* **Introduction**

In this project I am going to demonstrate the actual code of different sorting algorithms and see the practical outputs

Firstly, lets discuss the asymptotic notation of each algorithm in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Best Case | Average Case | Worst Case |
| Insertion Sort | Ω(n) | θ() | O() |
| Merge Sort | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| Quick Sort | Ω(n log(n)) | θ(n log(n)) | O() |
| Randomized Quick Sort | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| Hybrid Randomized (QuickSort & InsertionSort) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| Heap Sort | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |

In practical considerations if the input is sorted then Insertion Sort are expected to preform better since it is the best case for it, on the other hand it’s the worst case for the quicksort so as we can see the input has a lot of impact on those algorithms performance and in the table above we use asymptotic notation which ignores the constants and this is not accurate so we cant take that table as the only reference, we need to implement multiple test cases with different input sizes to see the performance of each algorithm.

* **Code**

For better visualization the implementation of those algorithms was done in python, specifically Jypter Notebook IDE.

The code at the beginning imports important libraries that is being used then implements each sorting algorithms then uses 3 main methods for sorting, plotting and showing outputs.

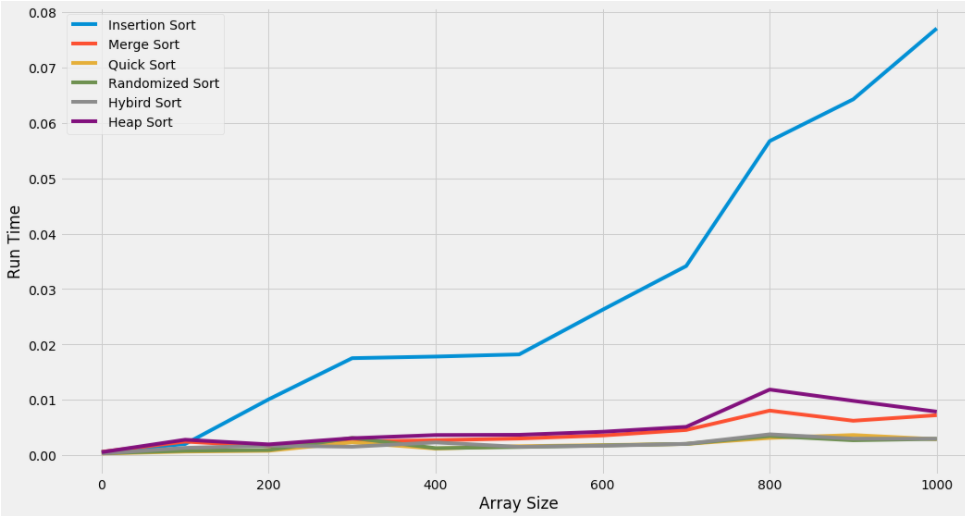
Main methods:

* startTheAlgo(): this function takes maxArraySize, arrayRange, stepSize and mode.

The maxArraySize allows the code to run the algorithms and increase by stepSize until the maxArraySize is reached. The arrayRange is the range of the actual values inside the array. The mode describes the mode of the input array, for example, Sorted, Random input, nearly sorted etc.

* allSorts(): all sorts send the same array into all the sorting algorithms and calculate the running time of each one of them.
* plot(): this method helps plotting and visualizing using the runtime of each algorithm as the Y axis and the array size as the X axis.
* **Test Cases:**

In order to compare between the algorithms let’s use the variables that have been declared before to change some major parts in the code like array size, range etc.

**Test Case 1**: Array will start with size of 100 and increment by 100 until 1000, Array values ranges between 0 and 1200 and the Array is initially inserted as **Random Input**.

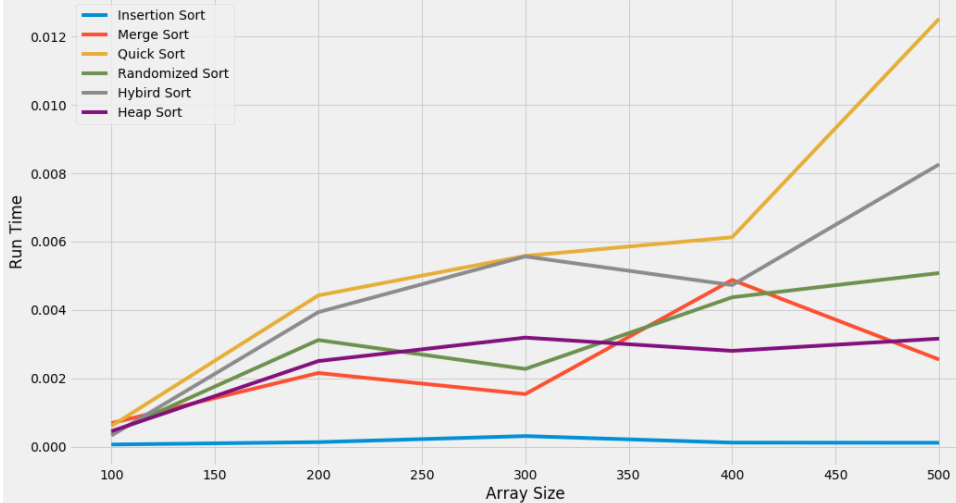
The following table does a comparison for the average time that used for the whole graph.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Insertion Sort | Merge Sort | Quick Sort | Randomized Sort | Hybrid Sort | Heap Sort |
| Avg Run Time | 0.02130 | 0.00337 | 0.00186 | 0.00206 | **0.00172** | 0.00399 |

The following table is shows average of 100 runtimes for test case 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array Size  Average Run time | 100 | 1000 | 5000 | 10000 | 20000 |
| Insertion Sort | 0.00085 | 0.0602 | 1.445 | 5.617 | 23.33 |
| Merge Sort | 0.00066 | 0.00639 | 0.03370 | 0.068 | 0.1514 |
| Quick Sort | **0.00029** | **0.00309** | **0.01654** | 0.035 | 0.0741 |
| Randomized Quick Sort | 0.00033 | 0.00310 | 0.01662 | 0.0347 | **0.0710** |
| Hybrid Randomized (QuickSort & InsertionSort) | 0.00041 | 0.00311 | 0.016693 | **0.0345** | 0.07462 |
| Heap Sort | 0.00070 | 0.00803 | 0.04510 | 0.0953 | 0.20711 |

By observing the pervious table, we can see that Quick Sort, Randomized Quick Sort and Hybrid Randomized have the best average run times when increasing the array size and insertion sort having the worst average run time.

**Test case 2:** Thearray will start with size of 100 and increment by 100 until 500, Array values ranges between 0 and 1200 and the Array is initially inserted as **Sorted**.

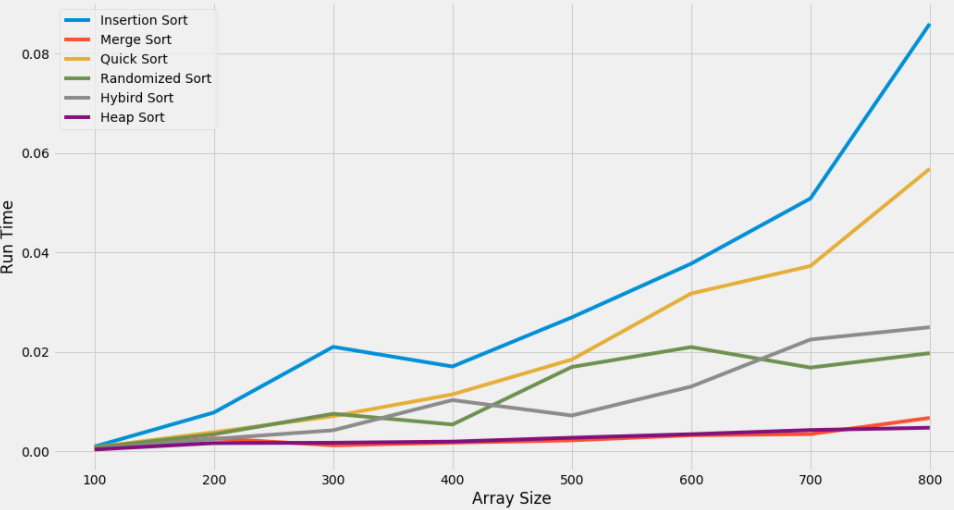
The following table does a comparison for the average time that used for the whole graph.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Insertion Sort | Merge Sort | Quick Sort | Randomized Sort | Hybrid Sort | Heap Sort |
| Avg Run Time | **0.00014** | 0.00236 | 0.00584 | 0.00304 | 0.00456 | 0.00242 |

The following table is shows average of 100 runtimes for test case 2:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array Size  Average Run time | 100 | 1000 | 5000 | 10000 | 20000 |
| Insertion Sort | **0.000032** | **0.00031** | **0.00202** | **0.0029** | **0.00697** |
| Merge Sort | 0.00045 | 0.00610 | 0.0394 | 0.0688 | 0.1570 |
| Quick Sort | 0.00050 | 0.04735 | Recursion Error | Recursion Error | Recursion Error |
| Randomized Quick Sort | 0.000401 | 0.03329 | Recursion Error | Recursion Error | Recursion Error |
| Hybrid Randomized (QuickSort & InsertionSort) | 0.000409 | 0.03078 | Recursion Error | Recursion Error | Recursion Error |
| Heap Sort | 0.00053 | 0.00923 | 0.0562 | 0.1080 | 0.2521 |

By observing the pervious table, we can see that Insertion Sort have the best average run time when increasing the array size and Quick Sort having the worst average run time, the reason behind that is the input array is already sorted.

**Test case 3**: Array will start with size of 100 and increment by 100 until 800, Array values ranges between 0 and 1200 and the Array is initially inserted as **Reverse Sorted**.

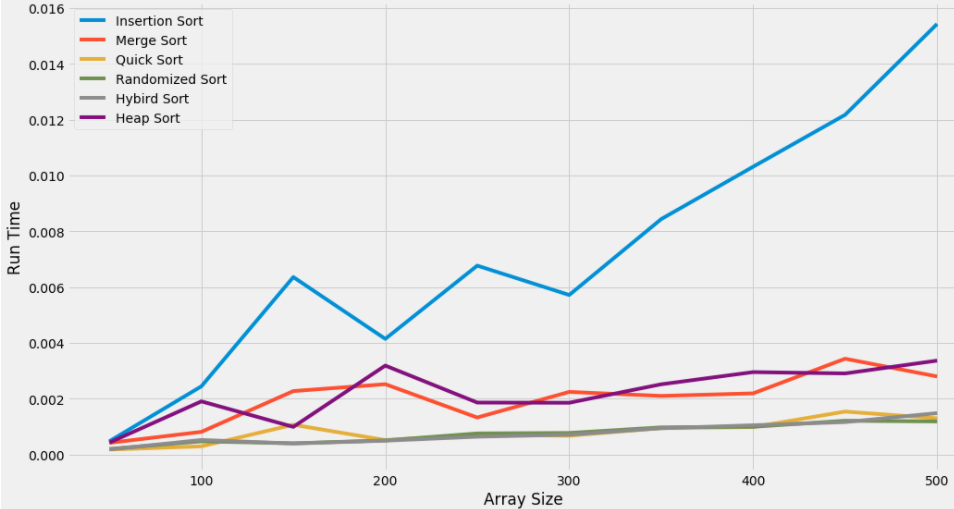
The following table does a comparison for the average time that used for the whole graph.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Insertion Sort | Merge Sort | Quick Sort | Randomized Sort | Hybrid Sort | Heap Sort |
| Avg Run Time | 0.0310 | 0.0027 | 0.0209 | 0.0114 | 0.0106 | **0.0026** |

The following table is shows average of 100 runtimes for test case 3:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array Size  Average Run time | 100 | 1000 | 5000 | 10000 | 20000 |
| Insertion Sort | 0.00109 | 0.123 | 2.79 | 15.13 | 45.663 |
| Merge Sort | **0.0004300** | **0.00588** | **0.02967** | **0.0801** | **0.1398** |
| Quick Sort | 0.00088 | 0.09320 | Recursion Error | Recursion Error | Recursion Error |
| Randomized Quick Sort | 0.00059 | 0.05549 | Recursion Error | Recursion Error | Recursion Error |
| Hybrid Randomized (QuickSort & InsertionSort) | 0.00060 | 0.05346 | Recursion Error | Recursion Error | Recursion Error |
| Heap Sort | 0.0004350 | 0.00723 | 0.0399 | 0.11492 | 0.20552 |

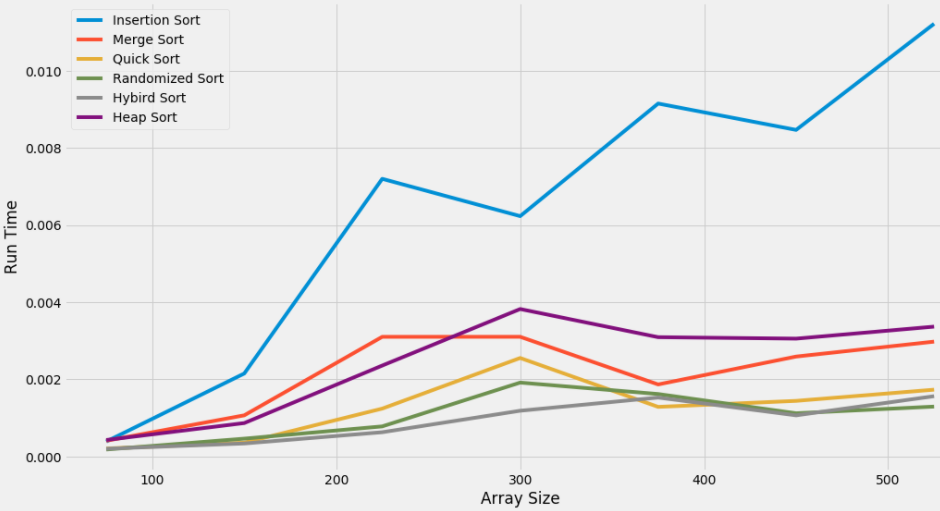
**Test case 4:** The array will start with size of 100 and increment by 50 until 500, Array values ranges between 0 and 1200 and the Array is initially inserted as **Partially Sorted**.

The following Graph implements test case 4 with 30% of the input is sorted:

The following table does a comparison for the average time that used for the whole graph.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Insertion Sort | Merge Sort | Quick Sort | Randomized Sort | Hybrid Sort | Heap Sort |
| Avg Run Time | 0.00722 | 0.00201 | 0.00082 | **0.00074** | 0.00075 | 0.00219 |

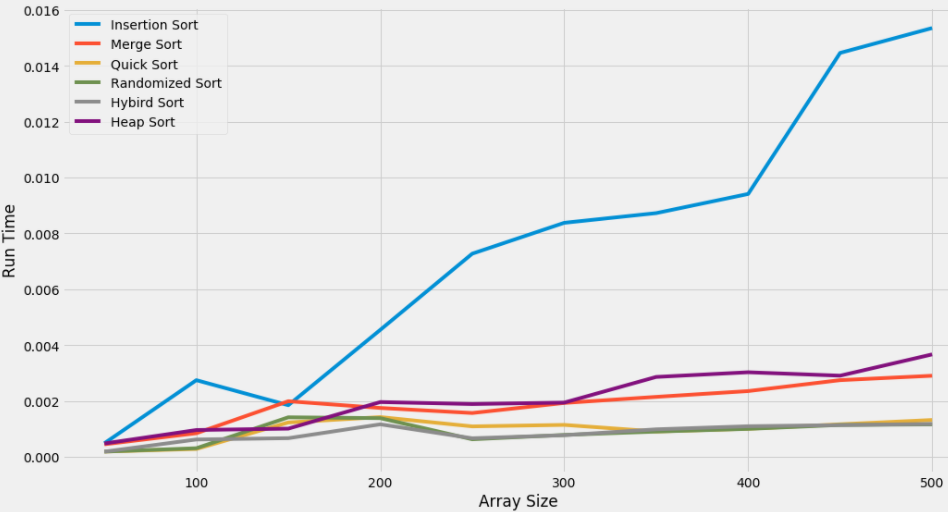
The following Graph implements test case 4 with 50% of the input is sorted:



The following table does a comparison for the average time that used for the whole graph.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Insertion Sort | Merge Sort | Quick Sort | Randomized Sort | Hybrid Sort | Heap Sort |
| Avg Run Time | 0.00640 | 0.00216 | 0.00126 | 0.00105 | **0.00093** | 0.00242 |

The following Graph implements test case 4 with 70% of the input is sorted:



The following table does a comparison for the average time that used for the whole graph.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Insertion Sort | Merge Sort | Quick Sort | Randomized Sort | Hybrid Sort | Heap Sort |
| Avg Run Time | 0.00732 | 0.00186 | 0.00097 | 0.00089 | **0.00084** | 0.00207 |

The following table is shows average of 100 runtimes for test case 4 (50% Sorted):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array Size  Average Run time | 100 | 1000 | 5000 | 10000 | 20000 |
| Insertion Sort | 0.00040 | 0.0522 | 1.288 | 4.2591 | 18.23 |
| Merge Sort | 0.00042 | 0.00668 | 0.0374 | 0.0673 | 0.1528 |
| Quick Sort | 0.000204 | 0.00397 | 0.02355 | 0.0446 | 0.1008 |
| Randomized Quick Sort | 0.000205 | **0.00347** | 0.02082 | 0.0384 | 0.0858 |
| Hybrid Randomized (QuickSort & InsertionSort) | **0.000203** | 0.00356 | **0.02053** | **0.03813** | **0.0844** |
| Heap Sort | 0.000476 | 0.00882 | 0.051308 | 0.10444 | 0.2176 |

* **Conclusion:**

I conclude that after running all those test cases that Merge and Heap sort are the most stable, they offer good performance for different inputs but they are not the fastest. Most other algorithms offer faster run times but they have a worst case depending on the input, for example the randomized and hybrid sort have the fastest run time in almost all cases but when the input was reverse sorted and array size was huge (test case 3) it created a recursion error due to too many recursive calls and the CPU memory stack raised an error. The slowest algorithm in most cases is insertion sort except a special case which is the input already sorted (test case 2) which may happen very rare.