Analysis of Algorithm (Background)

sum of n natural number

Input : n = 3
output : 6 // 1+2+3

Input : n = 5
output : 15 // 1+2+3+4+5

For less time
① Machine dependent
② programming lang

```
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
```

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

1 + 2 + 3 + 4 .....

$n*(n+1)/2$

$\frac{n^2}{2} + \frac{n}{2}$ ....

Asymptotic Analysis : (Theoritical Analyses)

→ No dependecy on Machine, programming language, etc

→ We do not have to implement all ideas / algorithms

→ Asymptotic Analysis is about measuring order of growth in terms of input size.

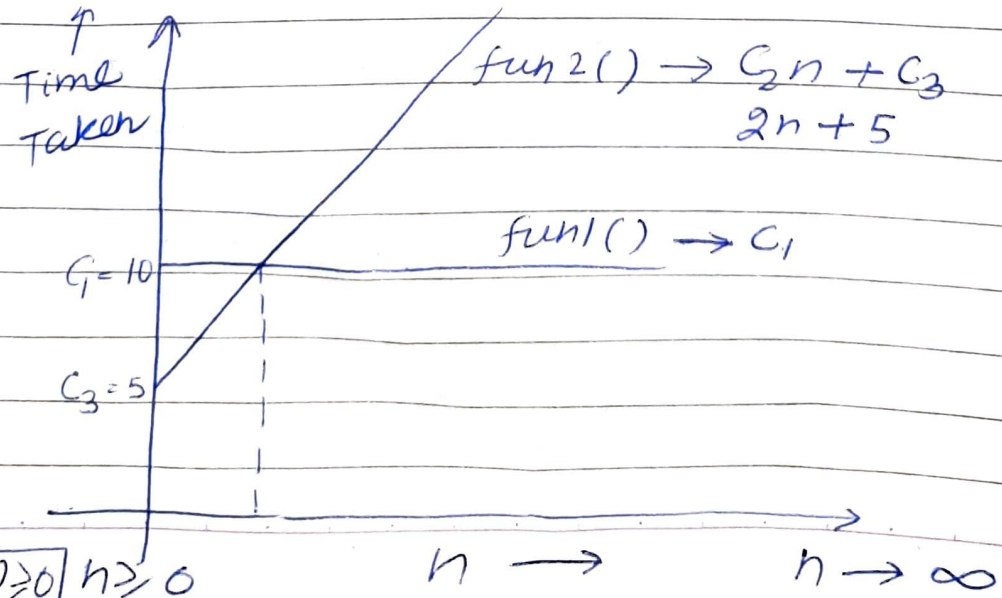
order of Growth :

fun1() ⟶ C_1 ← (any constant)

fun2() ⟶ $C_2 n + C_3$

fun3() ⟶ $C_4 n^2 + C_5 n + C_6$

Comparison of fun1() and fun2()

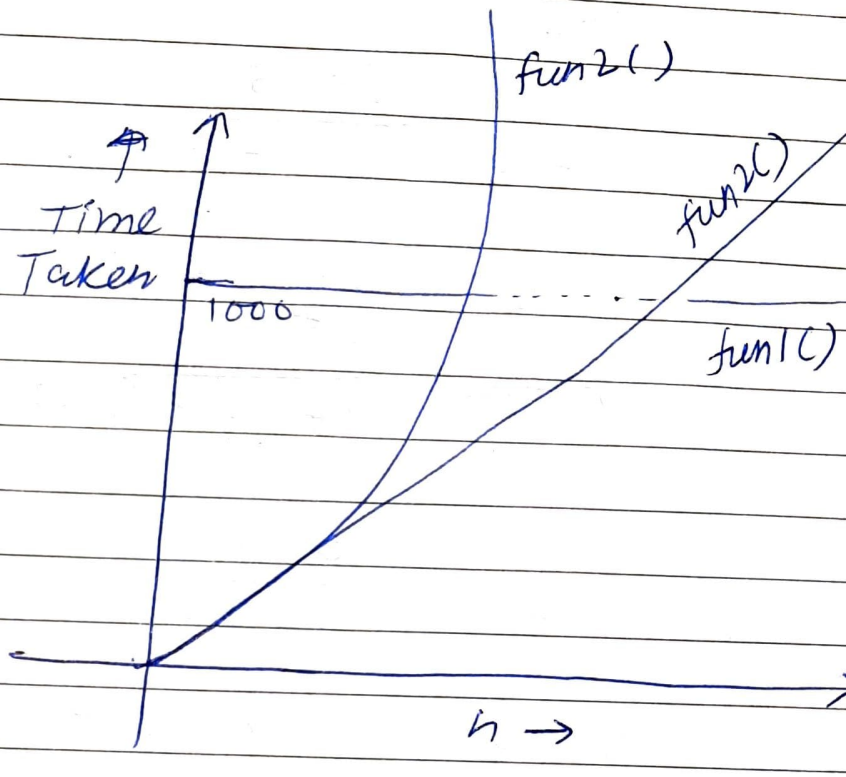


Date / /

$2n + 5 \geq 10$

$n \geq 2.5$

$n \geq 3$



fun1() = 1000

fun2() = n+1

$n+1 \geq 1000$

$n \geq 999$

order of Growth

$n \geq 0$

Time Taken ≥ 0

$f(n), g(n) \geq 0$

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

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

OR

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

$f(n) = n^2 + n + 6$

$g(n) = 2n + 5$

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

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

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

Here $f(n)$ is growing faster than $g(n)$

order of $f(n)$ is faster than $g(n)$
(growth of)

means
$g(n)$ is better algorithms

Direct way to Compare function.

i) Ignore lower order terms
ii) Ignore leading constant.

How do we know which terms are lower order ?

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

$f(n) = 2n^2 + n + 6$
$g(n) = 100n + 3$

$f(n)$'s order of growth is n^2

$g(n)$'s order of growth is n

$g(n)$ is better algo.

Camlin Page
Date / /

$f(n) = c_1 \log n + c_2$ ✓

$g(n) = c_3 n + c_4 \log \log n + c_5$

$f(n) = \log n$ order of growth is lower

$g(n) = n$ higher

better than $g(n)$

$f(n) = c_1 n^2 + c_3 n + c_4 \rightarrow n^2$

$g(n) = c_5 n \log n + c_6 n + c_7 \rightarrow n \log$

↳ $n \log n$ is faster ✓

Best, Average & worst case and Asymptotic Notations

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

function for sum of the element in the list

order of Growth

$C_1 n + C_2$

↳ Order of Growth is n (or linear)

```
def getsumodd(l):
    if len(l)%2==0:
        return 0
    sum=0
    for x in l:
        sum = sum+x
    return sum
```

→ constant

→ O(n) linear

- Best case : Constant O(1)
- Average case : linear O(n)
  (under the assumption that even & odd cases are equally likely)
- Worst case : linear O(n)

Asymptotic Notation

Big O : Represent exact or upper bound

Theta : Represent exact bound

Omega : Represent exact or lower bound

Big O $C_1 n + C_2$

$O(n)$, $O(n^2)$

Theta $C_1 n + C_2$

$\theta(n)$

Omega C $\Omega(1)$

$C_1 n + C_2$

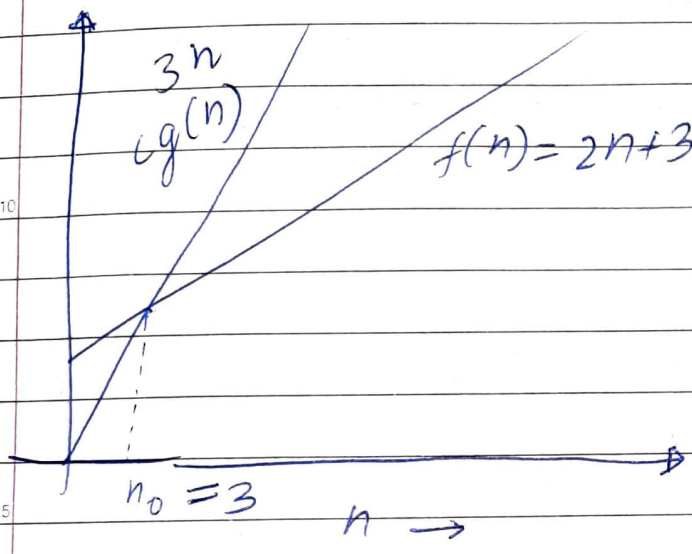
$C_1 + C_2$

$\Omega(n)$

$\Omega(1)$, $\Omega(\log n)$

Big O Notation
(Upper bound on order of growth)

We say $f(n) = O(g(n))$ iff there exist constants c and n_0 such that $f(n) \leq c\,g(n)$ for all $n \geq n_0$



$f(n) = 2n+3$

$O(n)$

$2n+3 \leq 3n$

$3 \leq n$

$n_0 = 3$

$f(n) \leq cg(n)$ for all $n \geq n_0$

$(2n+3) \leq cn$ for all $n \geq n_0$

$c = 3$

$(2n+3) \leq 3n$

$3 \leq n$

$n_0 = 3$

$n = 4$

$2(4)+3 \leq 12$

$11 \leq 12$ ✓

$f(n) = 3n^2 + 2n + 100$

$O(n^2)$

$f(n) = 4n + \log n + 30$

$O(n)$

order of growth

$\in O(n)$ { $\frac{n}{4}$, $2n+3$, $\frac{n}{100} + \log n$, $n+10000$, $n/10000$, 100, $\log n + 1$ }

$\in O(n^2)$ { n^2+n, $2n^2$, n^2+100n, $n^2+2\log n$, $\frac{n^2}{1000}$, ... }

$\in O(1)$ { 1000, 2, 1, 3, 10000, 1000000, ... }

Omega Notation (lower bound)

↳ opposite of $O(n)$

$f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that

$0 \le cg(n) \le f(n)$ for all $n \ge n_0$

$2n^2+3n+6$
$\Omega(n^2)$

Example:

$f(n) = 2n+3$

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

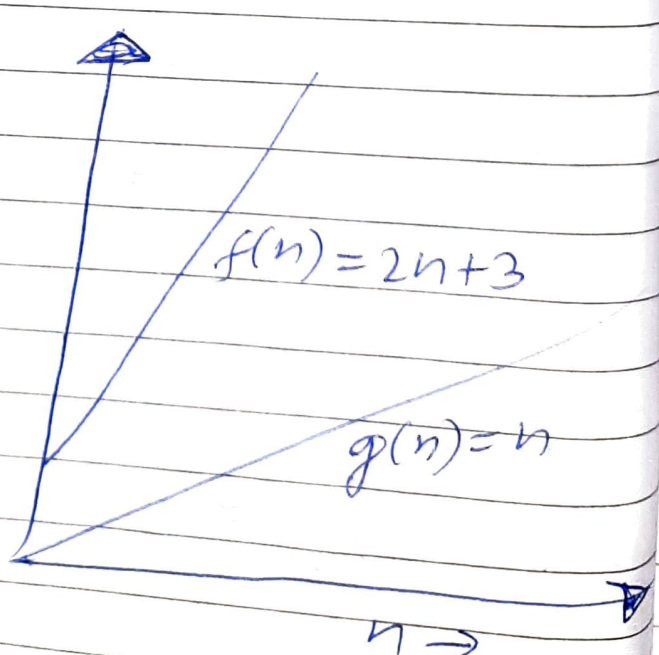
$\boxed{c=1}$

$c\,g(n) = n$

$n \le 2n+3$

$\boxed{n_0 = 0}$

$f(n) = 2n+3 = \Omega(n)$



Camlin Page
Date / /

Omega Notation

① $\{\frac{n}{4}, \frac{n}{2}, 2n, 3n, 2n+3, n^2, 2n^2, \ldots\ldots, n^n\} \in \Omega(n)$ ✓

② If $f(n) = \Omega(g(n))$
then $g(n) = O(f(n))$

③ Omega Notation is useful when we have lower bound on time complexity.

Theta Notation

↳ Exact order of growth

$f(n) = \Theta(g(n))$ iff there exist +ve constants C_1, C_2 and n_0 such that $0 <= C_1 g(n) <= f(n) <= C_2 g(n)$ for all $n >= n_0$

Example $f(n) = 2n+3 = \Theta(n)$

$C_1 g(n) \leq f(n) \leq C_2 g(n) \quad \forall n \geq n_0$

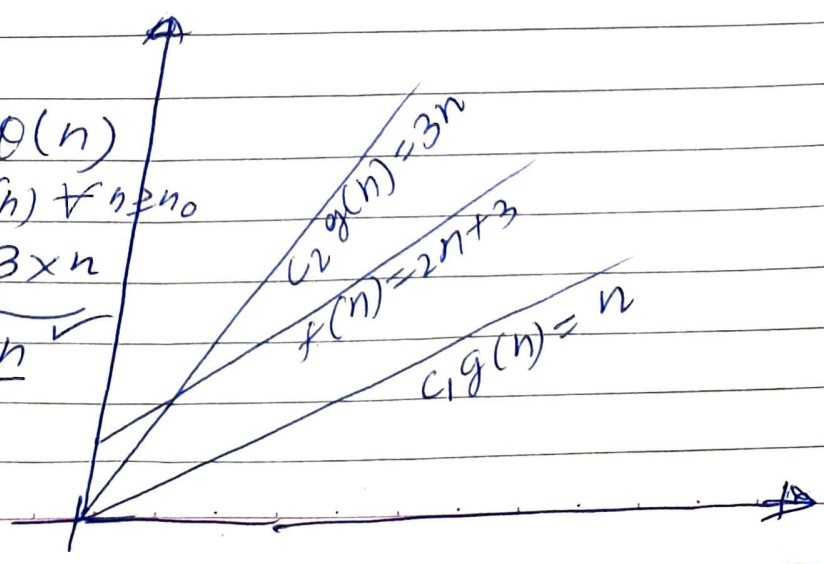
$C_1 = 1$, $C_2 = 3$

$1 \times n \leq 2n+3 \leq 3 \times n$ ✓

$n \geq 0$ — $3 \leq n$ ✓

max = 3

$n_0 = 3$ ✓



Theta Notation

i) If $f(n) = \Theta(g(n))$
then
$f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$
and
$g(n) = O(f(n))$ & $g(n) = \Omega(f(n))$

ii) Theta is useful to represent time complexity when we know exact bound. For example, time complexity to find sum, max and min in an array is $\theta(n)$.

(iii) $\{ \frac{n^2}{4}, \frac{n^2}{2}, \ldots 2n^2, 2n^2 + 1000n,$
$4n^2 + 2n \log n + 30, \ldots \} \in \Theta(n^2)$

Analysis of Common Loops

First example loop

```
i = 0
while i < n:
    i = i + C   (const)
    # some other constant
    # time work.
```

$n = 10$

$C = 2$

0
2
4
6
8

n/C times

$n = 11$

0
2
4
6
8
10

$\lceil \frac{n}{C} \rceil$

So, order of growth of this loop

$\theta(n)$

Equivalent for loop

```
for i in range(0, n, c):
    # some constant work
```

$\theta(n)$

#2nd loop

```
i = n
while i > 0:
    i = i - c
```

$\left\lceil \frac{n}{c} \right\rceil$

$O(n)$

10	n = 10
8	c = 2
6	
4	
2	

11	n = 11
9	c = 2
7	
5	
3	
1	

equivalent loop

```
for i in range(n, 0, -c)
```

$O(n)$

#3rd loop

```
i = 1
while i < n:
    i = i * c
    # ——
    # ——
```

n = 32
c = 2
1, 2, 4, 8, 16

n = 33
c = 2
1, 2, 4, 8, 16, 32

c^0

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

$c^{k-1} < n$

$k < \log_c n + 1$

$O(\log n)$

32, 16, 8, 4, 2. 33, 16, 8, 4, 2

Date / /

4th loop.

```
i = n
while i > 1 :
    i = i/c
    # ------
    # ------.
```

n = 32, c = 2
32, 16, 8, 4, 2

n = 33, c = 2
33, 16, 8, 4, 2

$O(\log_c n)$ ✓

5th loop

```
i = 2
while i < n :
    i = i**c
```

c = 2
n = 32
2, 4, 16,
$2^2, (2^2)^2, (2^2)^2$

$2, 2^c, (2^c)^c \cdots \left(\left((2^c)^c\right)^c\right)^c \cdots$

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

$\boxed{2^{c^{k-1}} < n}$ termination condition ✓

$2^{c^{k-1}} < \log_2 n$

$k-1 < \log_c \log_2 n$

$k < \log_2 \log_2 n + 1$

$O(\log \log n)$ ✓

6th loop (subsequent loops)

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

```
i = 1
while i<n:
    i = i*3         O(log n)
```

```
i = 1
while i<100         O(1)
    i = i+1
```

$$O(n) + O(\log n) + O(1)$$
$$\downarrow \quad x \quad x$$
$$\rightarrow O(n)$$

7th loop : Nested loop

```
i = 0
while i<n:
    j = 1
    while j<n:          } O(log n)   * } O(n)
        j = j*2                        } O(n log n)
    i = i+1
    # ---------
# ----------
```

8th loop : Mixed loops

```
i=0
while i<n:
    j=1
    while j<n:
        j = j*2
    i = i+1
```

$O(n \log n)$

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

$O(n^2)$

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

$$O(n^2)$$

9th loop : Multiple input

```
i=0
while i<n:
    j=1
    while j<n:
        j = j*2
    i = i+1
```

$O(n \log n)$

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

$O(m^2)$

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

Analysis of Recursion

```
def fun(n):
    if n==1:
        return
    for i in range(n):
        print("GFG")
    fun(n/2)
    fun(n/2)
```

$T(n) = 2T(n/2) + \Theta(n) \quad n>1$

Best case $T(1) = \Theta(1) \quad n=1$

```
def fun(n):
    if n==1:
        return
    print("GFG")
    fun(n/2)
    fun(n/2)
```

$T(n) = 2T(n/2) + \Theta(1) \quad n>1$

Best case $T(1) = \Theta(1) \quad n=1$

```
def fun(n):
    if n==1:
        return
    print(n)
    fun(n-1)
```

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

$T(1) = \Theta(1) \quad n=1$

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

↳ Recursive part

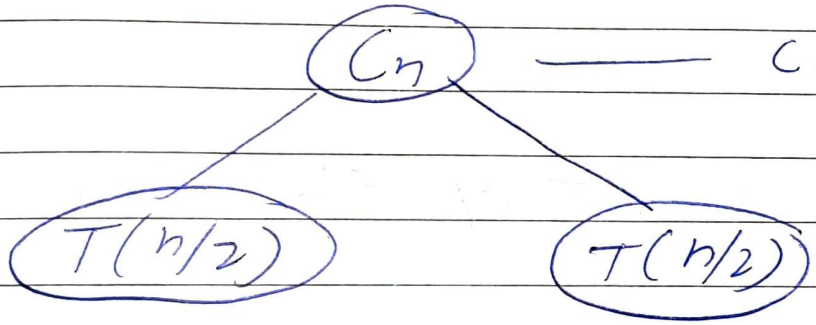
↳ non-recursive part

Recursion Tree Method

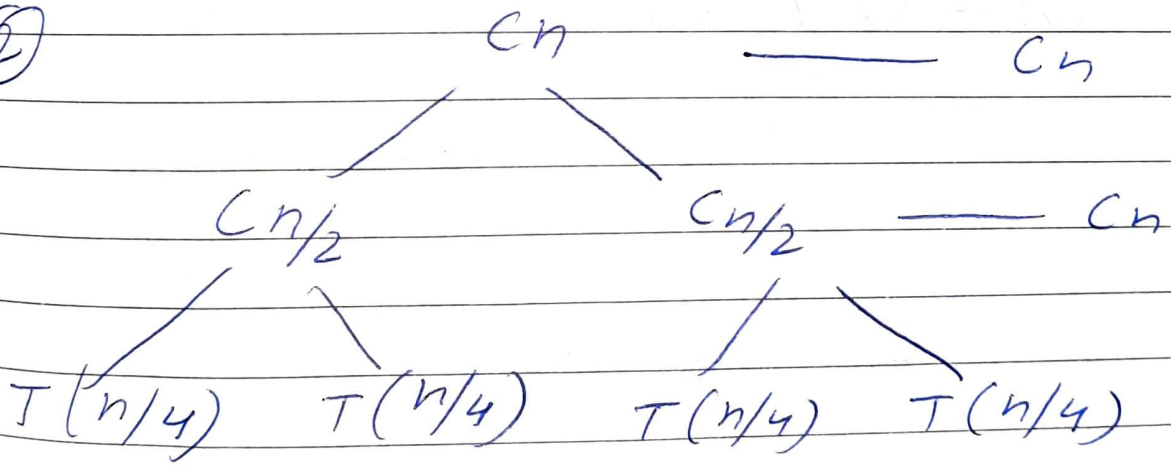
→ we write non-recursive part as root of tree & recursive parts as children

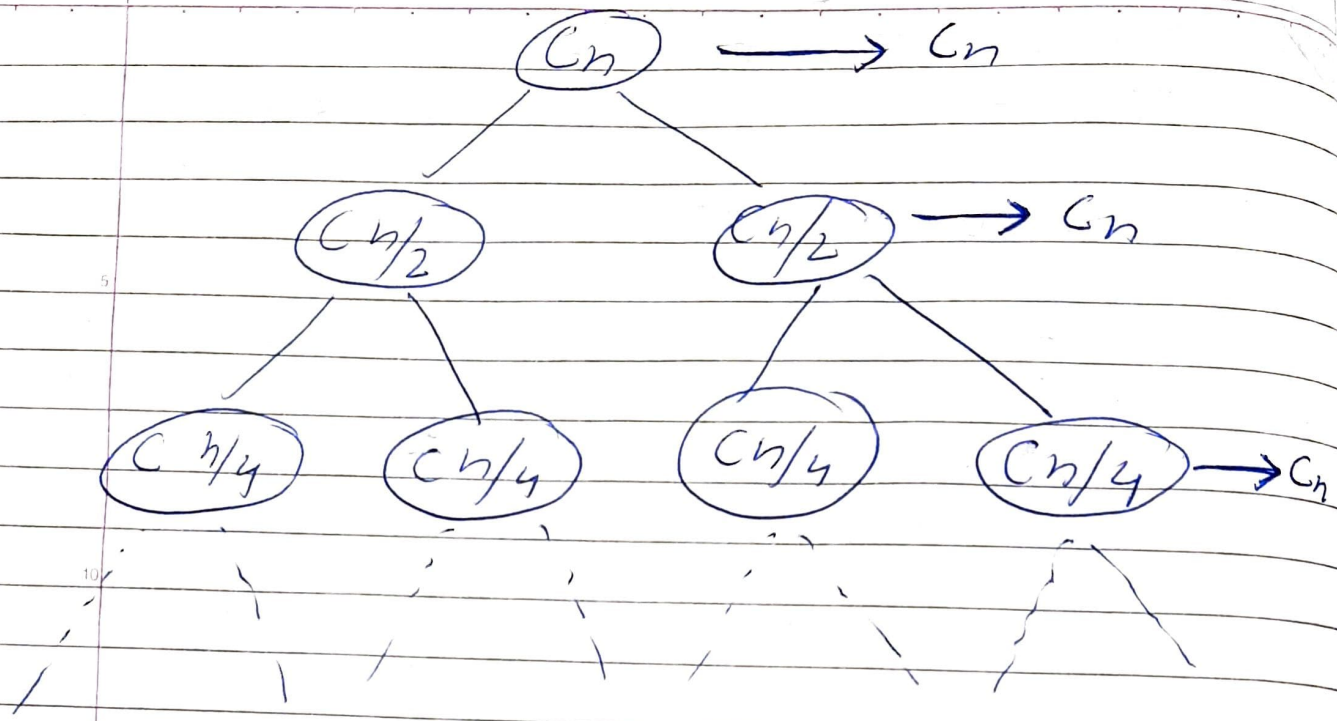
→ we keep expanding children until we see a pattern

(1)



(2)





$\log_2 n$ (height of the tree)

$\underbrace{Cn + Cn + Cn + \ldots\ldots + Cn}_{\log_2 n}$

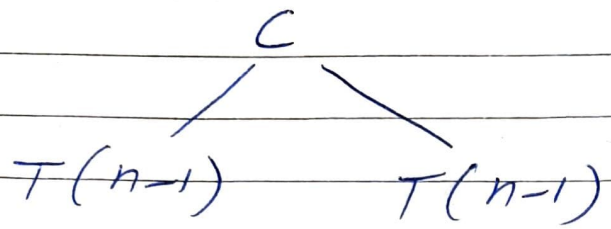
$Cn \times \log_2 n$

Merge Sort $\rightarrow O(n \log_2 n)$

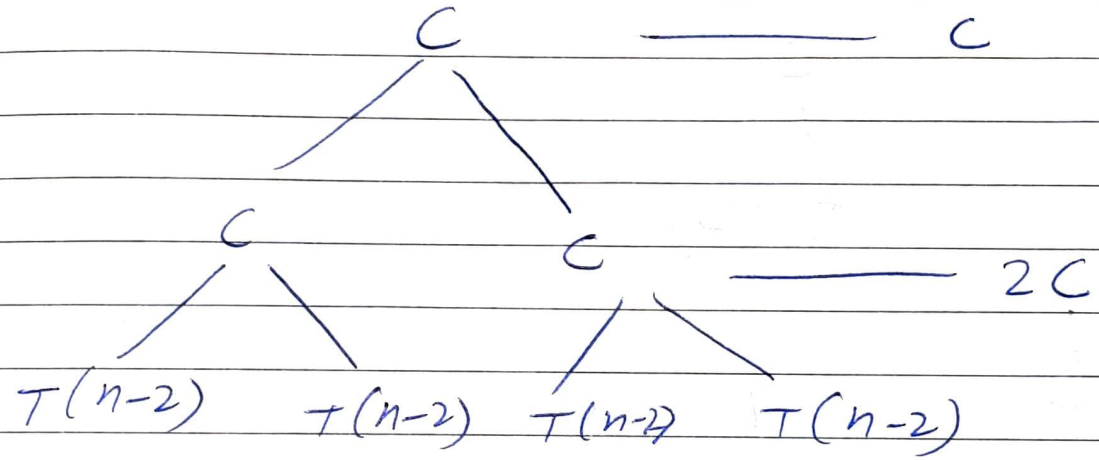
Camlin | Page
Date / /

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

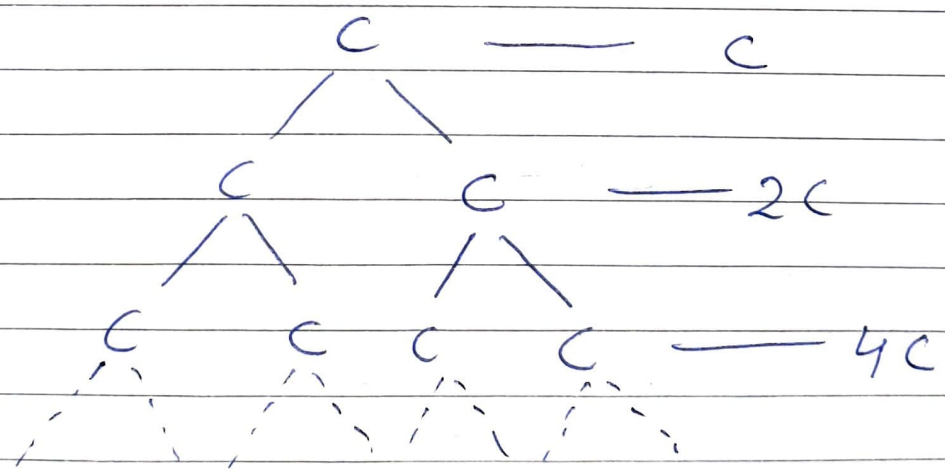
①



②



③



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

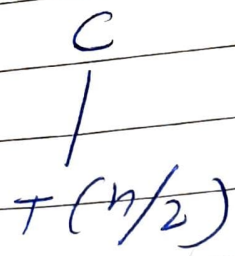
Geometric progression $= \frac{a \times (r^n - 1)}{r - 1}$

$\Theta(2^n)$

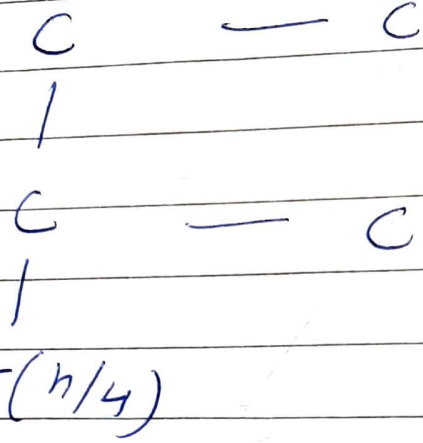
$T(n) = T(n/2) + c$

$T(1) = c$

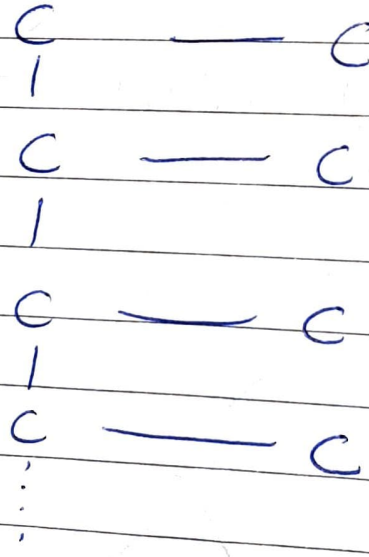
①



②



③



$\underbrace{c + c + c + c + \cdots}_{\log_2 n}$

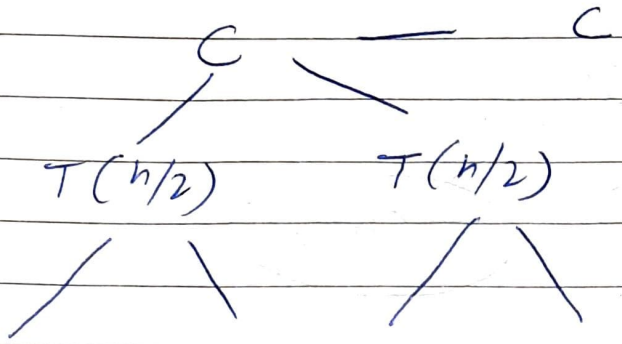
$c \times \log_2 n$

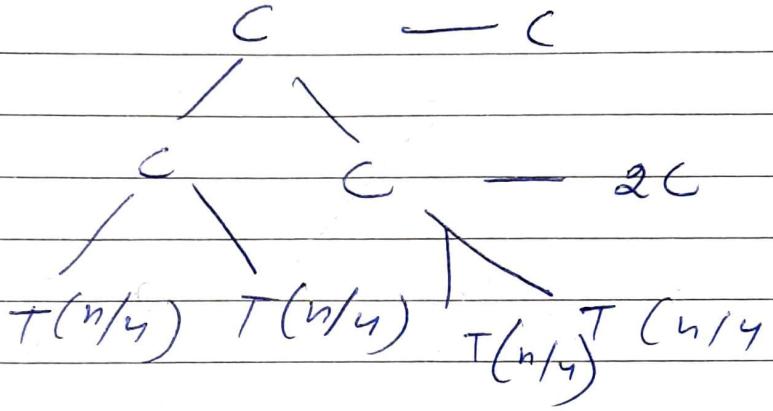
$O(\log n)$

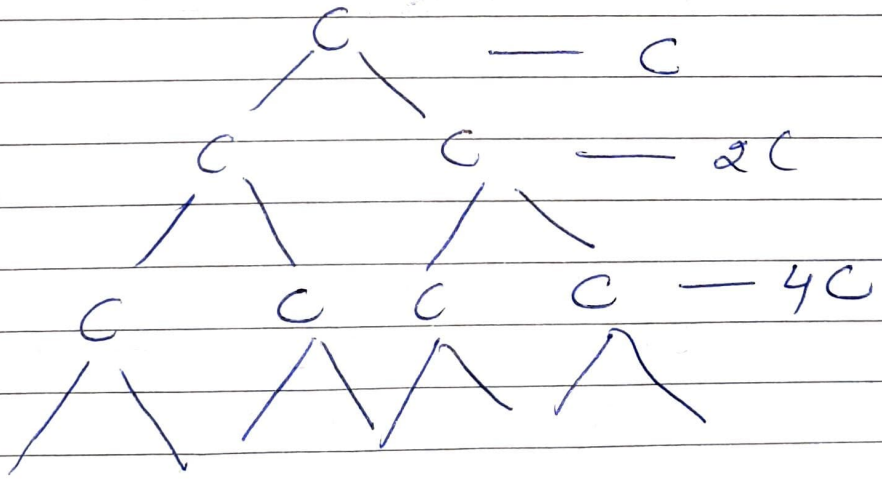
Camlin | Page
Date / /

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

$T(1) = C$

① 

② 

③ 

$C + 2C + 4C \ldots\ldots\ldots$

$\log_2 n$

$\frac{C \times (2^{\log_2 n} - 1)}{2 - 1}$

$O(n)$

Complex Recursion Tree

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

$T(1) = c$

① cn

$T(n/4)$ $T(n/4)$

② cn — cn

$cn/4$ $cn/2$ — $3cn/4$

$T(n/16)$ $T(n/8)$ $T(n/8)$ $T(n/4)$

③ cn — cn

$cn/4$ $cn/2$ — $3cn/4$

$cn/16$ $cn/8$ $cn/8$ $cn/4$ — $9cn/16$

$T(4)$

① ① ①

Assume Full Tree

$cn + 3cn/4 + 9cn/16 \cdots$

geometric progression $r = 3/4$

$\log n$

$O\left(Cn \times \left|\frac{1}{1-3/4}\right|\right)$

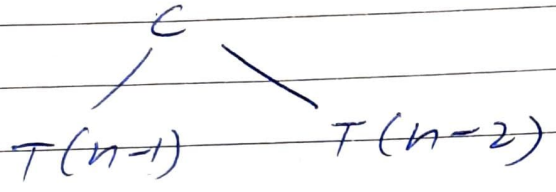
↳ Infinite series

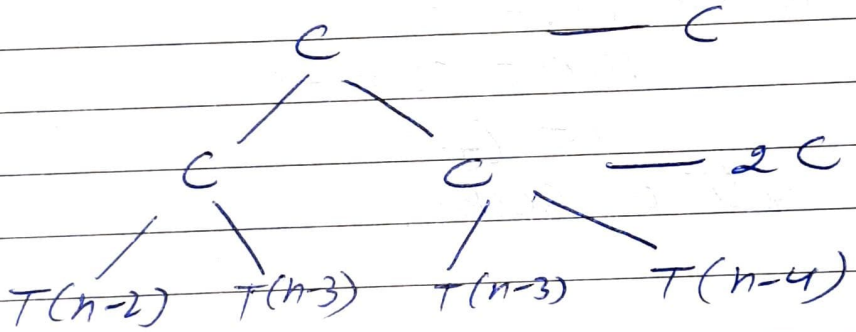
$O(n)$

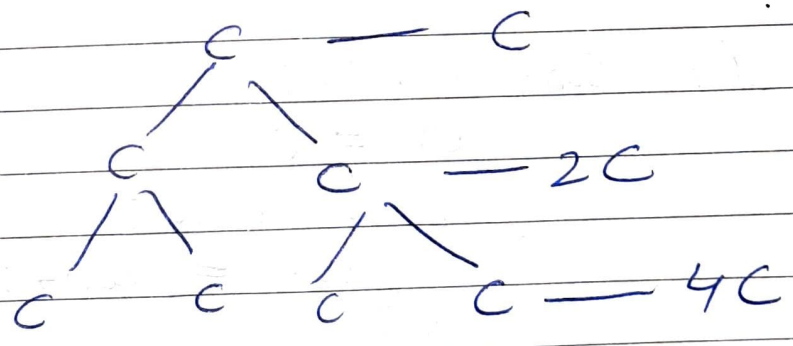
recussive Fibonnacci program ✓

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

$T(1) = C$

① 

② 

③ 

$C + 2C + 4C + \cdots$

$C(1 + 2 + 4 + \cdots)$

n term

$O(2^n)$

Space Complexity

order of growth of Memory (or RAM) usage in terms of input.

```
def getSum1(n):
    return n*(n+1)/2
```

O(1) or O(1)

```
def getsum2(n)
    sum=0
    i=1
    while i<=n:
        sum = sum+i
        i= i+1
    return sum
```

O(1) or O(1)

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

O(n)

Auxiliary space

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

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

Auxiliary: O(1)
Space complexity: O(n)

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

fun(3)
fun(2)
fun(1)
fun(0)

Space Complexity = O(n)

Auxiliary Space = O(n)