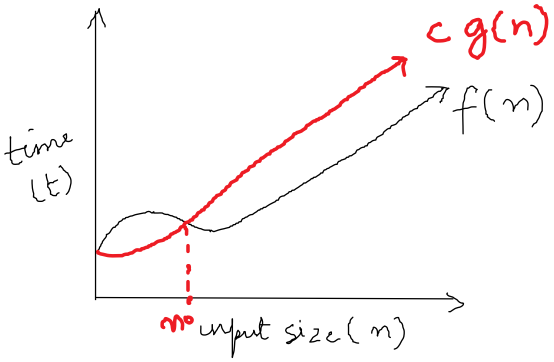
Algorithms: Space and Time Complexity

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

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

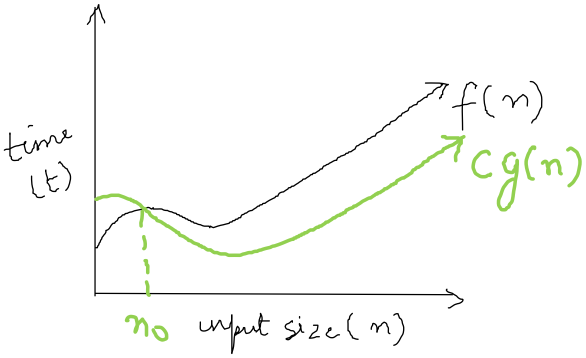
$\Rightarrow 3n+2 \leq c\,n$

$\underline{\text{let } c=4}$ $\Rightarrow 3n+2 \leq 4n \Rightarrow \boxed{2 \leq n}$ $\therefore \underline{c=4}$,

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

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:

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

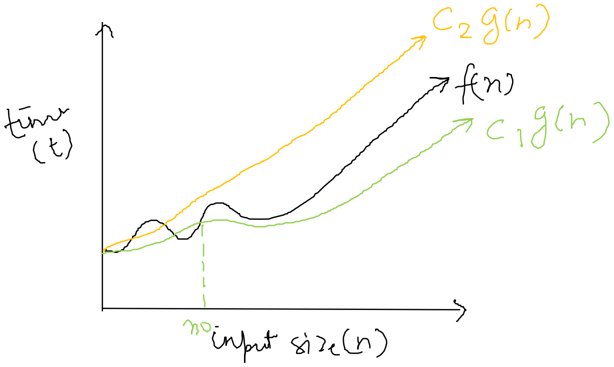
If the above satisfies, then we can say that

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

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))?

$f(n) = 3n+2 \quad g(n) = n$

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

$3n+2 \ge c_2 n$

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

$\therefore\ 3n+2 \ge n$ 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")
}
```

$\boxed{\therefore O(n^2)}$

1991· 2

```
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:

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

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

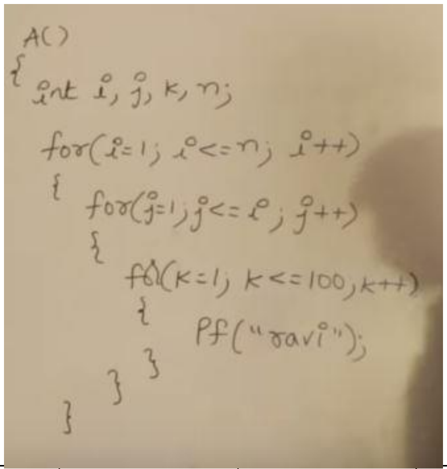
⇨ $K = O(\sqrt{n})$

3

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

$\therefore O(\sqrt{n})$

4



```
A()
{
 int i, j, k, n;
  for(i=1; i<=n; i++)
  {
    for(j=1; j<=i; j++)
    {
      for(k=1; k<=100, k++)
      {
          pf("ravi");
      }
    }
  }
```

from the following table:

innermost loop will get executed:

100 + 200 + 300 . . . . nx100 or

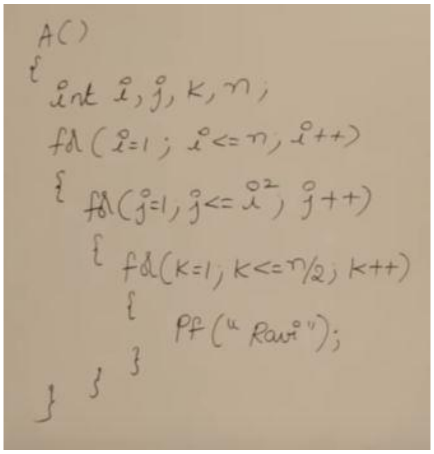
1x100 + 2x100 + 3x100 . . . nx100 or

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

$100\,(n(n+1)/2) \Rightarrow \boxed{O(n^2)}$

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

5



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

"Ravi" will get printed:

$$\frac{n}{2}\times 1 + \frac{n}{2}\times 2 + \frac{n}{2}\times 9 + \frac{n}{2}\times 16 \cdots \frac{n}{2}\times n^2$$

$$\Rightarrow \frac{n}{2}\left(1+2+9+16+n^2\right)$$

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

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

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'.

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

$\Rightarrow$ when program reaches 'n'

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

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

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

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

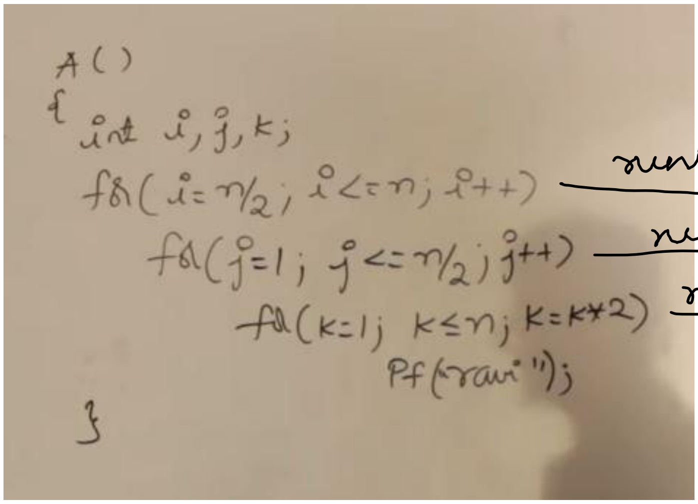
I = I * 3 : $O(\log_3 n)$ Therefore, for I = I *m complexity will be **$O(\log_m n)$**

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;
 for(i=n/2; i<=n; i++)
    for(j=1; j<=n/2; j++)
       for(k=1; k≤n; k=k*2)
            pf("ravi");
}
```

runs → $n/2$ times

runs → $n/2$ times

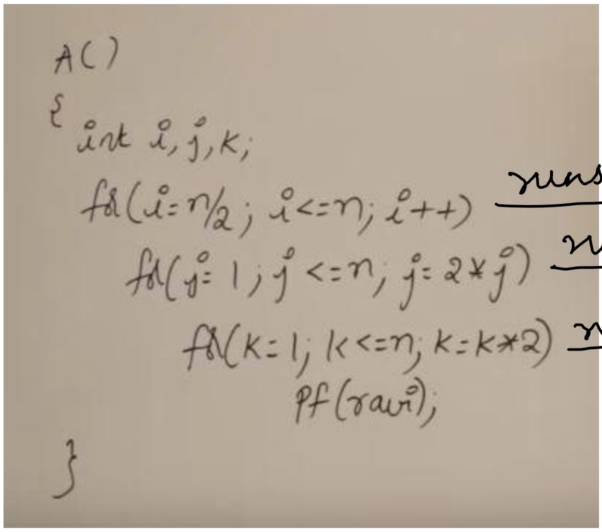
runs → $\log_2 n$ times (prev example)

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

$= \boxed{O(n^2 \log_2 n)}$

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;
 for(i=n/2; i<=n; i++)
    for(j=1; j<=n; j=2*j)
       for(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 $n=2^k$ wille 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

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

→ $O(\lfloor \log_2 n \rfloor)$

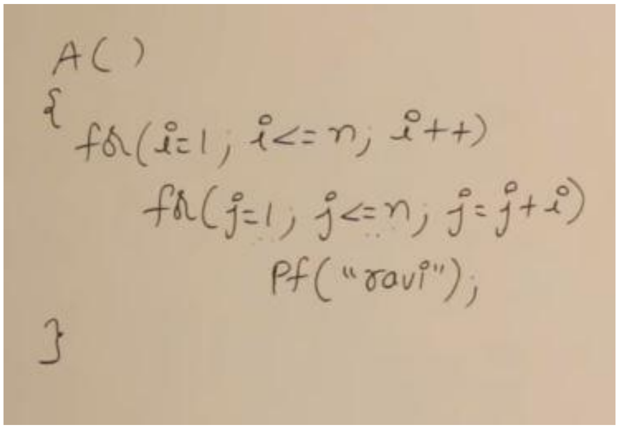
For question 9 above, if n was updating like:

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:

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

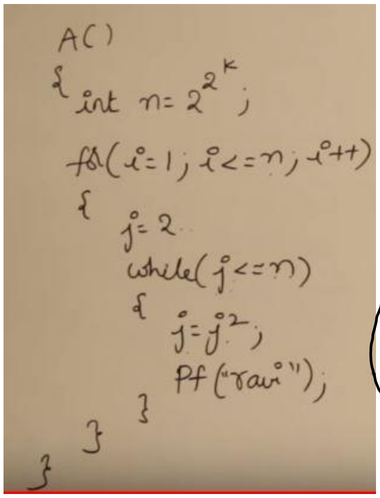
$= N(1 + 1/2 + 1/3 + \dots + 1/N)$

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

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= 2^2^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

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

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

$\Rightarrow n(\log\log n + 1) = n\log\log n + n = \boxed{O(n \log\log n)}$

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

$T(n) = n + T(n-1) \quad - (1)$

<u>Solution</u> : $T(n-1) = (n-1) + T(n-2) \quad - (2)$

$T(n-2) = (n-2) + T(n-3) \quad - (3)$

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

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

$\Rightarrow T(n) = n + (n-1) + (n-2) + \ldots + (n-k) + T(n-(k+1)) \quad - (5)$

let's eliminate T( ) in (5)
according to base condition
recursion will stop when n = 1
∴ T(1) will come
so $n-(k+1) = 1$
∴ $n-k-1 = 1$
∴ $\underline{k = n-2}$

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

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

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

$T(n) = \frac{n(n+1)}{2} \Rightarrow \boxed{T(n) = O(n^2)}$

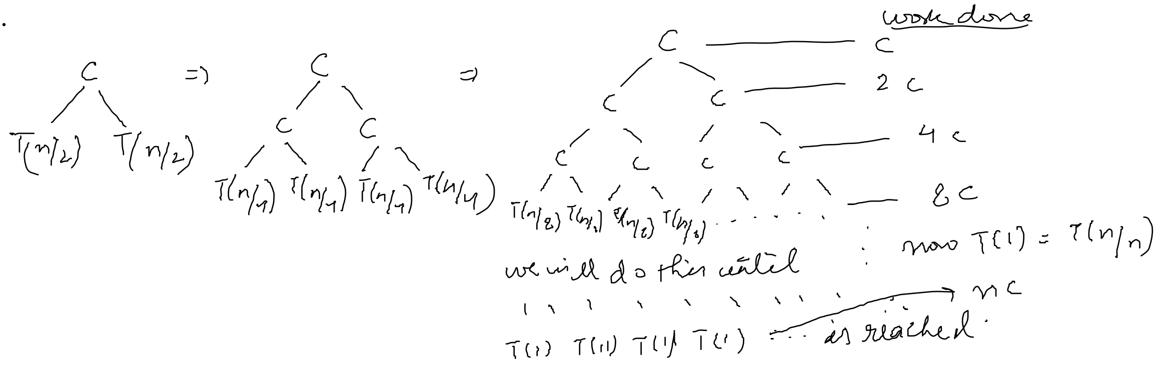
Let's see the next method which is called **RECURSION TREE** method.

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) =



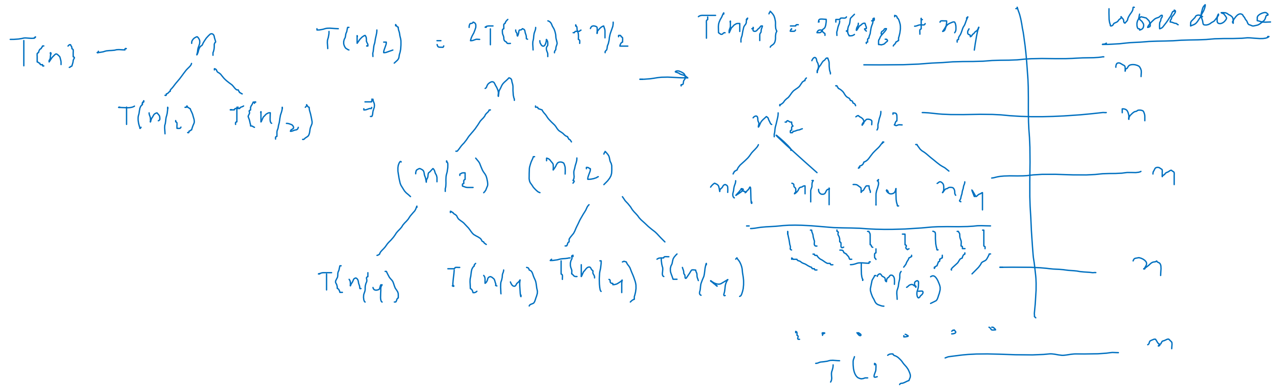
∴ Total work done: c + 2c + 4c + 8c +…+ nc

- ⇨ 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 $2^0, 2^1, 2^2, 2^3, \dots, 2^k$

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****”.***

<u>COMPARING VARIOUS FUNCTIONS TO ANALYZE TIME COMPLEXITY</u>

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 $n\log_2 2 = 2^{100}$
$2\log_2(2^{100}) = 2 \times 100 = 200$

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

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

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

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

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

① cancel common terms : n , $\log n$ | $n > \log n$ ∴ $f(x) = n^2$ 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 $n = 2^{128}$ $\quad \log_2 2^{126}$, $100\log_2\log_2 2^{128}$
$\Rightarrow 128, 100\log_2 128$
$\Rightarrow 128, 100 \times 7$
$\Rightarrow 128, 700$

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

∴ n is growing more later ∴ it's 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 : $n = 2^{1024}$, $\dfrac{1024 \times 1024}{(1024)^2}$, $\dfrac{1024 + 10}{10}$ ∴ $n\log n < n^{\log n}$ in asymptotic complexity

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

log both sides: $\frac{1}{2}\log\log n \qquad \log\log\log n$

$n = 2^{2^{10}}$
$\Rightarrow \frac{1}{2} \times 10$, 3.5

Example $f(x) = n^{\sqrt{n}}$ $g(x) = n^{\log n}$ is $f(x) = O(g(x))$? $n = 2^{2^{10}}$

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

$\therefore$ $n^{\sqrt{n}}$ 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

<u>MASTERS THEOREM</u>

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

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 $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
⇨ $T(n) = \Theta(n^{\log_2 4} \log^{0+1} n) = \mathbf{\Theta(n^2 \log n)}$

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

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

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

Example 11: $T(n) = 64T(n/8) - n^2\log n$
Here we have ‘-‘ instead of a +. This says that whenever we **don’t do $n^2\log n$ 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: $T(n) = 4T(n/2) + \log n$
Here $a = 4, b = 2, k = 0, p = 1$
$b^k = 1 \Rightarrow a > b^k$

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

Therefore, $T(n) = \Theta(n^{\log_b a})$

⇨ $T(n) = \Theta(n^{\log_2 4})$

⇨ $T(n) = \boldsymbol{\Theta(n^2)}$

Example 15: $T(n) = 2T(n/2) + \sqrt{n}$

Here a = 2, b = 2, k = 1/2, p = 0

$b^k = \sqrt{2} \Rightarrow a > b^k$

Therefore, $T(n) = \Theta(n^{\log_b a})$

⇨ $T(n) = \Theta(n^{\log_2 2})$

⇨ $T(n) = \boldsymbol{\Theta(n)}$

then $T(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_2 \mathrm{root2}})$

⇨ $T(n) = \boldsymbol{\Theta(\sqrt{n})}$

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	Example 2	Example 3

Algo(A[],1,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).
	Algo(A[],1,n)
	{ int i;
	extra
	create B[n];
	for (i=1 to n)
	B[i] = A[i];
	}
	Apart from space taken by the input, we have another array B which takes space of size n.
	∴ int i → 1space
	B[n] → n space
	Total space = 1+n or O(n)
		Algo(A[],1,n)
		{ create B[n,n] → n^2 space
		int i,j → 2 spaces
		for(i=1 to n)
		for(j=1 to n)
		B[i,j] = A[i] → computation so NO SPACES
		}
		Total = (n^2+2)
		or $\boxed{O(n^2)}$

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.

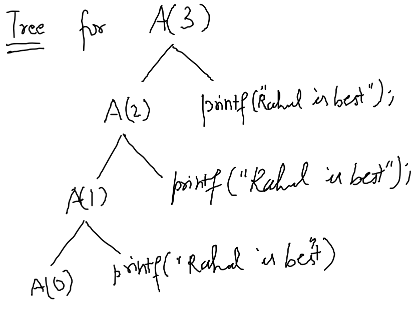
SPACE COMPLEXITY OF RECURSIVE ALGORITHMS

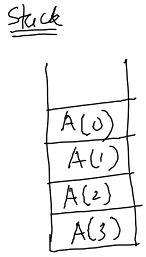
There are various ways to analyze the recursion algoirthms' space complexity.

*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.

```
A(n)
{ if (n ≥ 1)
  { A(n-1);
     printf ("Rahul is best");
  }
}
```





for n	size of stack
3	4
2	3
1	2
0	1
⋮	
n	n+1

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

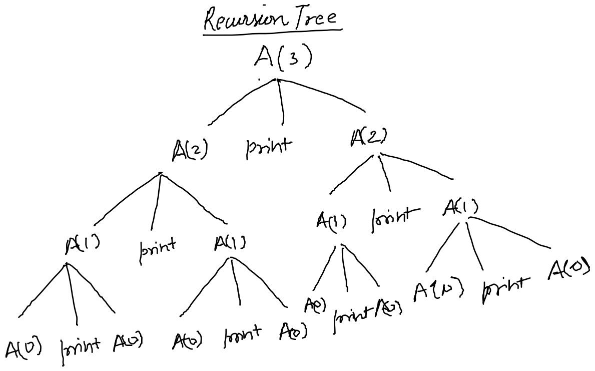
Questions that can be asked:

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:

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

$$T(n) = 2^2 T(n-2) + 2c + c \quad \text{---} \boxed{4}$$

Substituting $\boxed{3}$ in $\boxed{4}$ we get

$$T(n) = 2^2(2T(n-3) + c) + 2c + c$$

$$T(n) = 2^3 T(n-3) + 2^2 c + 2^1 c + 2^0 c$$

$$\Rightarrow T(n) = 2^k T(\underbrace{n-k}_{\downarrow \text{is going to be } 0.}) + 2^{k-1} c + 2^{k-2} c + \dots + c$$

I will slop when

It willbe 0 when $\underline{n=k}$

$$\therefore T(n) = 2^n T(0) + 2^{(n-1)} c + 2^{(n-2)} c + \dots + c$$

$$T(n) = 2^n c + 2^{(n-1)} c + 2^{(n-2)} c + \dots + c \Rightarrow c(\underbrace{2^n + 2^{(n-1)} + 2^{(n-2)} + 2^{(n-3)} \dots 1}_{\underline{gp}})$$

$$\Rightarrow T(n) = O(2^{n+1}) = \underline{\underline{T(n) = O(2^n)}}$$

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