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 + … + 2(n-1)

Or: 2(2-1) + 2(3-1) + 2(4-1) + … + 2(n-1)

Or: 2(1) + 2(2) + 2(3) + … + 2(n-1)

Or: 2 (1 + 2 + 3 + … + 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 + … + 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)

Space Complexity of Insertion Sort:

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**
	Ω(n)	**O(n^2)**	**Θ(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)
{

        $n_1$ = q-p+1
        $n_2$ = r – q
        Let L[1…$n_1$+1] and R[1 … $n_2$ + 1] be new arrays
        for( i=1 to $n_1$)

                L[i] = A[p + i-1]
        for( j=1 to $n_2$)

                R[j] = A[q + j]
        L[$n_1$ + 1] = ∞
        R[$n_2$ + 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.

1	5	7	8	2	4	6	9	
↑ 1	2	3	↑ 4	↑ 5	6	7	↑ 8	(Index)
p			q	q+1			r	

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.

1	5	7	8	∞
↑ i				

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

2	4	6	9	∞
↑ j				

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)

}
```

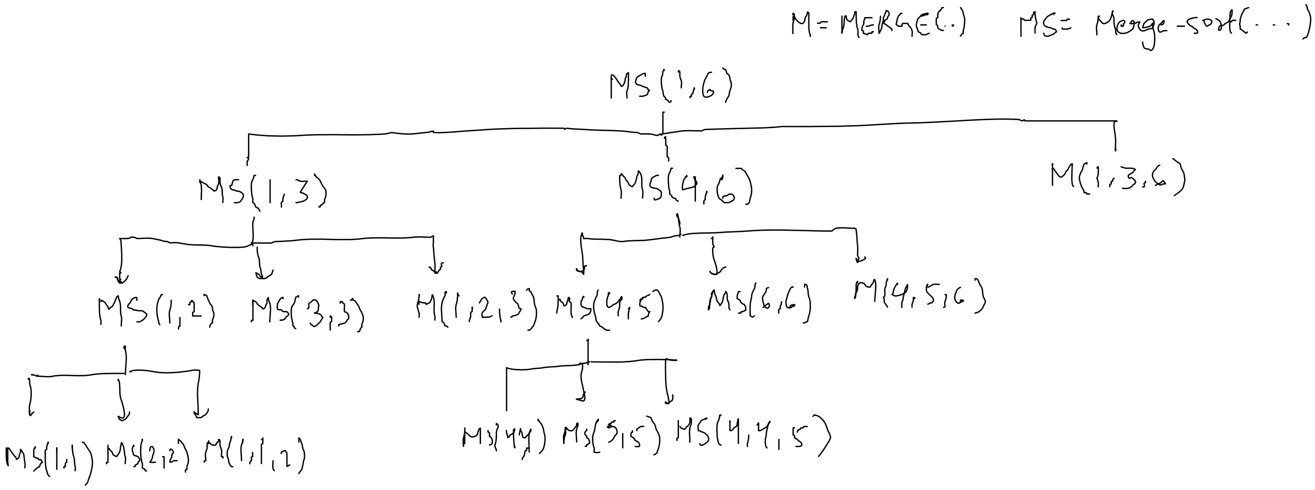
As said earlier, Mege Sort falls under the category of Divide and Conquer algorithms. In this sorting technique we divide the array to be sorted into single element array (single element array is already sorted) and move up from there to give us the final sorted array.

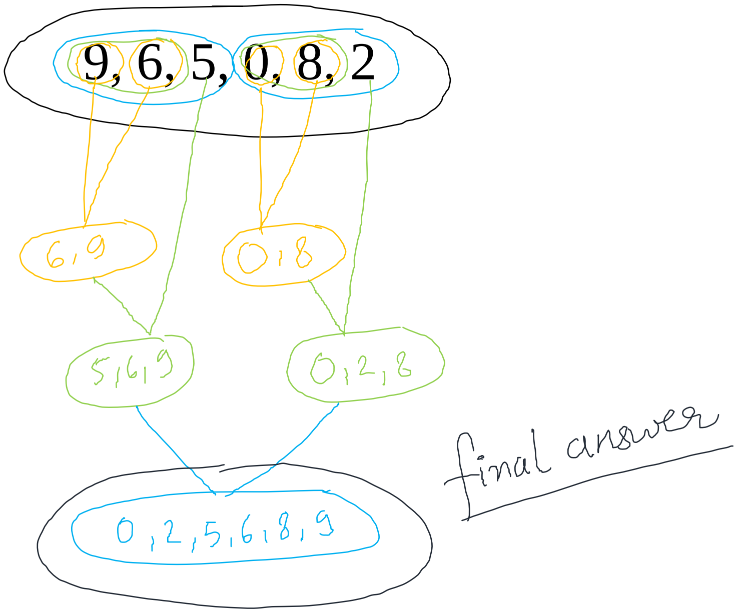
This is a recursive algorithm where each function call is going to call three functions:

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

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

$a = b^K$ & $p > -1$

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

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

$$\boxed{T(n) = \Theta(n\log n)}$$

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

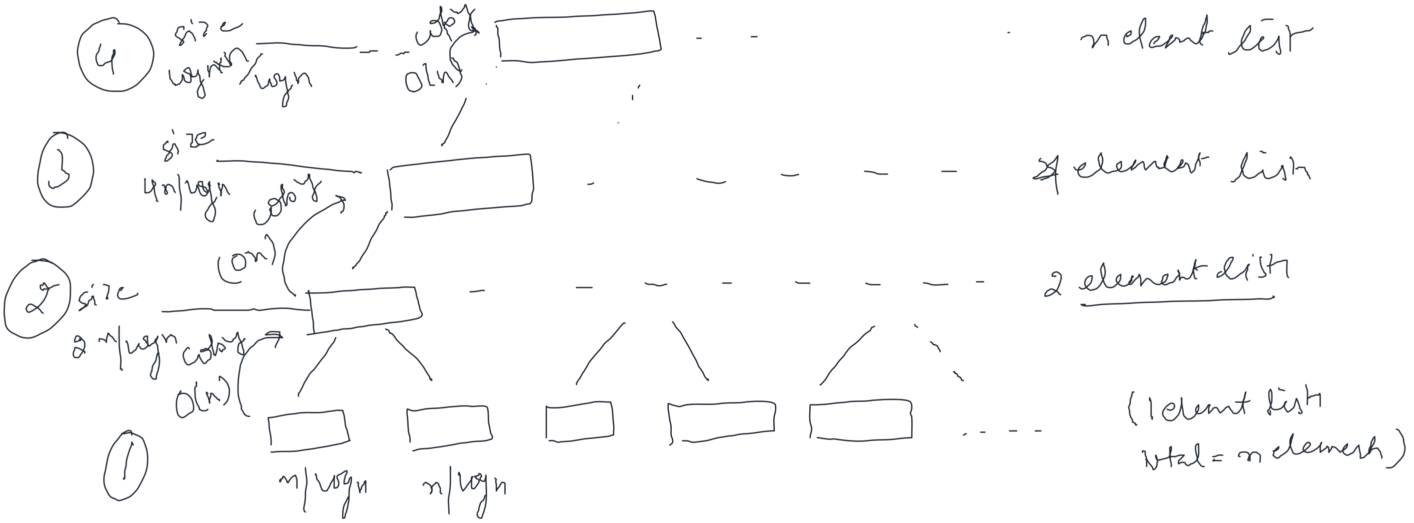
Each list is of size: n/logn

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 = log2 (or 1 climb)

" ④ to ③ " = log4 (or 2 climbs)

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

$$\boxed{\therefore \text{complexity} = O(n \log \log n)}$$

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^2logn)**

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.

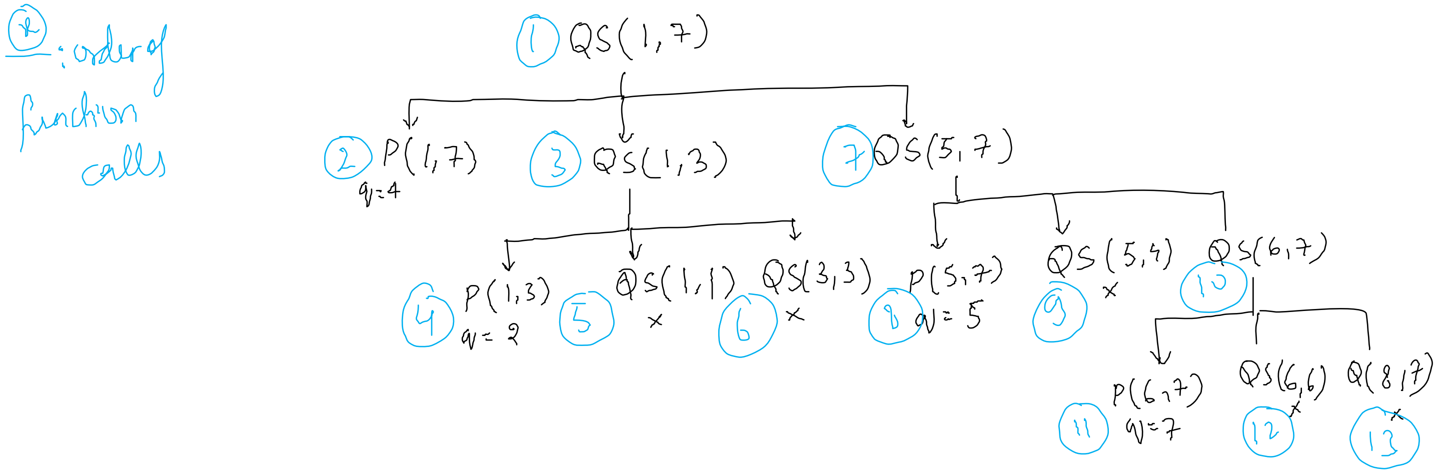
Let's take an example and elaborate the recursion tree.

Example:

pivot

$A = [5,7,6,1,3,2,\boxed{4}] \xrightarrow{\text{partition}} [1,3,2,\boxed{4},7,6,5]$

① ② ③ ④ ⑤ ⑥ ⑦ ← index



Space Complexity of Quick Sort:

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

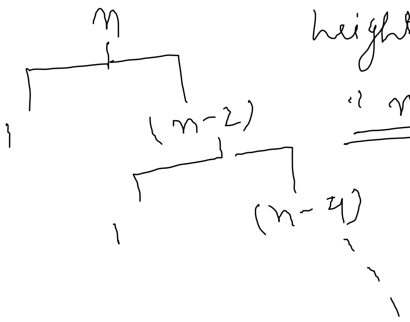
3. Use stack to evaluate the calls.

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.



height of the tree can go to "n"

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) ⟶ T(n)
{
        if(p < r)
        {
                q = PARTITION(A,p,r) — O(n)
                QUICKSORT(A, p, q-1) — T(n/2)  } ⟶ for "best case" scenario!
                QUICKSORT(A, q+1, r) — T(n/2)  }
        }
}
```

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)

⟹ $T(n) = 2 * T(n/2) + O(n)$

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

For the worst case:

```
QUICKSORT(A,p,r)                      → $T(n)$
{
        if(p < r)
        {
                q = PARTITION(A,p,r)          → $O(n)$
                QUICKSORT(A, p, q-1)          → $T(0)$
                QUICKSORT(A, q+1, r)          → $T(n-1)$
        }
}
```

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

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

back substitution

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

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

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

$$= \underline{\underline{O(n^2)}}$$

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

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

$$((1, 2, 3, 4), \overset{\text{pivot}}{\overset{\downarrow}{5}})\ \textcircled{6}\ \underline{\text{pivot}}$$

$$\therefore\ T(n) = \underset{\text{partition}}{\underset{\uparrow}{O(n)}} + \underset{\text{for array size } (n-1)}{\underset{\uparrow}{T(n-1)}}$$

$$\Rightarrow \underline{\underline{T(n) = O(n^2)}}$$

2. If the input is in descending order

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

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

$$= \underline{\underline{O(n^2)}}$$

3. If the array contains same elements

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

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

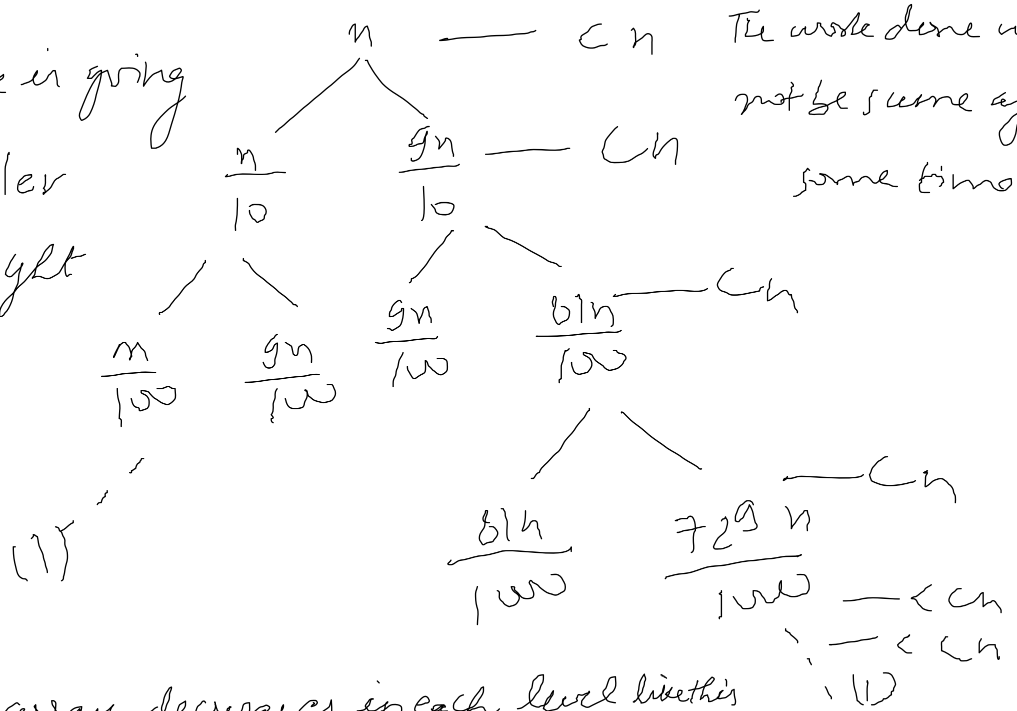
$= \underline{\underline{O(n^2)}}$

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

The left tree is going to be smaller than the right tree

The work done will not be same after some time



The size of the array decreases in each level like this

$n \to \frac{n}{(10/9)} \to \frac{n}{(10/9)^2}$ ---- 1 $\therefore$ in the tree levels will be $\log_{10/9} n$

or $\log_{10/9} n = \Theta(\log_2 n)$ & each level has "cn" (roughly) work

$\boxed{\therefore \text{ time complexity } = \Theta(n \log_2 n)}$

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.

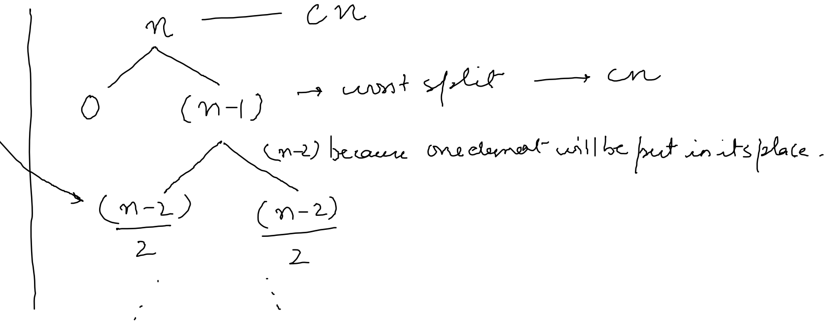
5. When the worst case and best case splitting is alternating

$$T(n) = cn + cn + 2T\left(\frac{n-2}{2}\right)$$

$$T(n) = 2cn + 2T\left(\frac{n-2}{2}\right)$$

$$T(n) \le O(n) + 2T(n/2)$$

$$\le \underline{O(n\log n)} \text{ by } \underline{MT}$$



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?

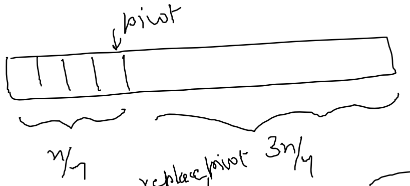
1. Finding the median takes O(n)
2. For median to be pivot, I'm going to put it at the end of the array in the partition function. This takes O(1) time.
3. Now since the median is going to be at the center of the sorted array, we are essentially going to split the array in approximately two equal halves.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

$$T_n = O(n) + O(1) + O(n) + 2T(n/2)$$

$$T_n = 2O(n) + O(1) + 2T(n/2)$$

$$\underline{MT} \Rightarrow O(n\log n)$$

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?

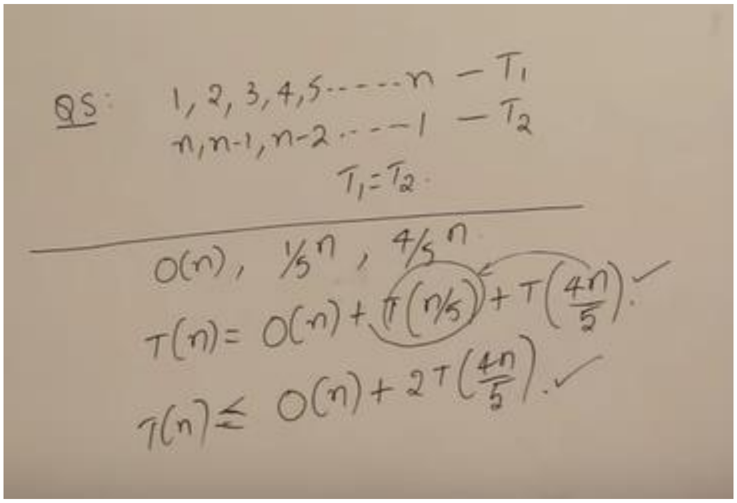


$$\therefore T(n) = \underset{\substack{\uparrow \\ \text{finding } n/4^{th} \text{ element}}}{O(n)} + \overset{\substack{\text{replace pivot} \\ \downarrow}}{1} + \underset{\substack{\uparrow \\ \text{partition}}}{O(n)} + \overbrace{T(n/4) + T(3n/4)}^{\text{split}}$$

$$\Rightarrow T(n) = O(n) + T(n/4) + T(3n/4)$$

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: $1, 2, 3, 4, 5 \cdots n$ — T_1

$n, n-1, n-2 \cdots 1$ — T_2

$T_1 = T_2$

$O(n), \ \frac{1}{5}n, \ \frac{4}{5}n$

$T(n) = O(n) + T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right)$ ✓

$T(n) \leq O(n) + 2T\left(\frac{4n}{5}\right)$. ✓

Let’s move on to our next segment.

INTRODUCTION TO HEAPS

Now we have seen some algorithms. Now all algorithms take an input and give some output. The way in which the input is given makes a lot of difference in the way the algorithm will take space or time. So depending on how the input is, the running time of the algorithm will change.

Let’s elaborate more with an example.

Let’s take a data structure as input to various algorithms. Then the time complexities for various operations is given in the table below.

Data structure	**Insertion complexity**	**Search complexity**	**Find minimum complexity**	**Delete minimum complexity**
Unsorted array	O(1)	O(n) (linear)	O(n)	O(n)
Ascending order Sorted array	O(n)	O(logn)(binary)	O(1)	O(n)
Unsorted Linked List	O(1)	O(n)	O(n)	O(n)

Now some of the algorithms need Insertion, finding minimum and delete minimum operations. We can see above that at least one of the operations in the array or linked list has a complexity of O(n).

This is where the data structure **Heap** comes in. This data structure is optimized for the operations: Insertion, finding minimum and deleting the minimum entity in it. Heaps are of two types: **max** and **min** heap. Let’s see their complexity for the same operations given in the above table.

Note: Depending on what kind of operations you want, choose the data structure.

Data structure	**Insertion complexity**	**Search complexity**	**Find minimum complexity**	**Delete minimum complexity**
Min Heap	O(logn)	O(n) (linear)	O(1)	O(logn)

Let’s take a deeper look into heaps.

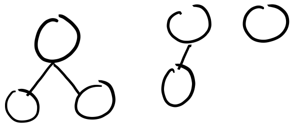
HEAP

Heaps can be implemented as a binary tree, 3-ary tree or in general an n-art tree.

For now let’s talk about the binary tree, later we will go to ternary or even n-ary tree implementation

Property of binary tree:

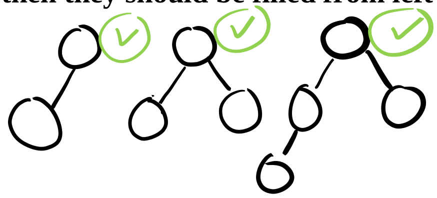
1. Every node can have at most two children



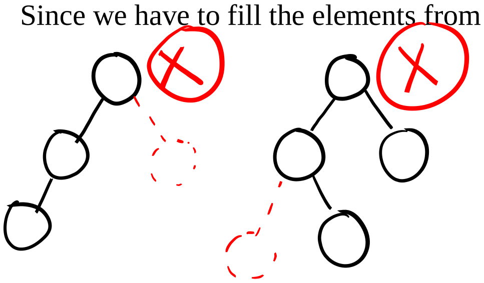
Every heap is an almost-complete binary tree. A complete binary tree is a binary tree where all of the nodes except the leaf nodes have 2-childrens. Now we said that heap is an almost complete binary tree.

Almost complete means that the leaves should be present only at the last level and last but one level. If all the leaves are present only in one level, then they should be filled from left to right.

Almost complete binary tree:



Since we have to fill the elements from left to right, the following binary trees can’t be used as heaps.

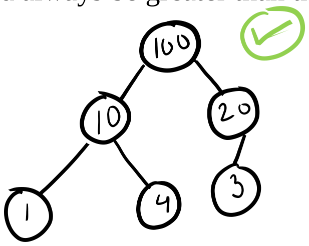


Since before going to the next level the previous level has to Be completely filled from L-to-R, and then the last level also Has to be filled from L-to-R, the Binary Trees on the Left can’t be used as heaps.

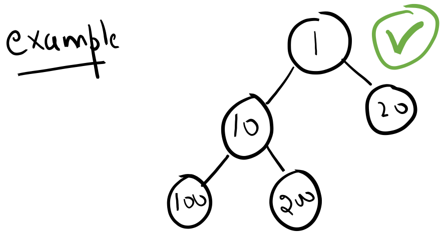
Let’s see examples of Heap.

1. **Max heap:** A max heap is a complete binary tree or almost-complete binary tree where the root element should always be greater than the elements in their child sub-trees.

example :



1. **Min heap:** A min heap is a complete binary tree or almost-complete binary tree where the root element should always be smaller than the elements in their child sub-trees.



Properties:
Max Heap Property: Maximum element should be at the root
Min Heap Property: Minimum element should be at the root

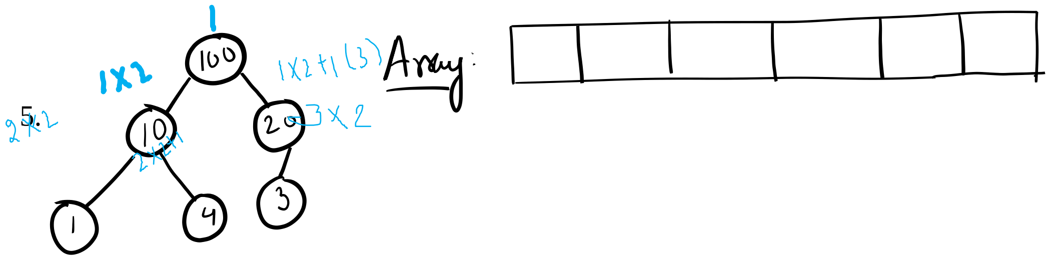
Now let's see how we are going to implement the heap.

<u>Implementing Heap</u>

Instead of making the linked list and then implement the binary tree which takes more space, we can implement the tree using an array. We are interpreting the tree as an array.

To fill the array with the elements of the tree:

1. First element will be the root of the entire tree
2. The index of left side will be calculated as (index of current root x 2)
3. The index of right side will be calculated as (index of current root x 2 + 1)
4. Example:
5.



So any complete binary tree could be placed inside an array which is completely filled without any gaps. *This is the main reason we need complete or almost complete binary trees.* If the tree is not almost complete, we might end up getting an array which has a lot of gaps in between.

If we are at a node, to find the parent of it just divide the index of the node with 2 and ceil it.

Parent of the node = ceil(index of the node/2).

If i is a node: Left child of i = $2i$

Right child of i = $2i + 1$

Parent of i = $\lfloor i / 2 \rfloor$

Multiplying i with 2 is: i << 2
Dividing i by 2 is: i >> 2

This is how you can put a binary tree in an array.

Now the questions that arise can be:

1. Given a max heap in the tree form what is the array representation of it?
2. Given an array, is the array a max heap or not?
3. If the array is not heap, till what element it is heap?

Before answering questions, we need to keep in mind:

1. Every leaf element is a heap
2. The length of the array is the total number of elements in the array. (A.length)
3. The heap size of the array is the number of elements till which array is heap. (A.heapsize)
4. If the array is in **asc** or **desc** order, then it is already in **min heap** and **max heap** respectively.

Let's use them in an example:

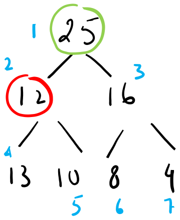
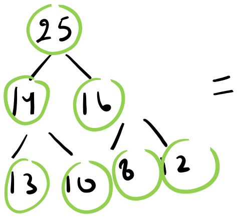
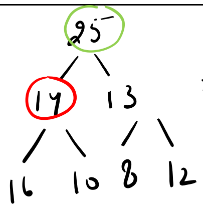
Example:

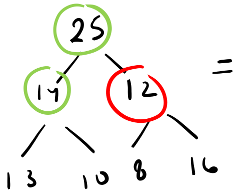
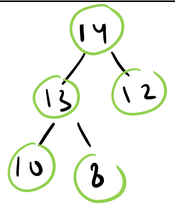
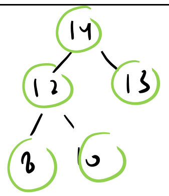
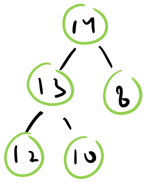
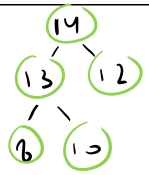
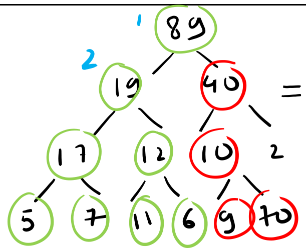
Q.1 Is the array given a max heap?

Q.2 If it is not max heap, till what elements it is heap? (heapsize)

Hint: If heapzise = index of last element in the array, then it is max heap.

For a given array, to get the almost complete binary tree, fill up the array with level order traversal.

Array A	A.length()	A.heapsize()
25, 12, 16, 13, 10, 8, 14	7	 1 25 2 12 16 3 4 13 10 8 4 5 6 7 = 1
25, 14, 16, 13, 10, 8, 12	7	 = 7 (maxheap)
25, 14, 13, 16, 10, 8, 12	7	 = 1

25, 14, 12, 13, 10, 8, 16	7	 = 2
14, 13, 12, 10, 8	5	 = 5 (maxheap)
14, 12, 13, 8, 10	5	 = 5 (maxheap)
14, 13, 8, 12, 10	5	 = 5 (maxheap)
14, 13, 12, 8, 10 ⇒ given a set of numbers the heaps can have various forms. (it may not be unique)	5	 = 5 (maxheap)
89, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70	13	 = 2

One may say that to make a max heap, the given elements should be sorted in decreasing order first then heap can be constructed. But, the sorting will take O(nlogn) time and that will be more as max heaps can be constructed in O(n), (we will see the algorithm later).

Now let’s go to the algorithms to construct a heap, insert an element in a heap, delete an element in a heap and finally heap sort.

MAX HEAPIFY ALGORITHM AND COMPLETE BINARY TREE

Before going on to the algorithms, let’s see the properties of **complete binary tree**.

1. Height of the node: The number of the edges on any path from that node to the leaf node such that the number of edges is maximum.
2. Height of the tree: The height of the root is the height of the tree
3. Given a height ‘h’, the **maximum number of nodes** in a complete binary tree: $2^{h+1} - 1$
4. Given the height ‘h’, the **maximum number of nodes** in a complete 3-ary tree: $\frac{3^{h+1}-1}{2}$
5. Given the height ‘h’, the **maximum number of nodes** in a complete n-ary tree: $\frac{n^{h+1}-1}{n-1}$
6. Given any complete or almost complete binary tree with ‘n’ nodes, the height of the tree: $\lfloor \log n \rfloor$ **Therefore, we can say height of any heap = Θ(logn)**

Let’s make our max heap now.

MAX HEAP CREATION FROM A GIVEN ARRAY (MAX-HEAPIFY FUNCTION)

Here we are working in the context of max heap. For min heap, the algorithm will have to be just reversed.

Now given an array, if I want to create a max heap, one thing I can do is:

1. Sort the array: An array sorted in descending order is already a max heap. Here the time taken will be O(nlogn)
2. Can we do something better to reduce the time complexity?

Before we answer the question, let’s see some more properties of complete/almost complete binary trees.

1. Where will the leaves start in a tree?
    a. In a complete binary tree, the last level will all have leaves, therefore, if leaves are starting from an index ‘i’ then all the indices after ‘i’ will be leaves. This property is also followed in almost complete binary tree.
2. In a complete/almost complete binary tree, the nodes from $\lfloor n \,/\, 2 \rfloor + 1 \; to \; n$ will all be leaves.

Let’s construct our max heap.

Working of MAX_HEAPIFY

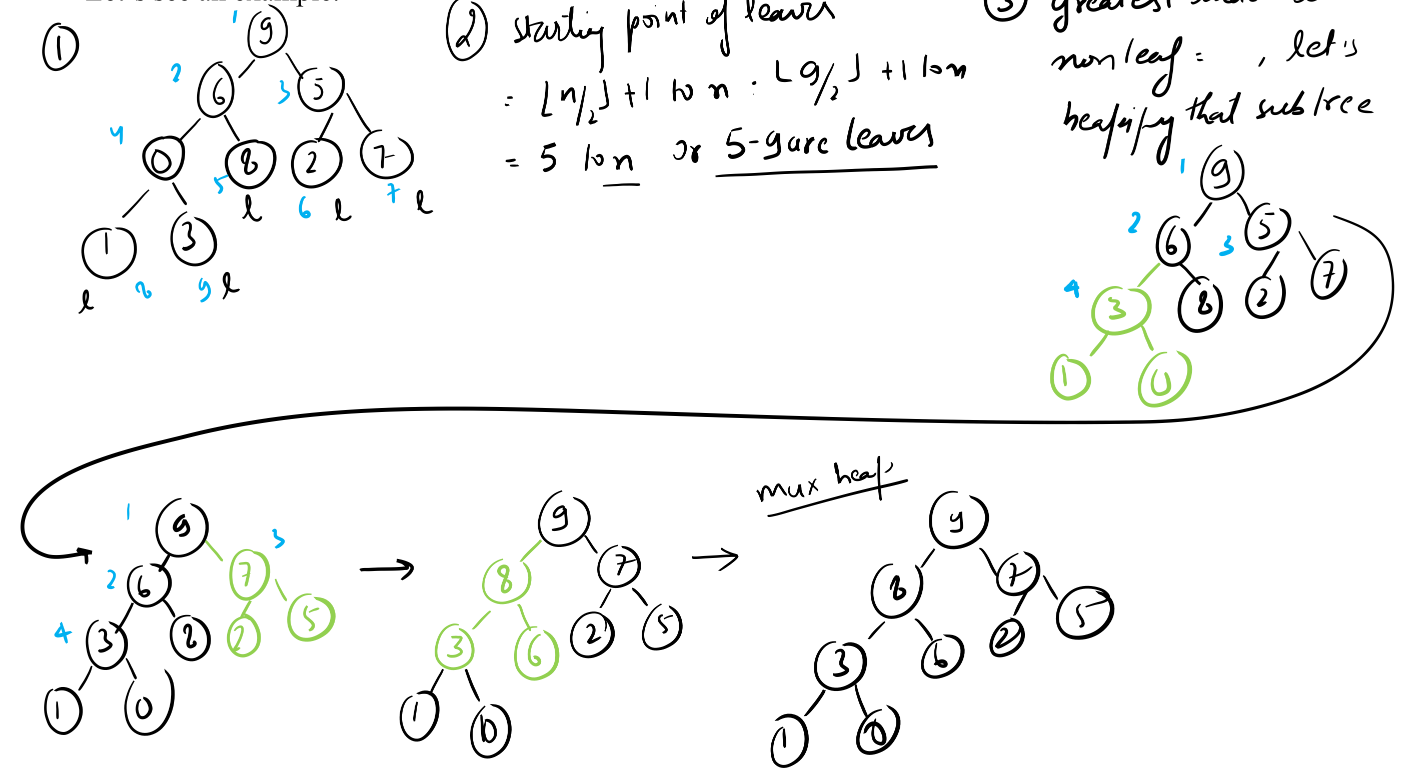
1. From the given array create a complete or non-complete binary tree in level order traversal.
2. Find out the elements who are the leaves ($\lfloor n \,/\, 2 \rfloor + 1 \; to \; n$). *All leaves are already max heap*

3. Find the greatest index in the tree which has a non-leaf element.
4. At this greatest index, check the children, if they contain the max number in its subtree, replace it with that number so that the root of this subtree has greatest number to become a max-heap.
5. Repeat this process for indices from the greatest index in the tree which is a non-leaf element to the root of the tree.

Let’s see an example.

Construct max heap from : 9 , 6 , 5 , 0 , 8 , 2 , 7 , 1 , 3

● = index



② starting point of leaves
= ⌊n/2⌋ + 1 to n : ⌊9/2⌋ + 1 to n
= 5 to n or 5-9 are leaves

③ greatest index which is non leaf : , let's heapify that subtree

This is the way to construct a max-heap. The entire idea of creating a max heap is:

“Given an array A with an index i, if the left subtree (can also be the leaf element) and the right sub tree are max heaps, then make the root node also a max heap”.

Let’s see the algorithm.

```
MAX_HEAPIFY(A, i)
{
        l = 2i;
        r = 2i+1;                                        to check if left child is there or not
        if(l<= A.heapsize and A[l]>A[i])
                largest = l;
        else largest = i;                                checks existence of right child
        if(r <= A.heapsize and A[r] > A[largest]
            largest = r;
        if(largest ≠ i)
            exchange A[i] with A[largest]
            MAX_HEAPIFY(A, largest)
}
```

Time complexity for max-heapify:

For worst case, we may need to get the root element to the bottom of the tree (height of tree), therefore, to get to the bottom of the tree complexity = O(logn)

Space complexity for max-heapify: Since we are not using any extra space for running the algorithm that depends on the input size, the only extra space will be taken by the stack for storing recursion calls. In the worst case space complexity will the height of the tree.

Therefore, space complexity = O(logn)

Note: 'n' is the number of nodes in the subtree for which 'i' is the root.

Let's see the BUILD_MAX_HEAP function and complete the creation of our max heap.

MAX HEAP CREATION FROM A GIVEN ARRAY (BUILD_MAX_HEAP FUNCTION)

```
BUILD_MAX_HEAP(A)
{
        A.heap_size = A.length;
        for(i = $\lfloor A.length/2 \rfloor$ down to 1)
                MAX_HEAPIFY(A, i)
}
```

The condition for the "for" loop is from A.length/2 to 1 is because we want to run the max_heapify algorithm only on the nodes which are not heaps and that too in descending order of the nodes which are not heaps (the leaf nodes are all heaps already, therefore, they will be A.length/2 + 1 to A.length).

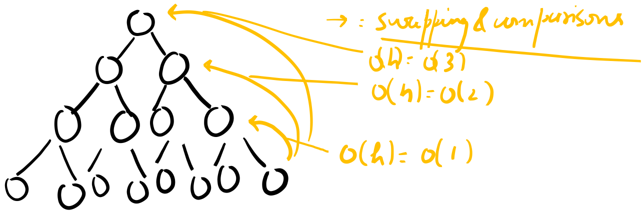
Let's talk about another property of Complete Binary Tree:

Q. How many nodes of height "h" can be present in a complete binary tree?

A. If we have 'n' nodes in the tree, then at height "h" we have $\left\lceil \frac{\boldsymbol{n}}{\boldsymbol{2^{h+1}}} \right\rceil$ nodes at height "h"

Let's now understand the time taken by the algorithm which makes the heap from an array. This is a bit complex so pay attention.

If I have a complete binary tree like this and applying MAX_HEAPIFY:



In general, if I apply Max Heapify on any Node the time taken will be O(h) or O(height of that node).

Let's say we are applying max_heapify on all the nodes including leaves, then the total work done at a particular height will be

Work done at height h = number of nodes in that height x O(h)

Number of nodes in height h = $\left\lceil \frac{\boldsymbol{n}}{\boldsymbol{2^{h+1}}} \right\rceil$

Work done for all the nodes in the tree = $\sum_{h=0}^{\log n} \left\lceil \frac{\boldsymbol{n}}{\boldsymbol{2^{h+1}}} \right\rceil . O(h) = \sum_{h=0}^{\log n} \left\lceil \frac{\boldsymbol{n}}{\boldsymbol{2^{h}.2}} \right\rceil . O(h)$

Now O(h) can be written as: 'c.h'

n, c and h are constants, then we can write: $\frac{Cn}{2} \cdot \sum_{h=0}^{\log n} \left\lceil \frac{h}{2^h} \right\rceil$ <= $\mathrm{O}\left(\frac{Cn}{2} \cdot \sum_{h=0}^{\infty} \left\lceil \frac{h}{2^h} \right\rceil\right)$

Now $\sum_{h=0}^{\infty} \left\lceil \frac{h}{2^h} \right\rceil = 2$, Therefore $\mathrm{O}\left(\frac{Cn}{2} \cdot \sum_{h=0}^{\infty} \left\lceil \frac{h}{2^h} \right\rceil\right) = \boldsymbol{O(n)}$

Therefore, the time taken to build a max heap is O(n) if the entire array is not at all in heap.

Space Complexity of BUILD_MAX_HEAP

When looking at the algorithm above, the extra space is only taken by the MAX_HEAPIFY algorithm which is O(logn), therefore, Space Complexity of BUILD_MAX_HEAP is O(logn) where n is the root of the entire tree.

Now, let's see how we are going to apply this heap data structure in order to do the *heap sort.*

HEAP SORT (preliminaries)

Before we jump to sorting, let's see some operations on heap.

1. **Delete the max element from max heap**
    a. Take the first element as the max, delete it
    b. Now there is a space to fill, so take the last element from the heap and put in the first position
    c. Check if the new tree (almost complete binary tree) is a max heap or not,
    d. If not, start the MAX_HEAPIFY routine

    The algorithm looks like below

```
HEAP_EXTRACT_MAX(A)
{
O(1)        If heap_size < 1   — check if heap has atleast 1 element
                    error "heap underflow"
O(1)        Max = A[1] ← since max heap so first element is largest
O(1)        A[1] = A[A.heap_size] ← replace the first element with last element of heap.
                                     heapsize = last index since already max heap is given
O(1)        A.heap_size = A.heap_size – 1   reduce heap size by 1
O(logn) ∵ we are always taking top element ∴ O(height) & height = O(logn)
            MAX_HEAPIFY(A, 1)   run max-heapify on reduced tree
            Return Max   return max
}   O(1)
```

Time complexity = O(logn)

Space Complexity = O(logn)

2. **Given a max heap and a node index, increment the value of the node.**
    a. Replace the key (the new value) with the element on the index
    b. Check if the subtree is in max heap
    c. If not, replace the elements and move up to the root until the tree is in max heap.

    The algorithm looks like below:

```
HEAP_INCREASE_KEY(A, i, key)
{
        If key < A[i]
                error

        A[i] = key
        while(i>1 and A[i/2] < A[i])
                exchange A[i] and A[i/2]
                i = i/2

}
```

check if key is not less than original value since it is an increment function

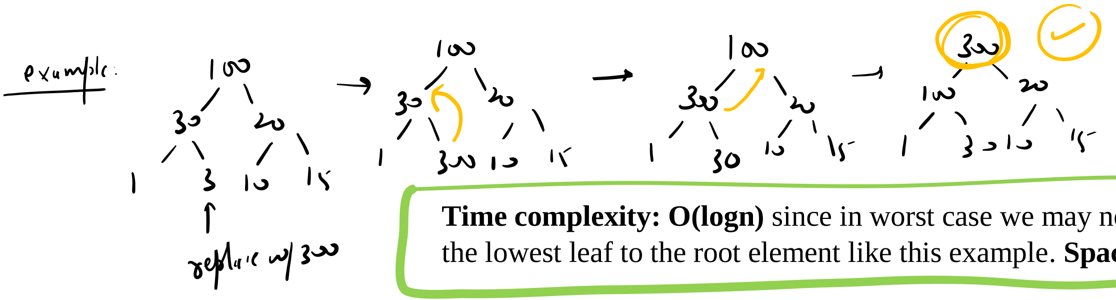
replace the position w/ key element

go until root of tree only

parent element < child?

exchange them

check next parent



Time complexity: O(logn) since in worst case we may need to go from the lowest leaf to the root element like this example. **Space = O(1)**

3. **Given a max heap and a node index, decrease the value of the node.**
    a. Replace the key (the new value) with the element on the index
    b. Now the left and right subtrees to this node will be max heaps so,
    c. Run MAX_HEAPIFY function from this node to max heapify it.

    Time complexity = Time to run max_heapify

    In worst case, we may need to run max heapify from the top element in the complete binary tree.
    Therefore, **Time complexity = O(logn)** and **Space Complexity = O(logn)**

    Note: In a max heap if you decrement an element, it is equivalent to calling Max_Heapify fuction so the complexity is judged by max_heapify function.

    Let's see how to insert an element in the max heap.

4. **Insert an element in max heap:** Whenever we want to insert an element, we insert it in the last position which is available in the heap.
    a. Insert the element in the last position of the heap
    b. Compare it with the parent
    c. If it is bigger than the parent, replace the parent with this element
    d. Repeat b and c until the tree is max heap

    Time complexity: In the worst case the element might have to travel from the leaf to the root.

⇨ Time complexity = **O(logn)**
⇨ Space complexity = **O(1)** (no extra space is taken)

SUMMARY OF OPERATIONS AND THEIR COMPLEXITY

We learnt about heap because for problems which require insertion, finding max and stuff, heap is a good data structure. Well it will not work for all of the operations (for ex. For deleting a random element). Let’s see where heap will be a good data structure and where we may need to analyze other algorithms than heap.

Green means heap is optimal for such problems

Red means heap is not optimal for such problems

Problem:	Find max	Delete max	Insert an element	Increase key	Decrease Key	Find min	Search Random	Delete Random
Complexity of Max Heap:	O(1)	O(logn)	O(logn)	O(logn)	O(logn)	O(n)	O(n)	O(n)

Now, let’s see how to implement a sorting algorithm using heap data structure.

HEAP SORT ALGORITHM AND ANALYSIS

```
HeapSort(A)
{
        BUILD_MAX_HEAP(A)
        for(i = A.length down to 2)
                exchange A[1] with A[i]
                A.heap_size = A.heap_size -1
                MAX_HEAPIFY(A, 1)
}
```

O(n)

runs ≈ n times

$nO(1)$

$nO(1)$

$nO(\log n)$

∵ in the end only one element will be remaining so no need to run MAX_HEAPIFY on that :)

The gist is that, “At every point we are going to take the largest element in the heap and write it in the end, then we are not going to see that element for the remaining part of the algorithm. Then we construct the heap from the remaining elements and repeat this process.”

Time Complexity: We are running the for loop ~ n times and MAX_HEAPIFY function is taking O(logn) time.

Therefore, **Time complexity for heap sort = O(nlogn)**

One can argue that the MAX_HEAPIFY will run on a smaller tree every time, but until one entire level is not removed the function will run O(number of leaves x logn) = O(n/2 x logn)= O(nlogn). This is not *same as that of BUILD_MAX_HEAP where the complexity depends on the height of the node. Therefore the complexity can't be O(n).*

QUESTIONS ON HEAP

Question 1. In a heap with 'n' elements with the smallest element at the root, the 7th smallest element can be found in time:

a. O(nlogn)
b. O(n)
c. O(logn)
d. O(1)

Answer 1: Since we are not taking smallest but 7th smallest element, therefore time can't be O(1).

To solve this we will have to extract first six elements from the heap which will take 6 x O(logn) time.

Find the 7th element will take O(1) time

Putting back 6 extracted elements will take 6xO(logn)

Total time complexity = 6O(logn) + O(1) + 6O(logn) = **O(logn) Therefore option (c) is the correct choice.**

Question 2: In a binary max heap containing 'n' numbers, the smallest element can be found in what time?

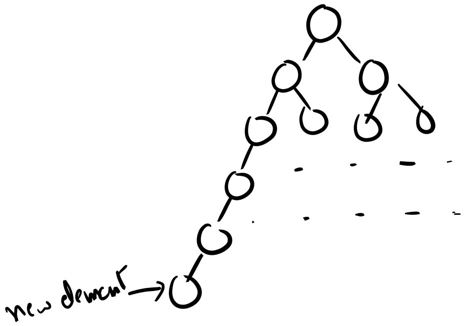
a. O(loglogn)
b. O(n)
c. O(logn)
d. O(1)

Answer 2: The minimum element will be present in the leaf nodes. In these leaf nodes we need to search for the smallest element.

Time complexity = Search for smallest element from $\left\lfloor \frac{n}{2} \right\rfloor + 1\ to\ n$ which will take **O(n/2)** or **O(n)** time.

Question 3: Consider the process of inserting an element into a max heap. If we perform a binary search on the path from new leaf to root to find the position of newly inserted element, the number of comparisons performed are?

Let's say the tree is like this:



To find the position of newly inserted element we need to apply binary search on the path from new leaf to root.

Path from leaf to root will have **log(n)** elements.

Binary search on **n** elements takes **O(logn)** time

Binary search on logn elements will take O(loglogn) time.

Therefore answer will be : O(loglogn)

If in the question it was asked "the element will be inserted in what time?". Then O(loglogn) will not be the answer as for insertion we need to run max_heapify. In that case the time complexity would have been

O(logn)

Question 4: We have a binary heap on 'n' elements and wish to insert 'n' more elements (not necessarily one after another) into this heap. The total time required for this is:

a. O(logn)
b. O(n)
c. O(nlogn)
d. O(n^2)

We know that inserting an element in a max heap taken O(logn) time and if we are inserting 'n' elements it will take O(nlogn) time.

The above will only happen if we insert elements in an order of ONE AFTER THE OTHER.

The question says that one after the other order is not necessary, so I can consider the already make heap that it is not in heap and add n elements at the end of the array.

The extended array will have **2n** elements.

I will run BUILD_MAX_HEAP on this array which will do the job in O(2n) time.

⇨ **The correct choice is (b) O(n)**

Let's move on to the next sorting algorithm.

BUBBLE SORT

Let’s see the algorithm for Bubble sort.

```
BUBBLE_SORT(int a[], n)
{

        int swapped, i, j;
        for(i = 0, i<n, i++)

        {
                swapped = 0;
                for(j=0;j<n-i-1;j++)

                {

                        If(a[j] > a[j+1]
                                swap(a[j], a[j+1]);
                                swapped = 1;

                }

        }

        if(swapped == 0)
                break;

}
```

0 1 2 3 4
A = {1, 5, 0, 6, 8}
j = 0 - 3
A = {1, 0, 5, 6, 8}
j = 0 - 2
A = {0, 1, 5, 6, 8}

Analysis of Bubble sort:

Now we can look at the complexity of the algorithm by looking at the number of swaps required, or even the number of comparisons required.

Worst case:

In the worst case scenario for an array which contains ‘n’ elements we have to do:

(n-1) + (n-2) + (n-3) + … + 1 comparisons (**worst case input will be an array sorted in descending order).**

This sum = (n-1)(n-2)/2 = **O(n^2)**

Best Case:

In the best case, the array will already be sorted and thus after the first iterations only (n-1) comparisons will be required.

Therefore, best case time complexity = Ω(n-1) or **Ω(n)**

<u>If we are stopping in the middle:</u>

Suppose we found that the array is sorted after 2 iterations doing (n-1) and (n-2) comparisons respectively in each iteration. Then the time complexity = O(n-2+n-2) = O(n) again. So, we can say that the time complexity will be O(kn) for all the middle cases. For the worst case, k becomes ‘n’ giving O(n^2) complexity.

Summary for Bubble Sort:

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

Let’s look at another sorting algorithm

BUCKET SORT

Bucket sort is applied for problems like below:

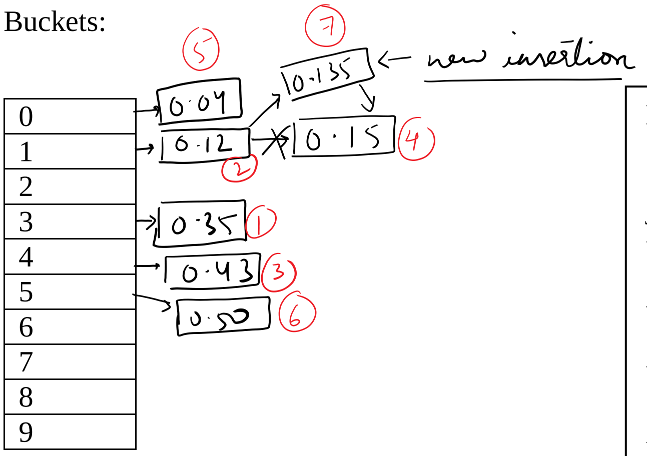
Sort a large set of floating point numbers which are in the range 0.0 to 1.0 and are uniformly distributed across the range.

Let’s take the numbers: 0.35, 0.12, 0.43, 0.15, 0.04, 0.50, 0.132

1. We take 10 “buckets” determined by the radix of the number.
2. Put the numbers to be sorted in the correct bucket (determined by radix)
3. If there are more than one numbers in a bucket to be inserted, use insertion sort for the correct placement
4. Read the bucket from top to bottom to get the sorted elements.

Let’s take a look at the sorting problem above.

Buckets:



o = insertion sequence in bucket

Best case:

If all of the elements are uniformly distributed, then we may just need to add them to the bucket and then take them out of the bucket (no insertion sort because no two elements are in the same bucket). Time complexity will be O(n) + O(n) = O(n) for best case. Space complexity = O(buckets + elements) = O(k+n)

Worst case:

All the buckets may get n/k elements and in these elements we may have to apply insertion sort. Therefore, worst case time complexity = O((n/k) * n) = O(n^2)

Summary for Bucket Sort:

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

Space Complexity = O(n+k) where k is the number of buckets and n is the number of elements to be sorted.

Let’s see our next sorting algorithm

COUNTING SORT

There is a restriction with counting sort. It can’t be applied like other sorting algorithms. The restriction is that *the keys or the numbers we have to sort, should always occur in a particular range.*

The restriction should be pre-defines.
Example: The numbers we have to sort is always between 1-5 or 10-1000 etc.

Let’s sort: 2, 2, 3, 4, 1, 5, 1, 5 Range: 1-5

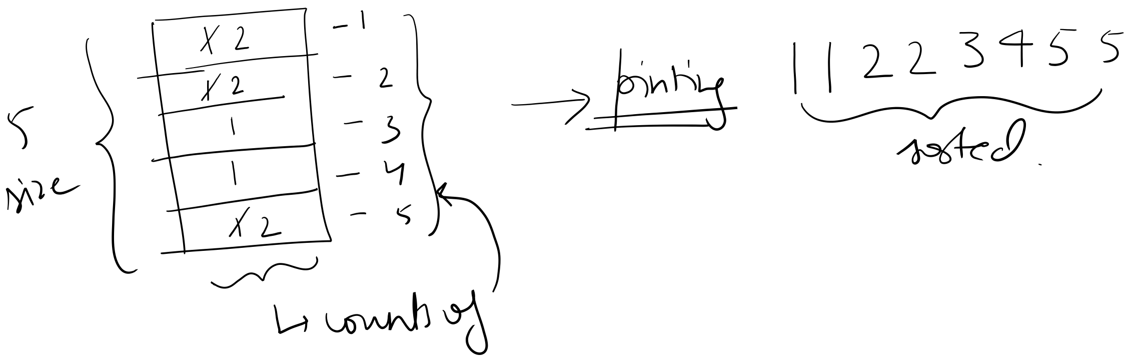
Here the keys are: 1, 2, 3, 4, and 5.

The gist is to use any data structure to store the ‘count’ of these keys.

1. Initialize a data structure (array or dictionary) with the size of number of keys
2. Traverse through the given list of numbers to be sorted
3. For each number seen, increment the count of it in the data structure we initialized until the list in question is completely traversed
4. Print each key the number of times it was seen to get the sorted elements.

Above question solution:

Since the keys range from 1-5 I am initializing an array to store their counts.



Analysis of counting sort:

Let's say the number of keys to be sorted = n

The range of these keys = k

We have to scan both of these lists one time, therefore **time complexity = O(n+k)**

Space Complexity: The extra space required is for the new data structure we are initializing to hold the counts. This space complexity will depend on the range of the keys

Therefore, **space complexity = O(k)**

Disadvantages:

1. *The restriction itself is a disadvantage*
2. *The space complexity will increase with an increased range.*
    a. *If we want to sort 1, 1000, 2, 4 these four elements the space of the data structure will be at least 1000 or O(1000)to store the count.*

RADIX SORT

Let's understand radix sort with an example and then move on to the algorithm. Let's say the numbers we have to sort is like this:

804
26
5
64
52
1

We are using base 10 here (which is decimal).

1. Determine the number of the digits in the largest number in the given list to be sorted, let's say it is 'm' digits
2. Convert all the numbers into 'm' digit number (in this example 1 will be written as 001)
3. Form the least significant digit to the most significant apply any stable sorting algorithm.
   a. Take the units place and arrange numbers according to the result. If two numbers are same, then arrange them in the way they occurred in the original sequence.
   b. Take the elements sorted by units place in point a. above and sort them using the tens place.
   c. Repeat this process until you end up sorting the numbers using most significant digit.
4. You'll get your sorted elements after doing this procedure.

Let's sort the elements:
804
26
5
64
52
1

final sorted elements

Original Sequence	Conversion to max digits	Units place sorting	Tens place sorting	Hundreds place sorting
804	804	001	001	001
26	026	052	804	005
5	005	804	005	026
64	064	064	026	052
52	052	005	052	064
1	001	026	064	804

RADIX SORT ALGORITHM

```
RADIX_SORT(A, d)
{
        //Each key in A[1…n]is a d-digit integer
        //Digits are numbered 1 to d from right to left

        For i=1 to d do:

                Use a stable sorting algorithm to sort A on digit i

}
```

A = the array containing the elements to be sorted, d= the maximum number of digits.

Analysis of Radix Sort

The intuition for the algorithm:

1. When we do the sorting based on the unit's digit, all the elements which have only 1 digits get sorted in their correct position.
2. When we do the sorting based on the tens digit, all the elements which have 2 digits (in the original sequence) get sorted in their correct position.
3. When we do the sorting based on the hundreds digit, all the elements which have 3 digits(in the original sequence before appending zeros) get sorted in their correct position.

Time complexity: The time complexity depends on the for loop that we have at the heart of the algorithm.

We will repeat the for loop depending on the number of digits present in every number.

In any number system of base b the maximum digits that you can have is b. Therefore, at any place (units, tens or hundreds) in the base 10 system we can have only 10 possible digits (0-9). Therefore, counting sort can be used as the stable sorting algorithm here.

Now, counting sort has to be applied 'd' times.

Complexity = d*O(n+b) where n is the number of keys in the input and b is the range of the input.

Now d is the number of digits in the largest number present in our input. Let's say that the largest number present in the input is l.

Therefore, $d = \log_b l$

Time complexity = $\log_b l$.O(n+b)

Let's assume that the largest number is of the order of n^k.

Then time complexity = $\log_b n^K\ (O.(n+b))$ = $k \log_b n\,.(O(n+k))$

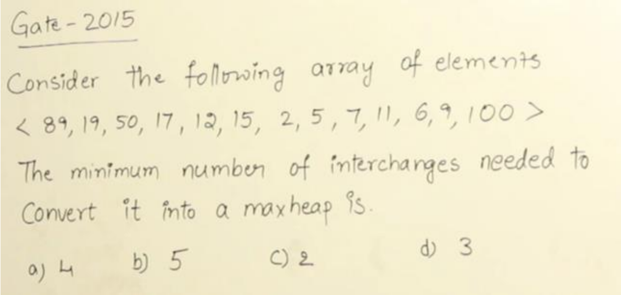
K is a constant, so time complexity = O($n \log_b n$) (worst case).

For the algorithm to give O(n) time complexity, the b and n should be same.

Space complexity: The extra space will be required by the counting sort algorithm to store the count of the numbers. This extra space depends on the base of the numbers.

Therefore, **space complexity = O(b).**

QUESTIONS



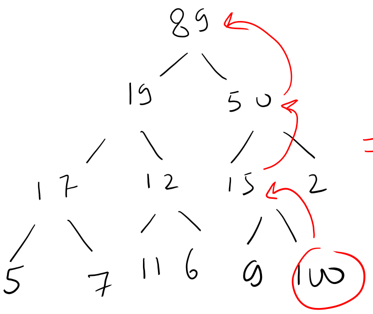
Gate-2015

Consider the following array of elements

< 89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100 >

The minimum number of interchanges needed to Convert it into a maxheap is.

a) 4 b) 5 c) 2 d) 3



⇒ 3 interchanges