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'

$\boxed{fun1() \rightarrow C_1}$

② In algo 2 → There is some constant work and rest depends linearly on 'n' $\boxed{fun2() \rightarrow C_1 + C_2(n)}$

- initialisation of variables
- return statements

③ In Algo 3 → $\boxed{fun() \rightarrow C_1 + C_2 n + C_4 n^2}$

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_1 comes out to be 1000.

Assumptions

$n \geq 0$

time taken ≥ 0

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

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

fun(·) → n+1

By predicting through order of growth, we always talk about cases when $n \to \infty$, 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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} \to \infty$ OR $\lim_{n\to\infty} \frac{g(n)}{f(n)} \to 0$

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

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

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

→ 0 (Hence f(n) is a bad algorithm)

Direct way of predicting Order of Growth

1 - Ignore lower order terms
2 - Ignore leading constants

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

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

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

(growing faster)

$f(n) = c_1 \log n + c_2$
$g(n) = c_3 n + c_4 \log\log n + c_5$

$f(n) \to \log n$
$g(n) \to n$.

g(n) grows faster

$f(n) = c_1 n^2 + c_3 n + c_4$
$g(n) = c_5 n \log n + c_6 n + c_7$

n^2
$n\log n$

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
}
```

Best case
↳ n is odd
const time complexity

Average case
we don't know how the user will provide input, if we know before hand we can find order of growth for each one of them & then find the avg.

Worst case
upper bound on running time algo
→ In this case when n is even
O(n)

Assumption
odd & even are equally likely hence
$f(n) \to O(n/2)$

Big O Notation (Upper Bound or order of growth)

- ignore the lower order term
- ignore the constant terms
- $f(n) = O(g(n))$ then $f(n) \leq c\,g(n)$ $\forall$ $n \geq n_0$

Ex:- $f(n) \to 2n+3$
↳ O(n)

Here $g(n) \to n$

$f(n) \leq c\,g(n)$
$2n+3 \leq c\,(n)$.
$2n+3 \leq 3n$
$n \geq 3$. hence $n_0 \to 3$
$n \geq n_0$

c → constant of heighest growing term + 1
in this ex → 3

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

$\{n/4, 2n+3, n/100 + \log n, n+1000, n/1000 \dots \log n\}$ ↳ $O(n)$

- 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.
~~$\theta(1)$~~ C_1

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

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

Omega Notation

opposite of Big O notation → provides a lower bound

$f(n) = \Omega(g(n))$ → order of growth.

$0 \le cg(n) \le f(n)$

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

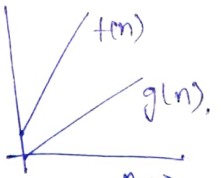
$1(n) \le 2n+3$

$n < 2n+3$

$n_0 = 0$

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

$n \ge n_0$



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

(1) $\{n/4, n/2, 2n, 3n \dots n^2, 2n^3, n!\} \to \in \Omega(n)$
as it bounds from lower side and order of growths greater than or equal to linear comes under $\Omega(n)$.

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

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

Theta Notation

Theta notation bounds a function from upper side and lower side → exact order of growth.

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

$C_1 g(n) \le f(n) \le C_2 g(n) \quad \forall \quad n \ge n_0$

$f(n) = 2n+3$

$\theta(n)$ $\quad C_1 g(n) \le f(n) \le C_2 g(n)$

$C_1 n \le 2n+3 \le C_2 n$

C_1: coefficient of leading term - 1 $(2-1) = 1$

C_2: coeff. of leading term + 1 $(2+1)$

$n \le 2n+3 \le 3n$

$n \ge 0$; $n \ge 3$ } max of these two

$n_0 = 3$

- if $f(n) = \theta(g(n))$
  then $f(n) = O(g(n))$ & $f(n) = \Omega g(n)$
  and $g(n) = O(f(n))$ & $g(n) = \Omega(f(n))$
- Theta notation is useful to represent time comp. when we know exact bound.

$\{n^2/4, n^2+n, 2n^2+\log n \dots\} \to \theta(n^2)$ highest order term must be n^2

Analysis of common loops

- `for (int i=0 ; i<n ; i= i+c){ -- work -- Θ(1) work }` → Θ(n/c) → Θ(n) (ignore const.)

  n = 10 → 10, 2, 4, 6, 8
  n = 11 → 0, 2, 4, 6, 8

- `for (int i=n ; i>0 ; i= i-c) { some Θ(1) work }`

  ex → n = 10 → 10, 8, 6, 4, 2 — 5 times
  n = 11 → 11, 9, 7, 5, 3, 1 — 6 times

  Θ(n)

  $\lceil n/c \rceil$ runs ceiling of n/c times

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

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

  order of growth → Θ(log n)

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

  0 → k-1 it runs k times

  $c^{k-1} < n$

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

  $k-1 < \log_c n$

  $k = \log_c n + 1$

- `for (int i=n; i>0 ; i= i/c){ —Θ(1) work— }` → Θ(log n) (Θ(log n))

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

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

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

  2 2^c $(2^c)^c$
  2 2^c 2^{c^2} ... $2^{c^{k-1}}$
  k times

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

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

  $k-1 < \log_c \log_2 n$

  $k < \log_c \log_2 n + 1$

- `fun (int n) {`
  `for (int i=0 ; i<n ; i++) { — Θ(1) work — }` → Θ(n)
  `for (int i=1 ; i<n ; i = i*2) { — Θ(1) work — }` → Θ(log n)
  `for (int i=1 ; i<100 ; i++) { — Θ(1) work — }` → Θ(1)

  Θ(n) + Θ(log n) + Θ(1) → Θ(n) (By ignoring lower order terms)

- `for (int i=0 ; i<n ; i++)` → Θ(n)
  `for (int j= 1 ; j<n ; j = j*2) {` → Θ(log n)
  `Θ(1) work.`
  `}`

  Θ(n) * Θ(log n) → Θ(n log n)

* Q Θ(n log n) + Θ(n^2) → Θ(n^2).

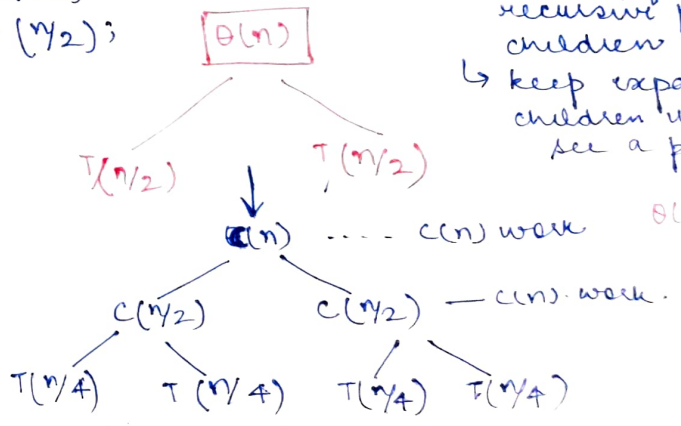
Analysis of Recursion

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

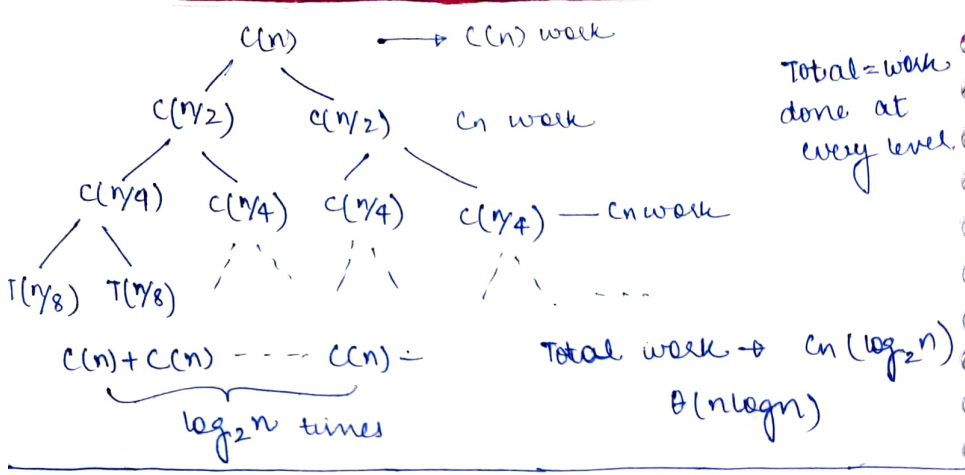
T(n) → 2T(n/2) + Θ(n)
T(1) → c

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



Θ(n) → (n

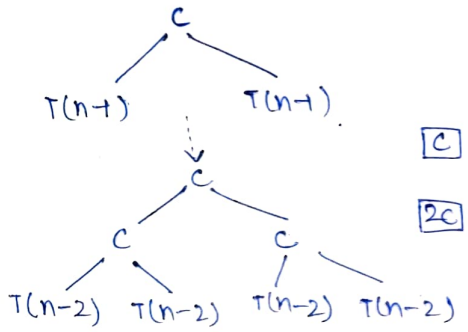


Total = work done at every level

Total work → $Cn(\log_2 n)$

$\theta(n \log n)$

$T(n) = 2T(n-1) + C$ $\quad T(1) \to C$



C

2C

$C + 2C + 4C \cdots$

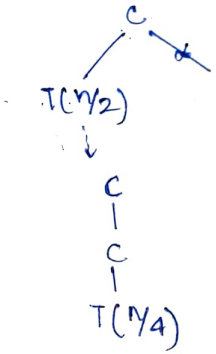
$C(2^0 + 2^1 + 2^2 \cdots 2^{n-1})$ — n terms.

$\to C\left(\frac{1(2^n - 1)}{2-1}\right)$

$\to C(2^n - 1)$

$= \theta(2^n)$

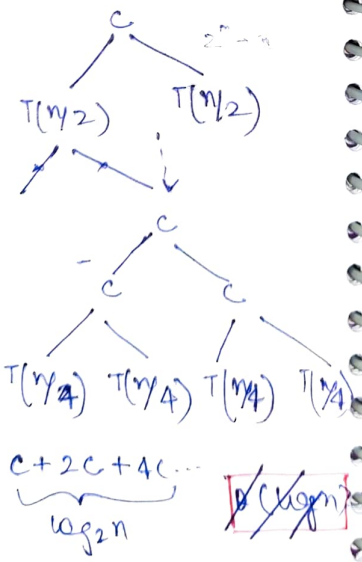
$T(n) = T(n/2) + C$ $\quad T(1) \to C$



$C + C + C \cdots$ ($\log_2 n$ terms)

$\theta(\log n)$

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



$C + 2C + 4C \cdots$ ($\log_2 n$ terms)

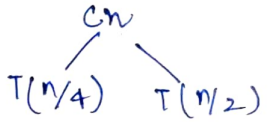
~~$\theta(\log n)$~~

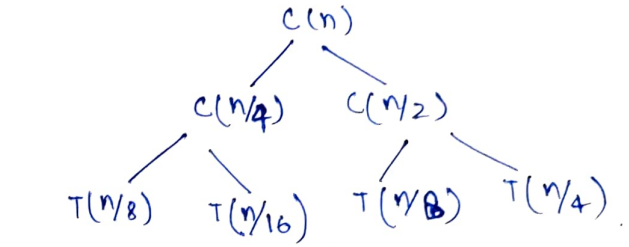
---

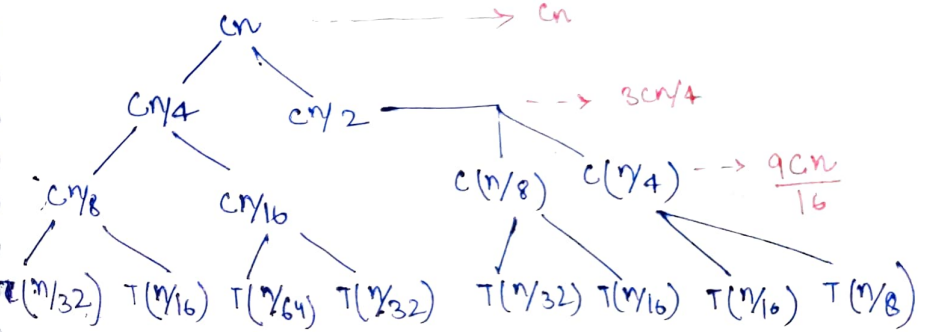
no of terms → $\log_2 n$

GP → $\frac{1(2^{\log_2 n} - 1)}{2-1} \to 2^{\log_2 n} \to \theta(n)$

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







$Cn + \frac{3Cn}{4} + \frac{9Cn}{16} + \cdots$

we don't know the no of terms

As the left most branch i.e reducing by 1/4 will terminate early in comparison to right-most branch.

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

i.e height - $\log n$

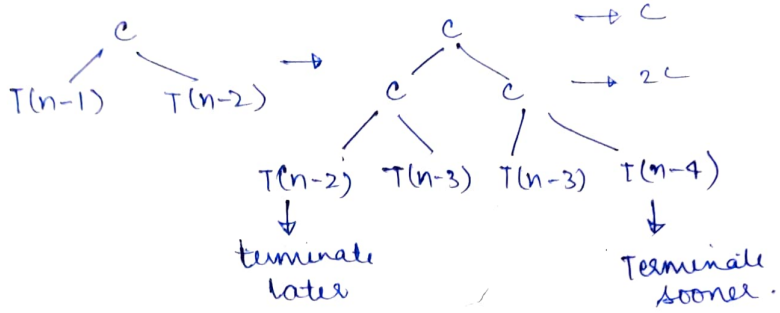
$Cn\left[\frac{1}{1 - 3/4}\right] \to Cn(4) \to O(n)$

$r < 1$ infinite series

↳ big O notation upper bound

Febonacci series — $T(n) \to T(n-1) + T(n-2) + C$

$T(n) \to T(n-1) + T(n-2) + C$



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

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

Space complexity → Memory taken

order of growth of memory taken

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

```
function (int n) {
    for (int i → n) {
        sum += i;
    }
}
```
→ O(1)

③
```
function ( int arr[], int n) {
    int sum = 0;
    for (int i = 0; i<n ; i++){
        sum = sum + arr [i];
    }
    return sum
}
```
→ array size depends on n

→ Θ(n)

Auxiliary space

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

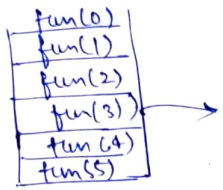
③ auxiliary space → Θ(1)

```
int fun (int n) {
    if (n ≤ 0)
        return 0
    return n + f (n-1)
}
```

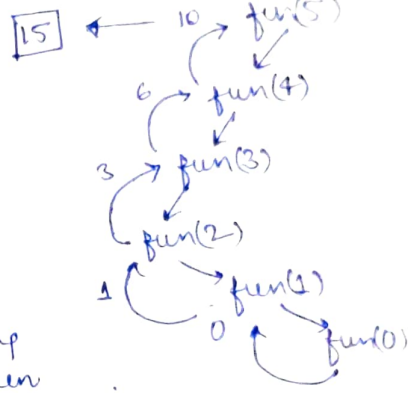
Space complexity

Recursion calls are stored in stack.

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



→ Θ(n)

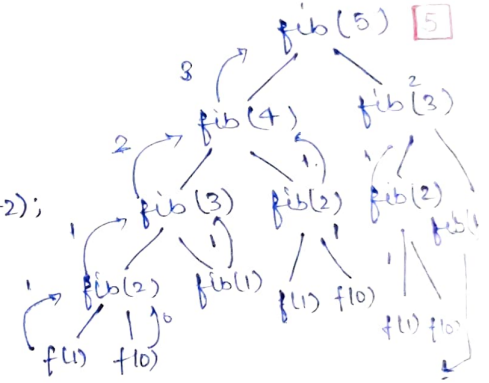


Febonacci series

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

0 1 1 2 3 5

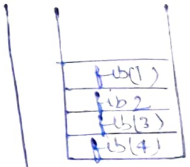
n=0 n=1 n=2 n=3 n=4 n=5



Auxiliary space → maximum length of leaf to root path → Θ(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];
```



→ Auxiliary space. Θ(n)

Space comp. Θ(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
}
```

space complexity → θ(1)

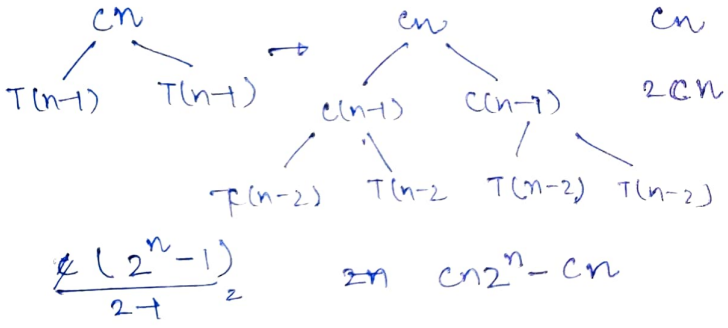
space comp (Auxiliary) θ(1)

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

$n \log 2$ → $\theta(n)$

$\log n \log n$ → $\theta(\log n)^2$

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



cn, 2cn, ...

$\frac{c(2^n-1)}{2-1}$ 2^n $cn2^n - cn$

Basic Mathematics

• Finding the no of digits

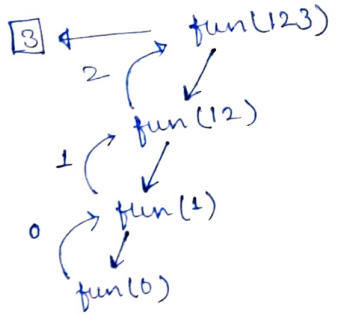
① iterative solution

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

② Recursive solution

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

Ex- 123



③ logarithmic solution

```
int countdigit (log n).{
    return floor(log10 n + 1)
}
```

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

• Arithmetic and Geometric Progression

AP → 2, 4, 6, 8 ---

avg → $\frac{\text{sum}}{n}$

sum → avg × n

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

GP → 2, 4, 8, 16 - - -

sum → $\frac{a(1-r^n)}{1-r}$

quadratic equations

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

• Mean and Median

mean → $\frac{\text{adding all the entries}}{\text{no of entries}}$

median → ordering all the numbers and picking the middle one

1, 1, 7 → 1, 4, 7 → [4]

for even entries we take mean of the middle terms

1, 4, 2, 8 → 1, 2, 4, 8 → [3]

Prime Numbers

- 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

```
30 ) 42 ( 1
     30
     --
     12 ) 30 ( 2
          24
          --
           6 ) 12 ( 2
               12
               --
               0
```

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

- LCM × HCF → a × b
- For fraction HCF → HCF(num) / LCM(den)

  LCM → LCM(num) / HCF(den)
- No of trailing zeroes in prime factorization of n.

  $5!$ → floor(5/5) + floor(5/25)... 1 trailing zero

  $n!$ → floor(n/5) + floor(n/25)....

Bitwise operators in C++

AND - &
OR - |
XOR - ^

- Operate on binary representation of numbers

  x = 3, y = 4

  x & y ⇒ 011 & 100 ⇒ 000 (0)

  x | y ⇒ 011 | 100 ⇒ 111 (7)

  x ^ y ⇒ 011 ^ 100 ⇒ 111 (7)

  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

Binary representation of 6 → 0...00110
might be 29 or 61 or 13 depending on the compiler. If compiler uses 32 bits for int then trailing zeroes will be 29.

Left shift Operator

a << b

a → no whose binary rep. is to be shifted

b → how many times it needs to be shifted

x = 3 : 0000... 011

(6) x << 1 : 000....0110 (shifted left by 1)

(12) x << 2 : 000....01100

x << y ⇒ x * 2^y (assuming that initial y bits in binary representation of x is less than or equal of to y.

Right Shift Operator

a >> b

last 'b' bits are ignored and 'b' 0 bits are added in the beginning

x = 4 → 0...00100

x >> 1 → 00...010

x >> 4 → 00....000

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

33 >> 1 → 16

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

Bitwise Not (~)

$x = 1$ → 00...001
$\sim x$ → 11...110
$x = 5$ → 00...101
$\sim x$ → 11...010

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

2's bit representation of a number x in n bits is $2^n - x$ (negative number)

$x \rightarrow -2$
$2^3 - 2 \rightarrow 6 = 110$

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

$x = 1 \rightarrow 000...01 \rightarrow 111...10 \rightarrow 2^{32} - 1 - 1$ (for all 1's, -1)
$= 2^{32} - 2$

$\sim x$.

Comparing
$2^{32} - 2 \Rightarrow 2^n - x$
$\sim x \rightarrow -2$

$x = 5 \rightarrow 000.0101 \Rightarrow 111...1010$
$\sim x \rightarrow -6$
$2^{32} - 1 - 5 \rightarrow 2^{32} - 6$

Check if k^{th} bit is set

I/P → n = 5, k = 1 → 000...0101 (k) — Yes

I/P → n = 8, k = 2 → 000...1000 (k) — No

I/P → n = 0, k = 3 → 000...000 (k) — No

$k \leq$ no of bits in binary

Algorithm

We are trying to make a number with only k^{th} bit set.

1 << (k-1) → k = 3

000...001 << 2
000...100
(only 3rd bit set)

and then doing an AND operation between them.

⑤ → 000...0101
1<<(k-1) → 000...0100
000...0100 ≠ 0
Yes

Second Method

Shift the k^{th} bit to the last position and then do a bit wise and with 1.
To do that we set the k^{th} bit to 1 by doing a right shift operation by (k-1)

n >> (k-1) → k^{th} bit is shifted to 1^{st} bit and then do a bitwise AND with 1.

Count Set Bits

I/P → 5 I/P = 13
O/P → 2 O/P = 3

2nd Approach

Brian Kernigam's Algo.

n = 40 ⇒ 000...0101000
after 1st Iteration ⇒ 000...0100000
after 2nd iteration ⇒ 000...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 101000
n = 39 100111
n & (n-1) 100000

1st Approach

- check the last bit (LB)
  if (LB = 1) count++
- ~~right~~ left shift by 1.

⇒ while(n > 0){ or → n & 1
  if ((n % 2) != 0){
    count++;
  }
  // Remove last bit
  n = n/2 or n = n >> 1

can be rewritten as
count = count + (n & 1);

Time complexity → θ(Total Bit in n)

cont...

n = 32	1 0 0 0 0 0
n = 31	0 1 1 1 1 1
n & (n-1)	0 0 0 0 0 0

It require less time than previous one
O (no of set bits)

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

Look up Table Method for 32 bit number

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

table[0] → 0
table[1] → 1 (0 1)
table[2] → 1 (1 0)
table[3] → 2 (1 1)
table[255] → 7 (1 1 1 1 1 1 1)

```
table[0] = 0
for (int i=0; i<255; i++) {
    table[i] = (i & 1) + table[i/2];
}
```

```
int count(int n) {
    res = res + table[n & 0xff];
    n >> 8 (shift the number by 8 bits)
    res += table[n & 0xff]
    n >> 8
    res += table[n & 0xff]
    n >> 8
    res += table[n & 0xff]
    return res;
}
```

O(1) solution

table[n & 0xff]
↓
bitwise and with '8' set bits
0xff → 11111111
n → 72 → 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
  ~~n = even (power of 2)~~
  ~~n = odd (Not a power of 2)~~
  ① n = 1 (Yes)
  n = odd (NO)

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

② Brain's Kernigam 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 2) {
    if(n == 0)
        return false
    return ((n & (n-1)) == 0);
}
```

⇓

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

Example:

n = 4	0 1 0 0
n = 3	0 0 1 1
	0 0 0 0 ✓
n = 6	0 1 1 0
n = 5	0 1 0 1
	0 1 0 0 ✗

Find only Odd occuring number

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

- only 1 odd occuring no. must be present

- using XOR operator

x ∧ 0 = x
x ∧ x = 0
x ∧ y = y ∧ x

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

res = 0
arr[0] = 4
res ∧ arr[0] → 0 ∧ 4 = 4
arr[1] → 3
3 ∧ 4 →
arr[2] = 4
3 ∧ 4 ∧ 4 = 3
3 ∧ 4 →
3 ∧ 4 ∧ 4 → 3
3 ∧ 5 →
3 ∧ 5 ∧ 5 → 3
res = 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

- Do XOR of all the numbers and then take the XOR of result with all the numbers from 1 to n+1

```
for (int i=0 ; i<n; i++){
    res = res ^ arr[i];
}
for (int i= ; i<=n+1; i++){
    res = res ^ i
}
return res;
```

Variation 1
number given in range [n, m].

Variation 2
finding 2 odd occuring number

Finding two odd occuring number

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

```
3 → 011
5 → 101
  → 110
```

- 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) {2, 2, 3, 3, 3}
group 2 (0) {4, 4, 5, 5, 5}

- It is ensured that both odd numbers are in different group. Now we can saparately 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 →      ...0 0 1 0 1 0 0 0
n-1 = 39 →    ...0 0 1 0 0 1 1 1     last set bit
~(n-1) →      ...1 1 0 1 1 0 0 0     is a position 4
n & ~(n-1) →  ...0 0 0 0 1 0 0 0
```

- To put element in different groups
  we make a AND with the number and setbit and
  if ans is 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 j=0; j<n; j++){
            if(counter & (1<<j)){
                print(str[j]);
            }
        print('\n');
    }
}
```

Time complexity → $\theta(2^n * n)$

I/P → ab

O/P → '', a, b, ab

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

Random case
counter = 3 → 011

① iteration
3 & (1<<0) → 3&1
→ 011 & 001 → 1≠0
print(a)

② 3 & (1<<1) → 3
011 & 010 → 010≠0
print b

③ 3 & (1<<2)
011 & 100 = 0

Recursion

- function calls itself
- Direct recursion

```
void fun1(){
    ---
    fun1();
    ---
}
```

Indirect recursion

```
void fun1(){
    ---
    fun2();
    ---
}
fun2(){
    ---
    fun1();
}
```

- without terminating condition

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

} Segmentation fault

Terminating condition

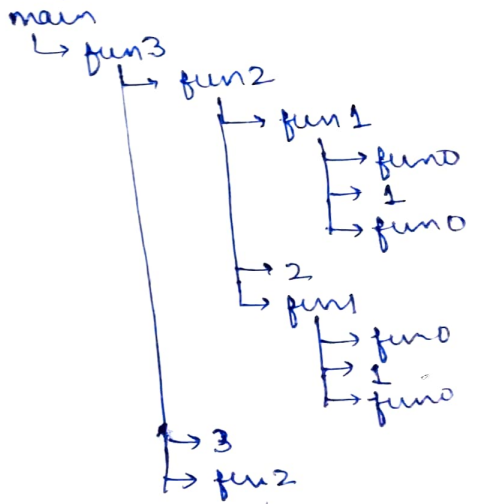
```
fun1(int n){
    if(n==0){
        return;
    }
    cout<<"ggg";
    fun1(n-1);
}
void main(){
    fun1(5);
}
```

} Base case (Terminating case)

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);      1
    cout<<n;       2
    fun(n-1);      1
}                  3
main(){            1
    fun(3);        2
}                  1
```



Q1

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

fun(16) → fun(8) + 1 → 1 + fun(4) → 1 + f(2) → 1+f(1) ④

fun(16) → 4

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

fun(20) ④
→ fun(10) + 1
→ fun(5) + 1
→ fun(2) + 1
→ fun(1) + 1
0

It returns $\log_2 n$

$\log_2(20) \to 4$

- In general it returns $\log_2(n)$

Generally

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

→ $\lfloor \log_m(n) \rfloor$

Ex→

```
int fun(int n){
    if (n < 3){
        return 0;
    }
    else return 1 + fun(n/3)
```

→ $\lfloor \log_3(n) \rfloor$

Q

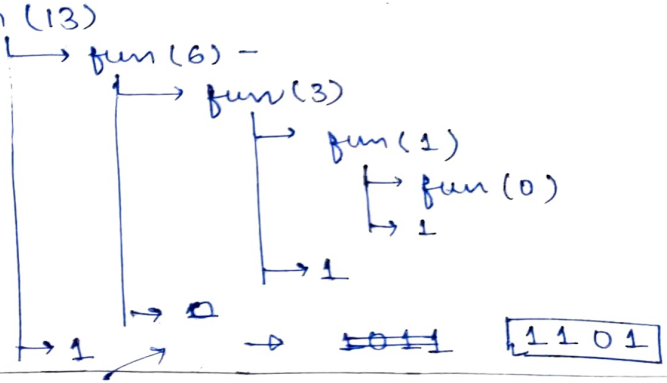
```
void fun(int n) {
    if (n == 0)
        return;
    fun(n/2);
    print(n % 2);
}
```

fun(7)
→ fun(3)
→ fun 1
→ fun 0

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 0;
    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
→ f(1)
→ 1

⑦ ⑥ ⑤ ④ ③ ② ①

θ(1)

Tail Recursion

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

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

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

tail Recursion is faster because the callee is not needed to save the state after execution

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

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

The changes are called → **Tail call Elimination.**

changing of normal function → Tail Recursion

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

→ k+1 to start printing nos from 1

Writing Base cases

(a) Factorial n when n ≥ 0 — 0! → 1, 1! = 1

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

(b) n^{th} fibonacci Number

0 1 1 2 3 5 8
0 1 2 3 4 5 6

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

Sum of n natural Numbers

```
fun(int n) {
    return n + fun(n-1);
}
```

Base case included

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

$\frac{5 \times 6}{2} = 15$

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

Check if String is Palindrome

I/P : abbcbba
O/P : Yes

We always try to solve for smaller cases

So $S_1\ S_2 - - - S_{n-2}\ S_{n-1}\ S_n$

check if inner string is palindrom

check if $S_0 = S_n$.

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

Base cases

① even length string

abba $n = 0$

② odd length

abcba $n = 1$

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

str = "racbabcaa"

racbabca

cbabc

bab

a

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

abaaba

baab

s	e
0	5
1	4
2	3

3 ≥ 2 ⇐ true

Time complexity → $\theta(n)$

Auxiliary space → $\theta(n)$

Sum of Digits in a number

I/P → 253

O/P → 10

n = 4537

sum(4537)

↳ 7 + sum(453)

↳ 3 + sum(45)

↳ 5 + sum(4)

↳ 4 + sum(0).

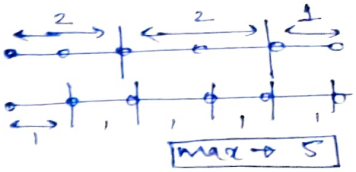
```
int sum (int n){
    if (n == 0){
        return 0;
    }
    return (n%10) + sum(n/10);
}
```

19

Q// 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 → n = 5 a = 2, b = 5, c = 1

O/P → 5



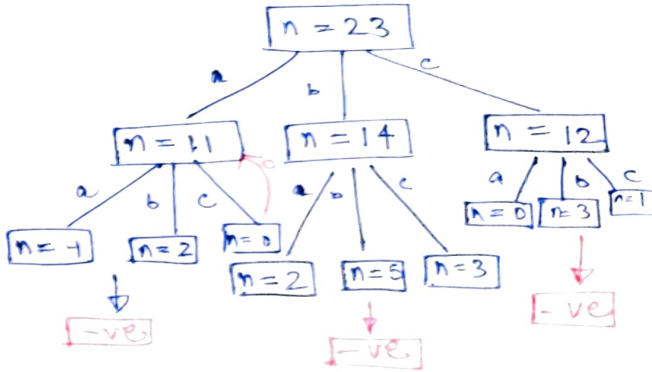
I/P → n = 5 a = 4, b = 2, c = 6

O/P → -1

I/P → n = 23 a = 12, b = 9, c = 11

O/P → 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 cases
    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))
```

Returns max of three values

```
    if (res == -1) return -1;
    return res + 1;
}
```

Q// Print all substrings

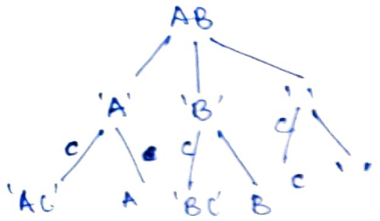
I/P → str → "ABC"

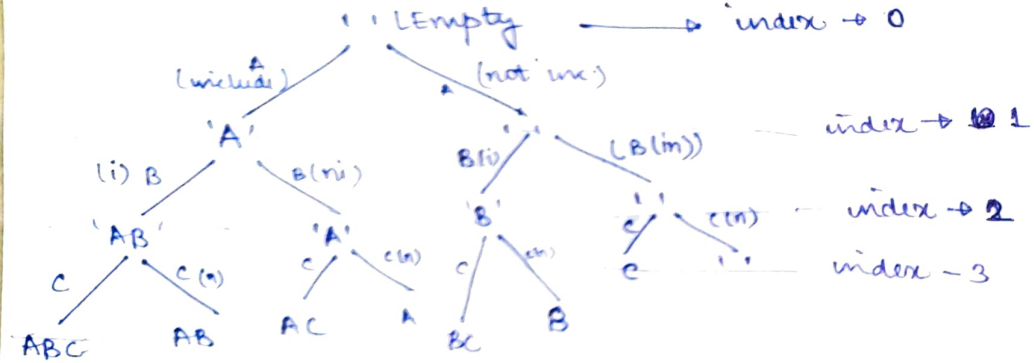
I/P → "", "A", "B", "C", "AB", "BC", "ABC".

If I 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 pt (string s, int curr=~~

```
void PrintSubStr(String str, String curr="", index=0){
    if(index == str.length){
        print(curr); return;
    }
    return PrintSubStr(str, curr.append(str[i]), index+1);
    return PrintSubStr(str, curr, index+1);
}
```

Base case

char is included / character is not inc.

Tower of Hanoi

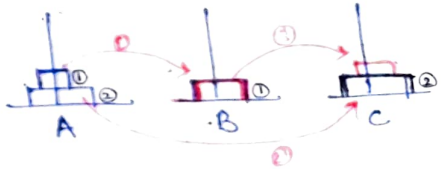
* move disc from A → C
* only 1 disc move at time
* largest is at bottom and smallest at top and only top disc can be shifted

I/P → n = 1

O/P → move disc 1 from A to C (direct)

I/P → n = 2

O/P → ① from A → B
② from A → C
③ from B → C.



Approach

- shift nth bar from A → B and then shift (n−1)th bar from A → C and then nth to B → C

~~shift~~ TOH(n, A, B, C)

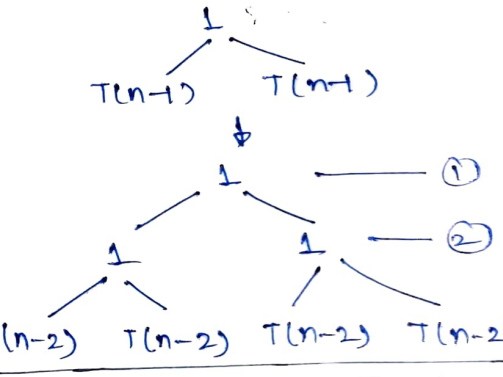
- shift (n−1) from A → B
  shift nth from A → C
  shift (n−1) from B → C

TOH(n−1, A, C, B)
~~TOH~~ move n from A → C
TOH(n−1, B, A, C)

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

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



$1, 2, 4, \dots 2^{n-1}$.

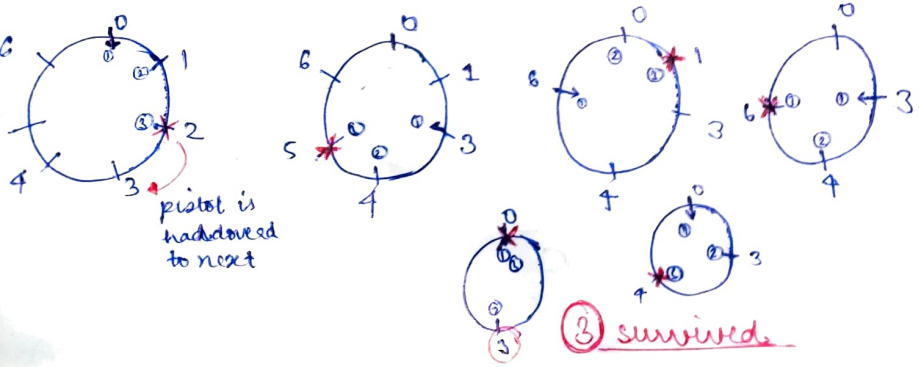
$$\frac{1 \times (2^n - 1)}{2 - 1}$$

$= 2^n - 1$ → no of movement

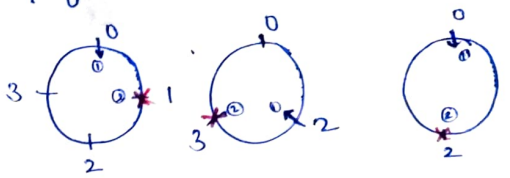
Josephus Problem

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

Ex:- $n = 7$ $k = 3$



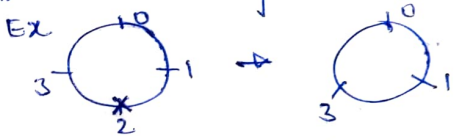
I/P : $n = 4$, $k = 2$.
O/P : 0




```
int jos(int n, int k){
    if (n == 1)
        return 0;
    return (n-1, k
```

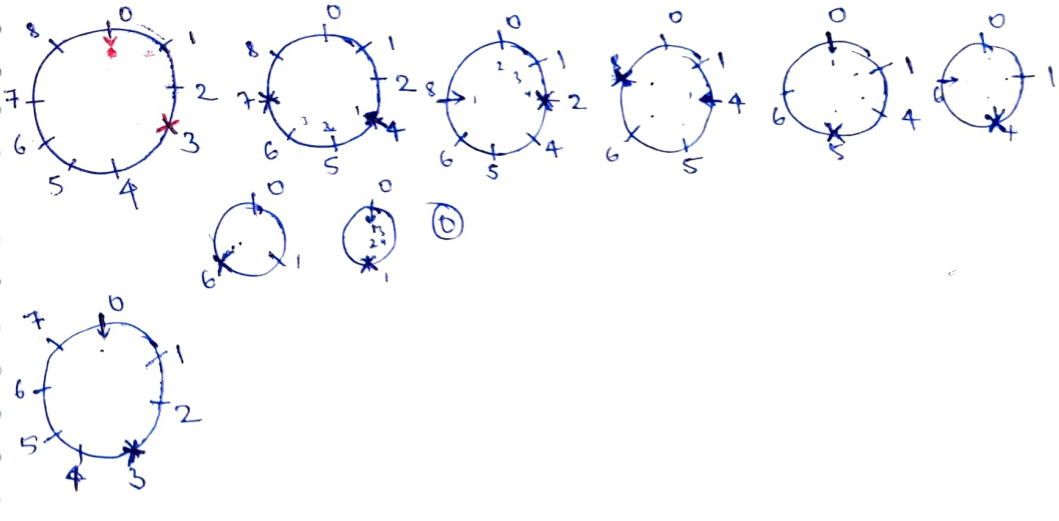
* we are using numbers as per previous case.

Ex



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

$n = 9$ $k = 4$



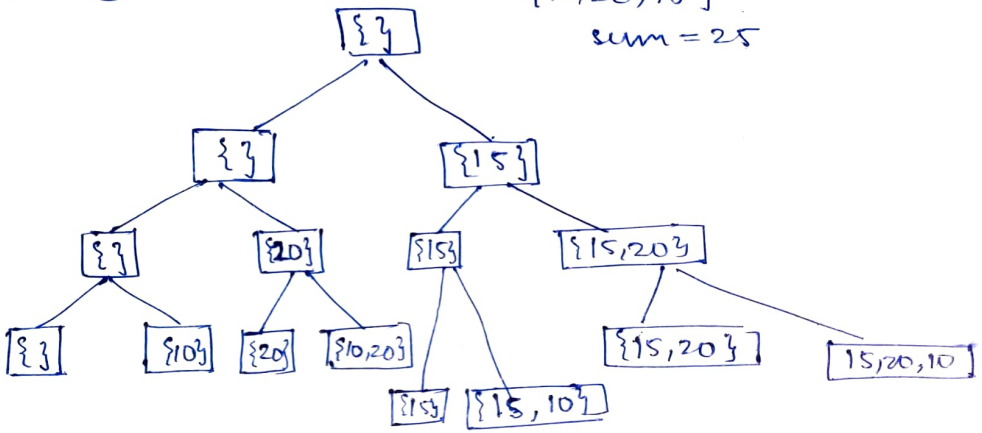
Subset sum Problem

I/P = {10, 5, 2, 3, 6}
sum → 8
O/P → (2)

I/P : {10, 20, 15}
sum = 0
O/P = 1

I/P : {1, 2, 3} sum → 4
O/P : (1)

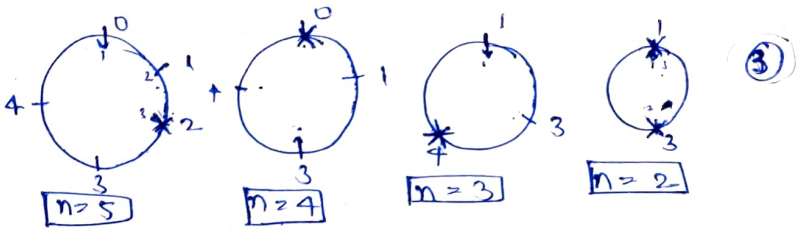
{10, 20, 15}
sum = 25



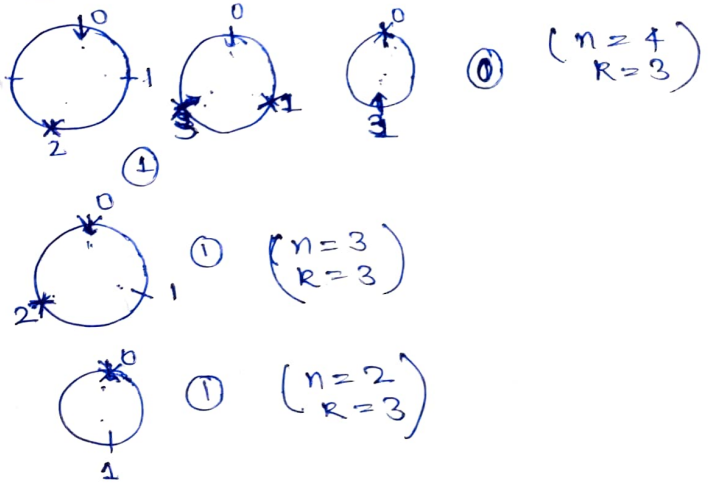

```
int check (int arr, int sub, index/sum) {
    if (index == arr.length()) {
        if (sum of el of sub == sum) {
            return 1;
    }
}
return check (int arr, int sub.push(arr[index]), sum);
check (int arr, int sub, index+1, sum);
```

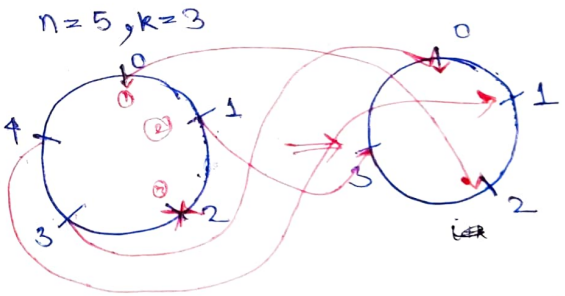
Contd Recursion (Josephus)

n = 5 k = 3



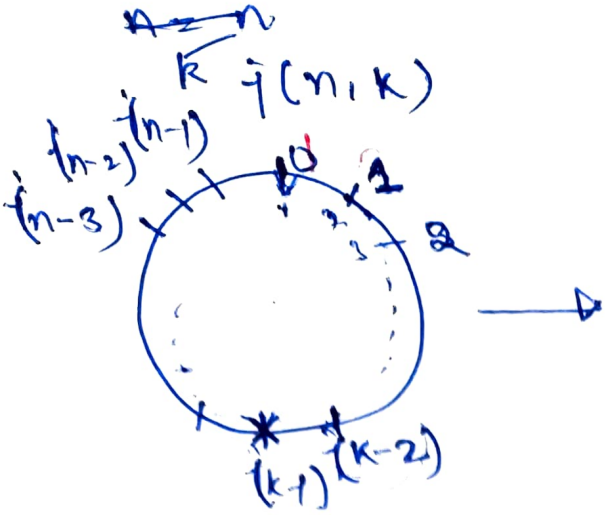
In manual we want
jos(4,3)
→ 0.
But by code. jos(4,3) will return ~~wat~~ what jos(3,3) will return

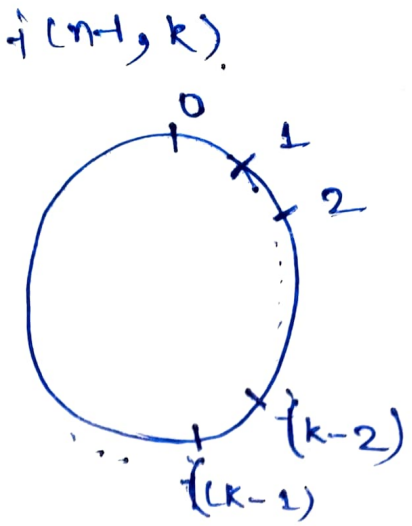


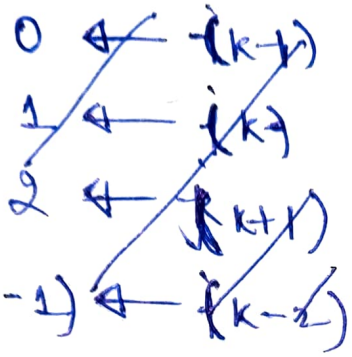


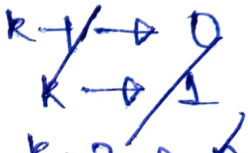
n = 5, k = 3	n = 4, k = 3
0	2
1	3
4	1
3	0

If j(4,3) will return 0, it means main func. will return (3).









$n = 5, k = 3$

$1 \to 3$

$(+k)$

$= (2+3) \% 5$

$5 \% 5$

$= 1$

$(i+k) \% n$

$i \quad n-1+k$

$(n+k-1) \% n$

$k-1$

$k \leftarrow 0$

$k+1 \leftarrow 1$

$k+2 \leftarrow 2$

$k+i \leftarrow i$

$k-2 \leftarrow n-1$

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 $T(n) \to T(n-1) + C$

$\hookrightarrow \theta(n)$.