Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Engineering

Academic Year 2021-2022

Complexity Analysis of Quicksort

Advance Algorithm Laboratory

By

Meith Navlakha	**60004190068**
Nishant Aridaman Kumar	**60004190077**

Guide(s):

Prof. Sudhir Bagul

Assistant Professor

Department of Computer Engineering
Academic Year 2021-2022

Complexity Analysis of Quicksort

1. PROBLEM STATEMENT

Quick sort is a Divide Conquer algorithm and the fastest sorting algorithm. In quick sort, it creates two empty arrays to hold elements less than the pivot element and the element greater than the pivot element and then recursively sort the sub-arrays. Quicksort is a highly efficient sorting technique that divides a large data array into smaller subarrays, one containing values smaller than the pivot, on which the partition is based and the other contains values greater than the pivot value. Our problem statement is that we want to address to the worst case scenario of the quicksort algorithm to sort the array consisting of n elements in a pre-sorted order i.e either ascending or descending.

2. INTRODUCTION

a. Need:

Since sorting can often reduce the complexity of a problem, it is an important algorithm in Computer Science. These algorithms have direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, and many more.

Some of the most common sorting algorithms are:

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Counting Sort
- Radix Sort
- Bucket Sort

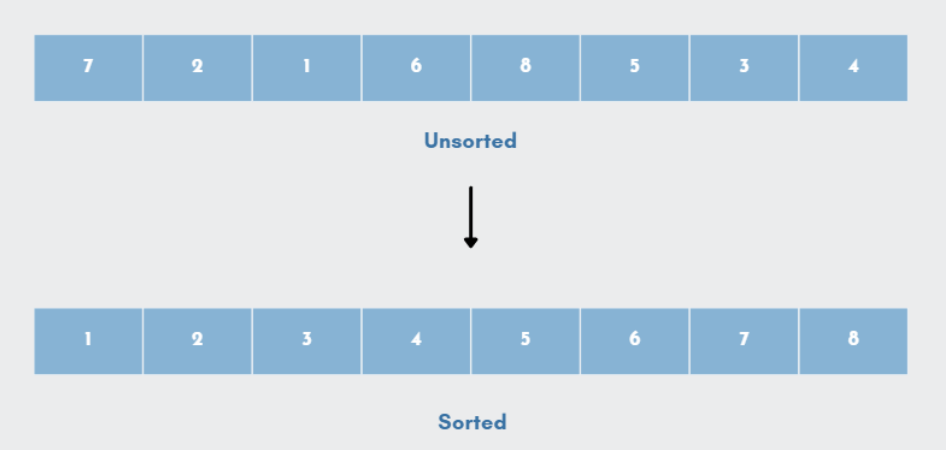
Classification of Sorting Algorithm

Sorting algorithms can be categorized based on the following parameters:

- Based on Number of Swaps or Inversion This is the number of times the algorithm swaps elements to sort the input. Selection Sort requires the minimum number of swaps.
- Based on Number of Comparisons This is the number of times the algorithm compares elements to sort the input. Using Big-O notation, the sorting algorithm examples listed above require at least O(nlogn) comparisons in the best case and O(n^2) comparisons in the worst case for most of the outputs.
- Based on Recursion or Non-Recursion Some sorting algorithms, such as Quick Sort, use recursive techniques to sort the input. Other sorting algorithms, such as Selection Sort or Insertion Sort, use non-recursive techniques. Finally, some sorting algorithm, such as Merge Sort, make use of both recursive as well as non-recursive techniques to sort the input.
- Based on Stability Sorting algorithms are said to be stable if the algorithm maintains the relative order of elements with equal keys. In other words, two equivalent elements remain in the same order in the sorted output as they were in the input.
- Insertion sort, Merge Sort, and Bubble Sort are stable
- Heap Sort and Quick Sort are not stable
- Based on Extra Space Requirement Sorting algorithms are said to be in place if they require a constant O(1) extra space for sorting.
- Insertion sort and Quick-sort are in place sort as we move the elements about the pivot and do not actually use a separate array which is NOT the case in merge sort where the size of the input must be allocated beforehand to store the output during the sort.
- Merge Sort is an example of out place sort as it require extra memory space for it’s operations.

Hence, sorting helps us improve the complexity of the algorithm. Quicksort being an efficient algorithm the task of our mini project is to analyze and thereby provide a way to reduce the complexity of Quicksort algorithm.

b. General Working of Quicksort:



1). Selecting Pivot

The process starts by selecting one element (known as the pivot) from the list; this can be any element. A pivot can be:

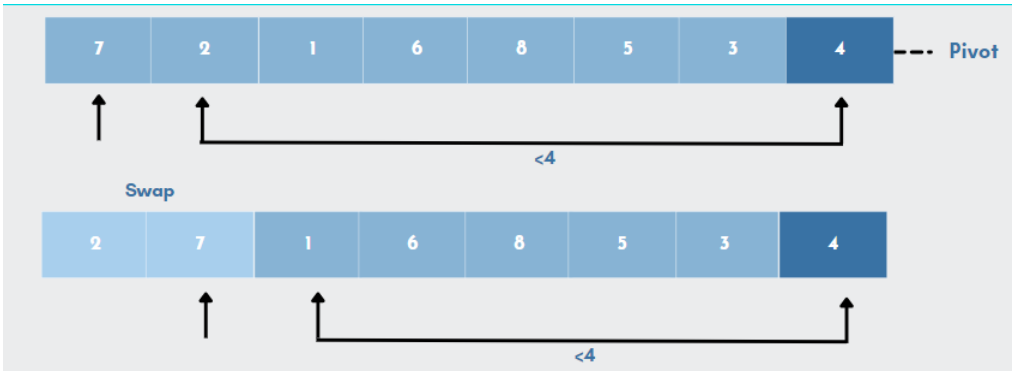
- Any element at random
- The first or last element
- Middle element

For this example, we'll use the last element, 4, as our pivot.

2). Rearranging the Array

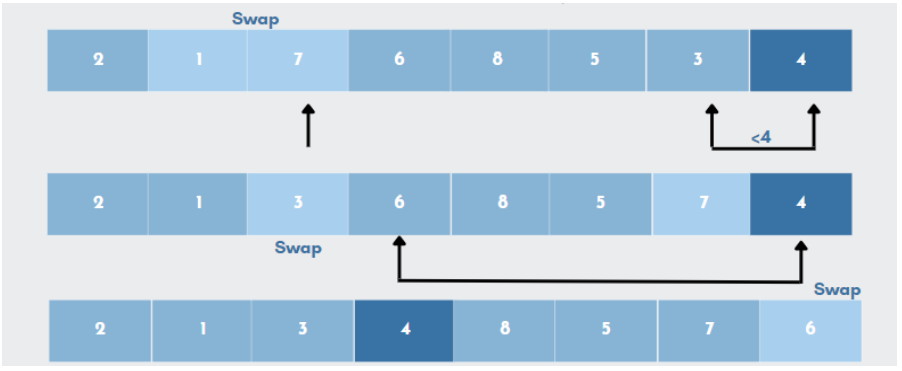
Now, the goal here is to rearrange the list such that all the elements less than the pivot are towards the left of it, and all the elements greater than the pivot are towards the right of it.

- The pivot element is compared to all of the items starting with the first index. If the element is greater than the pivot element, a second pointer is appended.
- When compared to other elements, if a smaller element than the pivot element is found, the smaller element is swapped with the larger element identified before.

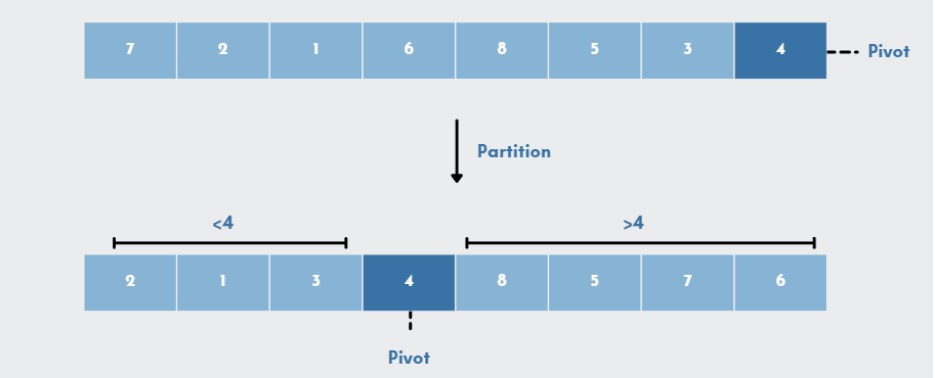


Let's simplify the above example,

- Every element, starting with 7, will be compared to the pivot(4). A second pointer will be placed at 7 because 7 is bigger than 4.
- The next element, element 2 will now be compared to the pivot. As 2 is less than 4, it will be replaced by the bigger figure 7 which was found earlier.
- The numbers 7 and 2 are swapped. Now, pivot will be compared to the next element, 1 which is smaller than 4.
- So once again, 7 will be swapped with 1.
- The procedure continues until the next-to-last element is reached, and at the end the pivot element is then replaced with the second pointer. Here, number 4(pivot) will be replaced with number 6.



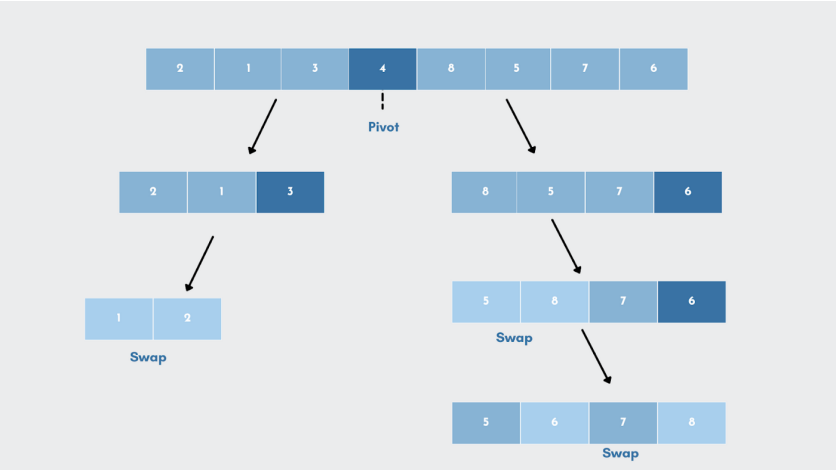
As elements 2, 1, and 3 are less than 4, they are on the pivot's left side. Elements can be in any order: '1','2','3', or '3','1','2', or '2','3','1'. The only requirement is that all of the elements must be less than the pivot. Similarly, on the right side, regardless of their sequence, all components should be greater than the pivot.



In simple words,the algorithm searches for every value that is smaller than the pivot. Values smaller than pivot will be placed on the left, while values larger than pivot will be placed on the right. Once the values are rearranged, it will set the pivot in its sorted position.

3). Dividing Subarrays

Once we have partitioned the array, we can break this problem into two sub-problems. First, sort the segment of the array to the left of the pivot, and then sort the segment of the array to the right of the pivot.



- In the same way that we rearranged elements in step 2, we will pick a pivot element for each of the left and right sub-parts individually.
- Now, we will rearrange the sub-list such that all the elements are less than the pivot point, which is towards the left. For example, element 3 is the largest among the three elements, which satisfies the condition, hence element 3 is in its sorted position.

- In a similar manner, we will again work on the sub-list and sort the elements 2 and 1. We will stop the process when we get a single element at the end.
- Repeat the same process for the right-side sub-list. The subarrays are subdivided until each subarray consists of only one element.
- Now At this point, the array is sorted.

c. Applications:

- Commercial Computing is used in various government and private organisations for the purpose of sorting various data like sorting files by name/date/price, sorting of students by their roll no., sorting of account profile by given id, etc.
- The sorting algorithm is used for information searching and as Quicksort is the fastest algorithm so it is widely used as a better way of searching.
- It is used everywhere where a stable sort is not needed.
- Quicksort is a cache-friendly algorithm as it has a good locality of reference when used for arrays.
- It is tail -recursive and hence all the call optimization can be done.
- It is an in-place sort that does not require any extra storage memory.
- It is used in operational research and event-driven simulation.
- Numerical computations and in scientific research, for accuracy in calculations most of the efficiently developed algorithm uses priority queue and quick sort is used for sorting.
- Variants of Quicksort are used to separate the Kth smallest or largest elements.
- It is used to implement primitive type methods.
- If data is sorted then the search for information became easy and efficient.

3. ALGORITHMS USED

a. Theory

i. Lumoto :

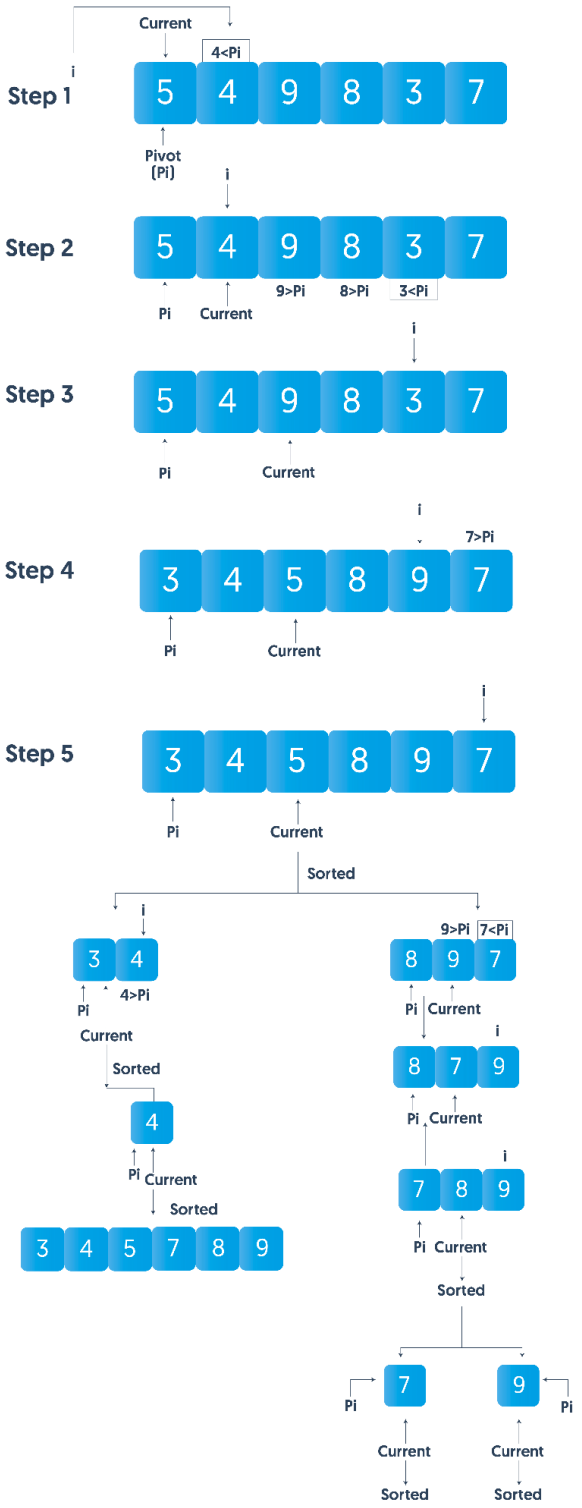
Lomuto Partitioning is attributed to Nico Lomuto. This works by iterating over the input array, swapping elements that are strictly less than a pre-selected pivot element. They appear earlier in the array, but on a sliding target index. This sliding target index is then the new partition index that we'll return for the next recursions of the greater algorithm to work with.

This is to ensure that our sliding target index is in a position such that all elements before it in the array are less than this element and that this element is less than all elements after it in the array.

Algorithm:

```
partition(arr[], lo, hi)
   pivot = arr[hi]
   i = lo     // place for swapping
   for j := lo to hi – 1 do
      if arr[j] <= pivot then
         i = i + 1
         swap arr[i] with arr[j]
   swap arr[i] with arr[hi]
   return i
```

Example:


Step 1
Step 2
Step 3
Step 4
Step 5
Current
4<Pi
Pivot (Pi)
9>Pi
8>Pi
3<Pi
7>Pi
Sorted
4>Pi
7<Pi
Pi
Current
i

ii. Hoare's Partitioning

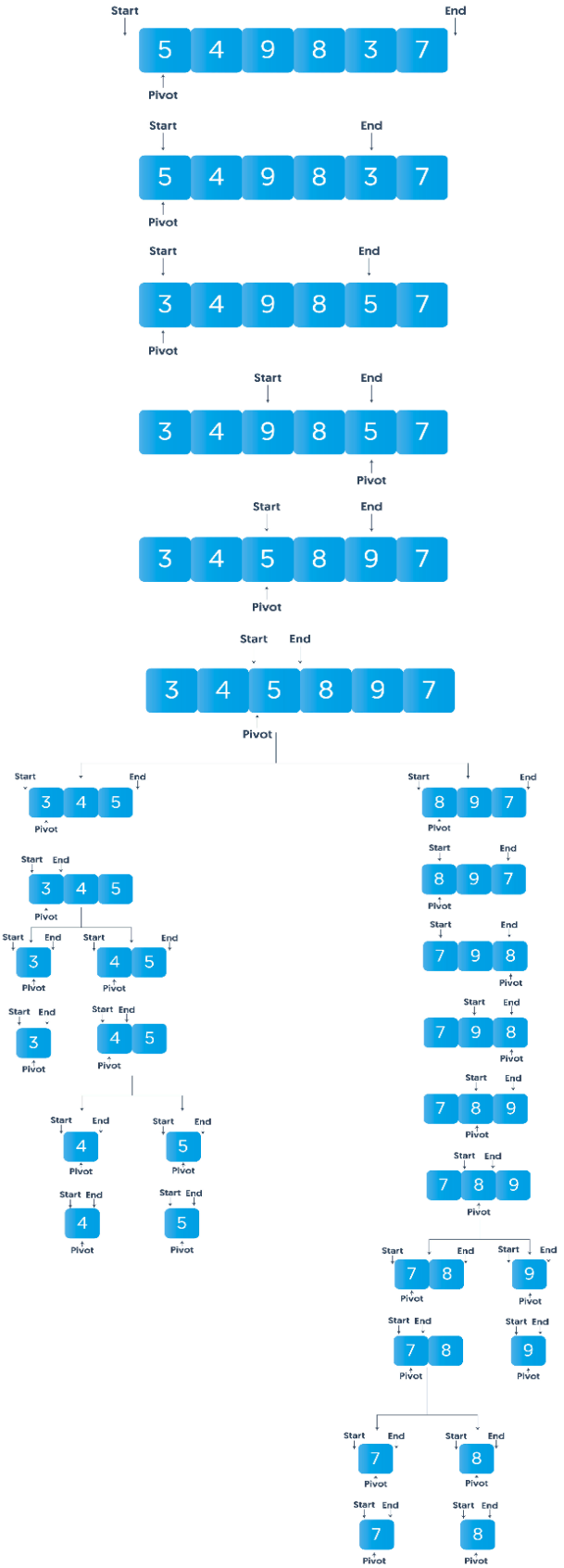
Tony Hoare invented quicksort and also developed the Hoare's partitioning algorithm. Hoare's partition involves having a pivot element and two indexes: one index at the start of the array and the other at its end. A pivot element can be anything in theory, but we typically use the first element as the pivot for Hoare's Partition Algorithm.

Ideally, we want a pivot that is the middle value, so the array is evenly partitioned to ensure optimal efficiency. Once the start index moves past and becomes greater than the end index, the values at the end index and the pivot element are swapped. The pivot element is now at the end index, which also represents its rightful place. (When we swap the value at endIndex with pivot element, we are storing the pivot element at the location endIndex points to.) This process continues till the sorting is complete.

Algorithm:

partition(arr[], lo, hi)

```
partition(arr[], lo, hi)
pivot = arr[lo]
i = lo - 1  // Initialize left index
j = hi + 1  // Initialize right index

// Find a value in left side greater
// than pivot
do
  i = i + 1
while arr[i] < pivot

// Find a value in right side smaller
// than pivot
do
  j--;
while (arr[j] > pivot);

if i >= j then
  return j
```


Start
End
5 4 9 8 3 7
Pivot
Start
End
5 4 9 8 3 7
Pivot
Start
End
3 4 9 8 5 7
Pivot
Start
End
3 4 9 8 5 7
Pivot
Start
End
3 4 5 8 9 7
Pivot
Start
End
3 4 5 8 9 7
Pivot
Start
End
3 4 5
Pivot
Start
End
3 4 5
Pivot
Start
End
3
Pivot
Start
End
4 5
Pivot
Start
End
3
Pivot
Start
End
4 5
Pivot
Start
End
4
Pivot
Start
End
5
Pivot
Start
End
4
Pivot
Start
End
5
Pivot
Start
End
8 9 7
Pivot
Start
End
8 9 7
Pivot
Start
End
7 9 8
Pivot
Start
End
7 9 8
Pivot
Start
End
7 8 9
Pivot
Start
End
7 8 9
Pivot
Start
End
7 8
Pivot
Start
End
9
Pivot
Start
End
7 8
Pivot
Start
End
9
Pivot
Start
End
7
Pivot
Start
End
8
Pivot
Start
End
7
Pivot
Start
End
8
Pivot

iii. Median of 3 Partition:

Quicksort is slowest when the pivot is always the smallest or largest possible value. The best possible pivot is the median of the segment b[h..k] being sorted. In our algorithm we propose the median of three values from the subarray –

arr[high], arr[(high+low)/2], and b[low]

The median is selected as the pivot and to the right, we have values greater than the pivot and to the left we have values lesser than the pivot.

Algorithm:

```
public static void qSort(int[] b, int h, int k) {
int h1= h;
int k1= k;
// invariant P: if b[h1..k1] is sorted, then so is b[h..k] while (h2 – k1 <= 9) {
medianOf3(b, h1, k1);
int j= partition(b, h1, k1);
// b[h1..j-1] ≤ b[j] ≤ b[j+1..k1]
if (j-h1 < k1-j) { // left is partition smaller
qSort(b, h1, j-1);
h1= j+1;
} else { // right partition is smaller
qSort(b, j+1, k);
k1= j-1; }
}
// post: P and b[h1..k1] has at most 10 elements insertionsort(b, h1, k1);
}
```

b. Applications

The Quick sort partition scheme can be used for various purposes including sorting the array, moving all even or odd elements to the front of the array, etc.

Lumoto is used when performance is not important and the application requires simpler and easier code. Hoare's is used when application performance is paramount and implementation is not an issue. Median of 3, can be used in both the cases.

c. Complexity Analysis

Using the Recurrence Relation – $T(n)$, the time needed to sort a list of size n.

i. In the most balanced case, a single quicksort call involves $O(n)$ work plus two recursive calls on lists of size $n/2$, so the recurrence relation is:

$$T(n) = O(n) + 2T(n/2)$$

By master theorem for divide-and-conquer recurrences, the time complexity:

$$T(n) = O(n \log n)$$

ii. In the most unbalanced case, a single quicksort call involves $O(n)$ work plus two recursive calls on lists of size 0 and $n-1$, so the recurrence relation is

$$T(n) = O(n) + T(0) + T(n-1)$$

This is the same relation as for insertion sort and selection sort, and it solves to worst case :

$$T(n) = O(n^2).$$

However, incase of Median of 3 partitioning, the worst case is prevented as we will always have atleast 1 element in either of the sub-array(L or H). Also, randomly selecting the 3 elements from the subarray also minimises the chances of only having 1 element in either of the array, thereby, ensuring a better split than the other 2 algorithms.

$T(n) = O(n \log n)$ …. for all the cases

4. IMPLEMENTATION

a. Screenshots of Code

1. Code for Partition

```
def Lumotopartition(array, lo, hi):
    pivot = array[hi]

    i = lo - 1
    for j in range(lo, hi):
        if array[j] < pivot:
            i += 1
            temp = array[i]
            array[i] = array[j]
            array[j] = temp
    temp = array[i + 1]
    array[i + 1] = array[hi]
    array[hi] = temp
    return i + 1


def LumotoQuicksort(array, lo, hi):
    if lo < hi:
        p = Lumotopartition(array, lo, hi)
        LumotoQuicksort(array, lo, p - 1)
        LumotoQuicksort(array, p + 1, hi)
    return array
```

```
def HoarePartition(array, low, high) -> int:
    pivot = array[low]
    (i, j) = (low - 1, high + 1)

    while True:

        while True:
            i = i + 1
            if array[i] >= pivot:
                break

        while True:
            j = j - 1
            if array[j] <= pivot:
                break

        if i >= j:
            return j

        swap(array, i, j)


def HoareQuicksort(array, low, high):
    if low < high:
        pivot = HoarePartition(array, low, high)
        HoareQuicksort(array, low, pivot)
        HoareQuicksort(array, pivot + 1, high)
    return array
```

```
def medianThreePartition(array, low , high) -> int:
    if high-low+1>2:
        index = random.randint(low+1, high-1)
    else:
        return low

    # MEDIAN INDEX
    if array[low]>=array[high] and array[low]>=array[index]:
        if array[index]>array[high]:
            return high
        else:
            return index

    elif array[index]>=array[high] and array[index]>=array[low]:
        if array[low]>array[high]:
            return high
        else:
            return low

    elif array[high]>=array[index] and array[high]>=array[low]:
        if array[index]>array[high]:
            return low
        else:
            return index


def medianQuicksort(array, lo, hi):
    if lo < hi:
        p = medianThreePartition(array, lo, hi)
        medianQuicksort(array, lo, p - 1)
        medianQuicksort(array, p + 1, hi)
    return array
```

```
def quicksort():

    timeArr=[]
    NumElem = [200 , 300, 500, 700 , 800]

    for N in NumElem:
      row=[]
      arr = generateNum(N)
      # RANDOMISE
      # arr = randomiseArr(arr)

      print("\nNO. OF ELEMENTS : ", len(arr))
      row.append(N)

      startHoare = time.perf_counter()
      hoare_arr = HoareQuicksort(arr, 0, len(arr) - 1)
      HoareTime= round(endHoare - startHoare , 9)
      print(f"TIME ELAPSED FOR HOARE : {HoareTime} seconds")
      row.append(HoareTime)


      startLumoto = time.perf_counter()
      lumotto_arr  =LumotoQuicksort(arr, 0, len(arr) - 1)
      endLumoto = time.perf_counter()
      LumottoTime= round(endLumoto - startLumoto , 9)
      print(f"TIME ELAPSED FOR LUMOTO : {LumottoTime} seconds")
      row.append(LumottoTime)


      startMedian = time.perf_counter()
      median_arr = medianQuicksort(arr, 0, len(arr) - 1)
      endMedian = time.perf_counter()
      MedianTime= round(endMedian - startMedian , 9)
      print(f"TIME ELAPSED FOR MEDAIN : {MedianTime} seconds")
      row.append(MedianTime)

      timeArr.append(row)

    return pd.DataFrame(timeArr , columns=['N', 'HOARE' , 'LUMOTO', 'MEDAIN'])
```

2. Time Distribution for Randomised Sequence

```
timePlot1 = quicksort()
timePlot1 = timePlot1.set_index('N')
```

3.Time Distribution for Randomised Sequence

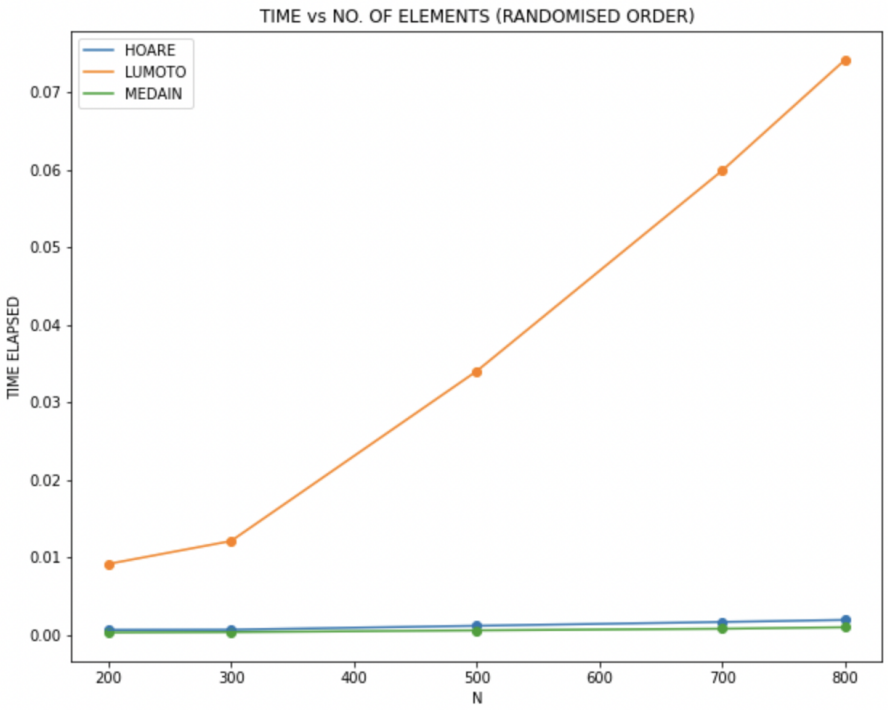
```
timePlot2 = quicksort()
timePlot2 = timePlot2.set_index('N')
```

b. Code of important functions

1. Time Elapsed for Randomised Elements

```
NO. OF ELEMENTS :  200
TIME ELAPSED FOR HOARE : 0.00060935 seconds
TIME ELAPSED FOR LUMOTO : 0.00911293 seconds
TIME ELAPSED FOR MEDAIN : 0.00027639 seconds

NO. OF ELEMENTS :  300
TIME ELAPSED FOR HOARE : 0.000634405 seconds
TIME ELAPSED FOR LUMOTO : 0.012073841 seconds
TIME ELAPSED FOR MEDAIN : 0.000326351 seconds

NO. OF ELEMENTS :  500
TIME ELAPSED FOR HOARE : 0.001138008 seconds
TIME ELAPSED FOR LUMOTO : 0.033997594 seconds
TIME ELAPSED FOR MEDAIN : 0.000546266 seconds

NO. OF ELEMENTS :  700
TIME ELAPSED FOR HOARE : 0.001629389 seconds
TIME ELAPSED FOR LUMOTO : 0.059895528 seconds
TIME ELAPSED FOR MEDAIN : 0.000760478 seconds

NO. OF ELEMENTS :  800
TIME ELAPSED FOR HOARE : 0.001893099 seconds
TIME ELAPSED FOR LUMOTO : 0.074184879 seconds
TIME ELAPSED FOR MEDAIN : 0.000941591 seconds
```

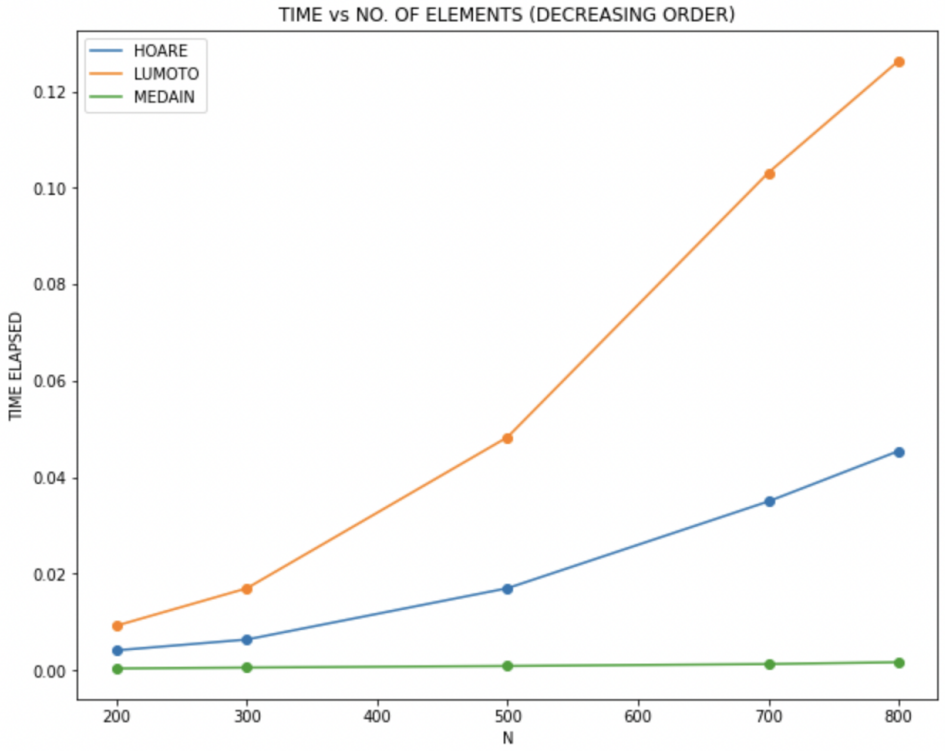
N	200	300	500	700	800
HOARE	0.000609	0.000634	0.001138	0.001629	0.001893
LUMOTO	0.009113	0.012074	0.033998	0.059896	0.074185
MEDAIN	0.000276	0.000326	0.000546	0.000760	0.000942



2. Time Elapsed for Descending Elements

```
NO. OF ELEMENTS :  200
TIME ELAPSED FOR HOARE : 0.004140106 seconds
TIME ELAPSED FOR LUMOTO : 0.00925487 seconds
TIME ELAPSED FOR MEDAIN : 0.000387104 seconds

NO. OF ELEMENTS :  300
TIME ELAPSED FOR HOARE : 0.006386151 seconds
TIME ELAPSED FOR LUMOTO : 0.017007991 seconds
TIME ELAPSED FOR MEDAIN : 0.000570542 seconds

NO. OF ELEMENTS :  500
TIME ELAPSED FOR HOARE : 0.017030715 seconds
TIME ELAPSED FOR LUMOTO : 0.048357717 seconds
TIME ELAPSED FOR MEDAIN : 0.000894418 seconds

NO. OF ELEMENTS :  700
TIME ELAPSED FOR HOARE : 0.035016679 seconds
TIME ELAPSED FOR LUMOTO : 0.103165692 seconds
TIME ELAPSED FOR MEDAIN : 0.001299763 seconds

NO. OF ELEMENTS :  800
TIME ELAPSED FOR HOARE : 0.045481358 seconds
TIME ELAPSED FOR LUMOTO : 0.126377967 seconds
TIME ELAPSED FOR MEDAIN : 0.001676744 seconds
```

N	200	300	500	700	800
HOARE	0.004140	0.006386	0.017031	0.035017	0.045481
LUMOTO	0.009255	0.017008	0.048358	0.103166	0.126378
MEDAIN	0.000387	0.000571	0.000894	0.001300	0.001677



5. COMPLEXTY ANALYSIS

Comparision between Lumoto, Hoare's and Median of 3 elements

Basis	Lumoto	Hoare's	Median of 3
No. of swaps and partitioning	Three times more swaps compared to Hoare's	Lesser swaps as compared to Lumoto	More swaps than Hoare's, but lesser than Lumoto
Time complexity when all elements are equal	$O(n^2)$	O(nlogn)	O(nlogn)
Time complexity when all elements are pre-sorted	$O(n^2)$	$O(n^2)$	O(nlogn)
Time complexity when all elements randomised	O(nlogn)	O(nlogn)	O(nlogn)
Implementation	Easier to imlement	Comparetively Harder to implement	Easier to implement
Efficiency	Less Efficient	Comparatively More efficient	Most efficient

The time elapsed for Lumoto, Hoare's and Medain of 3 elements partitioning methods for both randomised and pre-sorted sequences have been elicitated in the table below:

No. of Elements		Hoare's	Lumoto	Median of 3
Randomised Order	200	0.000609	0.009113	0.000276
	500	0.001138	0.033998	0.000546
	800	0.001893	0.074185	0.000942
Pre-Sorted Order	200	0.004140	0.009255	0.000387
	500	0.017031	0.048358	0.000894
	800	0.045481	0.126378	0.001677

6. CONCLUSION

Hence, median of 3 partitioning methods can be chosen for selecting the pivotal element in the quick sort as it helps to limit the worst-case of the problem. It is evident from the graphs in the previous sections that, Quicksort using Median of 3, gives a time complexity of O(n logn) even in the worst case where as Quicksort using Lumoto or Hoare's partition runs in $O(n^2)$ time in for the worst-case inputs. When the elements are in random order the Quicksort using Hoare's partition, Lumoto partition and Quicksort using Median of 3 all show a similar behavior giving a time complexity of O(n log n).

Although Haore's and Medain of 3 have marginal difference in time, Lumoto on the other hand shows quite a long time to perform the task which increases significantly with the number of elements owing to the more swaps required by lumoto compared to the other 2 methods. Hence, we have successfully determined that the selection of pivot as median helps us increase the efficiency of the quicksort algorithm. The algorithm