**MINI PROJECT REPORT**



**SORTING ALGORITHMS AND PLOTTING GRAPHS IN PYTHON**

NNN

**Submitted By:**

Name: SHOURYA NAGPAL

University Roll No. 1900320100154

**SUBMITTED TO:**

**Department of Computer Science & Engineering**

ABES ENGINEERING COLLEGE

GHAZIABAD

|  |
| --- |
| **CONTENTS** |
| 1. **ACKNOWLEDGEMENT** |
| 1. **DECLARATION** |
| 1. **LIST OF TABLES** |
| 1. **LIST OF FIGURES** |
| 1. **CHAPTER -1 INTRODUCTION TO PROJECT**   **5.1 AIM**  **5.2 OBJECTIVE** |
| 1. **CHAPTER -2 SURVEY**   **6.1 WHY DO WE NEED SORTING ALGORITHMS**  **6.2 WHY DIFFERENT SORTING ALGORITHMS**  **6.3 COMMON APPLICATIONS OF SORTING**  **6.4 REAL WORLD APPLICATIONS OF SORTING** |
| 1. **CHAPTER -3 TOOLS & TECHNOLOGY USED** |
| 1. **CHAPTER -4 CODE & SNAPSHOTS**   **7.1 BUBBLE SORT**  **7.2 INSERTION SORT**  **7.3 SELECTION SORT**  **7.4 MERGE SORT**  **7.5 QUICK SORT**  **7.6 HEAP SORT** |
| 1. **CHAPTER -5 RESULTS & DISCUSSIONS**   **9.1 TIME COMPLEXITY ANALYSIS**  **9.2 GRAPHS**  **9.2.1 BAR GRAPH**  **9.2.2 LINE GRAPH**  **9.2.3 PIE GRAPH** |
| 1. **CHAPTER -6 CONCLUSION & FUTURE**   **10.1**  **3 most common O(nlogn)comparison sorts**  **10.2 Which one is the best ?** |
| 1. **CHAPTER -7 REFERENCES** |

**ACKNOWLEDGEMENT**

I would like to acknowledge everyone who played a role in my accomplishment of this project. First of all, my parents, who supported me with love and understanding. Secondly, my teachers who helped me gather knowledge and skill to be able to complete this project successfully. And most importantly the mentor of my mini project, **Prof Santosh Kumar**, who was there to help and guide me through any and all of the difficulties I could ever face during the period of making of this bot. Thank you all for your unwavering support.

**DECLARATION**

I hereby declare that the Report entitled ("SORTING ALGORITHMS AND PLOTTING GRAPHS USING PYTHON") is an authentic record of my own work as requirements of Mini Project during the period from 20 October 2020 to 10 November 2020 for the award of degree of B.Tech. (Computer Science & Engineering), ABES Engineering College, Ghaziabad, under the guidance of (Prof.

Santosh Kumar).

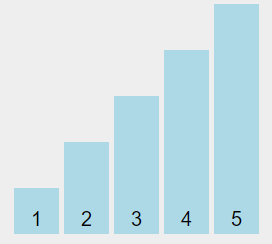
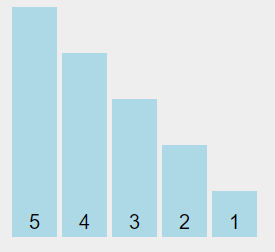
**SHOURYA NAGPAL**

**Date: 10 November 2020 1900320100154**

**LIST OF TABLES**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Sorting Techniques** | Merge  sort | Quick Sort | Selection Sort | Bubble  sort | Insertion Sort | Counting Sort | Heap Sort |
| **Time Taken** | 0.015594959259033203 | 0.015623331069946289 | 0.07999277114868164 | 0.26006364822387695 | 0.18012309074401855 | 0.00800013542175293 | 0.007998943328857422 |

**LIST OF FIGURES**

****

UNSORTED SORTED

**INTRODUCTION TO PROJECT**

SORTING ALGORITHMS CODE AND PLOTTING GRAPHS IN PYTHON USING MATPLOTLIB

Sorting means to arrange a following set of numbers in ascending/increasing/non decreasing or descending/decreasing/non increasing order, and we need certain algorithms in programming to implement the same.

We have various Sorting – Algorithms that we have covered in our project :

* Bubble- Sort
* Insertion-Sort
* Selection-Sort
* Merge-Sort
* Quick-Sort
* Heap-Sort

AIM

To implement **Sorting Algorithms** Code and analyse their **Time and Space Complexity** in **python** and plot time v/s input graph using **Matplotlib** .

OBJECTIVE

To Find the Best & Suitable Sorting Algorithm to Sort a Given Set Of Numbers and Plot Time v/s Input BarGraph using Matplotlib.

**SURVEY**

# *Why do we need sorting algorithms?*

A **Sorting Algorithms** will put items in a list into an order, such as alphabetical or numerical order.

For example, a list of customer names could be sorted into alphabetical order by surname, or a list of people could be put into numerical order by age.

*Why Different Sorting Algorithms ?*

In This modern scenario , we have different situations and we encounter different problems So we need different Sorting Techniques .

Some Algorithms complexity changes based on pre-sortedness and these algorithms are knows to be adaptive.( Insertion Sort is best example)

Problems with the applications with Divide and Conquer require algorithms like QuickSort and Merge Sort.

If the files are small with large values, selections sort works well. It doesn’t require any additional space.

Algorithms like Shell sort are best for medium size files. It is fastest of all O(n^2) sorting algorithms. If the array is in right order, shell sort works well.

Now, you got why we need so many sorting algorithms.  
 ***What are the uses of different sorting algorithms like bubble, selection, insertion, shell, merge, heap, quick, tree, radix, counting and bucket sort in real-life scenarios?***

*Common Applications of Sorting:*

* ***Searching*** -   
   In Binary Search we can Search Items in log(n) that requires Sorted list of items .
* ***Closest pair*** *-*    
   In The Situation when we want to find the closest pair of items who have smallest difference or maximum sum or greatest product in a given list of items .
* ***Element uniqueness*** *-*  
   To remove duplicates from Given set of data .
* ***Frequency distribution*** *-*  
   To See the Frequency of how many times an entry or data is appeared in a data .
* ***Selection*** *-*   
   To see the Kth largest element in the data .

While some of these problems (particularly median and selection) can be solved in linear time using more sophisticated algorithms, sorting provides quick and easy solutions to all of these problems. It is a rare application whose time complexity is such that sorting proves to be the bottleneck, especially a bottleneck that could

have otherwise been removed using more clever algorithms.

***Real World Applications of Sorting:***

* ***Merge Sort:*** Databases use an external merge sort to sort sets of data that are too large to be loaded entirely into memory. The driving factor in this sort is the reduction in the number of disk I/Os.
* ***Bubble Sort:*** *B*ubble sort is used in programming TV remote to sort channels on the basis of longer viewing time .
* ***Heap Sort:*** Heap sort is used in reading barcodes on plastic cards. The service allows to communicate with the database to constantly run checks to ensure that they were all still online and had to constantly report statistics on which readers were performing the worst, which ones got the most/least user activity, etc.
* ***Quick Sort:*** *Sports scores are organised by quick sort on the basis of win-loss ratio .*
* ***Radix Sort:*** eBay allows you to sort listings by the current Bid amount leveraging radix sort .
* ***SelectionSort:*** K12 education portal allows sorting list of pupils alphabetically through selection sort.

**TOOLS & TECHNOLOGY USED**

1. A computer system with any OS.
2. Python Programming language installed.
3. Pip package installed(already included in installation package of python 3.4 or above).
4. Python libraries – Matplotlib(install using pip if not present already).
5. Internet browser(preferably Google Chrome)
6. Google account.
7. Code editor for python/Python IDE(ex - Visual Studio Code, IDLE , Jupyter Notebook).
8. Microsoft Excel.

**SNAPSHOTS OF SORTING-TECHNIQUES**

**BUBBLE-SORT**

Bubble-Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

**ALGORITHM :**

begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort

**CODE:**

def bubbleSort(array):

n=len(array)

for i in range(n):

for j in range(0,n-i-1):

if(array[j] > array[j+1]):

array[j],array[j+1] = array[j+1],array[j]

return array

**ANALYSIS:**

(n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1 / / sum of n numbers

Sum = n(n-1)/2

i.e O(n^2)

* Worst Case Time Complexity [ Big-O ]: **O(n^2)**
* Best Case Time Complexity [Big-omega]: **O(n)**
* Average Time Complexity [Big-theta]: **O(n^2)**
* Space Complexity: **O(1)**

**INSERTION-SORT**

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 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort .

**ALGORITHM:**

INSERTION-SORT(A)

for i = 1 to n

key ← A [i]

j ← i – 1

while j > = 0 and A[j] > key

A[j+1] ← A[j]

j ← j – 1

End while

A[j+1] ← key

End for

**CODE:** def InsertionSort(arr):

for index in range(1,len(arr)):

currentvalue = arr[index]

pos = index

while pos>0 and arr[pos-1]>currentvalue:

arr[pos]=arr[pos-1]

pos = pos-1

arr[pos]=currentvalue

ANALYSIS:

As expected, the algorithm's complexity is ***O*(*n^*2)**.When analyzing algorithms, the average case often has the same complexity as the worst case. So insertion sort, on average, takes ***O*(*n^*2)** time.

Insertion sort has a fast best-case running time and is a good sorting algorithm to use if the input list is already mostly sorted. For larger or more unordered lists, an algorithm with a faster worst and average-case running time, such asMERGESORT ,would be a better choice.

Insertion sort is a stable sort with a [space complexity](https://brilliant.org/wiki/space-complexity/) of ***O*(1)**.

**SELECTION SORT**

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of Ο(n^2), where n is the number of items.

**CODE:**

def SelectionSort(A):

for i in range(len(A)):

mini = i

for j in range(i+1, len(A)):

if A[mini] > A[j]:

mini = j

A[i], A[mini] = A[mini], A[i]

**ANALYSIS:**

Selection Sort requires two nested for loops to complete itself, one for loop is in the function Selection Sort and inside the first loop we are making a call to another function

index of minimum , which has the second(inner) for loop.

Hence for a given input size of n , following will be the time and space complexityfor selection sort algorithm:

Worst Case Time Complexity [ Big-O ] : **O(n^2)**

Best Case Time Complexity [Big-omega]: **O(n^2)**

Average Time Complexity [Big-theta]: **O(n^2)**

Space Complexity: **O(1)**

**MERGE-SORT**

**ALGORITHM:**

procedure mergesort( var a as array )

if ( n == 1 ) return a

var l1 as array = a[0] ... a[n/2]

var l2 as array = a[n/2+1] ... a[n]

l1 = mergesort( l1 )

l2 = mergesort( l2 )

return merge( l1, l2 )

end procedure

procedure merge( var a as array, var b as array )

var c as array

while ( a and b have elements )

if ( a[0] > b[0] )

add b[0] to the end of c

remove b[0] from b

else

add a[0] to the end of c

remove a[0] from a

end if

end while

while ( a has elements )

add a[0] to the end of c

remove a[0] from a

end while

while ( b has elements )

add b[0] to the end of c

remove b[0] from b

end while

return c

end procedure

**CODE:**

def merge(A,B):

m,n=len(A),len(B)

C=[]

i,j=0,0

while i+j < m+n :

if i==m :

C.append(B[j])

j+=1

elif j==n:

C.append(A[i])

i+=1

elif A[i] <= B[j] :

C.append(A[i])

i+=1

elif A[i] > B[j] :

C.append(B[j])

j+=1

return C

def MergeSort(A,left,right):

if right-left <= 1:

return A[left:right]

else :

mid=(right+left)//2

L=MergeSort(A,left,mid)

R=MergeSort(A,mid,right)

return merge(L,R)

**ANALYSIS:**

Merge sort has one of **best time** complexity among pairwise comparison sorting algorithms. the other one is heap sort.

Think of merge sort as an array with is systematically halved until every portion has just but one element. for an array of size ’n’ this goes on to n-1 times being cut into 2 equal halves sorting them(that will be the time to sort (n/2) and merging them back . So the recurrence relation for time to sort ’n’ values will take the form

T(n)=T(n/2)+T(n/2)+n (1)

Now if we substitute(n/2)in the place of n in (1) and substitute it back into (1)it we get –

=>T(n)=2.(2.T(n/4)+n2)+nT(n)

=>4.T(n/4)+2n

generally this will become

T(n)=2i.T(n/2i)+i.n.....(2)

eventually we know T(1)=1T(1)=1 if we want to keep on back substituting for n/4, n/8 e.t.c we know at some point

n/2i = 1

=>i = log2(n)

substituting this back to – (1) we get

T(n)=n+n.log2(n)

Worst Case Time Complexity [ Big-O ]: **O(n\*log n)**

Best Case Time Complexity [Big-omega]: **O(n\*log n)**

Average Time Complexity [Big-theta]**: O(n\*log n)**

Space Complexity: **O(n)**

**QUICK-SORT**

QuickSort is a **Divide and Conquer algorithm**. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as **pivot**.
2. Always pick last element as pivot (implemented below)
3. Pick a **random** element as pivot.
4. Pick median as pivot.

The key process in **quickSor**t is **partition()**. Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

**ALGORITHM**:

quickSort(array, leftmostIndex, rightmostIndex)

if (leftmostIndex < rightmostIndex)

pivotIndex <- partition(array,leftmostIndex, rightmostIndex)

quickSort(array, leftmostIndex, pivotIndex)

quickSort(array, pivotIndex + 1, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)

set rightmostIndex as pivotIndex

storeIndex <- leftmostIndex - 1

for i <- leftmostIndex + 1 to rightmostIndex

if element[i] < pivotElement

swap element[i] and element[storeIndex]

storeIndex++

swap pivotElement and element[storeIndex+1]

return storeIndex +1

**CODE:**

def QuickSort(A,l,r):

if r-l <= 1 :

return ()

yellow=l+1

for green in range(l+1,r):

if A[green] <= A[l]:

A[yellow],A[green]=A[green],A[yellow]

yellow+=1

A[l],A[yellow-1]=A[yellow-1],A[l]

QuickSort(A,l,yellow-1)

QuickSort(A,yellow,r)

**ANALYSIS:**

Worst Case Complexity [Big-O]: **O(n^2)**

It occurs when the pivot element picked is either the greatest or the smallest element.

This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n-1 elements. Thus, quicksort is called only on this sub-array.

However, the quick sort algorithm has better performance for scattered pivots.

Best Case Complexity [Big-omega] :**O(n\*log n)**

It occurs when the pivot element is always the middle element or near to the middle element.

Average Case Complexity [Big-theta]: **O(n\*log n)**  
 It occurs when the above conditions do not occur.  
 Space Complexity

The space complexity for quicksort is **O(n\*log n)**

**HEAP-SORT**

Heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. The improvement consists of the use of a heap data structure rather than a linear-time search to find the maximum.

The heapsort algorithm involves preparing the list by first turning it into a max heap. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap. This repeats until the range of considered values is one value in length

**ALGORITHM:**

MAXHEAPIFY

Begin

for i := 1 to size do

node := i

par := floor (node / 2)

while par >= 1 do

if array[par] < array[node] then

swap array[par] with array[node]

node := par

par := floor (node / 2)

done

done

End

HEAPSORT

Begin

for i := n to 1 decrease by 1 do

heapify(array, i)

swap array[1] with array[i]

done

End

**CODE:**

def heapify(arr, n, root):

largest = root

left = 2 \* root + 1

right = 2 \* root + 2

if left < n and arr[root] < arr[left]:

largest = left

if right < n and arr[largest] < arr[right]:

largest = right

if largest != root:

arr[root], arr[largest] = arr[largest], arr[root]

# Heapify the root.

heapify(arr, n, largest)

def HeapSort(arr):

n = len(arr)

# Build a maxheap.

for i in range(n, -1, -1):

heapify(arr, n, i)

for i in range(n - 1, 0, -1):

arr[i], arr[0] = arr[0], arr[i]

heapify(arr, i, 0)

**ANALYSIS :**

Heap sort is an in-place algorithm.  
 ts typical implementation is not stable, but can be made stable

Time Complexity: Time complexity of heapify is **O(N\*LogN)**. Time complexity of createAndBuildHeap() is O(N) and overall time complexity of Heap Sort is O(N\*LogN) where N is the number of elements in the list or array.

Heap sort algorithm has limited use because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

**RESULTS & DISCUSSIONS**

**Time complexity Analysis** –

We have discussed the best, average and worst case complexity of different sorting techniques with possible scenarios.

**Comparison based sorting** –

In comparison based sorting, elements of an array are compared with each other to find the sorted array.

* **Bubble sort and 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.
* **Merge sort** –  
  Best, average and worst case time complexity: nlogn which is independent of distribution of data.
* **Heap sor**t –  
  Best, average and worst case time complexity: nlogn which is independent of distribution of data.
* **Quick sort** –  
  It is a divide and conquer approach with recurrence relation:

**Stable/Unstable technique –**

A sorting technique is stable if it does not change the order of elements with the same value.

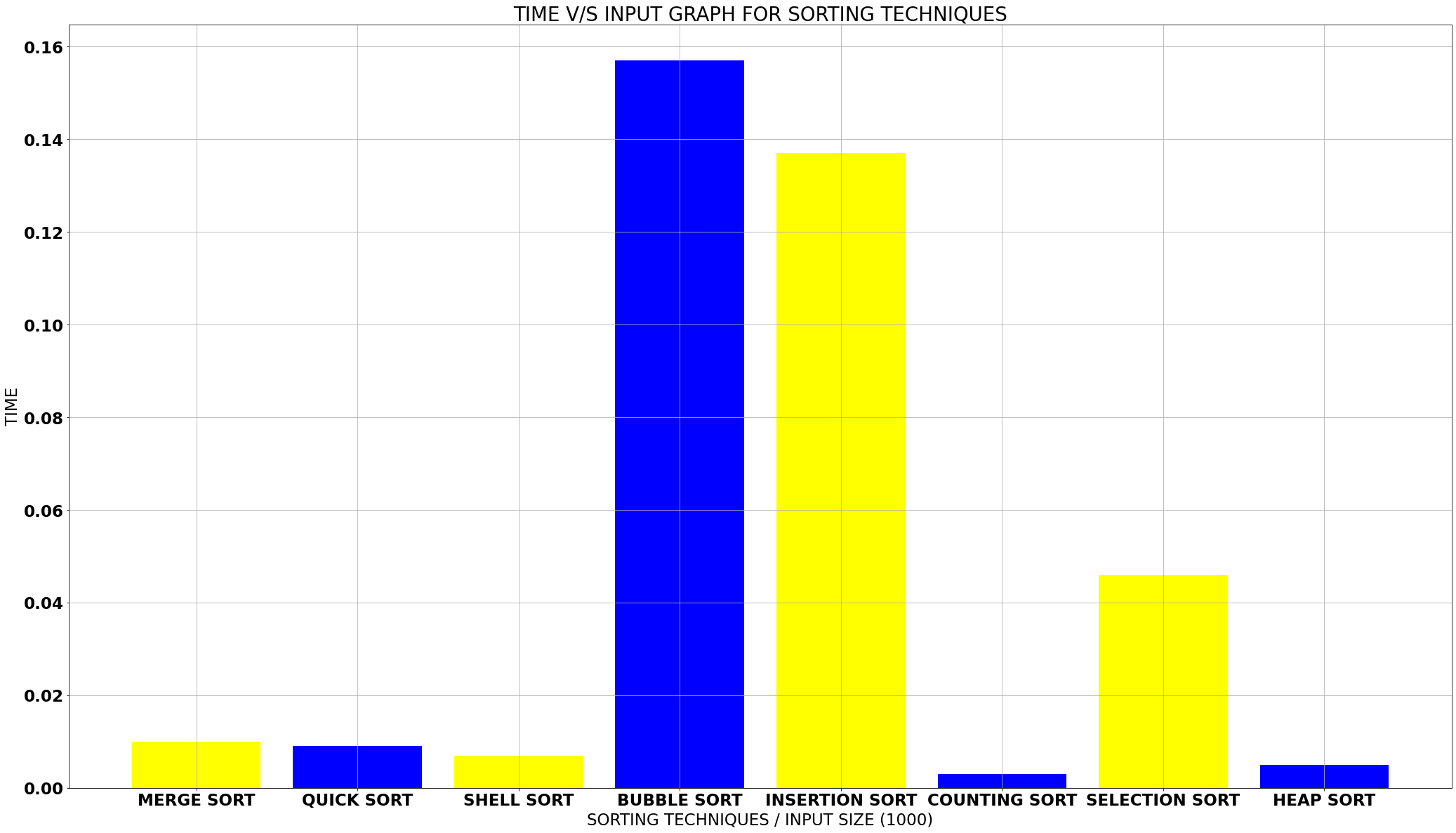
Out of comparison based techniques, bubble sort, insertion sort and merge sort are stable techniques. Selection sort is unstable as it may change the order of elements with the same value. For example, consider the array 4, 4, 1, 3.

In the first iteration, the minimum element found is 1 and it is swapped with 4 at 0th position. Therefore, the order of 4 with respect to 4 at the 1st position will change. Similarly, quick sort and heap sort are also unstable.

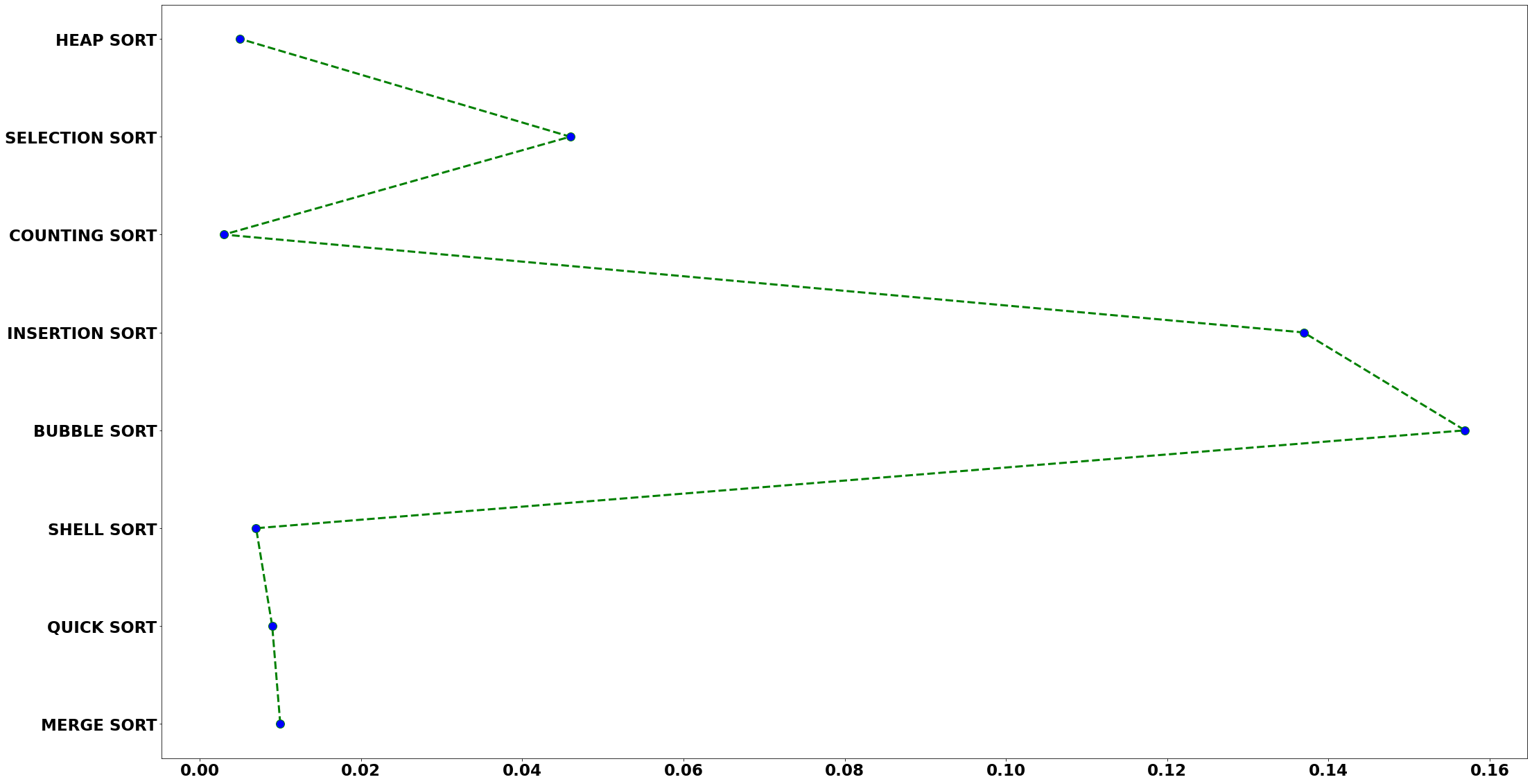
* When the array is almost sorted, insertion sort can be preferred.
* When order of input is not known, merge sort is preferred as it has worst case time complexity of nlogn and it is stable as well.
* When the array is sorted, insertion and bubble sort gives complexity of n but quick sort gives complexity of n^2.

**GRAPH PLOTTING (USING MATPLOTLIB)**

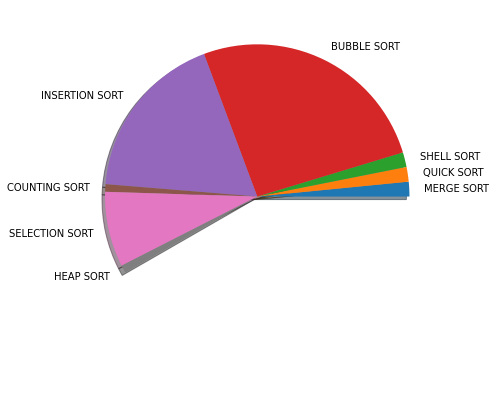
**BAR GRAPH**

****

**LINE GRAPH**

****

**PIE GRAPH**

****

**CONCLUSION & FUTURE**

There is no ideal sorting algorithm for every single case. The best algorithm to use varies from case to case. Consider the 3 most common

O(nlogn)comparison sorts.

**3 most common**

**O(nlogn)comparison sorts**

**Merge Sort**

Split your array in half. Recursively merge sort the left and right sub-arrays. Then, merge them together (linear time) to get the full sorted array.

Pros:

* Has O(nlog⁡n) worst-case run time.
* Of the 3 algorithms here, it is the only one that is stable, so if you want to retain the ordering of comparatively equivalent items, this should be your go-to.
* Easy to implement on linked list data structures. Does not require random access.

Cons:

* Has 0(n) space complexity, which is worse than the other 2 sorts.
* Slower than the other 2 algorithms in practice. Why? You have to write all your data into another array and back into your original one. Copying is usually slower than comparing.

**Heap Sort**

Generate a heap data structure on the array. Then, pop the top of the heap and move it to the end of the array. Repeat until the heap is empty and the entire array is sorted.

Pros:

* Has O(n logn) worst-case run time.
* Can sort in-place so uses
* O(1) additional space.

Cons:

* Unstable!
* Still much slower than Quick Sort on average. You still need to perform a swap on every element in the array, even if it’s already in the right place.

**Quick Sort**

Pick a pivot from the array and partition the array into sub-arrays. Everything in the left sub-array is less than the pivot. Everything in the right sub-array is greater than the pivot. Recursively sort the left and right sub-arrays.

Pros:

* Generally the fastest sorting algorithm in practice. It only swaps what needs to be swapped. Swaps are slow!
* Can be performed in-place, but in practice, the stack frames from recursive function calls results in
* O(log⁡n) space complexity.

Cons:

* Has O(n2) in the worst case. Ideally, the chosen pivot is the median. The further away it is from the median, the worse the performance.
* Like Heap Sort, it’s unstable!

**Which one’s the best?**

As you can see, each of the sorting algorithms have their pros and cons. Choose the sort that is best suited for your data. If you’re constrained in space, go for heap sort. If you need something stable, merge sort is your friend. For nearly sorted data, consider that insertion sort isO(n) time!

Modern sorting algorithms use hybrid sorts that combine the best qualities of the different basic sorting algorithms. Perhaps the most widely used (and arguably the “best”) is introsort (std::sort in C++) which runs quick sort and switches to heap sort when the recursion gets too deep. This way, you get the fast performance of quick sort in practice while guaranteeing a worst case O(nlog⁡n) run time.

**REFERENCES**

1. <https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/> For content purpose .
2. <https://www.quora.com/> For Queries .
3. <https://www.youtube.com/channel/UCeVMnSShP_Iviwkknt83cww> To learn python .
4. <https://www.youtube.com/playlist?list=PLu0W_9lII9agK8pojo23OHiNz3Jm6VQCH> To learn Matplotlib .
5. <https://www.google.com/> For content related help , gifs , images .
6. <https://www.python.org/> To install Python and read documentation