report

May 1, 2020

1 Project: Benchmarking Sorting Algorithms in Python

1.1 Table of Contents

2 Section 1. Introduction

In this section I will introduce the concept of sorting algorithms and discuss the relevance of concepts such as time and space complexity, performance, in-place sorting, stable sorting, comparator functions, comparison-based and non-comparison-based sorts.

Much of the information here is based on Algorithms in a Nutshell by George T. Heineman, Gary Pollice and Stanley Selkow and lecture notes for the Computational Thinking with Algorithms at GMIT.

Sorting is the process or operation of ordering items and data according to specific criteria. Arranging items, whether manually or digitally into some order makes many common tasks easier to do. Sorted information is nearly always easier for human beings to understand and work with, whether it is looking up a number in a phone book, finding a book on a library or bookshop shelf, referring to a statistical tables book, deciding what to watch on television. It is even more applicable with technology. Social media and news feeds, emails, search items in a browser or recommendations for where to eat to what to do are all sorted according to some algorithms whether it is by date, popularity, location etc.

Numerous computations and tasks can be simplified and made more efficient by having the data sorted in advance, such as searching for an item, checking for duplicate items, determining the smallest and largest values, or the most common or least common values. Sorting is a common operation in many computer applications and the search for efficient sorting algorithms dominated the early days of computing.

According to Algorithms to live by The Computer Science of Human Decisions[2], sorting is at the very heart of what computers do and is what actually brought computers into being in the first place. The task of tabulating the US Census in the late nineteenth century became very difficult as the population grew. An inventor by the name of Herman Hollerith was inspired by the punched railway tickets of the time to devise a system of punched cards and a machine (the Hollerith Machine) to count and sort them. This system was then used for the 1890 census. Hollerith's company later merged with others in 1911 to become the Computing Tabulating Recording Company which was later renamed to International Business Machines (IBM). Sorting then lay behind the development of the computer in the 20th century. By the 1960's it was estimated by one study that more than a quarter of the world's computing resouces were spent on sorting.

An algorithm is a set of rules to obtain the expected output from the given input. A computer programming task can be broken down into two main steps - writing the algorithm as an ordered sequence of steps that can be used to solve the problem and then implementing the sequenced of steps in a programming language such as Python so that the machine can run the algorithm. While humans can usually follow algorithms that are not very precise in order to accomplish a task, this is not the case for computers and therefore the algorithms have to be written very precisely.

There are many different ways that algorithms could be designed to achieve a similar goal. Therefore there needs to be some way of deciding which algorithm is preferable for a particular use case and therefore some way of comparing the alternative algorithms is required. A well-designed algorithm should produce the correct solution for a given input using computational resources efficiently. It should have well defined inputs and outputs and the algorithm should end after a finite number of steps. Every step of the algorithm should be precisely defined so there is no ambiguity as a computer can only do what it is instructed to do. The algorithm should always produce a correct solution, or at least one that is within an acceptable margin of error. The algorithm should be feasible giving the computational resources such as processing power, memory, storage etc.

Algorithms vary in their space and time efficiency, even those that have the same purpose such as sorting algorithms. While space efficiency looks at the amount of memory or storage needed to run a program/algorithm, time efficiency looks at the effect of the input data on the run time or number of operations needed to run an algorithm. An algorithms efficiency can be analysed in two different ways. A priori analysis (before) looks at the efficiency of an algorithm from a theoretical perspective without being concerned with the actual implementation on any particular machine. The relative efficiency of algorithms is analysed by comparing their *order of growth* with respect to the size of the input N and is a measure of the **complexity** of an algorithm, looking at the growth requirements as the input size increases. (The order of growth refers to how the size of the resource requirements increase as a function of the input size n. As input size increases so too does the number of operations required and the work required). Complexity measures an algorithm's efficiency with respect to internal factors, such as the time needed to run an algorithm and is a feature of the steps in the algorithms rather than the actual implementation of the algorithm.

A posteriori analysis, on the other hand, is a measure of the **performance** of an algorithm and evaluates efficiency empirically, comparing algorithms implemented on the same target platform to get their relative efficiency. Performance depends on the actual computer resources such as time, memory, disk, speed, compiler etc required to run a specific algorithm. Performance does not affect complexity but complexity can affect performace as the algorithm's design will feed into how the code is written and implemented. (This project involves measuring the actual performance of sorting algorithms and therefore depends on the specifics of the machine used.)

The speed of an algorithm to complete is one of the most important factors for choosing an algorithm. The speed will highly depend on the platform it is run on and therefore cannot really compare algorithms that are run on different machines with different capabilities or come to conclusions about the algorithm in general. While you could use a limited form of empirical comparison on your own machine, the results could not be applied generally. Instead the concept of complexity can be analysed mathematically. The actual time an algorithm takes to run doesn't tell the full story as it can be influenced by other factors such as the the available memory or the speed of the processor.

Complexity allows for algorithms to be compared by looking at their running time as a function of the input data size and in this way see which algorithms scale well to solve problems of a nontrivial size. Algorithmic complexity typically falls into one of a number of growth families (i.e. the growth in its execution time with respect to increasing input size n is of a certain order). Complexity looks at how the resource requirements grow as the input size increases, how the time required increases as the number of inputs increase. Memory or storage requirements of an algorithm could also be evaluated in this manner.

While there are other factors (including memory usage, storage usage, network usage, number of read-write operations) to be considered when evaluating the efficiency of an algorithm, the main focus in this project is the time efficiency - how the time taken varies with the size of the input. The algorithm should complete its task in an acceptable amount of time

The runtime of sorting algorithms can be measured by measuring the actual implementation of the algorithm using a timer function like pythons `timeit` module. The theoretical runtime complexity can be measured uses Big O notation which is a measure of the expected efficiency of an algorithm and measures the asymptotic behaviours of functions which means it measures how quickly a function grows or declines. The growth rate of a function is also called its *order*.
Big O represents the relationship between the size of the input n and the number of operations the algorithm takes and shows how quickly the the runtime grows as the input size increases. It is usually used to describe the complexity of an algorithm in the worst-case scenario. It could also be used to describe the execution time required or the memory space used by an algorithm. While for small input size n all algorithms are efficient, when the size becomes non-trivial then the order or growth of an algorithm will become more and more important.

If two algorithms have the same Big O notation, that does not mean they will execute in exactly the same times, but that the order of the number of operations that they will require to complete will be the same.

The details of the project will look at the runtime complexity of 5 different sorting algorithms.

According to wikipedia, *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. In-place algorithm updates input sequence only through replacement or swapping of elements. An algorithm which is not in-place is sometimes called not-in-place or out-of-place.*

The various sorting algorithms differ from each other in their memory requirements which depends on how the actual algorithm works. When a sorting algorithm is run, it has to first read the input data from a storage location to the computers RAM. An **in-place** sorting algorithm is one that only uses a fixed additional amount of working space, no matter what the input size whereas other sorting algorithms may require additional working memory which is often related to the input size. If there is limited memory available then in-place sorting is desirable. By producing the sorted output in the same memory space as the input data to be sorted avoids the need to use double the space. A sorting algorithm will still need some extra storage for working variables though.

Some algorithm create empty arrays to hold sorted copies and this requires more memory as the size of the input increases. (mention which ones). In-place sorting does not require additional arrays as the relative position of the elements are swapped within the array to be sorted and therefore additional memory should not be required.

A sorting algorithm is considered **comparison-based** if the only way to gain information about the total order is by comparing a pair of elements at a time using comparator operators to see which of the two elements should appear first in the sorted list. Comparison sorts do not make any assumptions about the data and compare all the elements against each other. Simple sorting algorithms such as Bubble Sort, Insertion Sort, and Selection Sort are comparison-based sorts. On the other hand, there are other sorting algorithms such as Counting Sort, Bucket Sort and Radix Sort which do make some assumptions about the data. These type type of algorithms consider the distribution of the data and the range of values that the data falls into and in doing so avoiding the need to compare all elements to each other.

Numbers and single characters can be quite easily sorted. While composite elements such as strings of characters are usually sorted by sorting each individual element of the string. Two elements can be compared to each other to see if they are less than, greater than or equal to each other. Sorting of custom objects may require a custom ordering scheme. In general a comparator function compares the elements p to q and returns 0 if $p = q$, a negative number if $p < q$, and a positive number if $p > q$. Using the example provided in the book, an airport terminal might list outbound flights in ascending order of destination city or departure time while flight numbers appear to be unordered.

According to Algorithms in a Nutshell, if a collection of comparable elements A is presented to be sorted in place where $A[i]$ and ai refer to the ith element in the collection with $A[0]$ being the first element in the collection, then to sort the collection the elements A must be reorganised so that if $A[i] < A[j]$, then $i < j$. Any duplicate elements must be contiguous in the resulting ordered collection. This means that if $A[i] = A[j]$ in a sorted collection, then there can be no k such that $i < k < j$ and $A[i]A[k]$. Finally, the sorted collection A must be a permutation of the elements that originally formed A.

The book also outlines how the elements in the collection being compared must admit a total ordering. That is, for any two elements p and q in a collection, exactly one of the following three predicates is true: `p = q`, `p < q`, or `p > q`.

Stability means that equivalent elements will retain their relative positioning after the sorting has taken place. With a stable sorting algorithm the order of equal elements is the same in the input and output. It is an important factor to consider when sorting key value pairs where the data might contain duplicate keys. This project only looks at sorting one dimensional arrays of integers but there are many other applications of sorting where the data has more than one dimension, in which case the data is sorted based on one column of data known as the key with the rest of the data is known as satellite data and should travel with the key when it is moved to another position. According to [Reference: Pollice G., Selkow S. and Heineman G. (2016). Algorithms in a Nutshell, 2nd Edition. O' Reilly.], if two elements (ai and aj) in the original unsorted data are equal (as determined by the comparator function), stable sorting refers to the fact such pairs of equal elements will retain their relative ordering after the sorting has taken place. If i < j then the final location of ai must be to the left of aj. Sorting algorithms that can guarantee this property are considered stable.

A sorting algorithm is considered stable if two objects with equal keys end up in the same order in the sorted output as they appear in the input. An unstable algorithm does not pay attention to the relationship between element locations in the original collection and does not guarantee the relative ordered will be kept after the sorting has taken place. See geeksforgeeks

3 Section 2. Sorting Algorithms

Introduce each of your chosen algorithms in turn, discuss their space and time complexity, and explain how each algorithm works using your own diagrams and different example input instances.

(by different example input instances, he means that each algorithm has its own average, best and worst case. The example inputs to be used in the report should highlight the behaviour of the algorithms under these different conditions.??)

3.1 2.1.1 Bubble Sort: A Simple Comparison based sort

The following sources are referred to in this section. - Computational Thinking with Algorithms lecture notes at GMIT. - runestone interactive python - wikipedia - programiz, - W3resources, - geekforgeeks, runestone interactive python

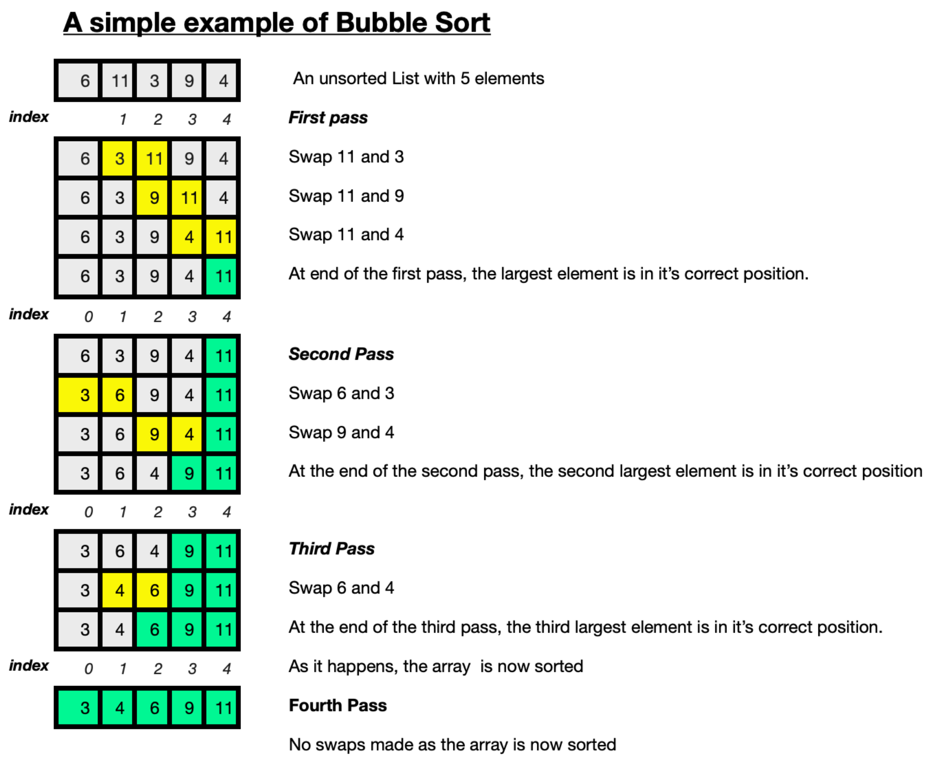
Bubble Sort is a fairly simple comparison-based sorting algorithm and is so named for the way larger values in a list "bubble up" to the end as sorting takes place. The algorithm repeatedly goes through the list to be sorted, comparing and swapping adjacent elements that are out of order. With every new pass through the data, the next largest element bubbles up towards it's correct position. Although it is quite simple to understand and to implement, it is slow and impractical for most problems apart from situations where the data is already nearly sorted.

Bubble Sort works by repeatedly comparing neighbouring elements and swapping them if they are out of order. It makes multiple passes or iterations through the list and with each pass through the list, the next largest element in it's proper place. It starts by comparing each element in the list (except the very last one) with it's neighbour to the right, swapping the elements which are out of order. At the end of the first pass, the last and largest element is now in it's final place. The second pass compares each element (except the last two) with the neighbour to the right, again swapping the elements which are out of order. At the end of the second pass through the data, the largest two elements are now in their final place. The algorithm continues by comparing and swapping the remaining elements in the list in the same way, except those now already sorted at the end of the list. With each iteration the sorted side on the right gets bigger and the unsorted side on the left gets smaller, until there are no more unsorted elements on the left.

3.2 Example

The above diagram illustrates how the Bubble Sorting algorithm works on a small array of 5 random numbers [6, 11, 3, 9, 4]. The example here shows how (4) $n-1$ passes are made through the array of n elements. there are $n-1$ comparisons performed on the first pass, $n-2$ on the second pass, $n-3$ on the third pass and $n-4$ on the fourth and last pass. The total number of comparisons is the sum of the first n-1 integers. For the first pass, the algorithm iterates through the array from left to right, the first pair of elements that are out of order is `(11,3)` so the order of this pair is swapped and the array becomes `[6, 3, 11, 9, 4]`. Then the elements `11` and `9` are compared and swapped resulting in `[6, 3, 9, 11 ,4]` then `11` and `4` are swapped. At the start of the second pass, the array is now `[6,3,9,11,4]`. The first pair of elements to be swapped is `(6,3)`

- In the first pass, swaps are made between three pairs of elements: `(11, 3)`, `(11, 9)` then `(11, 4)`.
- At the end of the first pass the array is now `[6, 3, 9, 4, 11]`.
- At the end of the first pass, the largest element `11` is now in its correct position.


A simple example of Bubble Sort
An unsorted List with 5 elements
index
First pass
Swap 11 and 3
Swap 11 and 9
Swap 11 and 4
At end of the first pass, the largest element is in it's correct position.
Second Pass
Swap 6 and 3
Swap 9 and 4
At the end of the second pass, the second largest element is in it's correct position
Third Pass
Swap 6 and 4
At the end of the third pass, the third largest element is in it's correct position.
As it happens, the array is now sorted
Fourth Pass
No swaps made as the array is now sorted

BubbleSort

- The first swap in the second pass is `(6, 3)`, then `(9, 4)` resulting in `[3, 6, 4, 9, 11]` at the end of the second pass.
- The second largest element `9` is now in it's correct place.
- On the third pass through the list, there is only one pair to be swapped `(6,4)`.
- At the end of the third pass, the array is sorted `[3, 4, 6, 9, 11]` with each element in its correct sorted order.
- Nevertheless the algorithm still does a fourth pass through the array, even though there are no more elements to be sorted.
- The algorithm is finished.

The python code to implement the Bubble Sort algorithm above is as follows. This code is widely available online and while there are some small differences, they are all largely the same. The code used in this project was adapted from code at runestone academy. There is also an optimised version of the Bubble Wort algorithm known as the `Short Bubble` which stops early if the algorithm finds that the list has become sorted already before all the loops have executed.

```python
def bubbleSort(array):
    # The outer loop goes through the elements n-1 times, if n is the number of elements in th

    for passnum in range(len(array)-1,0,-1):
        # count down to 0 as each time another element at the end of the list is sorted.
        # at each pass the last i elements are already in place so the inner loop is shorted b
        for i in range(passnum):
            # comparing each element i with the element right beside it (i+1)
            if array[i] > array[i+1] :
                # If the elements are out of order swap them so the largest element is right o
                array[i], array[i+1] = array[i+1], array[i]
```

A nested loop is used to compare each element and sort them into the correct place. The outer loop `for passnum in range(len(alist)-1,0,-1)` starts from the second last element in the list and gets shorter each time, taking account of the fact that the elements at the end of the list are becoming sorted with each iteration of the outside loop. The inner loop goes through the elements, comparing the element on the left with the element on the right

The `temp` variable can be replaced in the `Python` programming language by using simultaneous assignments `if array[i] > array[i+1] : array[i], array[i+1] = array[i+1], array[i]`

There are $n-1$ passes required where n is the number of elements in the array minus 1. The outer loop runs $n-1$ times. At the end of each iteration, another element will be in it's final sorted position. The inner loop goes through each element in the array up to the element(s) already sorted, each time comparing each element i with the element to the immediate right of it $i+1$. Using the > comparison operator, the elements are compared. If the element on the left (at index `i`) is greater in value than the element on it's right (at index `i`) then the elements are swapped.

There are $n-1$ passes through a list of n items. The total number of comparisons is the sum of the first $n-1$ integers which results in n^2 comparisons.

The best case occurs where the array is already sorted, as no exchanges are actually made but the comparisons still take place.

For the Bubble Sort algorithm, given an array that is almost or fully sorted it still requires $n-1$ passes through the input data. There is an optimised version of the Bubble Sort algorithm which can stop early.

The average case for the Bubble Sort algorithm is that exchnages are made half the time.

```
In [ ]:
```

3.2.1 Analysing Bubble Sort.

The Bubble Sort algorithm here has two `for` loops where it first performs $n-1$ comparisons, then $n-2$ comparisons and so on down to the final comparison.

Illustrating the worst-case scenario for the algorithm which occurs when the array to be sorted is in sorted reverse order:

- given an array of `[5, 4, 3, 2, 1]`
- In the first pass, 4 swaps are made, `(5,4)`,`(5,3)`,`(5,2)` and `(5,1)`
- In the second pass 3 swaps are made `(4,3)`, `(4,2)` and `(4,1)`
- In the third pass, the 2 swaps are made, `(3,2)` and `(3,1)`
- In the final pass, 1 swap is made, `(2,1)`.
- In this case where all the elements were in reverse order, it tooks 10 swaps to sort the 5 element array.

In the worst case the outer loop has to execute $n-1$ times and in the average case the inner loop executes about $\frac{n}{2}$ times for each outer loop. Inside the inner loop, the comparison and swap operations take constant time k.

So it total it performs $(n-1)+(n-2)+(n-3)...+2+1$ which is $fracn2+k = \frac{n^2}{k} \approx O(n^2)$. (removing the constants which don't change with input size simplifies it to n^2-n, the n is them removed as n^2 grows faster.

The worst case scenario for Bubble sort occurs when the data to be sorted is in reverse order.

The Bubble Sort algorithm here always runs in $O(n)$ times even if the array is sorted. The algorithm can be optimised to stop the algorithm if the inner loop didn't cause any swaps. If the optimised version of the Bubble Sort algorithm is applied on a nearly sorted array then the best case will be $O(n)$. This optimised version of the algorithm was not used here.

The average case is when the elements of the array are in random order.

The space complexity of Bubble Sort algorithm is $O(1)$. The only additional memory is needed for the temporary variable used for the swapping.

Bubble sort is an **in-place** sorting algorithm and it is **stable**. It is not a very practical algorithm to use is considered very inefficient sorting method as many exchanges are made before their final locations are known. One advantage of using the Bubble Sort over other sorting algorithms is that it is possible to determine that the list is already sorted if there are no exchanges made during the pass. The regular Bubble Sort algorithm needs to be modified to do this though. See runestone

3.3 Summary of time and space complexity of Bubble Sort:

- Best Case complexity: $O(n)$
- Average Case complexity: $O(n^2)$
- Worst Case complexity: $O(n^2)$
- Space complexity: $O(n)$

```
In [ ]:
```

3.4 2.1.2 Merge Sort: An efficient comparison based sort

- See merge sort, programiz

Merge Sort is an efficient, general-purpose, comparison-based sorting algorithm. It is a divide-and-conquer algorithm that was proposed by John von Neumann in 1945. It uses recursion to continually split the list in half. A sub-list of 0 or 1 items is considered sorted. Once the two halves are sorted a **merge** operation is performed which combines the two smaller sub-lists into a single sorted new sublist.

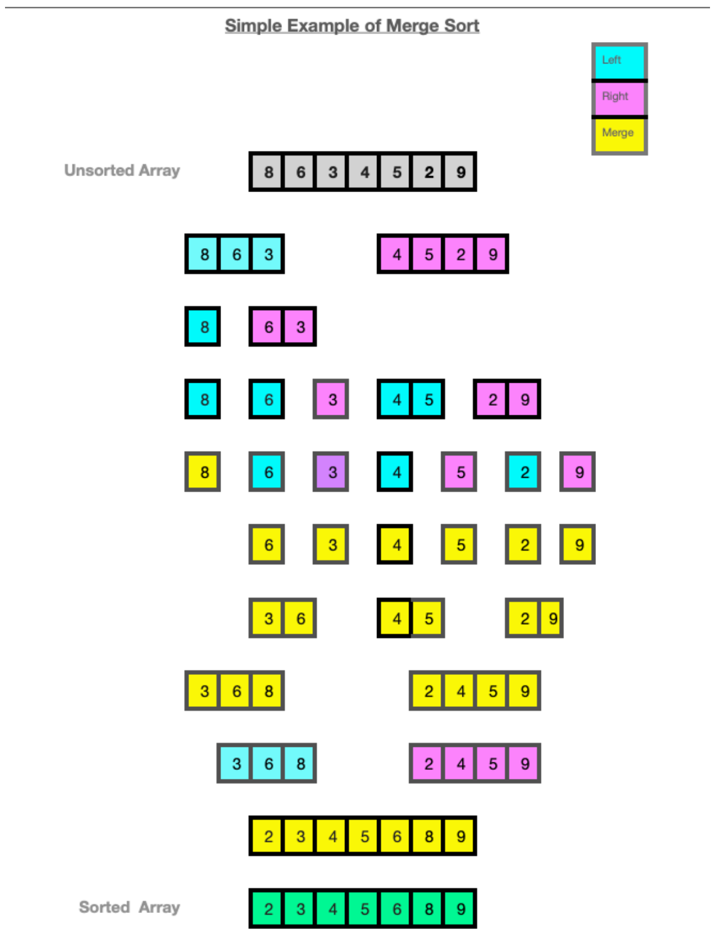
A divide-and-conquer algorithm recursively breaks a problem down into two or more sub-problems of the same or related type until these become simple enough to be solved directly. Then the solutions to the sub-problems are combined to produce a solution to the original problem.

The algorithm uses divide-and-conquer approach by breaking down the list into two evenly (as much as possible) sized halves, then repeatedly does this to each half until the sublist contains a single element or less. Each sub problem is then sorted recursively and the solutions to all the sub-lists are combined into a single sorted new list.

A list with one or less elements is considered sorted and is the base case for the recursion to stop. If the list has more than one item then it is split in half and the algorithm is recursively called on each half. When both halves are sorted, the smaller lists are then merged or combined into a single sorted list. All the smaller-sublists are repeatedly merged back into a new single sorted list. This algorithm will need extra memory to copy the elements when sorting. The extra space is needed to store the two halves when they are extracted using the slicing.

The diagram below demostrates how the MergeSort algorithm works on a small array of random integers `[8, 6, 3, 4, 5, 2, 9]`.

The array is split into a left array `[8, 6, 3]` and a right array `[4, 5, 2, 9]`. The left side of the array `[8, 6, 3]` is further split into `[8]` and `[6, 3]`. The `[8]` cannot be split any further so it



MergeSort

is then ready to be merged. `[6, 3]` is split into `[6]` and `[3]` which are then ready to be merged. `[3]` and `[6]` are then merged to become `[3,6]` and then they are merged with `[8]` resulting in a sub-array `[3,6,8]`.

The right array `[4, 5, 2, 9]` is further split into `[4,5]` and `[2,9]` which are further split resulting in 4 sub-arrays `[4]`,`[5]`,`[2]` and `[9]`. The sub-arrays cannot be split any further and the merging begins. `[4]` and `[5]` are merged into `[4,5]`. `[2]` and `[9]` are merged into `[2,9]`. The two sorted sub-arrays `[4,5]` and `[2,9]` are then merged to become `[2, 4, 5, 9]`. The sorted sub-array `[3,6,8]` is then merged with the sorted right array `[2, 4, 5, 9]`. The result is a single sorted array of `[2, 3, 4, 5, 6, 8, 9]`.

3.5 The python code for Merge Sort

The following is the python code for the Merge Sort algorithm. This code is widely available online. The code used in this project is based on the code at runestone academy. I have added comments to the code in the accompanying python script which is used in this benchmarking project. See `merge.py`.

```
def merge_sort(array):
    # the base case is a list with 0 or 1 elements which is is already sorted.
```

```
if len(array)>1:
    # find the middle of the list using integer division to find the split point
    mid = len(array)//2
    # divide the elements into two halves using the mid point
    # # The elements are copied into the temporary arrays left[] and right[]
    # left contains the elements from the first half of the list (up to the mid)
    left = array[:mid]
    # right contains the elements from the second half of the list, (from mid to the end)
    right = array[mid:]
    # recursively call the function on the first (left) half
    merge_sort(left)
    # recursively call the function on the second (right) half
    merge_sort(right)

    # once the function has been called on the left and right half, each half should be so
    # The following code does the merge part, merging the two smaller lists into a single
    # i, j and k represents the index of the left array, right array and merged arrays res
    i ,j, k = 0,0,0
    # The elements are placed back into the original list (array) by repeatedly taking the

    # until the left and right arrays are empty.
    while i < len(left) and j < len(right):
        # compare the first/next element in left and right arrays, if the left element is
        if left[i] <= right[j]:
            array[k]=left[i]
            # increment the index of left (for the next comparison between left and right
            i += 1
        else:
            # otherwise if the smallest element is in the right array, assign this element
            array[k]=right[j]
            # increment the index of the right array (for the next comparison between left
            j += 1
        # after assigning another element to the merged array, increment the index by 1
        k=k+1
    # no elements left in right array so check if any element left in the left array, if s
    while i < len(left):
        array[k]=left[i]
        i += 1
        k += 1
    # no elements left in left array, so check if any elements left in the right array, if
    while j < len(right):
        array[k]=right[j]
        j += 1
        k += 1
return array
```

The Merge Sort algorithm uses a recursive divide-and-conquer approach which results in a worst-case running time of $O(n \log n)$. The algorithm consists of two distinct processes, the splitting and the merging. - A list can be split in half $\log n$ times where n is the number of elements in the list. - Each item in the list will eventually be processed and placed in the sorted list. This means a list of size n will require n operations. Therefore there are $\log n$ splits, each of which costs n for a total of $n \log n$ operations - Merge Sort needs extra space to hold the left and right halves which can be a critical factor for large datasets. See interactive python

Merge-Sort gives good all around performances with similar best, worst and average cases with a linearitmic $n \log n)$ time complexity in each case. This makes it a good choice if predictable run-time is important. There are versions of Merge Sort which are particularly good for sorting data with slow access times such as data that cannot be held in RAM or are stored in linked lists. Merge Sort is a stable sorting algorithm.

3.6 Summary of time and space complexity of Merge Sort

- Best Case complexity: $O(n \log n)$
- Average Case complexity: $O(n \log n)$
- Worst Case complexity: $O(n \log n)$
- Space complexity: $O(n)$

```
In [ ]:
```