QUICK SORT

- developed by C.A.R Hoare in 1962, have an average run time of $O(n \log_2 n)$ and treated as a Better Sorting method.

- From the previous sorting method studied so far, we can draw a conclusion that sorting a smaller list with any one of these methods is faster than a larger list (for ex, if the number of elements in the list is doubled, sorting times increase quadratically)

  ↳ Hence way may be thought of as dividing a large list into a number of smaller lists, sort them separately and then combine the results to get original list sorted.

C.A.R Hoare adopted This "DIVIDE-AND-CONQUER" Approach.

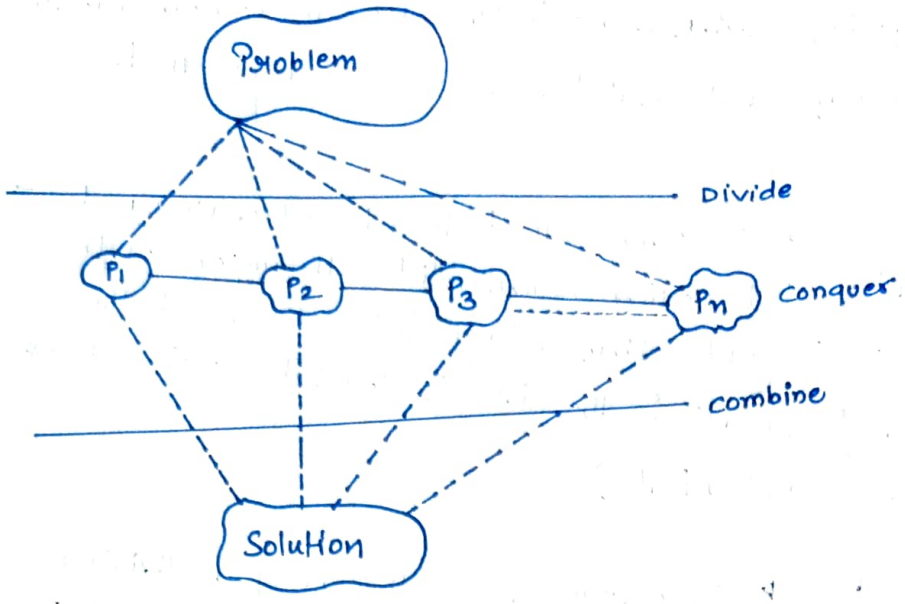
DIVIDE AND CONQUER APPROACH.

The principle of divide-and-conquer strategy is to solve a large problem by solving the smaller instances of the problem.

(a) Divide:- divide large problem into a number of smaller sub problem. If the sub-problem is not small enough then decompose them and continue the decomposition until all sub-problem are small enough.

(b) Conquer:- Find a suitable technique so that each sub-problem P_i $i = 1, 2, 3, \ldots, n$ can be solved quickly.

(c) Combine :- Combine the result of all solutions to all sub-problems to get the final solution.



Divide and Conquer Strategy

In general, a Recursive procedure is followed to acheive these tasks. The recursive framework of divide-and-conquer approach can be stated as below:→

```
Divide-and-Conquer(P)        // P is a problem to be solved.
1. n=size(P)                 // n denotes the size of the problem
2. If (n ≤ MIN)              // MIN denotes the desirable minimum size of a sub
                                problem and also the termination condition.
        S= Conquer(P)        // Solve the smallest unit using a simple method
3. Else {P1, P2, P3, ...., Pn} do   // Divide P into n sub-problems
4.      For each Pi ∈ {P1, P2, ..., Pn} do
            Si = Divide-and-Conquer(Pi)   // Call recursively
5. S= Combine (S1, S2, ..., Sn)   // Combine all solutions.
6. Return (S)
```

DIVIDE-AND-CONQUER APPROACH IN QUICKSORT

```
QuickSort (A, first, last)
1. If (first < last) then    // Termination condition.
2.    p = Partition (A, first, last)  // Partition list at P
3.    Quicksort (A, first, p-1)  // Recursive call for left subtree.
4.    Quicksort (A, p+1, last)  // Recursive call for right subtree.
5. Return
```

* **Divide:→**
  Partition the list $A[1,2,\cdots,n]$ into two sublists $A[1,\cdots p-1]$ (left sub-tree) and $A[p+1,\cdots,n]$ (right sub-list) such that each elements in the sub-list $A[1,\cdots,p-1]$ is less than or equal to $A[p]$ and each element in the sub-list $A[p+1,\cdots,n]$ is greater than $A[p]$.

* **Conquer:→**
  → Sort the two sublist $A[1,\cdots,p-1]$ and $A[p+1,\cdots,n]$ recursively, as if each sub-list is a list to be sorted and following the same divide-and-conquer strategy until sub-lists contain either zero or one element.

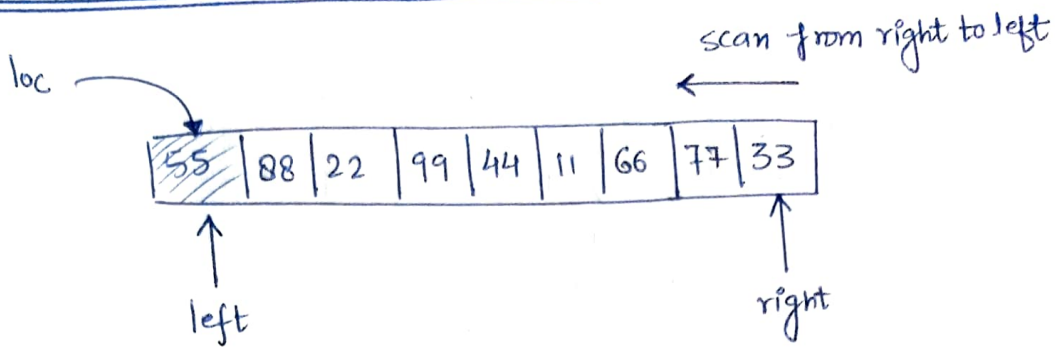
* **Combine:→**
  → Since the sub-lists are sorted in place, no work is needed to combine them. That is, the task Combine in the Quick sort is implicit.

Partition Method in QuickSort

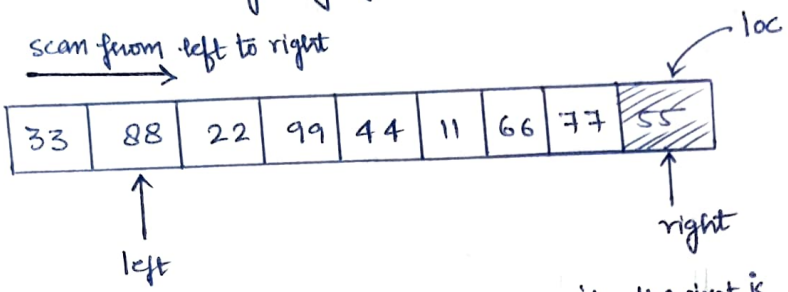
Steps

```
1. loc = left
2. While ((left<right)) do
3.    While (A[loc] ≤ A[right]) and (loc<right) do
4.       right= right-1
5.    End While
6.    If (A[loc] > A[right]) then
7.       Swap (A[loc], A[right])
8.       loc = right
9.       left= left+1
10.   EndIf
11.   While (A[loc] ≥ A[left]) and (loc>left) do
12.      left= left+1
13.   EndWhile
14.   If (A[loc] < A[left]) then
15.      Swap (A[loc], A[left])
16.         loc= left
17.         right= right-1
18.   EndIf
19. EndWhile
20. Return (loc)
21. Stop.
```

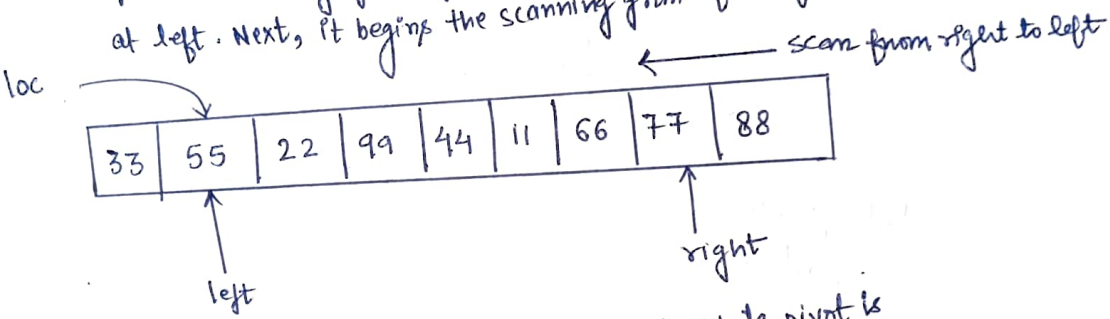
ILLUSTRATION OF THE PARTITION METHOD



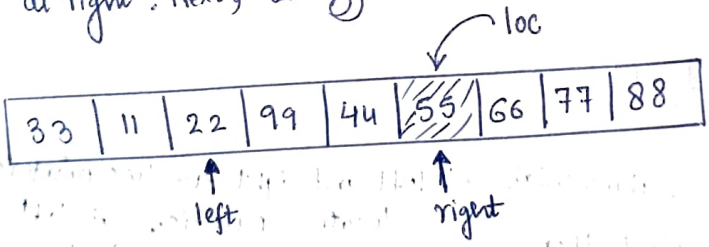
(A) At the begining of the partition.



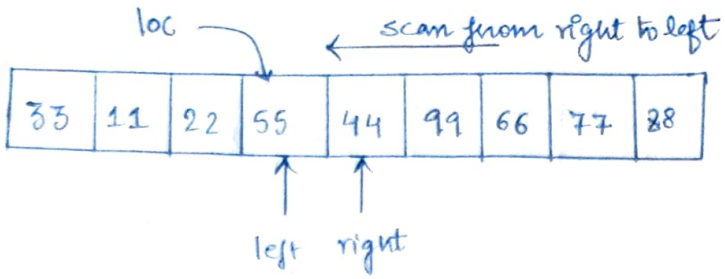
(b) After the scanning from right to left while the pivot is at left. Next, it begins the scanning from left to right



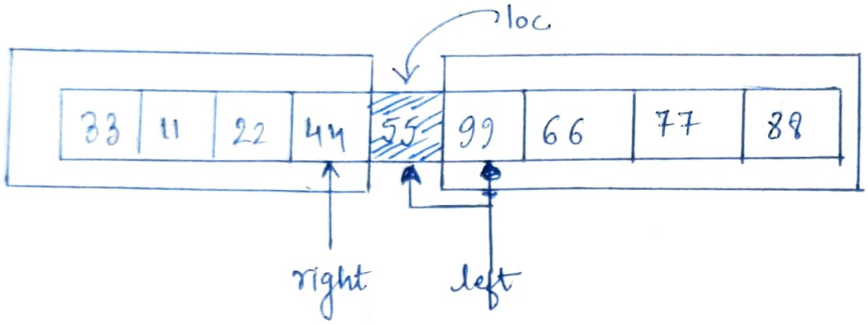
(c) After the scanning from left to right while pivot is at right. Next, it begins the scanning from right to left.



(d) After the scanning from right to left while the pivot is at left. Next, it begins the scanning from left to right.



(e) After the scanning from left to right while the pivot is at right. Next, it begins the scanning from right to left.



(f) After the scanning from right to left while the pivot is at left. left < right condition violates. Partition is done.

ALGORITHM QUICKSORT

Steps:

1. left = low , right = high // left and right are two pointers to locate partitions at left and right respectively.
2. If (left < right) then
3. loc = Partition (A, left, right)
4. QUICKSORT (A, left , loc-1) // Perform QuickSort over left subtree.
5. QUICKSORT (A, loc+1 , right) // Perform QuickSort over right subtree.
6. Endif
7. Stop

TIME COMPLEXITY OF QUICK SORT

→ From the algorithmic QuickSort → Time complexity of QuickSort depends upon PARTITION and CALLS of two QuickSort on two lists of smaller sizes.

→ If T(n) → rep. total time to sort n elements.
P(n) → " " " " performing partition.

then we write,

$$\boxed{T(n) = P(n) + T(n_1) + T(n_2)}$$

n_1 = no. of elements in left sublist.
n_2 = no. of elements in right sublist.

→ Note that basic operation in the QuickSort technique is comparison and exchange.

→ QuickSort → basically a repetition of partition methods.

→ Let C(n) and M(n) denotes the number of comparisons and movements.

Then $\boxed{T(n) = C(n) + M(n)}$

There are three major cases of the input arrangement for which we calculate the time complexities :→

Case 1: Elements in the list are stored in Ascending Order → ⓐ No. of Comparisons.
In this case, the partition method requires only one scan from right to left with n-1 comparisons. In this case partition sublist left remain empty and right subtree with n-1 element

RECURRENCE RELATION

$$C(n) = n-1 + C(n-1), \text{ with } C(1) \text{ and } C(0) = 0$$

Expanding with Recurrence Relation, we get

$$C(n) = (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$$

$$= \frac{n(n+1)}{2} \quad \frac{n(n-1)}{2}$$

(b) Number of Movement

No exchange operation is involved in this case.

Hence the number of movement M(n) is given by

$$M(n) = 0$$

CASE 2: Element in the list are in Descending Order.

(a) No. of comparison

In this case, the partition method requires n+1 comparisons.

Unlike the case 1, in this case the non empty sublists are alternatively left sub-list and right sublist.

It can be verified → for every odd numbered Recursive call right sub list is empty, whereas for every even numbered Recursive call the left sublist in empty.

Now if C(n) denotes the no. of comparisons in this case, we have

$$C(n) = n-1 + C(n-1), \text{ with } C(1) = C(0) = 0$$

* Expanding Recurrence Relation

$$C(n) = (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$$
$$= \frac{n(n-1)}{2}$$

(b) Number of Movements

So for the no. of movements is concerned,

→ Single exchanged operation in each odd numbered Recursive call.

→ No exchange is involved during any even numbered Recursive call.

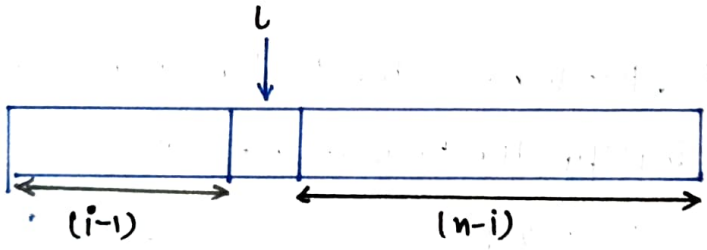
As there are altogether ~~of~~ n number of recursions, we have the number of movements as :→

$$M(n) = \begin{cases} \frac{n-1}{2}, & \text{if } n \text{ is odd} \\ \frac{n}{2}, & \text{if } n \text{ is even} \end{cases}$$

In other way, the same can be represented as $M(n) = \left\lfloor \frac{n}{2} \right\rfloor$

CASE 3: Elements in the list are stored in Random Order

(a) No. of comparisons



To find the final berth of the pivot element, we need (n-1) comparisons in the partition method on a list with n element.

Now assume that the partition method finds the i^{th} position for the pivot element, thus it results two sublists :

LEFT SUB-LIST with (i-1) elements and right subtree = n-i elements.

Now, all permutations of elements in these two sublists are equally likely. We assume that each possible position for the pivot element is equally likely.

That is value of i has any value b/w 1 and n with a probability = 1/n.

The recurrence equation for the no. of comparisons in this case as

$$C(n) = (n-1) + \sum_{i=1}^{n-1} \frac{1}{n} \left[C(i-1) + C(n-i) \right] \text{ with } C(1) = C(0) = 0$$

Above equation can be simplified as stated below:-

The term C(i-1) varies from C(0) to C(i-1)
whereas C(n-i) varies from C(n-1) to C(0)
over the entire range of summation series,
that is, i=1 to n-1.

In otherwords, two terms contribute the same series.
So we can simplify the Recurrence Relation

$$C(n) = (n-1) + \frac{2}{n} \sum_{i=1}^{n-1} C(i) \text{ for } n \geq 1$$

Replacing n by $n-1$, we have

$$C(n-1) = (n-2) + \frac{2}{n-1} \sum_{i=1}^{n-2} C(i)$$

Now $n \cdot C(n) - (n-1) \cdot C(n-1) = n(n-1) + 2\sum_{i=1}^{n-1} C(i) - (n-1)(n-2) - 2\sum_{i=1}^{n-2} C(i)$

$$= (n-1) - (n-n-2) + 2\left[\sum_{i=1}^{n-1} C(i) - \sum_{i=1}^{n-2} C(i)\right]$$

$$= 2(n-1) + 2C(n-1)$$

$$n \cdot C(n) = 2(n-1) + (n+1) \cdot C(n-1)$$

$$\frac{C(n)}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{C(n-1)}{n}$$

Let → $C'(n) = \frac{C(n)}{n+1}$ Then

$$C'(n) = C'(n-1) + \frac{2(n-1)}{n(n+1)}, \text{ with } C'(1) = 0$$

This is the simplified version of Recurrence Relation and can be solved just by simple expansion.

$$C'(n) = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} = \sum_{i=1}^{n} \frac{2(i+1)-4}{i(i+1)} = \sum_{i=1}^{n} \left[\frac{2}{i} - \frac{4}{i(i+1)}\right]$$

Finally $C'(n) = 2\sum_{i=1}^{n} \frac{1}{i} - 4\sum_{i=1}^{n} \frac{1}{i(i+1)}$

This is Harmonic Series whose sum. is

$$\sum_{i=1}^{n} \frac{1}{i} = \log_e n + 0.577$$

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \sum_{i=1}^{n} \left(\frac{1}{i} - \frac{1}{i+1}\right) = \left(\frac{1}{1} - \frac{1}{n+1}\right)$$

$$= \frac{n}{n+1}$$

Hence we get

$$C'(n) = 2(\log_e n + 0.577) - 4\frac{n}{n+1}$$

Substituting $C'(n) = \frac{C(n)}{n+1}$, we get

$$\boxed{C(n) = 2(n+1)(\log_e n + 0.577) - 4n}$$

Special Case.

Assume elements in the list arranged in such a way that every instance of partition method exactly partitions the lists into 2 lists of equal size.

The Recurrence Relation

$$C(n) = (n-1) + 2C(n/2)$$

Expand the recurrence relation

$$C(n) = (n-1) + 2C(n/2)$$

$$= (n-1) + 2\left[\left(\frac{n}{2}-1\right) + 2^2 C\left(\frac{n}{2^2}\right)\right]$$

$$= (n-1) + 2\left(\frac{n}{2}-1\right) + 2^2\left[\left(\frac{n}{2^2}-1\right) + 2C\left(\frac{n}{2^3}\right)\right]$$

$$= (n-1) + 2\left(\frac{n}{2}-1\right) + 2^2\left(\frac{n}{2^2}-1\right) + 2^3\left(\frac{n}{2^3}-1\right) + \cdots + 2^k\left(\frac{n}{2^k}-1\right)$$

For the sake of simplicity let $n = 2^K$ for some $K \geq 0$

Also using $C(0) = 0$, we get

$$C(n) = (n-1) + (n-2) + (n-2^2) + \cdots + (n-2^{K-1})$$

$$= n \cdot K - (2^0 + 2^1 + 2^2 + \cdots + 2^{K-1})$$

$$= n \cdot K - \left(\frac{2^K - 1}{2-1}\right)$$

$$= n \log_2 n - n + 1 \quad \because n = 2^K$$

This is in fact the simplest form of the number of comparisons.

(b) Number of Movements

- Let the location of pivot be i, where $1 \leq i \leq n$.
  Then after the partition, the pivot element is guaranteed to be placed in location i with $1 \ldots (i-1)$ elements in the left subtree and $(i+1) \ldots n$ elements

- The number of exchanges that is necessary to do this should not exceed $(i-1)$ for each of $(i-1)$ elements less than the pivot elements

- Let us denote $M(n)$, the average number of exchange operations done by the QuickSort on the list of size n. Then we have

$$M(n) = \frac{1}{n} \sum_{j=1}^{n} \left[(i-1) + M(i-1) + M(n-i) \right]$$

- Here we have taken the average of all no. of movements assuming that the pivot may be placed in any location blw 1 and n with a probability $= 1/n$.

- Simplifying the Recurrence Relation

$$M(n) = \frac{n-1}{2} + \frac{2}{n}\sum_{i=1}^{n-1} M(i).$$

- Since this recurrence relation is similar to the Recurrence Relation of the no. of comparisons, same steps followed to solve for $M(n)$, which can be obtained as :-

$$M(n) = 2(n+1)(\log_e n + 0.577) - 4n$$

ANALYSIS OF QUICKSORT ALGO.

Case	Comparison	Movement	Memory	Remark
Case 1	$C(n) = \frac{n(n-1)}{2}$	$M(n) = 0$	$S(n) = 1$	I/p list in sorted order
Case 2	$C(n) = \frac{n(n-1)}{2}$	$M(n) = \lfloor n/2 \rfloor$	$S(n) = 1$	I/p list sorted in reverse order
Case 3	$C(n) = 2(n+1).(\log_e n + 0.577) - 4n$	$M(n) = 2(n+1).(\log_e n + 0.577) - 4n$	$S(n) = \lceil \log_2 n \rceil + 1$	Input list is in random order

Time Complexity of the Quick Sort

RunTime, T(n)	Complexity	Remarks
$T(n) = c\frac{n(n-1)}{2}$	$T(n) = O(n^2)$	Worst Case
$T(n) = c\left(\frac{n(n-1)}{2} + \left\lfloor\frac{n}{2}\right\rfloor\right)$	$T(n) = O(n^2)$	Worst Case
$T(n) = 4c(n+1)(\log_e n + 0.577) - 8cn$ or $T(n) = 2c[n\log_2 n - n + 1]$	$T(n) = O(n\log_2 n)$	Best/ Average case

- We have seen Worst Case occur when the elements in the list are already sorted or are in Reverse Order. The worst case complexity is as Bad as Insertion, Selection or Bubble Sort.

- A Best case is possible when the partition method always divide the list into nicely half. and in that case complexity is $O(n\log n)$.

We have also analysed the average case behaviour of the QuickSort with time complexity $= 4c(n+1)(\log_e n + 0.577) - 8cn = O(n\log_2 n)$

We Therefore conclude that one fact that the average case complexity is very close to that of Best case that is $O(n\log n)$.

Memory Requirement of QuickSort

What ever it produces or procedure, it is always necessary to maintain a stack. Why a stack is required?

An answer to this question → is that, a stack is required → to store all sub-lists (nothing but the two boundary indexes of the list) which is yet to be processed.

Initially, the stack is pushed with two boundary indexes of the input list and first recursion starts with the call to this list.

On partition, it produces two sublists, which are pushed onto the stack.

It then pops a list, call quick sort on the list popped, pushes two lists on partition, and so on. These steps are repeated until stack is empty.

It is evident that the size of the stack depends upon the no. of sublists into which a list may be partitioned towards the completion of the sorting.

Now every call make one pop and two pushes operation, thereby net increase in the stack entry is one.

In other words, size of the stack should be as large as the No. of Recursive Calls involved.

No. of Recursive Call, however depends upon how the lists are partitioned during Recursion.

→ It can be analysed that least number of Recursive calls occurs when each recursion splits a list into two (nearly) equal halves.

→ In such a case maximum size of the stack would be $\lceil \log_2 n \rceil + 1$, where n is the number of elements in the input list.

→ On the other hand, if the input list is either sorted in ASCENDING or DESCENDING order then the number of Recursive call will be exactly n. In this case, there will be atmost one entry in the stack. This is because ONE SUB-LISTS will always be empty, and it pushes the non-empty sublists, which popped in the next Recursion and this continues till there is a single element in the non-empty sub-lists.

→ So considering **Worst Case situation**, the storage space requirement $S(n)$ to sort a list with n elements is

$$S(n) = \lceil \log_2 n \rceil + 1$$

→ Maximum No. of Recursion occurs when list partitioned into 2 halves.

MAXIMUM AND MINIMUM NUMBER OF RECURSIVE CALLS TO SORT A LIST OF N ELEMENTS USING QUICK SORT IS 2N-1 AND N, RESPECTIVELY

PROOF:

Maximum no. of Recursion occurs when each recursive call partitions the list into equal or near equal two sub-lists. For simplicity in the proof, let us assume that $N = 2^K$ where $K \geq 0$ is a constant.

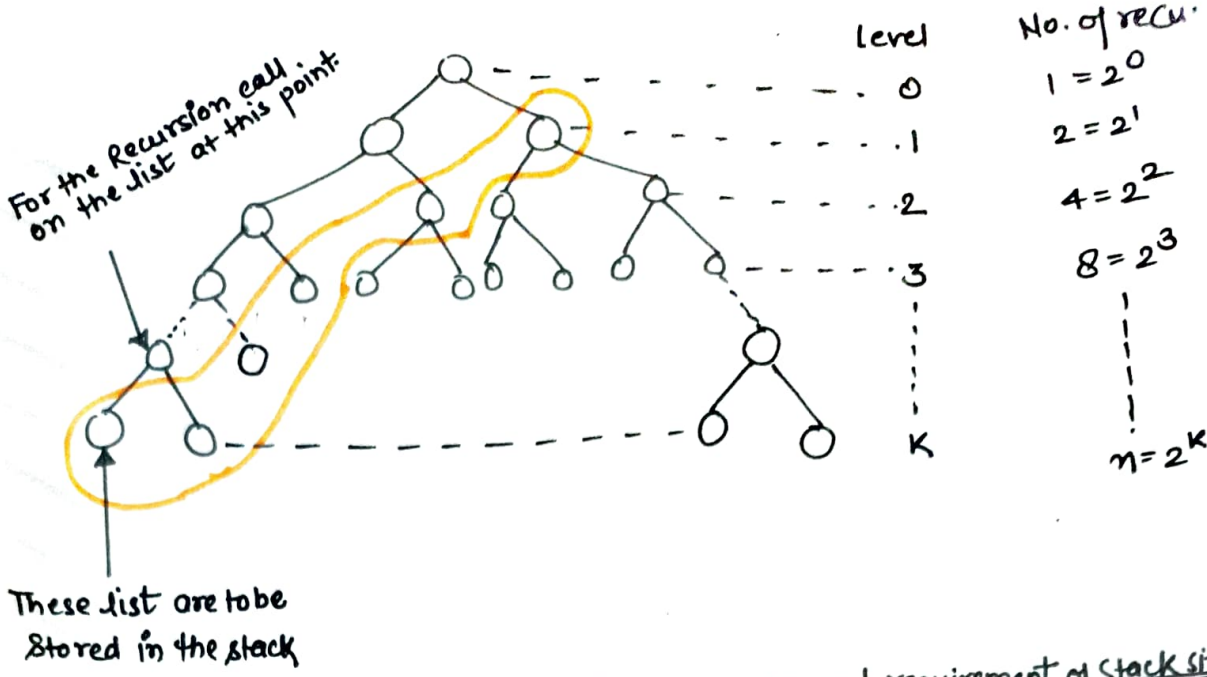
→ FIRST RECURSION BEGINS with the list of size N.
It partitions the list into two sublists each of size $\approx N/2$
This will be followed by the calls of two Recursions each on the the two sublists, which in turn create into four sublists each of size $\approx n/4 = n/2^2$.
Next, there will be four Recursive calls on each of the four lists. This will continue till a sub-list contains single elements, that is size of the list becomes $N/2^K$.

Thus maximum number of Recursive Calls in Order to Sort N elements using QUICK SORT. is

$$R_{max} = 1 + 2 + 4 + \dots + 2^K$$

$$= 2^{K+1} - 1 \quad = 2N - 1 \quad \text{(solve using log)}$$

$$= 2N - 1.$$



* Maximum number of Recursion calls and requirement of stack size.

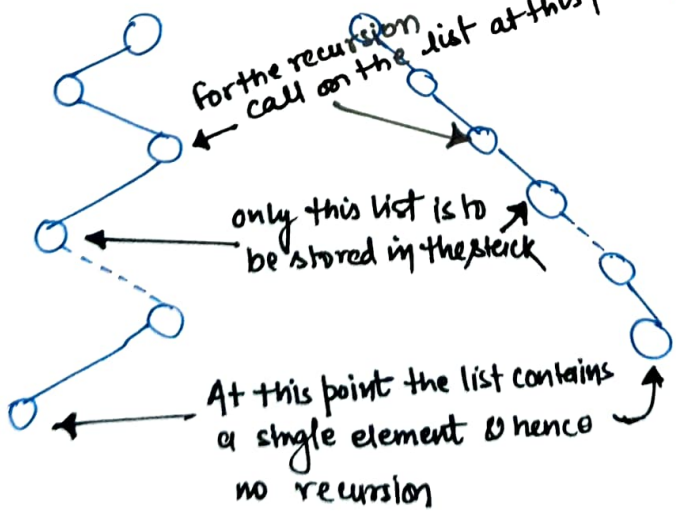
* Minimum No. of Recursion calls & requirement of stack size.

ON The other hand, minimum number of Recursive calls occur when all elements in the input list is already sorted in ASCENDING or DESCENDING ORDER.

Let us assume that the list is sorted in ASCENDING ORDER.

In this case, each Recursive call partitions the lists so that the left sub-~~tree~~ list is empty and Right sublist contains one element less than the size of the list prior to the call of the Recursion.

Such a Recursion Tree is a skewed Binary Tree with maximum no. of levels is ~~n~~ N-1 → Thus No. of Recursive call is given by $R_{max} = n$.



Level	No. of Recursion
0	1
1	1
2	1
3	1
⋮	
n-1	1