PYTHON APPLICATION (40%):

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

INTRODUCTION (10%)

Inversion is where two elements in a dataset are out of order, some algorithms perform better on almost sorted data sets

**FOREWORD**

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](https://www.domo.com/learn/data-never-sleeps-5?aid=ogsm072517_1&sf100871281=1) created each day at our current pace, but that pace is only accelerating with the growth of the Internet of Things (IoT).” [[1]](#footnote-1)

The [LinkedIn Workforce Report](https://economicgraph.linkedin.com/resources/linkedin-workforce-report-december-2017) 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**.[[2]](#footnote-2)** 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](https://web-assets.domo.com/blog/wp-content/uploads/2017/07/17_domo_data-never-sleeps-5-01.png), 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.[[3]](#footnote-3)

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 gigagbytes!) 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.[[4]](#footnote-4)

**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.’[[5]](#footnote-5)

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.’[[6]](#footnote-6)

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.[[7]](#footnote-7)

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 algorithms that provide a set of instructions for how a program or system should go about organizing a set of data.’[[8]](#footnote-8)

INPLACE – Require a fixed amount of space for sorting … does not depend on the input dataset

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.’ [[9]](#footnote-9)

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, divided into categories of comparison-based sort and 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. Only advanced data scientists will go further in challenging the most efficient way.

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.[[10]](#footnote-10)

Order of growth is how the time of execution depends on the length of the input. In the above example, we can clearly see that the time of execution is linearly depends on the length of the array. Order of growth will help us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behaviour of a function.

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.

**Space Complexity**

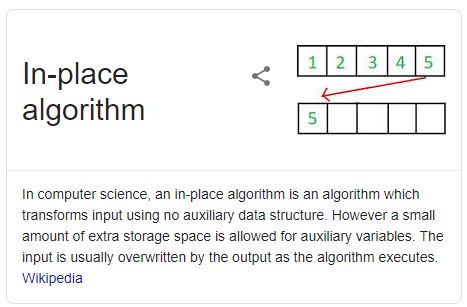
Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.[[11]](#footnote-11)

**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.[[12]](#footnote-12)

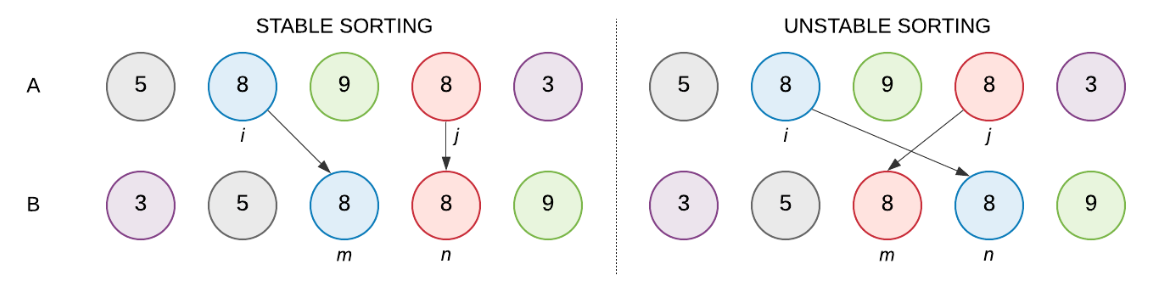
**In-place sorting,**

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.[[13]](#footnote-13)



**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.[[14]](#footnote-14)

****

**comparator functions,**

**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.[[15]](#footnote-15)

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.[[16]](#footnote-16)

**and 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.[[17]](#footnote-17)

PART 2 - SORTING ALGORITHMS (5 x 5 = 25%)

For this assignment, I will first be providing reviews of the following algorithms.

**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.[[18]](#footnote-18)
* This simple algorithm performs poorly in real world use and is used primarily as an educational tool.[[19]](#footnote-19)
* I cannot find anyone who takes credit for inventing this algorithm. The term “Bubble Sort ” was used by Kenneth Iverson in 1962.[[20]](#footnote-20) 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](https://www.britannica.com/technology/computer-programming-language) called [APL](https://www.britannica.com/technology/APL) (the initials of his book [A Programming Language](https://www.britannica.com/topic/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](https://www.britannica.com/science/mathematics) at [Harvard University](https://www.britannica.com/topic/Harvard-University) from 1955 to 1960 and served on the staff of the research division of [IBM](https://www.britannica.com/topic/International-Business-Machines-Corporation) from 1960 to 1980.[[21]](#footnote-21)

**Merge Sort** – 1945 – John von Neumann

* Merge Sort - An efficient comparison-based sort
* Merge sort is a [divide and conquer algorithm](https://en.wikipedia.org/wiki/Divide_and_conquer_algorithm) that was invented by [John von Neumann](https://en.wikipedia.org/wiki/John_von_Neumann) in 1945. A detailed description and analysis of bottom-up mergesort appeared in a report by [Goldstine](https://en.wikipedia.org/wiki/Herman_Goldstine" \o "Herman Goldstine) and [von Neumann](https://en.wikipedia.org/wiki/John_von_Neumann) as early as 1948. [[22]](#footnote-22)
* 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.[[23]](#footnote-23)

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. [[24]](#footnote-24)*

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. [[25]](#footnote-25)
* 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 tape[[26]](#footnote-26). To fulfill this task he discovered Quick Sort and later published the code in 1961.[[27]](#footnote-27) [[28]](#footnote-28)
* *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.[[29]](#footnote-29)*

Timsort – 2002 – Tim Peters

* Python uses an algorithm called Timsort: Timsort is a hybrid sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It was invented by Tim Peters in 2002 for use in the Python programming language.[[30]](#footnote-30)
* Timsort is a [hybrid](https://en.wikipedia.org/wiki/Hybrid_algorithm) [stable](https://en.wikipedia.org/wiki/Category:Stable_sorts) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm), derived from [merge sort](https://en.wikipedia.org/wiki/Merge_sort) and [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort), designed to perform well on many kinds of real-world data. It was implemented by [Tim Peters](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)) in 2002 for use in the [Python programming language](https://en.wikipedia.org/wiki/Python_(programming_language)). The algorithm finds subsequences of the data that are already ordered (runs) and uses them to sort the remainder more efficiently. This is done by merging runs until certain criteria are fulfilled. Timsort has been Python's standard sorting algorithm since version 2.3. It is also used to sort arrays of non-primitive type in [Java SE 7](https://en.wikipedia.org/wiki/Java_7),[[4]](https://en.wikipedia.org/wiki/Timsort#cite_note-4) on the [Android platform](https://en.wikipedia.org/wiki/Android_(operating_system)),[[5]](https://en.wikipedia.org/wiki/Timsort#cite_note-5) in [GNU Octave](https://en.wikipedia.org/wiki/GNU_Octave),[[6]](https://en.wikipedia.org/wiki/Timsort#cite_note-6) on [V8](https://en.wikipedia.org/wiki/V8_(JavaScript_engine)),[[7]](https://en.wikipedia.org/wiki/Timsort#cite_note-7) and [Swift](https://en.wikipedia.org/wiki/Swift_(programming_language))[[8]](https://en.wikipedia.org/wiki/Timsort#cite_note-8).
* It uses techniques from [Peter McIlroy](https://en.wikipedia.org/w/index.php?title=Peter_McIlroy&action=edit&redlink=1)'s 1993 paper "Optimistic Sorting and Information Theoretic Complexity".[[9]](https://en.wikipedia.org/wiki/Timsort#cite_note-9)
* Tim Peters is an American[[citation needed](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] software developer who created the [Timsort](https://en.wikipedia.org/wiki/Timsort" \o "Timsort) [hybrid](https://en.wikipedia.org/wiki/Hybrid_algorithm) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm), is a major contributor to the [Python](https://en.wikipedia.org/wiki/Python_(programming_language)) programming language and its original [CPython](https://en.wikipedia.org/wiki/CPython" \o "CPython) implementation, and is prominent in the Python community. A pre-1.0 CPython user,[[1]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-1) he was among the group of early adopters who contributed to the detailed design of the language in its early stages.[[2]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-2)
* He later created the Timsort algorithm (based on earlier work on the use of ["galloping" search](https://en.wikipedia.org/wiki/Exponential_search))[[3]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-3) which has been used in Python since version 2.3[[citation needed](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)], as well as in other widely used computing platforms,[[4]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-4)[[5]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-5)[[6]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-6) including the [V8 JavaScript engine](https://en.wikipedia.org/wiki/V8_(JavaScript_engine)) powering the [Google Chrome](https://en.wikipedia.org/wiki/Google_Chrome) and [Chromium](https://en.wikipedia.org/wiki/Chromium) web browsers, as well as [Node.js](https://en.wikipedia.org/wiki/Node.js).[[7]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-7)[[8]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-8) He has also contributed the [doctest](https://en.wikipedia.org/wiki/Doctest" \o "Doctest) and timeit modules to the Python standard library.[[9]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-DSA_announcement-9)
* Peters also wrote the [Zen of Python](https://en.wikipedia.org/wiki/Zen_of_Python), intended as a statement of Python's design philosophy, which was incorporated into the official Python literature as [Python Enhancement Proposal](https://en.wikipedia.org/wiki/Python_Enhancement_Proposal) 20 and in the Python interpreter as an [easter egg](https://en.wikipedia.org/wiki/Easter_egg_(media)" \o "Easter egg (media)).[[10]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-10) He contributed the [chapter on algorithms](https://www.safaribooksonline.com/library/view/python-cookbook/0596001673/ch17.html) to the Python Cookbook.[[11]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-Python_Cookbook_1st_ed.-11) From 2001 to 2014 he was active as a member of the [Python Software Foundation](https://en.wikipedia.org/wiki/Python_Software_Foundation)'s board of directors. Peters was an influential contributor to Python mailing lists.[[12]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-Happy_Medium_DSA_blogpost-12) He is also a highly ranked contributor to [Stack Overflow](https://en.wikipedia.org/wiki/Stack_Overflow), mostly for answers relating to Python.[[13]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-13)[[9]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-DSA_announcement-9)
* Peters' past employers include [Kendall Square Research](https://en.wikipedia.org/wiki/Kendall_Square_Research).[[14]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-14)[[15]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-15) Tim Peters was granted the Python Software Foundation's [Distinguished Service Award for 2017](https://www.python.org/community/awards/psf-distinguished-awards/#id1).[[9]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-DSA_announcement-9)[[12]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)#cite_note-Happy_Medium_DSA_blogpost-12)[[31]](#footnote-31)

**Python's Built-in Sort Functions**

**Bubble Sort – A more detailed explanation**

**Bubble Sort - 1962 - A simple comparison-based sort**

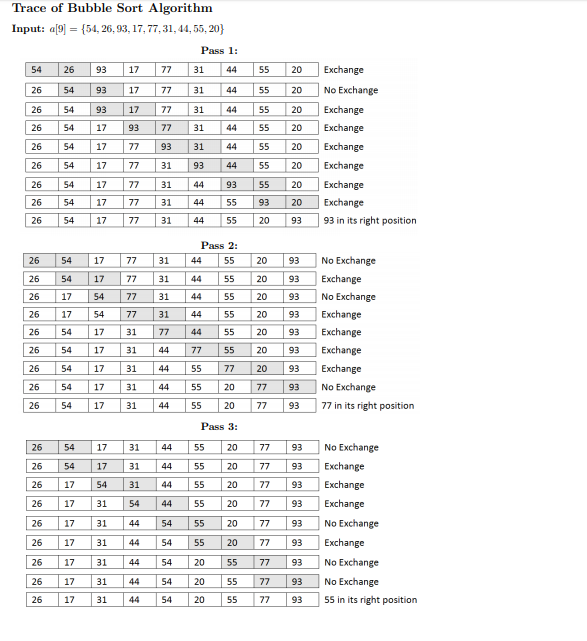
*The Bubble Sort Algorithm is a simple sorting algorithm that iterates over a list, comparing elements in pairs and swapping them until the larger elements "bubble up" to the end of the list, and the smaller elements stay at the "bottom".[[32]](#footnote-32)*

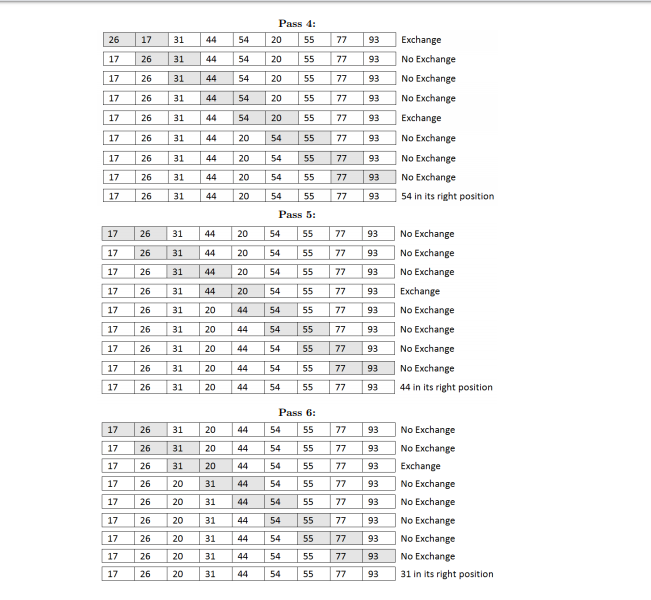
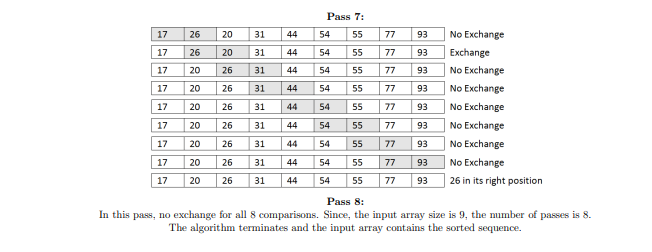
**Performance**

Bubble sort has a worst-case and average complexity of [О](https://en.wikipedia.org/wiki/Big_o_notation)(n2), 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 О(n2) sorting algorithms, such as [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort), generally run faster than bubble sort, and are no more complex. Therefore, bubble sort is not a practical sorting algorithm.

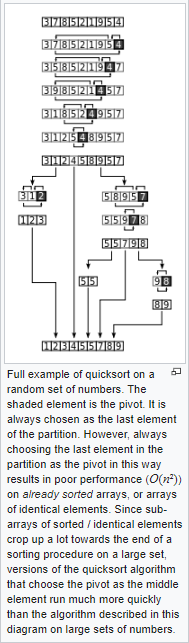
The only significant advantage that bubble sort has over most other algorithms, even [quicksort](https://en.wikipedia.org/wiki/Quicksort), but not [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort), is that the ability to detect that the list is sorted efficiently is built into the algorithm. 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](https://en.wikipedia.org/wiki/Average-case_complexity), perform their entire sorting process on the set and thus are more complex. However, not only does [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort) share this advantage, but it also performs better on a list that is substantially sorted (having a small number of [inversions](https://en.wikipedia.org/wiki/Inversion_(discrete_mathematics))).

Bubble sort should be avoided in the case of large collections. It will not be efficient in the case of a reverse-ordered collection.[[33]](#footnote-33)

****

****

**Quick Sort - A more detailed explanation**

****

**Radix Sort - 1954 - A more detailed explanation**

*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.*[[34]](#footnote-34)

*In many applications requiring super speeds and massive memory, the computer radix sort invented by Seward supersedes earlier slower comparison sorts.* [[35]](#footnote-35) *The Radix Sort algorithm was originally used to sort*[*punched cards*](http://encyclopedia.kids.net.au/page/pu/Punched_card)*in several passes. [[36]](#footnote-36)*

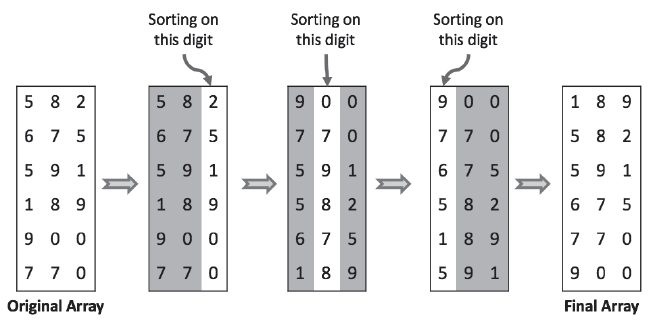
|  |
| --- |
| *The* ***punch card*** *(or "Hollerith" card) is a medium for holding information for use by automated machines. Made of stiff cardboard, the punch card represents information by the presence or absence of holes in predefined positions on the card. In the first generation of computing 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.[[37]](#footnote-37)* |

[Radix Sort](https://brilliant.org/wiki/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. [[38]](#footnote-38)

Radix Sort groups keys by individual digits that share the same significant position and value.[[39]](#footnote-39)

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.’*[[40]](#footnote-40)

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



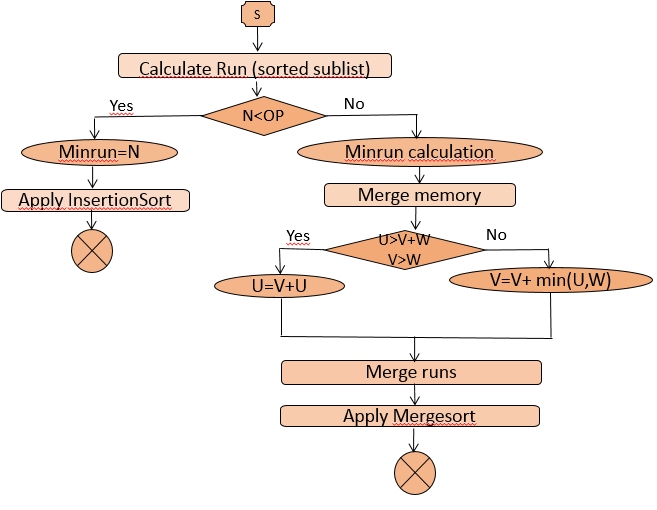
*Each invocation of the Counting Sort subroutine preserves the order from the previous invocations. [[41]](#footnote-41)*

**Merge Sort – A more detailed explanation**

|  |  |  |
| --- | --- | --- |
| **Wikipdeia Diagram Of Merge Sort [[42]](#footnote-42)** |  | Merge-Sort-Tutorial  **Wikipdeia Diagram Of Merge Sort Analysed**  The following diagram from [wikipedia](http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg" \t "_blank) 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.[[43]](#footnote-43) |

**My Own Algorithm – A more detailed explanation**

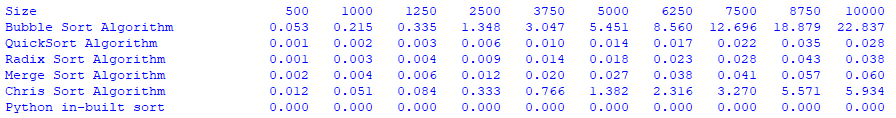
**Python's Built-in Sort Functions Summary**

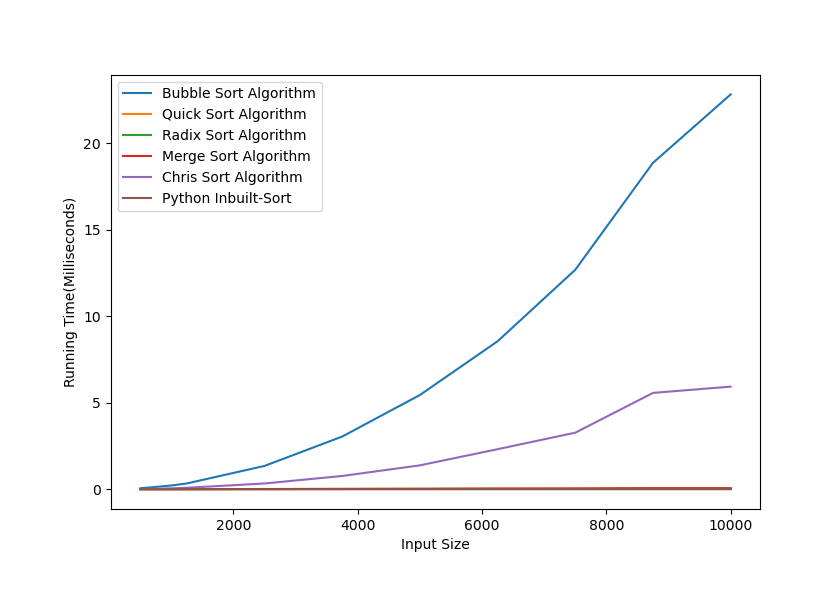
****

**Python's Built-in Sort Functions Summary**

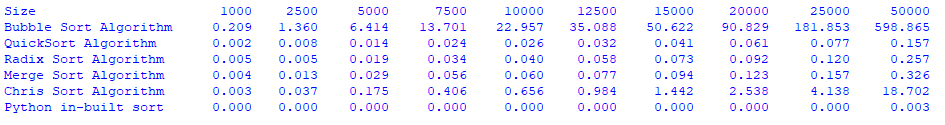
Unless you are an algorithm / programming expert, it is likely that a person operating in a role as a junior analyst is more likely to default to using the inbuilt sort functions within Python or programming software of company’s choice. The designers of Python / other software have had to pre-empt that users will default to the most expedient sort option available to them, and provide the best inbuilt sorting algorithm to cover most cases. With the advantages and disadvantages of the previous algorithms discussed, it should be no surprise that Python has integrated a hybrid algorithm called the Tim Sort algorithm, inspired by the methods of Merge Sort and Insertion Sort. [[44]](#footnote-44)

When I began benchmarking tests of sorting small data sizes of up to 10,000 numbers, where the requirement was to show sorting calculation speed in milliseconds up to three decimal points, the Python in-built sort out performed all other algorithms, with no fractions of a second being recorded.





PART 3 – IMPLEMENTATION AND BENCHMARKING



Implementation and design of the python code used for benchmarking.

STEP 1

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 testsizes 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 testsizes 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 = [] | chris500 = []  chris1000 = []  chris1250 = []  chris2500 = []  chris3750 = []  chris5000 = []  chris6250 = []  chris7500 = []  chris8750 = []  chris10000 = [] | python500 = []  python1000 = []  python1250 = []  python2500 = []  python3750 = []  python5000 = []  python6250 = []  python7500 = []  python8750 = []  python10000 = [] |

STEP 2

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

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

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 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 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.rcdefaults()  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} {:>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} {:7.3f}'.format("Bubble Sort Algorithm", 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)))  Result |

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()

Implementation & Benchmarking (25%): This section will describe the process followed when implementing the application above, and will present the results of your benchmarking. Discuss how the measured performance of the algorithms differed – were the results similar to what you would expect, given the time complexity of each chosen algorithm? In this section you should use both a table and a graph to summarise the results obtained (see samples below).

1. <https://blazon.online/data-marketing/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/> [↑](#footnote-ref-1)
2. <https://www.morningfuture.com/en/article/2018/02/21/data-analyst-data-scientist-big-data-work/235/> [↑](#footnote-ref-2)
3. <https://blazon.online/data-marketing/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/> [↑](#footnote-ref-3)
4. <http://archive.oreilly.com/oreillyschool/courses/data-structures-algorithms/largeData.html> [↑](#footnote-ref-4)
5. <https://en.wikipedia.org/wiki/Sorting> [↑](#footnote-ref-5)
6. <https://medium.com/basecs/sorting-out-the-basics-behind-sorting-algorithms-b0a032873add> [↑](#footnote-ref-6)
7. <https://www.quora.com/Where-does-the-word-algorithm-come-from> [↑](#footnote-ref-7)
8. <https://medium.com/basecs/sorting-out-the-basics-behind-sorting-algorithms-b0a032873add> [↑](#footnote-ref-8)
9. <http://archive.oreilly.com/oreillyschool/courses/data-structures-algorithms/largeData.html> [↑](#footnote-ref-9)
10. <https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/> [↑](#footnote-ref-10)
11. <https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/> [↑](#footnote-ref-11)
12. <http://www.btechsmartclass.com/data_structures/performance-analysis.html> [↑](#footnote-ref-12)
13. <https://stackoverflow.com/questions/16585507/sorting-in-place> [↑](#footnote-ref-13)
14. <https://www.baeldung.com/cs/stable-sorting-algorithms> [↑](#footnote-ref-14)
15. <https://en.wikipedia.org/wiki/Comparison_sort> [↑](#footnote-ref-15)
16. <https://diego.assencio.com/?index=143ab4f37feadc3ac69f33f43fde3a2a> [↑](#footnote-ref-16)
17. <https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html> [↑](#footnote-ref-17)
18. <https://en.wikipedia.org/wiki/Bubble_sort> [↑](#footnote-ref-18)
19. <https://en.wikipedia.org/wiki/Bubble_sort> [↑](#footnote-ref-19)
20. Iverson, K: A Programming Language. John Wiley, 1962. [↑](#footnote-ref-20)
21. <https://www.britannica.com/topic/A-Programming-Language> [↑](#footnote-ref-21)
22. <https://en.wikipedia.org/wiki/Merge_sort> [↑](#footnote-ref-22)
23. <https://stackabuse.com/sorting-algorithms-in-python/#mergesort> [↑](#footnote-ref-23)
24. <https://www.programiz.com/dsa/radix-sort> [↑](#footnote-ref-24)
25. <https://en.wikipedia.org/wiki/Quicksort> [↑](#footnote-ref-25)
26. Shustek, L: Interview: An interview with C.A.R. Hoare. Comm. ACM 52 (3): 3841, (2009). [↑](#footnote-ref-26)
27. Hoare, C. A. R: Algorithm 64: Quicksort. Comm. ACM 4 (7): 321, (1961). [↑](#footnote-ref-27)
28. <http://www.iiitdm.ac.in/old/Faculty_Teaching/Sadagopan/pdf/DAA/SortingAlgorithms.pdf> [↑](#footnote-ref-28)
29. <https://stackabuse.com/sorting-algorithms-in-python/#quicksort> [↑](#footnote-ref-29)
30. <https://stackoverflow.com/questions/10948920/what-algorithm-does-pythons-sorted-use> [↑](#footnote-ref-30)
31. <https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)> [↑](#footnote-ref-31)
32. <https://stackabuse.com/sorting-algorithms-in-python/#bubblesort> [↑](#footnote-ref-32)
33. <https://en.wikipedia.org/wiki/Bubble_sort> [↑](#footnote-ref-33)
34. <https://en.wikipedia.org/wiki/Harold_H._Seward> [↑](#footnote-ref-34)
35. <http://encyclopedia.kids.net.au/page/pu/Punched_card> [↑](#footnote-ref-35)
36. <http://encyclopedia.kids.net.au/page/ra/Radix_sort> [↑](#footnote-ref-36)
37. <http://encyclopedia.kids.net.au/page/pu/Punched_card> [↑](#footnote-ref-37)
38. <https://www.baeldung.com/cs/stable-sorting-algorithms> [↑](#footnote-ref-38)
39. <https://www.baeldung.com/cs/stable-sorting-algorithms> [↑](#footnote-ref-39)
40. <http://www.ritambhara.in/radix-sort/> [↑](#footnote-ref-40)
41. <https://www.baeldung.com/cs/stable-sorting-algorithms> [↑](#footnote-ref-41)
42. <https://commons.wikimedia.org/wiki/File:Merge_sort_algorithm_diagram.svg> [↑](#footnote-ref-42)
43. <https://www.geeksforgeeks.org/merge-sort/> [↑](#footnote-ref-43)
44. <https://stackabuse.com/sorting-algorithms-in-python/#sortinginpython> [↑](#footnote-ref-44)