Applications Of Recursion :-

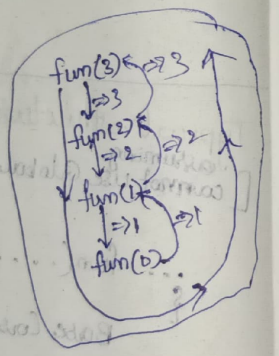
(1) Many algorithm techniques are based on Recursion
- Dynamic programming
- Backtracking
- Divide and conquer
  [Binary search, Quick Sort and Merge Sort]

(2) Many problems inherently recursive
- Tower of Hanoi
- DFS based traversals
  [DFS of Graph and Inorder/preorder/postorder traversal of tree]

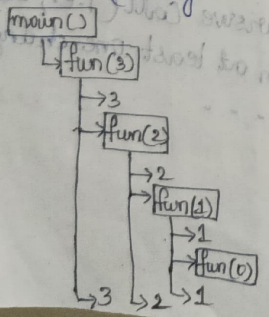
Recursion practice

(1)

```
class Test
{
  Static Void fun (int n)
  {
    if (n==0)
      return;
    System.out.println(n);
    fun(n-1);
    System.out.println(n);
  }
  public static Void main (String[] args)
  {
    fun(3);
  }
}
```



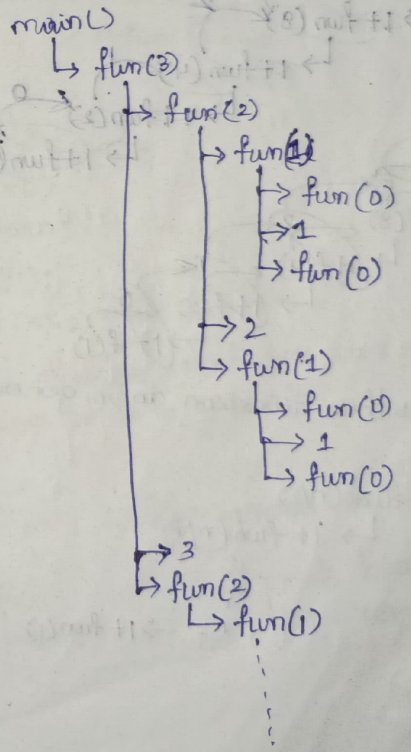
O/P: 3 2 1 1 2 3



(2)

```
class Test {
  Static Void fun (int n)
  {
    if (n==0)
      return;
    fun(n-1);
    System.out.println(n);
    fun(n-1);
  }
  public static Void main (String[] args)
  {
    fun(3);
  }
}
```

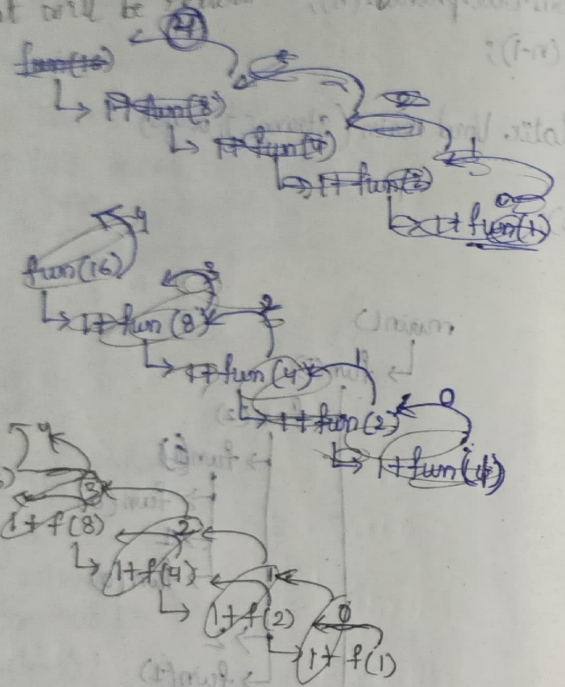
O/P: 1 2 1 3 1 2 1



③

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

What will be return value of fun(16)? 4



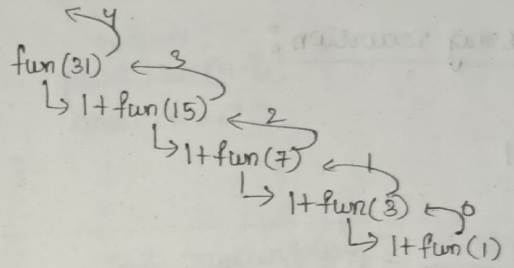
What does this function do in general for n≥1

fun(n)
↳ 1+fun(n/2)
↳ 1+fun(n/4)
⋮
↳ 1+fun(1)

fun(1) = 0
fun(2) = 1+fun(1) = 1
fun(4) = 1+fun(2) = 1+1 = 2
fun(8) = 1+fun(4) = 1+2 = 3
fun(16) = 1+fun(8) = 1+3 = 4

$\lfloor \log_2 n \rfloor$

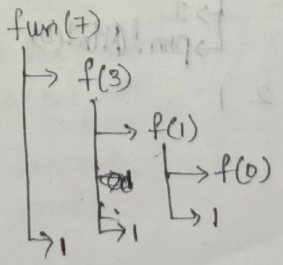
fun(18) = 1+fun(9)
= 1+(1+fun(4))
fun(20) = 1+fun(10)
= 1+(1+fun(5))
= 1+(1+(1+fun(2)))
= 4



*

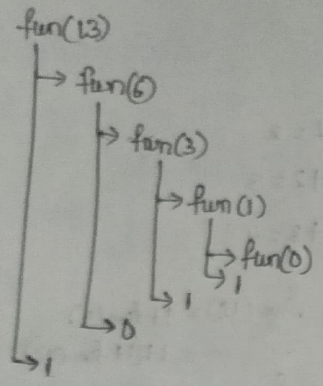
```
void fun(int n)
{
  if(n==0)
     return;
  fun(n/2);
  print(n%2);
}
```

What does this function print for n=7?
And what does it do in general?



fun(7) prints 111

It prints binary equivalent of n where n>0.



13 ~~decimal~~ binary equivalent is 1101.

print n to 1 using recursion:

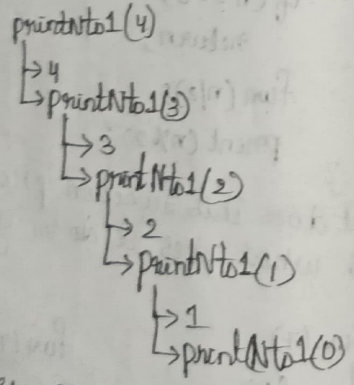
I/p: n=5

O/p: 5 4 3 2 1

I/p: n=2

O/p: 2 1 (n≥1)

```
void printNto1(int n)
{ if (n==0)
    return;
  cout<<n<<" ";
  printNto1(n-1);
}
int main()
{ int n=4;
  printNto1(n);
  return 0;
}
```



O/p: 4 3 2 1

T.C = $\theta(n)$

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

A.S = ~~$\theta(n)$~~ $O(n)$

print 1 to N using recursion:

I/p: n=4

O/p: 1 2 3 4

T.C = $\theta(n)$

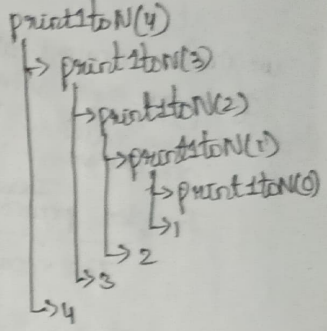
worst A.S = ~~$\theta(n)$~~ $O(n)$

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

I/p: n=5

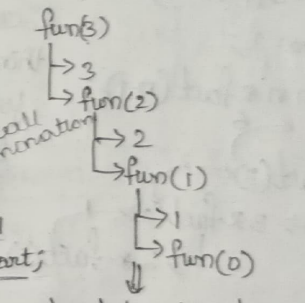
O/p: 1 2 3 4 5

```
void print1toN(int n)
{ if (n==0)
    return;
  print1toN(n-1);
  cout<<n<<" ";
}
```



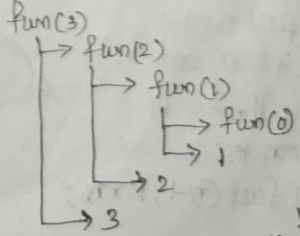
Tail recursion: ⇒ last thing happens in function is recursive calls

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



This takes less time as modern compilers because it is tail recursive

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



Not a tail recursive

Not Tail Recursive ⟹ equivalent Tail Recursive

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

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

We can't convert all Non Tail recursive to Tail recursive

Eg: merge sort & Quick sort (faster)
(Non tail recursive) (Tail recursive)

Traversals: Inorder, preorder (Tail recursive), postorder (Non tail recursive)

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

fact(3) → 6, fact(2) → 2, 3*fact(2) → 6, 2*fact(1) → 2

NOT Tail Recursive

Is this Tail recursive?
⇒ Not tail recursive, bcs, Recursive call is not the last thing that happens in this function.
We need know
i) fact(n-1)
ii) To store result we have to multiply
parent call can't finish after the child call

Tail Recursive:

```
// Initially pass k=1.
int fact(int n, int k)
{
  if(n==0||n==1)
    return k;
  return fact(n-1, k*n);
}
```

fact(3,1) → fact(2,3) → fact(1,1*3*2) → 6

Writing Base Cases in Recursion:

factorial n where n ≥ 0

I/p: n=4, O/p: 24
I/p: n=0, O/p: 1

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

fact(2) → fact(1) → fact(0)
fact(3) → fact(2) → fact(1) → fact(0)

⇒
```
int fact(int n)
{
  if(n==1) ⇒ // wrong Base case.
    return 1;
  return n*fact(n-1);
}
```

fact(0) → fact(-1) → fact(-2) → ... Segmentation fault

nth fibonacci numbers where n ≥ 0:

0, 1, 1, 2, 3, 5, 8, 13, 21 . . . .

I/p: n=3, O/p: 2
I/p: n=0, O/p: 1
I/p: n=1, O/p: 1

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

(if(n<=1) return n;)

fib(3) → fib(2) → fib(1), fib(0); fib(1)
fib(2) → fib(1), fib(0)

```
int fib(int n)
{
  if(n==0)  // wrong base case
    return 0;
  return fibo(n-1) + fibo(n-2);
}
```

fib(1)
↳ fib(0)
↳ fib(-1)
↳ fib(-2)
↳ fib(-3)
Segmentation fault

Sum of Natural Number Using Recursion:

I/p: n=2 O/p: 3

I/p: n=4 O/p: 10

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

getSum(4) ← 10
↳ 4+getSum(3) ← 6
↳ 3+getSum(2) ← 3
↳ 2+getSum(1) ← 1
↳ 1+getSum(0)

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

T.C = $\theta(n)$

AS = $O(n)$

getSum(4) ← 10
↳ 4+getSum(3) ← 6
↳ 3+getSum(2) ← 3
↳ 2+getSum(1) ← 1
↳ 1+getSum(0) ← 0

Palindrome check using recursion:

I/p: abbcbba O/p: yes

I/p: abba O/p: yes

I/p: geeks O/p: NO

s0 s1 s2 ----- sn-3 sn-2 sn-1

Base cases:

n=0 OR n=1: Return true

for even length; for odd length

// Initially start=0, end=n-1

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

short circuiting

ispalin("abcbca")
↳ ispalin("cbc")
↳ ispalin("b")

ispalin("abba")
↳ ispalin("bb")
↳ ispalin("")

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

T.C = $O(n)$

AS = $O(n)$ ⇒ n/2 space call

Sum of digits using Recursion:

I/p: n = 253 (n≥0)

o/p: 10

I/p: n = 9987

o/p: 33

I/p: n = 10

o/p: 1

getSum(253) → 10
↳ getSum(25)+3 → 7
↳ getSum(2)+5 → 2
↳ getSum(0)+2 → 0

n%10 ⇒ Gives last digit of n

n/10 ⇒ Removes the last digit of n.

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

T.C = Θ(d)
A.S = Θ(d)

getSum(873) → 18
↳ getSum(87)+3 → 15
↳ getSum(8)+7 → 10
↳ getSum(0)+8 → 8 ⇒ if (n<=9) return n;

Iterative sol

```
int getSum(int n)
{ int res = 10;
  while (n>0)
  { res = res + n%10;
    n = n/10;
  } return res;
}
```

T.C = Θ(d)
A.S = Θ(1)

Rope cutting problem: (max pieces)

Length of every piece (after cuts) should be in set {a,b,c}
0 < a,b,c ≤ n

I/p: n=5, a=2, b=5, c=1

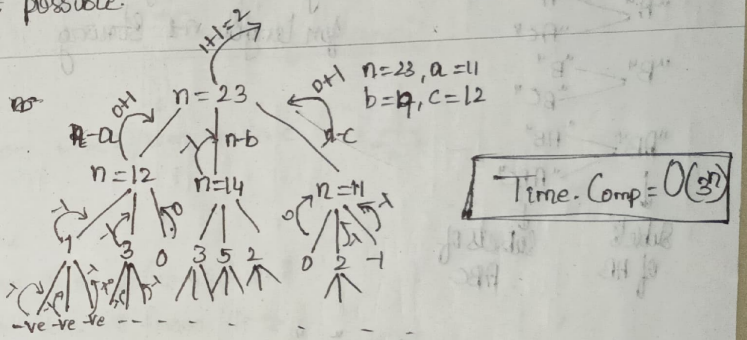
o/p: 5
we make 5 pieces of length 1 each

I/p: n=23, a=12, b=9, c=11

o/p: 2
we make 2 pieces of length 11 and 12

I/p: n=5, a=4, b=2, c=6

o/p: -1
not possible.



n=23, a=11, b=9, c=12

Time. Comp = O(3^n)

Idea for the Recursive solution.

```
⇒ int maxPieces(int n, int a, int b, int c)
{ if (n==0) return 0;
  if (n<0) return -1;
  int res = max(maxPieces(n-a,a,b,c),
                maxPieces(n-b,a,b,c),
                maxPieces(n-c,a,b,c))
  if (res == -1)
      return -1;
  return res+1;
}
// Pseudo Code.
```

Interesting Corner Case:
n=9, a=2, b=2, c=2

Generate Subsets:

All characters in the I/P string are distinct

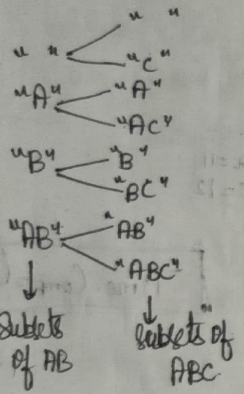
I/P: S = "AB"

O/P: "" "A" "B" "AB" (subsequences)

I/P: S = "ABC"

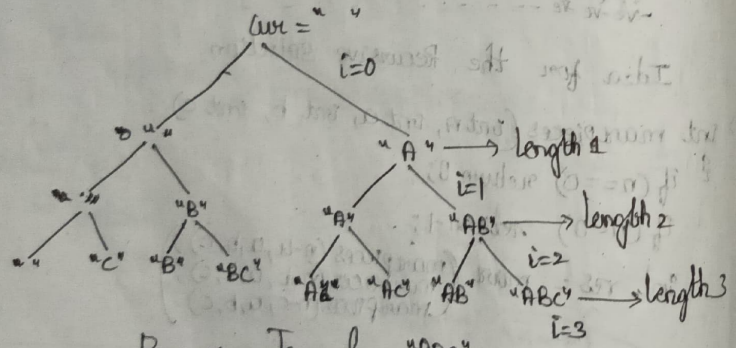
O/P: "" "A" "B" "C" "AB" "AC" "BC" "ABC"

for a string of length n, there are going to be 2^n subsets



⇒ we can generate all subsets for length n string using subsets for length n-1 string

⇒



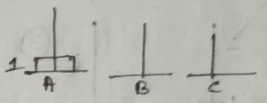
Recursion Tree for "ABC"

```
Void Subsets (String s, String cur = "", int i=0)
{
  if (i == s.length())
  {
    print(s);
    return;
  }
  Subsets(s, cur, i+1);
  Subsets(s, cur+s[i], i+1);
}
// pseudo code
```

Tower of Hanoi

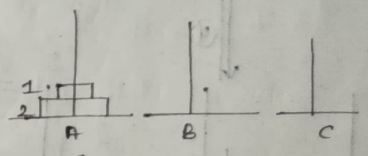
I/P: n = 1

O/P: move Disc 1 from A to C



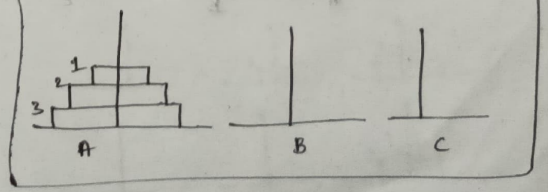
I/P: n = 2

O/P move Disc 1 from A to B
move Disc 2 from A to C
move Disc 1 from B to C



Rules: 1. Only one Disc moves at a time.
2. NO larger Disc above Smaller.
3. Only the top Disc of a tower can be moved.

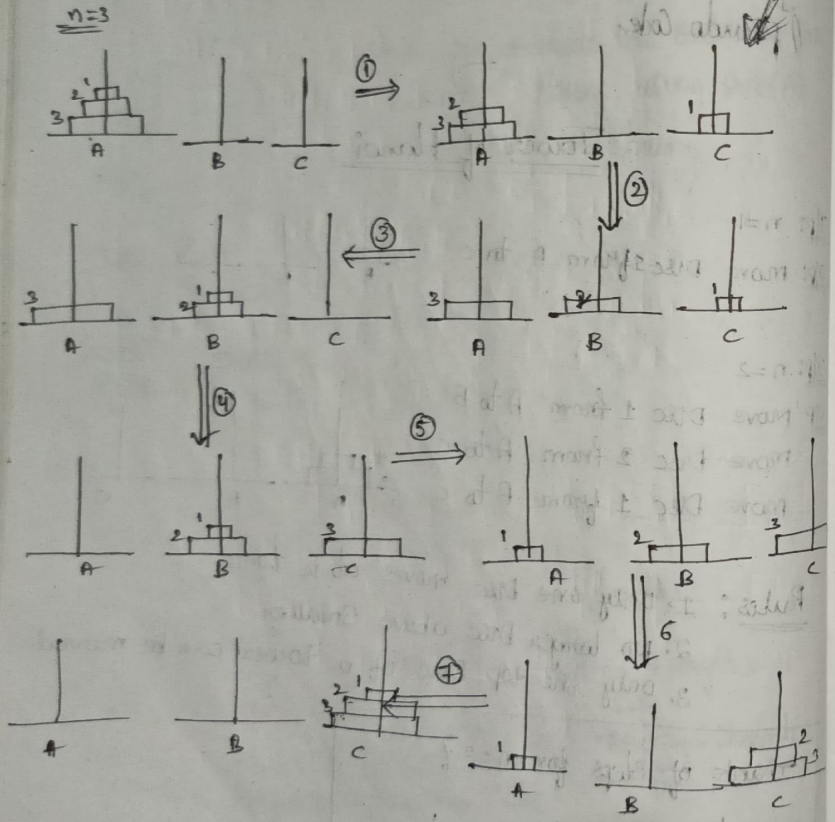
think of steps for n = 3 ?



I/p: n=3

O/p: move Disc 1 from A to C
2 from A to B
1 from C to B
3 from A to C
1 from B to A
2 from B to C
1 from A to C

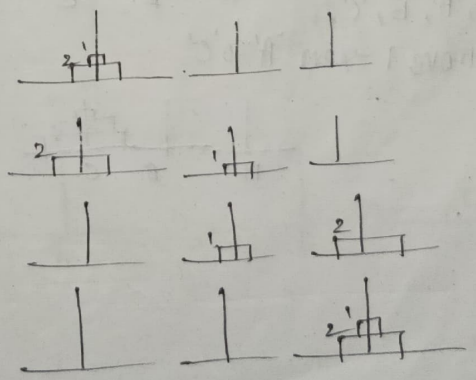
n=3



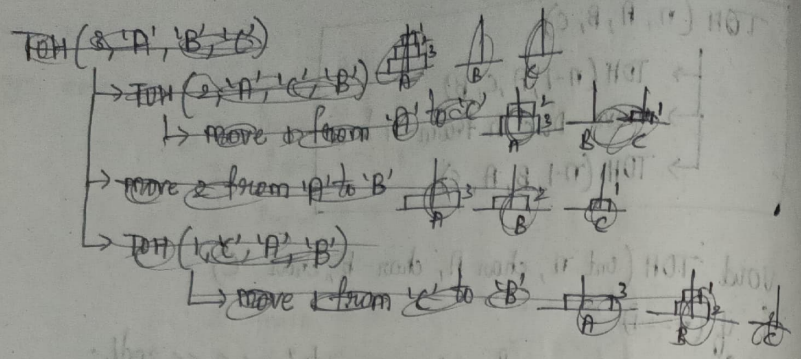

```
TOH (n, A, B, C)
    ↳ TOH (n-1, A, C, B)
    ↳ move Disc n from A to C
    ↳ TOH (n-1, B, A, C)
```

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

```
TOH (2, 'A', 'B', 'C')
    ↳ TOH (1, 'A', 'C', 'B')
        ↳ move 1 from 'A' to 'B'
    ↳ move 2 from 'A' to 'C'
    ↳ TOH (1, 'B', 'A', 'C')
        ↳ move 1 from 'B' to 'C'
```

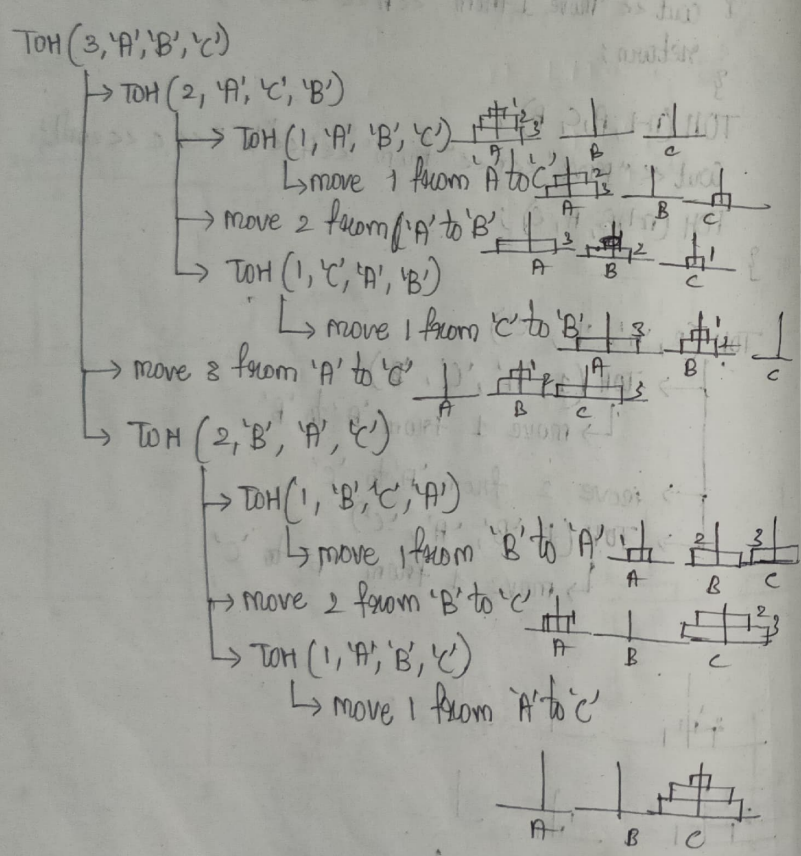


TOH(3, 'A', 'B', 'C')



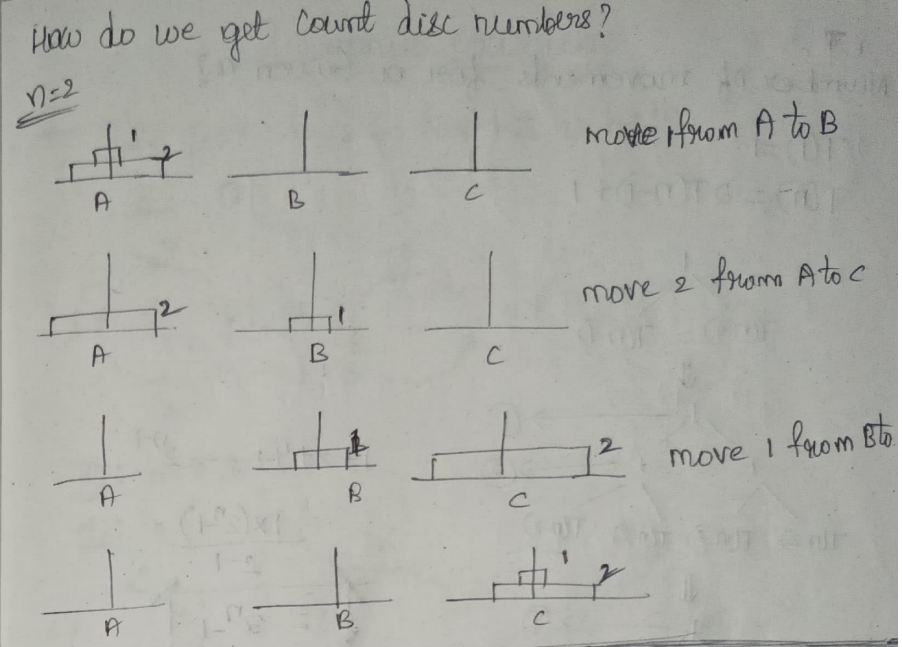
TOH(3, 'A', 'B', 'C')

- TOH(2, 'A', 'C', 'B')
  - TOH(1, 'A', 'B', 'C')
    - move 1 from 'A' to 'C'
  - move 2 from 'A' to 'B'
  - TOH(1, 'C', 'A', 'B')
    - move 1 from 'C' to 'B'
- move 3 from 'A' to 'C'
- TOH(2, 'B', 'A', 'C')
  - TOH(1, 'B', 'C', 'A')
    - move 1 from 'B' to 'A'
  - move 2 from 'B' to 'C'
  - TOH(1, 'A', 'B', 'C')
    - move 1 from 'A' to 'C'

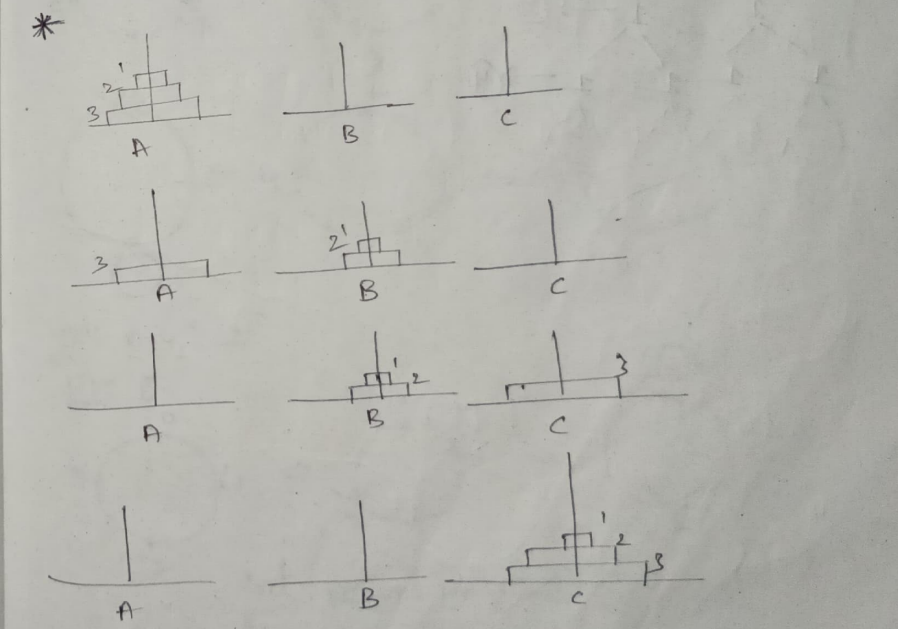


How do we get count disc numbers?

n=2



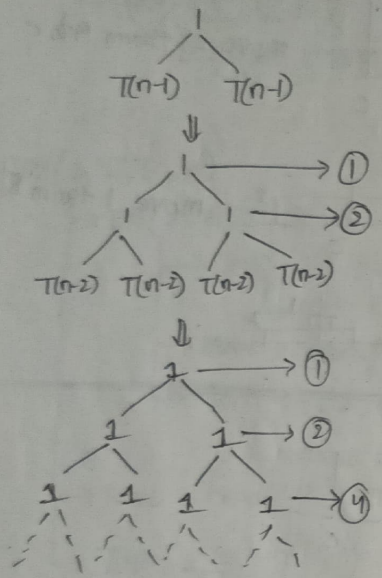
*



Number of movements for a Given n?

$T(1) = 1$

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



$1+2+4+\cdots+2^{n-1}$

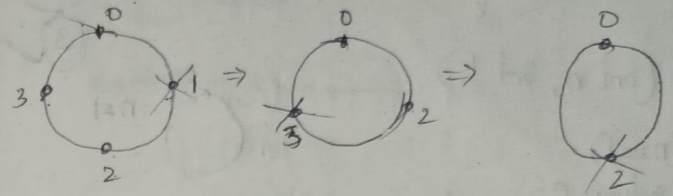
$$\frac{1\times(2^n-1)}{2-1}$$

$$= 2^n - 1$$

Josephus problem

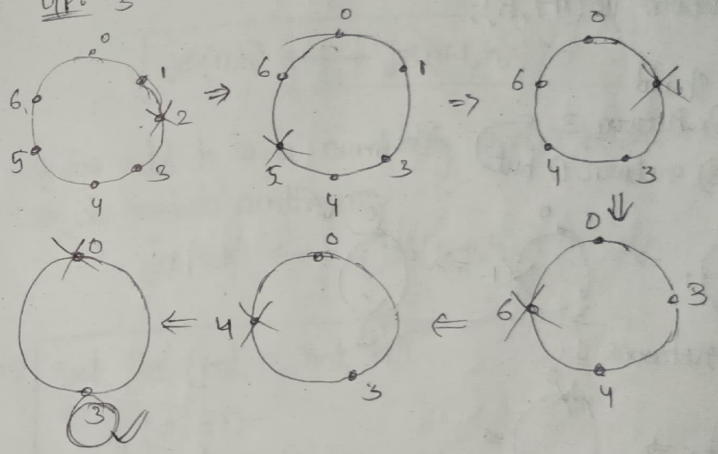
* I/p: n=4, k=2 => first hit is $k-1^{th}$

O/p: 0



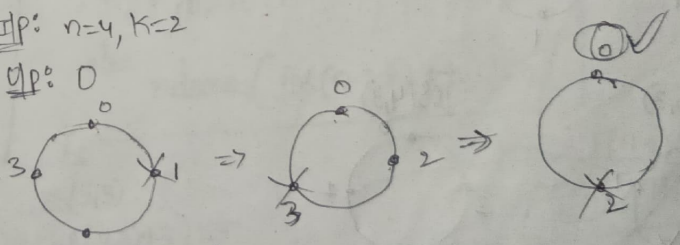
I/p: n=7, k=3

O/p: 3



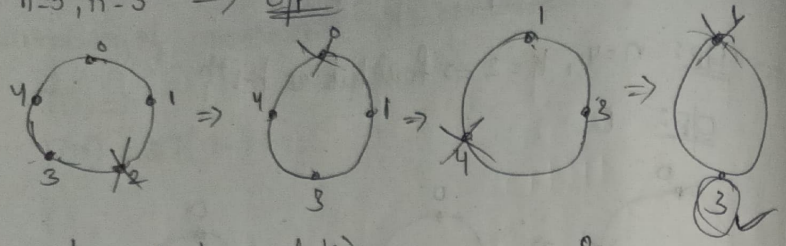
I/p: n=4, k=2

O/p: 0



```
int jos(int n, int k)
{

}
```

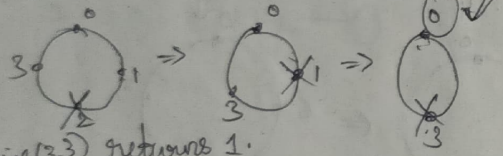
n=5, k=3 ⟹ o/p=3



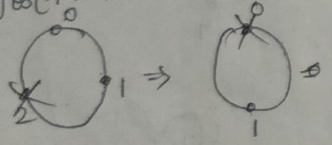
```
int jos (int n, int k)
{
  if(n==1)
    return 0;

  return jos(n-1,k);
}
```

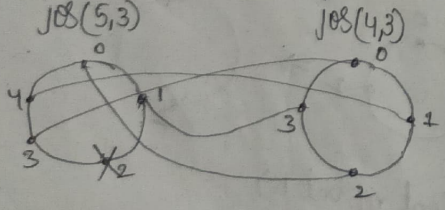
n=1

Should
jos(5,3) return 3 ←
jos(4,3) return 0 but



jos(3,3) returns 1.

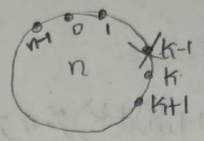


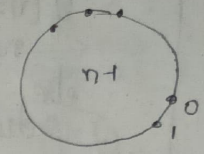


3 ← 0
4 ← 1
0 ← 2
1 ← 3

jos(n,k) ← jos(n-1,k)

General case: for any n, k-1th is the first kill





k ← 0
(k)%n ← 0
(k+1)%n ← 1
(k+2)%n ← 2
(k+i)%n ← i

jos(n,k) ← jos(n-1,k) + k

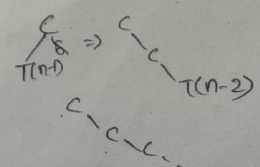
If we add k to a number it will become more than n
so do modulo arithmetic.

jos(n,k) ← (jos(n-1,k) + k)%n

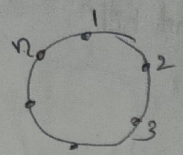
⇒

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

  else
    return (jos(n-1,k)+k)%n;
}
```

T(n) = T(n-1) + c
T(n) = θ(n)



jos(5,3)
⇒ (jos(4,3)+3)%5
⇒ ((jos(3,3)+3)%4+3)%5
⇒ (((jos(2,3)+3)%3+3)%4+3)%5
⇒ ((((jos(1,3)+3)%2)+3)%3+3)%4+3)%5
= 3




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

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

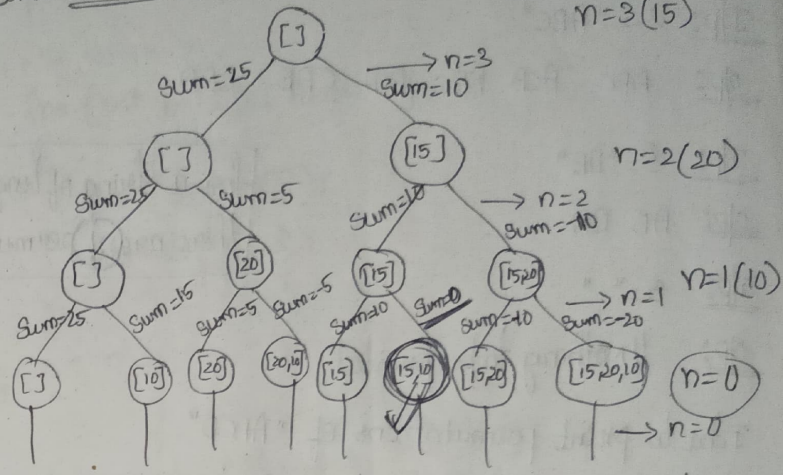
I/p: [10, 20, 15]
Sum = 37
O/p: 0

I/p: [10, 20, 15]
Sum = 0
O/p: 1

subsets of [1,2,3] are 2^n

[ ], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]

Idea for the recursive solution for set [10, 20, 15] and sum = 25



we donot construct the subsets here only found the subsets with sum value.

```
int CountSubsets(int arr[], int n, int sum)
{
  if(n==0)
    return (sum==0)? 1 : 0;
  return CountSubsets(arr, n-1, sum) +
         CountSubsets(arr, n-1, sum-arr[n-1]);
}
```

$TC = 2^n + (2^n - 1)$
$= \theta(2^n)$

print all permutations of a string

I/p: s = "ABC"

o/p: ABC ACB BAC BCA CAB CBA

I/p: s = "AB"

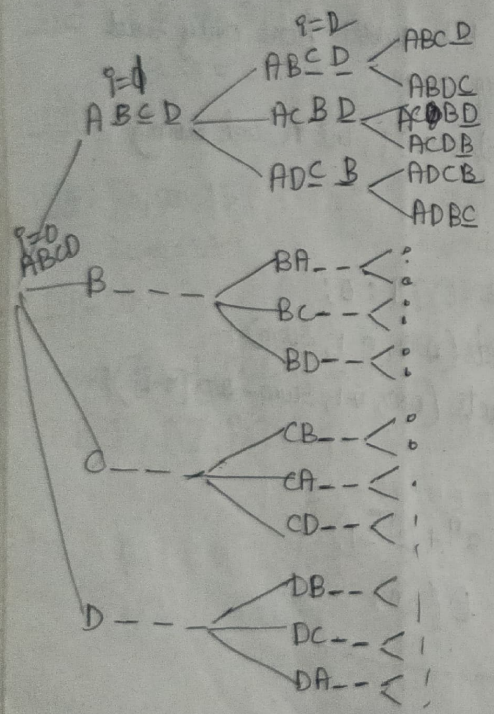
o/p: AB BA

I/p: s = ""

o/p: //Nothing to be printed

for a string of length n, there are n! permutations

Idea to print permutations of "ABCD"



ABCD
i=0
ABCD
BACD
CBAD
DBCA

```
Void permute (String s, int i=0)
{
  . . . . . .
  for (int j=i; j<s.length ; j++)
  {
    swap(s[i], s[j]);
    permute(s, i+1);
    - - - - - - - - -
  }
  - - - - - - - - -
}
```

⇒

```
Void permute (String s, int i=0)
{
  if(i==s.length()-1)
  { cout<<s<<" ";
    return;
  }
  for (int j=i ; j<s.length(); j++)
  {
    Swap (s[i], s[j]);
    permute (s, i+1);
    Swap(s[i], s[i]);
  }
}
```