

Contents

FOREWORD	3
PYTHON APPLICATION:	5
PART 1 INTRODUCTION.....	6
FOREWORD	6
The concept of sorting and sorting algorithms.....	7
What is sorting?	7
Why does data need to be sorted?.....	7
What are sorting algorithms?	7
How is sorting done?	8
Time Complexity, Order of Growth, O-notation	8
Demonstration of Iteration	9
Demonstration of Recursion	10
Order of growth	11
Space Complexity.....	11
Relevance of performance,	12
In-place sorting,	12
Stable Sorting.....	13
Comparison-based sorting.....	13
Non-comparison-based sorts, etc.,	13
PART 2 SORTING ALGORITHMS.....	14
Bubble Sort – 1962 – A simple comparison based sort	15
About Bubble Sort.....	15
Bubble Sort Diagram	16
Bubble Sort Space Usage	17
Bubble Sort Time Complexity.....	18
Bubble Sort Code https://realpython.com/sorting-algorithms-python/#the-bubble-sort-algorithm-in-python	20
Merge Sort	21
About Merge Sort	21
Merge Sort Diagram.....	21
Merge Sort Space Usage	22
Merge Sort Time Complexity	24
Merge Sort Code https://realpython.com/sorting-algorithms-python/#the-merge-sort-algorithm-in-python	25
Radix Sort	27
Christopher’s comments on Radix Sort	27

About Radix Sort	28
Radix Sort Diagram.....	29
Radix Sort Space Usage	30
Radix Sort Time Complexity	31
Radix Sort Code - https://www.programiz.com/dsa/radix-sort	32
Quick Sort.....	33
About Quick Sort.....	33
Quick Sort Diagram - https://en.wikipedia.org/wiki/Quicksort	34
Quick Sort Space Usage.....	35
Quick Sort Time Complexity.....	35
Quick Sort Code https://realpython.com/sorting-algorithms-python/#the-quicksort-algorithm-in-python	36
Tim Sort.....	37
Tim Sort.....	37
Tim Sort Diagram	38
Tim Sort Space Usage.....	39
Tim Sort Time Complexity	39
Tim Sort Code https://realpython.com/sorting-algorithms-python/#the-timsort-algorithm-in-python	40
PART 3 – IMPLEMENTATION AND BENCHMARKING	43
Implementation and design of the python code used for benchmarking.....	45
STEP 1 – Setup variables for containing benchmark results	45
STEP 2 – Generate random numbers.....	47
STEP 3 – Create function for generating random arrays	47
STEP 4 – Create container function to store the algorithms for testing.....	48
STEP 5 – Edits of individual algorithms	48
STEP 6 – Reporting	50
Testing unsorted data in the benchmark.....	53
Testing already sorted data in the benchmark.....	53
Testing data sorted in reverse in the benchmark	53
Benchmarking computer specifications.....	54
Printing the operating system.....	54
Ensuring Data Integrity	54

FOREWORD

In my research I found there is often much assumed understanding in blog posts and even basic books on algorithm design, of a reader's ability to be able to understand and read certain mathematical concepts before jumping into the world of sorting algorithms.

FOREWORD ABOUT PYTHON APPLICATION

I have been programming in Python for approximately one year.

Using basic functions and while loops I was able to link up the random array generator example provided for this assignment to the algorithms that I copy and pasted off the internet.

There is no original algorithm design on my part, I have taken the original coding in its entirety for Quicksort¹, Mergesort², Timsort³, Bubble Sort⁴, Radix Sort⁵ from the sources below.

Algorithm code sources

¹ <https://realpython.com/sorting-algorithms-python/#the-quicksort-algorithm-in-python>

² <https://realpython.com/sorting-algorithms-python/#the-merge-sort-algorithm-in-python>

³ <https://realpython.com/sorting-algorithms-python/#the-timsort-algorithm-in-python>

⁴ <https://realpython.com/sorting-algorithms-python/#the-bubble-sort-algorithm-in-python>

⁵ <https://www.programiz.com/dsa/radix-sort>

FOREWORD ABOUT PART 1 INTRODUCTION

Although I have tried to cover the core topics requested for this assignment, my best attempt to answer the requested information in many cases, has been to collate and quote the most easily understandable text and diagrams that I found easiest to read from a layman's perspective. Even then it was sometimes difficult for me to completely assimilate the full meaning of all of it, to put all sources and diagrams into my own words and drawings. I do realize that quoting others texts and diagrams does not fully satisfy the requirement of using own words and diagrams, but I have done this to ensure that there are no gaps in the requested topics.

Even searching across stackoverflow questions and answers, youtube video presentations, there seems to be a massive lack of easily accessible information available in layman's terms about Big O Notation, without a lot of reading and sifting for relevant information.

Trying to understand Big O Notation and some of the mathematical terminology used by data scientists for properly explaining the strengths and weaknesses of sorting algorithms and the time and space complexities of each one, can be extremely difficult. You'll come across Omegas and Thetas, and many other things well beyond a foundation level of maths.

FOREWORD ABOUT PART 2 SORTING DIAGRAMMS

For this assignment, I will be providing reviews, diagrams and source code of Quicksort, Mergesort, Timsort, Bubble Sort, Radix Sort, which I have also integrated into my Python benchmarking test. Where possible I have annotated and put information in my own words, otherwise I have provided a link to the source of the information in the footnotes.

Sometimes providing the history of who invented the algorithms and why the algorithms were designed in some cases sheds some light on their practical uses and strengths, but for completeness on answering the strengths and weaknesses of each algorithm, I have included mathematical extracts from <http://bigocheatsheet.com/> the lecturer and others will understand the significance.

PYTHON APPLICATION:

My python application is built around testing four algorithms taken from <https://realpython.com/sorting-algorithms-python/> , Bubble sort, Merge Sort, Quick Sort and Tim Sort.

I have taken the Radix sort algorithm from <https://www.programiz.com/dsa/radix-sort>

These algorithms and their original comments are pasted in their entirety between my benchmarking code which passes the arrays, and runs reports on the arrays.

My python application can be found in the zip folder and its draft and final versions found on my github folder

<https://github.com/g00387822/AlgorithmAssignment>

PART 1 INTRODUCTION

FOREWORD

To set the scene, I will quote from *The Art of Computer Programming: Volume 3: Sorting and Searching* (2nd Edition).

'Computer manufacturers of the 1960's estimated that more than 25 percent of the running time of their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either

- 1. there are many important applications of sorting, or*
- 2. many people sort when they shouldn't, or*
- 3. inefficient sorting algorithms have been in common use.'*⁶

Although computing has changed since the 1960s, John Cook Consulting writes, not *'but not so much that sorting has gone from being extraordinarily important to unimportant.'*⁷

There is so much data in the world today, that there is a growing career demand for data scientists and data analysts to be able to sort it. Over 90 percent of the data in the world has been generated since 2016. "The amount of data we produce every day is truly mind-boggling. There are 2.5 quintillion bytes of data created each day at our current pace, but that pace is only accelerating with the growth of the Internet of Things (IoT)."⁸

The LinkedIn Workforce Report maintains that, in the USA, demand for these professional figures has grown sixfold compared to five years ago, and data analysts will continue to be the most sought after profiles over the next five years. This is further confirmed by IBM, which claims that the annual demand for data scientists, data developers and data engineers will lead to 700,000 new recruitments by 2020.⁹ The biggest challenge for data scientists is to be able to sort data of a colossal scale. According to Domo's Data Never Sleeps 5.0 report, these are numbers generated every minute of the day: Snapchat users share 527,760 photos; Users watch 4,146,600 YouTube videos; 456,000 tweets are sent on Twitter, Instagram users post 46,740 photos.¹⁰

⁶ [The Art of Computer Programming: Volume 3: Sorting and Searching \(2nd Edition\) 2nd Edition](#)

⁷ <https://www.johndcook.com/blog/2011/07/04/sorting/>

⁸ <https://blazon.online/data-marketing/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/>

⁹ <https://www.morningfuture.com/en/article/2018/02/21/data-analyst-data-scientist-big-data-work/235/>

¹⁰ <https://blazon.online/data-marketing/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/>

Different sorting algorithms are not only necessary because of speed, but also because of hardware processing space capabilities.

O'Reilly discusses some of the challenges with creating sorting algorithms.

"The primary issue with this code is that it is simply impossible to create a contiguous array to contain a large collection of elements. You can try to increase the heap space available to your Java virtual machine, but eventually the computer on which you are running will exhaust its available memory. So how is it possible to deal with extremely large data sets? You will need to develop techniques that manage the transfer of data from external storage (such as a hard disk) into main memory (what is commonly called RAM). In the early days of computing, main memory was measured in kilobytes (not gigabytes!) and programmers learned how to work within these constraints. In this era of "Big Data" where data can be measured in terabytes and petabytes, even modern programmers have to make some fundamental adjustments."¹¹

The concept of sorting and sorting algorithms

What is sorting?

'Sorting is any process of arranging items systematically, and has two common, yet distinct meanings: ordering: arranging items in a sequence ordered by some criterion; categorizing: grouping items with similar properties.'¹²

Why does data need to be sorted?

'Sorting is particularly helpful in the context of computer science for two reasons: From a strictly human-friendly perspective, it makes a single dataset a whole lot easier to read. It makes it easier to implement search algorithms in order to find or retrieve an item from the entire dataset.'¹³

What are sorting algorithms?

The word algorithm comes from the name of the 9th century Persian and Muslim mathematician Abu Abdullah Muhammad ibn Musa Al-Khwarizmi, he was mathematician, astronomer and geographer during the Abbasid Caliphate, a scholar in the House of Wisdom in Baghdad. ... It was translated into Latin as Algoritmi de numero Indorum.'¹⁴

However the description that I like the best is 'An algorithm is just fancy term for a set of instructions of what a program should do, and how it should do it. In other words: it's nothing more than a manual for your code. That's it. (Really!) Some of the most-referenced algorithms in the world of software are generally a subset of sorting algorithms, or

¹¹ <http://archive.oreilly.com/oreillyschool/courses/data-structures-algorithms/largeData.html>

¹² <https://en.wikipedia.org/wiki/Sorting>

¹³ <https://medium.com/basecs/sorting-out-the-basics-behind-sorting-algorithms-b0a032873add>

¹⁴ <https://www.quora.com/Where-does-the-word-algorithm-come-from>

*algorithms that provide a set of instructions for how a program or system should go about organizing a set of data.*¹⁵

How is sorting done?

According to O'Reilly when discussing computer algorithms to sort data 'Most sorting algorithms operate over an array of values, swapping elements in the array until the elements are in order.'¹⁶

For someone uninitiated in maths or computer science it can be a surprise to learn just how many different algorithms people have created to sort data, to name but a few: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quicksort, Heap Sort, Counting Sort, Bucket Sort, Radix Sort.

Most main stream algorithms are divided into two main categories of 1) comparison-based sort and 2) non-comparison sort, there are also hybrid sorting algorithms such as the Tim Sort which is inspired by both Merge Sort and Insertion Sort.

Most people will never have to learn how to create a sorting algorithm, as sort commands such as `sort()` and `sorted()` in python are provided as inbuilt functions of the tools of the industry, and Excel comes with its own sort methods. For benchmarking comparisons I will show how Python's own inbuilt function performs compared to other algorithm codes that I have copied from the internet.

Not all sorting algorithms or how they are coded is created equal. How code is written for computers to swap elements in the array until all elements are in order, can have a massive impact on the time that it will take for the sorting task to be done, the space needed on the computer for its processing resources to perform the task. In this project we will be considering: **relevance of concepts such as complexity (time and space), relevance of performance, in-place sorting, stable sorting, comparator functions, comparison-based and non-comparison-based sorts.**

Time Complexity, Order of Growth, O-notation

The easiest way to classify an algorithm is by *time complexity*

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.¹⁷

¹⁵ <https://medium.com/basecs/sorting-out-the-basics-behind-sorting-algorithms-b0a032873add>

¹⁶ <http://archive.oreilly.com/oreillyschool/courses/data-structures-algorithms/largeData.html>

¹⁷ <https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/>

All sorting algorithms either perform their processes iteratively or recursively.

*'An **iterative process** is a **process** for calculating a desired result by **means** of a repeated cycle of operations. An **iterative process** should be convergent, i.e., it should come closer to the desired result as the number of **iterations** increases. * Next. * Previous. * Index.'*¹⁸

*'**Recursion** is a computer **programming** technique involving the use of a procedure, subroutine, function, or algorithm that calls itself in a step having a termination condition so that successive repetitions are processed up to the critical step where the condition is met at which time the rest of each repetition is ... '*¹⁹

Whether algorithms use iteration or recursion in the sorting processes can massively impact their speed and performance.

Demonstration of Iteration



Diagram concept from Thinking Recursively https://abhirag.in/thinking_recursively/index.html

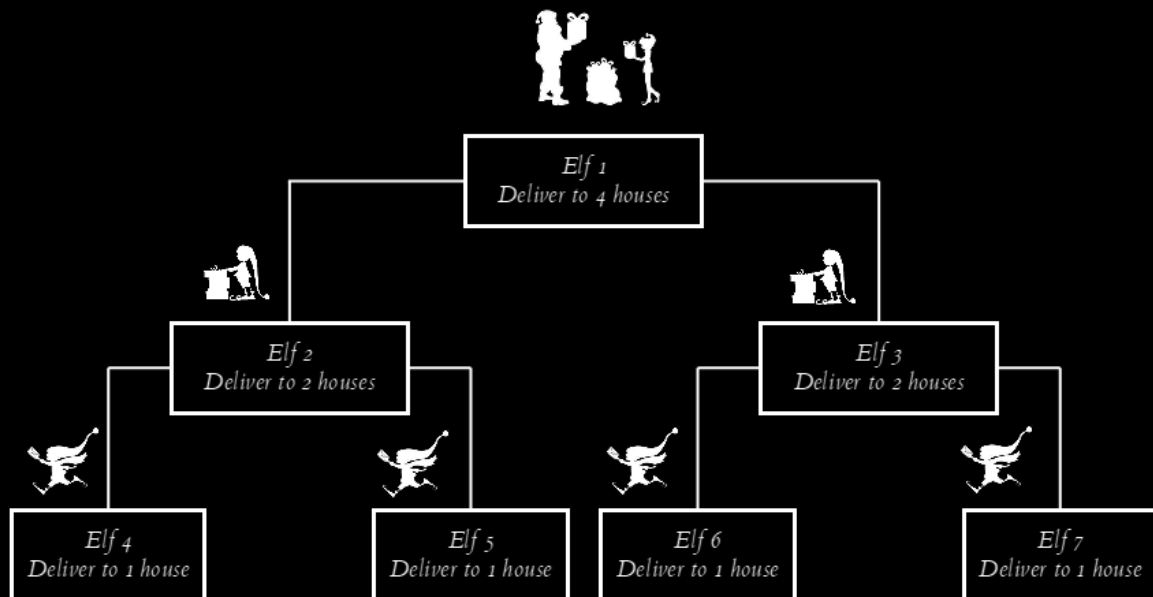
¹⁸ http://pespmc1.vub.ac.be/ASC/ITERAT_PROCE.html

¹⁹ <https://www.sparknotes.com/cs/recursion/whatisrecursion/section1/>

Some algorithms work by recursive methods.

These algorithms, generally get the job done quicker, but require more computing resource to recursively break down the sort into multiple processes of smaller tasks.

This is like Santa appointing a manager elf, who appoints further manager elves, who divides responsibility to deliver among further manager elves, until there are only worker elves.



THE EXCEPTIONS

Some recursive sorting methods may not be as effective or necessary use of processing resource for a small list.



Some sorting methods are more or less effective on already sorted or nearly sorted lists.



Order of growth

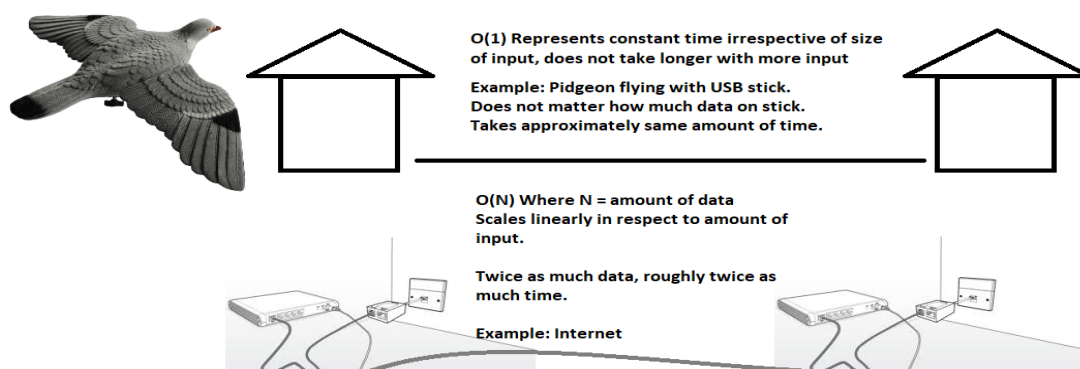
Order of growth is how the time of execution depends on the length of the input.

Big O Notation is frequently discussed regarding Time and Space Complexities of algorithms.

It is a mistake to assume when a sorting algorithm receives more data to sort that the algorithm takes double the time.

One of the simplest explanations I found about Big O Notation was on a Youtube video which explained that there was once a company in South Africa with two premises approximately 50km apart, who struggled with slow broadband when they were sending large data. They found that it was sometimes quicker to send data via carrier pigeon.

Scaling sizes of data down the internet, e.g. sending double the data $O(N)$ generally meant that it took double the time to arrive, meant increased slowness, however scaling sizes of data on USB stick carried by a carrier pigeon resulted in no extra time and was often faster than sending large sizes of data down the internet, the this illustrated $O(1)$.



Although the above is a rather crude example, it is a simple reminder not to assume that all algorithm speeds scale linearly on increase of data.

While analysing an algorithm, we mostly consider O -notation because it will give us an upper limit of the execution time i.e. the execution time in the worst case. Every algorithm will be different.

Space Complexity

Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.²⁰

²⁰ <https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/>

In sorting algorithm methods, just like the Santa and elves example, using recursion and more advanced programming methods for the same task can utilise different amounts of resources to save time. You could employ a pigeon to deliver your internet data, but that is going to take up additional space in your office, and if you are going to want to sort huge amounts of data you are going to need to know in advance whether you are going to allocate extra server space and advanced algorithms, or to save computer resource for other processes be prepared to use more basic algorithms and wait longer.

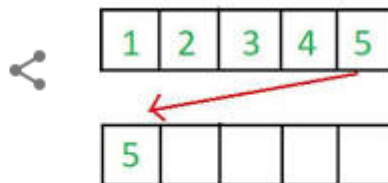
Relevance of performance,

*Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task. That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.*²¹

In-place sorting,

*In-place sorting means sorting without any extra space requirement. According to wiki , it says. an in-place algorithm is an algorithm which transforms input using a data structure with a small, constant amount of extra storage space. Quicksort is one example of In-Place Sorting.*²²

In-place algorithm



In computer science, an in-place algorithm is an algorithm which transforms input using no auxiliary data structure. However a small amount of extra storage space is allowed for auxiliary variables. The input is usually overwritten by the output as the algorithm executes.

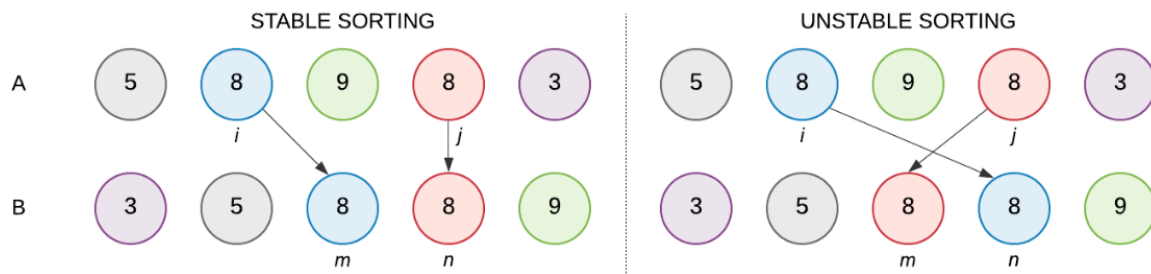
[Wikipedia](#)

²¹ http://www.btechsmartclass.com/data_structures/performance-analysis.html

²² <https://stackoverflow.com/questions/16585507/sorting-in-place>

Stable Sorting

The stability of a sorting algorithm is concerned with how the algorithm treats equal (or repeated) elements. Stable sorting algorithms preserve the relative order of equal elements, while unstable sorting algorithms don't. In other words, stable sorting maintains the position of two equals elements relative to one another.²³



Comparison-based sorting

A comparison sort is a type of sorting algorithm that only reads the list elements through a single abstract comparison operation (often a "less than or equal to" operator or a three-way comparison) that determines which of two elements should occur first in the final sorted list.²⁴

By "comparison-based", we mean a sorting algorithm which accesses input array elements only via comparisons as is the case for general-purpose sorting algorithms such as merge sort, quicksort, and heapsort.²⁵

Non-comparison-based sorts, etc.,

The process is known as non-comparison sorting and algorithms are known as the non-comparison based sorting algorithms. No comparison sorting includes Counting sort which sorts using key value, Radix sort, which examines individual bits of keys, and Bucket Sort which examines bits of keys.²⁶

²³ <https://www.baeldung.com/cs/stable-sorting-algorithms>

²⁴ https://en.wikipedia.org/wiki/Comparison_sort

²⁵ <https://diego.assencio.com/?index=143ab4f37feadc3ac69f33f43fde3a2a>

²⁶ <https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html>

PART 2 SORTING ALGORITHMS

Below is a table which I have modified from <http://biggocheatsheet.com/> , which from a mathematical perspective it specifically addresses and answers the Time Complexity and Space Complexity strengths and weaknesses for each array sorting algorithm. To read it and understand it you will need to be familiar with Omegas Ω and Thetas Θ and Big O Notation.

	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$	$O(n \log(n))$
Mergesort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$	$O(n)$ $O(N)$ describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. 27
Timsort	$\Omega(n)$	$\theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Bubble Sort	$\Omega(n)$	$\theta(n^2)$ Number of items in array squared	$O(n^2)$ Number of items in array squared	$O(1)$ $O(1)$ describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.28
Radix Sort	$\Omega(nk)$	$\theta(nk)$	$O(nk)$	$O(n+k)$

²⁷ <https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/>

²⁸ <https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/>

Bubble Sort – 1962 – A simple comparison based sort

About Bubble Sort

- *Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.*²⁹
- *This simple algorithm performs poorly in real world use and is used primarily as an educational tool.*³⁰
- I cannot find anyone who takes credit for inventing this algorithm. The term "Bubble Sort " was used by Kenneth Iverson in 1962.³¹ In his book Iverson, K: A Programming Language. John Wiley, 1962.
- *Kenneth Iverson, was a Canadian mathematician and computer scientist who pioneered a very compact high-level computer programming language called APL (the initials of his book A Programming Language [1962]). The language made efficient use of the slow communication speeds of the computer terminals of that time, and APL enjoyed an enthusiastic following. Iverson taught mathematics at Harvard University from 1955 to 1960 and served on the staff of the research division of IBM from 1960 to 1980.*³²

²⁹ https://en.wikipedia.org/wiki/Bubble_sort

³⁰ https://en.wikipedia.org/wiki/Bubble_sort

³¹ Iverson, K: A Programming Language. John Wiley, 1962.

³² <https://www.britannica.com/topic/A-Programming-Language>

Bubble Sort Diagram

Edited original found on internet

Trace of Bubble Sort Algorithm

Input: a[9] = {54,26,93,17,77,31,44,55,20}

Pass 1:

54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in its right position

Pass 2:

26	54	17	77	31	44	55	20	93	No Exchange
26	54	17	77	31	44	55	20	93	Exchange
26	17	54	77	31	44	55	20	93	No Exchange
26	17	54	77	31	44	55	20	93	Exchange
26	17	31	54	77	44	55	20	93	Exchange
26	17	31	44	77	55	20	93	93	Exchange
26	17	31	44	55	77	20	93	93	Exchange
26	17	31	44	55	20	77	93	93	No Exchange
26	17	31	44	55	20	77	93	93	77 in its right position

Pass 3:

26	54	17	31	44	55	20	77	93	No Exchange
26	54	17	31	44	55	20	77	93	Exchange
26	17	54	31	44	55	20	77	93	Exchange
26	17	31	54	44	55	20	77	93	Exchange
26	17	31	44	54	55	20	77	93	No Exchange
26	17	31	44	54	55	20	77	93	Exchange
26	17	31	44	54	20	55	77	93	No Exchange
26	17	31	44	54	20	55	77	93	No Exchange
26	17	31	44	54	20	55	77	93	55 in its right position

Pass 4:

26	17	31	44	54	20	55	77	93	Exchange
17	26	31	44	54	20	55	77	93	No Exchange
17	26	31	44	54	20	55	77	93	No Exchange
17	26	31	44	54	20	55	77	93	No Exchange
17	26	31	44	54	20	55	77	93	Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	54 in its right position

Pass 5:

17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	44 in its right position

Pass 6:



17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	31 in its right position

Pass 7:

17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	26 in its right position

Pass 8:

In this pass, no exchange for all 8 comparisons. Since, the input array size is 9, the number of passes is 8. The algorithm terminates and the input array contains the sorted sequence.

 indicates numbering order swapped
 indicates number in correct final position

Bubble Sort Space Usage

Array Sorting Algorithms: <http://bigocheatsheet.com/>

	Space Complexity
	Worst
Bubble Sort	$O(1)$ $O(1)$ describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set. ³³

The main advantage of Bubble Sort is the simplicity of the algorithm.

The space complexity for Bubble Sort is $O(1)$, because only a single additional memory space is required i.e. for temp variable. ³⁴

³³ <https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/>

³⁴ <https://www.studytonight.com/data-structures/bubble-sort>

Bubble Sort Time Complexity

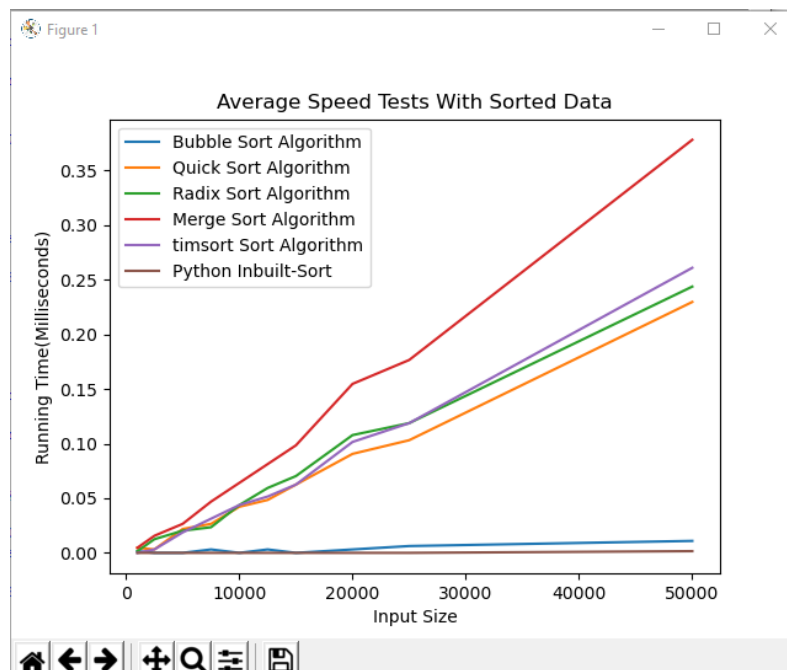
Array Sorting Algorithms: <http://bigocheatsheet.com/>

	Time Complexity		
	Best	Average	Worst
Bubble Sort	$\Omega(n)$	$\theta(n^2)$ Number of items in array squared	$O(n^2)$ Number of items in array squared

Best Case

According to online sources the best case time complexity for Bubble Sort Algorithm will be $O(n)$, for a list that is already sorted.

In the graph below, generated from my Python software benchmarking tests, you will see on average the Bubble Sort Algorithm on SORTED lists out performs all other algorithms apart from Python's own Inbuilt-Sort which relies on Tim Sort. The Bubble Sort Algorithm is even faster than version of Tim Sort code copied from the internet.



Worst case

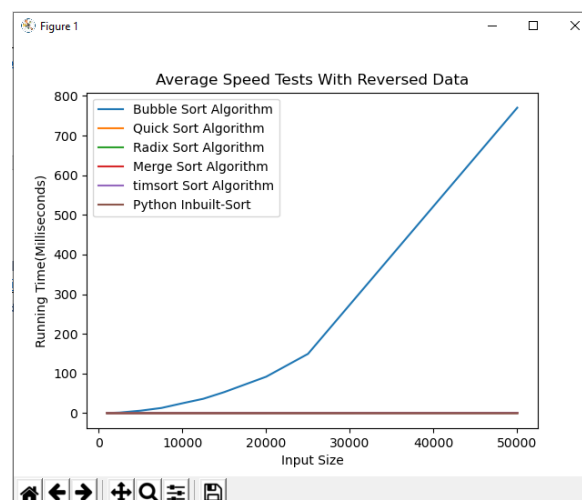
Bubble sort has a worst-case and average complexity of $O(n^2)$, where n is the number of items being sorted.

Most practical sorting algorithms have substantially better worst-case or average complexity, often $O(n \log n)$. Even other $O(n^2)$ sorting algorithms, such as insertion sort, generally run faster than bubble sort, and are no more complex. Therefore, bubble sort is not a practical sorting algorithm.

The only significant advantage that bubble sort has over most other algorithms, even quicksort, but not insertion sort, is that the ability to detect that the list is sorted efficiently is built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only $O(n)$. By contrast, most other algorithms, even those with better average-case complexity, perform their entire sorting process on the set and thus are more complex. However, not only does insertion sort share this advantage, but it also performs better on a list that is substantially sorted (having a small number of inversions).

Bubble sort should be avoided in the case of large collections. It will not be efficient in the case of a **reverse-ordered** collection.³⁵

My benchmarking graph below shows Bubble Sort is the slowest at sorting a reverse-ordered collection.



Benchmark test comparison results sorting reverse ordered lists

Size	10000	12500	15000	20000	25000	50000
Python in-built sort (fastest)	0.000	0.000	0.000	0.000	0.000	0.001
Quicksort (fastest code)	0.042	0.046	0.063	0.089	0.117	0.242
Merge sort (2 nd slowest)	0.073	0.094	0.109	0.143	0.181	0.396
Tim sort (3 rd slowest)	0.044	0.054	0.068	0.092	0.117	0.281
Bubble sort (slowest)	24.822	36.176	53.039	91.774	149.451	770.311

³⁵ https://en.wikipedia.org/wiki/Bubble_sort

Bubble Sort Code <https://realpython.com/sorting-algorithms-python/#the-bubble-sort-algorithm-in-python>

```
1 def bubble_sort(array):
2     n = len(array)
3
4     for i in range(n):
5         # Create a flag that will allow the function to
6         # terminate early if there's nothing left to sort
7         already_sorted = True
8
9         # Start looking at each item of the list one by one,
10        # comparing it with its adjacent value. With each
11        # iteration, the portion of the array that you look at
12        # shrinks because the remaining items have already been
13        # sorted.
14        for j in range(n - i - 1):
15            if array[j] > array[j + 1]:
16                # If the item you're looking at is greater than its
17                # adjacent value, then swap them
18                array[j], array[j + 1] = array[j + 1], array[j]
19
20            # Since you had to swap two elements,
21            # set the `already_sorted` flag to `False` so the
22            # algorithm doesn't finish prematurely
23            already_sorted = False
24
25        # If there were no swaps during the last iteration,
26        # the array is already sorted, and you can terminate
27        if already_sorted:
28            break
29
30    return array
```

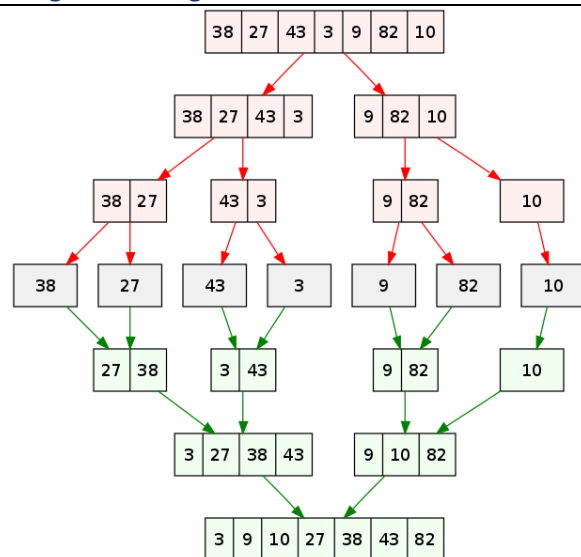
Merge Sort

About Merge Sort

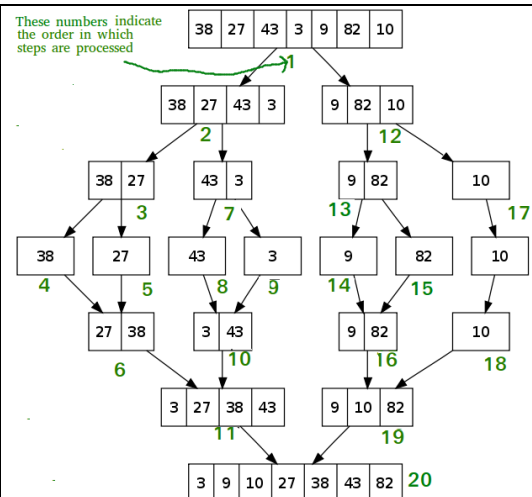
Merge Sort – 1945 – John von Neumann

- Merge Sort - An efficient comparison-based sort
- *Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945. A detailed description and analysis of bottom-up mergesort appeared in a report by Goldstine and von Neumann as early as 1948.*³⁶
- *This divide and conquer algorithm splits a list in half, and keeps splitting the list by 2 until it only has singular elements. Adjacent elements become sorted pairs, then sorted pairs are merged and sorted with other pairs as well. This process continues until we get a sorted list with all the elements of the unsorted input list.*³⁷

Merge Sort Diagram



Wikipdeia Diagram Of Merge Sort³⁸



Wikipdeia Diagram Of Merge Sort Analysed

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.³⁹

³⁶ https://en.wikipedia.org/wiki/Merge_sort

³⁷ <https://stackabuse.com/sorting-algorithms-in-python/#mergesort>

³⁸ https://commons.wikimedia.org/wiki/File:Merge_sort_algorithm_diagram.svg

³⁹ <https://www.geeksforgeeks.org/merge-sort/>

Merge Sort Space Usage

Array Sorting Algorithms: <http://bigocheatsheet.com/>

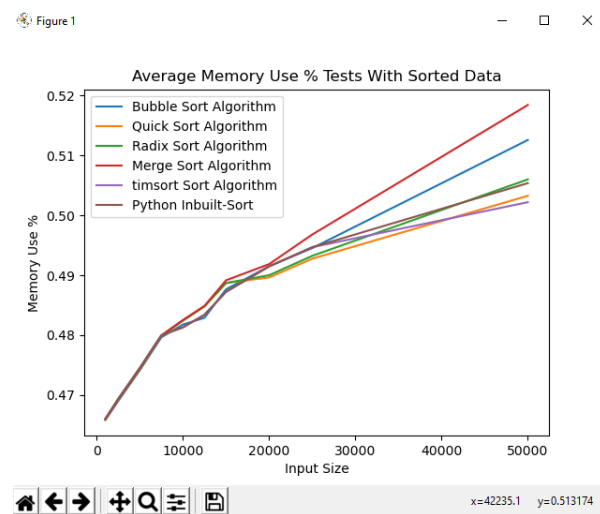
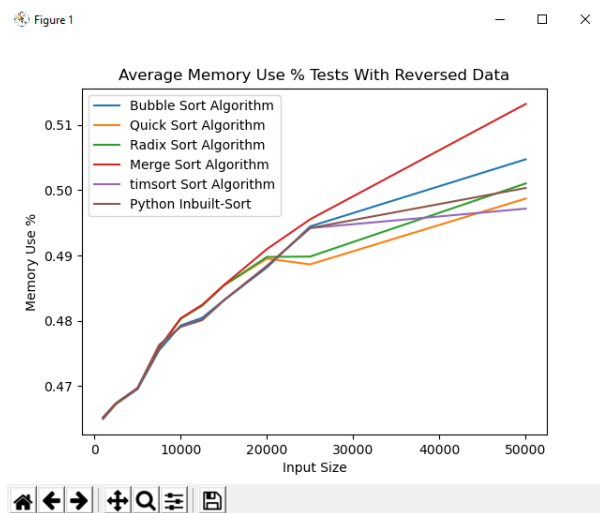
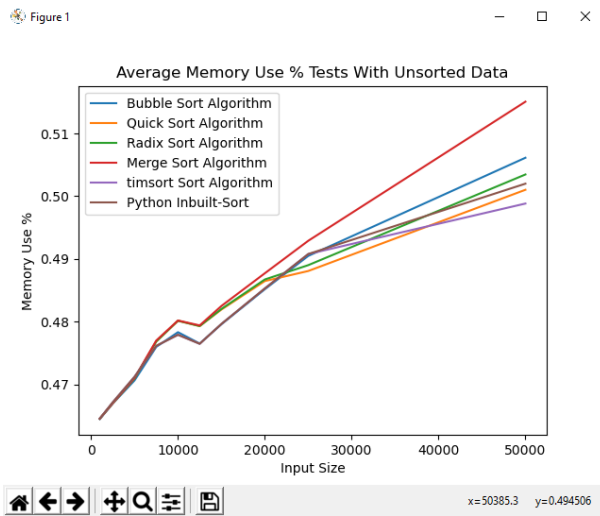
	Space Complexity
	Worst
Mergesort	$O(n)$ $O(N)$ describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. ⁴⁰

*Unlike some (efficient) implementations of quicksort, merge sort is a stable sort. Merge sort's most common implementation does not sort in place; therefore, **the memory size of the input must be allocated for the sorted output to be stored in.*** ⁴¹

⁴⁰ <https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/>

⁴¹ https://en.wikipedia.org/wiki/Merge_sort

My benchmarking graphs below shows Merge Sort uses most memory across tests with Unsorted, Reversed and Sorted Data. The memory usage is in the same range each time.



Merge Sort Time Complexity

Array Sorting Algorithms: <http://bigocheatsheet.com/>

	Time Complexity		
	Best	Average	Worst
Mergesort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$

According to the geeksforgeeks website '***Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets. Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.*** Sorting method : The quick sort is internal sorting method where the data is sorted in main memory.⁴²

My findings so far agree with the comments quoted above that Quicksort is faster than Merge sort on smaller data sets. On ALL the benchmark tests that I ran with unsorted data sets with unsorted arrays of 50,000 numbers or under, Quicksort was quicker than Merge Sort. My benchmark tests must be considered small array / data sizes.

Benchmark test comparison results on small data sets

Size	10000	12500	15000	20000	25000	50000
Quicksort	0.043	0.046	0.066	0.093	0.122	0.257
Merge sort	0.070	0.082	0.113	0.159	0.200	0.419

I would have to run tests with much bigger data sets to prove that Merge sort is faster than Quick sort.

⁴² <https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/>

Merge Sort Code <https://realpython.com/sorting-algorithms-python/#the-merge-sort-algorithm-in-python>

```
1 def merge(left, right):
2     # If the first array is empty, then nothing needs
3     # to be merged, and you can return the second array as the result
4     if len(left) == 0:
5         return right
6
7     # If the second array is empty, then nothing needs
8     # to be merged, and you can return the first array as the result
9     if len(right) == 0:
10        return left
11
12    result = []
13    index_left = index_right = 0
14
15    # Now go through both arrays until all the elements
16    # make it into the resultant array
17    while len(result) < len(left) + len(right):
18        # The elements need to be sorted to add them to the
19        # resultant array, so you need to decide whether to get
20        # the next element from the first or the second array
21        if left[index_left] <= right[index_right]:
22            result.append(left[index_left])
23            index_left += 1
24        else:
25            result.append(right[index_right])
26            index_right += 1
27
28        # If you reach the end of either array, then you can
29        # add the remaining elements from the other array to
30        # the result and break the loop
31        if index_right == len(right):
32            result += left[index_left:]
33            break
34
35        if index_left == len(left):
36            result += right[index_right:]
37            break
```

```
38
39     return result
40
41 def merge_sort(array):
42     # If the input array contains fewer than two elements,
43     # then return it as the result of the function
44     if len(array) < 2:
45         return array
46
47     midpoint = len(array) // 2
48
49     # Sort the array by recursively splitting the input
50     # into two equal halves, sorting each half and merging them
51     # together into the final result
52     return merge(
53         left=merge_sort(array[:midpoint]),
54         right=merge_sort(array[midpoint:]))
```

Radix Sort

Christopher's comments on Radix Sort

While studying Radix sort, I came across explanations of what a Radix sort is that seem to conflict with each other. The first explanation is that Radix is an integer sorting algorithm. Based on the fact that in mathematics, radix refers to the base, the number of unique digits, where decimal is known as base 10.⁴³ *'Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value). Radix sort uses counting sort as a subroutine to sort an array of numbers.'*⁴⁴ Unlike quicksort which is universal, radix sort is only useful for fix length integer keys. Thus in real life scenario quicksort will be very often faster than radix sort.⁴⁵

Then there are indications that Radix Sort, can operate in other domains, *'Radix sort can be applied to data that can be sorted lexicographically, be they integers, words, punch cards, playing cards, or the mail.'*⁴⁶ This explanation that Radix Sort is designed to work in broader domains than integers, domains which can be sorted lexicographically very much brings it back to its original origins, that the Radix Sort algorithm was originally used to sort punched cards in several passes.⁴⁷

Radix sort, has evolved into an algorithm which calls upon other algorithms as a subroutine to sort individual digits, e.g. a Radix sort can use counting sort, insertion sort, bubble sort, or bucket sort as a subroutine to sort individual digits.

*As a result a Radix sort may require additional memory space to sort digits, especially when using counting sort.*⁴⁸

The time complexity of the Radix sort will depend on the subroutine algorithm which it calls. In the benchmarking code for this assignment, I have used a Radix sort which uses counting sort.

⁴³ http://syllabus.cs.manchester.ac.uk/ugt/2018/COMP26120/SortingTool/radix_sort_info.html

⁴⁴ <https://brilliant.org/wiki/radix-sort/>

⁴⁵ <https://softwareengineering.stackexchange.com/questions/77529/why-isnt-radix-sort-used-more-often>

⁴⁶ https://en.wikipedia.org/wiki/Radix_sort

⁴⁷ http://encyclopedia.kids.net.au/page/ra/Radix_sort

⁴⁸ http://syllabus.cs.manchester.ac.uk/ugt/2018/COMP26120/SortingTool/radix_sort_info.html

About Radix Sort

Radix Sort – 1954 – Harold H. Seward

- Radix Sort - A non-comparison sort
- *Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of same place value. Then, sort the elements according to their increasing/decreasing order.*⁴⁹
- *Radix Sort was invented by Harold H. Seward. Seward was a computer scientist, engineer, and inventor. He developed the radix sort and counting sort algorithms in 1954 at MIT. He also worked on the Whirlwind Computer and developed instruments that powered the guidance systems for the Apollo spacecraft and Polaris missile.*⁵⁰
- *In many applications requiring super speeds and massive memory, the computer radix sort invented by Seward supersedes earlier slower comparison sorts.*⁵¹ *The Radix Sort algorithm was originally used to sort punched cards in several passes.*⁵²

The **punch card** (or "Hollerith" card) is a medium for holding information for use by automated machines. Made of stiff cardboard, the punch card represents information by the presence or absence of holes in predefined positions on the card. In the first generation of computing during the 1960s and 1970s, punch cards were a primary medium for data storage and processing, but are now long obsolete outside of a few legacy systems. The punched card actually predates computers considerably, originating in 1801 as a control device for Jacquard looms. Such cards were also used as an input method for the primitive calculating machines of the late 19th century.⁵³

Radix Sort is an integer sorting algorithm that depends on a sorting subroutine that must be stable. It is a non-comparison based sorting algorithm that sorts a collection of integers.⁵⁴

Radix Sort groups keys by individual digits that share the same significant position and value.⁵⁵

⁴⁹ <https://www.programiz.com/dsa/radix-sort>

⁵⁰ https://en.wikipedia.org/wiki/Harold_H._Seward

⁵¹ http://encyclopedia.kids.net.au/page/pu/Punched_card

⁵² http://encyclopedia.kids.net.au/page/ra/Radix_sort

⁵³ http://encyclopedia.kids.net.au/page/pu/Punched_card

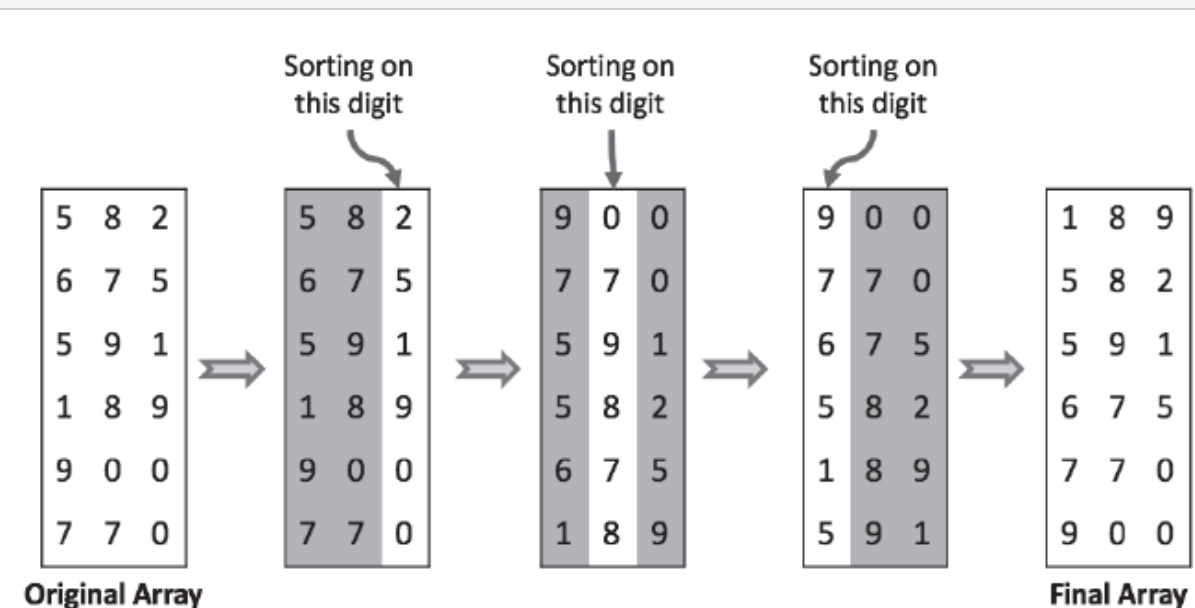
⁵⁴ <https://www.baeldung.com/cs/stable-sorting-algorithms>

⁵⁵ <https://www.baeldung.com/cs/stable-sorting-algorithms>

Radix Sort Diagram

Perhaps the easiest way to demonstrate how the Radix Sort works is provided by Kamal Rawat of Ritambhara Technologies. He says 'Consider an array that stores account numbers of all employees. One unique thing about account numbers is that they have equal number of digits. Below is an example of where, account numbers are three digits long, and array has 6 account numbers are shown below.'⁵⁶

```
int arr[ ] = {582, 675, 591, 189, 900, 770}
```



Each invocation of the Counting Sort subroutine preserves the order from the previous invocations.⁵⁷

⁵⁶ <http://www.ritambhara.in/radix-sort/>

⁵⁷ <https://www.baeldung.com/cs/stable-sorting-algorithms>

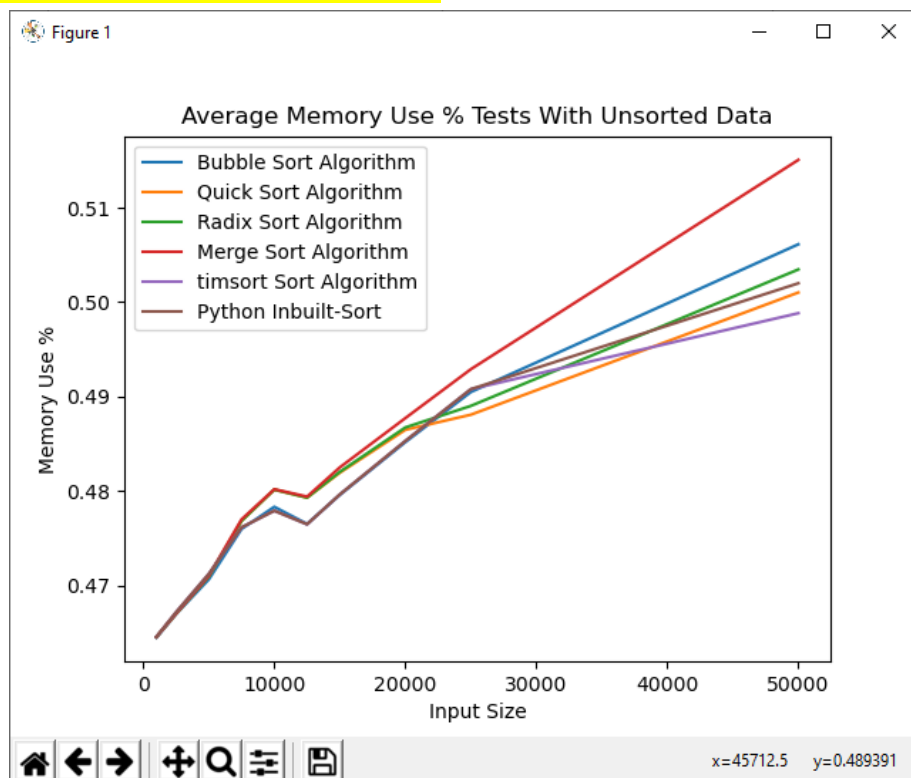
Radix Sort Space Usage

Array Sorting Algorithms: <http://bigocheatsheet.com/>

	Space Complexity
	Worst
Radix Sort	$O(n+k)$

Radix sort can use counting sort, insertion sort, bubble sort, or bucket sort as a subroutine to sort individual digits. As a result, may require additional memory space to sort digits, especially when using counting sort.⁵⁸

SCREENSHOT COMPARISON BELOW OF ALGORITHMS AND MEMORY USAGE SHOWS HOW RADIX COMPARES TO OTHER ALGORITHMS WHEN USING A COUNTING SORT AS A SUBROUTINE



Radix sort's space complexity is bound to the sort it uses to sort each radix. In best case, that is counting sort. As you can see, counting sort creates multiple arrays, one based on the size of K , and one based on the size of N . B is the output array which is size n .⁵⁹

⁵⁸ http://syllabus.cs.manchester.ac.uk/ugt/2018/COMP26120/SortingTool/radix_sort_info.html

⁵⁹ <https://stackoverflow.com/questions/44477979/why-does-radix-sort-have-a-space-complexity-of-ok-n>

Radix Sort Time Complexity

Array Sorting Algorithms: <http://bigocheatsheet.com/>

	Time Complexity		
	Best	Average	Worst
Radix Sort	$\Omega(nk)$	$\theta(nk)$	$O(nk)$

Radix sorts operates in $O(nw)$ time, where n is the number of keys, and w is the key length.⁶⁰ Optimized radix sorts can be very fast when working in a domain that suits them.⁶¹

The runtime for radix sort is $O(nk)$, where n is the length of the array, k is the maximum number of digits.⁶²

Radix sort can use counting sort, insertion sort, bubble sort, or bucket sort as a subroutine to sort individual digits. As a result, may require additional memory space to sort digits, especially when using counting sort.⁶³

"Radix sort only applies to integers, fixed size strings, floating points and to "less than", "greater than" or "lexicographic order" comparison predicates, whereas comparison sorts accommodates different orders."⁶⁴

⁶⁰ https://en.wikipedia.org/wiki/Radix_sort

⁶¹ https://en.wikipedia.org/wiki/Radix_sort

⁶² http://syllabus.cs.manchester.ac.uk/ugt/2018/COMP26120/SortingTool/radix_sort_info.html

⁶³ http://syllabus.cs.manchester.ac.uk/ugt/2018/COMP26120/SortingTool/radix_sort_info.html

⁶⁴ <https://stackoverflow.com/questions/4146843/when-should-we-use-radix-sort/4146887#4146887>

Radix Sort Code - <https://www.programiz.com/dsa/radix-sort>

```
# Radix sort in Python
# Using counting sort to sort the elements in the basis of significant
places
def countingSort(array, place):
    size = len(array)
    output = [0] * size
    count = [0] * 10

    # Calculate count of elements
    for i in range(0, size):
        index = array[i] // place
        count[index % 10] += 1

    # Calculate cumulative count
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Place the elements in sorted order
    i = size - 1
    while i >= 0:
        index = array[i] // place
        output[count[index % 10] - 1] = array[i]
        count[index % 10] -= 1
        i -= 1

    for i in range(0, size):
        array[i] = output[i]

# Main function to implement radix sort
def radixSort(array):
    # Get maximum element
    max_element = max(array)

    # Apply counting sort to sort elements based on place value.
    place = 1
    while max_element // place > 0:
        countingSort(array, place)
        place *= 10

data = [121, 432, 564, 23, 1, 45, 788]
radixSort(data)
print(data)
```


Quick Sort

About Quick Sort

Quick Sort – 1959 – C.A.R. Hoare

- *Quick Sort - An efficient comparison-based sort*
- *Quicksort developed by British computer scientist Tony Hoare (Birthname Charles Antony Richard Hoare) in 1959 and published in 1961, it is still a commonly used algorithm for sorting. Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm.* ⁶⁵
- *The quick sort algorithm was developed in 1959 by Tony Hoare while he was a visiting student at Moscow State University. At that time, Hoare worked on a project on machine translation for National Physical Laboratory. As part of the translation process, he had to sort the words of Russian sentences prior to looking them up in a Russian-English dictionary which was already sorted in alphabetic order and stored in magnetic tape⁶⁶. To fulfill this task he discovered Quick Sort and later published the code in 1961.* ^{67 68}
- *Quick Sort begins by partitioning the list - picking one value of the list that will be in its sorted place. This value is called a pivot. All elements smaller than the pivot are moved to its left. All larger elements are moved to its right.* ⁶⁹

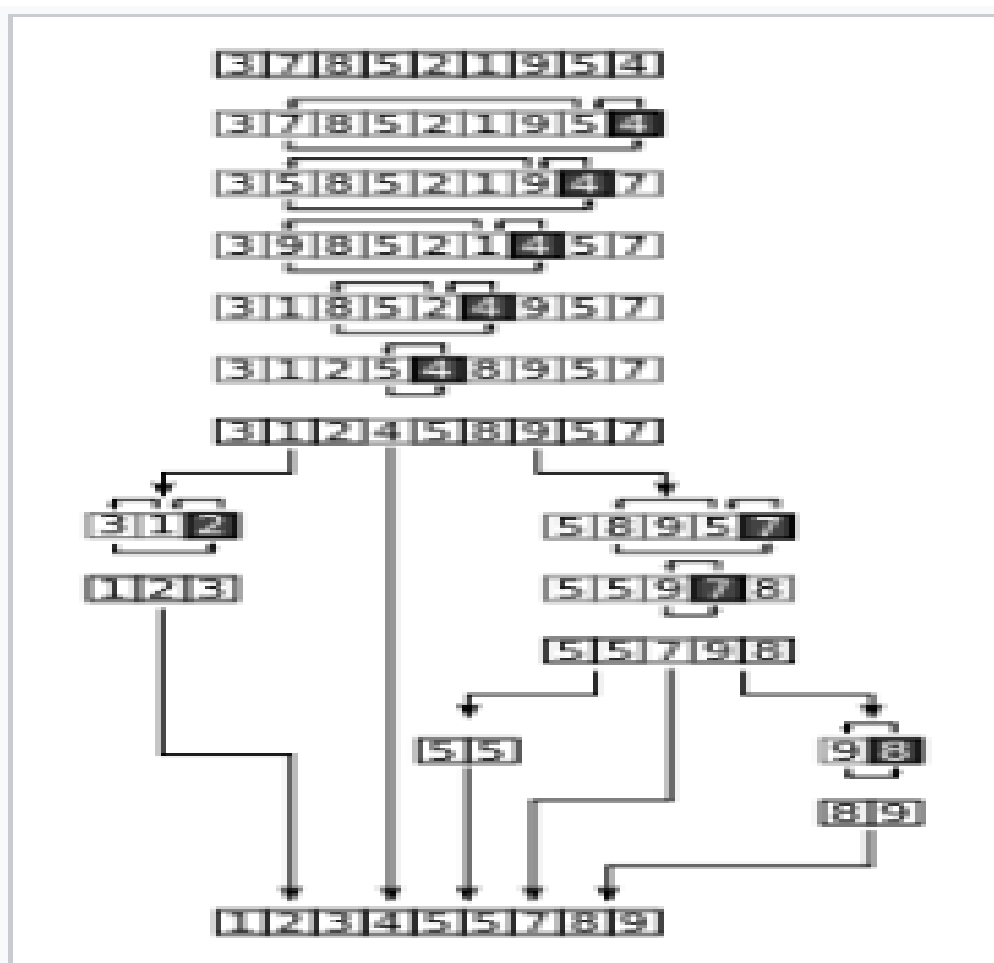
⁶⁵ <https://en.wikipedia.org/wiki/Quicksort>

⁶⁶ [Shustek, L: Interview: An interview with C.A.R. Hoare. Comm. ACM 52 \(3\): 3841, \(2009\).](#)

⁶⁷ Hoare, C. A. R: Algorithm 64: Quicksort. Comm. ACM 4 (7): 321, (1961).

⁶⁸ http://www.iiitdm.ac.in/old/Faculty_Teaching/Sadagopan/pdf/DAA/SortingAlgorithms.pdf

⁶⁹ <https://stackabuse.com/sorting-algorithms-in-python/#quicksort>



Full example of quicksort on a random set of numbers. The shaded element is the pivot. It is always chosen as the last element of the partition. However, always choosing the last element in the partition as the pivot in this way results in poor performance ($O(n^2)$) on *already sorted* arrays, or arrays of identical elements. Since sub-arrays of sorted / identical elements crop up a lot towards the end of a sorting procedure on a large set, versions of the quicksort algorithm that choose the pivot as the middle element run much more quickly than the algorithm described in this diagram on large sets of numbers.

Quick Sort Space Usage

Array Sorting Algorithms: <http://bigocheatsheet.com/>

	Space Complexity
	Worst
Quicksort	$O(n \log(n))$

*'Quicksort with in-place and unstable partitioning uses only constant additional space before making any recursive call. Quicksort must store a constant amount of information for each nested recursive call. Since the best case makes at most $O(\log n)$ nested recursive calls, it uses $O(\log n)$ space. However, without Sedgewick's trick to limit the recursive calls, in the worst case quicksort could make $O(n)$ nested recursive calls and need $O(n)$ auxiliary space.'*⁷⁰

Quick Sort Time Complexity

Array Sorting Algorithms: <http://bigocheatsheet.com/>

	Time Complexity		
	Best	Average	Worst
Quicksort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$

*'Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data.'*⁷¹

⁷⁰ https://en.wikipedia.org/wiki/Quicksort#Space_complexity

⁷¹ <https://www.geeksforgeeks.org/quick-sort/>

Quick Sort Code <https://realpython.com/sorting-algorithms-python/#the-quicksort-algorithm-in-python>

```
1 from random import randint
2
3 def quicksort(array):
4     # If the input array contains fewer than two elements,
5     # then return it as the result of the function
6     if len(array) < 2:
7         return array
8
9     low, same, high = [], [], []
10
11     # Select your `pivot` element randomly
12     pivot = array[randint(0, len(array) - 1)]
13
14     for item in array:
15         # Elements that are smaller than the `pivot` go to
16         # the `low` list. Elements that are larger than
17         # `pivot` go to the `high` list. Elements that are
18         # equal to `pivot` go to the `same` list.
19         if item < pivot:
20             low.append(item)
21         elif item == pivot:
22             same.append(item)
23         elif item > pivot:
24             high.append(item)
25
26     # The final result combines the sorted `low` list
27     # with the `same` list and the sorted `high` list
28     return quicksort(low) + same + quicksort(high)
```

Tim Sort

Tim Sort

Timsort – Created by Tim Peter in 2001

- **Timsort** is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on almost any kind of real-world data. Tim Peter created Timsort (and its awesome name) for the Python programming language in 2001.⁷²
- So how efficient is it really? Well since its invention in 2001, its been used as the default sorting algorithm in Python, Java, Android OS, and Octave! Its a trusted and go-to method for sorting.⁷³

⁷² <https://medium.com/@george.seif94/this-is-the-fastest-sorting-algorithm-ever-b5cee86b559c>

⁷³ <https://medium.com/@george.seif94/this-is-the-fastest-sorting-algorithm-ever-b5cee86b559c>

Tim Sort Diagram

*Timsort actually takes a very intuitive approach to sorting. Rather than jump right into calculations and operations like other sorting algorithms do, Timsort takes a step back. It first analyses the data that it will be sorting, and then based on that analysis will choose the most appropriate approach for the job!*⁷⁴

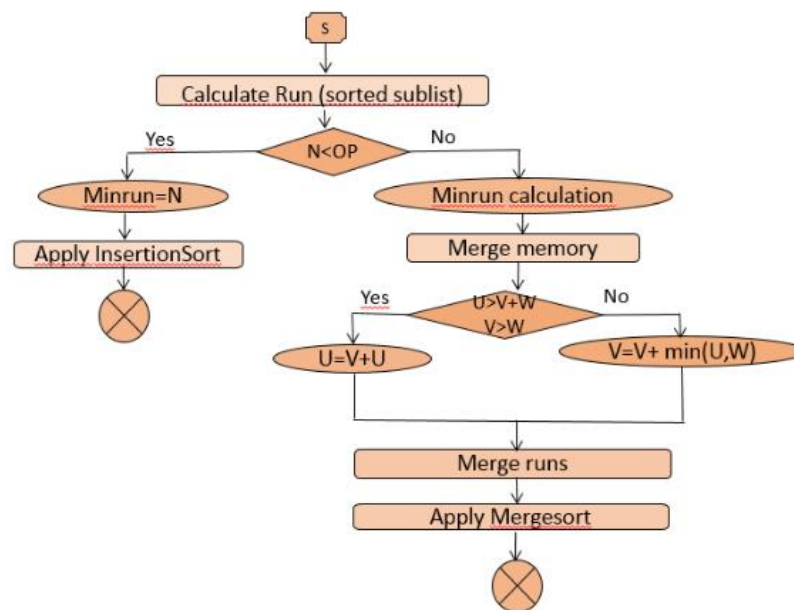


Image source https://www.researchgate.net/figure/TimSort-Algorithm-20_fig5_332549201

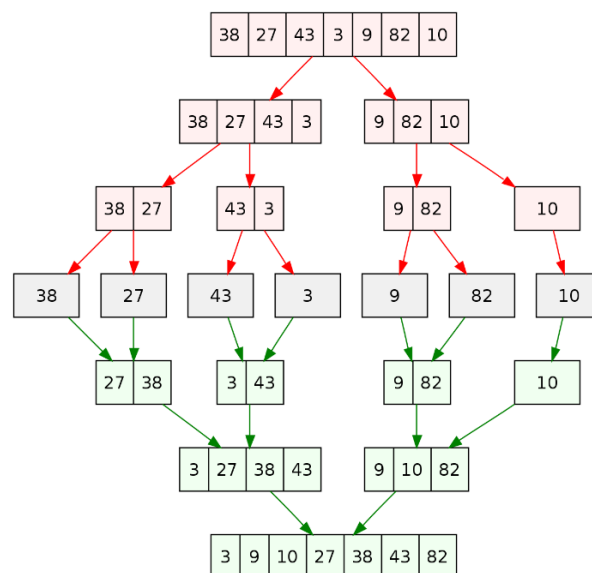


Image source: <https://medium.com/@rylanbauermeister/understanding-timsort-191c758a42f3>

⁷⁴ <https://medium.com/@george.seif94/this-is-the-fastest-sorting-algorithm-ever-b5cee86b559c>

Tim Sort Space Usage

Array Sorting Algorithms: <http://bigoocheatsheet.com/>

	Space Complexity
	Worst
Timsort	$O(n)$ $O(N)$ describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. ⁷⁵

In order to use less space, merges are done partially in-place, with the smaller of two adjacent runs being copied into scratch space. Because the smaller run is always chosen, TimSort will never need more than $O(N/2)$ extra space. Then, the "insertion point" is found with a binary search.⁷⁶

Tim Sort Time Complexity

Array Sorting Algorithms: <http://bigoocheatsheet.com/>

	Time Complexity		
	Best	Average	Worst
Timsort	$\Omega(n)$	$\theta(n \log(n))$	$O(n \log(n))$

Strengths

Timsort performs exceptionally well on already-sorted or close-to-sorted lists, leading to a best-case scenario of $O(n)$. In this case, Timsort clearly beats merge sort and matches the best-case scenario for quicksort.⁷⁷

⁷⁵ <https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/>

⁷⁶ <https://wiki.c2.com/?TimSort>

⁷⁷ <https://realpython.com/sorting-algorithms-python/#the-timsort-algorithm-in-python>

Weakness

But the worst case for Timsort is also $O(n \log_2 n)$, which surpasses quicksort's $O(n^2)$ ⁷⁸

Tim Sort Code <https://realpython.com/sorting-algorithms-python/#the-timsort-algorithm-in-python>

```
1 def insertion_sort(array, left=0, right=None):
2     if right is None:
3         right = len(array) - 1
4
5     # Loop from the element indicated by
6     # `left` until the element indicated by `right`
7     for i in range(left + 1, right + 1):
8         # This is the element we want to position in its
9         # correct place
10        key_item = array[i]
11
12        # Initialize the variable that will be used to
13        # find the correct position of the element referenced
14        # by `key_item`
15        j = i - 1
16
17        # Run through the list of items (the left
18        # portion of the array) and find the correct position
19        # of the element referenced by `key_item`. Do this only
20        # if the `key_item` is smaller than its adjacent values.
21        while j >= left and array[j] > key_item:
22            # Shift the value one position to the left
23            # and reposition `j` to point to the next element
24            # (from right to left)
25            array[j + 1] = array[j]
26            j -= 1
27
28        # When you finish shifting the elements, position
29        # the `key_item` in its correct location
30        array[j + 1] = key_item
```

⁷⁸ <https://realpython.com/sorting-algorithms-python/#the-timsort-algorithm-in-python>


```

31
32     return array

def timsort(array):
    2     min_run = 32
    3     n = len(array)
    4
    5     # Start by slicing and sorting small portions of the
    6     # input array. The size of these slices is defined by
    7     # your `min_run` size.
    8     for i in range(0, n, min_run):
    9         insertion_sort(array, i, min((i + min_run - 1), n - 1))
10
11     # Now you can start merging the sorted slices.
12     # Start from `min_run`, doubling the size on
13     # each iteration until you surpass the length of
14     # the array.
15     size = min_run
16     while size < n:
17         # Determine the arrays that will
18         # be merged together
19         for start in range(0, n, size * 2):
20             # Compute the `midpoint` (where the first array ends
21             # and the second starts) and the `endpoint` (where
22             # the second array ends)
23             midpoint = start + size - 1
24             end = min((start + size * 2 - 1), (n-1))
25
26             # Merge the two subarrays.
27             # The `left` array should go from `start` to
28             # `midpoint + 1`, while the `right` array should
29             # go from `midpoint + 1` to `end + 1`.
30             merged_array = merge(
31                 left=array[start:midpoint + 1],
32                 right=array[midpoint + 1:end + 1])
33
34             # Finally, put the merged array back into
35             # your array

```

```
36         array[start:start + len(merged_array)] = merged_array
37
38     # Each iteration should double the size of your arrays
39     size *= 2
40
41     return array
```

PART 3 – IMPLEMENTATION AND BENCHMARKING

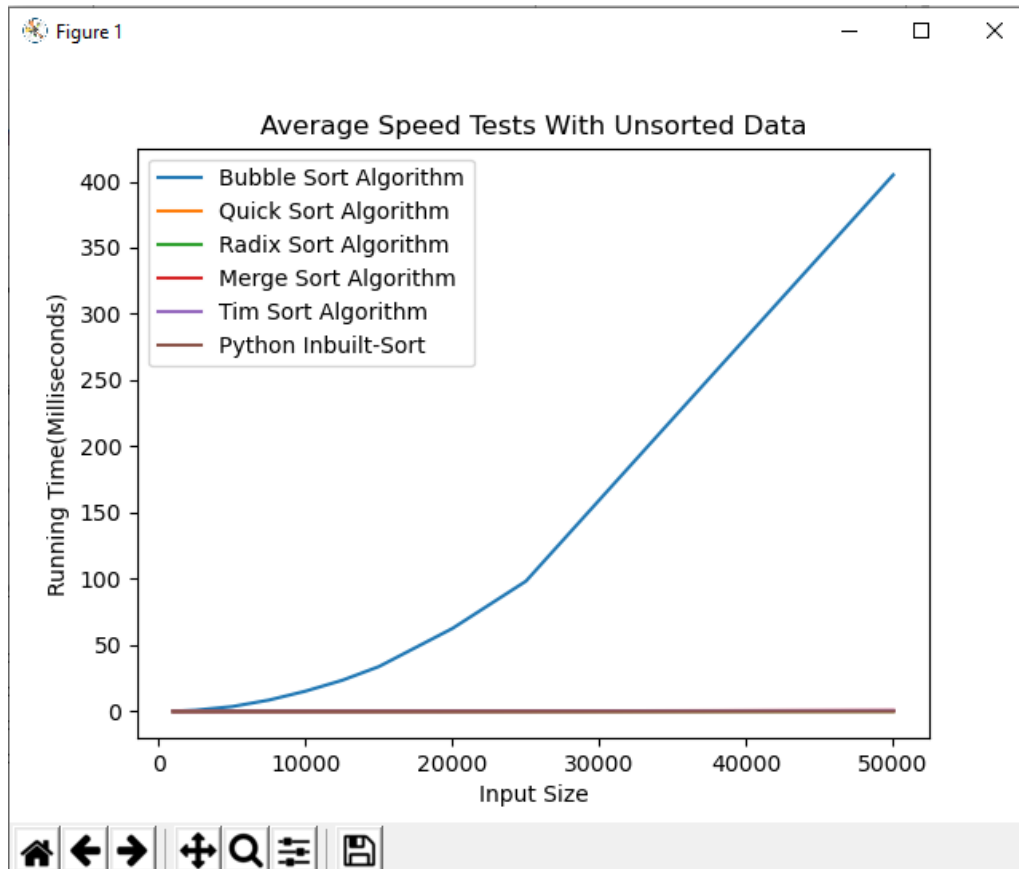
As set out in the project brief, I first designed and created a Python program which could send different array sizes of random unsorted data to five sorting algorithms, and run the process ten times to be able to print a screen report and graph the average time it took each algorithm to process the different array sizes.

I then added enhancements.

The more I learned about the strengths and weaknesses of different algorithms, the more I realized that my original benchmarking application wasn't fit for purpose.

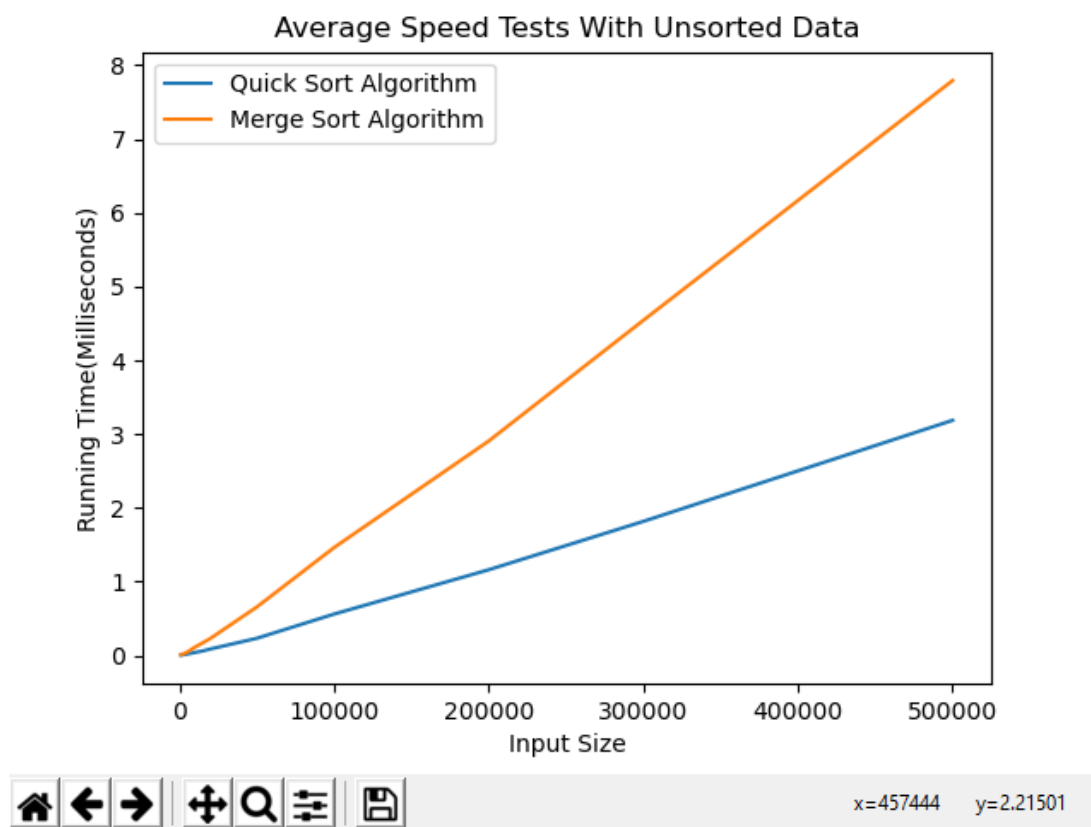
I discovered that the more I tested the python program which I designed to print results to screen and plot graphs, slower algorithms such as the Bubble Sort, on a graph would dwarf the results of other algorithms so much, that it made the rest look like they were on a straight line along the bottom. Also running the Bubble Sort algorithm on much larger data sets in conjunction with other algorithms could take days.

Slower algorithms such as the Bubble Sort dwarf the results of other algorithms



Towards the end of this assignment I recognised that in order to prove whether the text book assertions of Quicksort running faster with smaller data sets and Merge Sort running faster with larger data sets, I would need to create a bench marking tool that could isolate just those two algorithms and run small and large tests on both. I have tested datasets upto 500,000 records and still haven't proved that Merge Sort can be quicker than Quicksort.

Figure 1



Similarly there were text book assertions that Bubble Sort is one of the fastest algorithms for dealing with data already sorted in order. This assertion could only be proved if the user was offered the option to run tests with already sorted data.

There was also a text book assertion, that a list ordered in reverse, for Bubble Sort to put in non reverse order was worst case sorting scenario.

Later on I enhanced and redesigned the python program so that it could work with ten test sizes determined by the user, I also made it possible for the user to isolate or include the algorithms that they wanted to test.

```
Enter a value for test size 11000
Enter a value for test size 22000
Enter a value for test size 33000
Enter a value for test size 44000
Enter a value for test size 55000
Enter a value for test size 66000
Enter a value for test size 77000
Enter a value for test size 88000
Enter a value for test size 99000
Enter a value for test size 1010000
Do you want to run test with Bubble Sort? Y or N.Y
Do you want to run test with Quick Sort? Y or N.Y
Do you want to run test with Radix Sort? Y or N.Y
Do you want to run test with Merge Sort? Y or N.Y
Do you want to run test with Tim Sort? Y or N.Y
Do you want to run test with Python_Inbuilt Sort? Y or N.Y
How many tests do you want to run?5
Do you want to run test with unsorted data(U), reversed data(R) or sorted data(S)? Choose U, R or S.U
```

My initial design process is described below.

Implementation and design of the python code used for benchmarking.

STEP 1 – Setup variables for containing benchmark results

Initially I setup the test benchmark range in list of the following values.

testsizes = [500,1000,1250,2500,3750,5000,6250,7500,8750,10000]

Although the above was interesting for speed comparisons, I wanted to see if I could see any sizeable difference in memory usage comparisons. After running tests with larger test sizes which took over two days to run where I ran into processing problems, I finally settled with the test sizes below.

testsizes = [1000,2500,5000,7500,10000,12500,15000,20000,25000,50000]

After running tests with larger test sizes which took over two days to run where I ran into processing problems

Variable lists such as the one in the table below were also required to capture the timed test results, so that an average of ten tests could be assessed later. I created an empty variable list for capturing each type of timed test result per an algorithm, e.g.

bubble1000 = [] would later contain the ten timed test results of how quickly the bubble algorithm could sort 1000 random numbers, bubble2500 = [] would later contain the ten

timed test results of how quickly the bubble algorithm could sort 2500 random numbers, and so on, for each test number range up to 50,000 and for each algorithm type being benchmarked.

bubble500 = [] bubble1000 = [] bubble1250 = [] bubble2500 = [] bubble3750 = [] bubble5000 = [] bubble6250 = [] bubble7500 = [] bubble8750 = [] bubble10000 = []	quick500 = [] quick1000 = [] quick1250 = [] quick2500 = [] quick3750 = [] quick5000 = [] quick6250 = [] quick7500 = [] quick8750 = [] quick10000 = []	radix500 = [] radix1000 = [] radix1250 = [] radix2500 = [] radix3750 = [] radix5000 = [] radix6250 = [] radix7500 = [] radix8750 = [] radix10000 = []
merge500 = [] merge1000 = [] merge1250 = [] merge2500 = [] merge3750 = [] merge5000 = [] merge6250 = [] merge7500 = [] merge8750 = [] merge10000 = []	timsort500 = [] timsort1000 = [] timsort1250 = [] timsort2500 = [] timsort3750 = [] timsort5000 = [] timsort6250 = [] timsort7500 = [] timsort8750 = [] timsort10000 = []	python500 = [] python1000 = [] python1250 = [] python2500 = [] python3750 = [] python5000 = [] python6250 = [] python7500 = [] python8750 = [] python10000 = []

*** Later on I enhanced and redesigned the python program so that it could work with ten test sizes determined by the user, and moved away from naming variable lists set sizes.

I moved to using user input to set the sizes of the test size.

```

testsize1 = int(input("Enter a value for test size 1"))
testsize2 = int(input("Enter a value for test size 2"))
testsize3 = int(input("Enter a value for test size 3"))
testsize4 = int(input("Enter a value for test size 4"))
testsize5 = int(input("Enter a value for test size 5"))
testsize6 = int(input("Enter a value for test size 6"))
testsize7 = int(input("Enter a value for test size 7"))
testsize8 = int(input("Enter a value for test size 8"))
testsize9 = int(input("Enter a value for test size 9"))
testsize10 = int(input("Enter a value for test size 10"))

```

STEP 2 – Generate random numbers

I used the code provided to generate random numbers. This code was later referenced in STEP 3, where the testsizes were iterated and passed to n, and resulting random numbers would later be used.

```
from random import randint #import of randint required for function below  
def random_array(n):  
    array= []  
    for i in range (0,n,1):  
        array.append(randint(0,n))  
    return array
```

STEP 3 – Create function for generating random arrays

I setup the python code and loops to iterate ten times* through the testsizes list, calling first on the [random_array\(test\)](#) to generate random numbers based on the range sizes listed in testsizes list, e.g. from generating 500 random numbers, right through to generating 10,000 random numbers.

*the iteration was done ten times because an average result of ten tests was required later. The next process was to iterate the returned random numbers for the [random_array](#) to the [main_testing_algorithm](#).

```
for test in testsizes:  
    testsize = test  
    count = 0  
    while count < 10:  
        count += 1  
        array_to_sort = random_array(test)  
        main_testing_algorithm(array_to_sort)  
        print("TEST " + str(count) + " OF TEST SIZE " + str(test))  
    count = 0
```

STEP 4 – Create container function to store the algorithms for testing

The **main_testing_algorithm(array_to_sort)** was basically a container function to send the randomised numbers generated by **random_array(test)** to five algorithms which it stored and ran sequentially.

The main_testing_algorithm contained the following algorithms, **Bubble Sort, Quick Sort, Radix Sort, Merge Sort, My Own Algorithm**, also for benchmarking purposes I also included **Python's Built-in Sort Function** to carry out a sort of the same test data.

The first part of building the **main_testing_algorithm** was essentially a copy and paste exercise. As a data science student with little experience in algorithm design, I followed the advice of the assignment and made use of external material for **Bubble Sort, Quick Sort, Radix Sort, Merge Sort, Python's Built-in Sort Function**. I also designed my own algorithm.

All of the individual algorithms which I found done by other Python programmers on the internet, as well as one which I originally wrote myself were copy pasted one below the other into the **main_testing_algorithm**.

STEP 5 – Edits of individual algorithms

Edit1 – Algorithms edited to receive random data passed to main function

With fairly minimal Python programming skills, each individual algorithm had to be edited to receive the data passed to it. Original examples taken from the internet usually referenced their own randomised list of data. Where algorithms referenced other lists of data, these were edited to reference **array_to_sort**.

Edit 2 – Adding timers to each algorithm and capturing timed tests.

Before each algorithm began its sort, it required a start timer at the beginning, and an end timer at the end. Start timers were placed as close as possible to point of function that received the random array of data, end timers placed after the last step of the sort before any other reporting.

```
import time #used at the beginning of the python program
start_time = time.time() #used at beginning of every algorithm
end_time = time.time() #used at end of every algorithm
```


To capture how long a timed test would take the following code was used at the end of each algorithm.

time_elapsed = end_time - start_time

To minus end_time from – start_time gave result of how quickly th

time_elapsed = float(time_elapsed)

I converted time_elapsed to a float, because at one point in the algorithm design I discovered that there were errors in calculation reporting occurred because the time_elapsed results were being passed as strings rather than as numbers.

Edit 3 – Storing timed tests

The end of each algorithm was edited to store the timed result to a variable. After an individual algorithm had run its operation and time had been captured, code was used to append the timed result to the correct test size and algorithm list for analysis later, e.g. if test size dataset of 500 numbers was run on the bubble algorithm, the timed result was appended to the bubble500 list, if the test size dataset was 1000 numbers it would be bubble1000 list.

```
if testsize == 500:
```

```
    bubble500.append(time_elapsed)
```

```
elif testsize == 1000:
```

```
    bubble1000.append(time_elapsed)
```

Before each algorithm began its sort, it required a start timer at the beginning, and an end timer at the end. Start timers were placed as close as possible to point of function that received the random array of data, end timers placed after the last step of the sort before any other reporting.

STEP 6 – Reporting

For the algorithm benchmarking application to be able to do reports and produce tidy text. The requirements were achieved as follows, by use of imports at the beginning of the code, and print formatting and graph plotting code at the bottom.

Correct Imports:

```
#import for calculating averages and other potential statistical analysis
import statistics
```

```
#imports needed for plotting graph
import matplotlib.pyplot as plt; plt.rcParamsdefaults()
import numpy as np
import matplotlib.pyplot as plt
```

Use Of Python Print Formatting:

Print formatting code used to align text right > with 3 decimal places .3f, using the `mean` function from statistics to perform an average calculation on the ten test results stored in each variable created in STEP 1.

Sample code

```
print('{:30} {:>7} {:>7} {:>7} {:>7} {:>7} {:>7} {:>7} {:>7} {:>7}'.format("Size",
"500","1000","1250","2500","3750","5000","6250","7500","8750","10000"))
print('{:30} {:7.3f} {:7.3f} {:7.3f} {:7.3f} {:7.3f} {:7.3f} {:7.3f} {:7.3f} {:7.3f}'.format("Bubble
Sort Algorithm",
statistics.mean(bubble500),statistics.mean(bubble1000),statistics.mean(bubble1250),statistics.me
an(bubble2500),statistics.mean(bubble3750),statistics.mean(bubble5000),statistics.mean(bubble
6250),statistics.mean(bubble7500),statistics.mean(bubble8750),statistics.mean(bubble10000)))
```

Result

[illegible]

```

plt.plot([500,1000,1250,2500,3750,5000,6250,7500,8750,10000],[statistics.mean(bubble500),statistics.mean(bubble1000),statistics.mean(bubble1250),statistics.mean(bubble2500),statistics.mean(bubble3750),statistics.mean(bubble5000),statistics.mean(bubble6250),statistics.mean(bubble7500),statistics.mean(bubble8750),statistics.mean(bubble10000)],label='Bubble Sort Algorithm')

plt.plot([500,1000,1250,2500,3750,5000,6250,7500,8750,10000],[statistics.mean(quick500),statistics.mean(quick1000),statistics.mean(quick1250),statistics.mean(quick2500),statistics.mean(quick3750),statistics.mean(quick5000),statistics.mean(quick6250),statistics.mean(quick7500),statistics.mean(quick8750),statistics.mean(quick10000)],label='Quick Sort Algorithm')

plt.plot([500,1000,1250,2500,3750,5000,6250,7500,8750,10000],[statistics.mean(radix500),statistics.mean(radix1000),statistics.mean(radix1250),statistics.mean(radix2500),statistics.mean(radix3750),statistics.mean(radix5000),statistics.mean(radix6250),statistics.mean(radix7500),statistics.mean(radix8750),statistics.mean(radix10000)],label='Radix Sort Algorithm')

plt.plot([500,1000,1250,2500,3750,5000,6250,7500,8750,10000],[statistics.mean(merge500),statistics.mean(merge1000),statistics.mean(merge1250),statistics.mean(merge2500),statistics.mean(merge3750),statistics.mean(merge5000),statistics.mean(merge6250),statistics.mean(merge7500),statistics.mean(merge8750),statistics.mean(merge10000)],label='Merge Sort Algorithm')

plt.plot([500,1000,1250,2500,3750,5000,6250,7500,8750,10000],[statistics.mean(chris500),statistics.mean(chris1000),statistics.mean(chris1250),statistics.mean(chris2500),statistics.mean(chris3750),statistics.mean(chris5000),statistics.mean(chris6250),statistics.mean(chris7500),statistics.mean(chris8750),statistics.mean(chris10000)],label='Chris Sort Algorithm')

plt.plot([500,1000,1250,2500,3750,5000,6250,7500,8750,10000],[statistics.mean(python500),statistics.mean(python1000),statistics.mean(python1250),statistics.mean(python2500),statistics.mean(python3750),statistics.mean(python5000),statistics.mean(python6250),statistics.mean(python7500),statistics.mean(python8750),statistics.mean(python10000)],label='Python Inbuilt-Sort')

plt.legend()

plt.xlabel("Input Size")

plt.ylabel("Running Time(Milliseconds)")

plt.show()

```

Enhancements:

With the initial design in place, I was able to expand the program to be able to test and report other things.

The user would be able to set their own variable of how many runs the program would do.

The user would be able to choose whether the algorithms would work with unsorted data, reversed data or sorted data:

```
#lets users choose whether to test algorithms with unsorted data, reversed data or sorted data
```

```
Sorted_Or_Unsorted = input("Do you want to run test with unsorted data(U), reversed data(R) or sorted data(S)? Choose U, R or S.")
```

```
while Sorted_Or_Unsorted != "U" or "S":
```

```
    if Sorted_Or_Unsorted == "U":  
        unsorted_algorithm()  
        break
```

```
    if Sorted_Or_Unsorted == "R":  
        reversed_algorithm()  
        break
```

```
    elif Sorted_Or_Unsorted == "S":  
        sorted_algorithm()  
        break
```

```
    else:  
        Sorted_Or_Unsorted = input("Do you want to run test with unsorted data(U) or sorted data(S)? Choose U or S.")
```

Testing unsorted data in the benchmark
unsorted data,

```
#this function is used for creating randomised unsorted data arrays
def random_array(n):
    array= []
    for i in range (0,n,1):
        array.append(randint(0,n))
    return array
```

Testing already sorted data in the benchmark
already sorted data,

```
#this function is used for creating sorted data arrays
def sorted_array(n):
    array= []
    for i in range (0,n,1):
        array.append(randint(0,n))
    array.sort() #sorts numbers in order
    return array
```

Testing data sorted in reverse in the benchmark
data sorted in reverse, so every element of **n** would be out of place.

```
#this function is used for creating sorted data arrays
def reversed_array(n):
    array= []
    for i in range (0,n,1):
        array.append(randint(0,n))
    array.sort(reverse = True) #sorts numbers in reverse order
    return array
```

I also added a memory usage monitoring reporting python library, and provide details of this for space complexity benchmarking.

```
#imports needed for measuring memory
import os
import psutil

process = psutil.Process(os.getpid()) #used to initiate memory usage recording

bubbletestsiz1memory.append(float(process.memory_percent())) #used to append
memory usage recording to list
```

Benchmarking computer specifications

I have also put a mechanism in the Python program to report the details of the operating system being used to test, so this benchmarking tool can be monitored on other operating systems.⁷⁹

It is necessary to do pip installs to get this part of the python program to work.

pip install --upgrade wmi

pip install --upgrade pypiwin32

```
import wmi

computer = wmi.WMI()
computer_info = computer.Win32_ComputerSystem()[0]
os_info = computer.Win32_OperatingSystem()[0]
proc_info = computer.Win32_Processor()[0]
gpu_info = computer.Win32_VideoController()[0]
os_name = os_info.Name.encode('utf-8').split(b'|')[0]
os_version = ''.join([os_info.Version, os_info.BuildNumber])
system_ram = float(os_info.TotalVisibleMemorySize) / 1048576 # KB to GB
```

Printing the operating system

```
print("Algorithms being tested on:")
print('\nOS Name:', os_name, '\nOS Version:', os_version, '\nCPU:', proc_info.Name,
'\nRAM:', system_ram, 'GB', '\nGraphics Card:', gpu_info.Name)
```

Ensuring Data Integrity

I made duplicate copies of the array to sort, to ensure that each algorithm was tested on the same data.

```
def main_testing_algorithm(array_to_sort):

    #ensure all tests are the same
    #copy original array pre sort into duplicate arrays to be sorted for each tested algorithm
    array_to_sortB = []
    array_to_sortQ = []
    array_to_sortR = []
    array_to_sortM = []
    array_to_sortT = []
    array_to_sortP = []

    for i in array_to_sort:
        array_to_sortB.append(i)
        array_to_sortQ.append(i)
        array_to_sortR.append(i)
        array_to_sortM.append(i)
        array_to_sortT.append(i)
        array_to_sortP.append(i)
```

⁷⁹ <https://stackoverflow.com/questions/38103690/get-system-informationcpu-speed-total-ram-graphic-card-model-etc-under-window>

In Summary

The final bench marking tool could take variables from user input.

So user could determine

- size of ten benchmarking tests,
- which of five algorithms to include
- whether to run tests with unsorted data, reversed sorted data or sorted data

```
*Python 3.7.6 Shell*
File Edit Shell Debug Options Window Help
>>>
= RESTART: C:/Users/Owner/Desktop/algorithms/AlgorithmsAssignment_Final_03052020.py
Enter a value for test size 11000
Enter a value for test size 22500
Enter a value for test size 35000
Enter a value for test size 47500
Enter a value for test size 510000
Enter a value for test size 612500
Enter a value for test size 715000
Enter a value for test size 820000
Enter a value for test size 925000
Enter a value for test size 1050000
Do you want to run test with Bubble Sort? Y or N.Y
Do you want to run test with Quick Sort? Y or N.Y
Do you want to run test with Radix Sort? Y or N.Y
Do you want to run test with Merge Sort? Y or N.Y
Do you want to run test with Tim Sort? Y or N.Y
Do you want to run test with Python_Inbuilt Sort? Y or N.Y
How many tests do you want to run?1
Do you want to run test with unsorted data(U), reversed data(R) or sorted data(S)? Choose U, R or S.U
TEST 1 OF TEST SIZE 1000
TEST 1 OF TEST SIZE 2500
TEST 1 OF TEST SIZE 5000
TEST 1 OF TEST SIZE 7500
TEST 1 OF TEST SIZE 10000
TEST 1 OF TEST SIZE 12500
TEST 1 OF TEST SIZE 15000
TEST 1 OF TEST SIZE 20000
TEST 1 OF TEST SIZE 25000
TEST 1 OF TEST SIZE 50000
```

The final benchmarking tool reported mean, min, max for both speed and memory usage for each algorithm tested.

The final benchmarking tool reported the operating system that the user was using to test the algorithms.

It was able to print to the results to screen and present as line graphs.

Algorithms being tested on:

OS Name: b'Microsoft Windows 10 Pro'

OS Version: 10.0.18363.18363

CPU: Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz

RAM: 11.85890579236328 GB

Graphics Card: Intel(R) HD Graphics 4000

Tests With Unsorted Data

Speed Test

Size	1000	2500	5000	7500	10000	12500	15000	20000	25000	50000
Bubble Sort Algorithm	0.140	0.905	3.625	8.512	15.276	23.482	33.833	62.617	98.266	404.944
QuickSort Algorithm	0.003	0.008	0.018	0.018	0.039	0.053	0.063	0.086	0.094	0.266
Radix Sort Algorithm	0.002	0.008	0.016	0.041	0.042	0.051	0.067	0.095	0.125	0.250
Merge Sort Algorithm	0.007	0.020	0.044	0.078	0.101	0.132	0.182	0.251	0.312	0.672
timSort Sort Algorithm	0.006	0.018	0.040	0.087	0.092	0.139	0.166	0.207	0.250	0.594
Python in-built sort	0.000	0.000	0.001	0.001	0.001	0.002	0.000	0.000	0.016	0.016

Tests With Unsorted Data

Memory Test

Size	1000	2500	5000	7500	10000	12500	15000	20000	25000	50000
Bubble Sort Algorithm	0.463	0.466	0.474	0.478	0.481	0.484	0.487	0.496	0.498	0.525
QuickSort Algorithm	0.463	0.468	0.475	0.478	0.483	0.486	0.491	0.496	0.500	0.512
Radix Sort Algorithm	0.463	0.468	0.475	0.478	0.483	0.486	0.491	0.496	0.500	0.515
Merge Sort Algorithm	0.463	0.468	0.475	0.478	0.483	0.486	0.491	0.496	0.500	0.525
timSort Sort Algorithm	0.464	0.470	0.476	0.479	0.479	0.484	0.487	0.492	0.499	0.526
Python in-built sort	0.464	0.470	0.476	0.479	0.480	0.484	0.487	0.492	0.499	0.526

Line graph screenshots shown throughout this assignment.

