**Quick Sort:**

In Quick Sort is a divide and conquer sorting algorithm and it sorts “in place” where sorting doesn’t require extra space for sorting.

**Divide:**

Divides the given array into 2 subarrays around the pivot element x ,

(Elements in Lower SubArray)<= X >= (Elements in Upper SubArray)

|  |  |  |
| --- | --- | --- |
| **Elements in Lower SubArray** | **X** | **Elements in Upper SubArray** |

**Conquer:**

Recursively sorts 2 subArray elements

By tuning the pivot value we can reduce the expected time for sorting. Based on choosing the pivot, there are three flavors of Quick Sort Algorithms which will be discussed below

1. Basic Quick Sort
2. Randomized Quick Sort
3. Median-3-Partition Quick Sort

In the below analysis of three version of quick sort I am considering the number of swap operation along with the comparison operation since swapping operation is costly operation.

**Basic Quick Sort:**

**Pseudo Code:**

PARTITION(A, p, r)

X= A[r]

i =p -1

for j = p to r – 1{

if A[j] <= X {

i =i + 1

exchange A[i]\_ with A[j]

}

}

exchange A[i+1] with A[r]

return i+1

QUICKSORT(A, p, r)

If p < r{

q = PARTITION(A, p, r)

QUICKSORT(A, p, q-1)

QUICKSORT(A, q+1, r)

}

**Worst Case Scenario:**

For Basic Quick Sort the worst case input is if the inputs are already sorted or reverse sorted.

In this case the one side of the partition has no elements,

Therefore expected time for partitioning is T(n) = T(0)+ T(n-1)+O(n);

Where T(0) is expected time to sort 0 elements and O(n) is time taken for the portioning and T(n-1) is the expected time for sorting T(n-1) elements

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

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

=O(n^2)

In this case the expected time is similar to the insertion sort.

Even in the Output of the written program we can notice that number of comparison made is more near to the n^2 ( n is 1000 for the given Program) for the sorted and reverse Sorted Array.

**Recursion tree:**

C(n)

|

-------

| |

T(0) C(n-1)

|

------

| |

T(0) C(n-2)

|

--------

| |

T(0) C(n-3)

The Height of the Recursion Tree for the Basic Quick sort in worst case in **“n**” and it is highly unbalanced one.

**Best Case Scenario (Intuition):**

In this case the both side of the partition are almost exactly in half.

Therefore expected time for partitioning is T(n) = 2T(n/2)+O(n);

Where T(n/2) is expected run time to sort n/2 elements and O(n) is time taken for the partitioning array of size n

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

Based on the Case II of the master theorem we get

T(n)= O(nlogn)

**Recursion Tree:**

C(n)

|

--------------------

| |

C(1/10) C(9/10)

| |

--------- -----------

| | | |

C(1/100) C(9/100) C(9/100) C(81/100)

From the above it is derived the height of the tree will be O(nlogn) after adding all the leaf node with O(1) node.

**Programming Output:**

Worst Case scenario are highlighted below for the input array size of 1000

-bash-4.2$ ./BasicQuickSort.out

Avg Number OF Comparison required for Basic Quick Sort on Randomized Array = 450652

Avg Number OF Swapping required for Basic Quick Sort on Randomized Array = 451125

Avg Number OF Comparison required for Basic Quick Sort on Sorted Array = 499500

Avg Number OF Swapping required for Basic Quick Sort on Sorted Array = 500499

Avg Number OF Comparison required for Basic Quick Sort on reverse Sorted Array = 483875

Avg Number OF Swapping required for Basic Quick Sort on reverse Sorted Array = 469145

**Randomized Partitioning Quick Sort:**

In this sorting algorithm the pivot value is randomly chosen each time with in the given range of the array indices. In this the run time is independent of the given input in which the previous case(Basic Quick) run time is dependent on the order of the input. There is no specific input that can elicit the worst-case behavior. The worst case is determined only by the random number generator.

RANDOMIZED-PARTITION(A, p, r)

i = RANDOM(p r)

exchange A[r] with A[i]

return PARTITION(A, p, r)

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

RANDOMIZED-QUICKSORT(A, p, r)

if p < r

q = RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, q-1)

RANDOMIZED-QUICKSORT(A, q+1, r)

**Analysis:**

T(n) = random Variable for running time assuming random variables are generated independently.

For K =0,1,….n-1 let

1 if Partition generates as k:n-k-1 split

Xk = or 0 otherwise

Xk is called as indicator random Variable. The expectation of Xk is

E[Xk] = 0 Probabliity[Xk=0] + 1 Probabliity[Xk=1]

= 1 Probabliity[Xk=1]

= 1/n

Recurrence for T(n) is as follows

T(0)+T(n-1)+O(n) if 0:n-1 split

T(n) = T(1)+T(n-2)+O(n) if1:n-2 split

.

.

T(n-1)+T(0)+O(n) if n-1:0 split

T(n) = Sum Of (Xk(T(k)+T(n-k-1)+O(n)) where k 0 to n-1;

E[T(n)] = Sum Of (E(Xk)\* E[(T(k)+T(n-k-1)+O(n))] where k 0 to n-1;

= 2/n \* Sum Of(E[T(k)]) + O(n) where k 0 to n-1

Considering k =0 and 1 taken into O(n)

We have

E[T(n)] = 2/n \* Sum Of(E[T(k)]) + O(n) where k 2 to n-1

**Programing Output Analysis:**

In the below output we can notice that number of comparisons is almost same for the different input array is given which is nearly equal to O(nlogn)

-bash-4.2$ ./RandomizedQuickSort.out

Avg Number OF Comparison required for Randomized Quick Sort on Randomized Array = 10958

Avg Number OF Swapping required for Randomized Quick Sort on Randomized Array = 7006

Avg Number OF Comparison required for Randomized Quick Sort on Sorted Array = 10870

Avg Number OF Swapping required for Randomized Quick Sort on Sorted Array = 7176

Avg Number OF Comparison required for Randomized Quick Sort on reverse Sorted Array = 10875

Avg Number OF Swapping required for Randomized Quick Sort on reverse Sorted Array = 6837

**Median-3-Partiton:**

In the Median of 3 Partition Quick Sort we take 3 random Values from the given range of array and find the median of it. This median is used as pivot element. By choosing the median value as a pivot element the memory required for piling up the stack is getting minimized as tree is getting more balanced.

We will split the array based on the pivot element and sort them.

If we have good Pivot Value the expected time is ½T(3/4n)

If we have bad pivot Value the expected time is 1/2T(n-1)

The overall Expected time is

T(n) = ½T(3/4n)+ 1/2T(n-1)+O(n)

T(n-1) is nearly equal to T(n) therefore

1/2T(n) = ½T(3/4n)+O(n)

T(n) = T(3/4n)+O(n)

Now taking a = 1 and b =4/3 and d=1

(b^d) > a which follows Case 1 of the Master Theorem

According to the master theorem case 1 running time is O(n^d) and here d = 1 and the running time is O(n). So in the Median-3-Partiton Quick sort expected running time is O(n) which is linear.

E[T(n)] = O(n)

**Below is the analysis of sorting 1000 numbers using Medain 3 Partition Quick Sort:**

-bash-4.2$ ./MedianQuickSort.out

Avg Number OF Comparison required for Median Quick Sort on Randomized Array = 8875

Avg Number OF Swapping required for Median Quick Sort on Randomized Array = 5500

Avg Number OF Comparison required for Median Quick Sort on Sorted Array = 8759

Avg Number OF Swapping required for Median Quick Sort on Sorted Array = 5442

Avg Number OF Comparison required for Median Quick Sort on reverse Sorted Array = 9217

Avg Number OF Swapping required for Median Quick Sort on reverse Sorted Array = 5812

In the above output we can notice that **Medain 3 Partition Quick Sort** does to maximum of 9217 comparison and maximum of 5812 swapping operations for sorting 1000 numbers. So in the worst case Input like reverse sorted array the number of comparison is nearly equal to number of input elements.

**Overall Analysis from the 3 Version of Quick Sort:**

From Overall Analysis it is best if we use the Median 3 Partition Quick sort since the expected run time is O(n), this can be seen by the average number of comparison and swapping done in each version of Quick sort algorithm.

Reference:

1. Book Introduction To Algorithm Cormen 3rd Edition
2. <https://cs.nyu.edu/courses/spring99/V22.0102-002/quick.html>
3. <https://www.youtube.com/watch?v=rixanPJwWNU>
4. <https://www.youtube.com/watch?v=3DV8GO9g7B4>
5. https://www.youtube.com/watch?v=aUY7zHLU4L8