Analysis of Algorithms(Background)

Sum of n natural numbers

$\frac{n \times (n+1)}{2}$

Input: n = 3

Output: 6 // 1+2+3

```
for i in range(1, n+1)
    sum += i;
```

Input: n=5

Output: 15 //1+2+3+4+5

```
for i : 1 to n+1
    for j 1 to i
        sum += 1
```

Analysis of Algorithms(Background)

```
def fun(n):
    return n*(n+1)/2
```

```
def fun3(n):
    sum = 0
    for i in range (1,n+1):
        for j in range (1,i+1):
            sum = sum+1
    return sum
```

```
def fun2(n):
    sum=0
    for i in range(1,n+1):
        sum = sum + i
    return sum
```

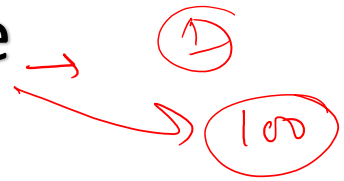
1+(1+1)+(1+1+1)+.....

1+2+3.....

Analysis of Algorithms(Background)

Which program is better depends upon several factors like:

- machine on which program is running
- the programming language it is written and
- even the load on the machine

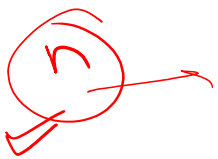


So we can't find figure out which program is better by running it and to solve this we have a theoretical solution a mathematical solution that is **asymptotic analysis**

Analysis of Algorithms(Background)

Asymptotic Analysis: (Theoretical Analysis)

- No dependency on machine, programing language, etc.
- We do not have to implement all ideas/algorithms.
- Asymptotic analysis is about measuring order of growth in forms of input size.



Analysis of Algorithms(Background)

```
def fun1(n):
    return n*(n+1)/2
```

```
def fun2(n):
    sum=0
    for i in range(1,n+1):
        sum = sum + i
    return sum
```

```
def fun3(n):
    sum = 0
    for i in range (1,n+1):
        for j in range (1,i+1):
            sum = sum+1
    return sum
```

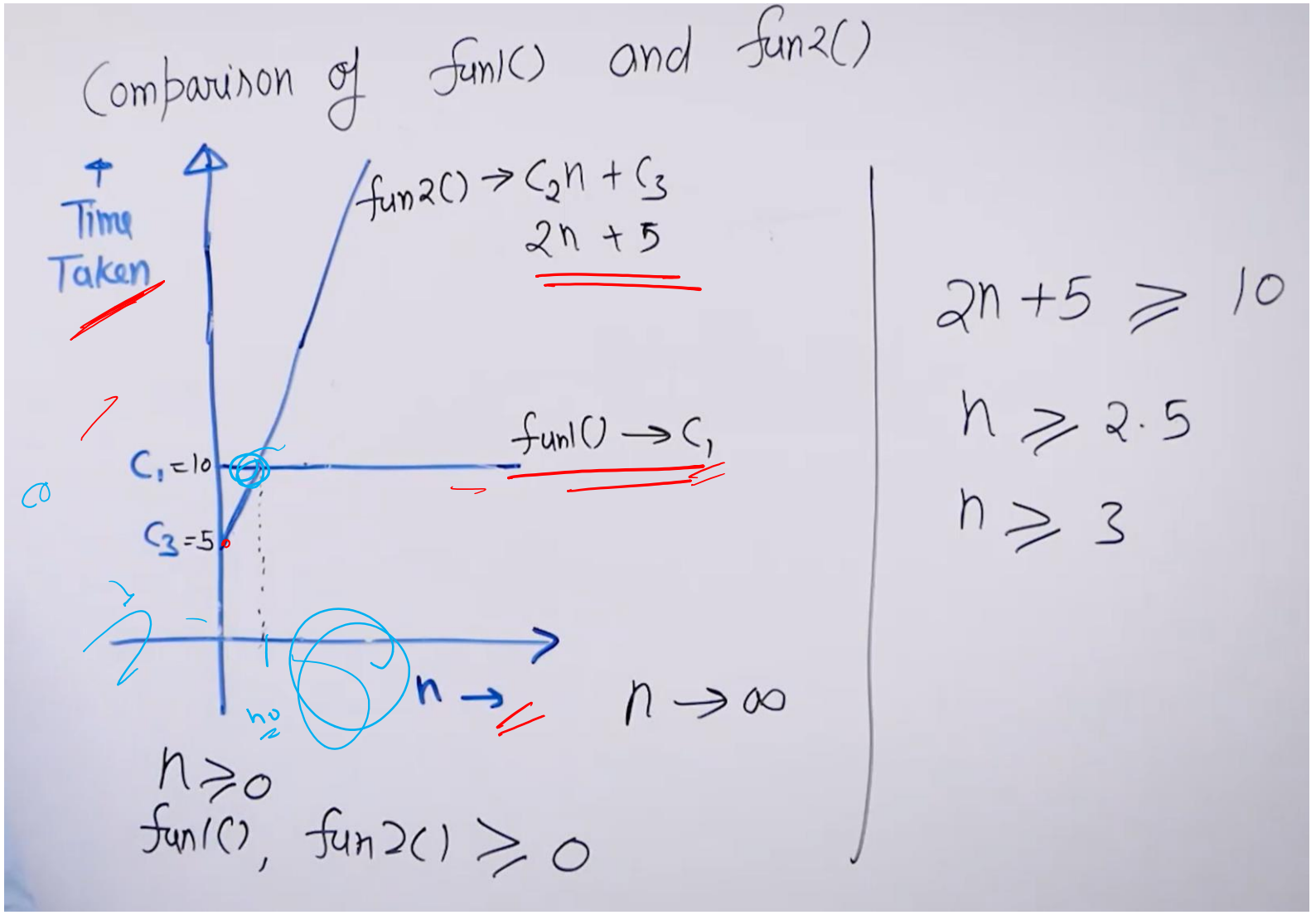
Order of Growth :

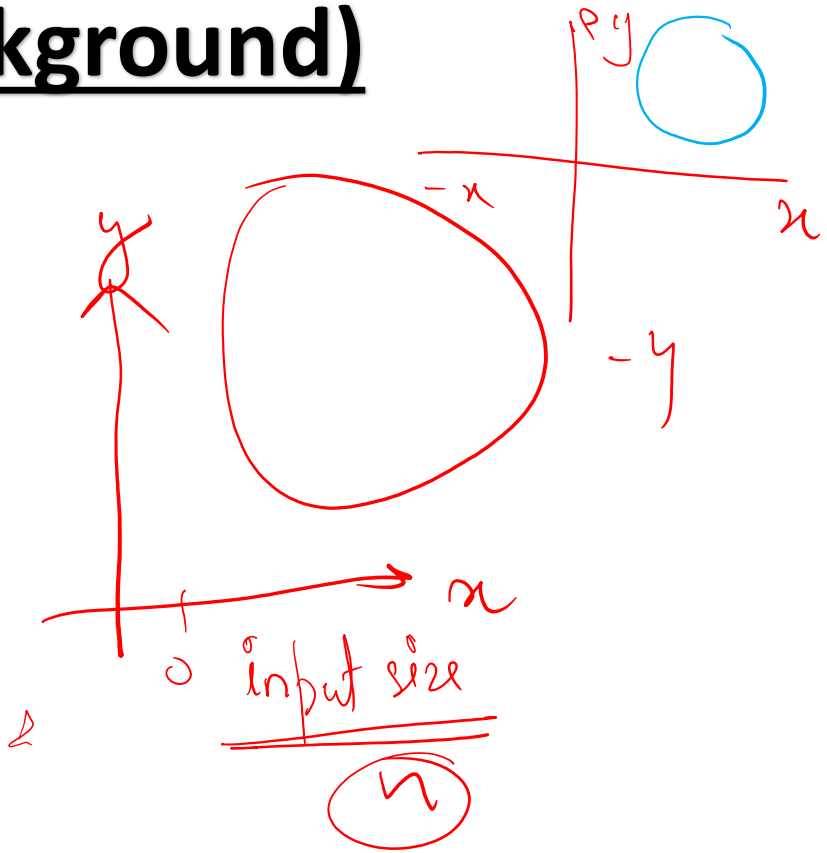
fun1() ➔ C_1

fun2()➔ $C_2n + c_3$

fun2()➔ $C_3n^2 + c_4n + c_5$

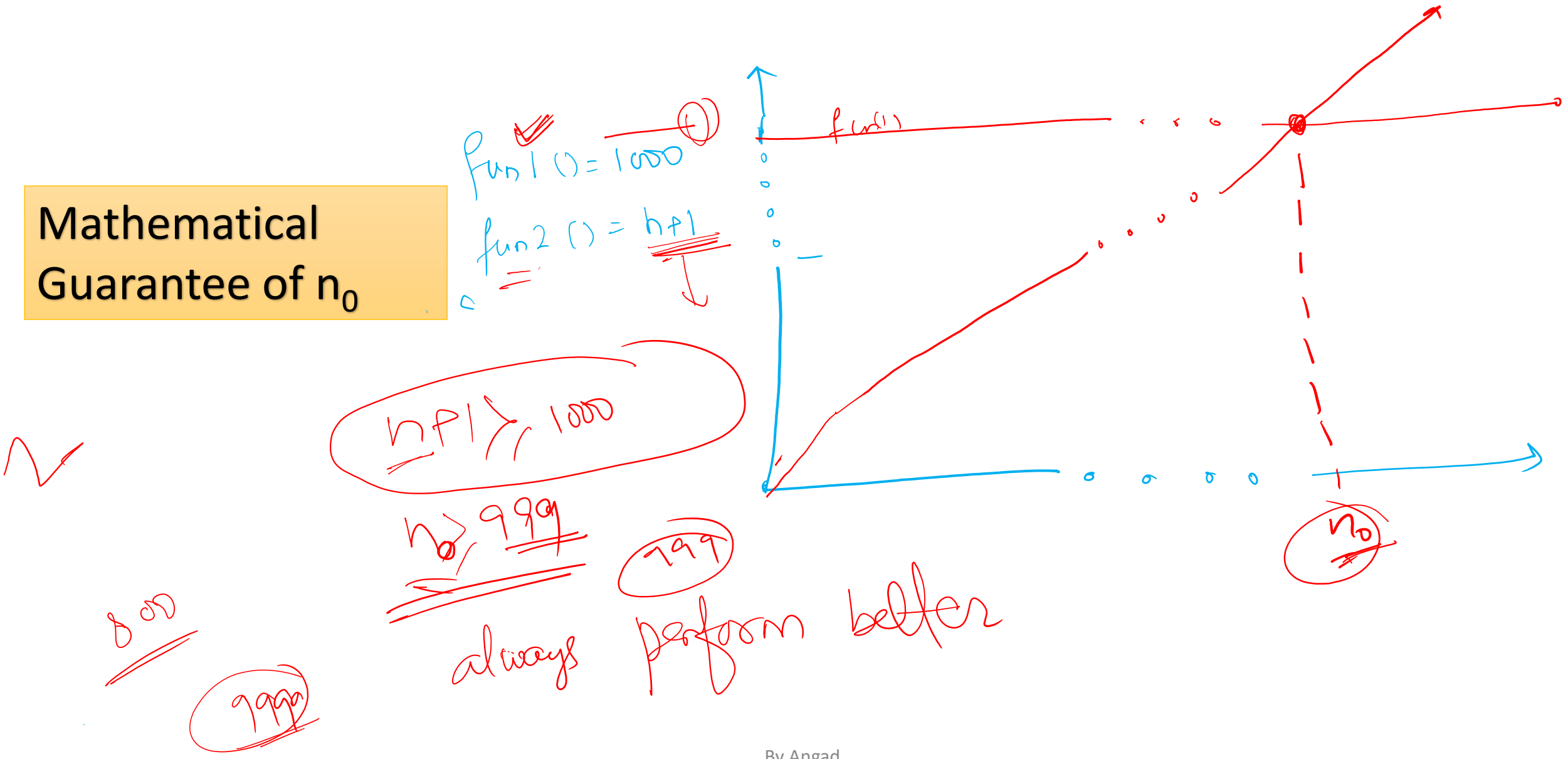
Analysis of Algorithms(Background)





Analysis of Algorithms(Background)

Mathematical Guarantee of n_0



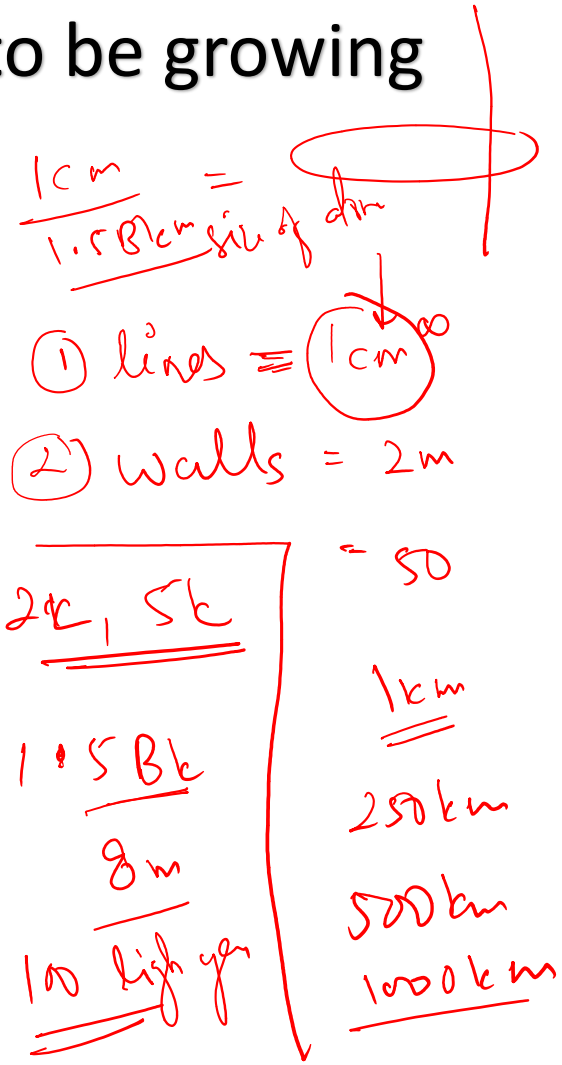
Order of Growth

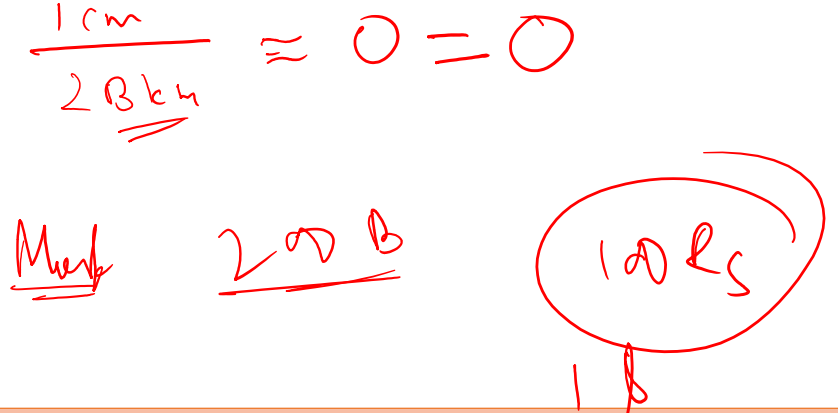
A function f(n) is said to be growing faster than g(n) if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

OR

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$





Here functions f(n) and g(n) represents **Time Taken** where

- n >= 0
- f(n),g(n) >= 0

Analysis of Algorithms(Background)

```
def printFun3(n):
    if n <= 1:
        return 0
    else:
        return 1 + printFun3(n/2)
```

Order of Growth

A function f(n) is said to be growing faster than g(n) if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Let's take two functions

- f(n) = $n^2 + n + 6$
- g(n) = $2n + 5$

$$= \lim_{n\to\infty} \frac{(2n+5)/n^2}{(n^2+n+6)/n^2}$$

$$= \lim_{n\to\infty} \frac{2/n + 5/n^2}{1 + 1/n + 6/n^2}$$

$$= \lim_{n\to\infty} \frac{0+0}{1+0+0} = \frac{0}{1}$$

$$= 0$$

Order of Growth

Direct way to find to find and compare growths

1. Ignore lower order term
2. Ignore leading term constant

Examples:

$f(n)=2n^2+n+6$, Order of Growth: n^2 (Quadratic)

$g(n)= 100n+3$, Order of Growth: n (linear)

How do we compare term ?

$$C < \overline{\log\log N} < \overline{\log N} < N^{1/3} < N^{1/2} < N < N^2 < N^3 < N^4 < 2^N < N^N$$

Order of Growth : Exercise

- $f(n) = C_1 \log(n) + C_2$
- $g(n) = C_3 n + C_4 \log\log(n) + C_5$

- $f(n) = C_1 n^2 + C_2 n + C_3$
- $g(n) = C_4 n \log(n) + C_5 n + C_6$

$$C < \log\log N < \log N < N^{1/3} < N^{1/2} < N < N^2 < N^3 < N^4 < 2^N < N^N$$

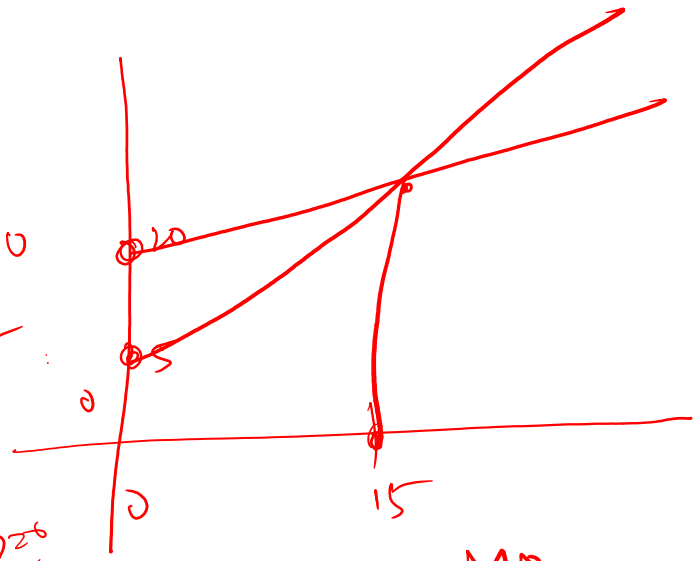
Big 0 Notation(Upper Bound on Order of Growth)

We say f(n)= 0(g(n)) iff{if and only if} there exist constants C and n_0 such that:

f(n) <= Cg(n) for all n >= n_0

Example:

f(n) = 2n+3 can be written as 0(n)



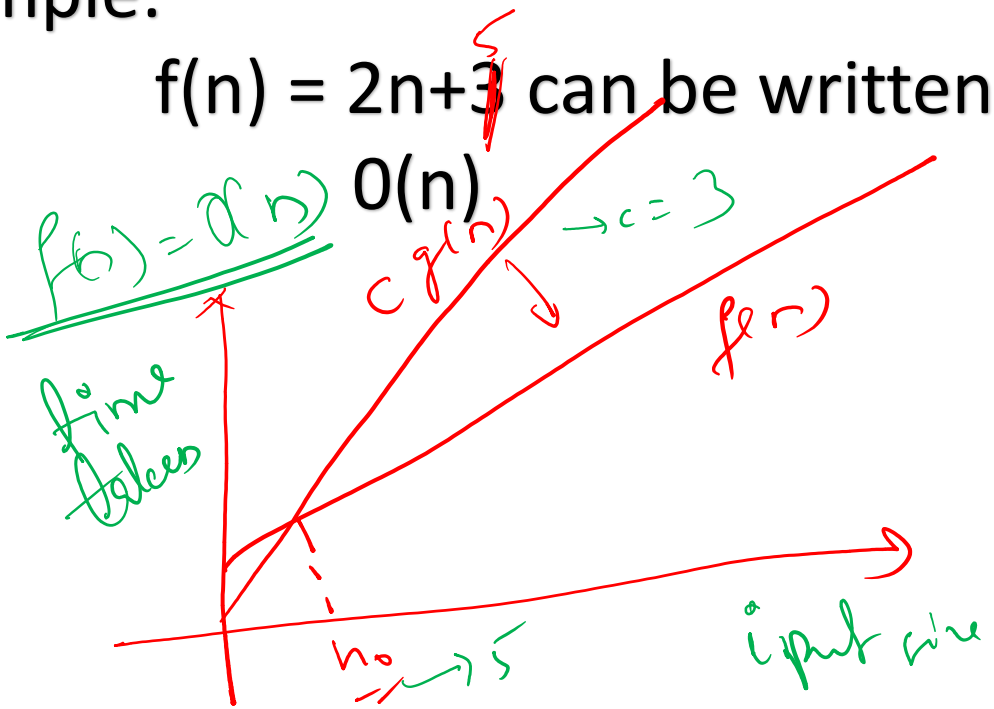
Big 0 Notation(Upper Bound on Order of Growth)

$f(n) <= Cg(n)$ for all $n >= n_0$

How to find C and N_0

Example:

f(n) = 2n+3 can be written as 0(n)



$g(n) = n$

$C = 3, \quad n_0 = 5$

$f(n) = 2n + 3$

$Cg(n) = 3n$

$n \geq 5$

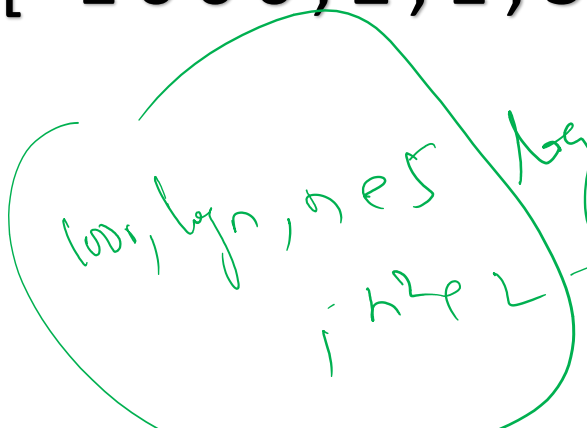
Topic

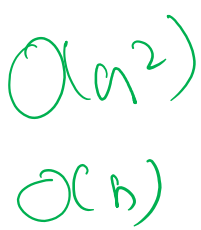
ExtraRequiredInformation

```
def printFun3(n):
    if n <= 1:
        return 0
    else:
        return 1 + printFun3(n/2)
```

Big 0 notation (upper bound on order of growth)

- [n/4 , 2n+3, n/100 + log(n) , n +1000, n/1000,100,log(n) + 100+ ……] ← O(n)
-
- [n^2 + n, $2n^2$,n^2 + 1000n , n^2 + 2log(n) ,……] ← $O(n^2)$
- [1000,2,1,3,10000,100000,…] ← O(1)





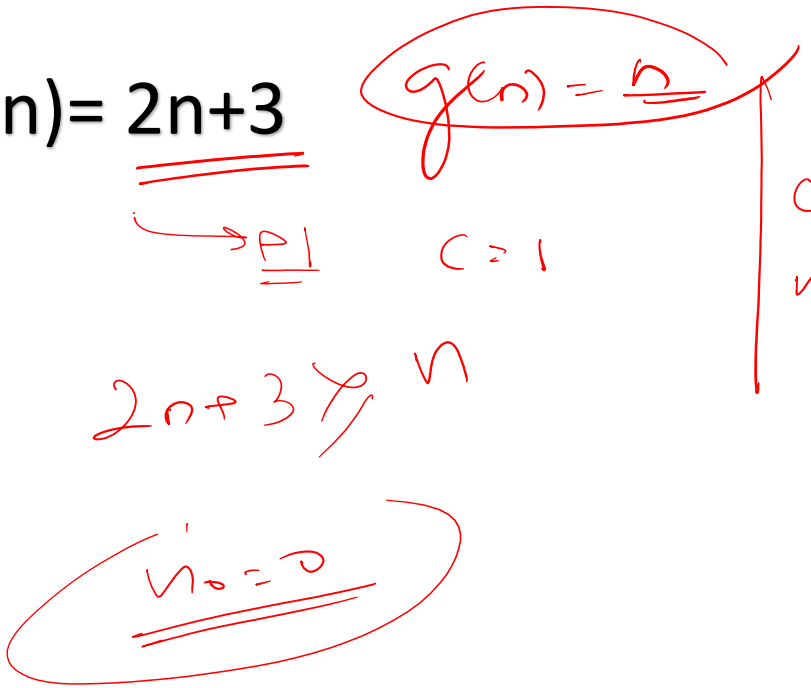
Omega Notation (Lower Bound)

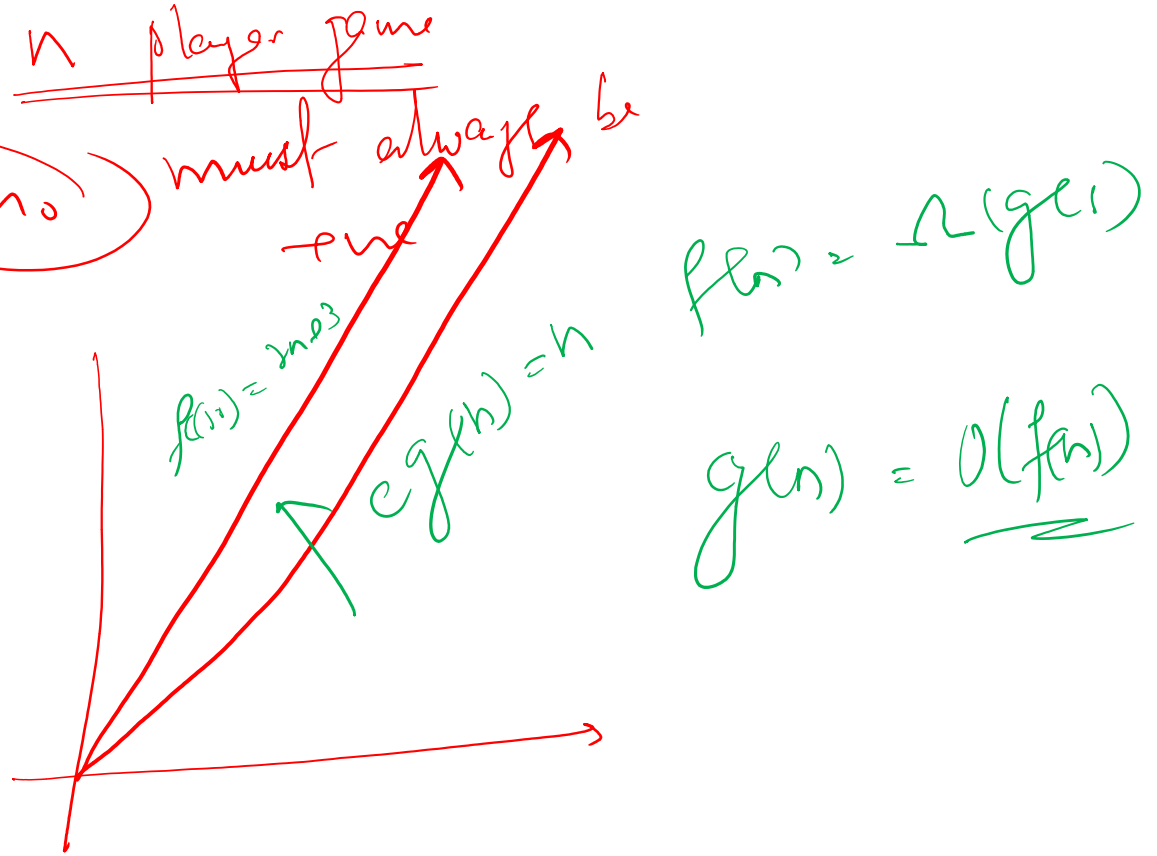
F(n)= Ω(g(n)) iff there exist positive constants C and N_0 such that

0 <= Cg(n) <= f(n) for all n>= n_0

Example:

f(n)= 2n+3





Omega Notation(Lower Bound)

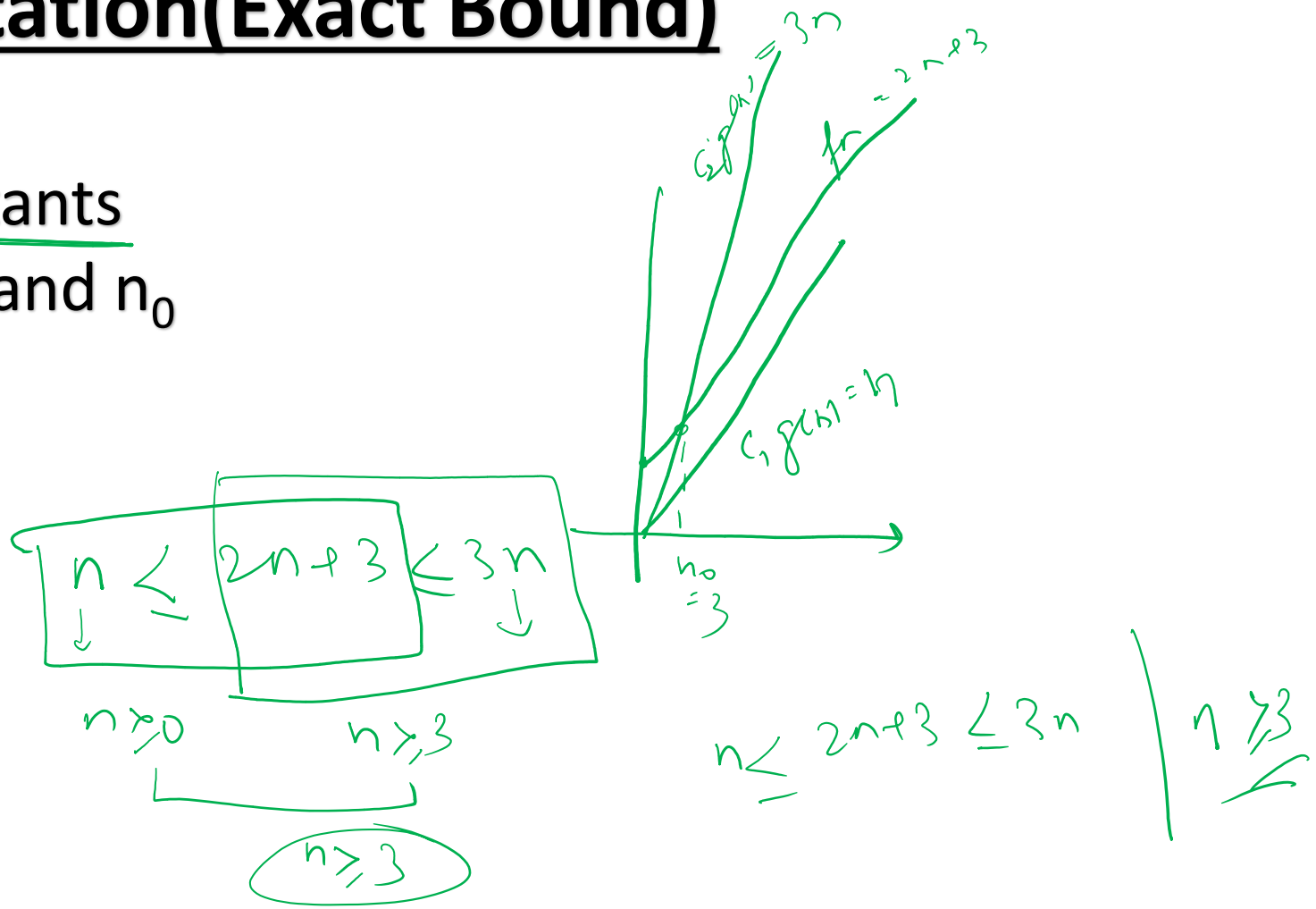
- $\{ n/4, n/2, 2n, 3n, 2n+3, n^2, 2n^2, ..., n^n\} \in \Omega(n)$
- If $f(n) = \Omega(g(n))$ then $g(n) = O(f(n))$
- Omega notation is useful when we have lower bound on time complexity.

Theta Notation(Exact Bound)

f(n)= θ(g(n)) iff there exist constants C_1 , C_2 (where $c_1 > 0$ and $c_2 > 0$) and n_0 (where n_0 >=) such that:

C_1 g(n) <= f(n) <= C_2 g(n)

For all n>= n_0



Example:

f(n) =2n+3 : θ (n)

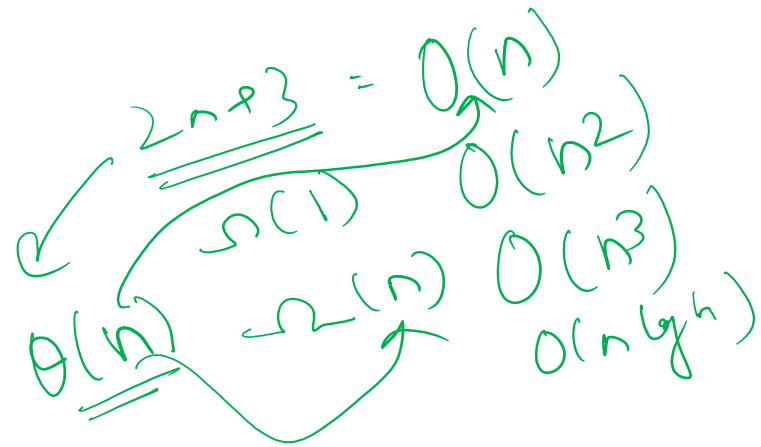
Theta Notation: Finding Order of Growth

Direct Method

- $1000n^2 + 100n\log^n + 2n$: $\theta(n^2)$
- $200n^3 + 30n + 5$: $\theta(n^3)$
- $2000n + 2\log^n$: $\theta(n)$

Theta notation

1. if $f(n) = \theta(g(n))$
   then $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$
   and $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$



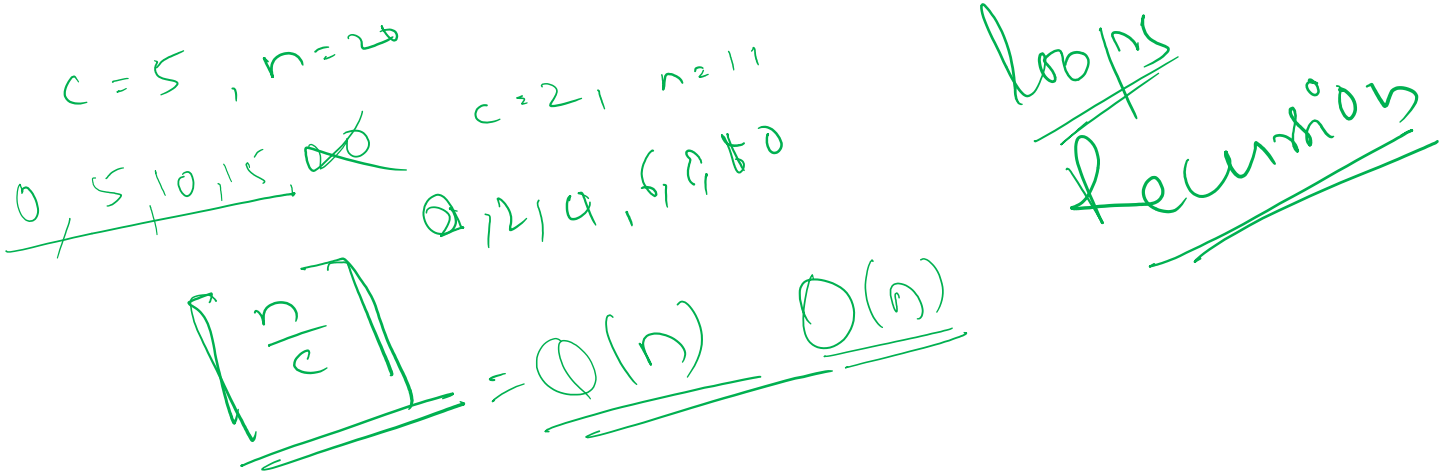
2. Represents Exact Bound

3. $[100, 10^5, \log 2000, \ldots\ldots\ldots\ldots] \in \theta(1)$
   $[100n, 2n+\log(n), 5n+3, \ldots\ldots] \in \theta(n)$
   $[2n^2, (2n^2/4)+ 5n\log(n), \ldots\ldots] \in \theta(n^2)$

Analysis of Common Loops(First Example)

```
def loopExampleA(n):
# First Example Loop
    i = 0
    c = 5
    while i < n:
        i = i + c
        # some other constant time work
```

```
def loopExampleA(n):
    # Equivalent for Loop
    for i in range(0, n, c):
        pass
        # some constant work
```



Analysis of Common Loops(Second Example)

```
def loopExampleA(n):
# Second Example Loop
    i = 0
    c = 5
    while i < n:
        i = i - c
        # some other constant time work
```

```
def loopExampleA(n):
    # Equivalent for Loop
    for i in range(0, n, -c):
        pass
        # some constant work
```

$c = 4$, $n = 20$

$\left\lceil \frac{n}{c} \right\rceil \rightarrow O(n)$

Analysis of Common Loops(Third Example)

```
def loopExampleA(n):
# Second Example Loop
    i = 0
    c = 5
    while i < n:
        i = i * c
        # some other constant time work
```

n=32

C=2

1,2,3,4,5,16

$1, c, c^2, c^3, c^4 \ldots c^{k-1}$

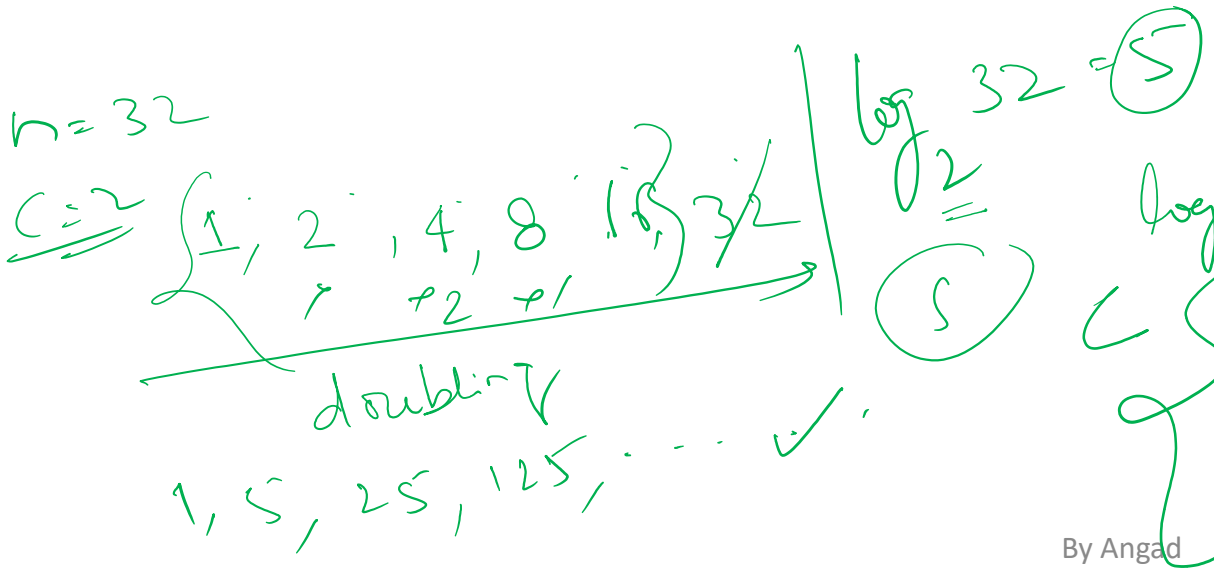
$1, c, c^2, \ldots c^{k-1}$

$c^{k-1} < n$

$k < \log^{(n)} + 1$

$\log_c(c^{k-1}) = \log_c(n)$

$\theta(\log(N))$

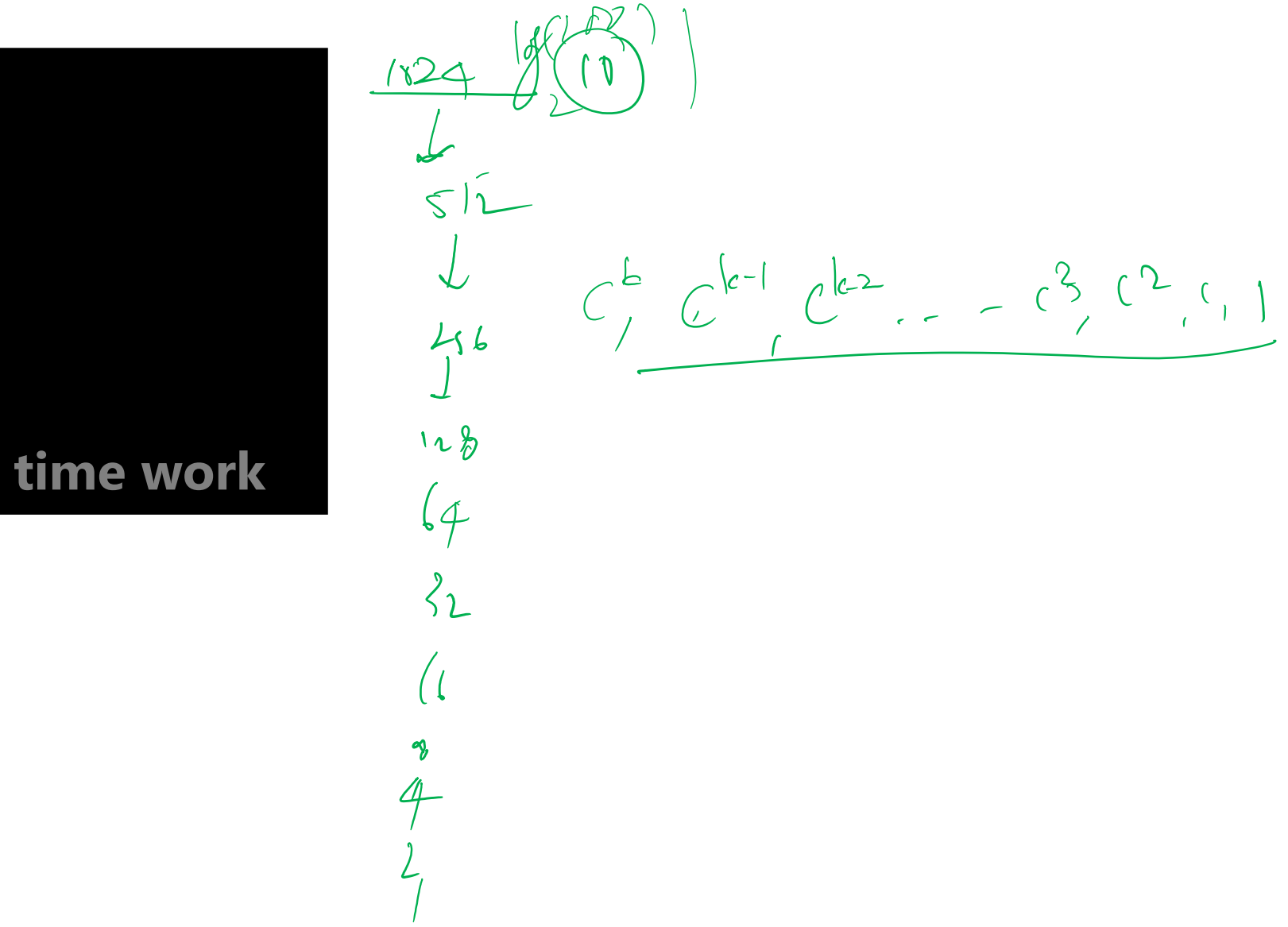


$k < \log(n) + 1$

$O(\log(n))$

Analysis of Common Loops(Fourth Example)

```
def loopExampleA(n):
# Second Example Loop
    i = 0
    c = 2
    while i < n:
        i = i / c
        # some other constant time work
```



Analysis of Common Loops(Fifth Example)

```
def loopExampleA(n):
# Second Example Loop
    i = 0
    c = 5
    while i < n:
        i = i ** c
        # some other constant time work
```

Analysis:

$2, 2^c, (2^c)^c \ldots ((2^c)^c)^c$

$2, 2^c, 2^{c2} \ldots 2^{ck-1}$

$2^{c^{k-1}} < n$

$c^{k-1} < \log_2 n$

$K-1 < \log_c \log_2 n$

$K < \log_c \log_2 n + 1$

$2, 4, 16, 256, \rightarrow$ | $2^{c^0}, 2^{c^1}, 2^{c^2}, 2^{c^3}, 2^{c^4} \ldots 2^{c^{k-1}}$

$k < \log_2 \log_2 (b) + 1$

Analysis of common loops

Example 6 : Subsequent Loops

```
def loopExampleB(n):
    i = 0
    while i < n:
        i = i + 2

    i = 1
    while i < n:
        i = i * 3

    i = 1
    while i < 100:
        i = i + 1
```

$O(n)$

$O(\log n)$

$O(1)$

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

$O(n)$

Analysis of common loops

Example 7 : Nested loop

```
def loopExample7(n):
    i = 0
    while i < n:
        j = 1
        while j < n:
            j = j * 2
            # Some Constant Work
        i = i + 1
        # Some Constant  Work
```

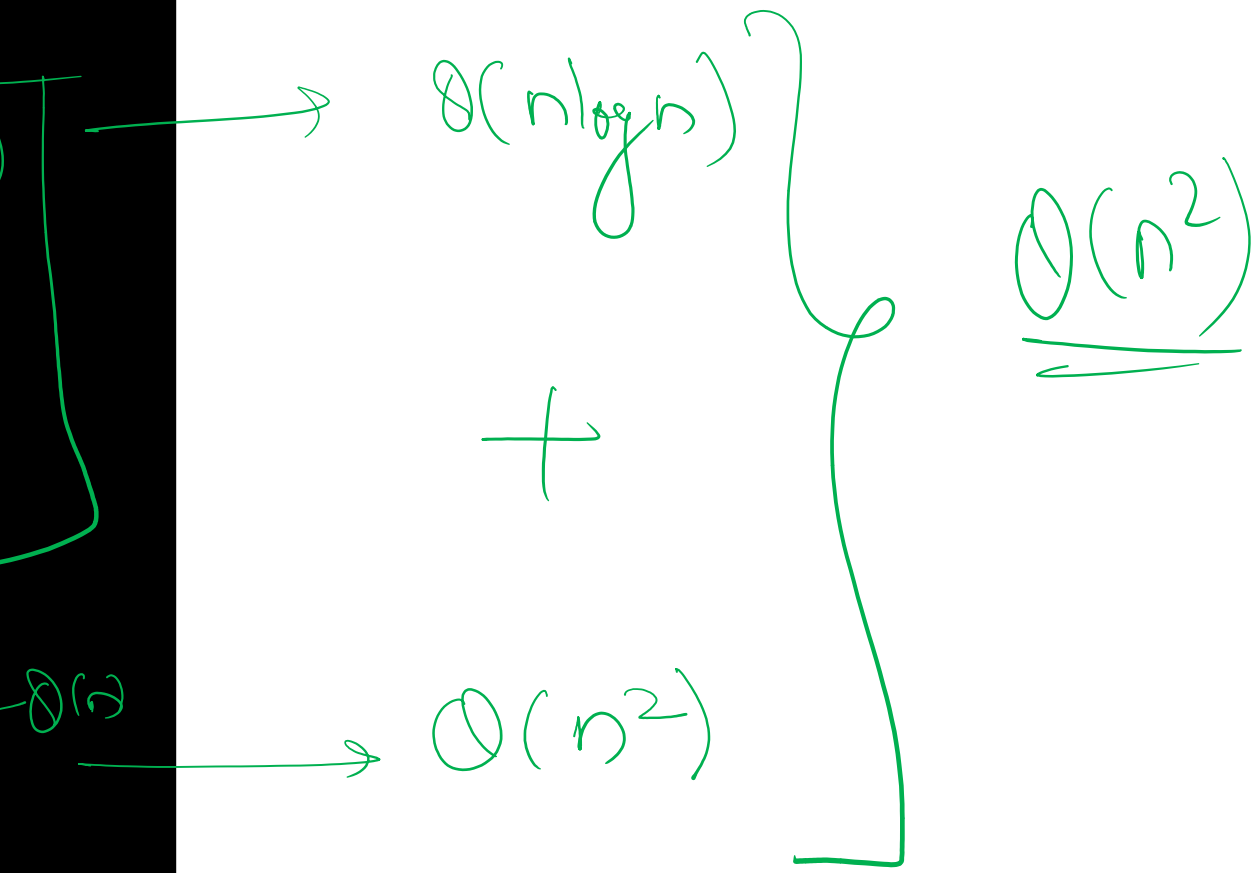
$O(n) \times O(\log n)$

$O(n \log(n))$

Analysis of common loops

Example 8 : mixed loops

```
def loopExample8(n):
    i = 0
    while i < n:
        j = 1
        while j < n:
            j = j * 2
        i = i + 1

    i = 0
    while i < n:
        while j < n:
            j = j + 1
        i = i + 1
```



Analysis of common loops

Example 9 : multiple input

```
def loopExample9(m, n):
    i = 0
    while i < n:
        j = 1
        while j < n:
            j = j * 2
        i = i + 1

    i = 0
    while i < m:
        while j < m:
            j = j + 1
        i = i + 1
```

$O(n)$

$O(\log n)$

$O(n \log n)$

$+$

$O(m)$

$O(m)$

$O(m^2)$

$O(n \log n + m^2)$

$O(E + V)$

Analysis of Recursion

```
def recursionExampleA(n):
    if n == 1:
        return

    for i in range(n):
        print("Angad")

    recursionExampleA(n / 2)
    recursionExampleA(n / 2)
```

Recurrence Relation

TC

$T(n) = 2T(n/2) + \theta(n)$

$T(1) = \theta(1)$

$T(n) = T(n/2) + T(n/2) + \theta(n)$

$T(n) = 2T(n/2) + \theta(n)$

$T(1) = \theta(1)$

Analysis of Recursion(#2)

```
def recursionExampleB(n):
    if n == 1:
        return

    print("Angad")

    recursionExampleA(n / 2)
    recursionExampleA(n / 2)
```

$T(n) = 2T(n/2) + \theta(1)$

$T(1) = \theta(1)$

$T(n) = 2\,T(n/2) + \theta(1)$

$T(n) = 2\,T(n/2^2) + \theta(1)$

$T(1) = \theta(1)$

Analysis of Recursion(#3)

```
def recursionExampleC(n):
    if n == 1:
        return
    print(n)

    recursionExampleC(n - 1)
```

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

Analysis of Recursion

$$T(n) = 2T(n/2) + Cn$$
$$T(1) = c$$

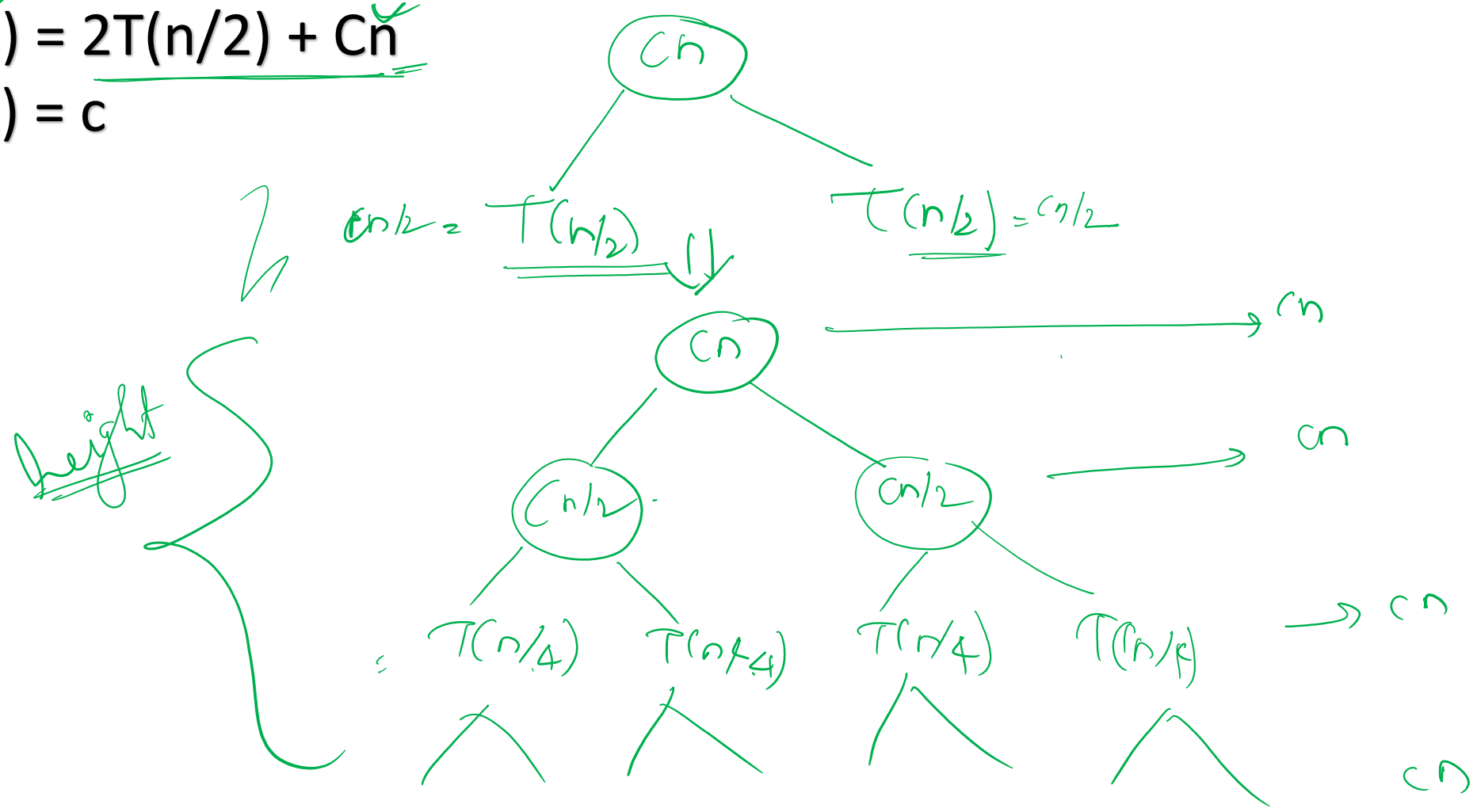
Recursion Tree Method

1. We write non-recursive part as root of tree and recursive part as children
2. We keep expanding children until we see a pattern

Analysis of Recursion

$T(n) = 2T(n/2) + Cn$

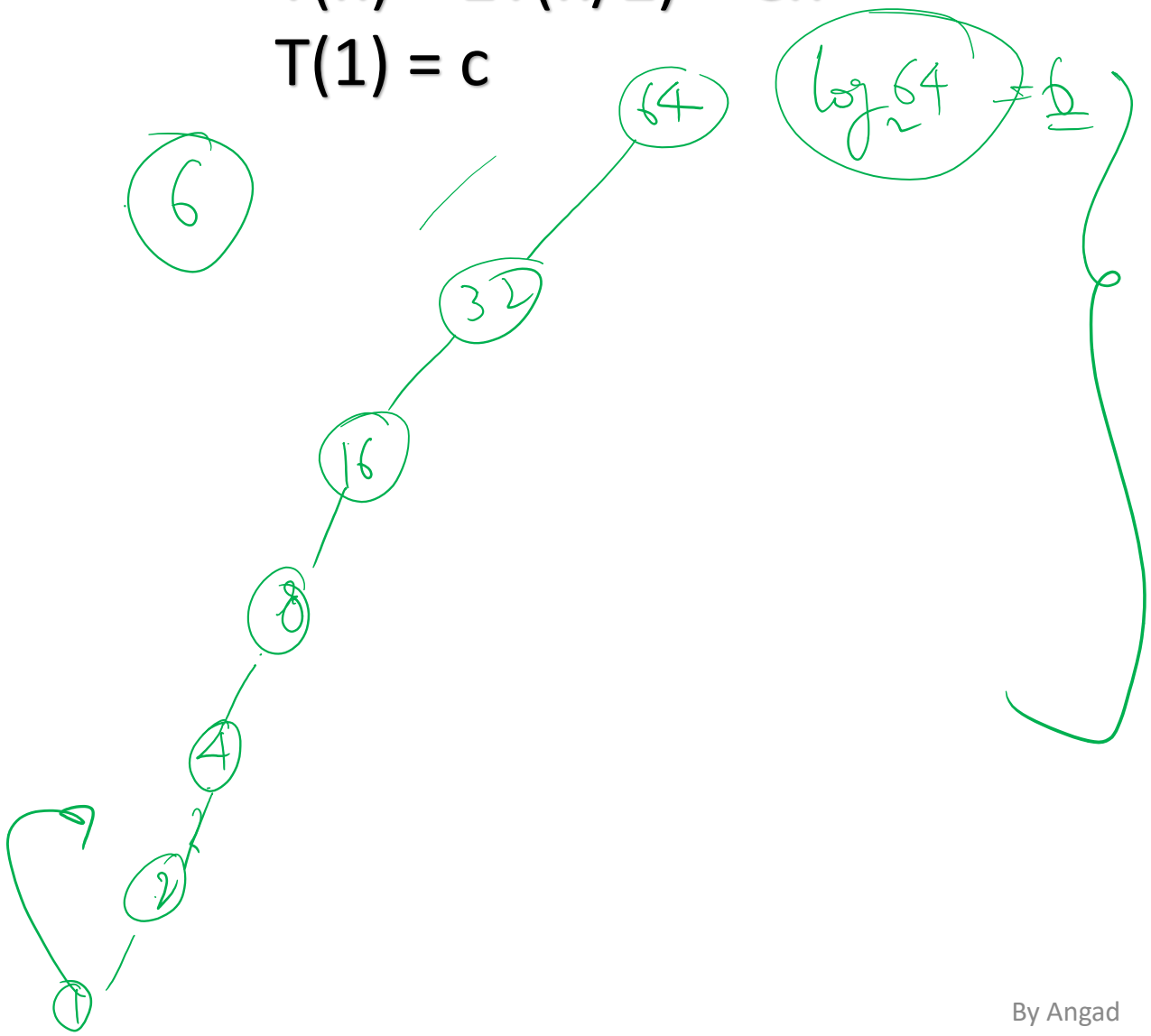
$T(1) = c$



Analysis of Recursion

$T(n) = 2T(n/2) + Cn$

$T(1) = c$



$\log_2 64 = 6$

$Cn + Cn + Cn + \dots$

height times

$\log(n)$ times

$\log(n)$

$4Cn$

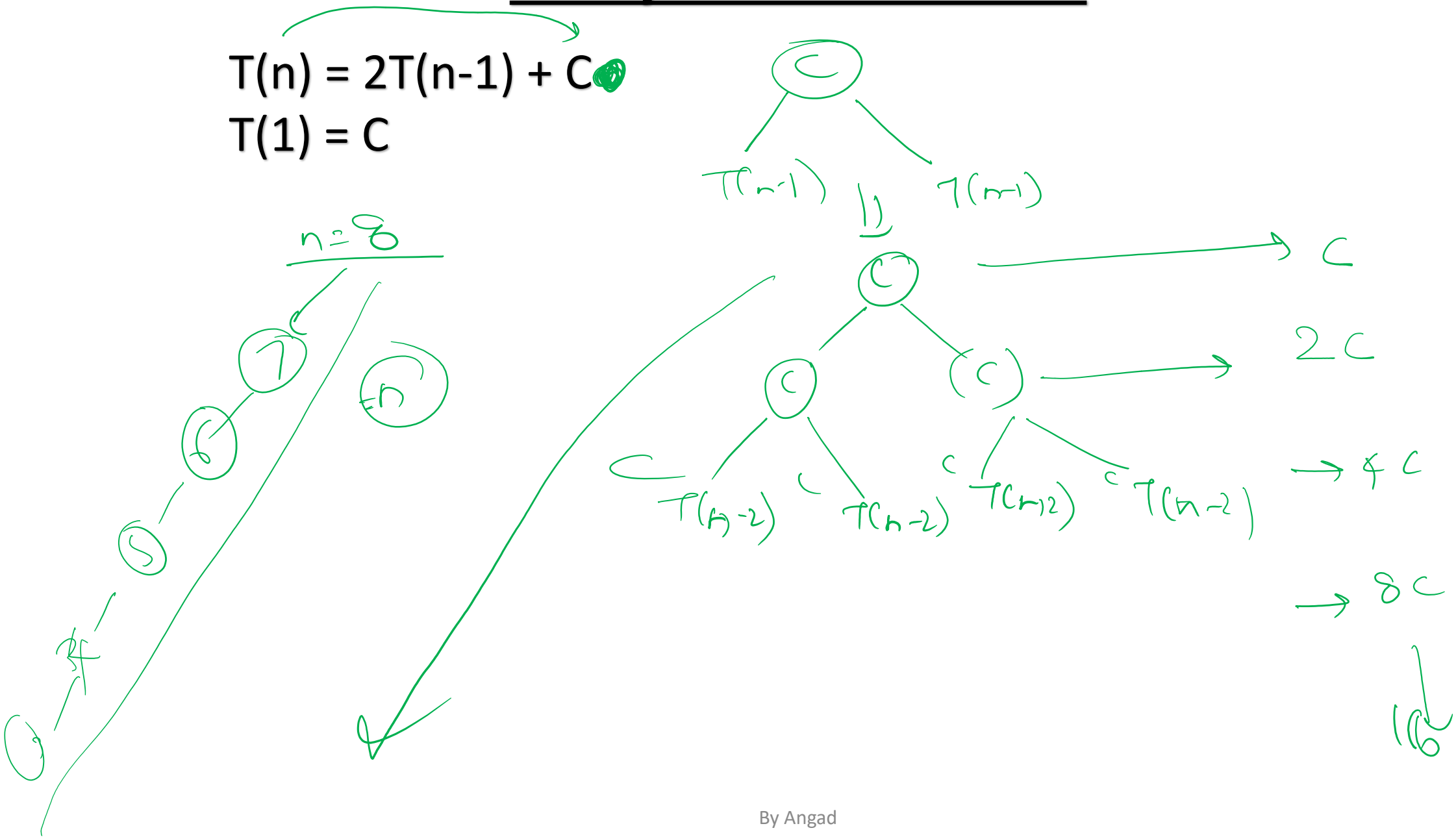
$Cn \times \log(n)$

$\Theta(n \log(n))$

Analysis of Recursion

$T(n) = 2T(n-1) + C$

$T(1) = C$



Analysis of Recursion

$T(n) = 2T(n-1) + C$

$T(1) = C$

$C + 2C + 4C + 8C + \cdots$ — n times

$= C\{1 + 2 + 4 + 8 + 16 + \cdots\}$

AP

GP

$= \frac{a \times (r^n - 1)}{r - 1}$

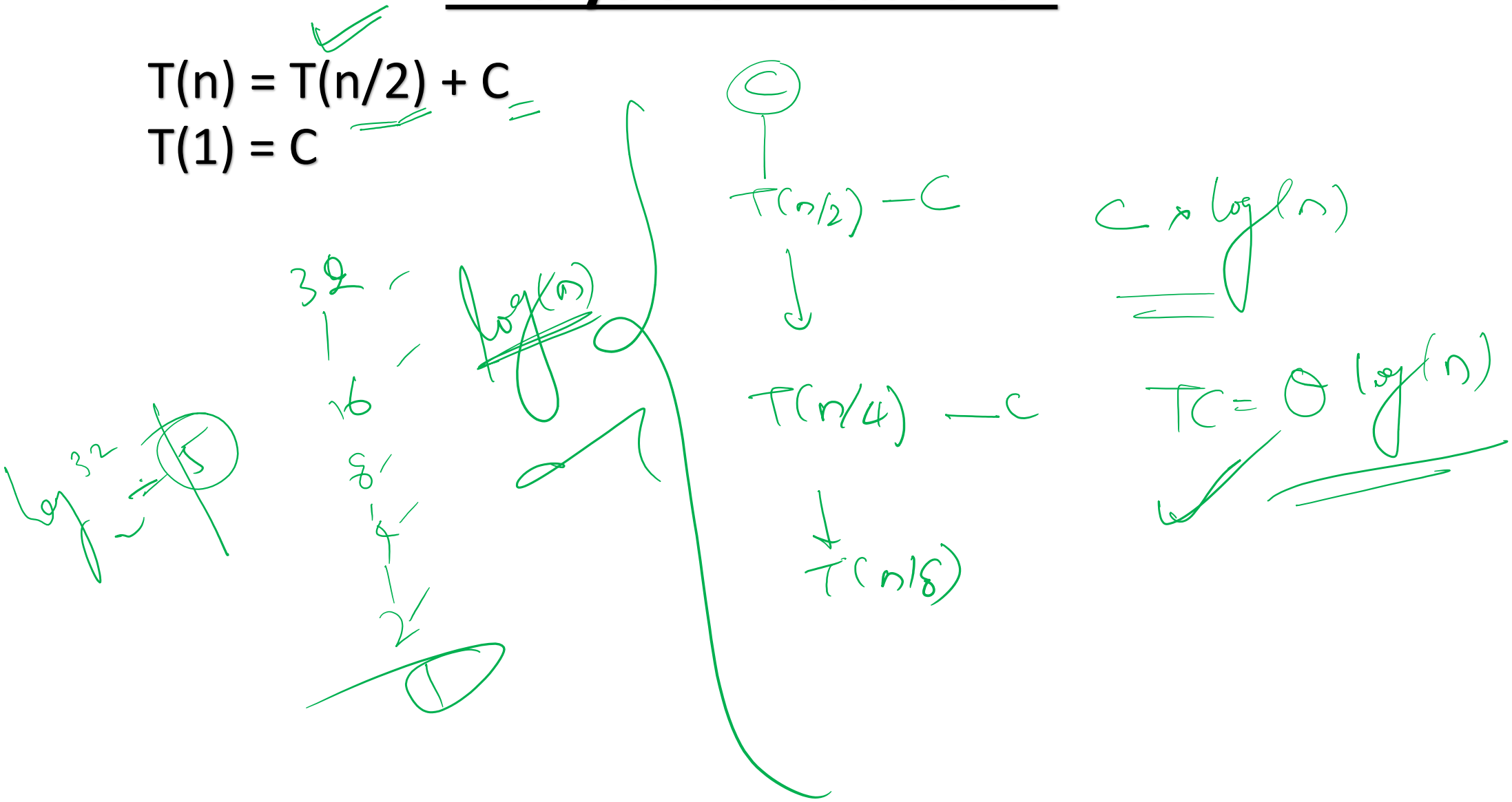
$= \frac{1 \times (2^n - 1)}{2 - 1}$

$= 2^n - 1$ $\Theta(2^n)$

Analysis of Recursion

$T(n) = T(n/2) + C$

$T(1) = C$



Analysis of Recursion

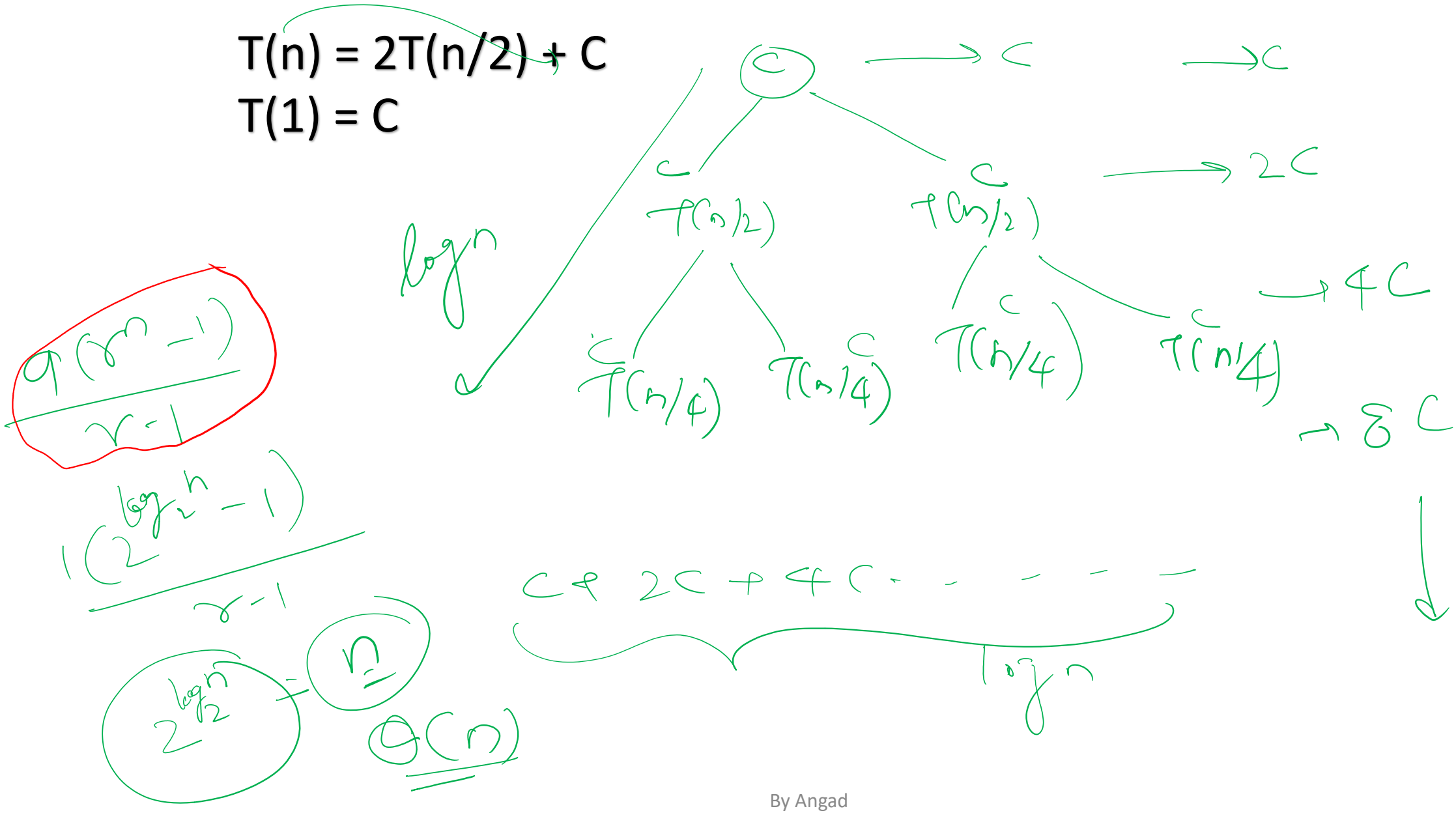
$T(n) = T(n/2) + C$

$T(1) = C$

Analysis of Recursion

$T(n) = 2T(n/2) + C$

$T(1) = C$



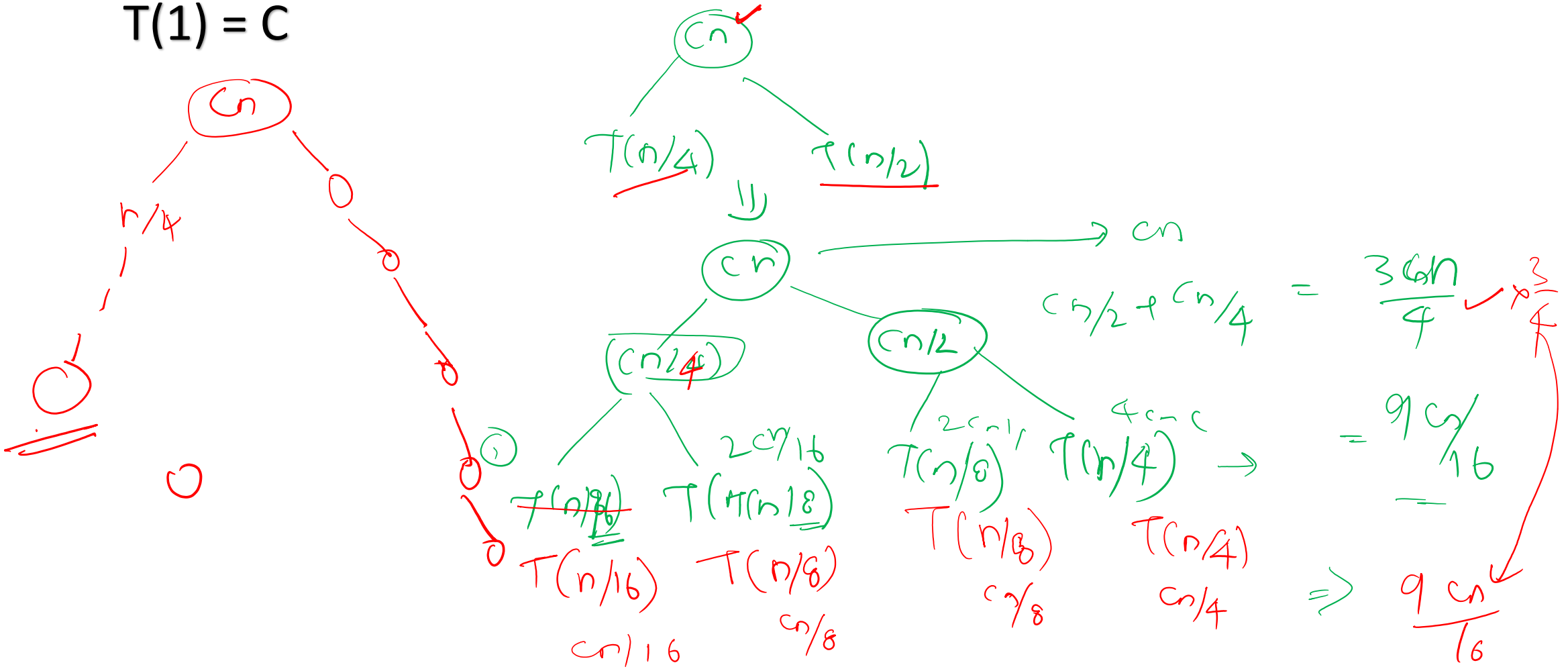
Analysis of Recursion

$$T(n) = 2T(n/2) + C$$
$$T(1) = C$$

Analysis of Recursion

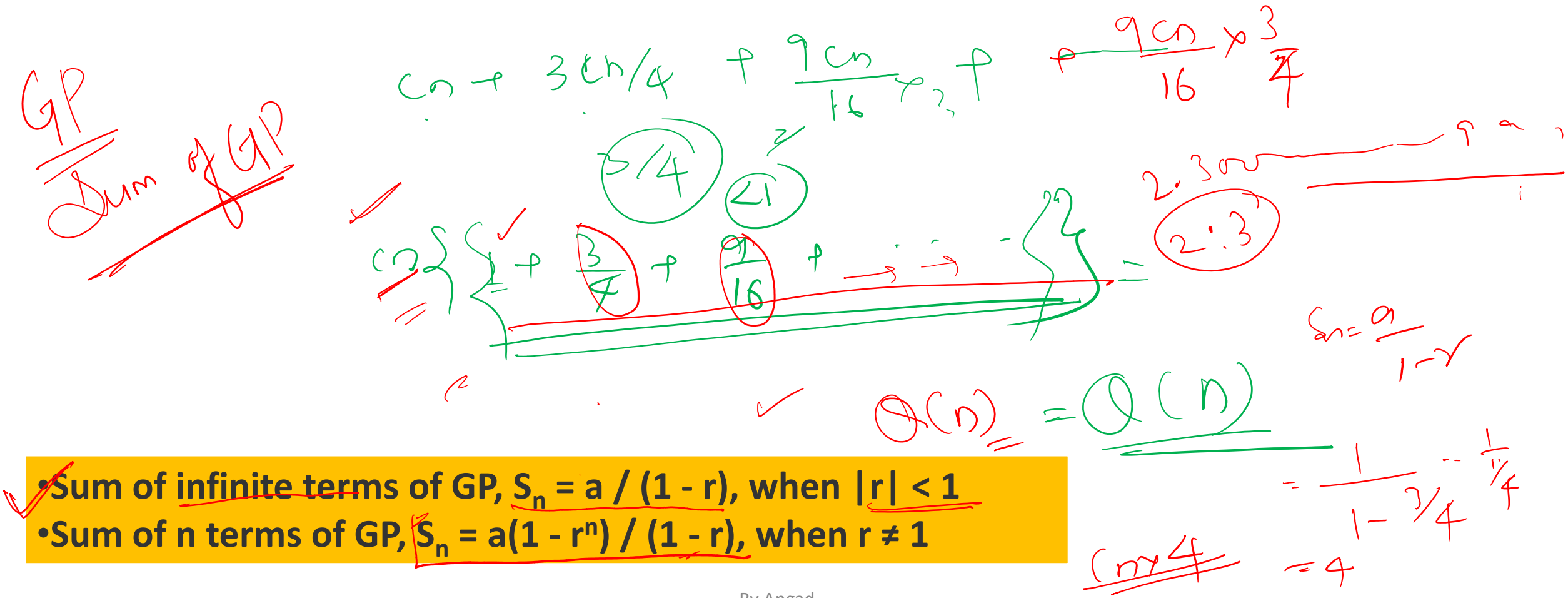
$T(n) = T(n/4) + T(n/2) + Cn$

$T(1) = C$



Analysis of Recursion

$$T(n) = T(n/4) + T(n/2) + Cn$$
$$T(1) = C$$



- **Sum of infinite terms of GP, $S_n = a / (1 - r)$, when $|r| < 1$**
- **Sum of n terms of GP, $S_n = a(1 - r^n) / (1 - r)$, when $r \neq 1$**

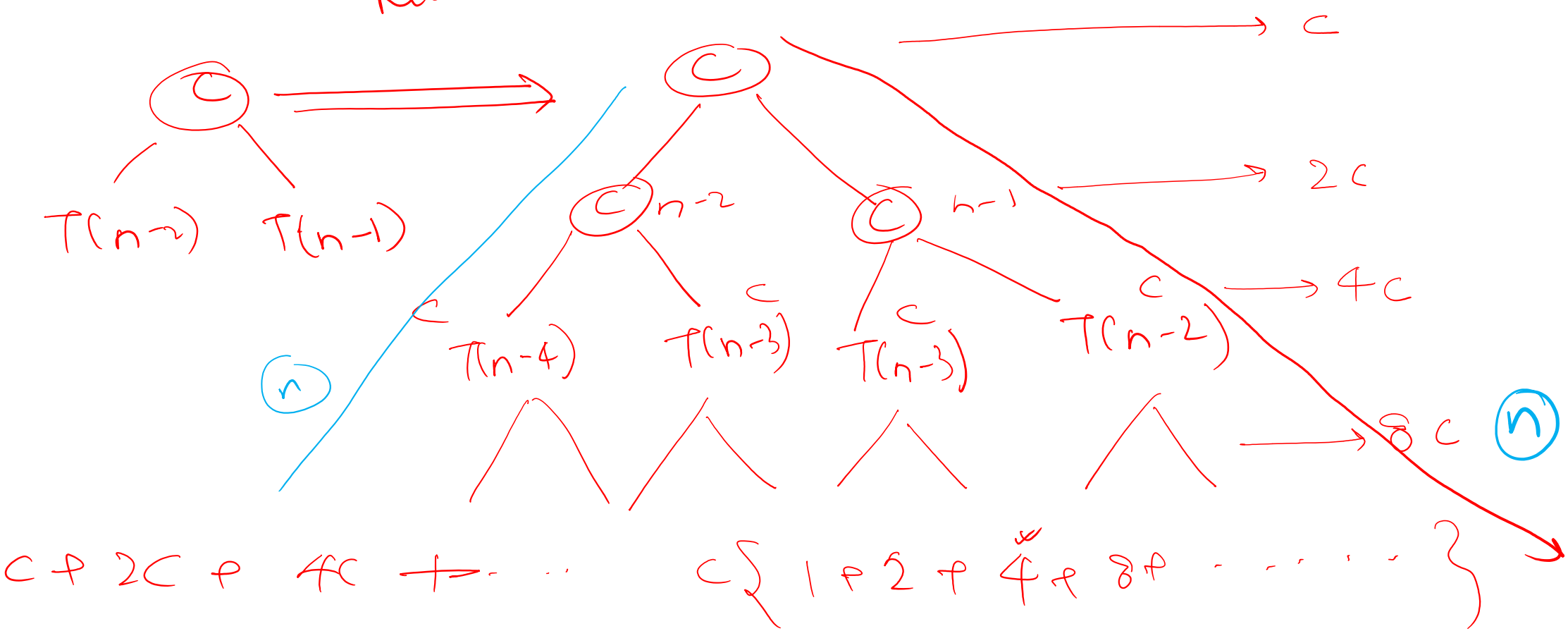
Analysis of Recursion

$T(n) = T(n-1) + T(n-2) + C$

$T(1) = C$

Recursive

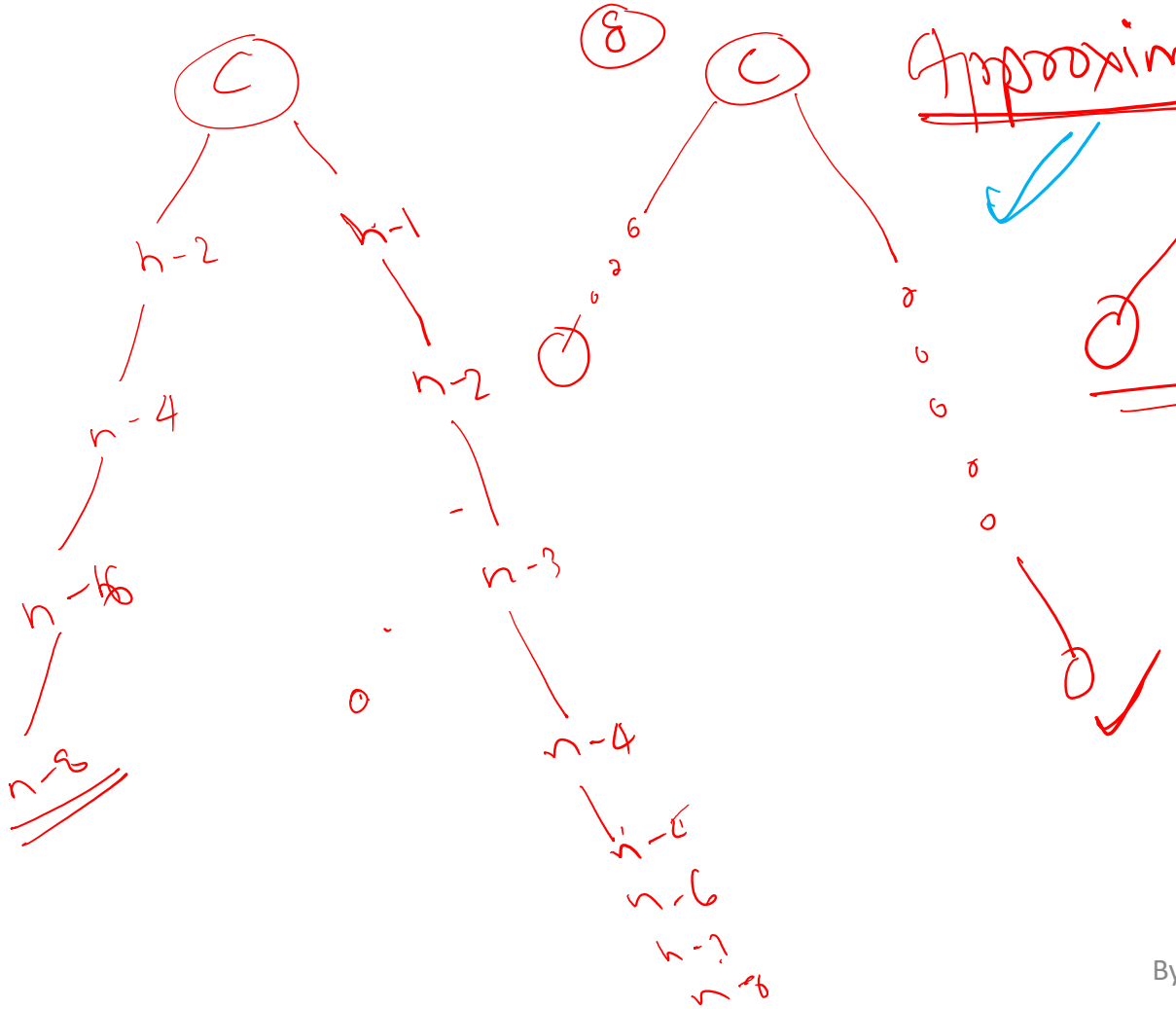
Non Recursive



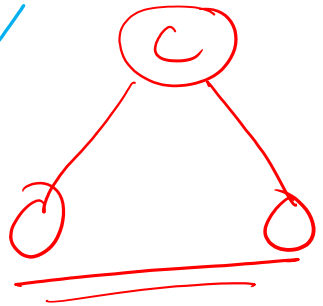
Analysis of Recursion

Time Complexity

$$T(n) = T(n-1) + T(n-2) + C$$
$$T(1) = C$$



Approximation



$C\{1+2+4+8+\cdots\}$ (n terms)

$$S_n = \frac{a(r^n - 1)}{r - 1}$$
$$= \frac{1(2^n - 1)}{2 - 1}$$
$$= 2^n$$
$$\approx O(2^n)$$

$\Theta(2^n)$

Space Complexity

Order of growth of memory (or RAM)

In terms of input

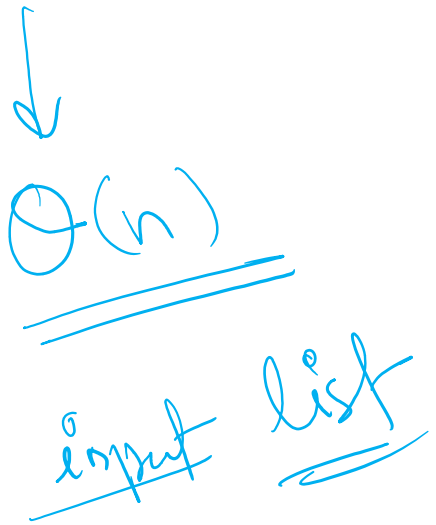
```
def getSum(n):
    return n * (n + 1) / 2


def getSum2(n):
    sum = 0
    i = 1
    while i <= n:
        sum = sum + i
        i = i + 1
    return sum
```

SC
$\Theta(1)$

SC
$\Theta(1)$

Space complexity

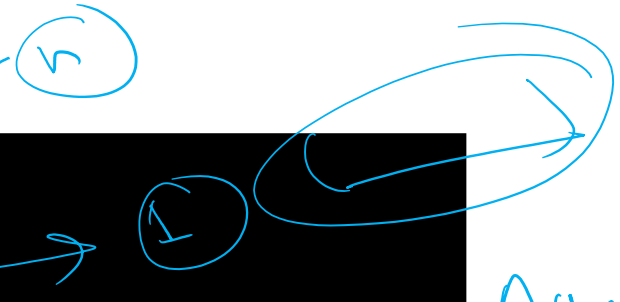
```
def listsum(l):
    sum=0
    for x in l:
        sum = sum + x
    return sum
```



Space complexity

Auxiliary Space : order of growth of extra space (space other than input/output)

```
def listsum(l):
    sum=0
    for x in l:
        sum = sum + x
    return sum
```



Aux. space : O(1)
Space Complexity : O(n)

Space complexity

```
def fun(n):
    if n <= 0:
        return 0
    else:
        return n + fun(n-1)
```

Aux Space : O(n)

Space Complexity : O(n)

function call stack