Algorithms: Sorting Algorithms

Given a sequence of numbers, you have to arrange them in the ascending or descending order. This is called sorting. Sorting algorithms can be iterative or recursive.

Let's learn about the different sorting algorithms and analyze their space and time complexity. First we are going to start with the iterative Insertion Sort.

INSERTION SORT ALGORITHM AND ANALYSIS

"Insertion sort", the name suggests that there is an insertion happening. Understand it with an example where you want to arrange the playing cards which are placed on a table, face down.

1. Take the first card in your left hand.
2. Take the second card in your right hand, compare it with the left hand card

   If smaller then keep this card in the most left side on the left hand
   If larger then keep this card in the right most side on the left hand

3. Repeat step 2 over again.

At every new picking of the card, your left hand will always have sorted cards. Let us right the algorithm in the context of an array containing integers which we will sort using **Insertion Sort.**

```
Insertion_Sort(A)

{

        for( j = 2 to A.length)
        {
                key = A[ j ];
                //insert A[ j ] into sorted sequence A[1 … j-1]
                i = j – 1;
                while( i > 0 and A[ i ] > key)
                        A[i+1] = A[i];
                        i = i - 1;

                A[i+1] = key;

        }

}
```

Let array = 9, 6, 5, 0, 8, 2, 7, 1, 3 (please follow the algorithm above to sort the list).

Time complexity:

The worst case time complexity for insertion sort can be calculated as below. For worst case to happen, the elements should be already in descending order in the array for which we have to do ascending order sorting using Insertion Sort.

For j	Comparison	Movements	Total Operations
2	1	1	2
3	2	2	4
4	3	3	6
…	…	…	…
N	n-1	n-1	2(n-1)

Therefore, total operations done: $2 + 4 + 6 + \ldots + 2(n-1)$

Or: $2(2-1) + 2(3-1) + 2(4-1) + \ldots + 2(n-1)$

Or: $2(1) + 2(2) + 2(3) + \ldots + 2(n-1)$

Or: $2 (1 + 2 + 3 + \ldots + n-1)$

- ⇨ $2(n(n-1))/2$
- ⇨ **$O(n^2)$**

The best case time complexity for Insertion Sort can be calculated as below. For best case to happen, the array should already have the sorted elements inside it.

For j	Comparison	Movements	Total Operations
2	1	0	1
3	1	0	1
4	1	0	1
…	…	…	…
N	1	0	N - 1

Total operations for best case time complexity: $1 + 1 + 1 + \ldots + N-1$

When the array is already sorted you will just compare the element in you right hand (card analogy) with the rightmost element in the left hand only once for each element in the array.

Therefore, best case time complexity = Ω(n)

<u>Space Complexity of Insertion Sort:</u>

Apart from the input array A[], I only need three variables ‘key’, ‘i’ and ‘j’. So, whatever is the size of the input, I only need 3 extra variables for Insertion sort to work.

Therefore, the space complexity for Insertion Sort is **O(1)** or constant space.

Note: Whenever a sorting algorithm executes in constant space, we call such a sorting algorithm INPLACE algorithm

In Insertion Sort, we saw that the time complexity depends on the number of comparisons involved and the number of movements involved. So how can we decrease this complexity? Let's try some methods.

1. Using Binary Search instead of sequential search in the sorted list

Comparisons using Binary Search	Movements using Binary Search	Total complexity
O(logn)	O(n) (no reduction)	O(nx(n-1) elements)) = $O(n^2)$

Using binary search will not reduce the time complexity of the Insertion Sort algorithm.

2. Using Linked List for insertion

Comparisons using Linked List	Movements using Linked List	Total complexity
O(n) (no reduction)	O(1) (direct insertion)	O(nx(n-1 elements)) = $O(n^2)$

Using linked list will also not reduce the time complexity of the Insertion Sort Algorithm.

Time and Space complexity for Insertion Sort

Time Complexity	**Best Case**	**Worst Case**	**Average Case**
	$\Omega(n)$	**$O(n^2)$**	**$\Theta(n^2)$**

Space Complexity	**O(1)**

Let's move on to the next sorting algorithm which is going to work better than Insertion Sort in terms of time complexity. This algorithm is called Merge sort.

MERGE SORT ALGORITHM AND ANALYSIS

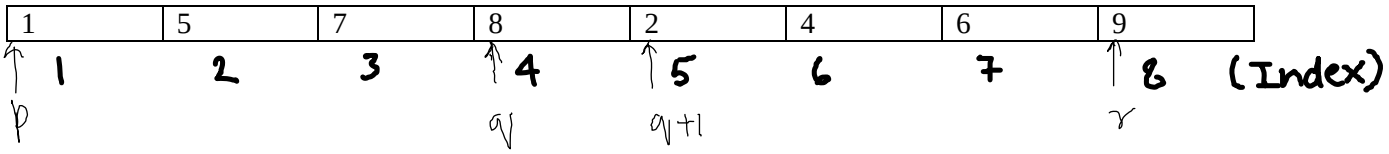
Merge sort falls under the Divide-and-conquer algorithms. It simply means you divide the problem into smaller steps and tackle the problem of sorting using bottom-up approach.

Before we go into the algorithm and analysis of it, let's understand the heart of the algorithm, which is a procedure called Merging. For clarity, here is the algorithm

```
MERGE(A, p, q, r)
{
        n1 = q-p+1
        n2 = r – q
        Let L[1…n1+1] and R[1 … n2 + 1] be new arrays
        for( i=1 to n1)
                L[i] = A[p + i-1]
        for( j=1 to n2)
                R[j] = A[q + j]
        L[n1 + 1] = ∞
        R[n2 + 1] = ∞
        for(k = p to r)
                if(L[i] <= R[j]
                        A[k] = L[i]
                        i=i+1
                else A[k] = R[j]
                        j=j + 1
}
```

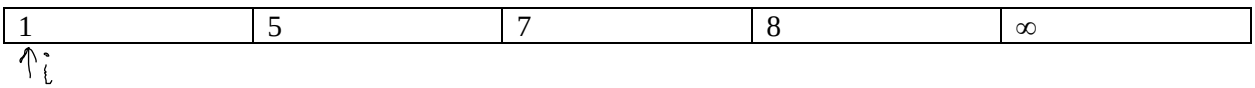
Working of MERGE:

We have a list of elements where left half and right half are sorted.

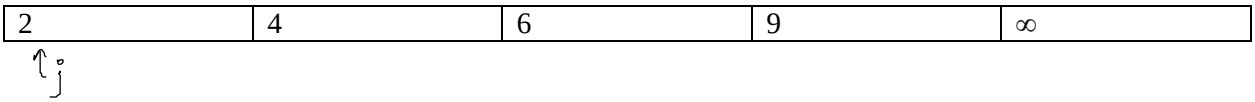


We divide the list above into two lists L and R containing the first half sorted elements and the second half sorted elements.

L = An array of size q-p+1 where last element is a very big number denoted by ∞ here.



R = An array of size r-q where last element is a very big number denoted by ∞ here.



Now, we compare the elements of L and R and arrange them in the ascending order in the final array. The final array looks like below.

A =

1	2	4	5	6	7	8	9

For merge sort to work we need to copy the elements from the input which takes ‘n’ extra space other than the input. And to compare the two sorted lists it takes one comparison per input, therefore it will take ‘n’ comparisons.

Time Complexity = O(n)

Space Complexity = O(n)

If the size of list L and R is ‘n’ and ‘m’ respectively, the time and space complexity will be O(n+m). The lists L and R should have sorted elements in them.

Note: We add ∞ in the end of the two lists for proper comparison. If L = 10,20,30,40, ∞ and R = 1,2,3,4, ∞ then A = 1,2,3,4.. initially but to enter the elements of L we need some element to compare it to, so the fifth element in R is compared with the elements of L to add them to the sorted list A. Then A will become 1, 2, 3, 4, 10, 20, 30, 40

Now that we have seen how the MERGE method is working. Let’s use it to create Merge Sort algorithm.

Working of MERGE SORT

```
Merge_sort(A, p, r)
{
        if p < r
                q = floor[(p+r)/2]
                Merge_sort(A, p, q)
                Merge_sort(A, q+1, r)
                MERGE(A,p,q,r)
}
```

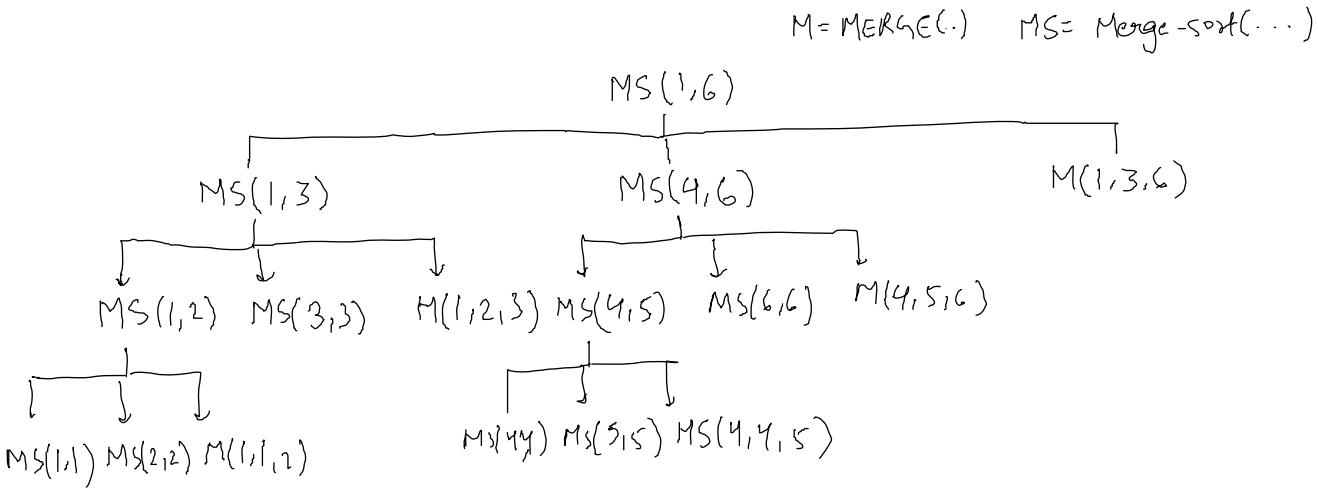
As said earlier, Mege Sort falls under the category of Divide and Conquer algorithms. In this sorting technique we divide the array to be sorted into single element array (single element array is already sorted) and move up from there to give us the final sorted array.

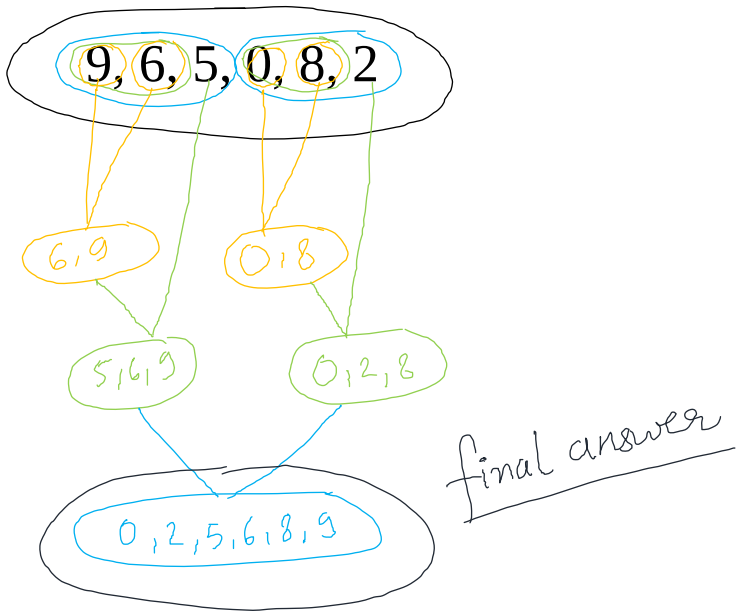
This is a recursive algorithm where each function call is going to call three functions:

Merge_sort(A, p, q)
Merge_sort(A, q+1, r)
MERGE(A,p,q,r) Let’s see how it is doing the work using an example.

Sort the array A = [9, 6, 5, 0, 8, 2] using merge sort.

Recursion tree for Merge Sort.





Space Complexity Analysis:

We know that the extra space required for the Merge Operation is O(n).

We need extra space in the stack for function calling. To analyze this, we need

1. Total number of function calls made
2. The order in which they are made
3. The height of the stack that is needed for this.

Total number of function calls made for above example: 16 so do we really need a stack of size 16 for this? Let's see.

That is indeed not the case: The height of the stack is equal to the height of the recursion tree made by the merge sort algorithm.

Note: All the function calls in a level will be carried out in the stack memory of same level. Level 3 calls will be made on the third memory cell of the stack. (Interesting).

The height of the recursion tree for an input size of n = ceil(logn) + 1. For every level we need a cell in the stack which let's say occupies 'k' space units.

Therefore, for stack we need k(logn + 1) space.

- ⇨ **Space complexity for stack: O(k(logn+1)) or O(logn)**
- ⇨ **For merge procedure space required is of O(n)**

Total space complexity = O(n + logn) or O(n) for merge sort.

Now let us analyze the time complexity of the merge sort algorithm

Time Complexity Analysis:

Let us say we are sorting an array of size n.

Let's say that the time taken by merge sort on array of size n = T(n)

```
Merge_sort(A, p, r)          → T(n)
{
        if p < r
                q = floor[(p+r)/2]
                Merge_sort(A, p, q)      → T(n/2)
                Merge_sort(A, q+1, r)    → T(n/2)
                MERGE(A,p,q,r)           → O(n)
}
```

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

Using Master's Theorem

$a = 2 \quad b = 2 \quad k = 1 \quad p = 0$

$a = b^k \; \& \; p > -1$

$\therefore T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

$T(n) = \Theta(n^{\log_2 2} \log^{0+1} n)$

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

Time and Space complexity for Merge Sort

Time Complexity	Best Case	Worst Case	Average Case
	Ω(nlogn)	**O(nlogn)**	**Θ(nlogn)**

Space Complexity	O(n)

Let's see some questions on Merge Sort.

Q1. Given "logn" sorted list each of size "n/logn". What is the total time required to merge them into one list?

Total number of sorted lists = logn

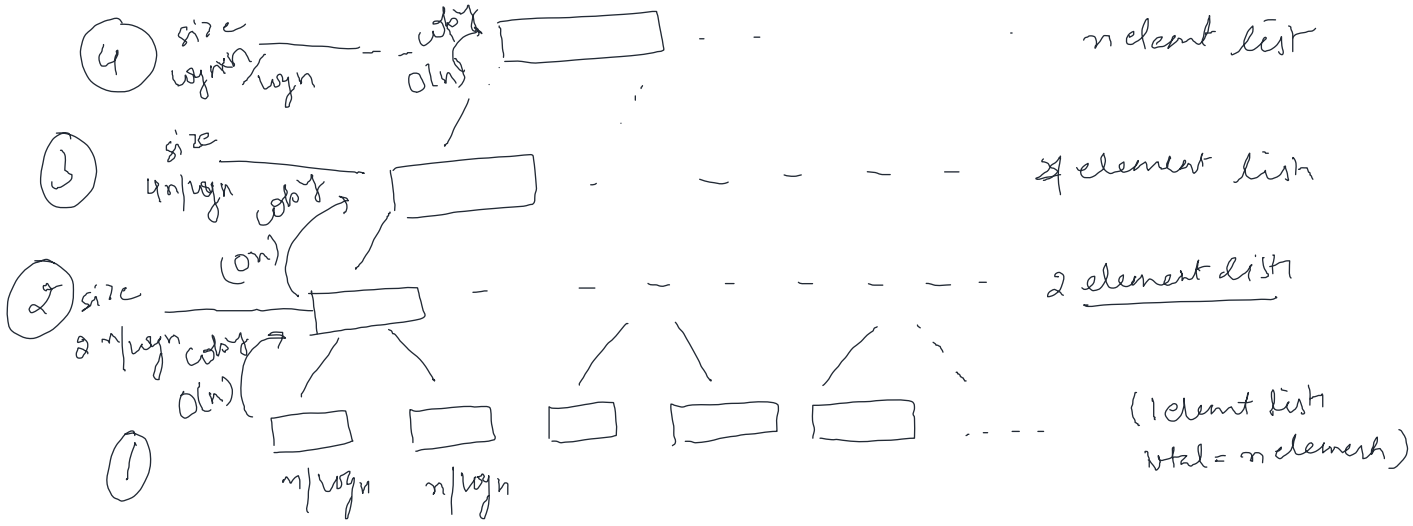
Each list is of size: n/logn

Total number of elements : logn x n/logn = n

For copying the elements in level of the recursion tree =O(n)

This copying has to be done for the height of the tree= loglogn

Total time complexity : O(nloglogn)



total complexity = O(n) X no. of climbings I have to do to reach last level of tree

from (1) to (2) climbings = log 2 (or 1 climb)

" (2) to (3) " = log 4 (or 2 climbs)

from 1 to last (top of tree) climbings = log (log n) (or log n climbings)

∴ complexity = O(n log log n)

Q2. "n" strings each of length "n" are given then what is the time taken to sort them?

If two strings are of length n, then the time taken to compare them will be O(n)

At the bottom level we are given n string each of length n.

For one comparison time taken = O(n)

For n comparisons time taken = $O(n^2)$

For copying in the next level time taken = n strings of size n = $O(n^2)$

Total time taken for comparison + copying in **one level** = $2.O(n^2)$ or $O(n^2)$

This has to be repeated until we reach the top level. So how many levels we are going to traverse for that?

To go from bottom to (bottom -1) level we take one step or log2 steps or ***log(size of array at bottom-1 level)***

From bottom to (bottom -2) level we take two steps or log4 steps or ***log(size of array at bottom -2 level)***

We know the size of array at the top level has to be 'n'.

Therefore to reach the top level, we have to take log(size of top array) or logn steps.

Total time complexity : (number of steps taken) X (time complexity for copying and comparison on each level)

- ⇨ **Total complexity = $O(n^2 logn)$**

POINTS ON MERGE SORT

1. Merge sort uses divide and conquer paradigm
2. The time complexity for two lists of size m and n to merge is O(m+n)
3. Questions can be asked like "What is the order of the array (given in question) in the second pass of two-way merge sort?"

LET'S MOVE ON TO OUR NEXT SORTING ALGORITHM

QUICK SORT ALGOIRTHM AND ANALYSIS

Just as we saw that the heart of the merge sort algorithm was the MERGE procedure, the heart of the quick sort algorithm is the PARTITION procedure. The name is given "quick sort" because for small values of n (100 etc.) quick sort has lesser time complexity than the merge sort. This algorithm also falls under "Divide and Conquer" category.

Before we jump to the sorting algorithm, let's see the PARTITION function first.

```
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

}
```

The partition function is quite easy to understand.

1. We take the last element of the array to be sorted in a variable (x) (*you can take any element; it doesn't need to be the last element of the array*)
2. The partition method is going to check from index p(first) to r-1(second last) and compare the elements on those index of the array with x
3. We take two pointers 'i' and 'j' pointing at index (p-1) and 'p' initially.
4. With each increment of 'j' the array element at A[j] will be compared with the element x
    a. If A[j] is bigger than x: We don't do anything
    b. If A[j] <= x:
        i. Increment i by 1 (i+1)
        ii. Replace A[i+1] with A[j]
5. Replace A[i+1] with x

This will repeat until the last but one index is reached by the pointer j. The essence is that, after the partition method completes its execution, all the elements to the left of x will be smaller than x and all the elements to the right of x will be greater than x.
The element x is already sorted now.

This procedure will be repeated recursively for the two new arrays (Array containing elements <= x and array containing elements > x).

The partition method runs for n-1 elements in the array, therefore the time complexity of it is O(n-1) or O(n).

This is pretty simple, now let's see the Quick Sort algorithm.

The heart of the Quick Sort algorithm is the PARTITION method.

Quick Sort Algorithm

```
QUICKSORT(A,p,r)
{
        if(p < r)
        {
                q = PARTITION(A,p,r)
                QUICKSORT(A, p, q-1)
                QUICKSORT(A, q+1, r)
        }
}
```

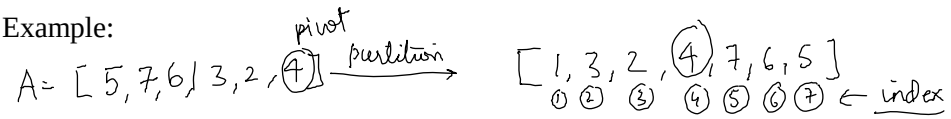
The index of the sorted element by the PARTITION algorithm is returned as q in the above algorithm.

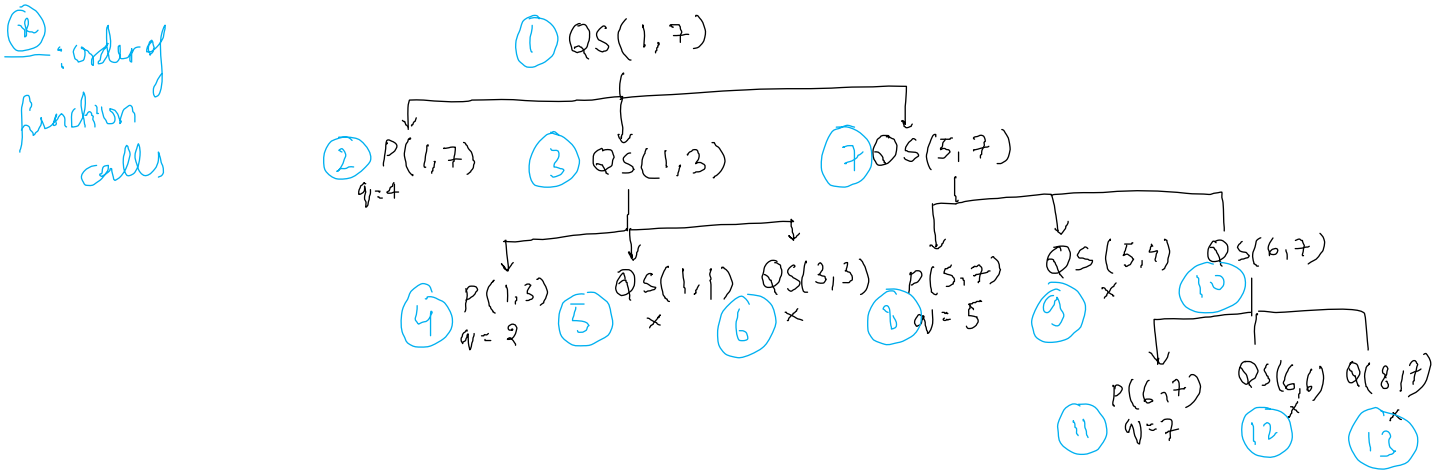
Note: In the merge sort algorithm, the arrays were divided into halves. In the quick sort algorithm the sub arrays can have different lengths depending on the value of 'q' returned by the PARTITION algorithm.

The element around which the partitioning is happening is called the PIVOT.

Let's take an example and elaborate the recursion tree.

Example:





Space Complexity of Quick Sort:

1. Determine the total number of function calls
2. Determine the order of function calls

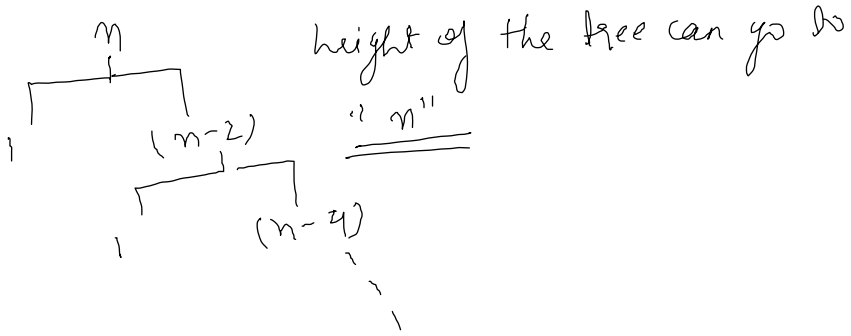
3. Use stack to evaluate the calls.

Total function calls = 13

This is not the size of the stack we require. **Infact the size of the stack will be the number of levels in the recursion tree.**

Case 1: If the input is divided into two equal halves, the height of tree will be logn and the space complexity will be O(logn) (best case)

Case 2: In case the input is divided into unbalanced arrays.



In this case the stack size can go to the size of the array, therefore in the **worst case** the space complexity will be O(n).

In **average case** the space complexity will be O(logn).

Time complexity of Quick Sort Algorithm:

```
QUICKSORT(A,p,r)
{
        if(p < r)
        {
                q = PARTITION(A,p,r)
                QUICKSORT(A, p, q-1)
                QUICKSORT(A, q+1, r)
        }
}
```

Let us assume that the time taken to sort 'n' elements is T(n). Let is consider the "Best Case" where the partition method is going to give the pivot element in the middle of the array every time. Then (look left)

$\Rightarrow T(n) = 2 * T(n/2) + O(n)$

$\Rightarrow$ through Master's theorem : $O(n \log n)$ "best case"

For the worst case:

```
QUICKSORT(A,p,r)
{
        if(p < r)
        {
                q = PARTITION(A,p,r)
                QUICKSORT(A, p, q-1)
                QUICKSORT(A, q+1, r)
        }
}
```

$T(n)$

$O(n)$

$T(0)$

$T(n-1)$

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

$T(n) = T(n-1) + cn$

back substitution

$= T(n-2) + c(n-1) + cn$

$= T(n-3) + c(n-2) + c(n-1) + cn$

$= c + c2 + c3 + \dots + cn$

$= O(n^2)$

Time and Space complexity for Quick Sort

Time Complexity	Best Case	Worst Case	Average Case
	Ω(nlogn)	**O(n^2)**	**Θ(nlogn)**

Space Complexity	Best Case	Worst Case	Average Case
	O(logn)	**O(n)**	**Θ(logn)**

Let's see some examples on quick sort.

1. If the input is in ascending order

$A = 1, 2, 3, 4, 5, 6$

pivot

$((1, 2, 3, 4), 5)$ (6) pivot

$\therefore T(n) = O(n) + T(n-1)$

↑ partition ↑ for array size $(n-1)$

$\Rightarrow T(n) = O(n^2)$

2. If the input is in descending order

$A = \{6, 5, 4, 3, 2, 1\}$

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

$= O(n^2)$

3. If the array contains same elements

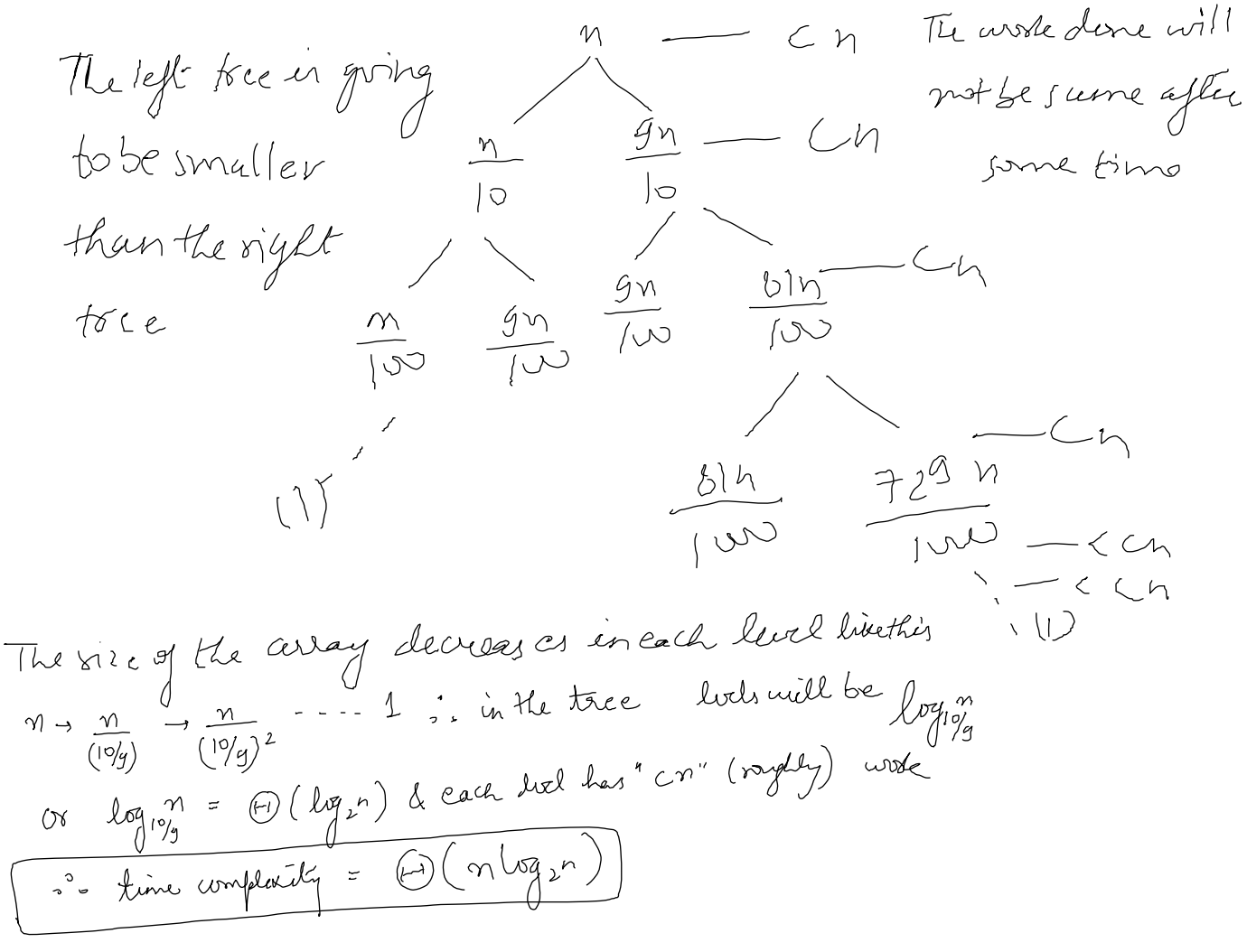
A = {2, 2, 2, 2, 2, 2}

Again: $T(n) = O(n) + T(n-1)$

$= O(n^2)$

Note: If the array is in ascending order/descending order/contains same elements, the time complexity is going to be worst case time complexity which is $O(n^2)$.

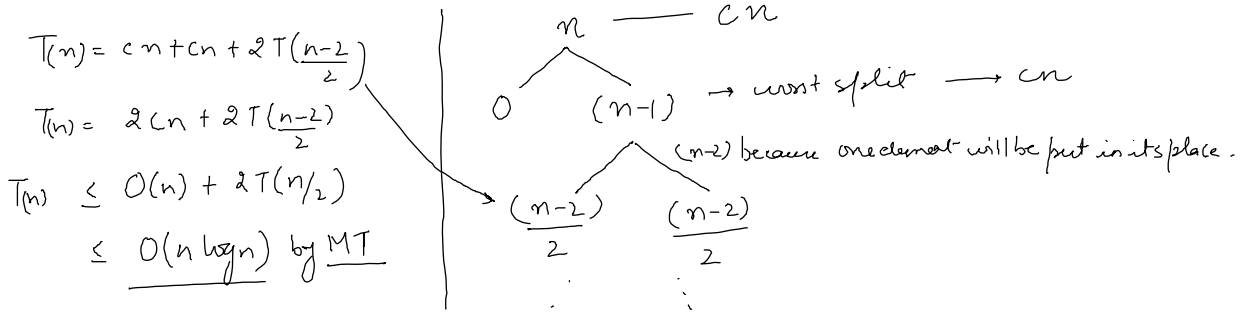
4. When the array is split in a ratio. Let's split the array in a ratio of 1:9



Therefore, even if the split is 1:9 we got the best case time complexity.

We can see that in quick sort, if the split is of 1:9, 1:99, 1:999, even then the best case time complexity Ω(nlogn) will be reached.

5. When the worst case and best case splitting is alternating

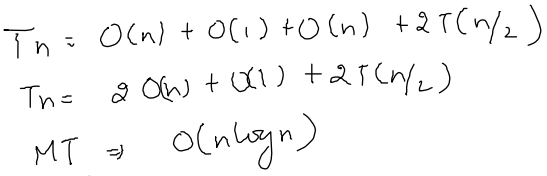


Therefore, even if the split is alternating between the one that produces best case and the one that produces worst case, we get the best case time complexity.

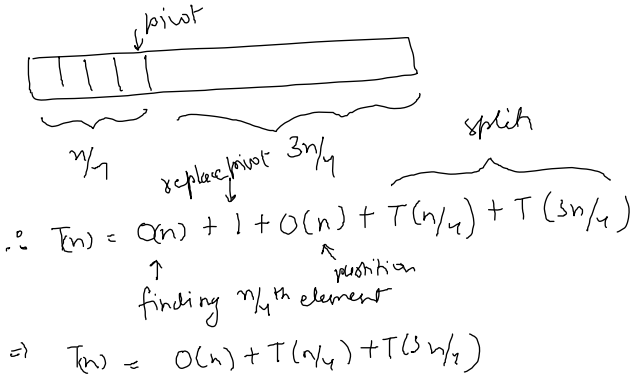
Let's see some questions on Quick Sort

Q1. The median of 'n' elements can be found in O(n) time. Which of the following is correct about the complexity of quick sort, in which median is selected as pivot?

1. Finding the median takes O(n)
2. For median to be pivot, I'm going to put it at the end of the array in the partition function. This takes O(1) time.
3. Now since the median is going to be at the center of the sorted array, we are essentially going to split the array in approximately two equal halves. For this we know the complexity is Θ(nlogn).
4. And we know that the partition algorithm takes O(n) time.

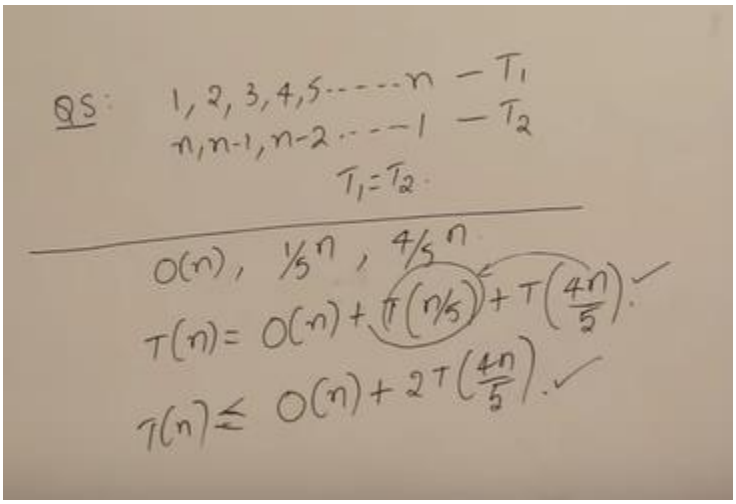


Q2. In quick sort, for sorting 'n' elements, the $(n/4)^{th}$ smallest element is selected as pivot using O(n) time algorithm. What is the worst case time complexity of quick sort?



This is essentially partitioning of the array in the ratio of 1:3. We have already seen that 1:9, 1:99, 1:999 has the time complexity of Θ(nlogn), so for a split of 1:3 also we are going to get the time complexity of

Θ(nlogn).


QS: 1, 2, 3, 4, 5 ..... n — T1
n, n-1, n-2 ..... 1 — T2
T1 = T2
O(n), 1/5 n, 4/5 n
T(n) = O(n) + T(n/5) + T(4n/5)
T(n) ≤ O(n) + 2T(4n/5)