Sorting Algorithms and Measuring Time Complexity using Python

EEE 244-Numerical Analysis

Submitted to: Dr. B Preetham Kumar

Submitted by: Neha Gour

Date: May 11, 2023

Table of Contents

Sr. No.	Title	Page No.
1.	Introduction…………………………………………………………………………	3
2.	Popular Programming languages.………………………………………………	4
3.	Asymptotic Analysis……………………………………………………………….	5
4.	Project Description……………………………………………………………….	6-15
i.	Selection sort…………………………………………………………….	6
ii.	bubble sort……………………………………………………………….	8
iii.	insertion sort…………………………………………………………….	10
iv.	merge sort……………………………………………………………….	12
v.	quick sort……………………………………………………………….	15
5.	Experiment Results……………………………………………………………….	19
6.	Conclusion………………………………………………………………………..	20
7.	References………………………………………………………………………..	21

Introduction:

Nowadays, we observe emerging areas like machine learning, data science, and data analytics growing to support information technology and the growth of the IoT. Algorithms like sorting and searching play a crucial role to promote the other algorithms efficiently. The developing fields are using Python and Java for their implementation in the new fields. Additionally, it is observed that Python and Java are widely used for asymptotic analysis (time complexity and space complexity). In this paper, we focus on the analysis of Sorting algorithms as well as the runtime performance measures using Python.

The sorting algorithm organized the data items in sequence. The most frequently used ordering is numerical order and lexicographical order. To make the best use of another algorithm correct and efficient sorting is key. The challenges of effectively solving sorting problems have drawn a lot of research. Sorting algorithms are categorized into two classes based on their time complexity n^2 and n log n. In this project, we will measure the runtime of sorting algorithms using a time module to measure the running time of implementation from a pragmatic standpoint.

We can address a variety of issues, including searching, selection, duplication, and distribution, with the use of a sorting algorithm. There are various methods to use sorting to save time and effort, from commercial to academic and everything in between. Selection sort, bubble sort, insertion sort, merge sort, and quick sort-like methods are introduced. This set of sorting is known as Popular Sorting Algorithm (PSAs)[1].

In this, we are recognizing and providing the notion of which kind of problem, and which sorting algorithm can deliver the efficient time complexity, However, additionally, it depends on a number of criteria, the primary one being input datasets and hardware. In the next sections, we will focus on popular programming languages and analyze the different sorting algorithms as well as measure how long it actually takes for your sorting algorithms to run.

For the trials to analyze the algorithms we will use the predefined distinct data sets. The experiments produced intriguing results that allowed us to draw a satisfying conclusion regarding Python compiler performance and design.

Popular Programing languages:

A computer is an electronic component that executes a set of instructions (or a program) that have been provided to it after being first made to store them in its memory. When programming first began, we used binary codes, which we referred to as machine language (the language of computers), and it required highly qualified specialists. Assembly language was given its term because mnemonic symbolic codes were later added to make it easier to recall and identify the instructions. When computer programming grew popular due to its usefulness in science and technology, as well as for some well-known businesses, computer scientists built interpreters and compilers that translated programs that were written in human-readable form, making programming more user-friendly and effective. subsequently, high-level programming languages were introduced to the market. This made problem-solving using a computer appear novel in terms of its applications. [1].

Java and Python were both developed during the 4th Generation Era, which pushed the use of programming languages that were as close to human speech as achievable. With the help of fourth-generation languages, programmers with less programming experience will find programming more flexible and simpler. Python, as opposed to Java, more effectively met the goals of 4GL. Due to Java's unique multi-threading features, internet programming occupied a separate space. We refer to Python as Popular Programming Language in light of the information above (PPLs).

Characteristics of Python language:

1. Python has been simplified for new programmers.
2. Python requires less memory than Java since it has less code.
3. Python has become a growing scripting language.
4. Declaring a variable is unnecessary because the Python compiler has intelligence built in and is dynamically typed.
5. Complex functions made available in its extensive libraries are easily executable by programmers.
6. Python has become increasingly popular in recent years due to its use in applications like data science.[2]

Asymptotic Analysis:

The study of a given function f(n), as n varies in constrained limits, is known as asymptotic analysis. By looking at the amount of time and memory required to complete the task for which the algorithm was created, asymptotic bounds are used to calculate the algorithm's efficiency. We are focusing on time complexity in this project. Asymptotic analysis is carried out by using three separate factors. O(big-oh) is the upper bound, Ω(big-omega) is the lower bound, and

ϴ(theta) does from both upper and lower. The asymptotic analysis of any algorithm is possible on asymptotic input size [1].

Project Description:

i. Selection Sort:

This sorting algorithm works by repeatedly selecting the smallest item from the unsorted list and placing it at the start of the unsorted list.

Working:

1. Make the initial item the smallest possible [29,72,98,13,87,66,52,51,36].

2. Compared with the second item if the second item is less than the minimum value set to the second item. Contrast the minimum with the third item. Again, if the third item is smaller, assign a minimum to it; otherwise, do nothing. And so on till you reach the final part.

3. The smallest entry is moved to the top of the unsorted list after each iteration. Until all of the items are in their proper places.

Example:

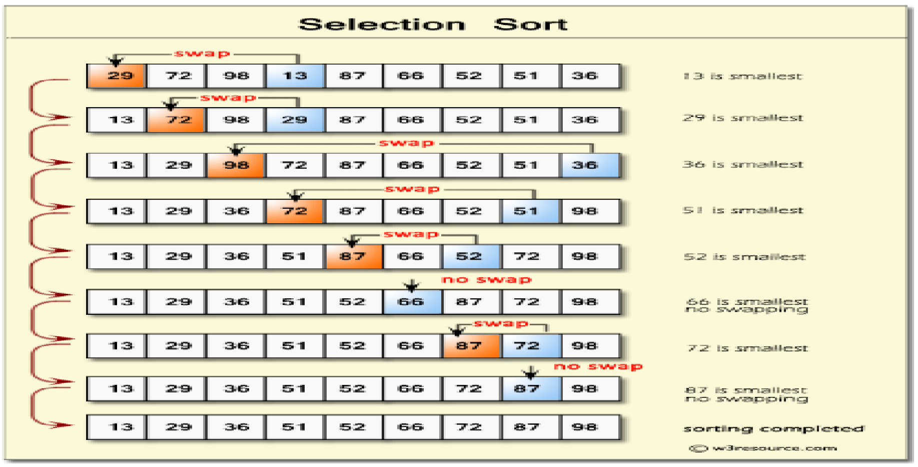


Figure1: Selection Sort Example in Pictorial Form [4]

Selection sort algorithm:

```
selectionSort(arrayList)
    repeat (len(arrayList)-1) times
    set first unsorted element as minimum
    for each of unsorted element
        if element < cuurentMinimumValue
            set element as new minimum value
    swap minimumValue with first unsorted position
end selectionSort
```

[3]

Selection Sort code in python

```
import time
def Selection_Sort(array_element):
    n = len(array_element)
    initial_time = time.perf_counter_ns()
    for i in range(0, n):
        Minimum_value = array_element[i]
        Minimum_index = i
        for j in range(i + 1, n):
            if array_element[j] < Minimum_value:
                Minimum_value = array_element[j]
                Minimum_index = j
        array_element[i], array_element[Minimum_index] = array_element[Minimum_index],
array_element[i]
    Final_time = time.perf_counter_ns()
    print(f"Execution time {Final_time - initial_time:0.2f} ns")
    return array_element

array_element = [20, 5, 6000, 8, 10]
Selection_Sort(array_element)
print(f"Sorted array is {array_element}")
```

Output:

```
Execution time 3700.00 ns
Sorted array is [5, 8, 10, 20, 6000]
```

Selection sort Time complexity:

Number of comparisons: (n-1) +(n-2) ……+1 = n(n-1)/2 this is equal to n^2. Time complexity is $O(n^2)$.

II. **Bubble Sort:**

Bubble sort compares two neighboring items and swaps them until they are in the right position.

Working:

1. Beginning with the first index, compare it to an adjacent item.

2. Swap if the first item is larger than the second; else, do nothing.

3. Next, compare the second and third items, and replace the larger one with a smaller one if the second one is larger.

4. Repeat until the last item is reached.

5. After each iteration, the largest item will be placed at the very right of the list among the unsorted items.

Example:

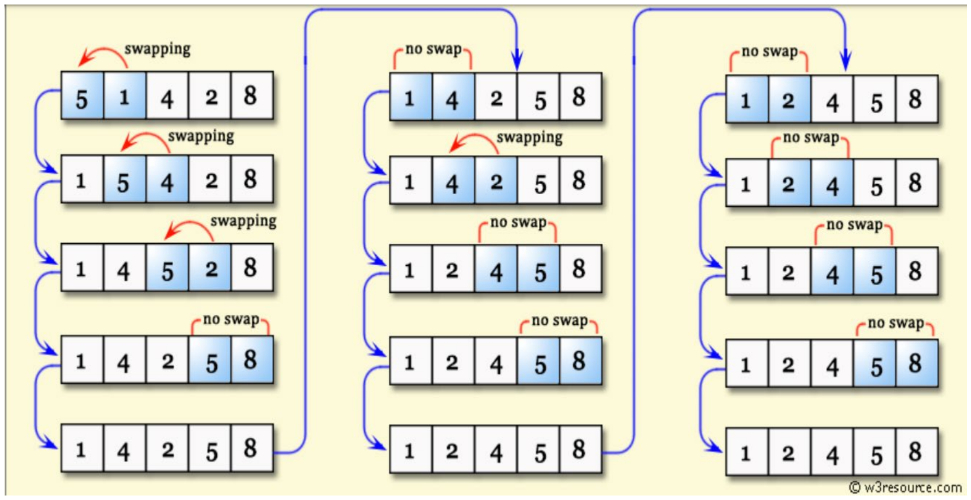


Figure 2: Bubble Sort Example in pictorial form [6]

Bubble sort algorithm:

```
bubbleSort(arrayList)
        for I <= to indexOfLastUnsortedElement -1
                if leftElement > rightElement
                        swap leftElement and rightElement
end bubbleSort
```

[5]

Bubble sort Code in Python:

```python
import time

def Bubble_sort(array_elements):
    n = len(array_elements)
    initial_time = time.perf_counter_ns()
    for i in range(0, n):
        for j in range(n - 1):
            if array_elements[j] > array_elements[j + 1]:
                array_elements[j], array_elements[j + 1] = array_elements[j + 1], array_elements[j]
    final_time = time.perf_counter_ns()
    print(f"Execution time {final_time - initial_time:0.2f} ns")
    return array_elements


array_elements = [20, 5, 6000, 8, 10]
Bubble_sort(array_elements)
print(f"Sorted array is {array_elements}")
```

Output:

```
Execution time 3900.00 ns
Sorted array is [5, 8, 10, 20, 6000]
```

Bubble Sort Time Complexity:

Number of comparisons: (n-1) + (n-2) ......+1 = n(n-1)/2 = n2; so, complexity is O(n2). Furthermore,

bubble sort requires two loops. As expected, the time complexity is n*n = n^2.

iii. Insertion Sort:

The insertion sort performs by looping over the sorted array one element at a time. While simple to use, it is not designed to handle massive amounts of data.

Working:

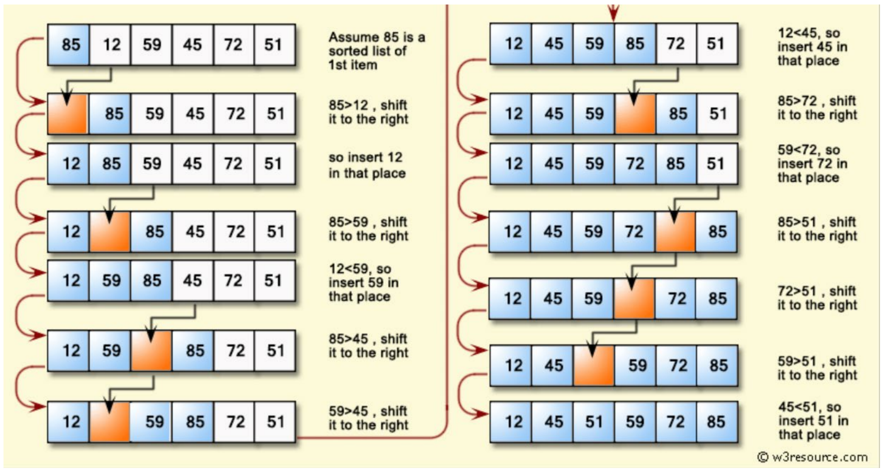
1. The array's first element is supposed to be sorted. Separately place the second component in a key.

2. Compare the key and the first element. If the first element is greater than it, the key is positioned in front of it. The first two elements have been sorted.

3. Compare the third element to the elements to its left. It was placed right behind the component that was smaller than it. If there is no element smaller than it, put it at the start of the array.

4. Arrange each element that was not sorted in the correct position.

Example:



Assume 85 is a sorted list of 1st item
85>12 , shift it to the right
so insert 12 in that place
85>59 , shift it to the right
12<59, so insert 59 in that place
85>45 , shift it to the right
59>45 , shift it to the right
12<45, so insert 45 in that place
85>72 , shift it to the right
59<72, so insert 72 in that place
85>51 , shift it to the right
72>51 , shift it to the right
59>51 , shift it to the right
45<51, so insert 51 in that place
© w3resource.com

Figure 3: Insertion Sort Example[7]

Insertion Sort algorithm:

```
insertionSort(arrayList)
    mark first element as sorted
    for each unsorted element X
        extract the element X
        for j from lastSortedIndex down to 0
            IF cuurentElement j>X
                Shift sorted element to the right by 1
    end InsertionSort
```

[8]

Insertion Sort code in Python:

```
import time


def Insertion_sort(array_elements):
   n = len(array_elements)
   initial_time = time.perf_counter_ns()
   for i in range(0, n):
      currentValue = array_elements[i]
      currentIndex = i
      while currentIndex > 0 and array_elements[currentIndex - 1] > currentValue:
         array_elements[currentIndex] = array_elements[currentIndex - 1]
         currentIndex = currentIndex - 1
         array_elements[currentIndex] = currentValue
   final_time = time.perf_counter_ns()
   print(f"Execution time {final_time - initial_time:0.2f} ns")
   return array_elements


array_elements = [20, 5, 6000, 8, 10]
Insertion_sort(array_elements)
print(f"Sorted array is {array_elements}")
```

```
OUTPUT
Execution time 3100.00 ns
Sorted array is [5, 8, 10, 20, 6000]
```

Insertion Sort Time Complexity:

Every element must be compared to every other element; thus, (n-1) comparisons take place for every nth element. As a consequence, the total number of comparisons is $n*(n-1) = O(n^2)$.

iv. Merge Sort:

One of the divides and conquer strategies is merge sort, in which we divide the problem into smaller problems and then solve them. The solutions from the subproblems are then combined to solve the main problem. This algorithm continually divides the array into two equal halves until we can begin merging.

Working:

1. To run Merge Sort on a subarray of size 1, or when leftHalf == rightHalf, the MergeSort function repeatedly splits the array in half.
2. Following that, the merge function is utilized to combine the smaller, sorted arrays into larger, merged arrays.
3. Up until the base case of an array with one element, the merge sort algorithm splits the array in half. Then, to gradually sort the entire array, the merge function selects the sorted sub-arrays and combines them.
4. Each and every recursive algorithm depends on a base case and the capacity to mix the outcomes of base cases. The same applies to merge sort. You probably guessed it, but the merging step is the most crucial stage in the merge sort algorithm
5. The merge step provides a straightforward solution to the issue of combining two sorted arrays into a single, massive sorted array.

6. The algorithm maintains three pointers: one for each of the two arrays and one for the most recently sorted array's most recent index.

Example:

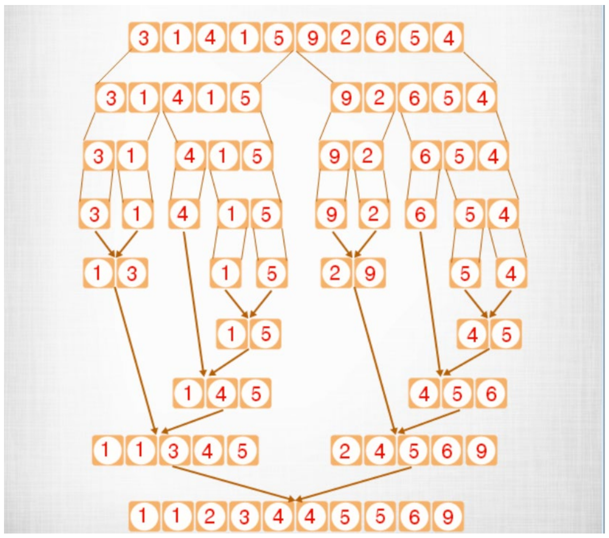


Figure 4 Example of Merge Sort [9]

Merge sort algorithm:

```
 MergeSort(arrayList)
 Declare mideElement, leftHalf,rightHalf
        if leftHalfElement > rightHalfElement
                Return
         MidElement = (leftHalfelement+rightElement)/2
         mergeSort(arrayList,lefthalf, midElement)
         mergeSort(arrayList,midElement+1, rightHalf)
        mergeSort(arrayList, leftHalf, midElement,rightHalf)
end mergeSort
```

Merge sort code in Python:

```
import time
initial_time = time.perf_counter_ns()
def Merge_sort(array_elements):
    if len(array_elements) > 1:
        median = int(len(array_elements)) / 2
        Left_half_elements = array_elements[:int(median)]   ## 0 to median-1
        Right_half_elements = array_elements[int(median):]    ## median to end element
        Merge_sort(Left_half_elements)
        Merge_sort(Right_half_elements)
        i = j = k = 0
        while i < len(Left_half_elements) and j < len(Right_half_elements):
            if Left_half_elements[i] <= Right_half_elements[j]:
                array_elements[k] = Left_half_elements[i]
                i = i+1
            else:
                array_elements[k] = Right_half_elements[j]
                j = j+1
            k =k+1
        while i < len(Left_half_elements):
            array_elements[k] = Left_half_elements[i]
            i = i+1
            k = k+1
        while j < len(Right_half_elements):
            array_elements[k] = Right_half_elements[j]
            j =j+1
            k =k+1

Final_time = time.perf_counter_ns()
print(f"Execution time {Final_time - initial_time:0.2f} ns")

array_elements = [20, 5, 6000, 8, 10]
Merge_sort(array_elements)
print(f"sorted array is {array_elements}")
```

OUTPUT:

```
Execution time 700.00 ns
sorted array is [5, 8, 10, 20, 6000]
```

Merge Sort Time Complexity:

A merge sort employs several repetitions of the input. The first pass merges segments of size 1, the second pass merges segments of size 2, and the ith pass merges segments of size 2i-1. In total, log2n passes have occurred. Because, as merge showed, we can merge sorted segments in linear time, each run takes O(n) time. Since there are log2n passes, the overall time complexity is O(nlogn).

v. Quick Sort:

Similarly, the Quicksort algorithm separates the input array into two lists, one with smaller items and the other with larger items, using the divide-and-conquer strategy. The algorithm then sorts both lists iteratively until the final, fully sorted list is obtained.

Working:

Pivot can be selected in below mentioned ways:

1. Element selected randomly.
2. Pivot Element can be the leftmost element of an array.
3. Pivot element can be the rightmost element of an array.
4. Pivot could be median.

It is good to have chosen a random element as a pivot, otherwise, there is probability to increase the time complexity.

1. The array holds a "pivot" item.

2. Set a pointer (the left pointer) to the very first element in the array.

3. The right pointer should start with the last element in the array.

4. Move the left pointer to the right while the leftmost value in the array is less than or equal to the pivot value (add 1). Repeat till the pivot value matches or surpasses the value of the left pointer.

5. Shift the array's right pointer to the left while the array's right pointer value is greater than the pivot value (subtract 1). Repeat until the pivot point's value is less than or equal to the value shown by the right pointer.

6. Swap the values in the array if the left pointer is less than or equal to the right pointer.

7. Move the right pointer one space to the left, and the left pointer one space to the right.

8. Return to step 1 if the left and right pointers do not meet.

Example:

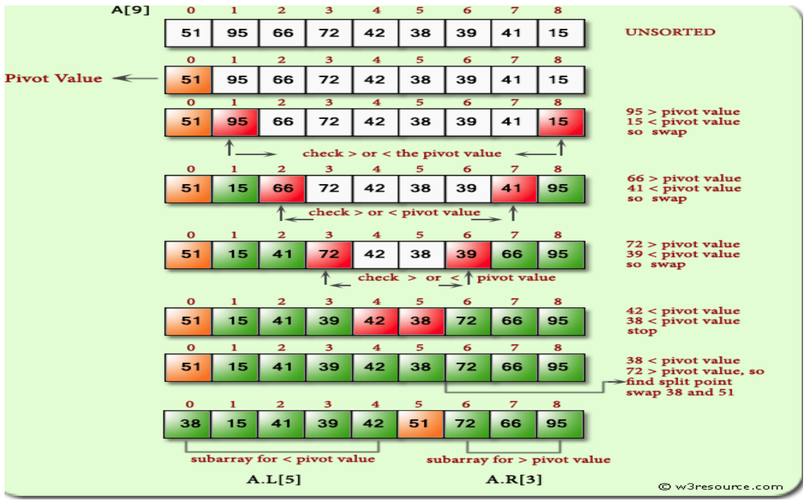


Figure 5: Merge Sort Example [10]

Quick sort Algorithm:

```
quickSort(arrayList, firstIndex,lastIndex)
        firstIndex < LastIndex
        pi <- partition(arrayList, firstIndex, lastIndex)
        And swap the pivot index with first index
        quickSort(arrayList, firstIndex, pi - 1)
        quickSort(arrayList, pi + 1, lastIndex)
partition (arrayList, firstIndex, lastIndex):
  for j in range (firstIndex + 1, lastIndex + 1):
        if element [j] <= pivotElement
            swap element [j] with element [pivot]
            pivotElement++
        swap pivot with element[lastindex]
    return pivot
```
[11]

Quick Sort code in Python:

```
import time
import random


def partition(array_elements, FIRST_index, LAST_index):
    PARTITION_elementvot = FIRST_index
    i = FIRST_index
    for j in range(FIRST_index + 1, LAST_index + 1):
        if array_elements[j] <= array_elements[PARTITION_elementvot]:
            (array_elements[i], array_elements[j]) = (array_elements[j], array_elements[i])
            i = i + 1
    (array_elements[PARTITION_elementvot], array_elements[i - 1]) = (
    array_elements[i - 1], array_elements[PARTITION_elementvot])
    PARTITION_elementvot = i - 1
    return PARTITION_elementvot


initial_time = time.perf_counter_ns()


def Quick_sort(array_elements, FIRST_index, LAST_index):
    n = len(array_elements)
    if FIRST_index < LAST_index:
        Random_PARTITION_elementvot = random.randrange(FIRST_index, LAST_index)
        array_elements[FIRST_index], array_elements[Random_PARTITION_elementvot] =
array_elements[
            Random_PARTITION_elementvot], array_elements[FIRST_index]
        PARTITION_element = partition(array_elements, FIRST_index, LAST_index)
        Quick_sort(array_elements, FIRST_index, PARTITION_element - 1)
        Quick_sort(array_elements, PARTITION_element + 1, LAST_index)


Final_time = time.perf_counter_ns()
print(f"Execution time {Final_time - initial_time:0.2f} ns")
```

```
array_elements = [20, 5, 6000, 8, 10]
Quick_sort(array_elements, 0, len(array_elements) - 1)
print(f"Sorted array is{array_elements}")

 OUTPUT
Execution time 300.00 ns
Sorted array is[5, 8, 10, 20, 6000]
```

Quick Sort Time Complexity:

Quicksort partitions the input list in linear time, O(n), then iteratively repeats this procedure an average of $\log_2 n$ times. As a result, the time complexity is $n\log_2 n$.

Having said that, keep in mind the discussion regarding how the algorithm's runtime is impacted by the pivot choice. When the chosen pivot is at the array's median, the best-case scenario is O(n), but an $O(n^2)$ scenario occurs when the pivot is the array's smallest or largest value.

The worst-case complexity should be O if the method concentrates first on locating the median value and then utilizes it as the pivot element n $\log_2$ n. Using the median of an array as the pivot ensures that the Quicksort portion of the code will run in O (1) time because finding the median of an array takes linear time n $\log_2 n$. You get a final runtime of O(n) + O by utilizing the median value as the pivot n $\log_2 n$. The logarithmic part increases significantly more quickly than the linear part, therefore you can reduce this to O (n $\log_2 n$)

Experiment Results:

ALGORITHM/EXECUTION TIME	SELECTION SORT	BUBBLE SORT	INSERTION SORT	MERGE SORT	QUICK SORT
TIME COMPLEXITY	O(n^2)	O(n^2)	O(n^2)	O ($n \log_2 n$)	O ($n \log_2 n$)
EXECUTION TIME	3.7e-6 seconds	3.9e-6 seconds	3.1e-6 seconds	7e-7 seconds	3e-7 seconds

Execution time of Sorting algorithm for Python (average case).

Conclusion:

The table above demonstrates how the sorting algorithms performed according to their asymptotic behaviors and order of growths stated in the theories and performance measurements. The sorting algorithm's execution time is compatible with theories. In other words, quick is the fastest sort, followed by merge sort, insertion sort, bubble sort, and selection sort.

References:

Paper Reference: [1] O. K. Durrani, A. S. Farooqi, A. G. Chinmai and K. S. Prasad, "Performances of Sorting Algorithms in Popular Programming Languages," 2022 International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON), Bangalore, India, 2022, pp. 1-7, doi: 10.1109/SMARTGENCON56628.2022.10084261.

Online Reference: [2] Top 11 features of Python You must know, link, 4/5/2023

Online Reference: [3] Selection Sort, selection-sort, 4/12/2023

Online Reference: [4] Example Figure of Selection Sort Figure 1, 4/13/2023

Online Reference: [5] Bubble Sort, bubble sort,4/13/2023

Online Reference: [6] Example Figure of Bubble Sort Figure 2, 4/13/2023

Online Reference: [7] Example Figure of Insertion Sort Figure 3, 4/16/2023

Online Reference: [8] Insertion Sort, insertion sort, 4/16/2023

Online Reference: [9] Example Figure of Merge Sort Figure 4, 4/20/2023

Online Reference: [10] Example Figure of Quick Sort, Figure 5, 4/26/2023

Online Reference: [11] Quick Sort, Quick Sort, 5/2/2023

The match percentage on this doc is **10%** from https://smallseotools.com/plagiarism-checker/