[Big-omega]: $O(n^2)$**

Average case this occurs when the list is in random order the average complexity is expressed as **[Big theta]: $O(n^2)$**

Space Complexity: $O(1)$

Sorting In Place: Yes

Stable: No

The selection sort is good for working on small lists, it does not require a lot of space for sorting, and it is not very good for working on large lists. (Guru99.com, 2021)

2.2 Quick Sort

This algorithm was introduced by C.A.R. Hoare in 1960, it is an efficient comparison-based sort. Quick Sort is a divide and conquer algorithm. It selects an element as pivot and partitions the given array around the chosen pivot. (George T. Heineman, 2016)

There are many different versions of the Quick Sort as different ways to pick that pivot can be chosen.

- i. Always pick the first element as pivot.
 - ii. Always pick the last element as pivot (implemented below)
 - iii. Pick a random element as pivot.
 - iv. Pick median as pivot.
- (Geeksforgeeks.org, 2021a)

To explain the Quick Sort algorithm in the sample array in Figure 2. takes the last element as the pivot. Anything lower than the pivot is separated to the left in this case 1,2,6 and 5.

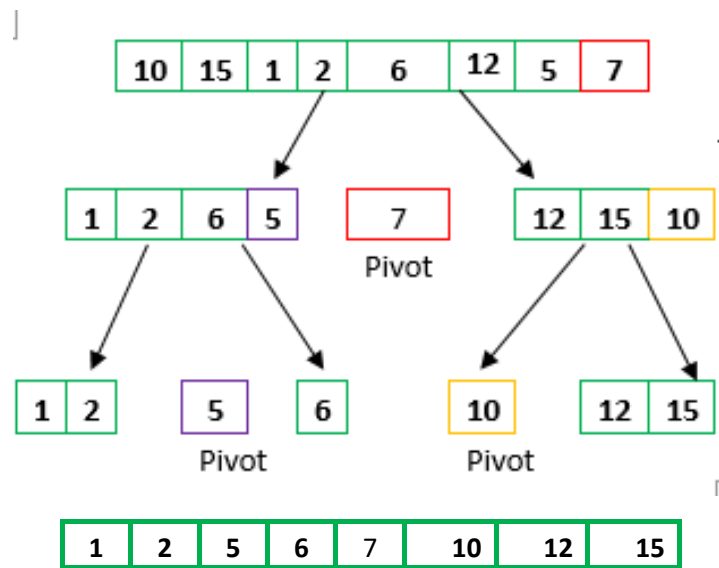
Everything higher than the pivot is separated to the right in this case 12,15 and 10.

The pivot element is placed between the two partitions; this will also be its final place.

For each individual array, a new pivot is picked in the example below on the left-hand side 5 is taken as the pivot and the array is broken down to everything below 5 1 and 2 to the left and everything above 5, 6 to the right.

The secondary array of 12, 15 and 10 we take 10 as the pivot 10 is moved to the right in this case 12 and 15.

The array is then reassembled in a sorted order.

Figure 2. Example of Quick Sort Algorithm process²**TIME & SPACE COMPLEXITIES QUICK SORT:** (Geeksforgeeks.org, 2021a)

Worst Case: $O(n^2)$. arises when the partition process constantly picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case will occur when the array is already sorted in increasing or decreasing order.

Best Case: $O(n \log n)$ occurs when the partition process always picks the middle element as pivot.

Average case: $O(n \log n)$

Space Complexity: $O(n)$

Sorting In-Place: Yes, as it uses extra space only for storing recursive function calls but not for manipulating the input.

Stable: No. The default implementation is not stable. Any sorting algorithm can be made stable by contemplating indexes as comparison parameters.

Quicksort can be implemented in various ways by altering the selection of pivot so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

(Geeksforgeeks.org, 2021a)

² The Chart is by Laura Brogan adapted from Dev.to, 2020. *Quick Sort In Ruby*. [Online] Available at: <https://dev.to/mwong068/quick-sort-in-ruby-2302> [Accessed May 2021].

2.3 Counting Sort

This is a non-comparison sort proposed by Harold H. Seward in 1954. (Mannion, 2017c)

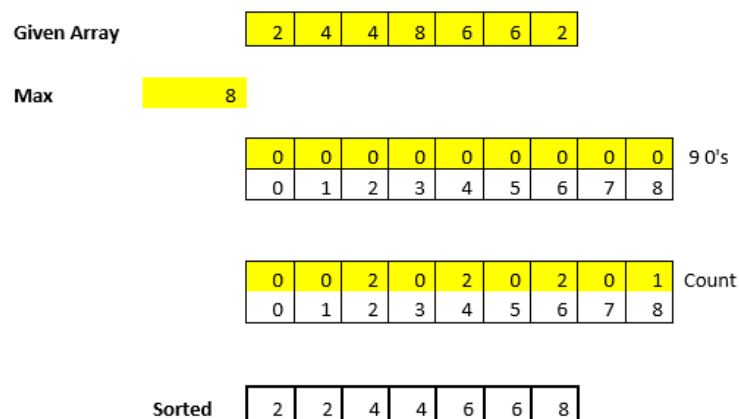
This algorithm works on the basis of counting the number of occurrences of each element in an array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array. (Programiz.com, 2021)

Looking at the Figure 3. It can be seen that in the given array, with 8 being the maximum element from the array.

The next line initialises an array of the max 8 plus 1 so 9 elements with all elements zero. The next line counts the occurrence of each element, so start with zero and one that is not in the array, so it stays at zero. Element 2 is in the array twice, element 3 is zero, element 4 is in the array twice, element 5 zero element 6 occurs twice, element 7 is not in the array and element 8 occurs once.

Then transpose the count to list out each occurrence of the element in the array to give the sorted output as below.

Figure 3. Example of Counting Algorithm process³



TIME & SPACE COMPLEXITIES COUNTING SORT: (Programiz.com, 2021)

Worst Case: $O(n+k)$

Best Case: $O(n+k)$

Average case: $O(n+k)$

In all the above cases, the complexity is the same because no matter how the elements are placed in the array, the algorithm goes through $n+k$ times.

Space Complexity: $O(\max)$ Larger the range of elements, larger is the space complexity.

³ The Chart is by Laura Brogan adapted from Programiz.com, 2021. *Counting Sort*. [Online] Available at: <https://www.programiz.com/dsa/counting-sort> [Accessed May 2021].

Sorting In-Place: No. Extra space is used for sorting the array elements.

Stable: Yes. As components with the same sort key appear in the output array in the same order as they do in the input array. (Cormen, 2013)

Counting sort, sorts the elements of an array by counting the number of incidents of each unique component in the array. The count of each unique component is stored in an auxiliary (count) array and the sorting is done by mapping the count as an index of the auxiliary (count) array. Counting sort works with whole numbers, but it can be modified to sort negative integers also. (Gupta, 2020)

2.4 Bubble Sort

A simple comparison-based sort in which each pair of adjacent elements is compared, and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets. (Tutorialspoint.com, 2021)

As can be seen in Figure 4. the algorithm compares each element except the last one with its neighbour to the right.

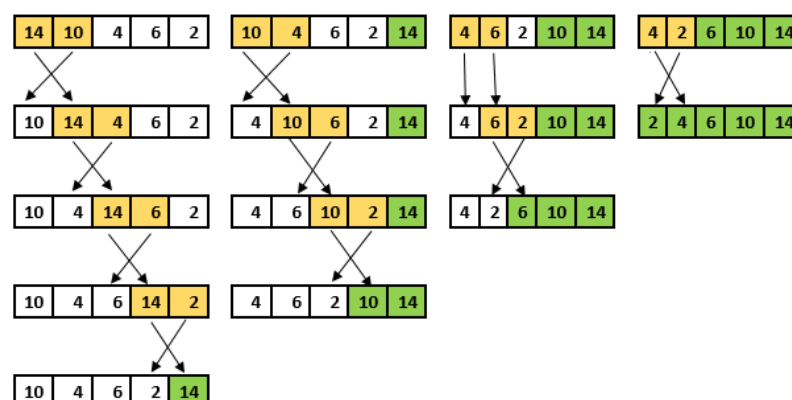
If they are not in the correct order, they are swapped this then puts the largest element at the very end and this is its correct and final place as shown in green below.

Next each element except the last two are compared to the right, if they are not in the correct order, they are swapped, and this then puts the second largest element in their correct and final place again as seen in green 10 & 14 below.

This process is continued until there are no unsorted elements on the left. In the example below this process had to be completed 4 times to get the list sorted.

This type of sorting can be useful for data that is nearly sorted already.

Figure 4. Example of Bubble Sort Algorithm process⁴



TIME & SPACE COMPLEXITIES OF BUBBLE SORT: (Studytonight.com, 2021)

⁴ The Chart is by Laura Brogan adapted from Mannion, P., 2017b. *Sorting Algorithms Part2*. [Online]

Available at: https://learnonline.gmit.ie/pluginfile.php/297434/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf [Accessed May 2021].

Worst Case: [Big-O]: $O(n^2)$

Best Case: [Big-omega]: $O(n)$ it is when the list is already sorted.

Average case: [Big-theta]: $O(n^2)$

Space Complexity: $O(1)$ because only a single additional memory space is required i.e., for temp variable.

Sorting In-Place: Yes, as it only requires **$O(1)$** constant space, we can say that it is an in-place algorithm, which operates directly on the inputted data. (Joshi, 2017)

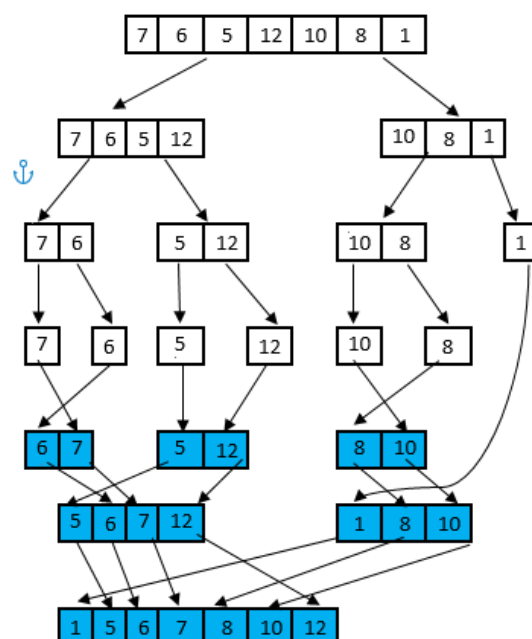
Stable: Yes

The simplicity of the Bubble Sort algorithm is one of its main advantages. With every complete iteration the largest element in the given array bubbles up to the last place or the highest index, just like a water bubble rises to the water surface therefore it became known as bubble sort. Sorting happens by moving through all the elements one-by-one and comparing it with the element adjacent and then swapping if required. (Studytonight.com, 2021)

2.5 Merge Sort

An efficient comparison-based sort. Merge Sort is like Quick Sort in that it is a divide and conquer algorithm. As can be seen in the array below the array is divided in half and then divided again and again until the array is recursively divided the size becomes one. Once we have individual items the merge process can come into action and start merging arrays back together until the complete array is merged into a sorted array as can be seen below.

Figure 5. Example of Merge Sort Algorithm process ⁵



⁵ The Chart is by Laura Brogan adapted from Mannion, P., 2017c. *Sorting Algorithms Part 3*. [Online]

Available at: https://learnonline.gmit.ie/pluginfile.php/297435/mod_resource/content/0/09%20Sorting%20Algorithms%20Part%203.pdf [Accessed May 2021].

TIME & SPACE COMPLEXITIES OF MERGE SORT: (Geeksforgeeks.org, 2021b)**Worst Case:** $\theta(n \log n)$ **Best Case:** $\theta(n \log n)$ **Average case:** $\theta(n \log n)$ **Space Complexity:** $O(n)$ **Sorting In-Place:** No**Stable:** Yes

Merge sort uses a new list to store partial results and therefore requires an additional linear amount of space this is a disadvantage to this sorting algorithm, compared with other sorting methods.

Merge sort can be one of the most time efficient sorting algorithms and one of the easiest to describe recursion is the reason for this, it makes the algorithm easy to describe and it provides a clean mechanism for splitting the problem into two smaller problems half the size which is the root of the algorithms time performance. (Harel, 2004)

3. Implementation and Benchmarking

Benchmarking or a posteriori analysis is an empirical method to compare the relative performance of algorithm implementations. Investigational (e.g., running time) data may be used to confirm theoretical or a priori analysis of algorithms. Various hardware and software factors such as system architecture, CPU design, choice of operating system, background processes, energy saving and performance enhancing technologies etc. can affect running time. Therefore, it is prudent to conduct various statistical runs using matching experimental setup, to ensure that your set of benchmarks are representative of the performance expected by an “average” user. (Mannion, 2019)

The hardware and software used for running the comparisons in this project are as follows:

LAPTOP- 9 J 3223n6

Manufacturer: HP

Model: HP Pavilion Laptop 14-BK0XX

Category: Notebook PC

Processor Intel(R) Core (TM) i3-7100U CP@ 2.40GHz

Installed memory (RAM): 8.00GB (7.89 GB usable)

System type: 64-bit Operating System, x 64 based processor.

The algorithms are implemented using the Python programming language, version 3.7.7.

Full Python code files are enclosed separately with this project. Included in the sections below are code snippets, no additional comments given as the code is commented throughout.

3.1 Selection Sort

```
#Resource used: https://stackabuse.com/selection-sort-in-python/
def selectionsort(arr):
    # i is the amount of items to be sorted
    for i in range(len(arr)-1):
        #Start with the first item in the array and assume it is the lowest
        min_index = i
        # j is used to loop through the remaining items in the array.
        for j in range(i+1, len(arr)-1):
            #If a lower item is found than that of min_index above min_index is updated
            if arr[j] < arr[min_index]:
                min_index = j
        #When the lowest item of the array is found swap it with the first item in the array
        arr[i], arr[min_index] = arr[min_index], arr[i]
```

3.2 Quick Sort

```
#Resource used: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html
def quicksort(array):
    # This is calling a recursive function that takes in the array
    quicksorthelper(array,0,len(array)-1)

#This function helps with the sorting process, if the first items is < 1 the item is sorted if not it partitions the array and recursively sorts.
def quicksorthelper(array,first,last):
    if first<last:
        #finding where to split the array
        splitpoint = partition(array,first,last)
        #call the function on the two halves of the array
        quicksorthelper(array,first,splitpoint-1)
        quicksorthelper(array,splitpoint+1,last)

#Function using pivot element to partition the array.
def partition(array,first,last):
    pivotvalue = array[first]

    #This function takes the first element as pivot, places the pivot element at the correct position in sorted array and places all higher
    #than the pivot to the left of the pivot and all smaller elements to right of the pivot.
    leftmark = first+1
    rightmark = last

    done = False
    while not done:
        #Moving the first point right until the first element is greater pivot is found.
        while leftmark <= rightmark and array[leftmark] <= pivotvalue:
            leftmark = leftmark + 1
        #Moving the second point right until the first element is less than pivot.
        while array[rightmark] >= pivotvalue and rightmark >= leftmark:
            rightmark = rightmark - 1
        #Stop when when cross over occurs
        if rightmark < leftmark:
            done = True
            #Swap the values that are out of place
        else:
            temp = array[leftmark]
            array[leftmark] = array[rightmark]
            array[rightmark] = temp
        #Swap with the pivot value
        temp = array[first]
        array[first] = array[rightmark]
        array[rightmark] = temp

    return rightmark
```

3.3 Counting Sort

```
#Resource used: https://www.programiz.com/dsa/counting-sort
def countingsort(array):
    size = len(array)
    output = [0] * size

    #Start count array
    count = [0] * 10000

    #Put the count of each elements in count array
    for i in range(0, size):
        count[array[i]] += 1

    #Keep cumulative count
    for i in range(1, 10000):
        count[i] += count[i - 1]

    #Find the index of each element of the original array in count array
    #place the elements in output array
    i = size - 1
    while i >= 0:
        output[count[array[i]] - 1] = array[i]
        count[array[i]] -= 1
        i -= 1

    #Copy the sorted elements into original array
    for i in range(0, size):
        array[i] = output[i]
```

3.4 Bubble Sort

```
#Resource used: https://realpython.com/sorting-algorithms-python/#the-bubble-sort-algorithm-in-python
def bubblesort(array):
    n = len(array)

    for i in range(n):
        # If there's nothing left to sort terminate the sort function
        already_sorted = True
        #Look at each item of the list one by one and compare it to the adjacent item.
        #Each iteration reduces the amount of the array that is looked at as the remaining items are already sorted.
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                #If the item you are looking at is larger than the adjacent value then they are swapped.
                array[j], array[j + 1] = array[j + 1], array[j]

                #As swaping two elements, mark already_sorted to false so the algorithm doesn't finish prematurely.
                already_sorted = False

        #If no swaps in the last iteration the array is already sorted and can be terminated.
        if already_sorted:
            break
```

3.5 Merge Sort

```
def mergesort(array):
    #print is used in testing to see the breakdown of sorting process
    print("Splitting ",array)
    #This recursive algorithm splits the list in half
    if len(array)>1:
        mid = len(array)//2
        lefthalf = array[:mid]
        righthalf = array[mid:]

        mergesort(lefthalf)
        mergesort(righthalf)

        i=0
        j=0
        k=0
        #It continues to divide the array until each item is individual
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] <= righthalf[j]:
                array[k]=lefthalf[i]
                i=i+1
            else:
                array[k]=righthalf[j]
                j=j+1
            k=k+1
        #Once it is divided it is then merged into the correct order as it look at each individual sub array
        while i < len(lefthalf):
            array[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            array[k]=righthalf[j]
            j=j+1
            k=k+1
    #print is used in testing to see the breakdown of sorting process
    print("Merging ",array)
```

3.6 Results

The Python file *benchmarking.py* contains the code that was used to calculate the results of benchmarking. This code is extensively commented to make it readable to all users, below I have detailed some of the higher-level details from the code.

The following libraries were imported to run benchmarking.

time: which is used for timing each of the sorting algorithms.

numpy: which is used for generating lists of random integers.

pandas: which is used to create a data frame to store the output from the trials.

matplotlib.pyplot: which is used to create a graphical representation of the benchmark tests.

Each of the sorting algorithms, Selection Sort, Quick Sort, Counting Sort, Bubble sort and Merge Sort are called taking an input array as an argument and returning a sorted array.

To calculate the average timing the following are used.

array_create(): Numpy package is used to generate random integers between 1 and 1000.

timer(): This function takes the start and end time of running the algorithm.

average_time(): This function uses the number of runs, test size and sorting algorithm as inputs and calculates the average time for running each of the algorithms.

Then the trials are run, and the output is formatted to make it human readable.

algo_trial: This takes each algorithm and test size, creates a random array of the specified size, and uses the average_time function to return the average time in milliseconds formatted to 3 decimal places.

col_create(): This function creates columns for the data frame for each test size.

df_create: This function creates a pandas data frame based on a data dictionary.

results_plot: This plots the relative performance of each algorithm using matplotlib.pyplot using a pandas data frame based on the results based on the data dictionary.

The following are the functions used in creating the user interface.

sorts: this is a list of the 5 selected sorting algorithms.

algorithms: this is a list of the 5 sorting algorithm functions.

N_trial: list the number of various test/trial sizes for testing of each algorithm.

trial: this is the function that starts the data frame from the data dictionary.

In the final section of *benchmarking.py* is the define `main()` function which allows the user to interact via the command line. The user is asked to enter one of the following values:

graph: to display the results graphically using matplotlib.pyplot.

table: to display the results as a formatted grid on the command line.

quit: to stop the application.

The Figure 6. shows the results of the benchmarking exercise as generated using the *benchmarking.py* Python application. The table shows the average time in milliseconds across the ten runs for each sorting algorithm using the different size inputs from 100 up to 10,000.

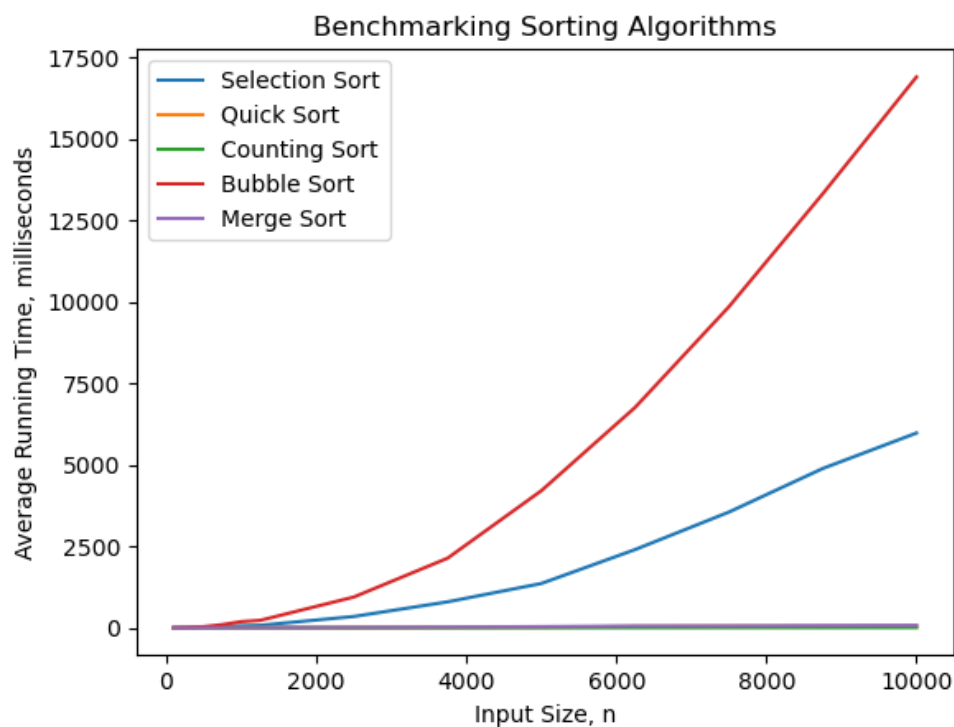
It can be seen from Figure 6. that Bubble sort is the worst performing algorithm overall as it takes an average of over 16 seconds to sort a random list of 10,000 numbers. Selection sort fared slightly better at over 5 seconds. It can also be seen that counting sort was the best performing algorithm taking 10 milliseconds. The counting sort algorithm was the slowest algorithm when run with $n=100$ but was the quickest overall.

Figure 6. Benchmarking Results

size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Selection Sort	0.798	3.887	16.755	38.286	67.021	79.594	349.873	798.764	1362.266	2402.187	3552.303	4887.550	5975.398
Quick Sort	0.598	0.691	1.396	2.596	4.275	3.591	8.678	14.966	21.644	41.096	50.165	41.994	55.654
Counting Sort	1.795	2.893	3.490	3.794	3.890	5.087	4.085	4.882	6.282	6.290	7.482	8.074	10.376
Bubble Sort	1.097	9.967	38.303	101.431	192.391	230.889	946.368	2138.575	4208.350	6762.825	9848.669	13309.809	16894.900
Merge Sort	0.698	1.692	3.391	4.884	6.082	7.783	15.065	24.331	37.595	60.545	62.522	72.901	78.586

The results are also displayed in Figure 7. outlining the running time of each of the algorithms chosen in the average case scenario.

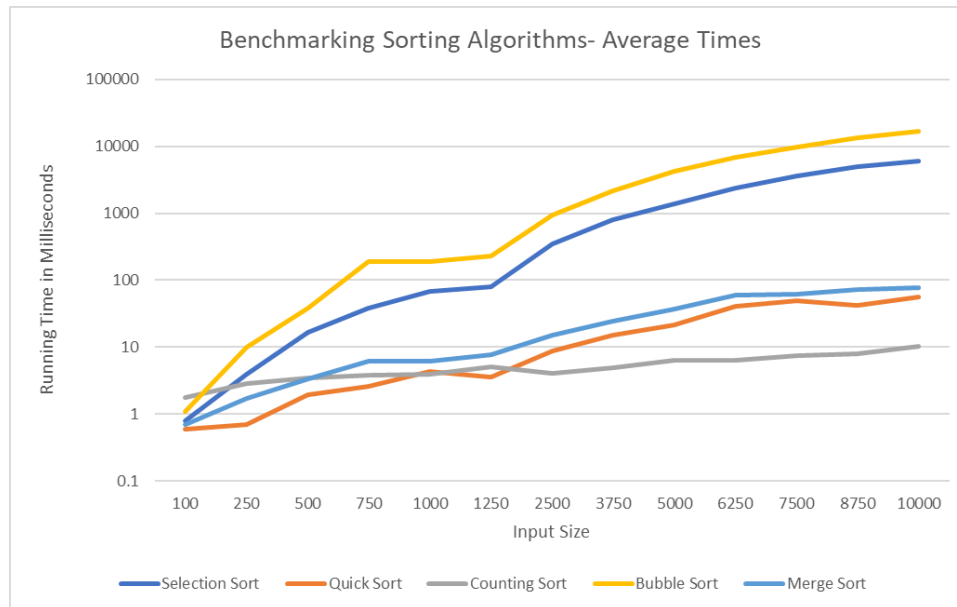
Figure 7. Average Case Time Running Algorithms



It is hard to distinguish counting sort, quicksort and merge sort on the above graph given the scale of the Y axis which is dominated by the very slow times for bubble sort. Figure 7. graph shows us the bubble sort increases at a fast rate and it is followed by selection sort.

A new graph is created Figure 8. to view the outcome of the benchmarking which makes it easier to distinguish the different sorting algorithms by changing the scale of the XY axis. The sorting algorithms can be clearly seen on one graph and confirms above analysis that bubble sort and selection sort are the worst performing. It can be seen that Quick Sort and Merge Sort perform in a similar fashion although Quick Sort is faster than Merge Sort.

Figure 8. Detailed Average Times



Performance of sorting algorithms is not easy to benchmark as different algorithms perform differently on different sets of data and on different. Based on my results in this project the counting sort algorithm is the best performing algorithm followed then by quicksort merge sort selection sort and bubble sort being the worst performing algorithm. Note that of the five algorithms looked at in this project the counting sort, bubble sort and merge sort were stable algorithms while the selection sort and quicksort were unstable algorithms.

The best performing algorithm was the counting sort it was a non-comparison sort with a potential runtime of $N + K$ in the best, worst and average case each item has a non-negative integer.

4. Conclusion

This project looked at five sorting algorithms the times were measured and compared to each other and to expected results. The input arrays contained random integers, generated using Python NumPy (`np.random.randint`). The array sizes I tested were `n_trial = [100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000]`. The Python time module was used to calculate the running time of the algorithms in milliseconds by measuring the time before and after the call of the sorting algorithm. The tests were run over 10 times on each algorithm and the average was taken. The actual times varied depending on factors such as the machine on which they were implemented. The random arrays generated for each contained random integers between zero and 1,000 the results in this benchmarking project showed the counting sort algorithm as being the fastest it should be noted that the

performance of the counting algorithm does decrease when the values to be sorted are larger.


It is clear the counting sort is a far faster sorting algorithm than the two simple sorting algorithms selection sort and bubble sort, merge sort and quicksort have similar linear arithmetic growth rates. The results of the benchmarking here are in line with the results expected given the details as outlined earlier for each of the five sorting algorithms studied and summarized in the below table.

Figure 9. Expected Benchmarking Results

Algorithm	Best Case	Worst Case	Average Case	Space Complexity
Selection Sort	n^2	n^2	n^2	1
Quick Sort	$n \log n$	n^2	$n \log n$	n
Counting Sort	$n+k$	$n+k$	$n+k$	max
Bubble Sort	n	n^2	n^2	1
Merge Sort	$n \log n$	$n \log n$	$n \log n$	$O(n)$

Below are the sorting algorithms listed in order of performance from best to worst, based on the results of my benchmarking.

Figure 10. Order of Performance from Benchmarking

Algorithm	Performance Order
Counting Sort	<div>Best</div>  <div>Worst</div>
Quick Sort	
Merge Sort	
Selection Sort	
Bubble Sort	

5. References

Allain, A., 2019. *Sortcomp*. [Online]

Available at: <https://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html>

[Accessed May 2021].

Cormen, T. H., 2013. *Algorithms Unlocked*, Cambridge, Massachusetts: The MIT Press.

Dev.to, 2020. *Quick Sort In Ruby*. [Online]

Available at: <https://dev.to/mwong068/quick-sort-in-ruby-2302>

[Accessed May 2021].

Geeksforgeeks.org, 2020. *Time Complexities of sorting Algorithms*. [Online]

Available at: <https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/>

[Accessed May 2021].

Geeksforgeeks.org, 2021a. *Quick Sort*. [Online]

Available at: <https://www.geeksforgeeks.org/quick-sort/>

[Accessed May 2021].

Geeksforgeeks.org, 2021b. *Merge Sort*. [Online]

Available at: <https://www.geeksforgeeks.org/merge-sort/>

[Accessed May 2021].

George T. Heineman, G. P. a. S. S., 2016. Algorithms in a Nutshell. In: Second, ed. *A PRACTICAL GUIDE*. United States of America.: O'Reilly, p. 69.

GreatLearning.com, 2020. *Why is time complexity essential*. [Online]

Available at: <https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/>

[Accessed May 2021].

Gupta, V., 2020. *Visualizing Designing and Aanalyzing The Counting Sort Algorithm*. [Online]

Available at: <https://levelup.gitconnected.com/visualizing-designing-and-analyzing-the-counting-sort-algorithm-834ba84c151e>

[Accessed 2021].

Guru99.com, 2021. *Selection Sort Aalgorithm*. [Online]

Available at: <https://www.guru99.com/selection-sort-algorithm.html>

[Accessed May 2021].

Harel, D. w. Y. F., 2004. *Algorithmics The Spirit of Computing*. Third ed. London: Pearson.

Joshi, V., 2017. *Bubbling Up With Bubble Sorts*. [Online]

Available at: <https://medium.com/basecs/bubbling-up-with-bubble-sorts-3df5ac88e592#:~:text=Since%20it%20only%20requires%20O,relative%20order%20of%20the%20elements.&text=Since%20it%20only%20requires%20O,relative%20order%20of%20the%20elements.>

[Accessed May 2021].

Mannion, P., 2017a. *Sorting Algorithms Part1*. [Online]

Available at:

https://learnonline.gmit.ie/pluginfile.php/297433/mod_resource/content/0/07%20Sorting%20Algorithms%20Part%201.pdf

[Accessed May 2021].

Mannion, P., 2017b. *Sorting Algorithms Part2*. [Online]

Available at:

https://learnonline.gmit.ie/pluginfile.php/297434/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf

[Accessed May 2021].

Mannion, P., 2017c. *Sorting Algorithms Part 3*. [Online]

Available at:

https://learnonline.gmit.ie/pluginfile.php/297435/mod_resource/content/0/09%20Sorting%20Algorithms%20Part%203.pdf

[Accessed May 2021].

Mannion, P., 2019. *Benchmarking Algorithms in Python*. [Online]

Available at:

https://learnonline.gmit.ie/pluginfile.php/297438/mod_resource/content/0/11%20Benchmarking%20in%20Python.pdf

[Accessed May 2021].

Programiz.com, 2021. *Counting Sort*. [Online]

Available at: <https://www.programiz.com/dsa/counting-sort>

[Accessed May 2021].

Studytonight.com, 2021. *Bubble Sort*. [Online]

Available at: <https://www.studytonight.com/data-structures/bubble-sort#:~:text=The%20main%20advantage%20of%20Bubble,the%20list%20is%20already%20sorted.>

[Accessed May 2021].

Techiedelight.com, 2021. *In Place vs Out Of Place Algorithms*. [Online]

Available at: <https://www.techiedelight.com/in-place-vs-out-of-place-algorithms/>

[Accessed May 2021].

Tutorialspoint.com, 2021. *Bubble Sort Algorithm*. [Online]

Available at: https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm

[Accessed May 2021].

W3resource.com, 2020. *Searching and Sorting Algorithm Exercise 11*. [Online]

Available at: <https://www.w3resource.com/csharp-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-11.php>

[Accessed May 2021].

Wigmore, I., 2021. *Sorting algorithm*. [Online]

Available at: <https://whatis.techtarget.com/definition/sorting-algorithm>

[Accessed May 2021].