19-8-2024.

DSA in Python

Why DSA → makes us a better software Developer
→ Helps us in getting a job.
→ Winning a sport of Competitive coding

Analysis of Algorithm:-

1 Sum of first n Natural Numbers.

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

(or)

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

(or)

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

Asymptotic Analysis (Theoritical analysis):

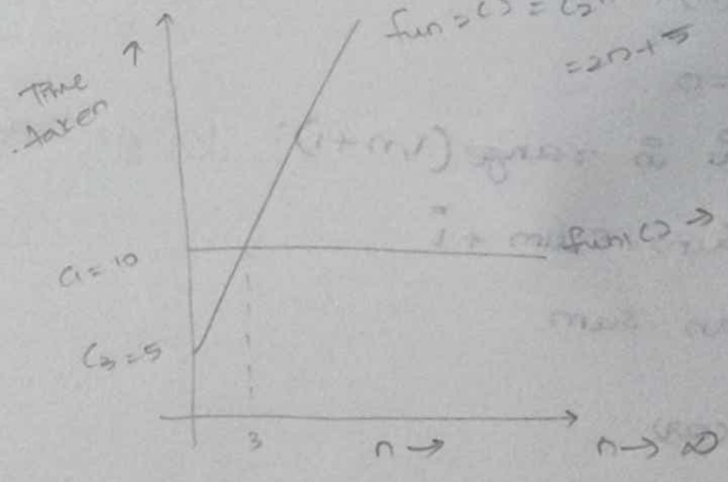
* No dependency on machine, programming language, load etc
* We donot have to implement all algorithm
* is about measuring order of growth in terms of input size.

Order of Growth:

fun1() → C_1 constant

fun2() → $C_2n + C_3$ linear

fun3() → $C_4n^2 + C_5n + C_6$. Quadratic



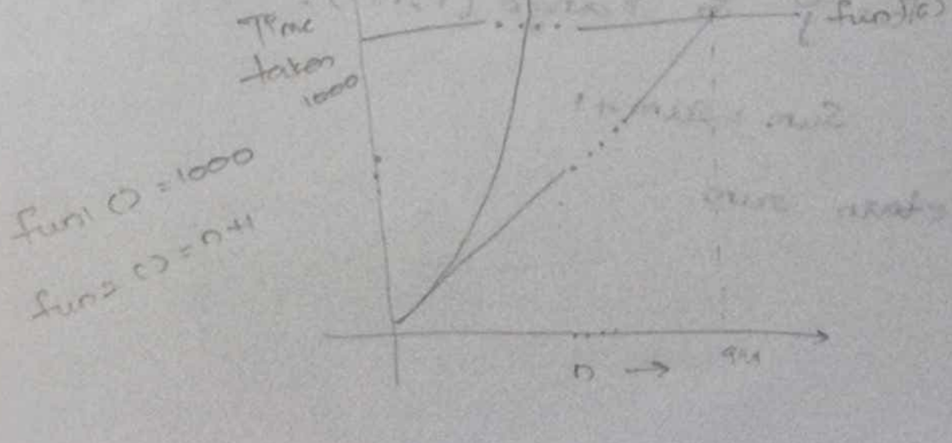
$2n+5 \geq 10$

$n \geq 2.5$

$n \geq 3$

$n \geq 0$

fun1(), fun2() ≥ 0



$n+1 \geq 1000$

$n \geq 999$

Order of Growth:

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

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

$f(n)$ and $g(n)$ represent Time taken.

OR

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

$n \geq 0$

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

Eg:

$f(n) = n^2 + n + 6$ ⇒ Order of growth : n^2

$g(n) = 2n + 5$ ⇒ " " " : n

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

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

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

Direct way to find and compare growth

① Ignore lower order terms

② Ignore leading term constant.

$f(n) = 2n^2 + n + 6$ → n^2

$g(n) = 100n + 6$ → n.

How do we compare terms?

Upper bound

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

$< n^4 < 2^n < n^n$

lower bound

Avg : no bound

Use what is closest

Ex. ① $f(n) = c_1 \log n + c_2 \Rightarrow \log n$

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

$g(n)$ is higher and Bad.

② $f(n) = c_1 n^2 + c_2 n + c_3 \Rightarrow n^2$

$g(n) = c_4 n \log n + c_5 n + c_6 \Rightarrow n \log n$

$f(n)$ is higher

Big O Notation :- Upper bound on Order of Growth.

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

Eg. $f(n) = 2n+3$ can be written as $O(n)$

$f(n) \leq c\,g(n)$ for all $n \geq n_0$

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

highest growing term const +1

$c = 2+1 = 3$

C = highest growing term const + 1

$c = 3$

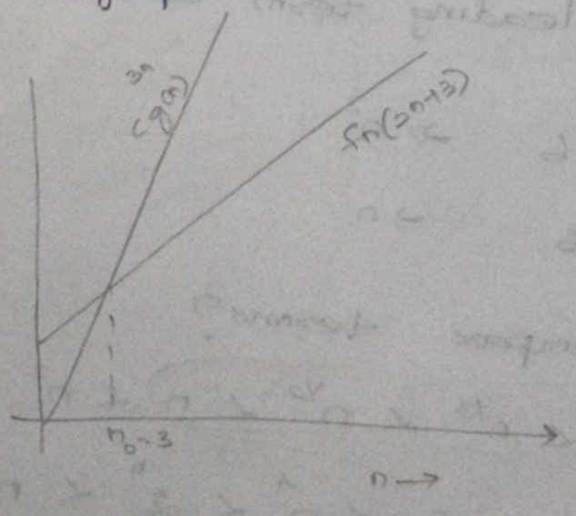
$2n+3 \leq 3n$ $\qquad 2n+3 \leq 10n$ (or) $2n+3 \leq 2n+3n$

simple Method

$2n+3 \leq 5n$

$3 \leq n$ $\qquad f(n) = O(n)$

$n_0 = 3$



$\left\{\frac{n}{4}, 2n+3, \frac{n}{100} + \log n, n+10000, \frac{n}{10000}, 100\log n \dots\right\} \in O(n)$

$\left\{n^2+n, 2n^2, n^2+1000n, n^2+2\log n, \frac{n^2}{1000}, \dots\right\} \in O(n^2)$

$\{1000, 2, 3, 1, 10, 100000 \dots\} \in O(1)$

Omega Notation : lower bound

Opposite of Big O Notation

$f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $0 \le c\,g(n) \le f(n)$ for all $n \ge n_0$

$f(n) \ge c * g(n)$

Eg. $f(n) = 2n+3$

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

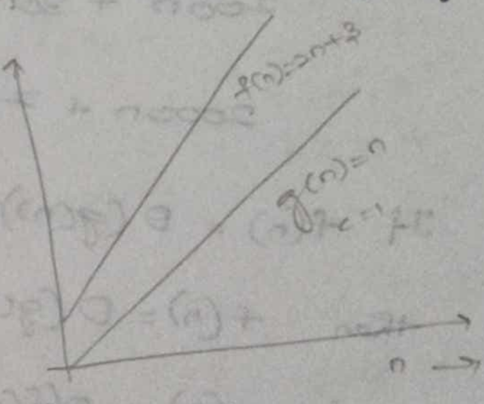
$\Omega(n)$

Smaller than constant of highest term

$c = 1$

$c\,g(n) = n$

$n \le 2n+3 \quad n_0 = 0$



① $\left\{\frac{n}{4}, \frac{n}{2}, 2n, 3n, 2n+3, n^2, 2n^2, \dots n^n\right\} \in \Omega(n)$

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.

O, Ω, Θ are different from best, Avg, worst case. Should not compare.

7. Nested loop.

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

$\Theta(n)$

$\Theta(\log n)$

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

$= \Theta(n \log n)$

8. Mixed loops:

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

$\Theta(n \log n)$

$\Theta(n^2)$

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

$\Rightarrow \Theta(n^2)$

9. Multiple input.

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

$\Theta(n \log n)$

$\Theta(m^2)$

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

$= \Theta(n \log n + m^2)$

Analysis of Recursion :-

Recurrence Relation

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

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

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

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

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

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

```
③ def fun(n):
      if n==1:
         return
      print(n)
      fun(n-1)
```

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

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

$\theta = c$

$\therefore \theta(n) = cn$

recursive

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

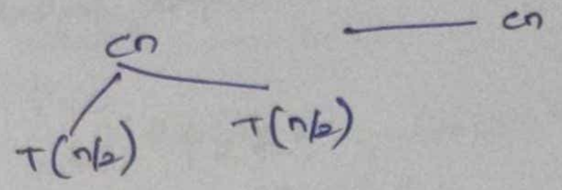
Non recursive

$T(1) = c$

Recursion Tree method:-

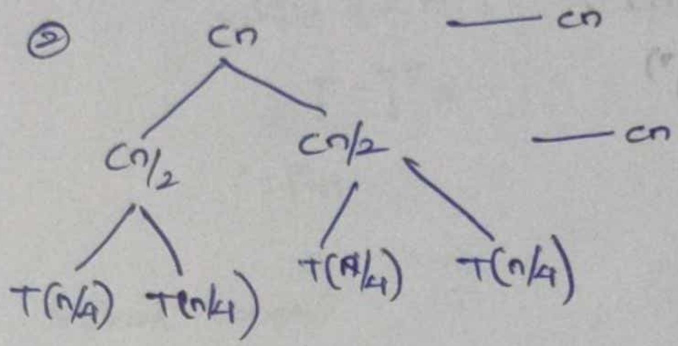
* We write non-recursive part as root of tree and recursive part as children.
* We keep expanding children until we see a pattern.

①



⇓

②



Cn+Cn+Cn+...Cn

↓

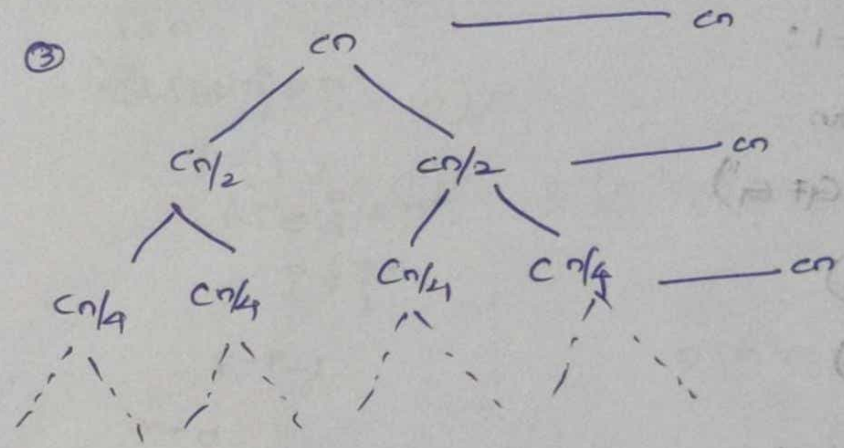
$\log_2 n$

$\log_2 n \times cn$

$\Theta(n \log_2 n)$

$= \Theta(n \log n)$

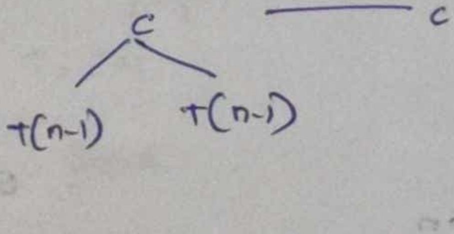
③



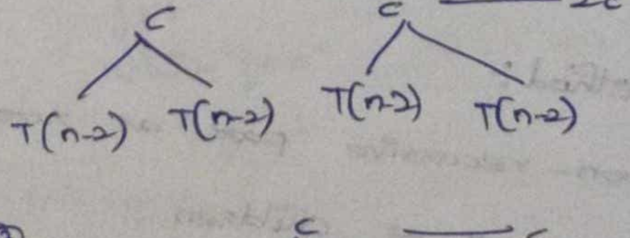
$T(n) = 2T(n-1) + c$

$T(1) = c$

①



②

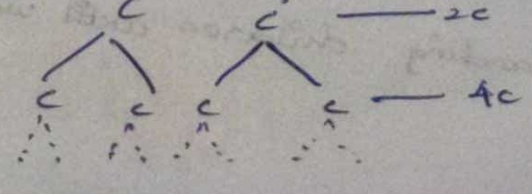


$\underbrace{c+2c+4c\ldots}_{n}$

$\cancel{\frac{n(n+1)}{2}}$ $\quad \frac{a \times (r^n - 1)}{r-1}$

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

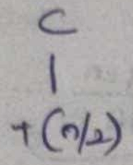
③



$\Theta 2^n$

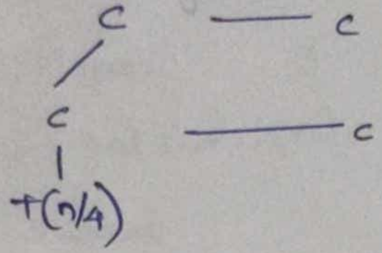
$T(n) = T(n/2) + c$

$T(1) = c$

① 

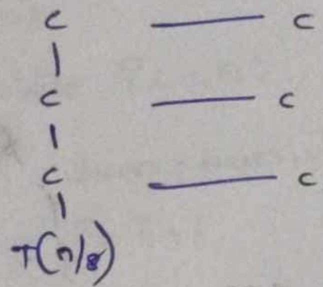
$c + c + c + \dots$

$\log_2 n$

② 

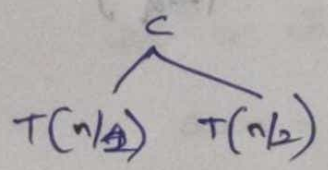
$c \log_2 n$

$\rightarrow \Theta(\log n)$

③ 

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

$T(1) = c$

① 

$c + 2c + 4c \dots$

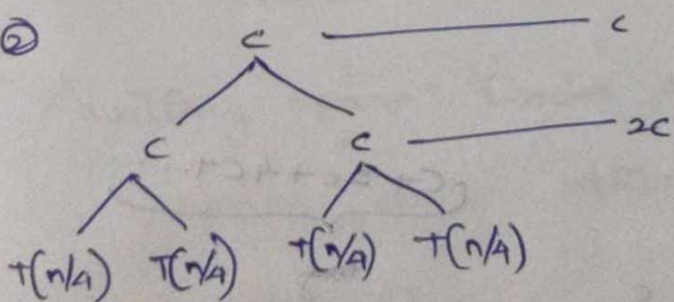
$c(1 + 2 + 4 + \dots)$

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

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

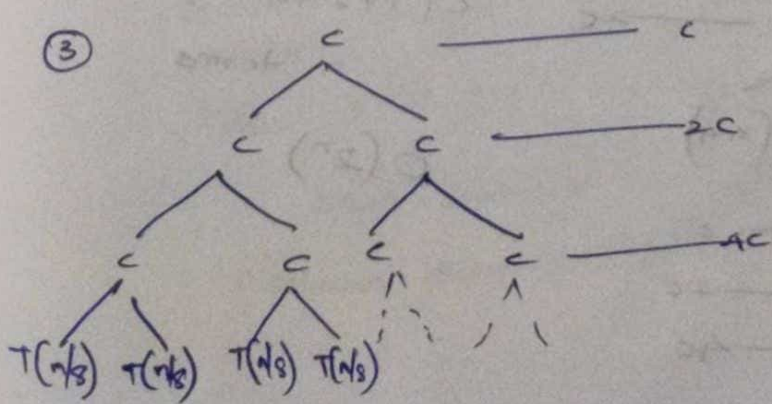
$r = 4/2 = 2$

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

② 

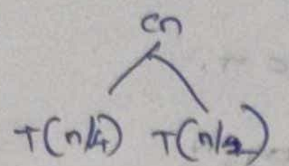
$\Theta(n)$

$\cancel{\Theta(2^n)}$

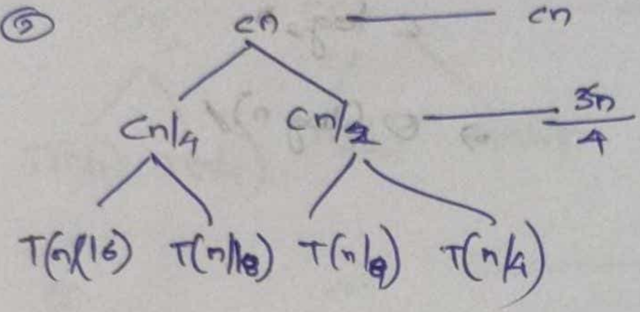
③ 

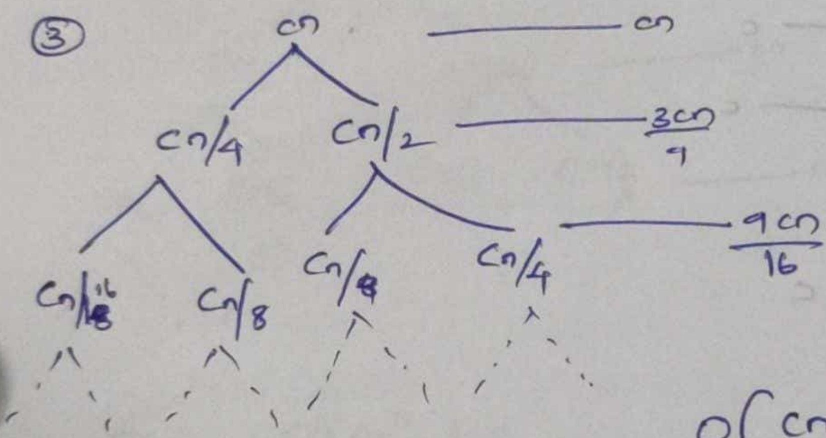
$T(n) = T(n/4) + T(n/2) + cn$

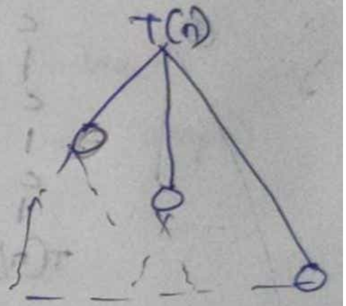
$T(1) = c$

① 

$\underbrace{cn + \frac{3cn}{4} + \frac{9cn}{16} \ldots\ldots}_{\log n}$

② 

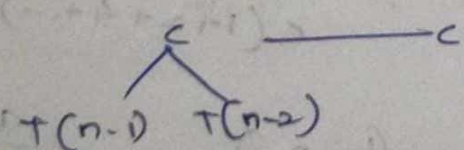
③ 



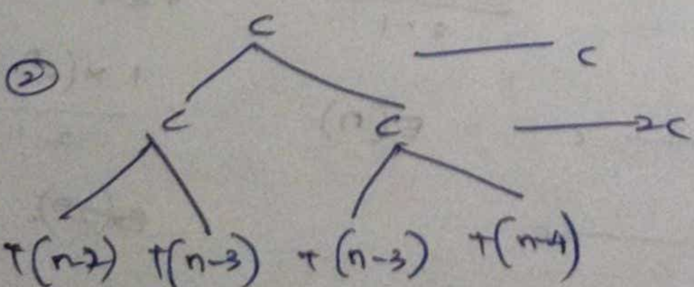
$O\left(cn \times \frac{1}{1-3/4}\right) \Rightarrow O(n)$

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

$T(1) = c$

① 

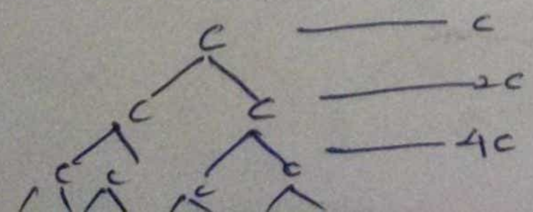
$\underbrace{c + 2c + 4c + \ldots\ldots}_{\log}$

② 

$c(1+2+4\ldots)$

n terms

$O(2^n)$

③ 

Space Complexity :-

Order of growth of memory (or RAM) usage in terms of input.

* def getsum1(n): $\Theta(1)$ or $O(1)$

```
def getsum1(n):
    return n*(n+1)/2
```

(OR)

* def getsum2(n): $\Theta(1)$ or $O(1)$

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

* def listsum(l): $\Theta(n)$

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

Auxiliary space: Order of growth of extra space (space other than I/P or O/P).

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

Auxiliary : $\Theta(1)$

Space complexity : $\Theta(n)$

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

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

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

$\theta(n)$ — Aux space
— Space complexity.

If n is input
n+1 function used
in stack