Camlin Page
Date / /

Introduction

Analysis of Algorithm:

Sum of first N Natural Number

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

$\uparrow$

$O(1)$

2) sum = 0

for (i in range of (n):

sum += i

$O(n)$

3) sum = 0

for i in range(n)

for j in range

sum += j

$O(n^2)$

Order of Growth

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

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

Direct Way:

1) Ignore lower ower term
2) Ignore leading constant

How do we Know which terms are lower order?

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

Best, Average and Worst case & Aysmptotic Notations

```
int getsum (int arr[], int n)
    if (n%2 != 0)
          return 0;                  → Best case: Constant
    else                             → Average case: Linear
    {   sum = 0;                       (under assumption that even
        for (int i=0; i<n; i++)         and odd are likely distributed)
            sum = sum + arr[i];      → Worst case: Linear
    return sum;
    }
```

Big O : Represent exact bond or upper bond
Theta : Represent exact bound
omega : Represent exact or lower bound

Big O Notation:

Mathematical Expression:

We say $f(n) = O(g(n))$ if there exist constant C and N_0 such that $f(n) \leq c\,g(n)$ for all $n \geq n_0$

Example

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

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

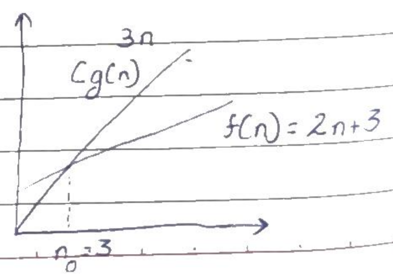
$2n+3 \leq Cn$ for all $n \geq n_0$

C must be greater than 2 here to hold this condition true

$2n+3 \leq 3n$ for $n \geq 3$ this condition will hold true

$3 \leq n$

$n \geq 3$ $\quad \therefore n_0 = 3$



$\left\{ \frac{n}{4},\ 2n+3,\ \frac{n}{100} + \log n,\ n+1000,\ \frac{n}{1000},\ 100,\ \log n + 100 \right\} \in O(n)$

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

$\{100, 2, 1000, \ldots\} \in O(1)$

Applications:

```
int linearsearch (int arr[], int n, int x)
{   for (int i=0, i<n, i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

Here if element is found at either 1 or 2 position, its best case can be $O(1)$ but to consider all cases we do $O(n)$ as worst case Time complexity.

Omega Notation: (Lower bound or Equal)

Mathematical Defination

$f(n) = \Omega(g(n))$ iff there exist positive constant c and N_0 such that $0 \le c\,g(n) \le f(n)$ for all $n \ge n_0$

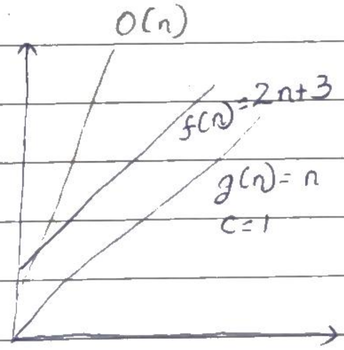
Example $\quad f(n) = 2n+3 \quad = \Omega(n)$

Here c will be less than this

$c = 1$

$c\,g(n) = n$

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



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

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. eg: Game

Thetha Notation (Exact order of growth)

Mathematical Defination:

$f(n) = \Theta(g(n))$ iff there exist positive constraint 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: $f(n) = 2n+3$ order of growth $= \Theta(n)$

$c_1 = 1$ as $c_1 < f(n)$

$c_2 = 3$ as $c_2 > f(n)$

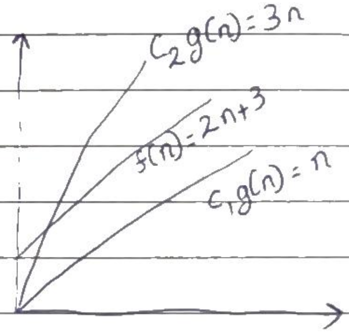
$c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

$1 \times n \le 2n+3 \le 3 \times n$

$n \ge 0$ $\qquad$ $3 \le n$

$n \ge 3$

Max of this two is 3 $\qquad \therefore n_0 = 3$



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

2) Theta is useful to represent time complexity when we know exact bound

Eg. TC to find sum, max and min in an array is $\Theta(n)$

3) $\left\{ \frac{n^2}{4}, \frac{n^2}{2}, \ldots, 2n^2, 2n^2+1000n, \ldots \right\} \in \Theta(n^2)$

→ **Analysis of Common Loop**

Loop	TC	Example
1) `for(int i=0; i<n; i+c)` `{ //some Θ(1) work}`	$\Theta(n/c)$ $\Theta(n)$	$n=10$ $c=2$ $i = 0\ 2\ 4\ 6\ 8$ $\lfloor \frac{n}{c} \rfloor$
2) `for(int i=n; i>0; i=i-c)` `{ //some Θ(1) work}`	$\Theta(n)$	$n=10$ $c=2$ $i = 10, 8, 6, 4, 2$ $\lceil \frac{n}{c} \rceil$
3) `for(int i=1; i<n; i=i*c)` `{ //some Θ(1) work}`	$\Theta(\log n)$	$n=32, c=2 \rightarrow i = 1, 2, 4, 8, 16$ $\lceil \frac{n}{c} \rceil$ $n=33, c=2 \rightarrow i = 1, 2, 4, 8, 16, 32$

In general it will work $1, c, c^2, c^3, \ldots c^{k-1}$

where it must be $c^{k-1} < n$

$$k-1 < \log_c n$$
$$k < \log_c n + 1$$

Note: Here base of log do not matter

Loop	TC	Example
4) `for(int i=n; i>1; i=i/c)` `{ //some Θ(1) work}`	$\Theta(\log n)$	$n=32, c=2 \rightarrow 32, 16, 8, 4, 2$ $n=33, c=2 \rightarrow 33, 16, 8, 4, 2$
5) `for(int i=2; i<n; i=pow(i,c))` `{ //some Θ(1) work}`	$\Theta(\log \log n)$	$n=32, c=2 \rightarrow 2, 4, 16$

In general it will work $2, 2^c, (2^c)^c, \ldots (2^c)^{k-1}$

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

6) void fun(int n)
{ for (int i=0; i<n; i++)] Θ(n)
   // some Θ(1) work
  for (int i=1; i<n; i*2)] Θ(logn)
   // some Θ(1) work
  for (int i=1; i<100; i++)] Θ(1)
   // some Θ(1) work
}

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

Overall TC

$\Theta(n)$

7) void fun(int n)
{ for (int i=0; i<n; i++) → Θ(n)
    for (int j=0; i<n; j=j*2) → Θ(logn)
     // some constant work Θ(1)
}

TC

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

$= \Theta(n \log n)$

8) void fun(int n)
{ for (int i=0; i<n; i++)
    for (int j=0; j<n; j=j*2)
     // some constant work Θ(1)
  ] Θ(nlogn)
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
     // some constant work Θ(1)
  ] Θ(n²)
}

Overall TC

$\Theta(n^2)$

Analysis of Recursion

```
Void fun (int n)
  if (n <= 1)
     return;
  for (int i=0 ; i<n; i++)  ]θ(n)
     print ("Hi")
  fun(n/2)  <- T(n/2)
  fun(n/2)
```

Let the TC be $T(n)$

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

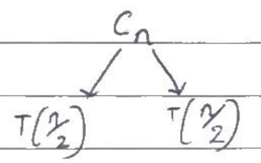
For $T(1) = 1$ (Base case)

Recursion Tree Method

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

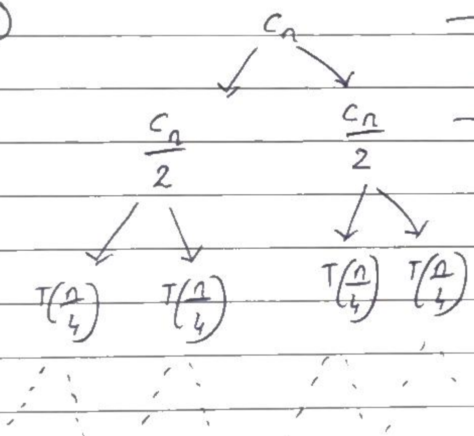
eg $\quad T(n) = \underbrace{2T(n/2)}_{\text{Recursive}} + \underbrace{Cn}_{\text{Non-Recursive}} \qquad T(1) = C$

①



→ Work Done is Cn

②



→ Cn

→ Cn

→ Cn

- Here Height of Tree is $\log_2 n$

This tree will stop to expand when base case is reached ie 1

Here total work done in above tree is

$\underbrace{Cn + Cn + Cn + \ldots}_{\log n \text{ times}}$

$cn \times \log_2 n$

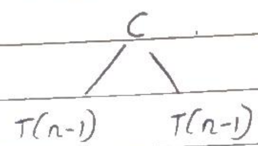
$\therefore \Theta(n \log n)$

Example 2:

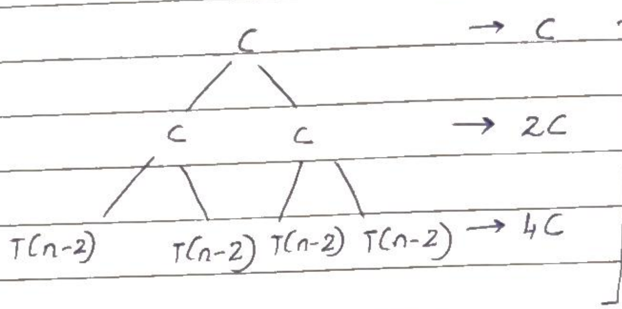
$T(n) = 2T(n-1) + \underline{C}$ → Here as it is constant and not dependent on n so in 2^{nd} step it will be same bcoz $C/2 \simeq C$ in asymptotic Analysis

$T(1) = C$

(1)



(2)



Here Height is n

$\underline{C + 2C + 4C \ldots}$ ↳ n times

$n(1 + 2 + 4 + \ldots) = \Theta(2^n)$

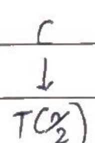
Example 3:

→ Here the recursion is only 1 time

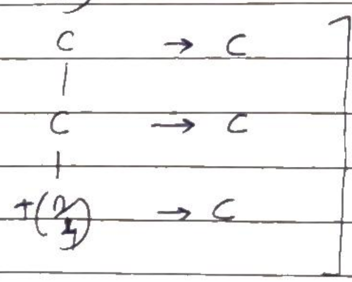
$T(n) = T(n/2) + C$

$T(1) = C$

(1)



(2)

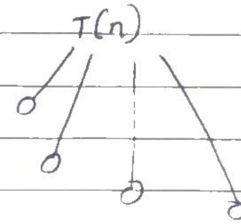


Height is $\log_2 n$

$(C+C+C\ldots)$ n times $\quad \because TC : \Theta(\log n)$

Example 4:

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



①

$$Cn$$
$$T(n/4) \quad T(n/2)$$

②

Height is log n

$$Cn \qquad\qquad Cn$$
$$\frac{Cn}{4} \quad \frac{Cn}{2} \qquad\qquad \frac{3Cn}{4}$$
$$T\left(\frac{n}{16}\right) \quad T\left(\frac{n}{8}\right) \quad T\left(\frac{n}{8}\right) \quad T\left(\frac{n}{4}\right) \qquad\qquad \frac{9Cn}{16}$$

Here as the tree is not symettric so for solving we assume this tree as a Linear and it is Full So as we have assumed here so we are using Big O notation

$$Cn + \frac{3Cn}{4} + \frac{9Cn}{16} + \ldots \quad \log n \text{ times}$$

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

Space Complexity

Order of growth of Memory (Or RAM) Space in terms of input size.

```
int getsum (int n)
{ return n*(n+1)/2;}
```

↑

Constant SC. $O(1)$ or $\Theta(1)$

```
int getsum1 (int n)
{ int sum=0;
  for (int i=0; i<=n; i++)
      sum=sum+i;
  return sum; }
```

SC: $O(1)$ or $\Theta(1)$

```
int arrsum (int arr[], int n)
{   int sum=0;
    for(int i=0; i<n; i++)
        sum=sum+ arr[i];
    return sum;
}
```

← Here as in this program we do not require extra space so auxilary space is

↓

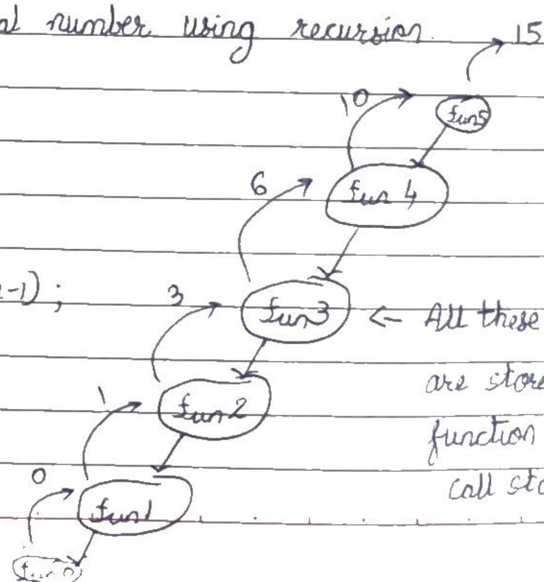
SC: $\Theta(n)$ Aux space: $\Theta(1)$

→ Auxilary space:

Order of growth of extra space or temperory space in terms of input size

→ Sum of first N natural number using recursion

```
int fun (int n)
{   if (n<=0)
        return 0;
    return n+ fun(n-1);
```



← All these function are store in function call stack

Here total n+1 call are stored in function call stack

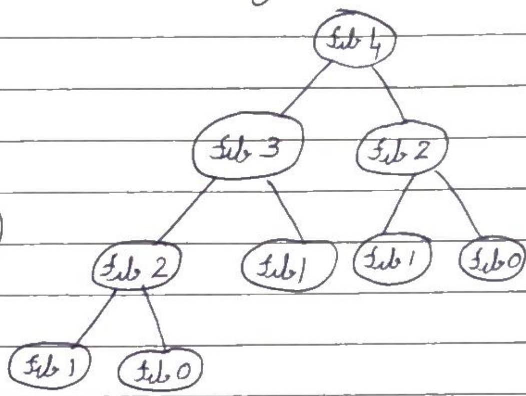
fun(0)
fun(1)
fun(2)
fun(3)
fun(4)
fun(5)

Here 6 call are stored in Stack

So space complexity is $\Theta(n)$

→ space complexity of Fibonaci Numbers using recursion

```
int fib(int n)
{    if (n==0 || n==1)
         return n;
     return fib(n-1)+fib(n-2)
}
```



Here SC is obtained by height of three. So Here SC is $\Theta(n)$