

31 July '20

## Analysis of Algorithm

\* Analysis of a function returning the sum of first 'n' natural numbers.

Algo-1    int fun1(int n){  
              return n\*(n+1)/2;  
      }

Algo 2    int fun2(int n){  
              for (int i=0; i<=n; i++){  
                  sum = sum + i;  
              }  
              return sum;  
      }

Algo-3

int fun3(int n){  
    for (i=1; i<=n; i++)  
        for (int j=1; j<=i; j++)  
            sum++;  
    return sum;  
}

### Asymptotic Notation

mathematical tool to represent the time complexity of algorithm

① In algo 1 → the time taken by the program to execute is constant & irrespective of 'n'

$$[fun1() \rightarrow C_1]$$

② In algo 2 → there is some constant work. • initialisation of variables  
and rest depends linearly on 'n'  
 $[fun2() \rightarrow C_1 + C_2(n)]$

③ In Algo 3 →  $[fun3() \rightarrow C_1 + C_2n + C_3n^2]$

### Order of Growth

way of predicting how execution time of a program & the space/memory occupied by it changes with input size. It gives worst case possibility for an algorithm

### Limitations

- suppose algo 1 is running in a
  - + slower machine
  - + slow prog. language
- hence C comes out to be 1000.

### Assumptions

$$n \geq 0$$

$$\text{time taken} \geq 0$$

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

On the other hand algo2 runs on a faster machine and faster prog. language, making  $c_1 + c_2 \rightarrow 1$

$$f(n) \rightarrow n+1$$

By predicting through order of growth, we always talk about cases when  $n \rightarrow \infty$ , but in practical case this situation might never happen.

$$1000 \geq n+1 \\ n \geq 999.$$

| Algo 1 is faster than algo 2 when  $n \geq 999$ . However in practical cases, this situation might never happen.

\* A function  $f(n)$  is said to be growing faster than  $g(n)$  if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \rightarrow \infty$  OR  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \rightarrow 0$

$$f(n) \rightarrow n^2 + n + 6 \\ g(n) \rightarrow 2n + 5$$

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \rightarrow \lim_{n \rightarrow \infty} \frac{2n+5}{n^2 + n + 6} \\ \rightarrow \lim_{n \rightarrow \infty} \frac{2/n + 5/n^2}{1 + 1/n + 6/n^2}$$

$\rightarrow 0$  (Hence  $f(n)$  is a bad algorithm)

Direct way of predicting Order of Growth

- 1 - ignore lower order terms
- 2 - ignore leading constants

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

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

$$g(n) = 100n + 3 \rightarrow n$$

$$f(n) > g(n).$$

(growing faster)



$$f(n) = c_1 \log n + c_2$$

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

$$f(n) \rightarrow \log n$$

$g(n) \rightarrow n$ .  $g(n)$  grows faster

$$f(n) = c_6 n^2 + c_7 n + c_8$$

$$g(n) = c_9 n \log n + c_{10} n + c_{11}$$

$$\rightarrow n^2$$

$f(n)$  grows faster

### Asymptotic Notations

int sum ( int arr[], int n){

    if ( n % 2 != 0 )

        return 0;

    for ( int i=0 ; i < n ; i++ ) {

        sum = sum + i

}

    return sum

#### Worst case

upper bound on running time algo

\* In this case when  $n$  is even

$O(n)$

Assumption  
odd + even are equally likely hence  
 $+ g(n) \rightarrow O(n/2)$

### Big O Notation (Upper Bound or order of growth)

• ignore the lower order term

• ignore the constant terms

•  $f(n) = O(g(n))$  then  $f(n) \leq c g(n) + n \geq n$

Ex:-  $f(n) \rightarrow 2n+3$   
 $\rightarrow O(n)$

Here  $g(n) \rightarrow n$

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

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

$2n+3 \leq 3n$

$n \geq 3$ . hence  $[n \rightarrow 3]$

$n \geq \underline{n_0}$

$c$  = constant of highest growing term  
in this ex  $\underline{+1}$

Big O notation is always equal or more than the order of growth

$$\{n/4, 2n+3, n_{100} + \log n, n+1000, n/1000, \dots \text{Log}^n\}$$

- It is not a good idea to belong  $\log n$  to  $O(n)$  but it can belong to  $O(n)$ , ideally it belongs to  $O(\log n)$

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

$\{1, 200, 10000, \dots\} \in O(1)$

### Application

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

Best Case  
Element matches with first index.  
 $\Theta(1)$

Average Case  
order of growth  
 $g(n) \rightarrow n/2$

Worst Case  
found at last position & not present- $O(n)$

### Omega Notation

opposite of Big O notation  $\rightarrow$  provides a lower bound

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

order of growth.

$c \rightarrow 1$  (coefficient of highest growing term - 1)

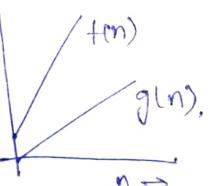
$$f(n) \rightarrow 2n+3$$

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

$$n < 2n+3$$

$$n_0 = 0$$

$$n \geq n_0$$



Big O notation bounds from upper side & Omega notation bound from lower side

- (4)  $\{n/4, n/2, 2n, 3n, \dots, n^2, 2n^3, n^4\} \rightarrow \Omega(n)$  as it bounds from lower side all order of growths greater than  $\infty$  equal to linear comes under  $\Omega(n)$ .

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

$$\text{then } g(n) = \Omega(f(n))$$

- (3) useful when we want to analyse lower bound of algorithms.

### Theta Notation

Theta notation bounds a function from upper side and lower side  $\rightarrow$  exact order of growth.

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

$$cg(n) \leq f(n) \leq Cg(n) \quad \forall n \geq n_0$$

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

$$\Omega(n) \quad cg(n) \leq f(n) \leq Cg(n)$$

$$cn \leq 2n+3 \leq Cn$$

coefficient of leading term  
 $-1$   
 $\frac{1}{2}$

coeff. of leading term + 1 ( $2+1$ )

$$n \leq 2n+3 \leq 3n$$

$$n \geq 0$$

$n \geq 3$   $\gamma$  max of these terms  
 $n_0 = 3$

$$\text{if } f(n) = \Theta(g(n))$$

$$\text{then } f(n) = \Omega(g(n)) \pm f(n) = \Omega(g(n))$$

$$\text{and } g(n) = \Omega(f(n)) \pm g(n) = \Omega(f(n))$$

- Theta notation is useful to represent time complexity when we know exact bound.

$$\{n^2/4, n^2+n, 2n^2+\log n, \dots\} \rightarrow \Theta(n^2)$$

highest order term must be  $n^2$ .

## Analysis of common loops

- `for (int i=0; i<n; i+=c) {  
 -- work -- O(1) work  
}`  $\rightarrow \Theta(n/c)$  ignore const.  
 $\rightarrow \Theta(n)$
- `for (int i=n; i>0; i-=c) {  
 some O(1) work  
}`  $n=10 \quad 10, 2, 4, 6, 8$   
 $n=11 \quad 0, 2, 4, 6, 8$
- ex  $\rightarrow n=10 \rightarrow 10, 8, 6, 4, 2 \quad 5$  times  
 $n=11 \rightarrow 11, 9, 7, 5, 3, 1 \quad 6$  times  
 $\Theta(n)$   $\lceil n/c \rceil$  runs ceiling of  $n/c$  times

- `for (int i=1; i<n; i=i*c) {  
 -- O(1) work --  
}`

$$n=32 \quad /c=2 \\ 1, 2, 4, 8, 16,$$

order of growth  
 $\rightarrow \Theta(\log n)$

for general c  
 $1, c, c^2, c^3, c^{k-1}$   
 $c^{k-1} < n$

$$(k-1)\log c < \log n \\ k-1 < \log n / \log c$$

$0 \rightarrow k-1$   
it runs k times

$$k = \frac{\log n}{\log c} + 1$$

- `for (int i=n; i>0; i-=i/c) {  
 -- O(1) work --  
}`  $\rightarrow \Theta(\log n)$

$$n=32 \quad c=2 \\ 32, 16, 8, 4, 2, 1$$

- `for (int i=2; i<n; i=pow(i, c)) {  
 -- O(1) work --  
}`

$$c=2, n=32$$

$$2, 4, 16$$

$$2 \cdot 2^c \cdot (2^c)^c$$

$$2 \cdot 2^c \cdot 2^{c^2} \cdots$$

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

$\Theta(\log \log n)$

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

$$k-1 < \log \log_2 n$$

• `fun (int n) {`

`for (int i=0; i<n; i++) {  
 -- O(1) work --  
 }`  $\rightarrow \Theta(n)$

`for (int i=1; i<n; i=i*2) {  
 -- O(1) work --  
 }`  $\rightarrow \Theta(\log n)$

`for (int i=1; i<100; i++) {  
 -- O(1) work --  
 }`  $\rightarrow \Theta(1)$

$$\Theta(n) + \Theta(\log n) + \Theta(1) \rightarrow \Theta(n) \quad (\text{by ignoring lower order terms})$$

`for (int i=0; i<n; i++)`  $\rightarrow \Theta(n)$

`for (int j=1; j<n; j=j*2) {  
 -- O(1) work.  
 }`  $\rightarrow \Theta(\log n)$

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

+Q  $\Theta(n \log n) + \Theta(n^2) \rightarrow \Theta(n^2)$ .

## Analysis of Recursion

```
void fun (int n) {
    if (n ≤ 1)
        return;
    for (int i=0; i<n; i++)
        print ('GFG');
    fun (n/2);
    fun (n/2);
```

3.

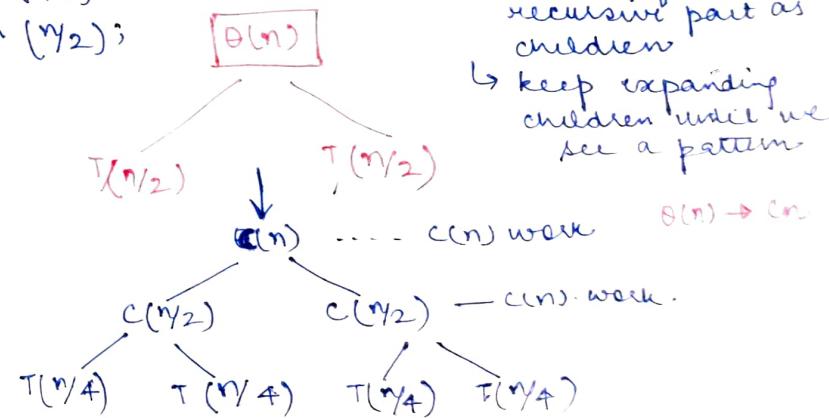
$$T(n) \rightarrow 2T(n/2) + \Theta(n)$$

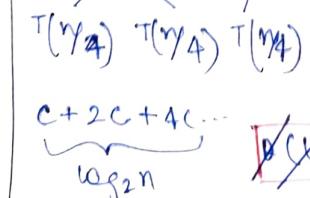
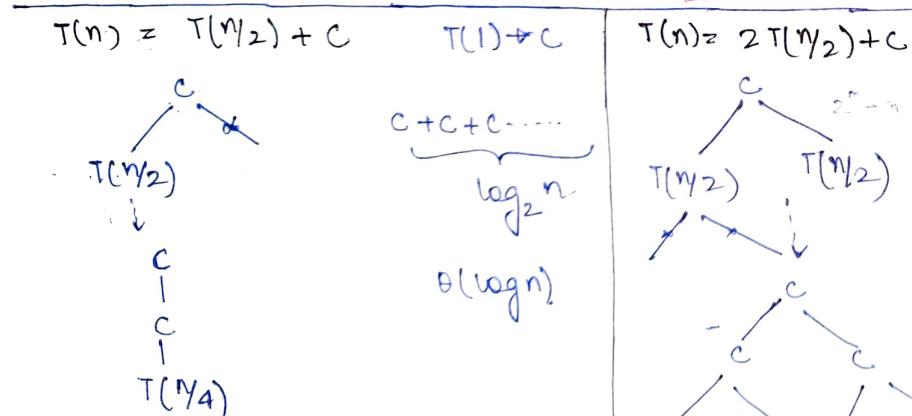
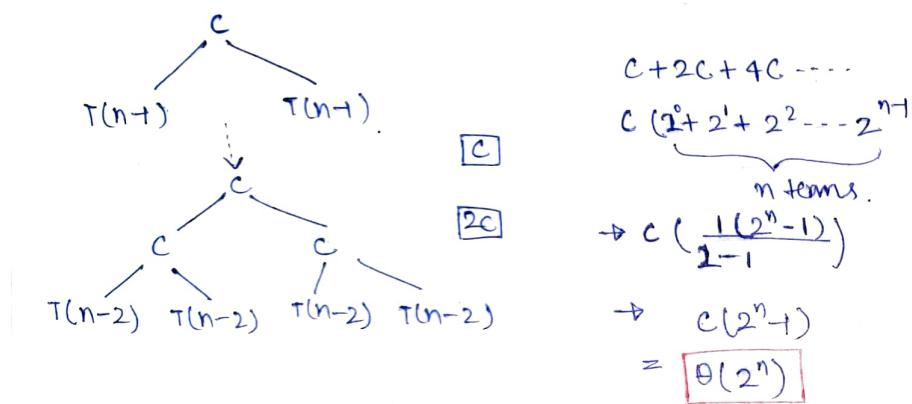
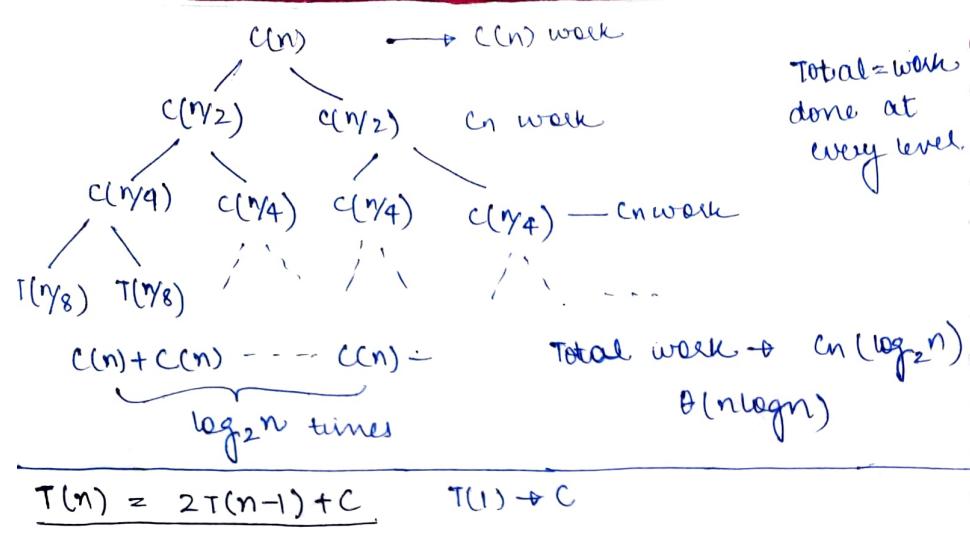
$$T(1) \rightarrow C$$

## Recursion Tree Method

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

↳ keep expanding children until we see a pattern

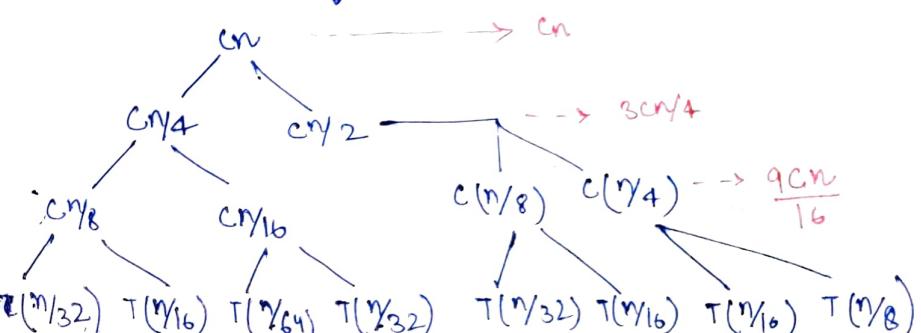
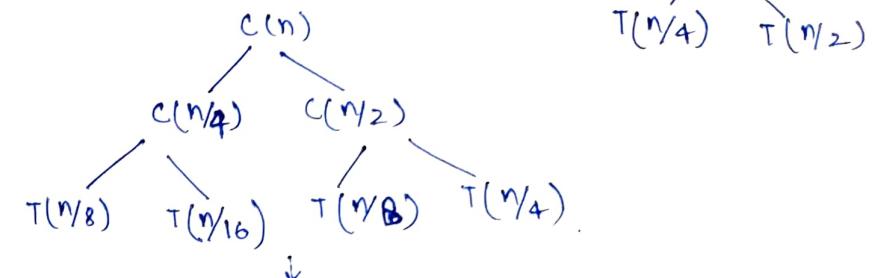




Total work done at every level.

no of terms  $\rightarrow \log_2 n$   
 $\text{GP} \rightarrow \frac{1(2^{\log_2 n} - 1)}{2-1} \rightarrow 2^{\log_2 n} - 1 \rightarrow \Theta(n)$

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



Hence we assume it as a full tree and calculate the upper bound.

As the left most branch i.e. reducing by  $n/4$  will terminate early in comparison to right most branch.

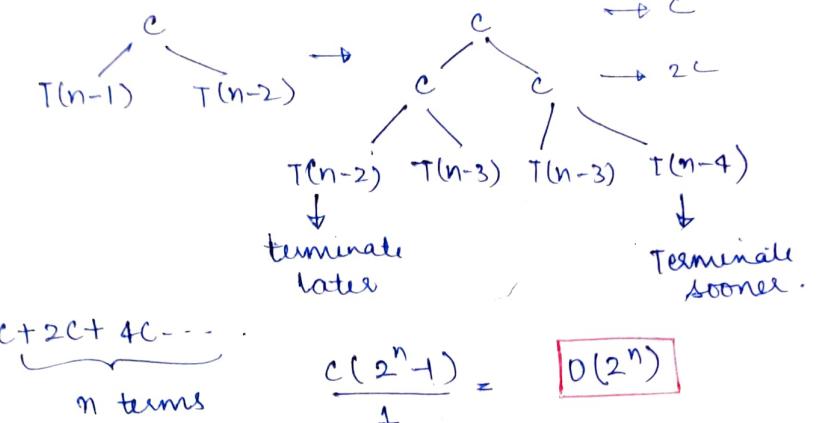
$$cn \left[ \frac{1}{1-3/4} \right] \rightarrow cn(4) \rightarrow O(n)$$

↪ big O notation upper bound

$\infty$  infinite series

Fibonacci series —  $T(n) \rightarrow T(n-1) + T(n-2) + c$

$T(n) \rightarrow T(n-1) + T(n-2) + c$



Space complexity → Memory taken

Order of growth of memory taken

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

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

→ array size depends on  $n$

$\Theta(n)$

② function (int arr[], int n){  
 int sum = 0;  
 for (int i=0; i<n; i++){  
 sum = sum + arr[i];  
 }  
 return sum  
}

Auxiliary Space

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

③ auxiliary space →  $\Theta(1)$

int fun(int n){

if ( $n \leq 0$ )

return 0

return  $n + f(n-1)$

3. Space complexity

Recursion calls are stored in stack.

Worst case → when  $fun(0)$  is being executed, then stack will look like

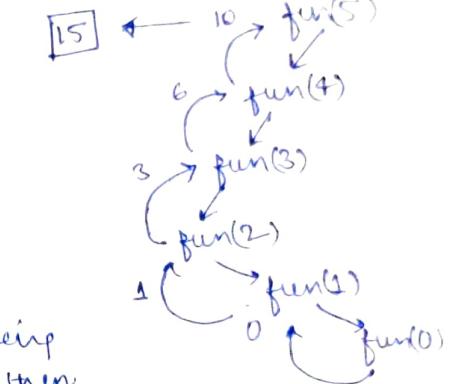
fun(0)
fun(1)
fun(2)
fun(3)
fun(4)
fun(5)

Fibonacci series

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

0 1 1 2 3 5  
↑ ↑ ↑ ↑ ↑ ↑  
 $n=0$   $n=1$   $n=2$   $n=3$   $n=4$   $n=5$

Auxiliary space → maximum length of leaf to root path →  $\Theta(n)$



Different method of finding  $n$  Fibonacci number

```
int fib(int n){  
    int f [n+1];  
    f[0] = 0;  
    f[1] = 1;  
    for (int i=2; i<=n; i++)  
        f[i] = f[i-1] + f[i-2];  
    return f[n];  
}
```

fib(0)
fib(1)
fib(2)
fib(3)
fib(4)

→ Auxiliary space:  $\Theta(n)$   
Space comp.:  $\Theta(n)$ .

## Another solution for fibonacci numbers

```

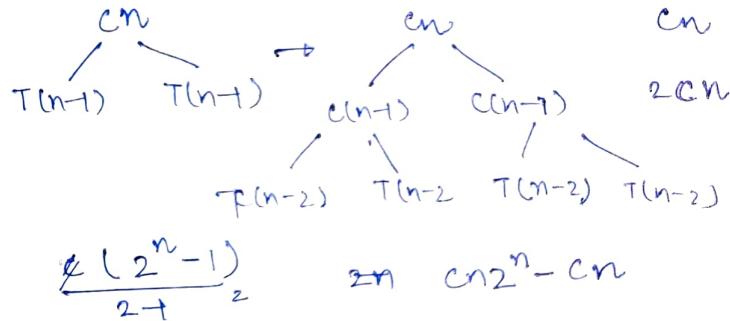
int fib(int n){
    if (n == 0 || n == 1)
        return n;
    int a = 0, b = 1;
    for (int i=2; i <= n; i++) {
        c = a+b;
        a = b;
        b = c;
    }
    return c;
}

```

comparison between  $2^n$  and  $n^{\log n}$

$$\begin{array}{ll} n^{\log 2} & \log n^{\log n} \\ \Theta(n) & \Theta(\log \log n) \quad \Theta((\log n)^2) \end{array}$$

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



space complexity  
 $\rightarrow \Theta(1)$   
 space comp.  
 (Auxiliary)  
 $\Theta(1)$

## Finding the no of digits

### ① Iterative solution

```

int countdigs (long n){
    int count = 0;
    while (n != 0) {
        n = n/10;
        count++;
    }
    return count;
}

```

### ② Recursive solution

```

int countdigit (log n).{
    return floor(log n + 1)
}

```

\*  $\log_{10}(123) \rightarrow 2.\text{something}$   
 $\log_{10}(1234) \rightarrow 3.\text{something}$

## Arithmetic and Geometric Progression

AP  $\rightarrow 2, 4, 6, 8, \dots$

$$\text{sum} \rightarrow \frac{\text{sum}}{n}$$

$$\text{sum} \rightarrow \text{avg} \times n$$

$$= \frac{FT + LT}{2} \times n + \frac{n}{2}(2a + (n-1)d)$$

GP  $\rightarrow 2, 4, 8, 16, \dots$

$$\text{sum} \rightarrow \frac{a(1-\gamma^n)}{1-\gamma}$$

## quadratic equations

$$ax^2 + bx + c = 0 \quad x \rightarrow \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

## Mean and Median

mean  $\rightarrow$  adding all the entries  
 no of entries

median  $\rightarrow$  ordering all the numbers and picking the middle one

$$4, 1, 7 \rightarrow 1, 4, 7 \rightarrow 4$$

for even entries we take mean of the middle terms

$$1, 4, 2, 8 \rightarrow 1, 2, 4, 8 \rightarrow 3$$

## Basic Mathematics

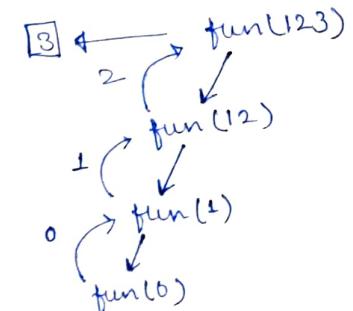
### ③ Recursive solution

```

int countdigs (long n){
    if (n == 0)
        return 0;
    return 1 + countdigs (n/10);
}

```

Ex - 123



## Prime Number

- divisible by 1 and itself only.
- every prime number can be represented as  $6n+1$  and  $6n+1$  except 2 and 3.
- 2 and 3 are only consecutive P.N.
- LCM & HCF

long division method to find HCF 30, 42

$$\begin{array}{r} 30 ) 42 ( 1 \\ \underline{30} \quad \quad \\ 12 ) 30 ( 2 \\ \underline{24} \quad \quad \\ 6 ) 12 ( 2 \\ \underline{12} \quad \quad \\ 0 \end{array}$$

```
function gcd(a, b) {
    if (a == 0)
        return b;
    return (b % a, a);
}
```

LCM  $\times$  HCF  $\rightarrow a \times b$

For fraction HCF  $\rightarrow \frac{\text{HCF}(\text{num})}{\text{LCM}(\text{den})}$

$$\text{LCM} \rightarrow \frac{\text{LCM}(\text{num})}{\text{HCF}(\text{den})}$$

- No of trailing zeroes in prime factorization of  $n$ .
- $5! \rightarrow \text{floor}(5/5) + \text{floor}(5/25) \dots$  1 trailing zero
- $n! \rightarrow \text{floor}(n/5) + \text{floor}(n/25) \dots$

## Bitwise Operators in C++

- Operate on binary representation of numbers

$$\begin{array}{ll} x = 3 & x \& y \Rightarrow 011 \& 100 \Rightarrow 000 (0) \\ y = 4 & x | y \Rightarrow 011 | 100 \Rightarrow 111 (7) \\ & x \wedge y \Rightarrow 011 \wedge 100 \Rightarrow 111 (7) \\ & \quad \quad \quad \downarrow \quad \downarrow \quad \downarrow \\ & \quad \quad \quad 1 \quad 1 \quad 1 \end{array}$$

output is 1

when input bits  
are different and 0  
when they are same

+ They do internally bitwise  
operations and produce O/P  
in decimal form

## Left Shift Operator

$a \ll b$   
 ↑  
 no whose  
binary rep.  
is to be shifted  
 how many  
times it  
needs to be  
shifted

$$x \ll y \Rightarrow x * 2^y$$

(assuming that initial  $y$  bits in  
binary representation of  $x$  is less  
than or equal to  $y$ .)

$$\begin{array}{l} x = 3 : 0000\dots 011 \\ \textcircled{1} x \ll 1 : 000\dots 0110 \text{ (shifted left by 1)} \\ \textcircled{2} x \ll 2 : 000\dots 01100 \end{array}$$

## Right Shift Operator

$a \gg b$   
 last ' $b$ ' bits are ignored and  
 $'b'$  0 bits are added in the  
 beginning

$$x \gg y \text{ is equivalent to } \left\lfloor \frac{x}{2^y} \right\rfloor$$

$$33 \gg 1 \rightarrow 16$$

$$\left\lfloor \frac{33}{2^1} \right\rfloor = \left\lfloor 16.5 \right\rfloor = 16$$

AND - &
OR -
XOR - ^

Binary representation of  
6 → 0...00110  
might be 29 or 81 or 13  
depending on the  
compiler. If compiler  
is 32 bit for int then  
trailing zeroes will be  
29.

## Bitwise Not ( $\sim$ )

$$\begin{array}{ll} x = 1 & 00 \dots 001 \\ \sim x \rightarrow & 11 \dots 110 \\ x = 5 & 00 \dots 101 \\ \sim x \rightarrow & 11 \dots 010 \end{array}$$

maximum value will be when all bits are 1  
 $2^n - 1 \rightarrow 2^{32} - 1$  (32 bit computer)

$2^k$ 's bit representation of a number  $x$  in  $n$  bits is

$$\begin{array}{ll} 2^n - x & x + -2 \\ \text{negative} \\ \text{number} & 2^3 - 2 \rightarrow 6 = 110 \end{array}$$

If computer uses  $2^k$ 's compliment method to represent numbers and in 32 bits

$$\begin{array}{ll} x = 1 \rightarrow & 000 \dots 01 \rightarrow 111 \dots 10 \rightarrow 2^{32} - 1 - 1 \\ \sim x. & \text{for all } 1's \\ \text{Comparing} & \boxed{= 2^{32} - 2} \\ 2^{32} - 2 \rightarrow 2^n - x & \boxed{\sim x \rightarrow -2} \end{array}$$

$$x = 5 \rightarrow 000.0101 \rightarrow 111 \dots 1010$$

$$\boxed{\sim x \rightarrow -6} \quad 2^{32} - 5 \rightarrow 2^{32} - 6$$

Check if  $k^m$  bit is set

$$\text{I/P} \rightarrow n = 5 \quad k = 1 \quad \rightarrow 000 \dots 0101 \quad \boxed{\text{Yes}}$$

$$\text{I/P} \rightarrow n = 8 \quad k = 2 \quad \rightarrow 000 \dots 1000 \quad \boxed{\text{No}}$$

$$\text{I/P} \rightarrow n = 0, \quad k = 3 \quad \rightarrow 000 \dots 000 \quad \boxed{\text{No}}$$

$k \leq$  no of bits in binary

## Algorithm

We are trying to make a number with only  $k^m$  bit set.

$$\begin{array}{l} 1 \ll (k-1) \\ \downarrow \\ 000 \dots 001 \ll 2 \\ 000 \dots 100 \\ \text{only } 3^{\text{rd}} \text{ bit set} \end{array}$$

$$\begin{array}{l} 5 \rightarrow 000 \dots 0101 \\ 1 \ll (k-1) \rightarrow 000 \dots 0100 \\ \hline 000 \dots 100 \neq 0 \\ \text{Yes} \end{array}$$

## Second Method

shift the  $k^m$  bit to the last position and then do a bit wise and with 1.

To do that we set the  $k^m$  bit to 1 by doing a right shift operation by  $(k-1)$

$n \gg (k-1) \rightarrow$   $\uparrow$   $k^m$  bit is shifted to 1<sup>st</sup> bit and then do a bit wise AND with 1.

## Count Set Bits

$$\begin{array}{ll} \text{I/P} \rightarrow 5 & \text{I/P} = 13 \\ \text{O/P} \rightarrow 2 & \text{O/P} = 3 \end{array}$$

## 2<sup>nd</sup> Approach

Brian Kernighan's Algo.

$$n = 40 \rightarrow 000 \dots 0101000$$

$$\text{after 1<sup>st</sup> iteration} \rightarrow 000 \dots 0100000$$

$$\text{after 2<sup>nd</sup> iteration} \rightarrow 000 \dots 0000000$$

\* When we subtract 1 from a number, all the bits after the last set bit becomes 1 and the last set bit becomes 0.

$$n = 40 \quad 101000$$

$$n = 39 \quad 100111$$

$$n \& (n-1) \quad \hline 100000$$

## 1<sup>st</sup> Approach

check the last bit (LB)

if (LB = 1) count++

left shift by 1.

→ while( $n > 0$ ) {      or       $n \& 1$

{ if ( $(n \& 1) != 0$ ) {       $n \& 1$

    count++; }

// Remove last bit

$n = n/2$  or  $n = n \gg 1$

can be rewritten as

count = count + ( $n \& 1$ );

Time complexity  $\rightarrow \Theta(\text{total bit } n)$

cont..

$n = 32$   
 $n = 31$   
 $n \& (n-1)$   
 $\underline{000000}$

1000000  
011111

It require less time than previous one  $O(\text{no of set bits})$ .

```

Hgo while(n > 0) {
    n = n & n & (n-1);
    count++;
}

```

### Lookup Table Method for 32 bit number

- divide number into 8 bit chunks
- we check the number of set bits in each number  
 $0 \rightarrow 255$ , we make a table (array) for it and  
 the value of  $\text{table}[i] = \text{no of set bits in } i$

table[0] $\rightarrow$ 0	table[0] $\rightarrow$ 0
table[1] $\rightarrow$ 1 (01)	table[1] $\rightarrow$ 1 (10)
table[2] $\rightarrow$ 1 (10)	table[2] $\rightarrow$ 2 (11)
table[3] $\rightarrow$ 2 (11)	table[3] $\rightarrow$ 3 (111)
⋮	⋮
table[255] $\rightarrow$ 7 (1111111)	⋮

```

functions int count(int n) {
    res = res + table[n & 0xFF],
    n  $\gg= 8$  (shift the number
    by 8 bits)
    res += table[n & 0xFF]
    n  $\gg= 8$ 
    res += table[n & 0xFF]
    n  $\gg= 8$ 
    res += table[n & 0xFF].
    return res;
}

```

$\Theta(1)$  solution

table[n & 0xFF]

+  
bitwise and with  
'8' set bits

$0xFF \rightarrow 11111111$

$n \rightarrow 72 \rightarrow 01001000$ ,

basically we are  
extracting last 8 bits  
from a number



To check whether a number is a power of 2

### ① Naive Solution

- Repeatedly divide number until the result becomes 1
- $n = \text{even power of 2} \rightarrow 1$
- $n = \text{odd} (\text{Not a power of 2}) \rightarrow \text{odd}$

①  $n=1$  (Yes)

$n=\text{odd}$  (No)

while( $n != 0$ ) {

```

    if (n % 2 != 0) {
        return false;
    }
    n = n / 2;
}

```

### ② Brian's Kernelgarn Algorithm

- using the fact that if a number is a power of 2 then it has only 1 bit set

### ③ 1 line solution

```
bool Pow2(int n) {
```

```

    if (n == 0)
        return false;
    return ((n & (n-1)) == 0);
}

```

```

    return (n != 0) && ((n & (n-1)) == 0)
}

```

Example :

$n=4$	0100
$n=3$	0011
	<u>0000</u>
$n=6$	0110
$n=5$	0101
	<u>0100</u> X

### Find only Odd occurring number

I/P : arr[]  $\rightarrow \{4, 3, 4, 4, 5, 5\}$ ,

O/P : 3

### • using XOR operator

$x \wedge 0 = x$

$x \wedge x = 0$

$x \wedge y = y \wedge x$

$res = 0$

```

for (int i=0; i<n; i++) {
    res = res  $\wedge$  arr[i];
}

```

```

return res;
}

```

• only 1  
odd occurring  
no. must be  
present

$res = 0$   
 $arr[0] = 4$   
 $res \wedge arr[0] \rightarrow 0$   
 $\Rightarrow 0 \wedge 4 = 4$   
 $arr[1] = 3$   
 $3 \wedge 4 \rightarrow 3$   
 $arr[2] = 4$   
 $3 \wedge 4 \wedge 4 \rightarrow 3$   
 $3 \wedge 4 \rightarrow 3$   
 $3 \wedge 4 \wedge 4 \rightarrow 3$   
 $3 \wedge 5 \rightarrow 3$   
 $3 \wedge 5 \wedge 5 \rightarrow 3$   
 $\cancel{3 \wedge 5 = 3}$

Find missing number in a array [1...n+1]

I/P arr[] = {1, 4, 3}.

O/P → 2

I/P arr[] = {1, 5, 3, 2}

O/P → 4

```
for (int i=0; i<n; i++) {  
    res = res ^ arr[i];  
}
```

```
for (int i=1; i<n+1; i++) {  
    res = res ^ i  
}
```

```
return res;
```

### Variation 1

number given in range [n, m].

### Variation 2

finding 2 odd occurring numbers

## Finding two odd occurring numbers

### ① Naive solution

loop through all elements and just count how many times it occurs and if count % 2 != 0 print the element.

### ② Optimised solution

XOR of all the numbers

I/P → arr[] → {2, 2, 3, 4, 4, 3, 3, 5, 5, 5}

O/P → 3, 5

res → 3 ^ 5

$$\begin{array}{r} 3 \rightarrow 011 \\ 5 \rightarrow 101 \\ \hline p \rightarrow 110 \end{array}$$

Now we divide 2 groups on the basis of set bit of p i.e (2nd last bit)

group 1 (having 2nd last bit 1)      group 2 (0)

{2, 2, 3, 3, 3}

{4, 4, 5, 5, 5}

It is ensured that both odd numbers are in different group. Now we can separately perform (find 1 odd no) algo in both & can find the answer.

## Implementation

```
void oddAppearing (int arr[], int n) {
```

```
    int XOR = 0, res1 = 0, res2 = 0;
```

```
    for (int i=0; i<n; i++) {
```

```
        XOR = XOR ^ arr[i];
```

```
}
```

```
int setbit = XOR & ~ (XOR - 1); Rightmost set bit
```

```
for (int i=0; i<n; i++) {
```

```
    if ((arr[i] & setbit) == 0) {
```

```
        res1 = res1 ^ arr[i]; }
```

```
else
```

```
    res2 = res2 ^ arr[i];
```

```
}  
print (res1, res2);
```

### To identify last set bit

Focus is to remove all the bits before last set bits

n = 40 → ...0010 1000

n-1 = 39 → 00100111

n(n-1) → 1101 1000

n & n(n-1) → 00001000

last set bit  
is at position 4

### To put element in different groups

we make an AND with the number and setbit and

"if ans == 0 → put in grp 2

1 → put in grp 1 .

## Generating Power Set

I/P → "abc"

O/P → "", "a", "b", "c", "ab", "ac", "bc", "abc"

I/P → n

O/P → 2^n

As we know that the output will be  $2^n$ .

counter	Binary	subset
0	000	,
1	001	'a'
2	010	'b'
3	011	'ab'
4	100	'c'
5	101	'ac'
6	110	'cb'
7	111	'abc'

```

void print(string str) {
    int n = str.length(); // n=3
    int powsize = pow(2, n); // 8
    for (int counter = 0; counter < powsize; counter++) {
        for (int i = 0; i < n; i++) {
            if (counter & (1 << i)) {
                print(str[i]);
            }
        }
        print("\n");
    }
}

```

Time complexity  $\rightarrow \Theta(2^n * n)$

## Recursion

- function calls itself

- Direct Recursion

```

void fun1() {
    fun1();
}

```

- without terminating condition

```

fun(int n) {
    cout << "a"; fun(2);
}
main() {
    fun(2);
}

```

I/P  $\rightarrow$  ab  
O/P  $\rightarrow$  "", a, b, ab

counter	B	ss.
0	00	,
1	01	'a'
2	10	'b'
3	11	'ab'

Random case  
counter = 3 + 011

- Iteration  
 $3 + (1 << 0) + 3 < 1$   
 $\rightarrow 011 + 001 \rightarrow 1 + 0$   
print(a)
- $3 < (1 << 1) + 3$   
 $011 + 010 \rightarrow 010 + 0$   
print(b)
- $3 < (1 << 2) + 3$   
 $011 + 100 = 0$

## Terminating condition

```

fun1(int n) {
    if (n == 0) {
        return;
    }
    cout << "ggg";
    fun1(n-1);
}

```

```

void main() {
    fun1(5);
}

```

## Typical Structure of Recursion

fun()  
Base cases

Recursive call (with atleast one change in parameter for termination).

```

fun(int n) {
    if (n == 0) {
        return;
    }
}

```

```

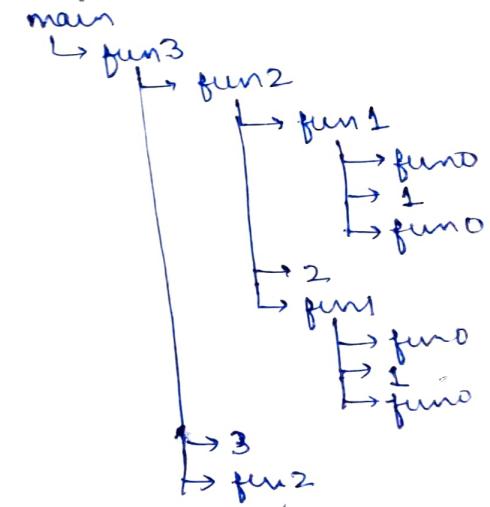
fun(n-1);
cout << n;
fun(n-1);

```

```

main() {
    fun(3);
}

```



```

D) int fun(int n) {
    if (n == 1) {
        return 0;
    } else
        return 1 + fun(n/2);
}

fun(1) → 0
fun(2) → 1 + fun(1) = 1
fun(4) → 1 + fun(2) = 2
;
fun(8) → 1 + fun(4) = 4
fun(16) → 1 + fun(8) = 4

```

↑  
fun(16)  
↳ fun<sup>8</sup> + 1  
↳ 1 + fun<sup>4</sup>  
↳ 1 + f<sup>(2)</sup>  
↳ H(10)

fun(16) → 4

it returns  $\log_2 n$

log<sub>2</sub>(20) → 4

\* in general it returns  $\log_m(n)$

Generally

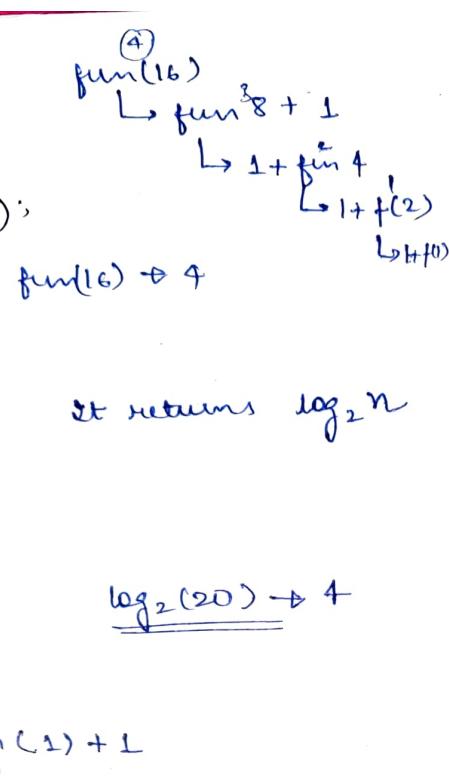
```

int fun(int n) {
    if (n < m) {
        return 0;
    } else
        return 1 + fun(n/m);
}

Ex) int fun(int n) {
    if (n < m) {
        return 0;
    } else
        return 1 + fun(n/m)

```

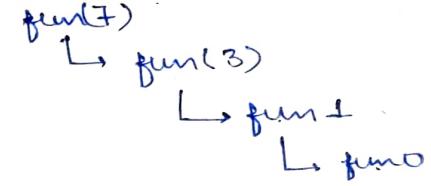
→  $\lfloor \log_m(n) \rfloor$



```

void fun(int n) {
    if (n == 0)
        return;
    fun(n/2);
    print(n % 2);
}

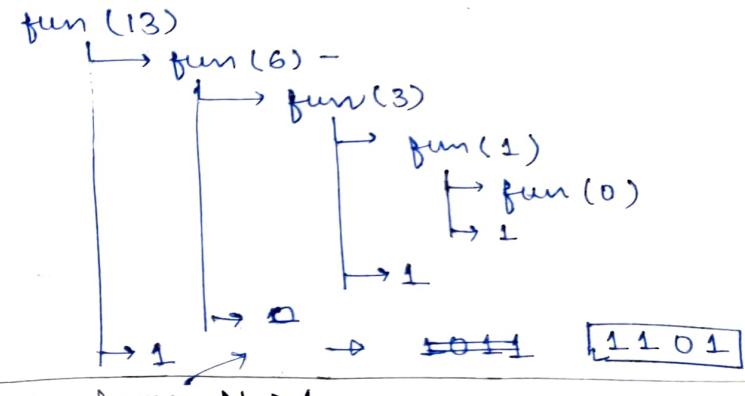
```



① ① ①

\* Prints the binary equivalent of a number

Ex) n = 13



Print numbers from N → 1

```

fun(int n) {
    cout << n;
    fun(n-1);
}

```

```

fun(int n) {
    if (n == 0) {
        return;
    }
    cout << n;
    fun(n-1);
}

```

θ(1)

```

fun(int x) {
    if (n == 0)
        return y;
    fun(n-1);
}

```

```

fun(int n) {
    if (n == 0) {
        return;
    }
    cout << n;
    fun(n-1);
}

```

fun(7)  
↳ 7  
↳ fun 6  
↳ 6  
↳ fun 5  
↳ 5  
↳ fun 4  
↳ 4  
↳ fun 3  
↳ 3  
↳ fun 2  
↳ 2  
↳ fun 1  
↳ 1

⑦ ⑥ ⑤ ④ ③ ② ①

## Tail Recursion

```
void fun(int n) {
    if (n == 0)
        return;
    print(n);
    fun(n-1);
}
```

- 3.
- A function is called tail recursive when the parent function has nothing more to do after child has returned a value.  
(Ex:- print N-1).

Note:- only modern compilers execute them faster by making the following changes internally.

```
void fun(int n) {
    start:
    if (n == 0)
        return;
    print(n);
    fun(n-1); } → n=n-1
    goto start
}
```

3.

The changes are called  $\rightarrow$  Tail call Elimination.

changing of normal function  $\rightarrow$  Tail Recursion

```
void fun(int n, int k) {
    if (n == 0)
        return;
    print(k);
    fun(n-1, k+1);
}
```

3.

to start printing  
no from 1

R+1

n-1

k

1

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

## Writing Base cases

(a) Factorial n when  $n > 0$

```
int fact(int n) {
    if (n == 0 || n == 1)
        return 1;
    return n * fact(n-1);
}
```

$$0! \rightarrow 1 \quad 1! = 1$$

(b)  $n^{th}$  fibonacci Number

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

$$\begin{matrix} 0 & 1 & 1 & 2 & 3 & 5 & 8 \\ 0 & , & 1 & , & 2 & , & 3 & , & 5 & , & 8 \end{matrix}$$

sum of n natural Numbers

fun(int n) {

return n + fun(n-1); }

}

X →

fun(5)

↳ 5 + fun(4)

↳ 4 + fun(3)

↳ 3 + fun(2)

↳ 2 + fun(1)

↳ 1 + fun(0)

Base case included

fun(int n) {

if (n == 0) {

return 0;

}

return n + fun(n-1);

Check if string is Palindrome

I/P : abba

O/P : Yes

We always try to solve for smaller cases

$s = s_1 s_2 \dots s_{n-2} s_{n-1}, s_n$   
check if inner string is palindrome  
check if  $s_0 = s_n$ .

$n=0$  or  $n=1$  (return true)

bool isPalindrome (string str, int start, int end) {

```
if (start >= end) {
    return true;
}
return (str[start] == str[end]) &&
    isPalindrome(str, start+1, end-1);
```

str = "aabcbabcba"

↓  
aabcbabcba

start	end
8	8
7	7
6	6
5	5
4	4
3	8
2	7
1	6
0	5
1	4
2	3
3	2
4	1
5	0
6	1
7	2
8	3

Time complexity  $\rightarrow O(n)$   
Auxiliary space  $\rightarrow O(n)$

Sum of Digits in a number

I/P  $\rightarrow 253$

O/P  $\rightarrow 10$

$n = 4537$

sum(4537)

$\rightarrow 7 + \text{sum}(453)$

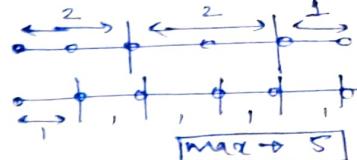
$\rightarrow 3 + \text{sum}(45)$

$\rightarrow 5 + \text{sum}(4)$   
 $\rightarrow 4 + \text{sum}(0)$ .

```
int sum (int n) {
    if (n == 0) {
        return 0;
    }
```

return (n%10) + sum(n/10);

Given a rope of length  $n$ , you need to find maximum number of pieces you can make, such that length of every piece is in the set  $\{a, b, c\}$  for given values  $a, b, c$ .



I/P  $\rightarrow n=5$   $a=2, b=5, c=1$

O/P  $\rightarrow 5$

I/P  $\rightarrow n=5$   $a=4, b=2, c=6$

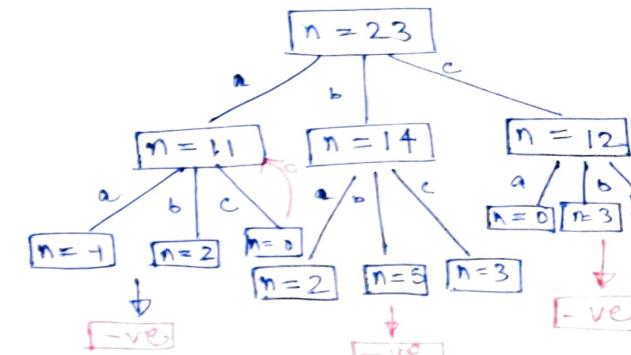
O/P  $\rightarrow -1$

I/P  $\rightarrow n=23$   $a=12, b=9, c=11$

O/P  $\rightarrow 2$

- If -1 is returned to a parent, choice is ignored.

- If 0 is returned, accepted as 1 of the solution



int maxCuts (int n, int a, int b, int c) {

if ( $n == 0$ ) return 0; // Base case

if ( $n < 0$ ) return -1;

int res = max(maxCuts(n-a, a, b, c),  
maxCuts(n-b, a, b, c),  
maxCuts(n-c, a, b, c));

if ( $res == -1$ ) return -1;

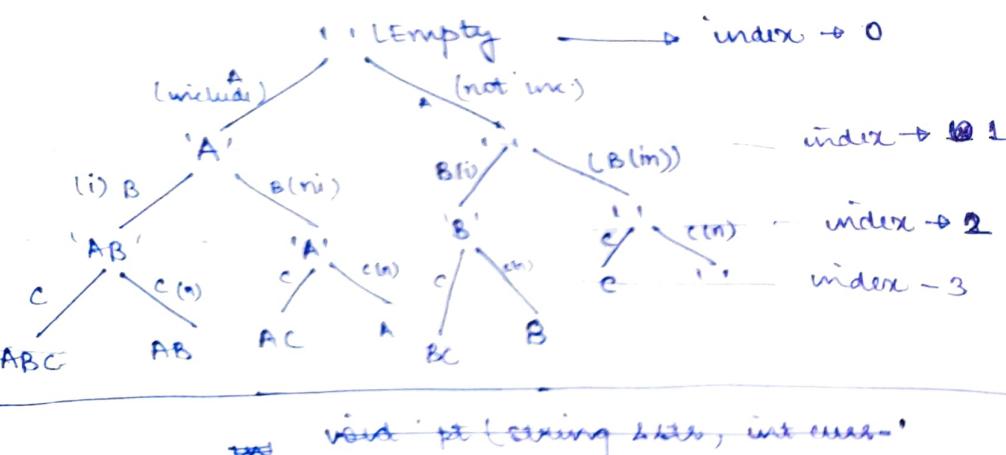
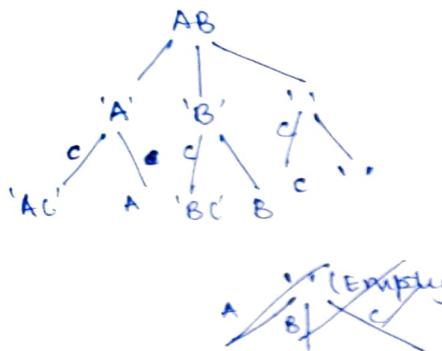
return res+1;

## Q/ Print all substrings

I/P  $\rightarrow$  str  $\rightarrow$  "ABC"

I/P  $\rightarrow$  "", "A", "B", "C", "AB", "BC", "ABC".

If we have a solution of string  $n-1$ , we can create solution for string of  $n$ .  
every character we have two choice  
include      not include



void printSubstr(string str, string curv="", int curr=0)

if (index == str.length) {  
    print(curv); return; }  
otherwise printSubstr(str, curv.append(str[i]), index+1);  
otherwise printSubstr(str, curv, index+1);

## Tower of Hanoi

- \* move disc from A  $\rightarrow$  C
- \* only 1 disc move at time
- \* largest is at bottom and smallest at top and only top disc can be shifted

I/P  $\rightarrow$  n = 1

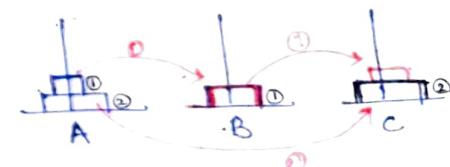
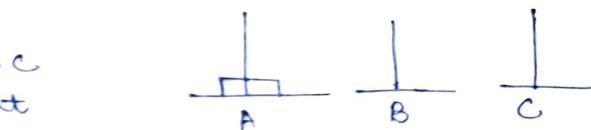
O/P  $\rightarrow$  move disc 1 from A to C (direct)

I/P  $\rightarrow$  n = 2

O/P  $\rightarrow$  ① from A  $\rightarrow$  B

② from A  $\rightarrow$  C

③ from B  $\rightarrow$  C.



## Approach

- shift  $n^m$  bar from A  $\rightarrow$  B and then shift  $(n-1)^m$  bar from A  $\rightarrow$  C and then  $n^m$  to B  $\rightarrow$  C

shift

- shift  $(n-1)$  from A  $\rightarrow$  B
- shift  $n^m$  from A  $\rightarrow$  C
- shift  $(n-1)$  from B  $\rightarrow$  C

TOH(n, A, B, C)

TOH(n-1, A, C, B)  
~~TOH~~ move n from A  $\rightarrow$  C  
TOH(n-1, B, A, C)

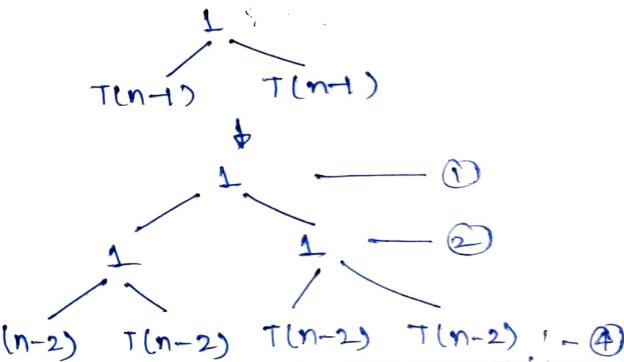
```

void TOH(int n, char A, char B, char C) {
    if (n == 1) {
        cout << "move" << A << "to" << C;
    } else {
        return TOH(n-1, A, C, B);
        cout << "move" << n << "from" << A << "to" << C;
        return TOH(n-1, B, A, C);
    }
}

```

No of movements (moves).

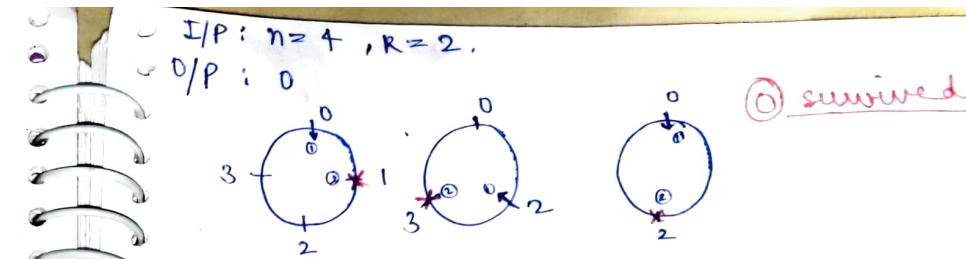
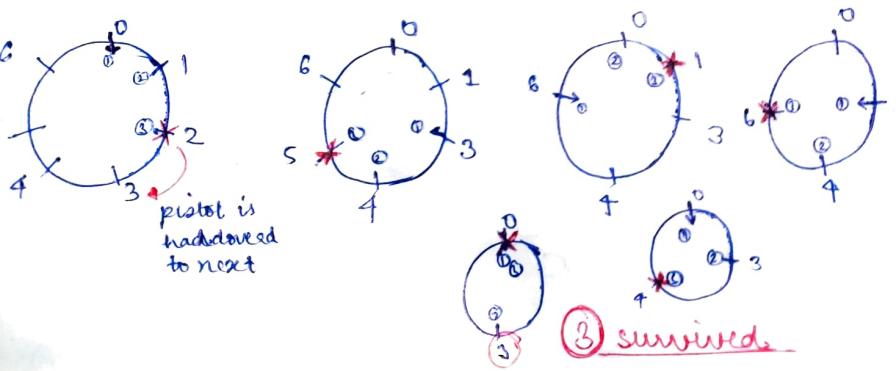
$$T(n) = 2T(n-1) + 1$$



### Josephus Problem

$n$  person are standing in a circle  
 $k$  person is to be killed (clockwise) and  
counting starts from the person itself.)

$$\text{Ex:- } n=7 \quad k=3$$

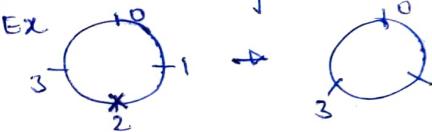


```

int jps(int n, int k) {
    if (n == 1)
        return 0;
    return (n-1, k)
}

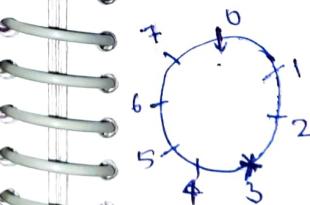
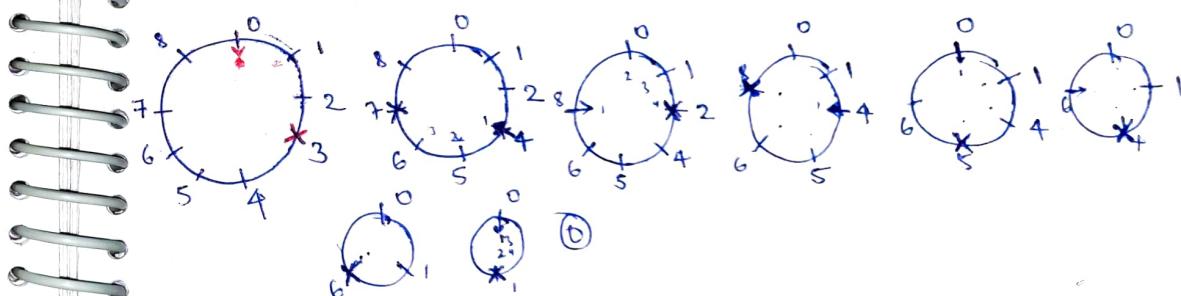
```

\* we are using numbers as per previous case.



But the compiler will not name them like this it will still name it as 0, 1, 2.

$$n=9 \quad k=4$$



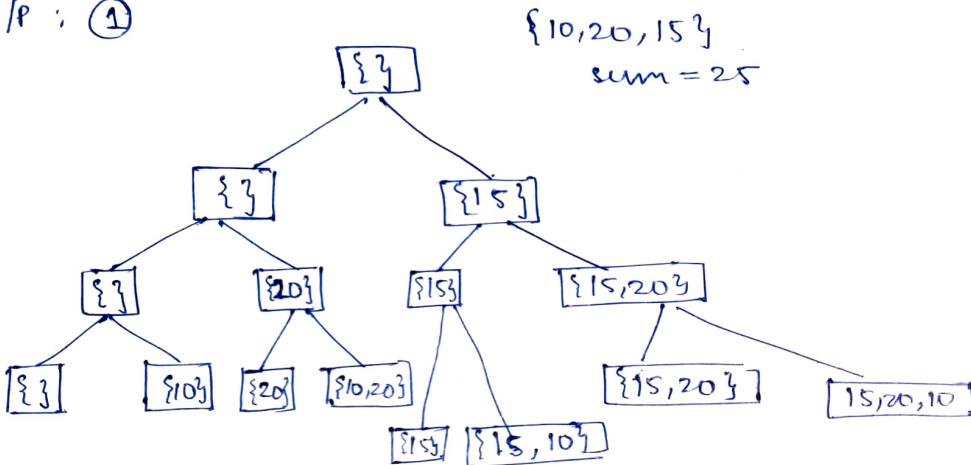
### Subset Sum Problem

I/P = {10, 5, 2, 3, 6} sum → 8

O/P → ②

I/P : {1, 2, 3} sum → 4

O/P : ①



```

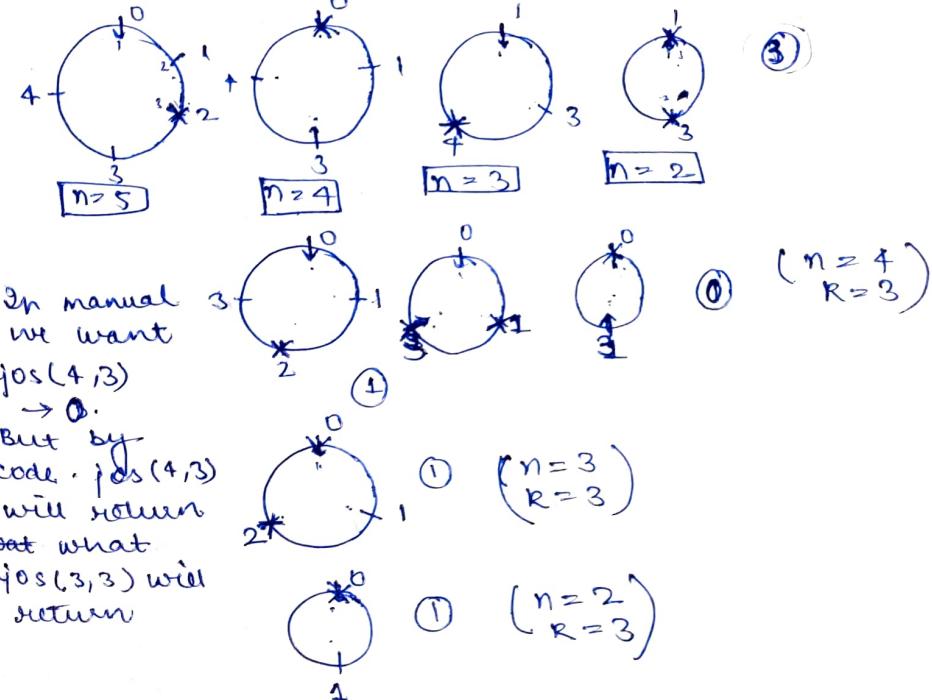
int check ( int arr, int sub, sum index) {
    if ( arr == index == arr.length()){
        if (sum of el. of sub == sum){
            return 1;
        }
    }
    return check ( int arr, int sub.push
                  ( arr [index]), sum)
          + check ( int arr, int sub, arr.index+1, sum);
}
  
```

I/P : {10, 20, 15} sum = 0

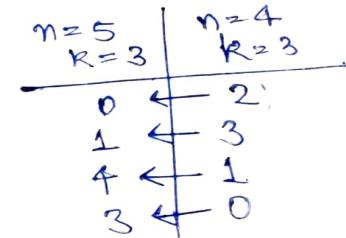
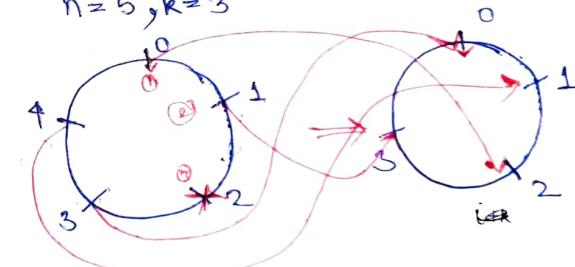
O/P = 1

### Contd. Recursion (Josephus)

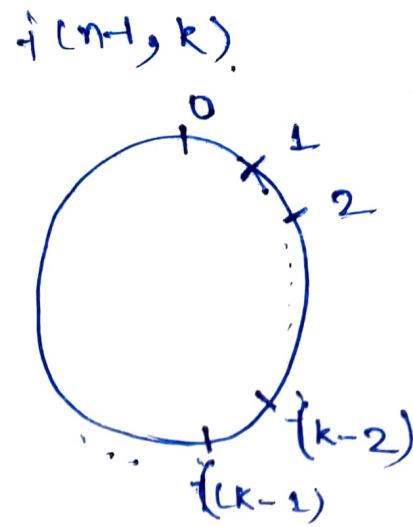
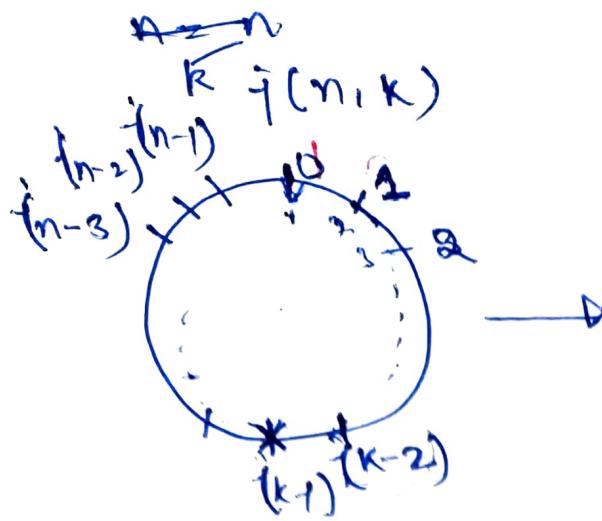
n = 5      k = 3



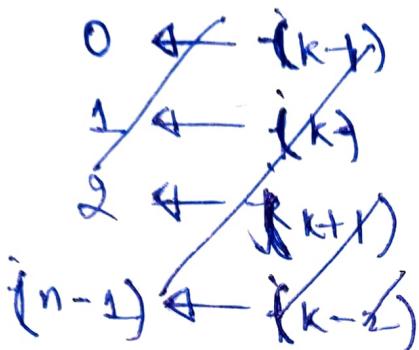
$n = 5, k = 3$



If  $\text{jos}(4, 3)$  will return 0, it means main func. will return ③.



$$\begin{aligned}
 & A = 1 \\
 & A = 2 \\
 & A = 3 \\
 & A = 4 \\
 & A = 5 \\
 & A = 6 \\
 & A = 7 \\
 & A = 8 \\
 & A = 9 \\
 & A = 10 \\
 & A = 11 \\
 & A = 12 \\
 & A = 13 \\
 & A = 14 \\
 & A = 15 \\
 & A = 16 \\
 & A = 17 \\
 & A = 18 \\
 & A = 19 \\
 & A = 20
 \end{aligned}$$



$$\begin{aligned}
 n &= 5, k = 3 \\
 1 &\rightarrow 3 \\
 R &\rightarrow 0 \\
 K &\rightarrow 1 \\
 K-2 &\rightarrow 4, n-1 = 5 \rightarrow 1 \\
 &= (R+3) \% 5 \\
 &= 1
 \end{aligned}$$

$$(i+R) \% n$$

$$i = \frac{n-1+k}{(n+k-1)} \% n$$

$$\left. \begin{array}{l} R \rightarrow 0 \\ K+1 \rightarrow 1 \\ K+2 \rightarrow 2 \\ K+i \rightarrow i \\ K-2 \rightarrow n-1 \end{array} \right\}$$

If  $j(n-1, k)$   
return  $i$   
then parent  
have to  
return  
 ~~$K+i$~~  to the  
parent

### Code for Josephus Problem

```

int jos (int n, int k) {
    if (n == 1) {
        return 0;
    }
    return (jos (n-1, k) + k) % n;
}

```

Time complexity

$$\begin{aligned}
 T(n) &\rightarrow T(n-1) + C \\
 &\hookrightarrow \Theta(n)
 \end{aligned}$$