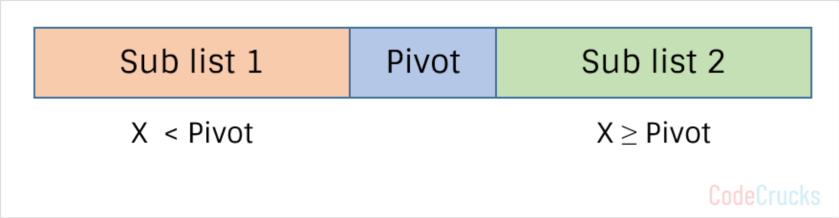
Quick Sort

Quick sort is one of the fastest sorting algorithms that use divide and conquer approach of problem-solving. Here are some excellent reasons to learn quicksort:

** Often the best choice for sorting because it works efficiently in O(nlogn) time on average.
** One of the best algorithms to learn recursion. Recursive structure, flow of recursion, and base case of quicksort are intuitive.
** An excellent algorithm to learn worst, best, and average-case analysis.
** Quick sort is an in-place sorting algorithm that works best in virtual memory environment.
** An excellent algorithm to learn idea of randomized algorithms. Randomization is a popular tool to improve efficiency of real-time systems.
** Moreover, the space requirement is also an important difference between quicksort and merge sort. Quicksort requires minimum space compared to merge sort.

The intuition of Quick Sort
From the above observation, we can try to think in a reverse way: Suppose we partition unsorted array around some input value **pivot** into two parts.

** All values in the left part are less than
** All values in the right part are greater than



After this partition process, pivot will get placed at correct position in the sorted output. Now we have a situation similar to sorted array with one difference: left and right halves are still unsorted. If we observe further, both unsorted halves are smaller sub-problems of the original sorting problem, i.e., a sorting problem for smaller input sizes.

If we sort both halves recursively, whole array will get sorted because all values in the left side of pivot are less than pivot, pivot is at correct position, and all values in the right side of pivot are greater than pivot.

Pivot = 6
i=-1, j=0

11	10	13	5	8	3	2	6

i=-1, j=1

11	10	13	5	8	3	2	6

i=-1, j=2

11	10	13	5	8	3	2	6

i=-1, j=3

11	10	13	5	8	3	2	6

i=0, j=4

5	10	13	11	8	3	2	6

i=0, j=5

5	10	13	11	8	3	2	6

i=1, j=6

5	3	13	11	8	10	2	6

i=2

5	3	2	11	8	10	12	6

Swap (arr[i+1],pivot) i=5

5	3	2	6	8	10	12	11

Divide and conquer idea of quicksort

The above approach is a divide and conquer approach of problem-solving: We divide sorting problem into two smaller subproblems using partition algorithm and solve each sub-problems recursively to get the final sorted array. We repeat same process of partition for smaller sub-arrays till we reach the base case.

Divide part: We divide problem into two smaller sub-problems by partitioning the array X[l…r] into two smaller subarrays around pivot. The partition process returns the pivotIndex i.e. index of pivot in the sorted array.

Subproblem 1: Sorting array X[] from l to pivotIndex - 1.

Subproblem 2: Sorting array X[] from pivotIndex + 1 to r.

Conquer part: Now, we solve both sub-problems or sort both smaller arrays recursively. We should not worry about the solution of sub-problems, because recursion will do this work for us.

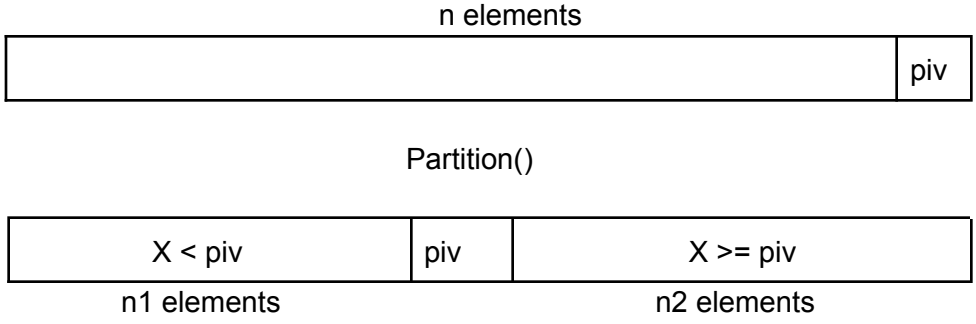
Combine part: This is a trivial case because after sorting both smaller arrays, whole array will get sorted. In other words, there is no need to write code for combine part.

Quick Sort Pseudocode

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

```
QUICKSORT(array A, int p, int r)
1  if (p < r)
2      then q ← PARTITION(A, p, r)
3           QUICKSORT(A, p, q − 1)
4           QUICKSORT(A, q + 1, r)
```

Time and Space Complexity Analysis



The recurrence solution/ equation:

T(n) = Time it to take solve the problem n1 + Time it to take solve the problem n1 +
To perform this partition operation.

= T(n1) + T(n2) + O(n) [TC of Partition(A,p, r) method is O(n)]

= T(n1) + T(n2) + cn

Worst Case

1	2	3	4	**5**

T(n1) = n-1 T(n2) = 0

1	2	3	**4**	**5**

T(n1) = n-2 T(n2) = 0

1	2	**3**	**4**	**5**

T(n1) = n-3 T(n2) = 0

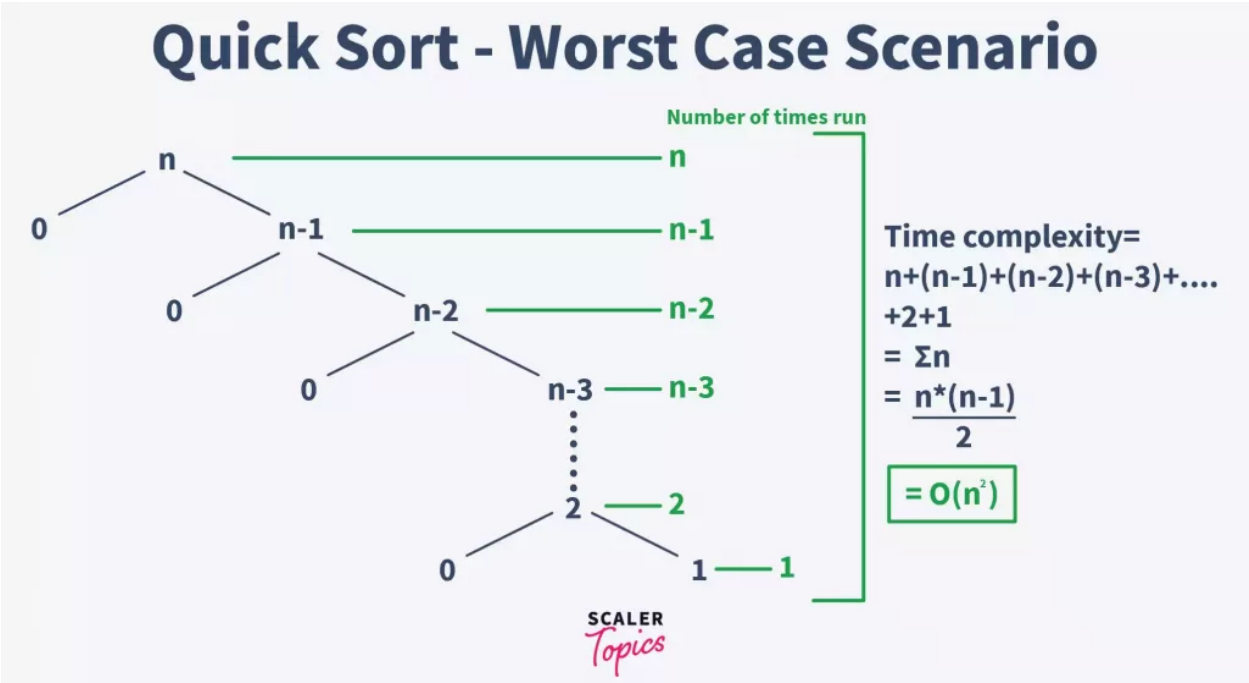
1	**2**	**3**	**4**	**5**

T(n1) = n-4 T(n2) = 0

1	**2**	**3**	**4**	**5**

T(n1) = n-5 T(n2) = 0

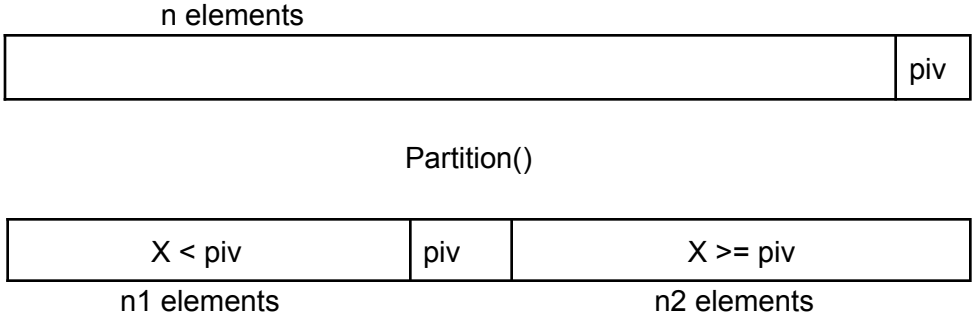
So, the recurrence soln/eqn=> T(n) = T(0) + T(n-1) + cn



So we can say that, when the array is already **sorted** and you always pick the largest element or smalest element as a **pivot.** The worst case senario is occured.

Time Coplexity = O(n^2)

Best Case

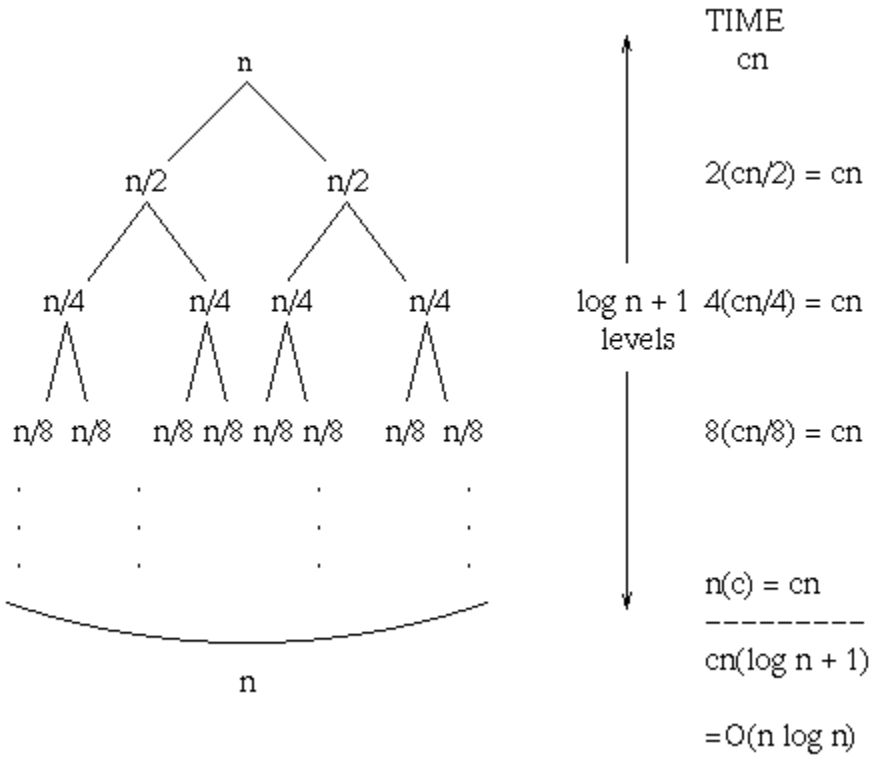


Where n1 and n2 both are almost equal to n/2;

So, the recurrence soln/eqn=> T(n) = T(n/2) + T(n/2) + cn

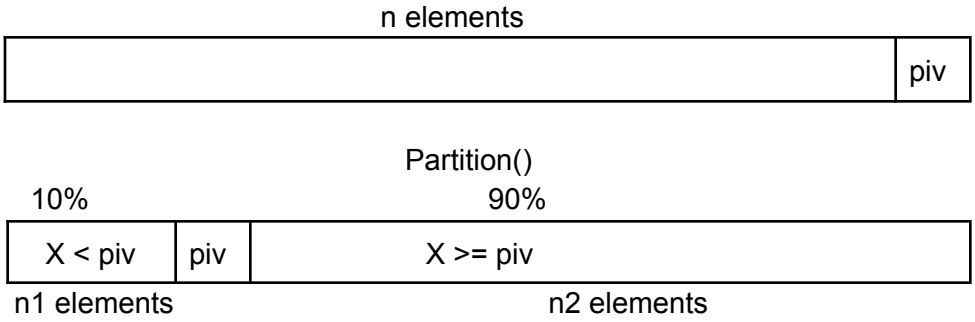
= 2T(n/2) + cn [equal to merge sort]

= O(nlogn)



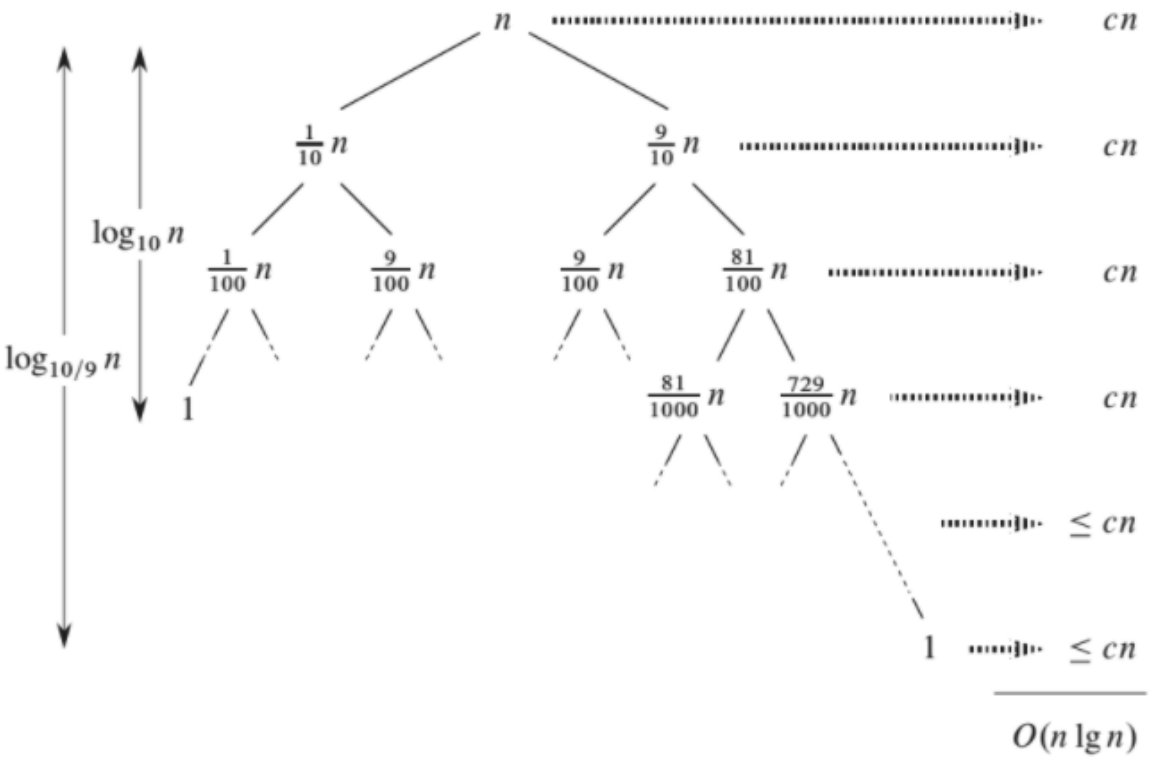
Time Complexity = O(nlogn)

Almost Best Case



n1 = n/10 n2 = 9n/10

So, the recurrence soln/eqn=> T(n) = T(n/10) + T(9n/10) + cn



Time Complexity = O(nlogn)

Randomized quick sort

p		m				r
x1	x1	x3	x4	x5	x6	x7

1. Pick a random index value from **p to r** store **m**
2. **swap arr[r] and arr[m]**
3. **pivot = arr[r]**

** probability of picking greatest or smallest element as a pivot is extremely low in Randomized Quick sort.

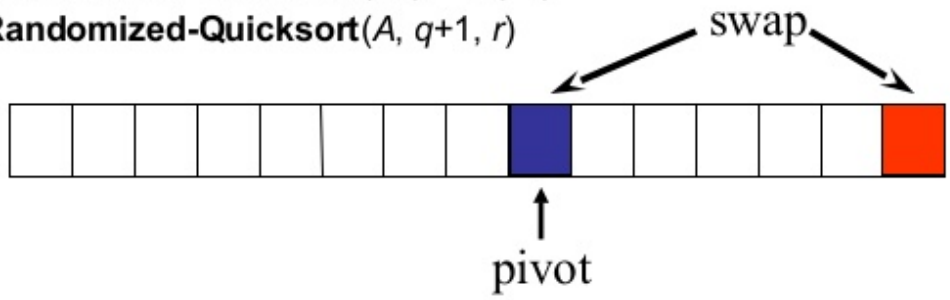
Randomized Quicksort

Randomized-Partition(*A, p, r*)

1. $i \leftarrow Random(p, r)$
2. exchange $A[r] \leftrightarrow A[i]$
3. **return Partition**(A, p, r)

Randomized-Quicksort(*A, p, r*)

1. **if** $p < r$
2. **then** $q \leftarrow$ **Randomized-Partition**(A, p, r)
3. **Randomized-Quicksort**$(A, p, q\text{-}1)$
4. **Randomized-Quicksort**$(A, q\text{+}1, r)$



Time Complexity: O(nlogn)

Space Complexity

Quicksort is an in-place sorting algorithm where we are not using extra space. But as we know, every recursive program uses a call stack in the background to execute recursion. So space complexity of quick sort depends on the size of recursion call stack, which is equal to the height of recursion tree.

In the worst-case scenario, partition is always unbalanced, and there will be only 1 recursive call at each level of recursion. In such a scenario, generated recursion tree is skewed in nature. So the height of tree = O(n) and recursion requires call stack of size O(n). **Worst-case space complexity** of quick sort = O(n).

Partition is always balanced in the best-case scenario, and there will be 2 recursive calls at each level of recursion. In such a scenario, generated recursion tree looks balanced in nature. So the height of tree = O(logn), and recursion requires call stack of size O(logn). **Best-case space complexity** of quick sort = O(logn).

CLASS LECTURE