Benchmarking Sorting Algorithms

Module:	Computational Thinking with Algorithms
Module Code :	46887
Programme:	H.Dip in Data Analytics
Student Name:	Freha Saleem
Student Id:	G00376465
Lecturer:	Dr. Dominic Carr
Date:	03/05/2020

Table of Contents

Benchmarking Sorting Algorithm

Introduction:

The objective of this project is to research, implement and benchmark five sorting algorithms.

Benchmarking:

Benchmarking is used to compare time complexity and performance of algorithms. It is a posteriori analysis, an empirical method to compare the relative performance of algorithm implementations.

Benchmarking result data can be used to validate theoretical or a priori analysis of algorithms. When an algorithm is implemented the performance of that algorithm depends upon various hardware and software factors such as system architecture, CPU design, choice of Operating System, background processes, programming language etc.

The aim of benchmarking is to assess performance of algorithms. Therefore, it is important to conduct multiple statistical runs using the same experimental setup, to ensure that benchmarks are representative of the performance expected by an "average" user

Sorting:

Sorting is the process of placing elements from a collection in some kind of order.

Sorting algorithm are one of the most thoroughly studied algorithms in computer science. There are many different sorting implementations and applications that can be used in the code.

Sorting is particularly helpful in the context of computer science for following reasons:

- From a strictly human-friendly perspective, it makes a single dataset lot easier to read.
- It is easier to implement search algorithms in order to find or retrieve an item from a sorted dataset (Searching for an item on a list works much faster if the list is sorted).
- Selecting items from a list based on their relationship to the rest of the items is easier with sorted data. For example, finding the kth-largest or smallest value, or finding the median value of the list, is much easier when the values are in ascending or descending order.

Analysing the frequency distribution of items on a list is very fast if the list is sorted. From commercial applications to academic research and everywhere in between, there are countless ways you can use sorting to save time and effort.

Sorting is the organizing of a set of a similar item in a collection by some property. There are important things to note here:

1. Sorted lists are permutations(reordering)of their original lists.
2. We can sort a collection of items in either increasing or decreasing order by any one property, and that property by can really be anything. By size, lexicographical (alphabetical) order, numerical order, date, time — etc!

3. We can only sort a dataset where the items are homogeneous, or are of the same type. In other words, we couldn't sort a dataset with both words and numbers, because that dataset doesn't have a shared property that we could actually sort by.

There are many sorting algorithms that can be used to sort items and these algorithms have their own pros and cons. An algorithm can be thought of a procedure or formula to solve a particular problem. There are different ways to classify, which algorithm to use to solve a specific problem when there exist multiple solutions to the problem.

When comparing various sorting algorithms, there are several things to consider.

- Time Complexity
- Space Complexity (or memory usage),
- Stability
- Internal or External
- Recursive or Non-Recursive
- Comparison Sort or Non-Comparison Sort.

These are important factors for programmers when they are writing a sorting algorithm or choosing which one to implement.

There are different sorting algorithms each have their own benefits and drawbacks. These classifications help to decide what those benefits and drawbacks are for any given sorting algorithms — and when that algorithm might be helpful (or maybe not so helpful!).

Let's take a look at each of these classifications in more detail:

1. Time complexity

The easiest way to classify an algorithm is by time complexity, or by how much relative time it takes to run the algorithm given a different input size.

Time complexity of an algorithm signifies the total time required by the program to run till its completion. The time complexity of algorithms is commonly expressed using the big O notation. It is an asymptotic notation to represent the time complexity, that estimated by counting the number of elementary steps performed by any algorithm to finish execution.

2. Space complexity/memory usage

Different algorithms require different amount of space, depending on their space complexity. Another way of thinking about space complexity in the context of sorting algorithms is by answering the question: How much memory will this algorithm need to run?

There are two types of classifications for the space complexity of an algorithm:

- **In-place**
- **Out-of-place.**

An in-place algorithm is one that operates directly on the inputted data. There is a possibility with this is that the data is getting completely transformed in the process of transforming it,

which means that the original dataset is effectively being destroyed. But it is more space-efficient, because the algorithm only needs a small bit of extra space in memory.

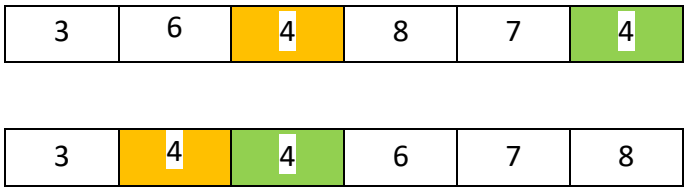
In-place algorithms can be helpful if there is not enough memory to spare.

Out-of-place algorithms don't operate directly on the original dataset; instead, make a new copy, and perform the sorting on the copied data. This can be safer, but the drawback is that the algorithm's memory usage grows with input size.

3. Stability

A sorting technique is stable if it does not change the order of elements with the same value, a stable algorithm is one that preserves the relative order of the elements; if the keys are the same, we can guarantee that the elements will be ordered in the same way in the list as they appeared before they were sorted. On the other hand, an unstable algorithm is one where there is no guarantee that, if two items are found to have the same sort key, that their relative order will be preserved.

Stable sort(preserve the order)



4. Internal vs. External

The way that an algorithm stores data while its sorting through records is yet another way that we can classify sorting algorithms. If all the data that needs to be sorted can be kept in main memory, the algorithm is an internal sorting algorithm.

if the data have to be stored outside of main memory, in other word, stored in external memory, in either a disk or a tape — the algorithm is referred to as an external sorting algorithm.

5. Recursive or non-recursive

Some algorithms do their sorting by dividing and conquering: that is to say, the split up a large dataset into smaller inputs, and then recursively call a sort function upon all of those smaller inputs. This is referred to as a recursive sorting algorithm.

In contrast, a non-recursive sorting algorithm is one that doesn't implement recursion!

Most algorithms don't have to be implemented recursively; they can be written and implemented iteratively. But a sorting algorithm is recursive or not is an easy way of classifying one set of algorithms from another.

6. Comparison Sort

it's possible to classify a sorting algorithm based on how it actually does the job of sorting elements.

Any sorting algorithm that compares two items — or a pair — at a time in the process of sorting through a larger dataset is referred to as a comparison sort. These algorithms uses some type of comparator (for example: >= or <=) to determine which of any two elements should be sorted first.

Sorting algorithms that do not use any type of comparators to do their sorting are referred to as non-comparison sorts.

comparator functions:

The elements in the list being compared must have a total ordering. That is, for any two elements p and q in list , exactly one of the following three predicates is true: p = q, p < q, or p > q. comparator function are to compare elements

Algorithm Analysis with Big-O Notation

Algorithm analysis refers to the analysis of the complexity of different algorithms and finding the most efficient algorithm to solve the problem at hand. Big-O Notation is a statistical measure, used to describe the complexity of the algorithm.

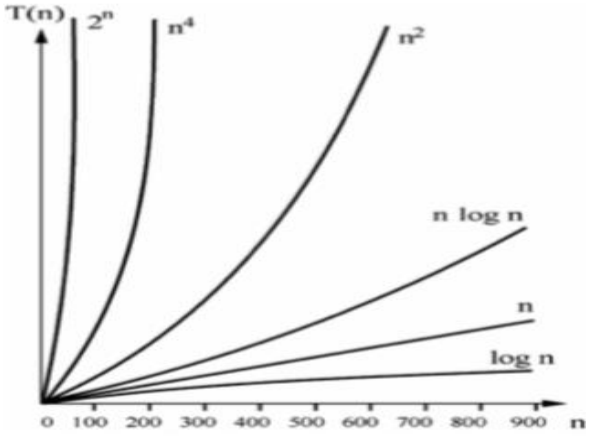
Big-O notation is a metrics used to find algorithm complexity. The time to run different algorithms can be influenced by several unrelated factors, including processor speed or available memory. Big O, on the other hand, provides a platform to express runtime complexity in hardware-agnostic terms. With Big O, you express complexity in terms of how quickly your algorithm's runtime grows relative to the size of the input, especially as the input grows arbitrarily large.

Assuming that *n* is the size of the input to an algorithm, the Big O notation represents the relationship between *n* and the number of steps the algorithm takes to find a solution. Big O uses a capital letter "O" followed by this relationship inside parentheses.

Big O	Complexity	Description
O(1)	constant	The runtime is constant regardless of the size of the input.
O(n)	linear	The runtime grows linearly with the size of the input.
$O(n^2)$	quadratic	The runtime is a quadratic function of the size of the input.
$O(2^n)$	exponential	The runtime grows exponentially with the size of the input.
O(log n)	logarithmic	The runtime grows linearly while the size of the input grows exponentially

For example, if there is a linear relationship between the input and the step taken by the algorithm to complete its execution, the Big-O notation used will be O(n). Similarly, the Big-O

notation for quadratic functions is $O(n^2)$. The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.



T(n) is running time n is the size of input

Before getting into specific algorithms, we should think about the operations that can be used to analyse a sorting process.

First, it will be necessary to compare two values to see which is smaller (or larger). In order to sort a collection, it will be necessary to have some systematic way to compare values to see if they are out of order. The total number of comparisons will be the most common way to measure a sort procedure.

Second, when values are not in the correct position with respect to one another, it may be necessary to exchange them. This exchange is a costly operation and the total number of exchanges will also be important for evaluating the overall efficiency of the algorithm.

Sorting Algorithms:

Following is the detail of the algorithms I am going to implement and benchmark.

- Selection Sort, Insertion Sort (simple comparison-based sort)
- Merge Sort, Quicksort(efficient comparison-based sort)
- Radix Sort (A non-comparison sort)

1. Selection Sort:

A selection sort looks for the largest/smallest value as it makes a pass and, after completing the pass, places it in the proper location. After the second pass, the next largest/smallest is in place. This process continues and requires n−1 passes to sort *n* items, since the final item must be in place after the (n−1) st pass.

- It is comparison-based sorting algorithm.
- This sorting algorithm selects the next value based on certain criteria.
- The selection sort only makes one exchange for every pass through the list.to do this, a selection sort looks for the smallest value as it makes a pass and, after completing the pass, places it in the proper location.
- After the first pass, the smallest item is in the correct place. After the second pass, the next smallest is in place. This process continues and requires n−1 passes to sort n items, since the final item must be in place after the (n−1)st pass.

  For example, if we are given numbers in an array it will choose the smallest number in all the array to place it in the 1st place by swapping original number at this place by the new number and then select the smallest amongst remaining numbers and place it in 2nd place by swapping original number at this place with the newer smaller number and so on until we are left with the smallest number that will automatically be placed at the start the array.

Pseudo Code:

Let A be the array of N numbers which needs to be sorted

- For Index *i* from 0 to N-1
  - Let Num be the number at *i th* place
  - For Index J from *i* to N-1
    - Find A number Num1 which is smallest number within range of J (*i* to N-1) at jth place in the array
- Swap Num at Ith place with Num2 at jth place

Demonstration:

1. Find the smallest number. Swap it with the first place number.
2. Find the second-smallest number. Swap it with the second number.
3. Find the third-smallest number. Swap it with the third number.
4. Repeat finding the next-smallest number and swapping it into the correct position until the array is sorted.

Selection Sort						Comparison	
8	5	7	1	9	3	n-1	First smallest
1	5	7	8	9	3	n-2	Second smallest
1	3	7	8	9	5	n-3	
1	3	5	8	9	7	2	
1	3	5	7	9	8	1	
1	3	5	7	8	9	0	
						Total comparisons= n(n-1)/2	

Sorted list
Current
Exchange

Selection Sort Classification	
Time Complexity	$O(n^2)$
Space Complexity	In-place
Stability	No
Internal/External	Internal
Recursive/non- Recursive	non-Recursive
Class	Comparison

Big O Runtime Complexity

Selection sort algorithm has nested loops that go over the list. The inner loop goes through the sub-list until it finds the position of the smallest element. This algorithm still has an ***O(n^2)*** runtime complexity on the best, average and worst case.

Strengths and Weaknesses of Selection Sort

The selection sort algorithm is very easy to implement. Even though selection sort is an *O(n^2)* algorithm, it's also much more efficient in practice than other quadratic implementations such as bubble sort.

Selection sort is very efficient when working on small lists.

2. Insertion Sort:

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. When you got a new card we look through the existing sorted cards in our hand and put the new card in its position.

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially, and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$, where n is the number of items to be sorted.

Pseudo Code:

Let A be the array of N numbers which needs to be sorted

Step 1 – assume first element, is already sorted and in sorted sub-list

Step 2 – Pick next element

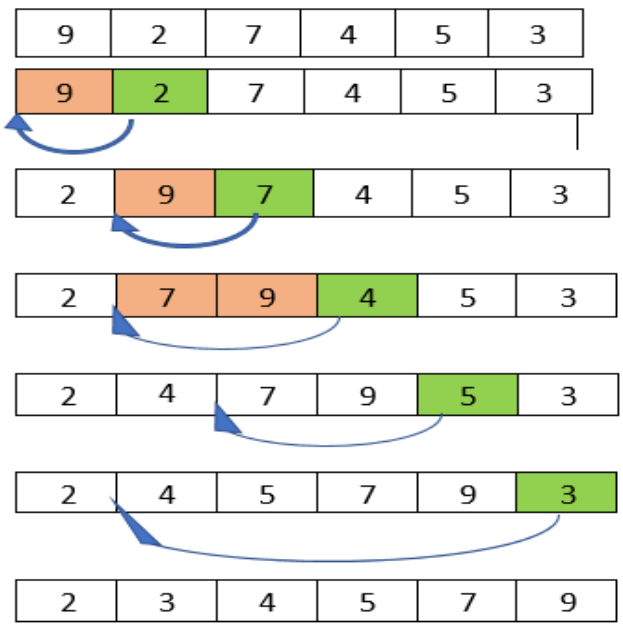
Step 3 – Compare with all elements in the sorted sub-list

Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 – Insert the value

Step 6 – Repeat until list is sorted

Demonstration:



summary of the steps of the algorithm when sorting the array:

1. The algorithm starts with key = 2 and goes through the subarray to its left to find the correct position for it. In this case, the subarray is [9].
2. Since 2 < 9, the algorithm shifts element 9 one position to its right. The resultant array at this point is [9, 9, 7, 4, 5,3].
3. Since there are no more elements in the subarray, the key is now placed in

its new position, and the final array is [2, 9, 7, 4, 5,3].

4. The second pass starts with key = 7 and goes through the subarray located to its left, in this case [2, 9].
5. Since 7< 9, shifts 9 to its right. The resultant array at this point is [2, 9, 9, 4, 5,3].
6. Since 7> 2, the algorithm doesn't need to keep going through the subarray, so it positions key and finishes the second pass. At this time, the resultant array is [2, 7 ,9, 4, 5,3].
7. The third pass through the list puts the element 4 in its correct position, and the fourth pass places element 5 in the correct spot, and the fifth pass places element 3 in the correct spot leaving the array sorted.

Big O Runtime Complexity

Insertion sort algorithm has a couple of nested loops that go over the list. The inner loop is pretty efficient because it only goes through the list until it finds the correct position of an element. That said, the algorithm still has an ***O(n^2)*** runtime complexity on the average case.

The worst case happens when the supplied array is sorted in reverse order. In this case, the inner loop has to execute every comparison to put every element in its correct position. This still gives *O(n^2)* runtime complexity.

The best case happens when the supplied array is already sorted. Here, the inner loop is never executed, resulting in an *O(n)* runtime complexity.

Strengths and Weaknesses of Insertion Sort

The insertion sort algorithm is very easy to implement. Even though insertion sort is an *O(n^2)* algorithm, it's also much more efficient in practice than other quadratic implementations such as bubble sort.

There are more powerful algorithms, including merge sort and quicksort, but these implementations are recursive and usually fail to beat insertion sort when working on small lists. Some quicksort implementations even use insertion sort internally if the list is small enough to provide a faster overall implementation.

Insertion Sort Classification	
Time Complexity	O(n^2)
Space Complexity	In-place
Stability	stable
Internal/External	Internal
Recursive/non- Recursive	non-Recursive
Comparison sort	Yes

3. **Merge Sort :**

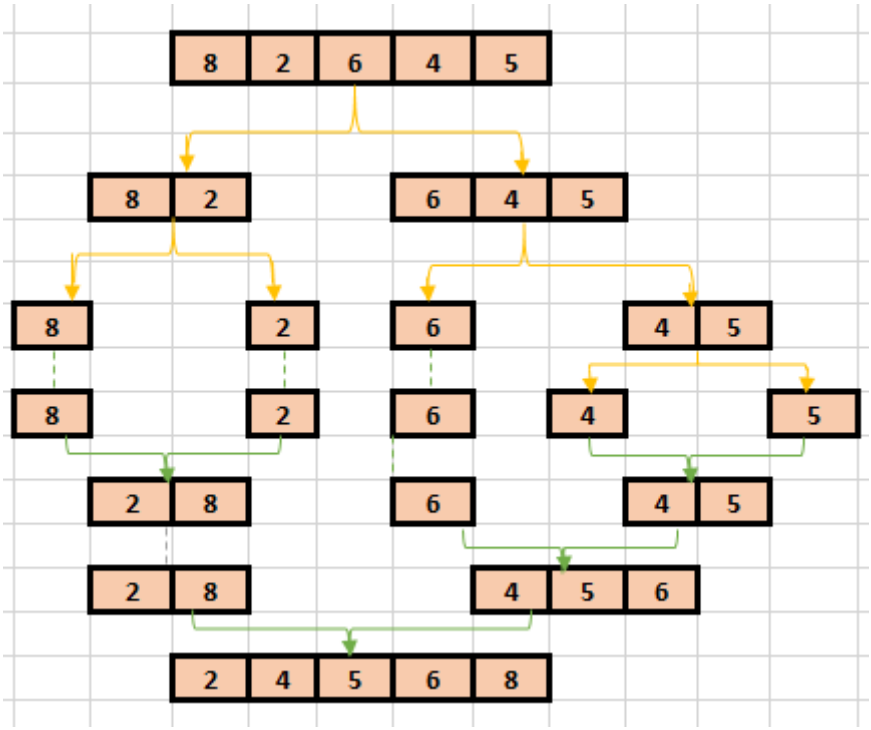
Merge sort is a very efficient sorting algorithm. It's based on the divide-and-conquer approach, a powerful algorithmic technique used to solve complex problems.

To properly understand divide and conquer, it is important to understand the concept of recursion. Recursion involves breaking a problem down into smaller subproblems until they're small enough to manage. In programming, recursion is usually expressed by a function calling itself.

Divide-and-conquer algorithms typically follow the same structure:

1. The original input is broken into several parts, each one representing a subproblem that's similar to the original but simpler.
2. Each subproblem is solved recursively.
3. The solutions to all the subproblems are combined into a single overall solution.

In the case of merge sort, the divide-and-conquer approach divides the set of input values into two equal-sized parts, sorts each half recursively, and finally merges these two sorted parts into a single sorted list.



The Merge Sort Process

The figure uses yellow arrows to represent halving the array at each recursion level. The green arrows represent merging each subarray back together. The steps can be summarized as follows:

1. with [8, 2, 6, 4, 5] defines midpoint as 2. The midpointis used to halve the input array into array[:2] and array[2:], producing [8, 2] and [6, 4, 5], respectively. merge_sort() is then recursively called for each half to sort them separately.

2. The call to merge_sort() with [8, 2] produces [8] and [2]. The process repeats for each of these halves.
3. The call to merge_sort() with [8] returns [8] since that's the only element. The same happens with the call to merge_sort() with [2].
4. At this point, the function starts merging the subarrays back together using merge(), starting with [8] and [2] as input arrays, producing [2, 8] as the result.
5. On the other side, [6, 4, 5] is recursively broken down and merged using the same procedure, producing [4, 5, 6] as the result.
6. In the final step, [2, 8] and [4, 5, 6] are merged back together with merge(), producing the final result: [2, 4, 5, 6, 8].

Measuring Merge Sort's Big O Complexity

To analyse the complexity of merge sort, we look at its two steps separately:

merge() has a linear runtime. It receives two arrays whose combined length is at most n(the length of the original input array), and it combines both arrays by looking at each element at most once. This leads to a runtime complexity of O(n).

The second step splits the input array recursively and calls merge() for each half. Since the array is halved until a single element remains, the total number of halving operations performed by this function is log(n). Since merge() is called for each half, we get a total runtime of O(n log(n)). O(n $\log_{(}$n)) is the best possible worst-case runtime that can be achieved by a sorting algorithm.

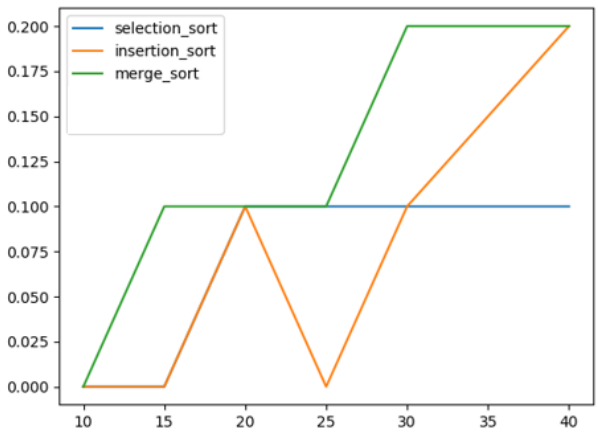
Strengths and Weaknesses of Merge Sort

Merge sort is a very efficient algorithm that scales well as the size of the input array grows. It's also straightforward to parallelize because it breaks the input array into chunks that can be distributed and processed in parallel if necessary.

That said, for small lists, the time cost of the recursion allows algorithms such as selection sort and insertion sort to be faster. For example, running an experiment with a list of 10-40 elements results in the following times (time is in sec):

	10	15	20	25	30	40
selection_sort	0	0	0.1	0.1	0.1	0.1
insertion_sort	0	0	0.1	0	0.1	0.2
merge_sort	0	0.1	0.1	0.1	0.2	0.2

The average time taken three algorithms to sort list of size 10,15,20,25,30,40



Both selection sort and insertion sort beat merge sort when sorting a small size element list.

Another drawback of merge sort is that it creates copies of the array when calling itself recursively. It also creates a new list inside merge() to sort and return both input halves. This makes merge sort use much more memory than selection sort and insertion sort, which are both able to sort the list in place.

Due to this limitation, you may not want to use merge sort to sort large lists in memory-constrained hardware.

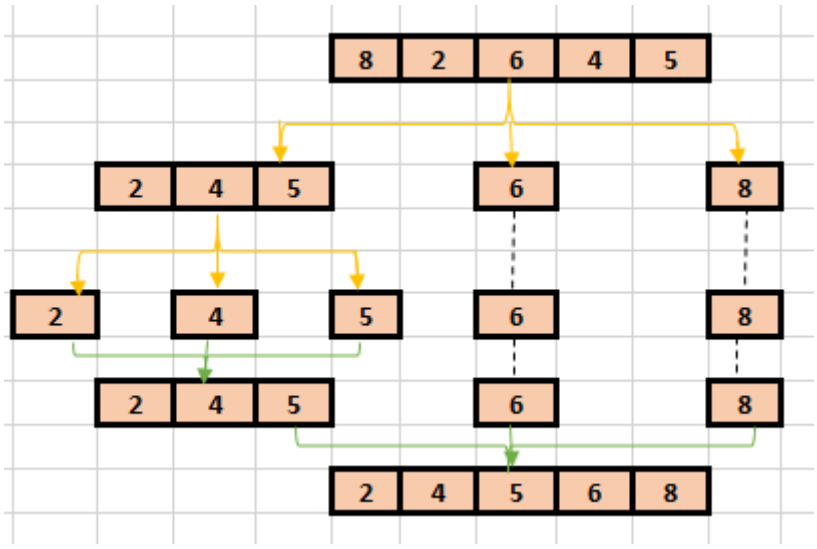
Merge Sort Classification	
Time Complexity	O(nlogn)
Space Complexity	Out of place
Stability	stable
Internal/External	external
Recursive/non- Recursive	Recursive
Class	Comparison

4. Quicksort

The quicksort algorithm applies the divide-and-conquer principle to divide the input array into two lists, the first with small items and the second with large items. The algorithm then sorts both lists recursively until the resultant list is completely sorted.

Dividing the input list is referred to as partitioning the list. Quicksort first selects a pivot element and partitions the list around the pivot, putting every smaller element into a low array and every larger element into a high array.

Putting every element from the low list to the left of the pivot and every element from the high list to the right positions the pivot precisely where it needs to be in the final sorted list. This means that the function can now recursively apply the same procedure to low and then high until the entire list is sorted.



The Quicksort Process

1. The yellow lines represent the partitioning of the array into three lists: `low`, `same`, and `high`. The green lines represent sorting and putting these lists back together. Here's a brief explanation of the steps:
2. The `pivot` element is selected randomly. In this case, `pivot` is 6.
3. The first pass partitions the input array so that `low` contains `[2, 4, 5]`, `same` contains `[6]`, and `high` contains `[8]`.
4. `quicksort()` is then called recursively with `low` as its input. This selects a random `pivot` and breaks the array into `[2]` as `low`, `[4]` as `same`, and `[5]` as `high`.
5. The process continues, but at this point, both `low` and `high` have fewer than two items each. This ends the recursion, and the function puts the array back together. Adding the sorted `low` and `high` to either side of the `same` list produces `[2, 4, 5]`.
6. On the other side, the `high` list containing `[8]` has fewer than two elements, so the algorithm returns the sorted `low` array, which is now `[2, 4, 5]`. Merging it with `same`(`[6]`) and `high` (`[8]`) produces the final sorted list.

Selecting the pivot Element

Selection of pivot is very important step in quicksort algorithm, because of how the quicksort algorithm works, the number of recursion levels depends on where pivot ends up in each partition. In the best-case scenario, the algorithm consistently picks the **median** element as the pivot. That would make each generated subproblem exactly half the size of the previous problem, leading to at most *log(n)* levels.

But if the algorithm consistently picks either the smallest or largest element of the array as the pivot, then the generated partitions will be as unequal as possible, leading to n-1 recursion levels. That would be the worst-case scenario for quicksort.

Quicksort's efficiency often depends on the pivot selection. If the input array is unsorted, then using the first or last element as the pivot will work the same as a random element. But if the input array is sorted or almost sorted, using the first or last element as the pivot could lead to a worst-case scenario. Selecting the pivot at random makes it more likely quicksort will select a value closer to the median and finish faster.

Another option for selecting the pivot is to find the median value of the array and force the algorithm to use it as the pivot. This can be done in *O(n)* time. Although the process is little bit more involved, using the median value as the pivot for quicksort guarantees you will have the best-case Big O scenario.

Quicksort's Big O Complexity

With quicksort, the input list is partitioned in linear time, *O(n)*, and this process repeats recursively an average of *logn* times. This leads to a final complexity of ***O(n logn)***.

As quick sort divides the list into halves every time, but we repeat the iteration n times(where n is the size of list). Hence time complexity is n*log(n). The running time consists of n loops

(iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

The *O(n)* best-case scenario happens when the selected pivot is close to the median of the array, and an $O(n^2)$ scenario happens when the pivot is the smallest or largest value of the array.

Randomly selecting the pivot makes the worst case very unlikely. Random pivot selection good enough for most implementations of the algorithm.

Strengths and Weaknesses of Quicksort

Quicksort is very fast. Although its worst-case scenario is theoretically O(n2), in practice, a good implementation of quicksort beats most other sorting implementations. Also, just like merge sort, quicksort is straightforward to parallelize.

One of quicksort's main disadvantages is the lack of a guarantee that it will achieve the average runtime complexity. Although worst-case scenarios are rare, certain applications can't afford to risk poor performance, so they opt for algorithms that stay within O(n log2n)regardless of the input.

Just like merge sort, quicksort also trades off memory space for speed. This may become a limitation for sorting larger lists.

Quick Sort Classification	
Time Complexity	O(nlogn) (average case)
Space Complexity	In-place
Stability	unstable
Internal/External	internal
Recursive/non- Recursive	Recursive
Class	Comparison

5. Radix Sort:

Radix sort is non comparative sorting method. it is an integer sorting algorithm, that sorts by grouping numbers by their individual digits (or by their radix). It uses each radix/digit as a key, and implements counting sort or bucket sort under the hood in order to do the work of sorting.

The term radix comes directly from Latin; in Latin, radix means root, which makes sense when you think about the root of a number and a number system being derived directly from its radix, and the total number of possible digits for a number's place values.

Radix sort handles sorting by implementing counting sort (or bucket sort) on one digit at a time. And it does this all in a very specific order,

There are Two classifications of radix sorts

- Least significant digit (LSD) radix sorts
- Most significant digit (MSD) radix sorts.

LSD radix sorts process the integer representations starting from the least digit and move towards the most significant digit. it uses counting or bucket sort internally and it is usually solved iteratively, and not recursively.

MSD radix sort. it starts with the largest number first, and then moves on to continue sorting until it reaches the smallest digit. This method uses either counting sort or bucket sort under the hood, and is usually solved by recursion

Input		1st pass		2nd pass		3rd pass
10		10		5		5
52		52		209		10
5		44		10		19
209		5		19		44
19		209		44		52
44		19		52		209

```
Pass 0
[10, 52, 44, 5, 209, 19]
Pass 1
[5, 209, 10, 19, 44, 52]
Pass 2
[5, 10, 19, 44, 52, 209]
Sorted: [5, 10, 19, 44, 52, 209]
```

The Radix Sort Algorithm

Do following for each digit i where i varies from least significant digit to the most significant digit.

a) Sort input array using counting sort (or any stable sort) according to the i'th digit. Example:

- Original, unsorted list: 10, 52, 05 ,209, 19, 44
- Sorting by least significant digit (1s place) gives: 10, 52, 44, 5, 209, 19
- Sorting by next digit (10s place) gives: 05,209, 10, 19, 44, 52
- Sorting by most significant digit (100s place) gives: 5,10,19,44,52,209

Radix sort's Big O Complexity

Radix sort complexity O(kn), where k is the length(number of digits) of the largest number and n is the size of the input array.

Strengths and Weaknesses of Radix sort

radix sort, like counting sort and bucket sort, is an integer-based algorithm (i.e. the values of the input array are assumed to be integers). Hence radix sort is among the fastest sorting algorithms around, in theory. But it is not performed good at small size lists.

the performance of the radix sort is heavily influenced by variations in the digits count or component size of the items. Radix sort uses a lot of space in creating new arrays or objects for grouping items.

Radix Sort Classification	
Time Complexity	O(kn)
Space Complexity	Out of place
Stability	stable
Internal/External	internal
Recursive/non- Recursive	Non-Recursive(recursive for MSD)
Class	Non-Comparison

SORTING ALGORITHM	TIME COMPLEXITY			SPACE COMPLEXITY
	BEST CASE	AVERAGE CASE	WORST CASE	WORST CASE
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	O(1)
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	O(1)
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	O(n)
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	O(n log n)
Radix Sort	$\Omega(n k)$	$\Theta(n k)$	$O(n k)$	O(n + k)

The time and space complexity of the algorithms that I am going to benchmark.

Implementation

As mentioned above benchmarking is the methodology of comparing algorithms with respect to a certain performance measure. The benchmarking process abstractly consists of three levels:

1. Setup
2. Execution
3. Analysis.

(1) The Setup defines the design of a benchmark experiment: data set, candidate algorithms and performance measure.
Radom integer list of size (Size 100 250 500 750 1000 1250 2500 3750 5000 6250 7500 8750 10000) used, and selection sort, insertion sort, merge sort, quick sort and radix sort algorithms are the algorithms to compare and runtime is used for performance measure.

Implementation of Selection

```
def selection_sort(alist):
    """
    #print(alist)
    # Traverse through all array elements
    for i in range(len(alist)):
        # assume that the first item of the unsorted segment is the smallest
        lowest_value_index = i
        # This loop iterates over the unsorted items
        for j in range(i + 1, len(alist)):
            if alist[j] < alist[lowest_value_index]:
                lowest_value_index = j
        # Swap values of the lowest unsorted element with the first unsorted
        # element
        alist[i], alist[lowest_value_index] = alist[lowest_value_index], alist[i]

    return alist
```

Implementation of Insertion Sort:

```
def insertion_sort(arr):
    for i in range(1, len(arr)): # Traverse through 1 to len(arr)

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j] #shifting the element
                j -= 1
        arr[j+1] = key #When you finish shifting the elements, you can put key  in its co
    return arr
```

Implementation of Merge Sort:

```
def merge_sort(alist):

    # If the list is a single element, return it
    if len(alist) <= 1:
        return alist
    # Use floor division to get midpoint, indices must be integers
    mid = len(alist) // 2
    # Sort and merge each half
    left_list = merge_sort(alist[:mid])
    right_list = merge_sort(alist[mid:])
    # Merge the sorted lists into a new one
    return merge(left_list, right_list)

def merge(left, right):
    # If the first array is empty, then nothing needs
    # to be merged, and you can return the second array as the resu
    if len(left) == 0:
        return right
    # If the second array is empty, then nothing needs
    # to be merged, and you can return the first array as the resul
    if len(right) == 0:
        return left
    result = []
    index_left = index_right = 0
    # Now go through both arrays until all the elements
    # make it into the resultant array
    while len(result) < len(left) + len(right):
        # The elements need to be sorted to add them to the
        # resultant array, so you need to decide whether to get
        # the next element from the first or the second array
        if left[index_left] <= right[index_right]:
            result.append(left[index_left])
            index_left += 1
        else:
            result.append(right[index_right])
            index_right += 1
        # If you reach the end of either array, then you can
        # add the remaining elements from the other array to
        # the result and break the loop
        if index_right == len(right):
            result += left[index_left:]
            break
        if index_left == len(left):
            result += right[index_right:]
            break
    return result
```

The implementation of the merge sort algorithm have two different pieces:

1. merge_sort(alist) function that recursively splits the input in half
2. merge(left, right) function that merges both halves, producing a sorted array

merge(left,right) receives two different sorted arrays that need to be merged together.

Implementation of Quick Sort:

```
def quick_sort(alist):

    # If the input array contains fewer than two elements,
    # then return it as the result of the function
    if len(alist) < 2:
        return alist

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

    # Select a `pivot` element randomly
    pivot = alist[randint(0, len(alist) - 1)]

    for item in alist:
        # Elements that are smaller than the `pivot` go to
        # the `low` list. Elements that are larger than
        # `pivot` go to the `high` list. Elements that are
        # equal to `pivot` go to the `same` list.
        if item < pivot:
            low.append(item)
        elif item == pivot:
            same.append(item)
        elif item > pivot:
            high.append(item)

    # The final result combines the sorted `low` list
    # with the `same` list and the sorted `high` list
    return quick_sort(low) + same + quick_sort(high)
```

In quick_sort we are selecting pivot randomly and make two list one with numbers lower than pivot and other with number higher than pivot. And call the quick_sort()recursively.

Implementation of Radix Sort:

I use two implementations of the radix sort one that is using buckets as underlying algorithm

```
def radix_Sort(nums):
    base=10
    result_list = []
    power = 0
    while nums:
        bins = [[] for _ in range(base)]# create 10 bins
        for x in nums:
            bins[x // base**power % base].append(x)#add the numbers to the bins
        nums = []
        for bin in bins:
            for x in bin:
                if x < base**(power+1):
                    result_list.append(x)#append numbers to the sorted list
                else:
                    nums.append(x)#append the remaing number in the unsorted list
        power += 1
   ## print(result_list)
    return result_list
```

Following is the working of radix_Sort([10,52,7,209,19,44,6,78,98,12])

```
[[], [], [], [], [], [], [], [], [], []]
[[10], [], [52, 12], [], [44], [], [6], [7], [78, 98], [209, 19]]
[6]
[6, 7]
[[], [], [], [], [], [], [], [], [], []]
[[209], [10, 12, 19], [], [], [44], [52], [], [78], [], [98]]
[6, 7, 10]
[6, 7, 10, 12]
[6, 7, 10, 12, 19]
[6, 7, 10, 12, 19, 44]
[6, 7, 10, 12, 19, 44, 52]
[6, 7, 10, 12, 19, 44, 52, 78]
[[], [], [], [], [], [], [], [], [], []]
[[], [], [209], [], [], [], [], [], [], []]
[6, 7, 10, 12, 19, 44, 52, 78, 98, 209]
```

The other one using counting sort under hood to sort the list in it first we need to find the largest number in list and to determine the number of passes need to sort the list, as number of passes is equal to the number of digits represent the largest number in the list

```
def radix_sort(A):
    #radix is the base of the number system
    radix = 10
    #k is the largest number in the list
    k = max(A)
    #output is the result list we will build
    output = A
    #compute the number of digits needed to represent k
    digits = int(math.floor(math.log(k, radix)+1))
    #print(digits)# to check
    for i in range(digits):
        output = counting_sort(output,i,radix)
        #print("Pass",i)
        #print(output) #to look at how the array is sorting in every pass

    return output
```

```
def counting_sort(arr, digit, radix):
    #"output" is a list to be sorted, radix is the base of the number system, digit is the di
    #we want to sort by
    n=len(arr)
    #create a list output which will be the sorted list
    output = [0]*n
    count = [0]*int(radix)

    #counts the number of occurences of each digit in arr
    for i in range(0, n):
        digit_of_Arri = int(arr[i]/radix**digit)%radix
        #print(digit_of_Arri)
        count[digit_of_Arri] = count[digit_of_Arri] +1
        #now count[i] is the value of the number of elements in arr equal to i
    #print(count) to look at the count list
    #this FOR loop changes cont to show the cumulative # of digits up to that index of count
    for j in range(1,radix):
        count[j] = count[j] + count[j-1]
        #here count is modifed to have the number of elements <= i
    #print(count) to check working
    for m in range(len(arr)-1, -1, -1): #to count down (go through arr backwards)
        digit_of_Arri = int(arr[m]/radix**digit)%radix

        count[digit_of_Arri] = count[digit_of_Arri] -1
        output[count[digit_of_Arri]] = arr[m]
        #print(output) to check working

    return output
```

```
[10, 52, 5, 209, 19, 44]
[1, 0, 1, 0, 1, 1, 0, 0, 0, 2]
[1, 1, 2, 2, 3, 4, 4, 4, 4, 6]
[0, 0, 44, 0, 0, 0]
[0, 0, 44, 0, 0, 19]
[0, 0, 44, 0, 209, 19]
[0, 0, 44, 5, 209, 19]
[0, 52, 44, 5, 209, 19]
[10, 52, 44, 5, 209, 19]
Pass 0
[10, 52, 44, 5, 209, 19]
[2, 2, 0, 0, 1, 1, 0, 0, 0, 0]
[2, 4, 4, 4, 5, 6, 6, 6, 6, 6]
[0, 0, 0, 19, 0, 0]
[0, 209, 0, 19, 0, 0]
[5, 209, 0, 19, 0, 0]
[5, 209, 0, 19, 44, 0]
[5, 209, 0, 19, 44, 52]
[5, 209, 10, 19, 44, 52]
Pass 1
[5, 209, 10, 19, 44, 52]
[5, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[5, 5, 6, 6, 6, 6, 6, 6, 6, 6]
[0, 0, 0, 0, 52, 0]
[0, 0, 0, 44, 52, 0]
[0, 0, 19, 44, 52, 0]
[0, 10, 19, 44, 52, 0]
[0, 10, 19, 44, 52, 209]
[5, 10, 19, 44, 52, 209]
Pass 2
[5, 10, 19, 44, 52, 209]
Sorted: [5, 10, 19, 44, 52, 209]
```

To benchmarking these sorting algorithms. I use time() module of python to measure the time taken by algoritm to sort the list. Time() retun the time in seconds.

There I use benchmark function that get the name of the function and list as argument and and make a copy of the list now run 10 times sorting funtion on the list and shuffle it after every run than and add the time elapsed in the results after completing 10 runs return the average time in milliseconds

```
"""
def benchmark(func,array):
    results=[]
    unsorted=list(array)
    #print("unsorted",unsorted)# to check results
    for j in range(0,10):
    #unsorted = random_array(i)
        start =time() #start time
        sortedl= list(func(unsorted))
        end=time()#end time
        res=end-start #time elapsed
        np.random.shuffle(unsorted)#shuffle array for next run
        #print("sorted",sortedl)# to cofirm if it is working fine
        results.append(res)
```

Results:

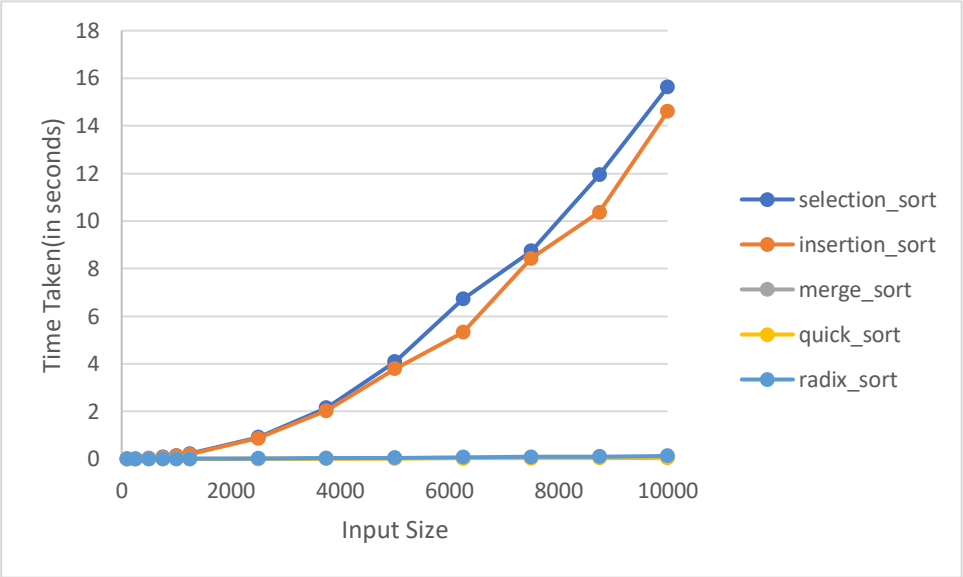
Choosing the best sorting algorithm is as much about knowing what you sorting as it is about the relative performance of the algorithms.

- I used Thirteen arrays of different sizes from 100 -10000
    i. Unsorted
    ii. Sorted
    iii. Reverse sorted

Random integers are obtained with randint() I used only the positive integers

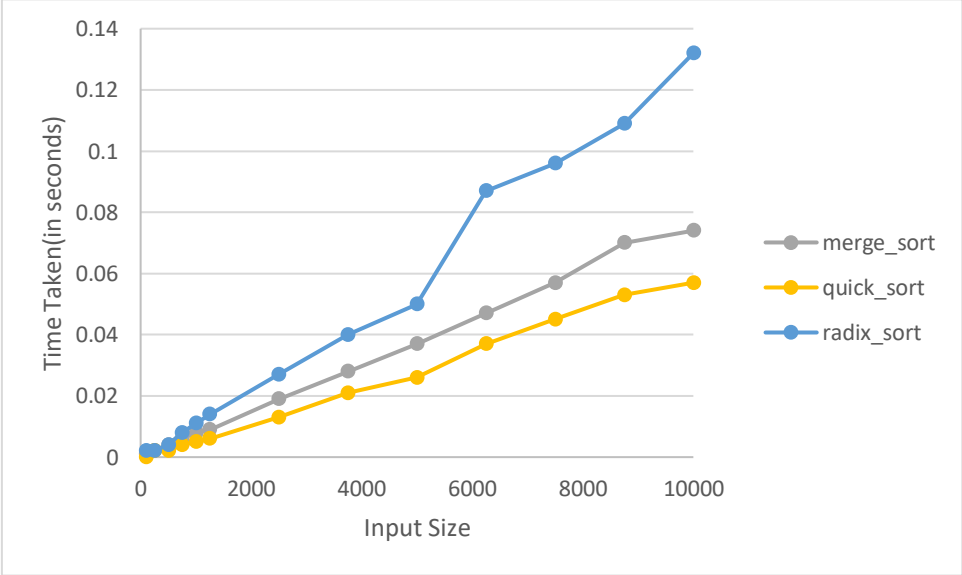
1. First execute **each algorithm on each size of array only once** that give the following results

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
selection_sort	0.001	0.009	0.037	0.084	0.147	0.235	0.915	2.152	4.090	6.725	8.749	11.938	15.632
insertion_sort	0.001	0.008	0.034	0.073	0.138	0.207	0.874	2.032	3.788	5.331	8.439	10.366	14.602
merge_sort	0.001	0.002	0.004	0.006	0.008	0.009	0.019	0.028	0.037	0.047	0.057	0.070	0.074
quick_sort	0.000	0.002	0.002	0.004	0.005	0.006	0.013	0.021	0.026	0.037	0.045	0.053	0.057
radix_sort	0.002	0.002	0.004	0.008	0.011	0.014	0.027	0.040	0.050	0.087	0.096	0.109	0.132



Time is in seconds (the code of this is sort_benchmark1.py) the results

As on this graph there is not clear picture about merge, quick and radix sort so we draw them separately



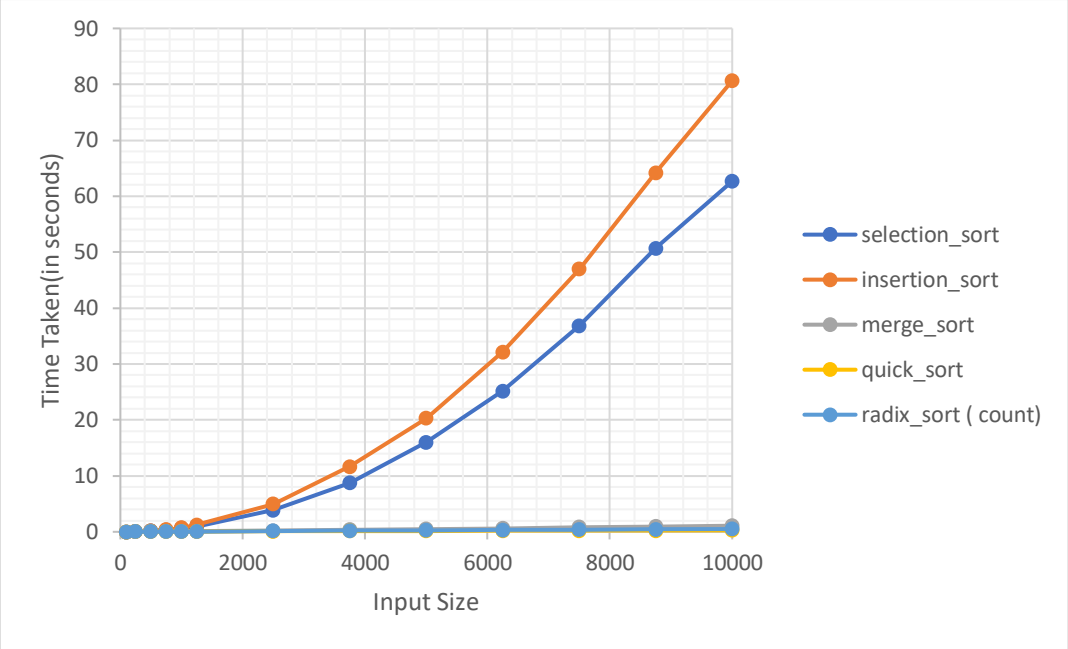
2. Than on next step I looked at the timeit.repeat() module and used it to to make the results more accurately get use the sort function **10 time for one size and take the average of that time.** The code for the is in (avg_benchmark.py)

```
100
Algorithm: selection_sort. average execution time: 0.006
Algorithm: insertion_sort. average execution time: 0.006
Algorithm: merge_sort. average execution time: 0.005
Algorithm: quick_sort. average execution time: 0.004
Algorithm: radix_sort. average execution time: 0.005

250
Algorithm: selection_sort. average execution time: 0.035
Algorithm: insertion_sort. average execution time: 0.04
Algorithm: merge_sort. average execution time: 0.015
Algorithm: quick_sort. average execution time: 0.01
Algorithm: radix_sort. average execution time: 0.012

500
Algorithm: selection_sort. average execution time: 0.15
Algorithm: insertion_sort. average execution time: 0.155
Algorithm: merge_sort. average execution time: 0.033
Algorithm: quick_sort. average execution time: 0.024
Algorithm: radix_sort. average execution time: 0.024

750
Algorithm: selection_sort. average execution time: 0.337
Algorithm: insertion_sort. average execution time: 0.381
Algorithm: merge_sort. average execution time: 0.055
Algorithm: quick_sort. average execution time: 0.03
Algorithm: radix_sort. average execution time: 0.036

1000
Algorithm: selection_sort. average execution time: 0.629
Algorithm: insertion_sort. average execution time: 0.75
Algorithm: merge_sort. average execution time: 0.075
Algorithm: quick_sort. average execution time: 0.039
Algorithm: radix_sort. average execution time: 0.047
```

```
1250
Algorithm: selection_sort. average execution time: 0.965
Algorithm: insertion_sort. average execution time: 1.215
Algorithm: merge_sort. average execution time: 0.097
Algorithm: quick_sort. average execution time: 0.046
Algorithm: radix_sort. average execution time: 0.059

2500
Algorithm: selection_sort. average execution time: 3.887
Algorithm: insertion_sort. average execution time: 4.946
Algorithm: merge_sort. average execution time: 0.234
Algorithm: quick_sort. average execution time: 0.083
Algorithm: radix_sort. average execution time: 0.128

3750
Algorithm: selection_sort. average execution time: 8.734
Algorithm: insertion_sort. average execution time: 11.648
Algorithm: merge_sort. average execution time: 0.358
Algorithm: quick_sort. average execution time: 0.129
Algorithm: radix_sort. average execution time: 0.19


5000
Algorithm: selection_sort. average execution time: 16.029
Algorithm: insertion_sort. average execution time: 20.29
Algorithm: merge_sort. average execution time: 0.498
Algorithm: quick_sort. average execution time: 0.143
Algorithm: radix_sort. average execution time: 0.238

6250
Algorithm: selection_sort. average execution time: 25.189
Algorithm: insertion_sort. average execution time: 32.144
Algorithm: merge_sort. average execution time: 0.633
Algorithm: quick_sort. average execution time: 0.182
Algorithm: radix_sort. average execution time: 0.321

7500
Algorithm: selection_sort. average execution time: 36.821
Algorithm: insertion_sort. average execution time: 46.994
Algorithm: merge_sort. average execution time: 0.819
Algorithm: quick_sort. average execution time: 0.217
Algorithm: radix_sort. average execution time: 0.38

8750
Algorithm: selection_sort. average execution time: 50.673
Algorithm: insertion_sort. average execution time: 64.171
Algorithm: merge_sort. average execution time: 0.935
Algorithm: quick_sort. average execution time: 0.232
Algorithm: radix_sort. average execution time: 0.426

10000
Algorithm: selection_sort. average execution time: 62.677
Algorithm: insertion_sort. average execution time: 80.658
Algorithm: merge_sort. average execution time: 1.084
Algorithm: quick_sort. average execution time: 0.259
Algorithm: radix_sort. average execution time: 0.488
```

these times are in seconds and calculated by using timeit.repeat()

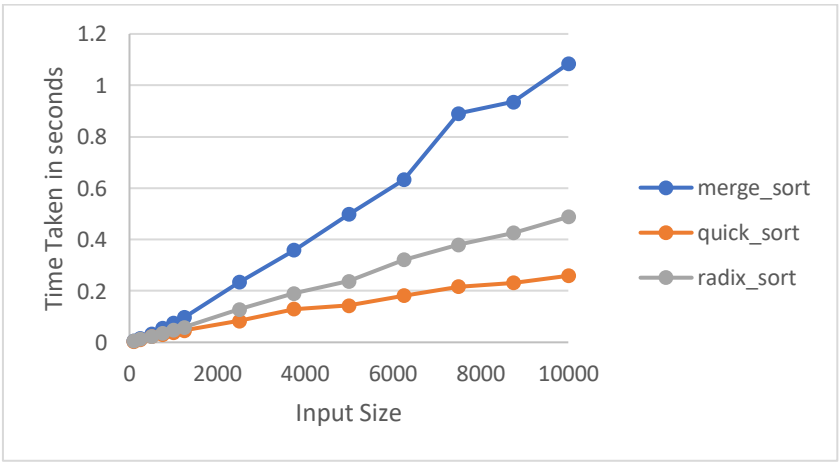
```
times = repeat(setup=setup_code, stmt=stmt, repeat=3, number=10)

# Finally, display the name of the algorithm and the
# minimum time it took to run
print(f"Algorithm: {algorithm}. average execution time: {round(min(times),3)}")
```



	100	250	500	750	1000	1250	2500	3750	5000	6250
selection_sort	0.006	0.035	0.150	0.337	0.629	0.965	3.887	8.734	16.029	25.189
insertion_sort	0.006	0.040	0.155	0.381	0.750	1.215	4.946	11.648	20.290	32.144
merge_sort	0.005	0.015	0.033	0.055	0.075	0.097	0.234	0.358	0.498	0.633
quick_sort	0.004	0.010	0.024	0.030	0.039	0.046	0.083	0.129	0.143	0.182
radix_sort	0.005	0.012	0.024	0.036	0.047	0.059	0.128	0.190	0.238	0.321

	7500	8750	10000
selection_sort	36.821	50.673	62.677
insertion_sort	46.994	64.171	80.658
merge_sort	0.819	0.935	1.084
quick_sort	0.217	0.232	0.259
radix_sort	0.380	0.426	0.488

As it shows that insertion was doing better in one run but when taking average of 10 runs it a bit behind selection sort. And radix sort was also doing better than merge sort



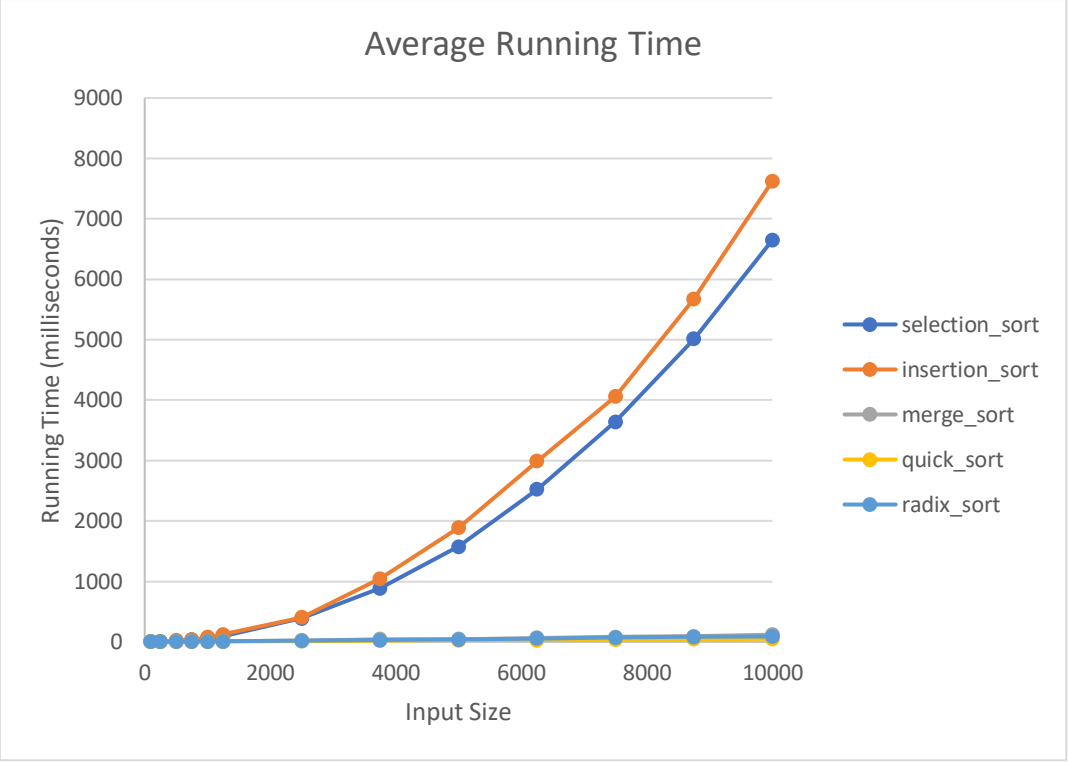
We were using the same list 10 times for and repeating it three times but array was different for each algorithm therefore made a change

3. To get the fair comparison I used the same unsorted array across all algorithms and run the sort function 10 times and shuffle list for every time. Than calculate the **average time in milliseconds for 10 runs**

The results are

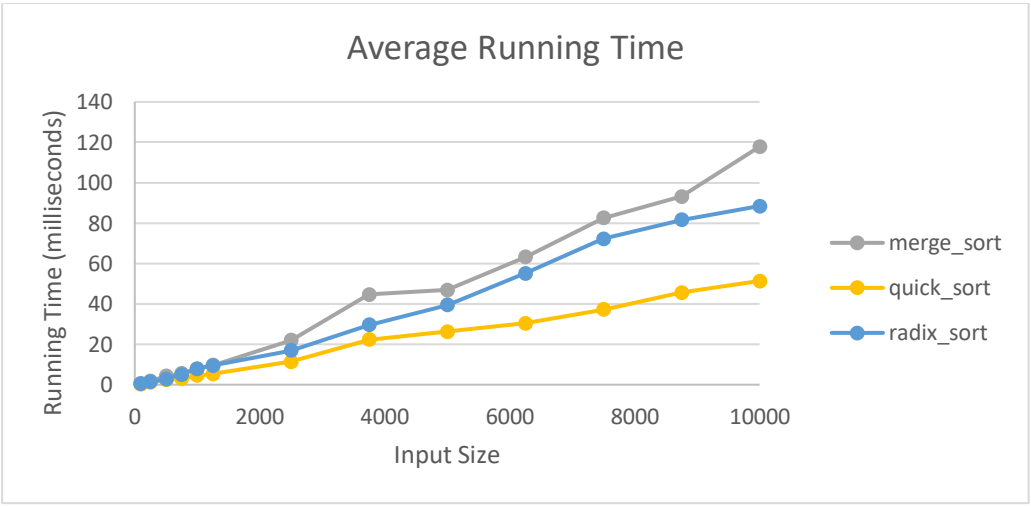
```
Average time taken for 10 runs for each array size
                100    250    500    750    1000    1250    2500     3750   \
selection_sort 0.600 3.198 18.988 33.779 65.658  94.841 388.259  887.150
insertion_sort 0.600 4.098 23.985 34.778 75.453 121.525 406.947 1045.252
merge_sort     0.500 1.899  4.297  5.497  7.895   9.694  21.986   44.572
quick_sort     0.300 1.099  2.498  2.998  4.597   5.297  11.293   22.286
radix_sort     0.400 1.399  2.798  4.997  7.895   9.394  16.890   29.581

                  5000     6250     7500     8750     10000
selection_sort 1575.823 2525.135 3639.244 5015.791 6649.179
insertion_sort 1888.229 2987.748 4062.082 5675.282 7624.274
merge_sort       46.871   63.261   82.448   93.142  117.827
quick_sort       26.284   30.381   37.177   45.672   51.368
radix_sort       39.475   55.066   72.255   81.549   88.445
```

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
selection_sort	0.6	3.198	18.988	33.779	65.658	94.841	388.259	887.15	1575.823	2525.135	3639.244	5015.791	6649.179
insertion_sort	0.6	4.098	23.985	34.778	75.453	121.525	406.947	1045.252	1888.229	2987.748	4062.082	5675.282	7624.274
merge_sort	0.5	1.899	4.297	5.497	7.895	9.694	21.986	44.572	46.871	63.261	82.448	93.142	117.827
quick_sort	0.3	1.099	2.498	2.998	4.597	5.297	11.293	22.286	26.284	30.381	37.177	45.672	51.368
radix_sort	0.4	1.399	2.798	4.997	7.895	9.394	16.89	29.581	39.475	55.066	72.255	81.549	88.445



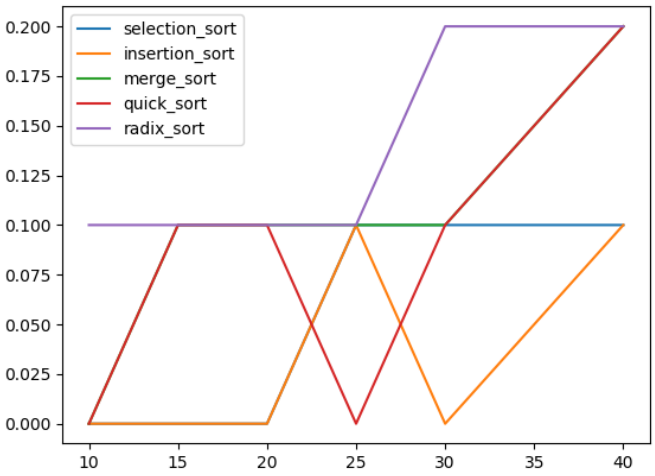
As it is clear from the above results that merge sort, quick sort and radix sort perform better on large sizes lists. In theory selection and insertion sort both quadratic sorts $O(n^2)$ and we can see that in the graph.

To look at the other three sorts I draw a separate graph only showing three sorts here it shows quick sort is faster than merge and radix.



Smaller size Lists:

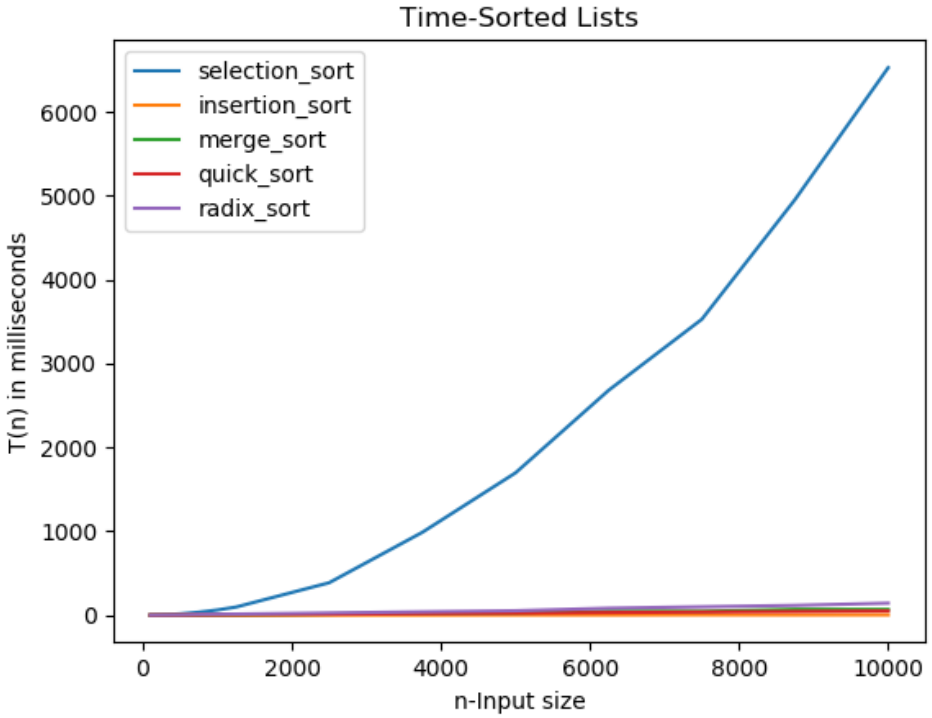
To check the behaviour of sorts for smaller sizes of list. Run the sort on size(10,15,20,25,30,40)



the results for the lists of smaller sizes shows that selection and insertion perform better and radix sort is not a good choice if sorting small lists.

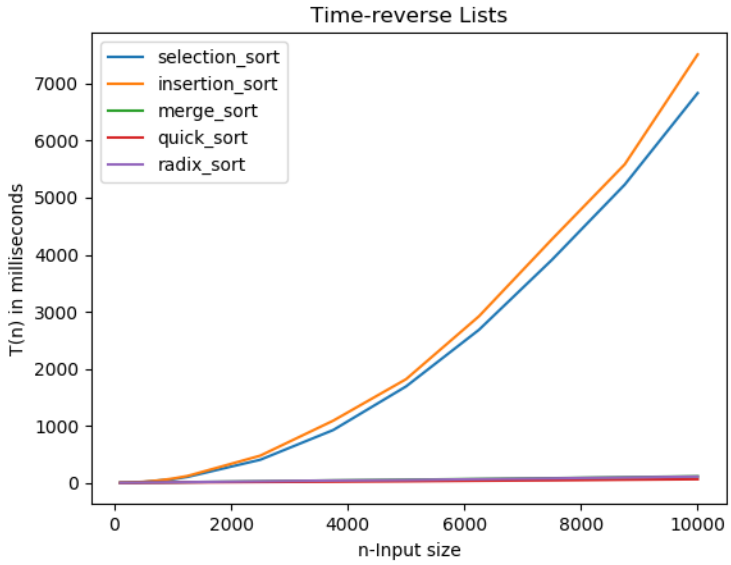
Sorted lists:

As in theory we see that the insertion sort best case time is when have to sort a already sorted list to check that I did the runtime test for all algorithms with sorted lists and it is clear from the following graph that insertion sort perform best on sorted list.



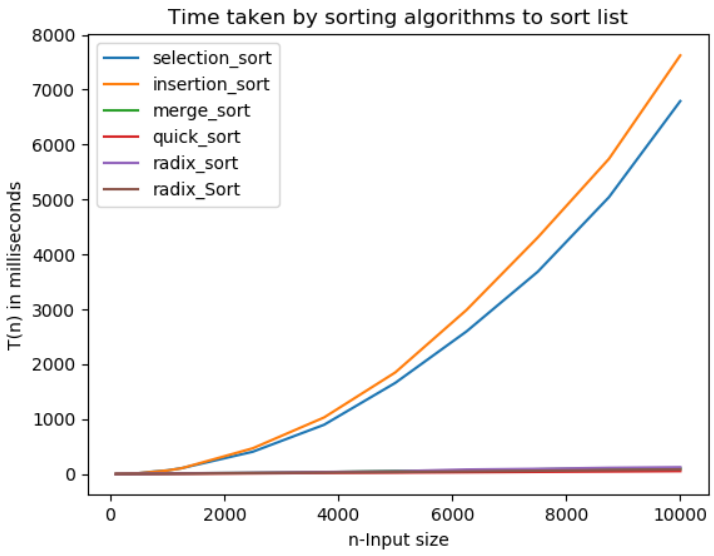
Reverse Lists:

I also tried to time the sorts with reversed lists following is the graph of results and it shows that reverse list does not make a huge difference in times taken by each algorithm to sort the list.

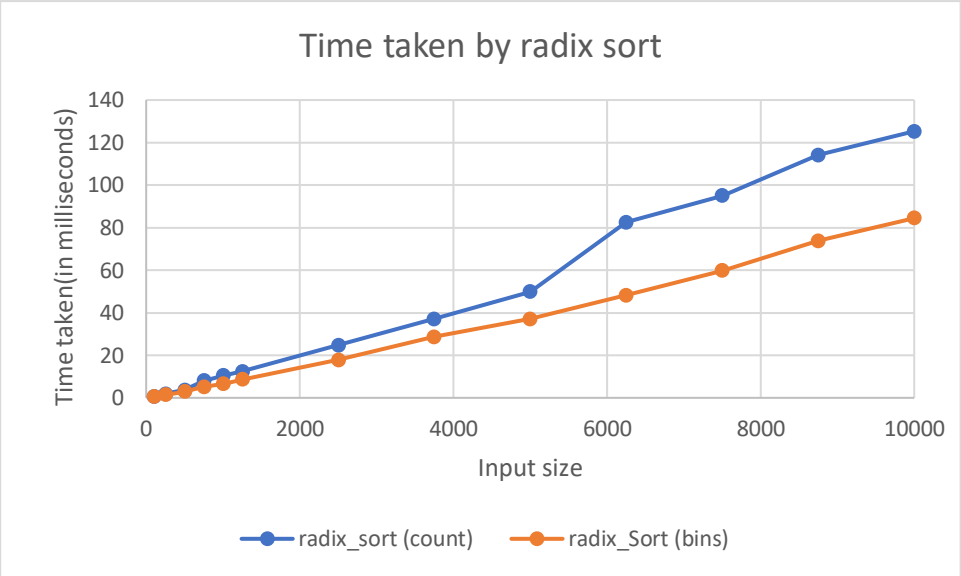


4. As the performance of the radix sort depend upon the underlying algorithm used therefore, I tried both algorithm one using count sort and other using bins and get following results. These results are average run time for 10 runs

	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
selection_sort	0.6	3.695	15.491	43.972	66.359	106.734	406.748	897.444	1660.57	2597.89	3685.12	5043.67	6791.69
insertion_sort	0.6	4.397	15.591	40.475	65.16	105.735	469.708	1030.66	1850.95	2988.75	4308.93	5741.54	7623.68
merge_sort	0.699	1.6	3.497	5.497	7.895	12.492	21.887	34.179	57.463	61.162	85.147	101.437	103.436
quick_sort	0.4	1.199	2.699	3.298	5.397	5.597	11.993	20.987	26.184	32.579	39.875	46.27	53.266
radix_sort	0.699	1.799	3.698	7.995	10.394	12.394	24.784	37.077	49.869	82.548	95.04	114.129	125.321
radix_Sort	0.599	1.4	2.998	5.197	6.597	8.695	17.789	28.682	37.077	48.17	59.862	73.854	84.447



Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
radix_sort (count)	0.669	1.799	3.698	7.995	10.394	12.394	24.784	37.077	49.869	82.548	95.04	114.129	125.321
radix_Sort (bins)	0.599	1.400	2.998	5.197	6.597	8.695	17.789	28.682	37.077	48.17	59.862	73.854	84.447



As we can see the difference between both implementations results. Radix sort using bins are performing better.

Merge sort performs similarly for all types of input
Radix sort is performs very well in all cases
Quicksort has the best all-round performance
Selection sort is clearly the better performer among both quadratic sort implementations. But in theory it says that insertion sort do better but on average it is not the case with my implementation
Insertion sort as expected, does better with sorted data than with reversed data.

Conclusion:

The aim of the project was to research and implement five sorting algorithms and benchmark to assess their performance.

There are two main criteria to judge a sorting algorithm, time taken to sort the given data and memory space required to do so. In this project runtime is used as a performance measure.

For benchmarking, I run all algorithms on a set of test data and extract performance measures from the generated data.

I benchmarked five sorting algorithms in three groups

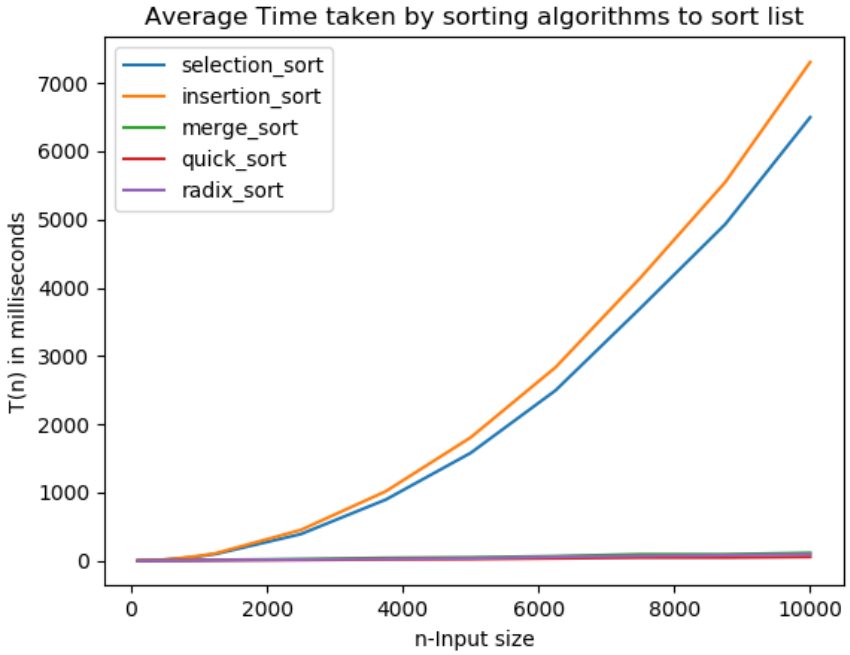
1. **Comparison based sorting**
   Insertion sort :Average and worst case time complexity: n^2
   Best case time complexity: n when array is already sorted.
   Worst case: when the array is reverse sorted.
   Selection sort : Best, average, and worst-case time complexity: n^2 which is independent of distribution of data.
2. **Efficient comparison-based sort**
   Merge sort : Best, average, and worst case time complexity: nlogn which is independent of distribution of data.
   Quick sort: Worst case: when the array is sorted or reverse sorted, the partition algorithm divides the array in two subarrays with 0 and n-1 elements. Therefore,
   time complexity = $O(n^2)$
   Best case and Average case: On an average, the partition algorithm divides the array in two subarrays with equal size. Therefore, time complexity = O(nlogn)
3. **Non-Comparison sorting**
   Radix sort: Best, average and worst-case time complexity: nk where k is the number of digits in largest element of array.

The runtime of the sorting algorithms was measured to see which algorithm performed better.

The quick sort performed better in all the tests. And the results prove it. But I noticed that there were little changes in the runtime one every test. The fluctuation was not by a large amount, it may have been because of factors like cpu, memory and other processes or the random list that is generated is different every time.

The results of the benchmarking are correspond to the big-O notation as obvious from the graphs .

```
Average time taken for 10 runs for each array size
                100    250    500    750    1000    1250    2500     3750   \
selection_sort 0.599 4.597 16.589 35.877 63.361  99.242 390.758  895.645
insertion_sort 0.999 4.497 16.589 40.974 71.155 110.431 451.725 1017.469
merge_sort     0.400 1.499  3.798  5.696  8.295  12.492  23.785   40.675
quick_sort     0.400 0.899  2.098  3.499  4.797   5.896  11.793   20.887
radix_sort     0.500 1.499  2.399  5.696  7.196   8.794  17.889   26.884

                  5000     6250     7500     8750     10000
selection_sort 1579.621 2496.157 3702.505 4932.343 6498.476
insertion_sort 1806.585 2834.548 4146.729 5548.561 7308.073
merge_sort       49.270   66.458   94.341   94.442  115.328
quick_sort       24.984   35.078   45.471   46.271   55.865
radix_sort       35.378   58.163   71.855   80.350   95.940
```



The results show that quicksort and merge sort also pay the price of recursion when the list is sufficiently small, taking longer to complete than both insertion sort and selection.

Each sorting algorithms has its own specific strengths and weaknesses:

• Comparison-based sorts are widely applicable but are limited to $n\log n$ running time in the best case.

• Non-comparison sorts can achieve linear n running time in the best-case scenarios, but they are less flexible.

There is no single algorithm which is best for all input instances.

Insertion sort and merge sort are stable techniques. Selection sort and quick sort is unstable as it may change the order of elements with the same value. quick sort and heap sort are also unstable. radix sort stability depends on the underlying algorithm used for sorting.

As I researched all these algorithms the question, “What is the best sorting algorithm?” really depends what you are sorting. The length of the input array and how sorted it is already, and whether the array values are bounded.

For example, insertion sort is the worst case quadratic n^2, but it is extremely fast for long arrays that are already nearly sorted. It is also better on very short input arrays. For arrays with only small values, radix sort (with counting sort as a subroutine) provides near-linear sort-time. but if you have many small values and a few large ones?

It was an interesting and great learning experience. It took a long time to solve the problems in my coding files as I test each sorting algorithm separately and then combine the result and make that all sorts get the same copy of the list. This improves my coding skills as well.

I read time complexity big O notation in detail and its use in priori analysis, In conclusion overall results were consistent with the theory the only thing that I was not sure about as in theory it says insertion sort perform better than selection but my results showed opposite, even the difference was small.

References:

1. Class notes provided
2. Sorting Algorithms available online at https://www.robasworld.com/sorting-algorithms/ accessed on 10 April 2020
3. Time Complexity of Algorithm available online at https://www.studytonight.com/data-structures/time-complexity-of-algorithms accessed on 12 April 2020
4. https://realpython.com/sorting-algorithms-python accessed on 14 April 2020
5. https://www.lewuathe.com/radix-sort-in-python.html accessed on 16 April 2020
6. https://www.geeksforgeeks.org/timeit-python-examples/ accessed on 17 April 2020
7. https://www.oreilly.com/library/view/algorithms-in-a/9781491912973/ch04.html accessed on 20 April 2020
8. Getting to the root of the sorting with radix sort available online at https://medium.com/basecs/getting-to-the-root-of-sorting-with-radix-sort-f8e9240d4224 accessed on 21 April 2020
9. Big O notation and algorithm analysis with python available onlineat https://stackabuse.com/big-o-notation-and-algorithm-analysis-with-python-examples/ accessed on 23 April 2020
10. Insertion sorting algorithm available online at https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm accessed on 30 April 2020
11. Benchmarking int sorting available online at http://alejandroerickson.com/j/2016/08/02/benchmarking-int-sorting-algorithms.html accessed on 1 May 2020

Appendix:

The code files used for this project are

1	sorting.py	Contains all sorts functions
2	sort_benchmark1.py	Contain the code for benchmark on different size of arrays
3.	avg_bechmark.py	Contain the code for benchmark using timeit.repeat()
4.	final_benchmark.py	Contain the final benchmark code for average time for all 5 sorting algorithms to run on same input for 10 times.
5.	test_benchmark	Contain code used to test on different array size and type
6.	resultsort	Excel file use to save the results dataset