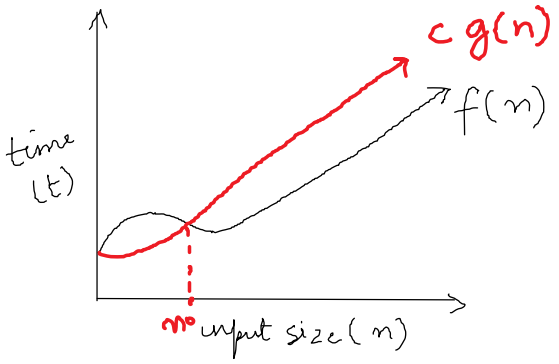
Algorithms: Space and Time Complexity

INTRODUCTION TO ASYMPTOTIC NOTATION

There will be many solutions to a given problem each of which will be given by an algorithm. Now, we want that algorithm which is going to give me the result quickest while consuming not much of the space (memory). Therefore, design and analysis of algorithms is the subject to design the various algorithms for a problem and analyze them for the best possible algorithm to choose.

Before proceeding, we have some notations to understand. These are called asymptotic notations.

Let's say we have a function f(n) and as the n increases the rate of growth of time increases in a certain way.



1. **Big-Oh (O)**: Let's find another function 'cg(n)' (above graph) in such a way that the after it gets an input n_0 the value of this function is always greater than the f(n). Then if:

$$f(n) \leq cg(n)$$
$$n \geq n_0$$
$$c > 0, n \geq 1$$

If the above satisfies, then we can say that

$$f(n) = O(g(n))$$

Which is equivalent to say that f(n) is smaller than g(n).

Let's take an example: f(n) = 3n + 2 and g(n) = n, is f(n) = O(g(n))?

$f(n) = 3n+2$, $g(n) = n$, for $f(n) = O(g(n))$ then

$\Rightarrow f(n) \leq c\,g(n)$

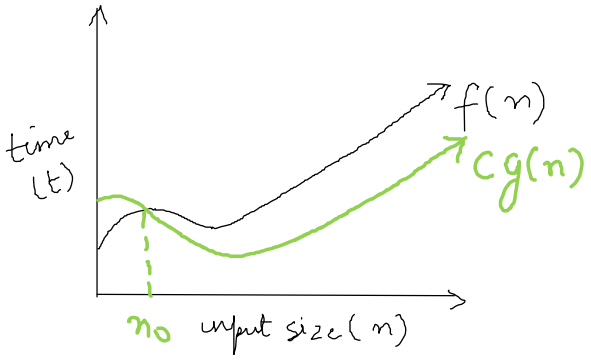
$\Rightarrow 3n+2 \leq cn$

Let $c=4 \Rightarrow 3n+2 \leq 4n \Rightarrow 2 \leq n$ $\therefore c=4$,

$\therefore f(n) = c\,g(n)$

Note: The tightest bound here is 'n'. If f(n) = O(g(n)) then definitely n^2, n^3… will be upper bounds, but we need to see the tightest bound which is 'n' in this case.

2. **Big-Omega(Ω)** :



Let's find another function 'cg(n)' (above graph) in such a way that the after it gets an input n_0 the value of this function is always smaller than the f(n). Then if:

$$f(n) \geq cg(n)$$
$$n \geq n_0$$
$$c > 0, n \geq 1$$

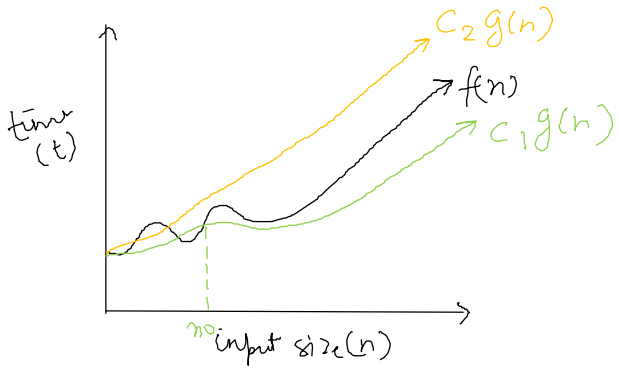
If the above satisfies, then we can say that

$$f(n) = \Omega\big(g(n)\big)$$

Which is equivalent to say that f(n) is greater than g(n).

We will see examples about it.

3. **Big-Theta(Θ):** We say that a function f(n) = Θ(g(n)), if the function f(n) is bounded by both lower and upper bounds. Let's see the graph for same.



Therefore,

$$c_1 g(n) \leq f(n) \leq C_2 g(n)$$

$$C_1, C_2 > 0$$

$$n \geq n_0, n_0 \geq 1$$

If the above is satisfied then we can say f(n) = Θ(g(n))

Example: f(n) = 3n + 2, g(n) = n. Is f(n) = Θ(g(n))?

$f(n) = 3n+2$, $g(n) = n$

(1) $f(n) \le c_1 g(n)$

$3n+2 \le c_1 n$ (for $c_1 = 4$ this is true)

$\therefore 3n+2 \le 4n$ for all $n_0 \ge 1$

$\therefore f(n) = O(g(n))$ — (i)

(2) $f(n) \ge c_2 g(n)$

$3n+2 \ge c_2 n$

$3n+2 \ge n$ $(c_2 = 1)$

$\therefore 3n+2 \ge n$ for all $n_0 \ge 1$

$\Rightarrow f(n) = \Omega(g(n))$ — (ii)

(i) & (ii) $\Rightarrow$ $f(n) = \Theta(g(n))$

Practical significance of the symbols:

Given any algorithm:

- O is going to say "worst case time" or the "upper bound". In other words, if you are giving time complexity in O notation, that will mean "In any case the time will not exceed this".
- Ω is going to say "best case time" which means in any case you can never achieve better than this.
- Θ is giving you the average case

In practicality we worry about the question "What is the worst case time the algorithm will take for any input?"

For some algorithms, the best and worst case will be same. In those scenarios we go for Θ.

We have seen that an algorithm's time complexity is given as a function f(n). This function is an *approximation* of the time taken by the algorithm and NOT the actual time it may take. Now let's see how to find f(n) which means *what is the approximate time taken by the algorithm.* Before we go into that let's understand more about algorithms.

There are two types of algorithms:

1. Iterative
2. Recursive

- It is worth noting that *any program that can be written using iteration can be written using recursion and vice versa.* Therefore, both are equivalent in power.
- For iterative program, you count the number of times the loop is going to get executed to find the time it's going to take. To find the time taken by a recursive program we use recursive equations. This means that they are both same in power but their analysis methods are different.
- ***In case if the algorithm does not contain either iteration or recursion it means that there is no dependency of the running time on the input size which means whatever is the size of the input the running time will be a constant value.***

Let's see some of the iterative programs and how to do complexity analysis of them.

TIME COMPLEXITY ANALYSIS OF ITERATIVE PROGRAMS

Example 1: Do the complexity analysis of the following program.

1)

```
A()
{
  int i,j;
  for(i=1 to n)        <- n times
    for(j=1 to n)      <- n times
      pf("Rahul")      : O(n^2)
}
```

2)

```
1991. A()
{ i=1, s=1;
  while (s<=n)
  { i++;
    s=s+i;
    pf("Rahul");
  }
}
```

Le 'k' be the value that 's' takes before the loop stops.

With each loop:

i=1, 2, 3, 4, … k

s = 1, 3, 6, 10, … n

We see that for every increase in 'i' s is the sum of first 'i' natural numbers. Therefore at when the loop reaches k to halt:

$K(k+1)/2 > n$

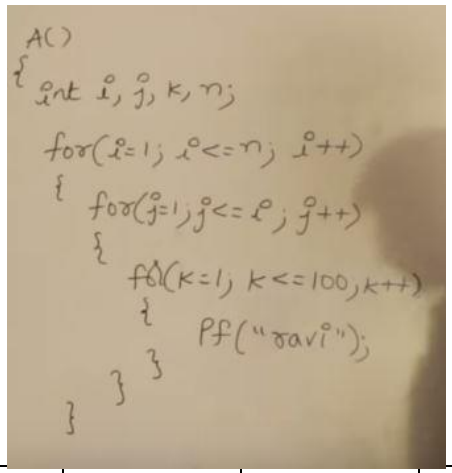
$(K^2 + k)/2 > n$

⇨ $K = O(\sqrt{n})$

3)

```
A()
{ for(i=1; i^2<=n; i++) ≡ for(i=1; i<=√n; i++))
  { pf("Hello");        : O(√n)
  }
}
```

4)



from the following table:

innermost loop will get executed:

100 + 200 + 300 .... nx100 or

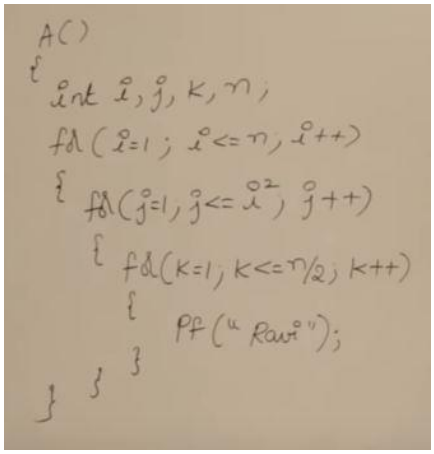
1x100 + 2x100 + 3x100 ... nx100 or

100 (1+2+3...n) or

100 (n(n+1)/2) ⇒ $O(n^2)$

I	1	2	3	4	…	N
J (times)	1	2	3	4	…	N
K(times)	1x100	2x100	3x100	4x100	…	Nx100

5



"Ravi" will get printed:

$n/2 \times 1 + n/2 \times 2 + n/2 \times 9 + n/2 \times 16 \cdots n/2 \times n^2$

$\Rightarrow n/2 \,(1 + 2 + 9 + 16 + n^2)$

$\Rightarrow n/2 \left(\frac{n(n+1)(2n+1)}{6}\right)$

$\Rightarrow \boxed{O(n^4)}$

I	1	2	3	4	…	N
J(times exec)	1	2	9	16	…	N^2
K(times exec)	(N /2)x1	(N /2)x2	(N /2)x9	(N /2)x16	…	(N/2)*N^2

$F(n) = n^k + n^{k-1} + n^{k-2} \ldots = O(n^k)$

6

```
A()
{ for (i=1 ; i<=n; i*=2)
  { print("Rahul"); }
}
```

Analysis:

for loops: $i = 1, 2, 4, \ldots, n$

Let us say it is going to take 'k' iterations by the time we reach 'n'.

∴ iterations: $2^0, 2^1, 2^2, 2^4 \ldots 2^k$

⇒ when program reaches 'n'

$2^k = n \Rightarrow \log(2^k) = \log(n)$

$\Rightarrow k = \log(n)$ ∴ $\boxed{O(\log n)}$

If i was increasing at the rate of 3, 4, 5, 6, instead of 2 in above example:

Then: for I = I * 2 : $O(\log_2 n)$

I = I * 3 : $O(\log_3 n)$ Therefore, for I = I *m complexity will be **$O(\log_m n)$**

I = I * 4 : $O(\log_4 n)$

I = I * 5 : $O(\log_5 n)$

Let's see an example depending on this example.

7

```
A()
{ int i, j, k;
  fo(i=n/2; i<=n; i++)
    fo(j=1; j<=n/2; j++)
      fo(k=1; k<=n; k=k*2)
        Pf("ravi");
}
```

runs → $n/2$ times

runs → $n/2$ times

runs → $\log_2 n$ times (prev example)

$\therefore$ complexity $= n/2 * n/2 * \log_2 n$

$= O(n^2 \log_2 n)$

We don't need to unroll the loops in this example because every loop's conditional statement is independent of the variable of the other loops.

8

```
A()
{ int i, j, k;
  fo(i=n/2; i<=n; i++)
    fo(j=1; j<=n; j=2*j)
      fo(k=1; k<=n; k=k*2)
        Pf(ravi);
}
```

runs → $n/2$ times

runs → $\log_2 n$ times

runs → $\log_2 n$ times

complexity: $n/2 * \log_2 n * \log_2 n$

$= O(n(\log_2 n)^2)$

In this example also the loops are independent of other loops' variables. So, no need to unroll them.

9 assume $n \geq 2$

```
A()
{ while(n>1)
  { n = n/2;
  }
}
```

Case 1: n is power of 2

for $n = 2^1$ while will execute 1 time

for $n = 2^2$ while will execute 2 time

for $n = 2^3$ while will execute 3 times

for $n = 2^k$ while will execute k times

$\therefore$ for n power of 2: $k = \log_2 n$

Case 2: n is not power of 2

let's say $n = 20$

→ 20 → 10 → 5 → 2 → 1 (4 times)

$\lfloor \log_2 20 \rfloor \simeq 4$

$\hookrightarrow O(\lfloor \log_2 n \rfloor)$

For question 9 above, if n was updating like:

N = n/5 then the complexity would be $O(\log_5 n)$

N = n/m then the complexity would be $O(\log_m n)$

Important result

10

```
A()
{
  for(i=1; i<=n; i++)
    for(j=1; j<=n; j=j+i)
      Pf("ravi");
}
```

from the following table:

"ravi" will get printed:

$N + N/2 + N/3 + \dots + N/N$

$= N(1 + 1/2 + 1/3 + \dots + 1/N)$

$= N(\log N) = O(n \log n)$

The inner loop is dependent on the variable of the outer loop. In this example we will have to unroll the loops to see wassup!

For I equals to	1	2	3	4	…	K	N
J will execute	1 to N	1 to N	1 to N	1 to N	…	1 to N	1 to N
J will execute times	N times	N/2 times	N/3 times	N/4 times	…	N/K times	N/N times

11

```
A()
{ int n = 2^k;
  for(i=1; i<=n; i++)
  {
    j=2
    while(j<=n)
    {
      j=j^2;
      Pf("ravi");
    }
  }
}
```

from the table below:

printf("ravi") will run

$n*(k+1)$ times

$n = 2^{2^k} \Rightarrow \log(n) = 2^k$

$\Rightarrow \log(\log(n)) = k$

$\Rightarrow n(\log\log n + 1) = n\log\log n + n = O(n \log\log n)$

At K	1	2	3	K
N will be	4	16	2^8	2^K
J values after each loop	2, 4	2, 4, 16	2, 4, 16, 256	2, 4, 16, …, 2^K
While will execute	N x 2 times	N x 3 times	N x 4 times	N x (K+1) times

TIME COMPLEXITY ANALYSIS OF RECURSIVE ALGORITHMS

Whenever a recursive function is given, the way you analyze the time complexity is different than the way you analyze the time complexity of iterative algorithms mainly because there is nothing to count in them.

To analyze the time taken by the recursive algorithm the first step is to find a recursive equation for the problem given and then proceed with that. Let's see that with the help of an example.

Example 1:

```
A(n)
{ if (n>1)
  return (A(n-1));
}
```

Now, let's suppose that the time taken to execute A(n) is T(n), then:

For the statement *if(n>1)* let's say some constant time 'c' is taken and

For A(n-1) the time will be T(n-1). Then we can say that our recursive equation for

The time complexity will be.

Recursive Equation : T(n) = c + T(n-1)

To solve the recursive equation, there are different ways:

1. Back substitution

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

   Substituting 2 in 1 we get: T(n) = c + c + T(n-2) = 2c + T(n-2)
   Substituting 3 in the above equation: T(n) = 2c + c + T(n-3) = 3c + T(n-3)
   …
   T(n) = kc + T(n-k) --- 4

   So when are we going to stop the algorithm? Take a look at the algorithm. Now 1 above is valid only when n>1. Therefore, whenever n = 1 we are going to stop.

   This condition when the recursion will stop is called ***anchor condition, base condition or stopping condition.***

   Now if we want to stop, we need T(1) in the back substitution chain.

   From 4 above, we need n-k =1 to get T(1)
   => k = n-1

   ⇨ T(n) = (n-1)c + T(n-(n-1))
   ⇨ T(n) = (n-1)c + T(1)
   ⇨ T(n) = (n-1)c + c
   ⇨ T(n) = nc
   ⇨ T(n) = O(n)

When using back substitution:

1. Find the recursive equation
2. Back substitute (find the general T(n))
3. Find the stopping condition
4. Solve

Example 2: For the given recursive relation, find the time complexity

T(n) = n + T(n-1), n > 1

1, n = 1

Solution : $T(n) = n + T(n-1)$ — (1)

$T(n-1) = (n-1) + T(n-2)$ — (2)

$T(n-2) = (n-2) + T(n-3)$ — (3)

substituting (2) in (1) we get: $T(n) = n + (n-1) + T(n-2)$ — (4)

substituting (3) in (4) we get: $T(n) = n + (n-1) + (n-2) + T(n-3)$

$\Rightarrow T(n) = n + (n-1) + (n-2) + \dots + (n-k) + T(n-(k+1))$ — (5)

let's eliminate T( ) in (5) according to base condition recursion will stop when n = 1

∴ T(1) will come

so $n-(k+1) = 1$

∴ $n-k-1 = 1$

∴ $k = n-2$

(5) $\Rightarrow T(n) = n + (n-1) + (n-2) + \dots + (n-(n-2)) + T(1)$

$T(n) = n + (n-1) + (n-2) + \dots + 2 + T(1)$

$T(n) = n + (n-1) + (n-2) + \dots + 2 + 1$

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

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

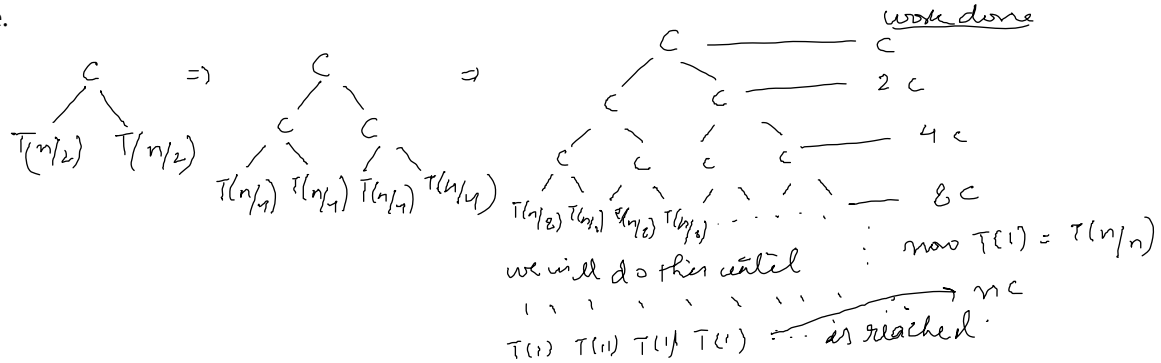
Let's see the next method which is called **RECURSION TREE** method.

Example: T(n) = 2T(n/2) + c, n > 1

1 , n = 1

Here we can think that if I do 'c' amount of work, then I can reduce the problem into two sets that taken T(n/2) time. Therefore.

T(n) =



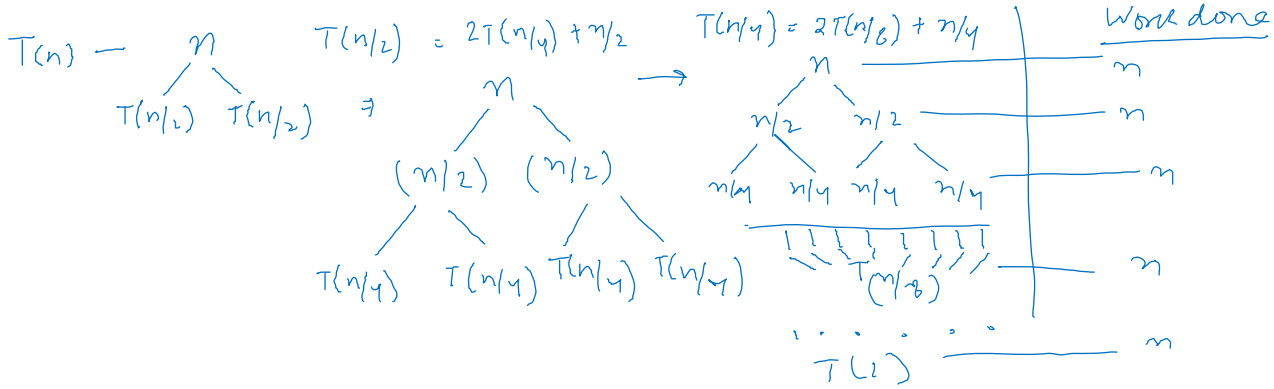
∴ Total work done: c + 2c + 4c + 8c +…+ nc

- ⇨ C(1+2+4+8+…+n) => C(sum of GP)
- ⇨ Let' say $n = 2^k$ => $C\ (\ 1(2^{k+1} -1)/(2-1))$
- ⇨ C(1+2+4+8+…+2^k) => C(2n -1) = O(n)

Example 2: T(n) = 2T(n/2) + n, n > 1

1 , n = 1

In this problem for every 'n' work done we can divide the problem into two halves, the time for which will be n/2



Total work: n + n + n … but how many n's are there?

Total n = n * number of levels, but how many levels are there?

Number of element in each level follow the fashion: 1, 2, 4, 8,… or $2^0, 2^1, 2^2, 2^3, \ldots, 2^k$

Therefore number of levels = K + 1 (0…k)

Number of n's = 2^k

K = logn

- ⇨ Number of levels = logn + 1
- ⇨ Total work done = n(logn + 1)
- ⇨ O(nlogn)

We saw recursion and back substitution methods for finding out time complexity of an algorithm. But, we need not do any of this as we are going to now learn about MASTER's THEOREM.

But before learning about master's theorem, we need to know "*how to compare two functions****" or answering questions like "****given two functions which one is asymptotically larger****".***

COMPARING VARIOUS FUNCTIONS TO ANALYZE TIME COMPLEXITY

If two function f(n) and g(n) are given and we need to know which one is larger.

1. Substitute LARGE values to 'n' and see or
2. Apply logarithm on both sides and then substitute
3. Whenever you cancel out the common terms and constants are remaining then the functions are *asymptotically equal.* Note: constants may not be equal.

Example: n^2 or 2^n. which one is more asymptotically complex?

Let $f(x) = n^2$ and $g(x) = 2^n$
applying log to both functions

$n^2 : 2 \log n \quad 2^n = n \log 2$

now let's substitute large values
let $n = 2^{100}$ then $n \log_2 2 = 2^{100}$
$2 \log_2 (2^{100}) = 2 \times 100 = 200$

$\therefore g(x) = 2^n$ is asymptotically more complex

Example: $f(x) = 3^n \quad g(x) = 2^n$

apply log: $n \log_2 3 \quad n \log_2 2$

remove common: $n \log_2 3 \quad n \log_2 2 \rightarrow \log_2 3 > \log_2 2 \therefore f(x) = 3^n$ is asymptotically more complex.

Example: $f(x) = n^2, \; g(x) = n \log n$

① cancel common terms: $n, \log n \mid n > \log n \therefore f(x) = n^2$ has greater complexity

Example $f(x) = n \quad g(x) = (\log n)^{100}$

applying log: $\log_2 n, \; 100 \log_2 \log_2 n$

let $n = 2^{128}$ $\log_2 2^{128}, \; 100 \log_2 \log_2 2^{128} \Rightarrow 128, 100 \log_2 128$
$\Rightarrow 128, 100 \times 7$
$\Rightarrow 128, 700$

let $n = 2^{1024}$ $\log_2 2^{1024}, \; 100 \log_2 \log_2 2^{1024} \rightarrow 1024, 100 \times \log_2 1024$
$\Rightarrow 1024, 100 * 10$
$\Rightarrow 1024, 1000$

$\therefore n$ is growing more later $\therefore$ its complexity is more!!!

Example $f(x) = n^{\log n} \quad g(x) = n \log n$, is $f(n) = O(g(n))$?

log on both sides we get: $\log n \log n, \; \log n + \log \log n$

substitute: $n = 2^{1024}, \; \frac{1024 \times 1024}{(1024)^2}, \; 1024 + 10, \; \frac{10}{}$ $\therefore n \log n < n^{\log n}$ in asymptotic complexity

Example $f(x) = \sqrt{\log n} \quad g(x) = \log \log n$

log both sides: $\frac{1}{2} \log \log n \quad \log \log \log n \mid n = 2^{2^{10}}$
$\Rightarrow \frac{1}{2} \times 10, \; 3.5$

Example $f(x) = n^{\sqrt{n}}$ $g(x) = n^{\log n}$ is $f(x) = O(g(x))$?

log: $\sqrt{n}\log n$ $\log n \log n$ | $\sqrt{n}$, $\log n$ | $\frac{1}{2}\log n$, $\log\log n$ | $n = 2^{10}$ → $\frac{1}{2} \times 2^{10}$, 10

∴ $n^{\sqrt{n}}$ has higher complexity asymptotically.

Example $f(n) = \begin{cases} n^3, & 0 < n < 10000 \\ n^2, & n \geq 10000 \end{cases}$ $g(n) = \begin{cases} n, & 0 < n < 100 \\ n^3, & n > 100 \end{cases}$

We worry about the large numbers. In this example, let's take the values the functions f(n) and g(n) take when n > 10000. F(n) takes n^2 and g(n) takes n^3. Therefore, in this example f(n) = O(g(n)).

Example $f_1 = 2^n$, $f_2 = n^{3/2}$, $f_3 = n\log n$ $f_4 = n^{\log n}$

For this problem to solve. You have to compare all of the functions with each other. Pick a function which looks complex (like f_1) and then compare it with others. Proceed this with others also.

Answer: f1 > f4 > f2 > f3

MASTERS THEOREM

This theorem is used to solve the recursive equations and find out the time complexity in terms of asymptotic notations.

If the recursive equation is of the form:

$$T(n) = aT(n / b) + \Theta(n^K \log^p n)$$

$$a \geq 1, b > 1, k \geq 0 \text{ and p is a real number}$$

Then:

1. If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$
2. If $a = b^k$
    a. If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
    b. If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
    c. If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$
3. If $a < b^k$
    a. If $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$
    b. If $p < 0$, then $T(n) = \Theta(n^k)$

Master's theorem is used to find the time complexity of the recursive relations given. For a given recursive equation:

1. Find out what is a, b, k and p
2. Compare 'a' with 'b^k'
3. Follow the results on the comparison in step 2.

Let's see some examples.

Example 1: $T(n) = 3T(n/2) + n^2$

Here a = 3, b = 2, k = 2, p = 0
$b^k = 4$, => $a < b^k$
and p>= 0 => $T(n) = \Theta(n^k \log^p n) = \mathbf{\Theta(n^2)}$

Example 2: $T(n) = 3T(n/2) + n^2$

Here a = 4, b = 2, k = 2, p = 0
$b^k = 4 \Rightarrow a = b^k$
and p > -1, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
⇨ $T(n) = \Theta(n^{\log_2 4} \log^{0+1} n) = \mathbf{\Theta(n^2 logn)}$

Example 3: $T(n) = T(n/2) + n^2$

Here a = 1, b = 2, k = 2, p = 0
$b^k = 4 \Rightarrow a < b^k$
and p >= 0, then $T(n) = \Theta(n^k \log^p n)$
⇨ $T(n) = \Theta(n^2 \log^0 n) = \mathbf{\Theta(n^2)}$

Example 4: $T(n) = 2^nT(n/2) + n^n$
Here $a = 2^n$ which is not a constant, therefore Master's Theorem can't be applied for this problem. ***We will see how to convert the problems for which master's theorem can't be applied into one where it can be applied.***

Example 5: $T(n) = 16T(n/4) + n$
Here a = 16, b = 4, k = 1, p = 0
$b^k = 4 \Rightarrow a > b^k$
⇨ $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_4 16})$
⇨ $T(n) = \mathbf{\Theta(n^2)}$

Example 6: $T(n) = 2T(n/2) + nlogn$
Here a = 2, b = 2, k = 1, p = 1
$b^k = 2 \Rightarrow a = b^k$
and p > -1, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
⇨ $T(n) = \Theta(n^{\log_2 2} \log^{1+1} n) = \mathbf{\Theta(nlog^2 n)}$

Example 7: $T(n) = 2T(n/2) + n/logn$
Or $T(n) = 2T(n/2) + nlog^{-1}n$
Here a = 2, b = 2, k = 1, p = -1
$b^k = 2 \Rightarrow a = b^k$
and p = -1, then $T(n) = \Theta(n^{\log_b a} loglogn)$
⇨ $T(n) = \Theta(n^{\log_2 2} loglogn)$
⇨ $T(n) = \mathbf{\Theta(nloglogn)}$

Example 8: $T(n) = 2T(n/4) + n^{0.51}$
Here a = 2, b = 4, k = 0.51, p = 0
$b^k = 4^{0.51} \Rightarrow a < b^k$
and p >= 0, then $T(n) = \Theta(n^k \log^p n)$
⇨ $T(n) = \Theta(n^{0.51} \log^0 n)$
⇨ $T(n) = \mathbf{\Theta(n^{0.51})}$

Example 9: $T(n) = 0.5T(n/2) + 1/n$
Here a = 0.5, b = 2, k = -1
Not valid for master's theorem as 'a' and 'k' values are not valid.

Example 10: $T(n) = 6T(n/3) + n^2logn$
Here a = 6, b = 3, k = 2, p = 1
$b^k = 9 \Rightarrow a < b^k$
and p >= 0, then $T(n) = \Theta(n^k \log^p n)$
⇨ $T(n) = \Theta(n^2 \log^1 n)$
⇨ **$T(n) = \Theta(n^2 logn)$**

Example 11: $T(n) = 64T(n/8) - n^2logn$
Here we have '-' instead of a +. This says that whenever we **don't do n^2logn work** the problem is going to divide itself into 64 parts. Therefore, this question is not valid for Master's theorem.

Example 12: $T(n) = 7T(n/3) + n^2$
Here a = 7, b = 3, k = 2, p = 0
$b^k = 9 \Rightarrow a < b^k$
p >= 0, then $T(n) = \Theta(n^k \log^p n)$
⇨ $T(n) = \Theta(n^2 \log^0 n)$
⇨ $T(n) = \mathbf{\Theta(n^2)}$
⇨

Example 13: $T(n) = 4T(n/2) + logn$
Here a = 4, b = 2, k = 0, p = 1
$b^k = 1 \Rightarrow a > b^k$

Example 14: $T(n) = \sqrt{2}T(n/2) + logn$
Here $a = \sqrt{2}$, b = 2, k = 0, p = 1
$b^k = 1 \Rightarrow a > b^k$

Therefore, $T(n) = \Theta(n^{\log_b a})$
⇨ $T(n) = \Theta(n^{\log_2 4})$
⇨ $T(n) = \mathbf{\Theta(n^2)}$

then $T(n) = \Theta(n^{\log_b a})$ => $T(n) = \Theta(n^{\log_2 root2})$
⇨ $T(n) = \mathbf{\Theta(\sqrt{n})}$

Example 15: $T(n) = 2T(n/2) + \sqrt{n}$
Here a = 2, b = 2, k = 1/2, p = 0
$b^k = \sqrt{2}$ => $a > b^k$
Therefore, $T(n) = \Theta(n^{\log_b a})$
⇨ $T(n) = \Theta(n^{\log_2 2})$
⇨ $T(n) = \mathbf{\Theta(n)}$

OTHER REMAINING EXAMPLES ARE NOT WRITTEN DUE TO THEIR SIMPLICITY.

Now that we have seen the Master's Algorithm and Time Complexity of algorithms, let's move on to the analysis of Space Complexity of the algorithms.

ANALYZING SPACE COMPLEXITY OF ITERATIVE AND RECURSIVE ALGORITHMS

So what is space?

Space: Given a program, how many memory cells are required to finish running it.

Space has to be analyzed in terms of given input size. This means if given the size of input as 'n' what is the total space required in relation to this input size.

Intuition:

If I need 'n' extra cells to run this program, space complexity will be O(n)
If I need '10' extra cells to run a program, space complexity will be O(1)
If I need 'n^2' extra cells to run a program, the space complexity will be O(n^2)

Note: Most of the times there is a tradeoff between the space and time complexity. It can be said that these two complexities are inversely proportional to each other.

Let's start with how to analyze the algorithm and answer what will be the total space required by the algorithm.

SPACE COMPLEXITY OF ITERATIVE ALGORITHMS

The space complexity of the algorithm is calculated using the "extra space" required to run the algorithm "apart from the input size given". Let's understand with the help of some examples.

Example 1

```
Algo(A[],l,n)
{ int i,j=10;
  for(i=1 to j)
     A[i]=0;
}
```

Apart from the space taken by the input, This algorithm is not taking any extra space. Therefore space complexity is O(1).

Example 2

```
Algo(A[],l,n)
{ int i;
  create B[n];
  for(i=1 to n)
     B[i]=A[i];
}
```

extra

Apart from space taken by the input, we have another array B which takes space of size n.
∴ int i → 1 space
B[n] → n space
Total space = 1+n or O(n)

Example 3

```
Algo(A[],l,n)
{ create B[n,n]
  int i,j
  for(i=1 to n)
    for(j=1 to n)
      B[i,j]=A[i]
}
```

n^2 space
2 spaces
Total = (n^2+2) or $O(n^2)$
computation so NO SPACES

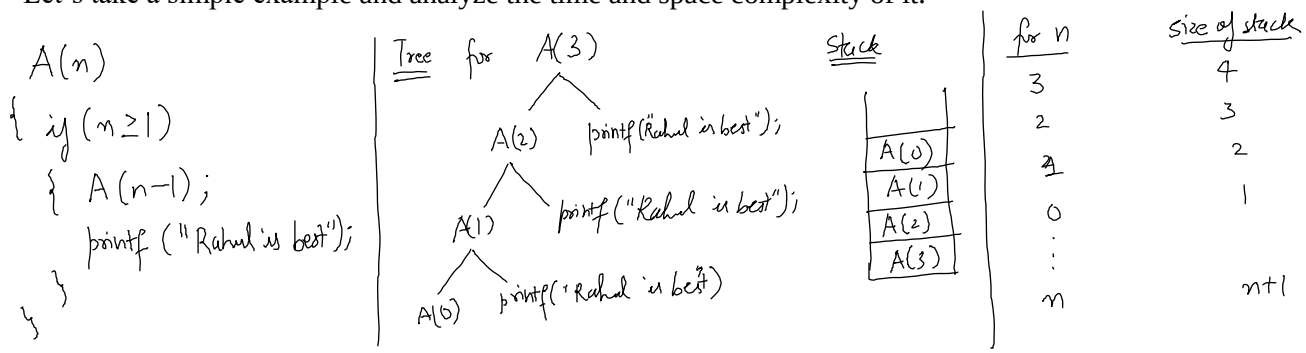
Finding the space complexity of an iterative program is quite easy. The process of finding the space complexity of a recursive program is quite complex. Let's see how it is done.

SPACE COMPLEXITY OF RECURSIVE ALGORITHMS

There are various ways to analyze the recursion algoirthms' space complexity.

*Note: Whenever the size of the program is small, use **Tree Method.** (3-4 lines)*
*Whenever the size of the program is big, use **Stack Method.** (lot of lines)*

Let's take a simple example and analyze the time and space complexity of it.



Points to know before we do anything.

1. ***Every function call is given a space in the stack memory (RAM).***
2. ***The calls of the recursive functions are stored in the stack until the halting/anchor condition is reached.***
3. ***The space taken by that stack is the space complexity of the recursive algorithm. For each call we can assume that the stack allocates 'k' units of space.***

As seen above, you can use the tree or stack method to see the size of the memory which will be required by the recursive algorithm. Now let's say that 'k' is the size of each stack cell.

Therefore, total memory required for input size of 'n' = (n+1)k or **O(kn)** or **O(n)**

Let's also analyze the time complexity of this algorithm.

Assume that the time taken by A(n) to run is T(n). Therefore, for A(n-1) time will be T(n-1).

⇨ Recursive equation: T(n) = T(n-1) + c (*for printing)* ---(1)

Since this equation does not follow the rules required for Master's Theorem, therefore it can't be applied here. Let's do the back substitution.

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

Substituting (2) in (1) we get: T(n) = T(n-2) + 2c --- (4)
Substituting (3) in (4) we get: T(n) = T(n-3) + 3c

Therefore: T(n) = T(n-k) + kc

for halting condition n >=1. T(n-k) will become T(0) when n-k = 0or k = n

- ⇨ T(n) = T(n-n) + nc
- ⇨ T(n) = c + nc
- ⇨ T(n) = (n+1)c or **O(n)**

IMPORTANT

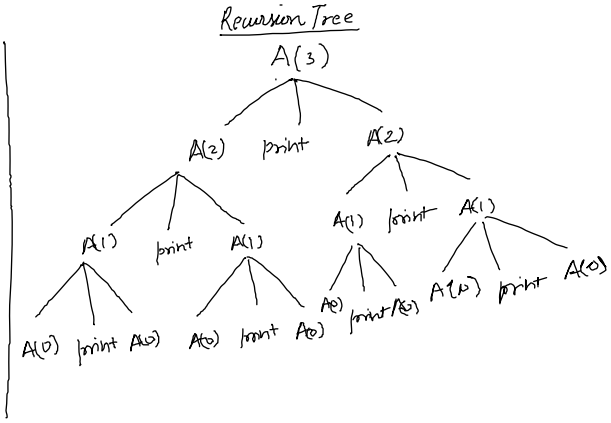
Questions that can be asked:

1. ***How many times the recursive function A(n) is called***
2. ***What is the space complexity?***
3. ***What is the time complexity?***

Let's see one more example

Example 2:

```
A(n)
{  if (n>1)
   {  A(n-1);
      printf(" Rahul is great! ");
      A(n-1);
   }
}
```



N	Number of calls for A(n)
3	$15 = 2^{3+1} - 1$
2	$7 = 2^{2+1} - 1$
1	$3 = 2^{1+1} - 1$
N	$2^{N+1} - 1$

Note: The number of recursive calls to a function does not define the amount of space taken by the program. Though it looks like this algorithm's space complexity is O(2^n), that is not the case. Let's see how.

The interesting thing is that the stack for the program does not take more than 4 cells to deal with all of the recursive calls for n = 3. This is because for every leaf node reached in the recursion tree, it gets popped out of the stack. Thus the pushing of non-leaf nodes from left-bottom-right mechanism and popping off is a constant process which doesn't take more than 4 cells of stack for n = 3.

Therefore, for this algorithm for an input size of 'n' the stack will take n+1 cells. If each cell is of size k.

Space complexity = O((n+1)k) or simply O(n).

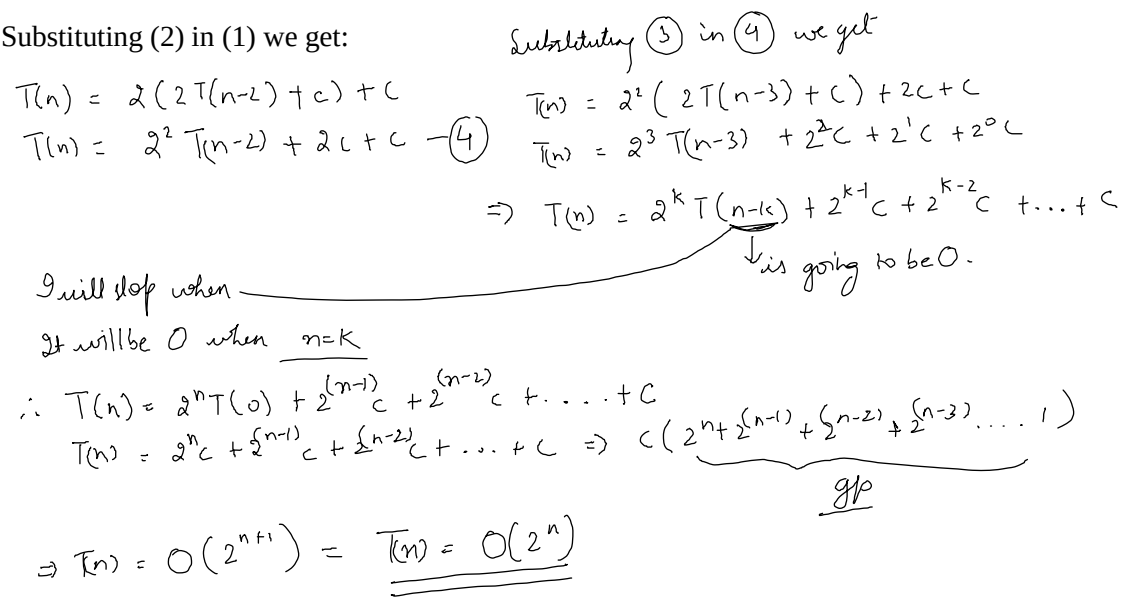
Let's see the time complexity of this algorithm.

T(n) = T(n-1) + T(n-1) + c or T(n) = 2T(n-1) + c (not eligible for Master's theorem). --- (1)

Let's use back substitution method.

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

Substituting (2) in (1) we get:



The space complexity of the algorithm is O(n) and the time complexity is $O(2^n)$. This is an exponential time complexity. This can be reduced by application **of *dynamic programming*** where we can decide dynamically using lookup table (where the previous calculation results are already saved) to not run another recursive call which has already happened.

Thank you for reading my notes. I hope they help you. Next up is Sorting Algorithms which is in a different file. Please go to that file for the continuation of the notes. <3

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

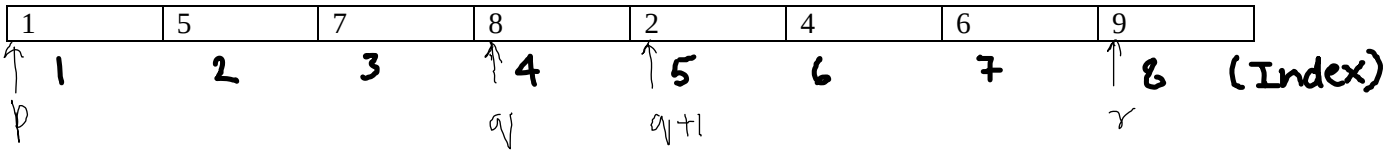
Merge sort falls under the Divide-and-conquer algorithms. It simply means you divide the problem into smaller steps and tackle the problem of sorting using bottom-up approach.

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
}
```

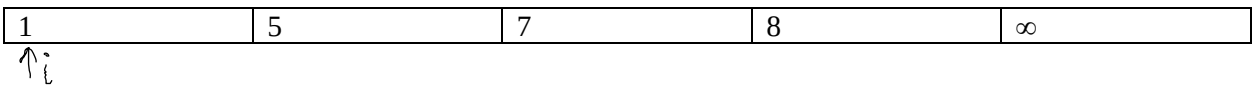
Working of MERGE:

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

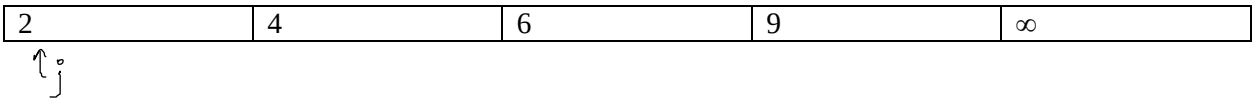


We divide the list above into two lists L and R containing the first half sorted elements and the second half sorted elements.

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)
}
```

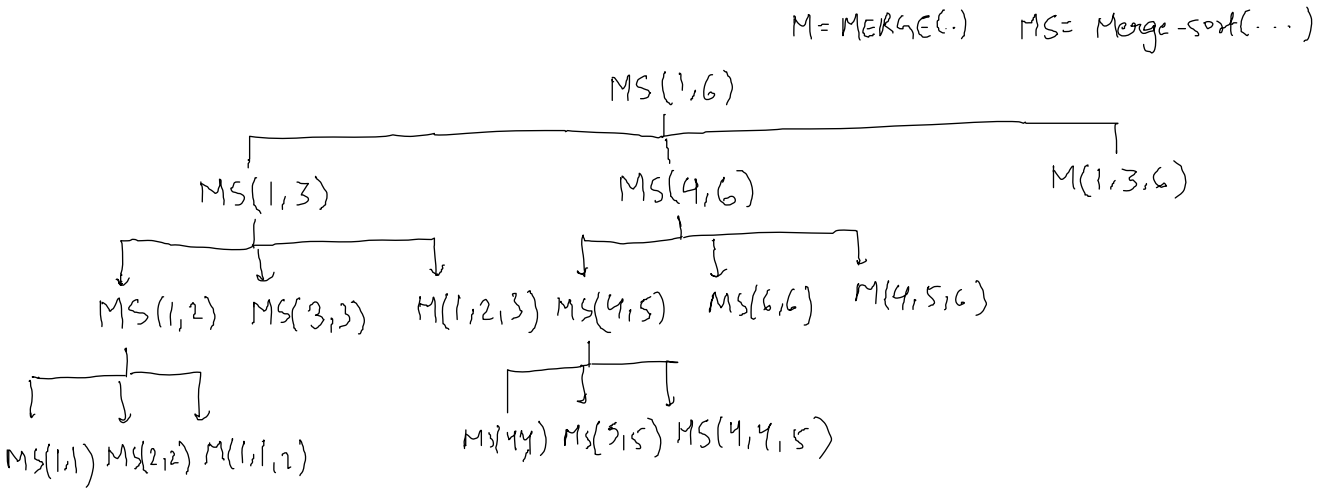
As said earlier, Mege Sort falls under the category of Divide and Conquer algorithms. In this sorting technique we divide the array to be sorted into single element array (single element array is already sorted) and move up from there to give us the final sorted array.

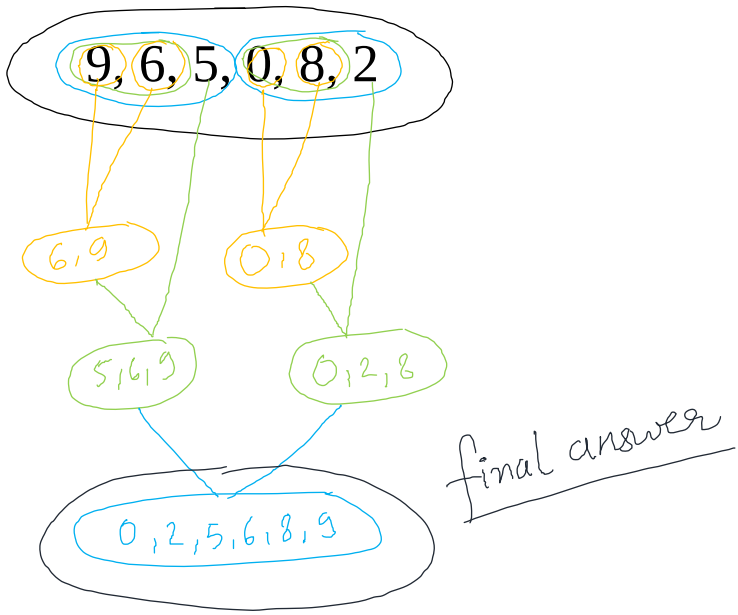
This is a recursive algorithm where each function call is going to call three functions:

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

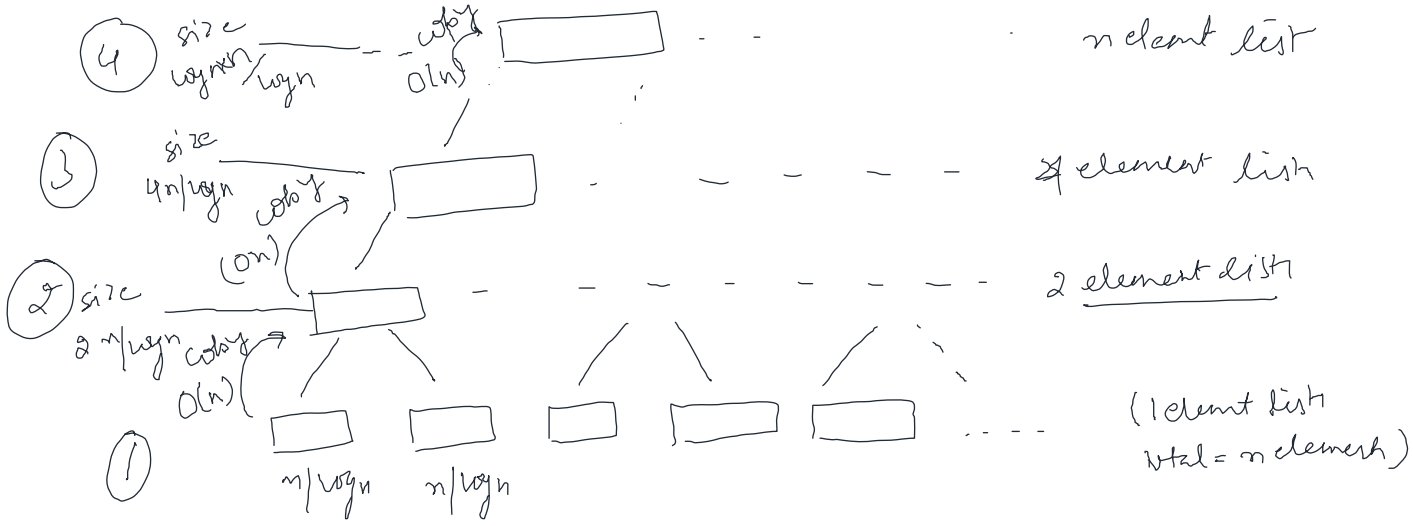
Each list is of size: n/logn

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)
        }
}
```

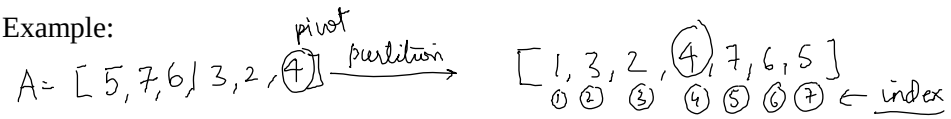
The index of the sorted element by the PARTITION algorithm is returned as q in the above algorithm.

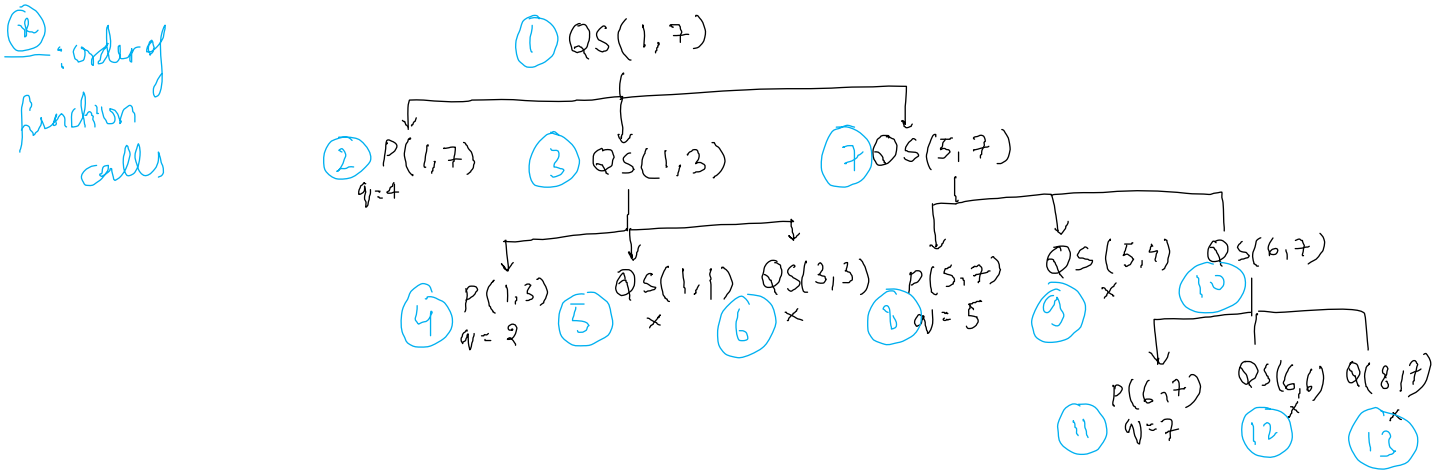
Note: In the merge sort algorithm, the arrays were divided into halves. In the quick sort algorithm the sub arrays can have different lengths depending on the value of 'q' returned by the PARTITION algorithm.

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

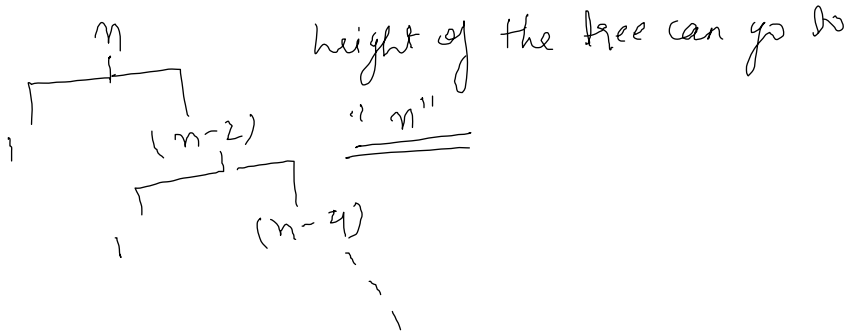
3. Use stack to evaluate the calls.

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

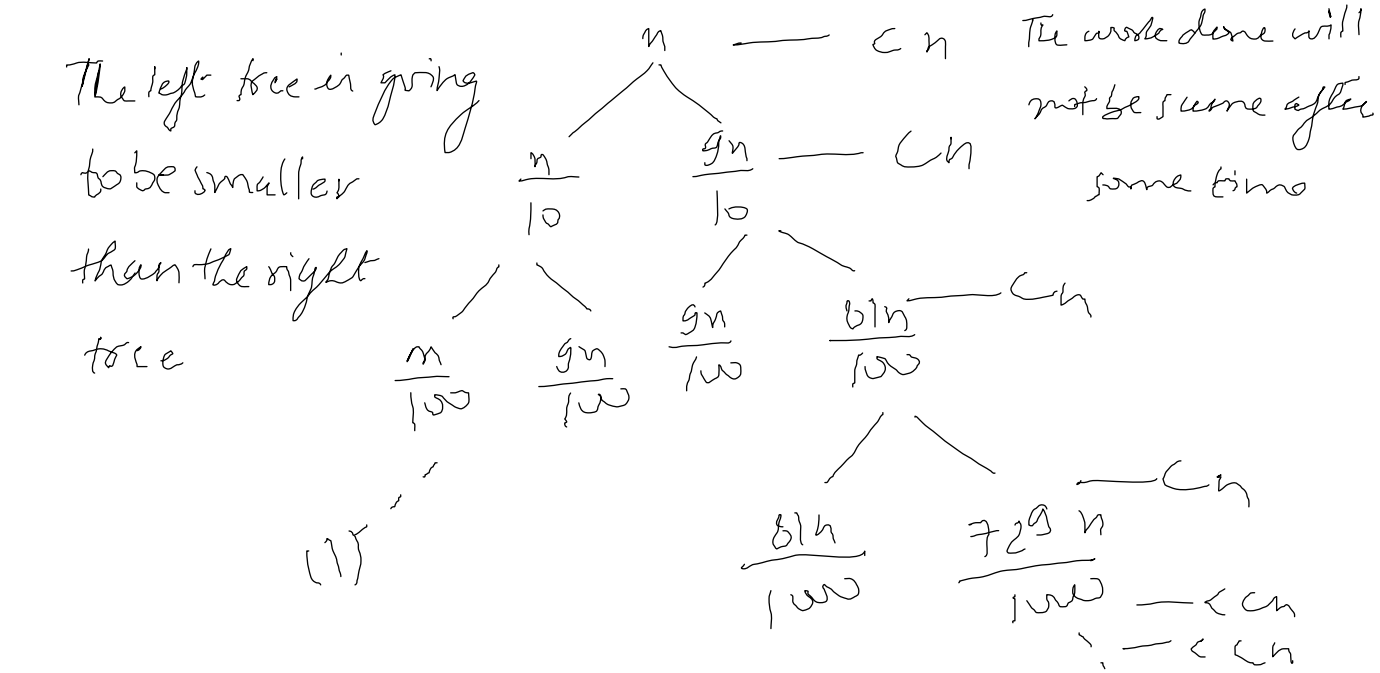
A = {2, 2, 2, 2, 2, 2}

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



The size of the array decreases as in each level like this

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

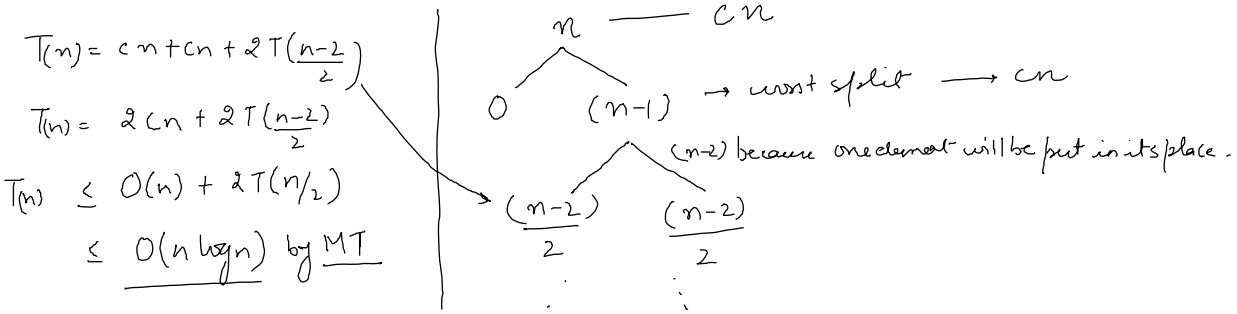
or $\log_{10/9} n = \Theta(\log_2 n)$ & each level has "cn" (roughly) work

∴ 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

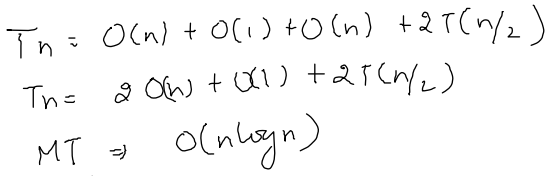


Therefore, even if the split is alternating between the one that produces best case and the one that produces worst case, we get the best case time complexity.

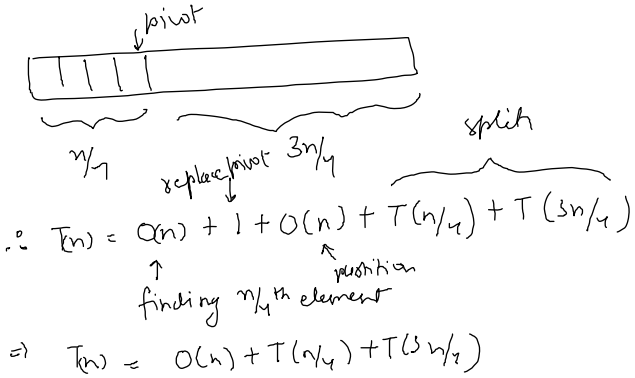
Let's see some questions on Quick Sort

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.

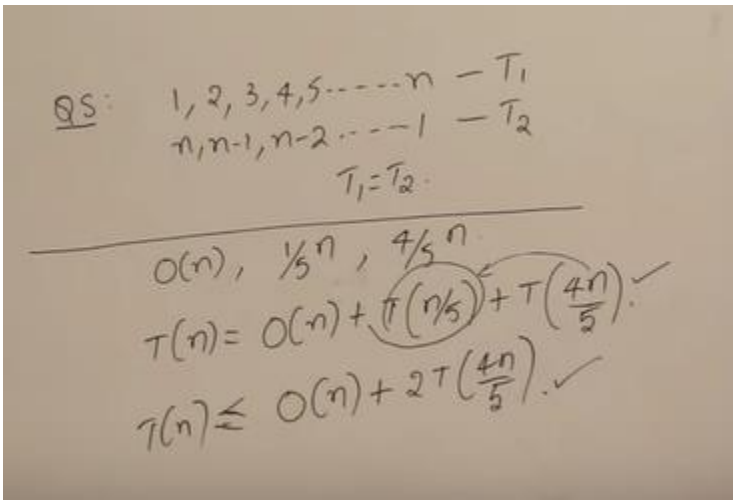


Q2. In quick sort, for sorting 'n' elements, the $(n/4)^{th}$ smallest element is selected as pivot using O(n) time algorithm. What is the worst case time complexity of quick sort?



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).



--- (add the data from the work computer).

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:

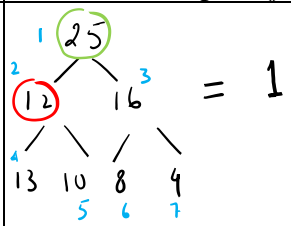
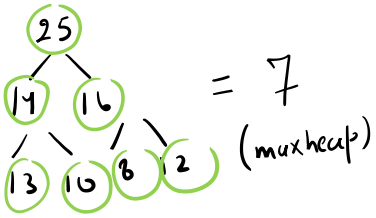
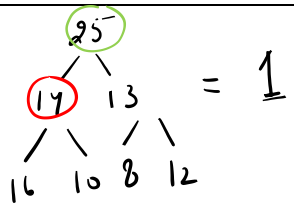
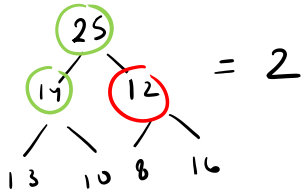
Example:

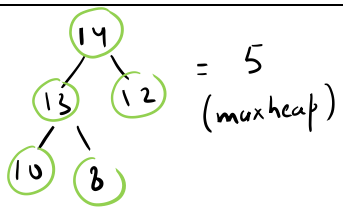
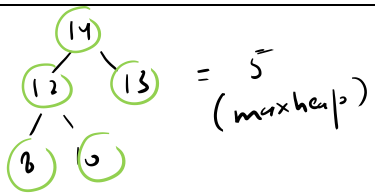
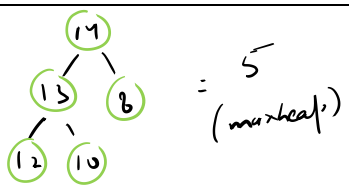
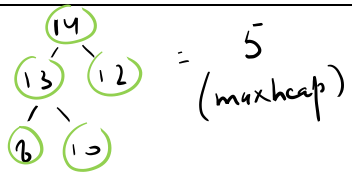
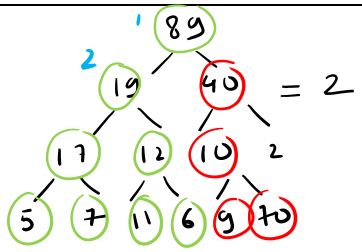
Q.1 Is the array given a max heap?

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	
25, 14, 16, 13, 10, 8, 12	7	
25, 14, 13, 16, 10, 8, 12	7	
25, 14, 12, 13, 10, 8, 16	7	

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

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.

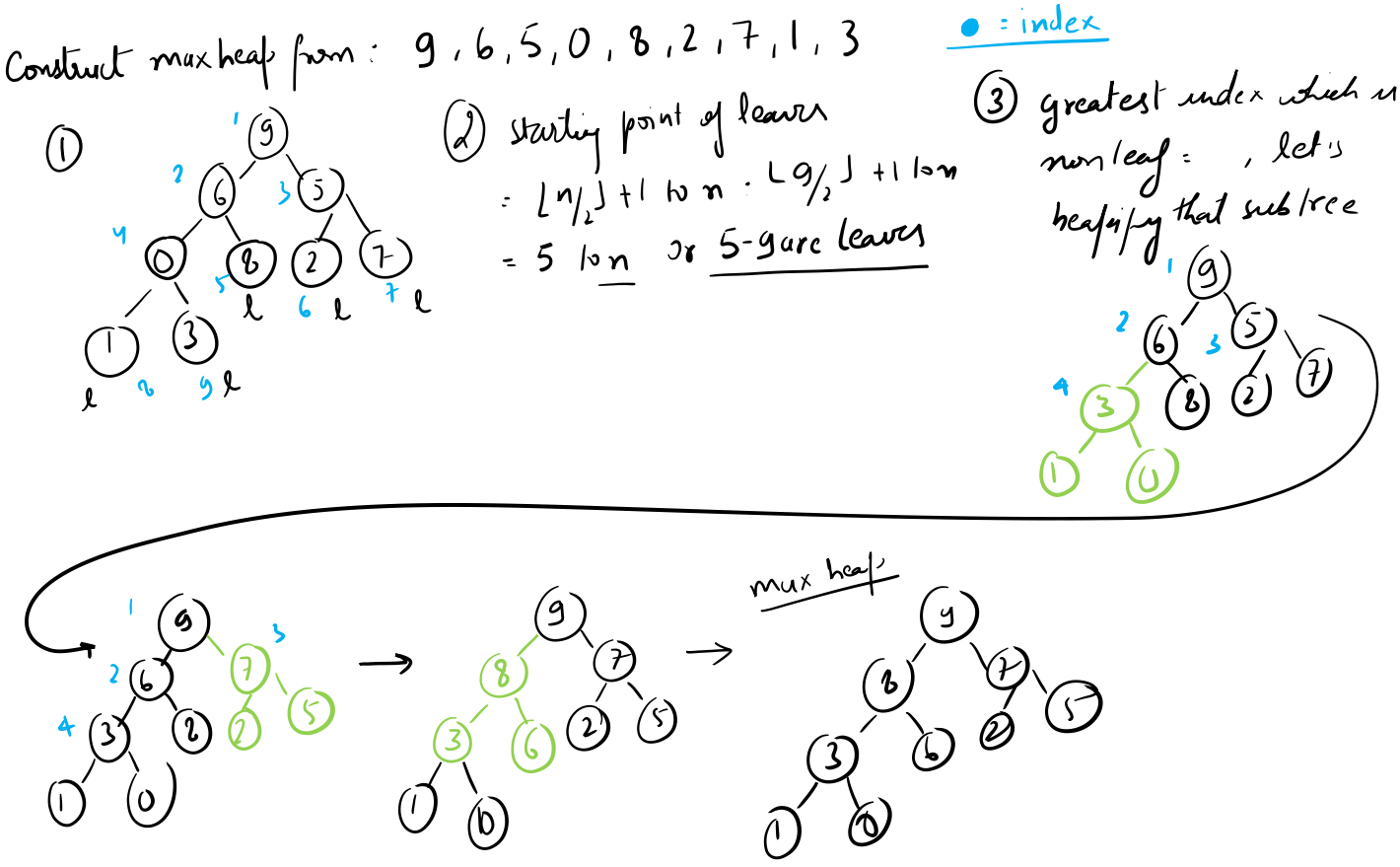
1. Where will the leaves start in a tree?
   a. In a complete binary tree, the last level will all have leaves, therefore, if leaves are starting from an index 'i' then all the indices after 'i' will be leaves. This property is also followed in almost complete binary tree.
2. In a complete/almost complete 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.



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;
        if(l<= A.heapsize and A[l]>A[i])
              largest = l;
        else largest = i;

        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 = ⌊A.length/2⌋ 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).

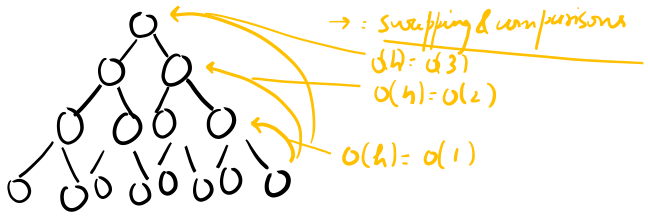
Let's talk about another property of Complete Binary Tree:

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 $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ 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 = $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

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

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

n, c and h are constants, then we can write: $\frac{Cn}{2} \cdot \sum_{h=0}^{\log n} \left\lceil \frac{h}{2^h} \right\rceil$ <= 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 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)
{
    If heap_size < 1
        error "heap underflow"
    Max = A[1]
    A[1] = A[A.heap_size]
    A.heap_size = A.heap_size – 1
    MAX_HEAPIFY(A, 1)
    Return Max
}
```

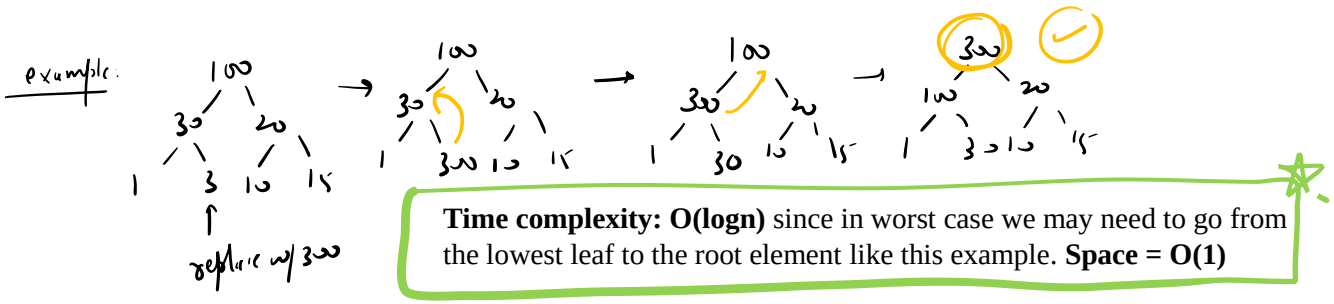
Time complexity = O(logn)

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
}
```



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.

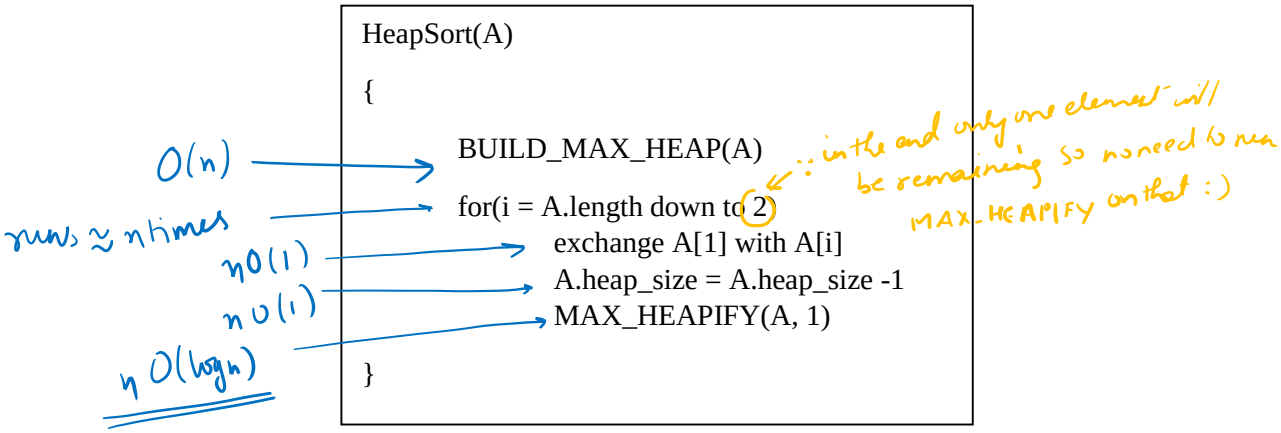
Green means heap is optimal for such problems

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



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?

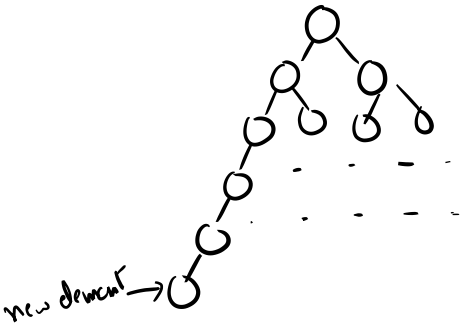
a. O(loglogn)
b. O(n)
c. O(logn)
d. O(1)

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)**