Paridhi Gupta

Agenda:

1. Power function
2. Print array
3. All indices
4. Check Palindrome

Recursion: A function that calls itself.

Rize 2: Internal Data structures used is stack

Base case

TLE = Time Limit Exceeded

MLE = Memery Limit Exceeded

Quiz 1: what is Recursion?

A fuction that solves the problem by Bracking it into smaller subproblems and calling itself.

Quize 3: 0 Factorial is 1

Base case if N == 0 return 1.

Rize 4: Nth Faitorial

TC: ON SC: ON

TC: (Num of func calls) x (Time taken by each function)

=) $N, N-1, N-2 \ldots 1 = N$

SC: Num of func call x space taken per function

Qize 5: Fibunach:

$$F_N = F_{N-1} + F_{N-2}$$

1. Power Function

⇒ Given 'a' and 'n'

⇒ Find a^n using recursion. ($n >= 0$)

ans = a**n

a = 2 n = 3 ⇒ $2^3 = 8$ = ans

pow(2,3) = 2 x pow(2,2)

pow(a,n) = a x pow(a,n-1)

$2^3 = 2 \times 2 \times 2$

```
def pow(a,n)
    if n == 0 : return 1
    return a x pow(a,n-1)
```

TC!

Base case constant task

number of calls = N ↑

Time per call = Constant

O(N)

n
n-1
n-2
:
0 → ↓

SC :

numer of calls is equal to N

Space per function call = Constant

O (N)

Dry Run : a = 5, n = 3

```
def pow(a, n)          # 5 3
    if n == 0 : return 1
    return a x pow(a, n-1)
```

5 x 25 = 125

5 x 5 = 25

```
def pow(a, n)          # 5 2
    if n == 0 : return 1
    return a x pow(a, n-1
```

sf1 = 5

```
def pow(a, n)          # 5 1
    if n == 0 : return 1
    return a x pow(a, n-1)
```

1

```
def pow(a, n)          # 5 0
    if n == 0 : return 1
    return a x pow(a, n-1
```

method 2

$a = 2$
$n = 3$

$a^n = a \times a \times a \times a \ldots a$

idea: dividing it into two parts.

if even:

$2^8 = 2^4 \times 2^4$

if n & 1 == 0: # even

↓ return fun(n/2) * fun(n/2)

if odd:

$2^7 = 2^6 \times 2^1$

$= 2^3 \times 2^3 \times 2^1$

else:

↓ return a * fun(n/2) * fun(n/2)

a^n

Code:

```
def fun(a, n):
    if n == 0: return 1
    if n%2 == 0: #even
    ↓  return fun(a, n/2) * fun(a, n/2)
    else:
    ↓  return fun(a, n/2) * fun(a, n/2) * a
```

Dry Run a = 2, n = 10

```
def fun(a, n):          2, 10
    if n == 0: return 1
    if n%2 == 0:  #even
        return fun(a, n/2) * fun(a, n/2)
    else:
        return fun(a, n/2) * fun(a, n/2) * a
```

32 × 32 = 1024

32 → same exact func calls gets triggered

```
def fun(a, n):          2, 5
    if n == 0: return 1
    if n%2 == 0:  #even
        return fun(a, n/2) * fun(a, n/2)
    else:
        return fun(a, n/2) * fun(a, n/2) * a
```

4 × 4 × 2 = 32

4 → exact same func calls gets Triggred

```
def fun(a, n):          2, 2
    if n == 0: return 1
    if n%2 == 0:  #even
        return fun(a, n/2) * fun(a, n/2)
    else:
        return fun(a, n/2) * fun(a, n/2) * a
```

2 × 2 = 4

2 → exact same func calls Triggred

```
def fun(a, n):          2, 1
    if n == 0: return 1
    if n%2 == 0:  #even
        return fun(a, n/2) * fun(a, n/2)
    else:
        return fun(a, n/2) * fun(a, n/2) * a
```

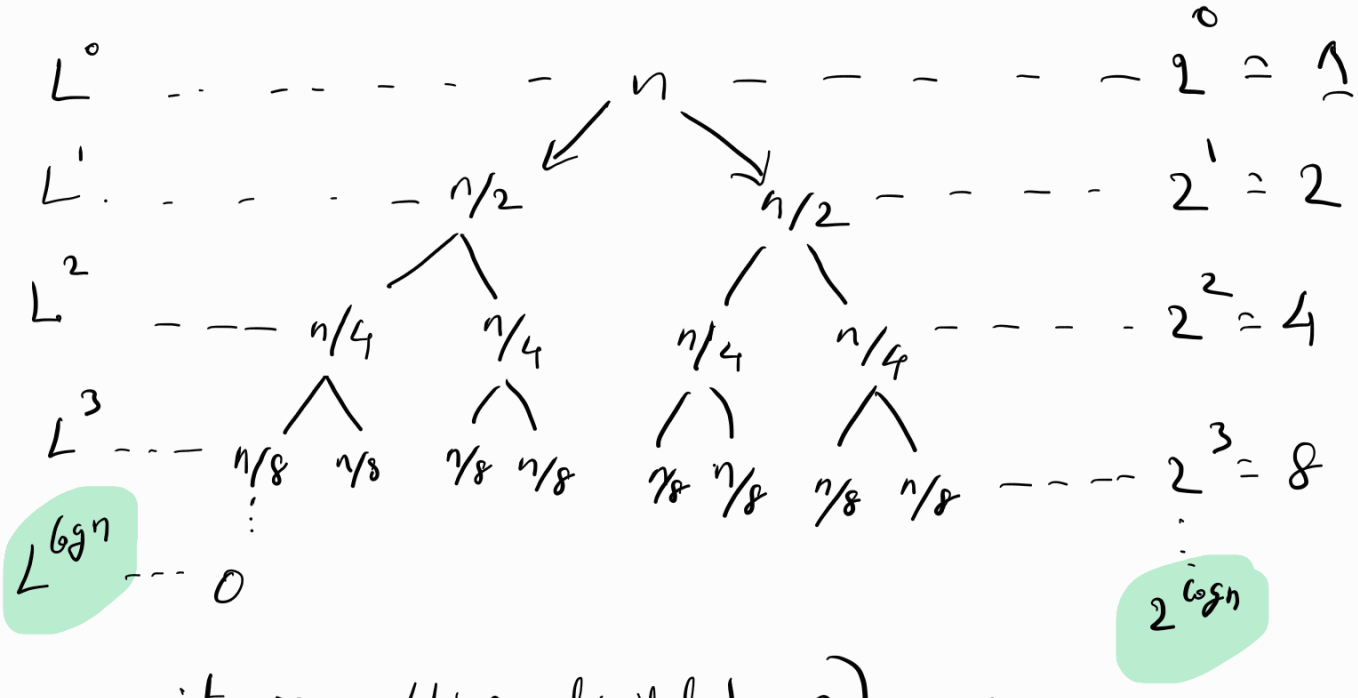
1 × 1 × 2 = 2

1

```
def fun(a, n):          2, 0
    if n == 0: return 1
    if n%2 == 0:  #even
        return fun(a, n/2) * fun(a, n/2)
    else:
        return fun(a, n/2) * fun(a, n/2) * a
```

```
def fun(a, n):          2, 0
    if n == 0: return 1
    if n%2 == 0:  #even
        return fun(a, n/2) * fun(a, n/2)
    else:
        return fun(a, n/2) * fun(a, n/2) * a
```

Time Complexity



as it is getting divided by 2 every time } $\log n$

#calls $2^0 + 2^1 + 2^2 + 2^3 .. + 2^{\log n}$

First Term = a

2nd Term = ar

3rd Term = ar^2

ratio = 2 (r)

$a = 2^0 = 1$

# of Terms = $[0 - \log n]$

$= \log n - 0 + 1$

$\Rightarrow \log n + 1$

Substitution:

Sum of Terms } $\frac{a(r^{\#\text{ of Terms}} - 1)}{r - 1}$ Geometric Progression

Substitution:

$$\Rightarrow \frac{1\left(2^{\log n+1}-1\right)}{2-1} \Rightarrow$$

$$\Rightarrow 2^{\log N-1}-1$$

$$\Rightarrow \cancel{2} \times 2^{\log_2 N} - \cancel{1}$$

$$\Rightarrow$$

$$K = 2^{\log_2 N}$$

Apply $\log_2 N$ both sides

$$\log_2 K = \log_2 \left(2^{\log_2 N}\right)$$

$$x = \log_2 N$$

$$\cancel{\log_2 K} = \cancel{\log_2 N}$$

$$K = N$$

Formula

$$\log_a a^x = x$$

$$TC = O(N)$$

SC = The hight of the Tree

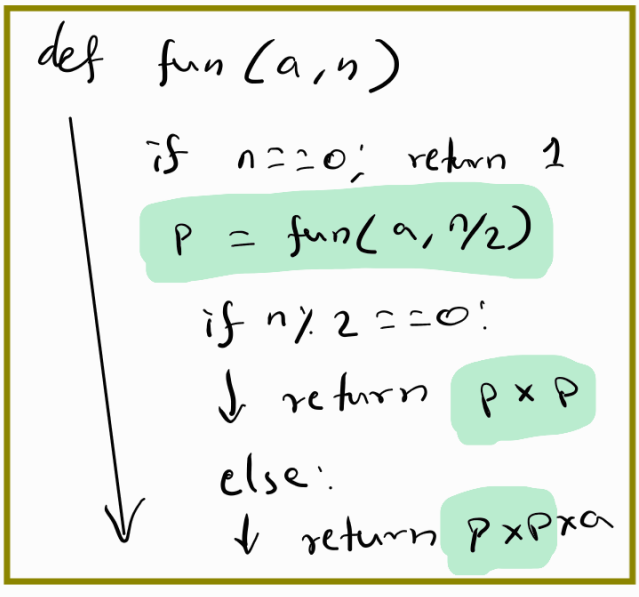
$$SC = O(\log N)$$

space is reduced from Previous method

Optimzed approch: Fast power method

Code:

```
def fun(a, n):
    if n==0: return 1
    if n%2==0:  #even
        ↓ return fun(a, n/2) x fun(a, n/2)
    else:
        ↓ return fun(a, n/2) x fun(a, n/2) xa
```



a = 2 , n = 10

```
def fun(a,n)  2,10
    if n==0: return 1
    p = fun(a, n/2)
    if n%2==0:
        return p*p
    else:
        return p*p*a
```

=> 1024

32

```
def fun(a,n)  2,5
    if n==0: return 1
    p = fun(a, n/2)
    if n%2==0:
        return p*p
    else:
        return p*p*a
```

4

```
def fun(a,n)  2,2
    if n==0: return 1
    p = fun(a, n/2)
    if n%2==0:
        return p*p
    else:
        return p*p*a
```

2

```
def fun(a,n)  2,1
    if n==0: return 1
    p = fun(a, n/2)
    if n%2==0:
        return p*p
    else:
        return p*p*a
```

1

```
def fun(a,n)  2,0
    if n==0: return 1
    p = fun(a, n/2)
    if n%2==0:
        return p*p
    else:
        return p*p*a
```

Time Complexity

number of calls = $\log n$

Time per call = $O(1)$

$\Rightarrow \log n$

n

$n/2$

$n/4$

$n/8$

$\vdots$

$n/\log n$

Space Complexity:

Depth of calls = $\log n$

space per call = $O(1)$

$\Rightarrow \log(n)$

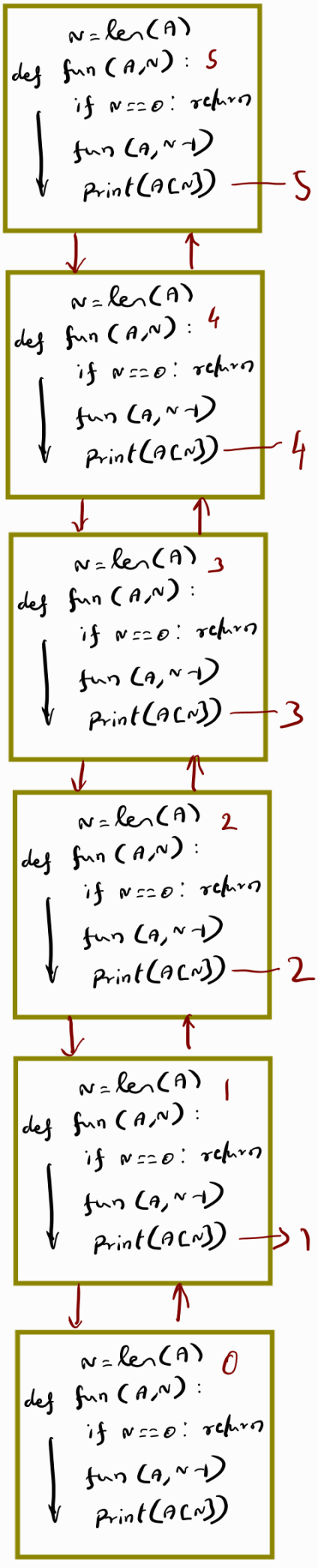
Problem 2 : Print array using recursion

Given int[] A, print all element

A[] = 1, 2, 3, 4, 5 N=5

fun(A,N) = fun(A,N-1) then Print(A[N])

```
N = len(A)
def fun(A,N):
    if N==0: return
    fun(A,N-1)
    Print(A[N])
```



$A = [1, 2, 3, 4, 5]$

$N = 5$

TC:

Number of calls = $N+1$

Time per call = Constant

$\Rightarrow O(N)$

SC:

Number of calls = $N+1$

Space per call = Constant

$\Rightarrow O(N)$

All indeces of Array

Given array int[] A and target int B, return indicies of B.

A = [4, 5, 3, 1, 5, 4, 5] (0 1 2 3 4 5 6)

o/p = [1, 4, 6]

```
ans=[]
for i in range(N)
    if A[i] == B:
        ans.append(i)
return ans
```

```
def fun(A, B, N)
    if N==-1 : return []
    ans: []
    if A[N] == B:
        ans.append(N)
    return ans + fun(A,N,N-1)
```

$\Rightarrow 0 - (N-1)$

$\Rightarrow N - 0$

$\Rightarrow N$

Time per func = Constant

TC = O(N)

SC = max depth of function calls + space per function

$= N + N$

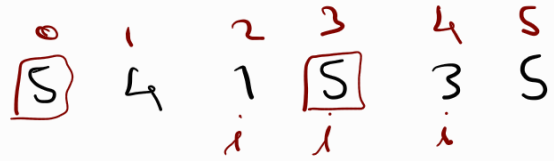
$\Rightarrow O(N)$

one method

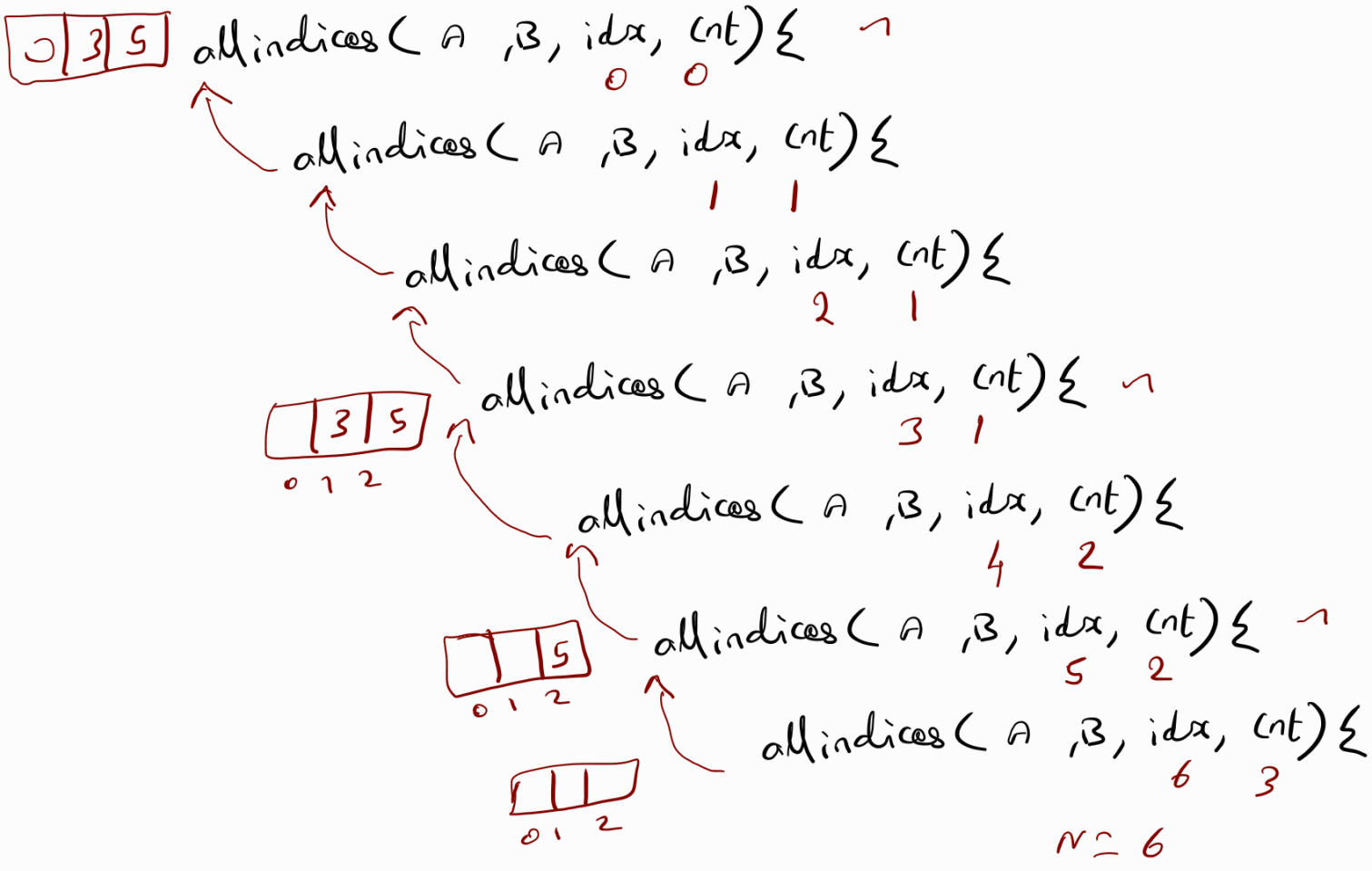
```
int [] allindices(A, B, idx, cnt){
    if (idx == N)
        return int[cnt]
    if (A[idx] == B) {
        int[] arr = allindices(A, B, idx+1, cnt+1)
        arr[cnt] = A[idx]
    }
    if (A[idx] != B) {
        int[] arr = allindices(A, B, idx+1, cnt)
        return arr
    }
}
```

```
def allindices(A, B, idx, cnt):
    if N == idx:
        return [0]*cnt
    if A[idx] == B:
        arr = allindices(A, B, idx+1, cnt+1)
        arr[cnt] = idx
        return arr
    if A[idx] != B:
        arr = allindices(A, B, idx+1, cnt)
        return arr
```

Dry Run



B = 5 N = 6



TC:

# of function call = N+1

Time per function = constant

=> O(N)

SC:-

Dept of the function = N+1

space per function = constant

=> O(N)

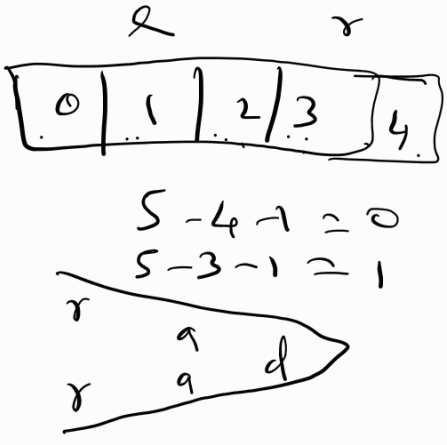
Palindrom or not

Given String A, check if it is palindrom or not using recursive.

Example: "radar" = True

"area" = False

```
N = len(A)
i = 0, j = N-1
while (i < j)
    if A[i] != A[j]:
        return False
    i++
    j--
```




```
def Palindrom(A, N, idx)
                      0
    if idx >= N: return True
    idx2 = idx - N - 1
    if A[idx] == A[idx2]
        Palindrom(A, N, idx+1)
    else:
        return False
```

0	1	2	3	4
r	a	d	a	r

idx → N = 5

idx	idx2	
0	4	5 - 0 - 1 = 4
1	3	5 - 1 - 1 = 3
2	2	5 - 2 - 1 = 2

```
bool checkpalin(A, s, e)
    if (s > e) return True
    if (A[s] != A[e]); return False
    return checkpalin(A, s+1, e-1)
```

TC:

# of func calls = $\frac{N}{2}+1$

Time per function = Constant

$= O(N)$

SC:

Depth of the func call = $N/2+1$

Space per func calls = Constant

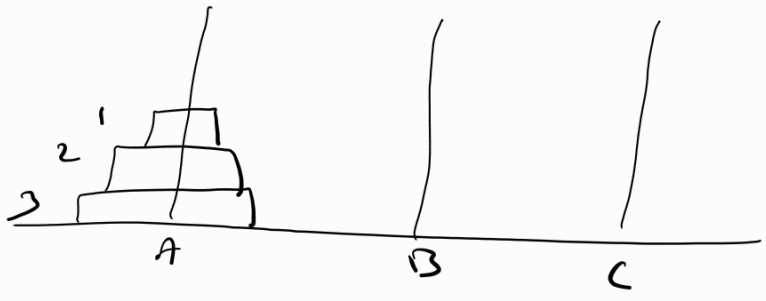
$= O(N)$

Problem: Tower of Hanoi

=> you have 3 towers numbered from 1 to 3 (left to right)

=> A disks numbered from 1 to A (top to bottom) of different sizes which can slide into any tower

=> The puzzle starts with disks **sorted in ascending order of size from top to bottom**



=> only one disk can be removed at a time

=> disk is slide off the top of one tower on to another.

=> disk can not be replaced on top of smaller disk.

=> you have to return 2D array of $m \times 3$ where m is the minimum number of moves needed to solve the problem.

=> in each row there should be 3 int (disk, start, end)

disk = number of the disk being moved

start = number of tower **from** which the disk is being moved.

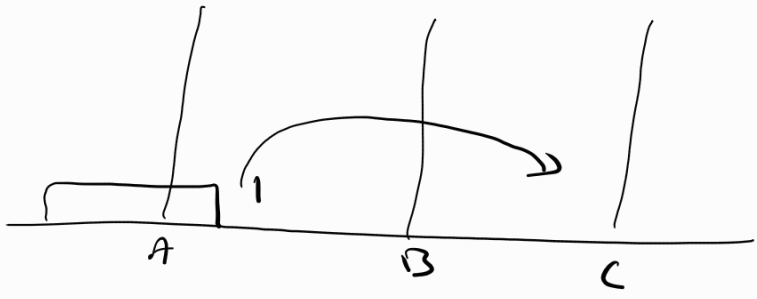
end = number of tower to which disk is being moved.

1 <= A <= 18

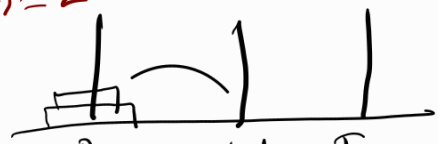
A = 1

[disk From To]

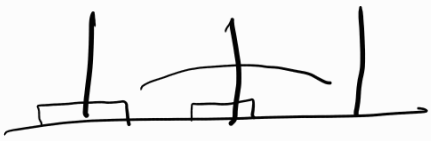
[1 A C]



A = 2



1 A B



2 A C

1 B C

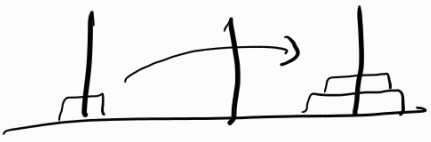
A = 3

1 A C

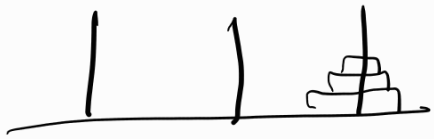
2 A B

1 C B
3 A C

1 B A
2 B C



1 A C



move the disks from A to C

1. moving n-1 disk to Helper tower
2. move the nth disk into C

moving two disks

1	A	B
2	A	C
1	B	C

3. Helper tower and From tower changes.

move(N, A, C, B) = move(N-1, A, B, C) + move nth disk manually to C

move(3, A, C, B) = move(2, A, B, C)
+ move 3rd disk manually to C

move(2, A, B, C) =
move(2, B, C, A)

if disk == 2 :

1	A	C
2	A	B
1	C	B

Source
Helper
Destination

Code Overview :

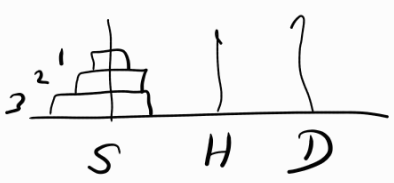
```
def toh(N, S, H, D):          # S: source, H: helper, D: destination
    if (N == 0) return
    toh(N-1, S, D, H)
    Print(nth disk from A -> C)
    toh(N-1, D, S, H)
```

Code

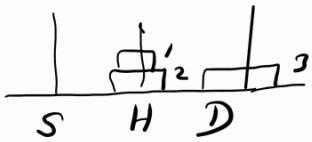
```
steps = [ ]

def towerOfHanoi(N, start, helper, end, steps):
    if N==0:
        return
    # moving N-1 disks into helper tower
    towerOfHanoi(N-1, start, end, helper, steps)
    # manualy moving Nth disk to destination Tower
    temp = [ ]
    temp.append(N) # disk
    temp.append(start) # From Tower
    temp.append(end) # To Tower
    steps.append(temp)
    # moving N-1 disks into Destination Tower
    towerOfHanoi(N-1, helper, start, end, steps)
```

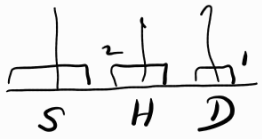
Dry-Run



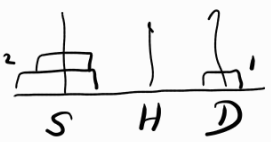
toh(3, S, H, D)



- toh(2, S, D, H)



  - toh(1, S, H, D)



ans:-

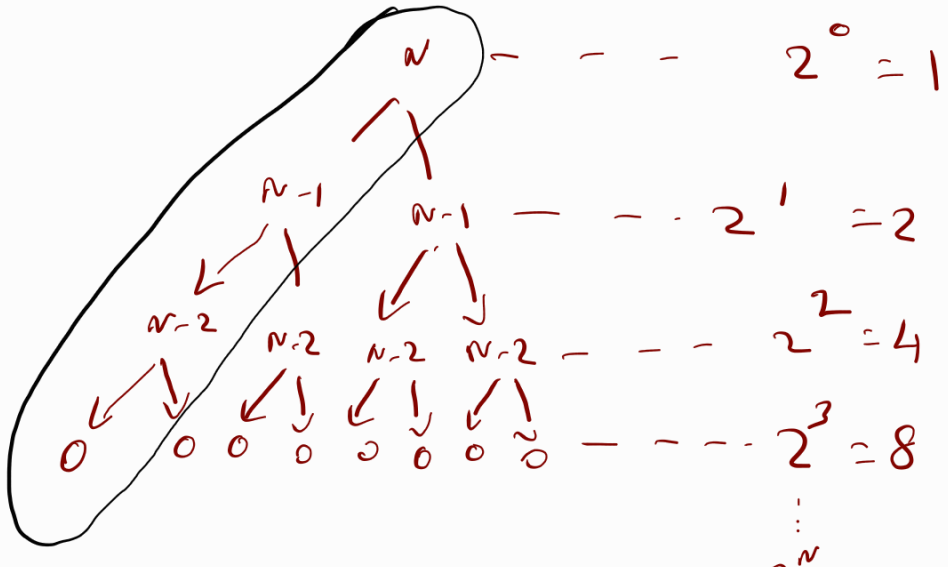
1	S	D
2	S	H
1	D	H
1	H	S
2	H	D
1	S	D

toh(1, D, S, H)

toh(2, H, S, D)

toh(1, H, D, S)

toh(1, S, H, D)



TC:

$\Rightarrow 2^0 + 2^1 + 2^2 \ldots 2^N$

$\Rightarrow 2^N + 1$

TC = 2^N

SC:

Depth of function calls = N+1

Space per function call = Constant

$\Rightarrow O(N)$

Problem: Is magic

=> Given int A, check if it is magic number or not

=>

A = 83557 = 28 = 10 = 1

A = 1291 = 13 = 4

idea:

fun(123) = 123%10 + fun(123/10)

```
def isMagic(A):
    if A == 0:
        return 0
    value = (A%10) + fun(A/10)
    if value > 9:
        value = (value%10) + fun(value/10)
    return value
if isMagic(A) == 1:
    return 1
else:
    return 0
```

TC: O(logN)

SC: O(logN)

1. we are dividing n/10 every time
2. makes recursive call with reduced n
3. # func calls proportional to num. digits in A.

Problem: max of an array using Recursion.

Given array A, find max element.

idea: fun([1,2,3,4]) = max(A[0], func([2,3,4]))

```
def findMax(A, i=0)
    if i == N: return 0
    output = findmax(A, i+1)
    if A[i] > output:
        return A[i]
    else
        return output

N = len(A)
if N == 1:
    return A[0]
return findmax(A)
```

Problem : First Index using Recursion

=> Given int[] A and int B

=> return First index of element is equal to B

=> else return -1

A = [-3, 5, 6, 2] B = 6

func(A, B, i=0) = if A[i] != B

fun(A, B, i+1)

else:

return 1

Base case: if i==N: return -1

i = 0

:

i = N

```
def findeleB(A, B, i=0)
    if i==N: return -1
    if A[i] != B:
        return findeleB(A, B, i+1)
    else return i
```

TC: O(N)

SC: O(N)

Problem: Last index using Recursion

=> Given array A and int B

=> return last index at which int B is found in array.

=> else return -1

```
def findeleB(A, B, N)
    if N == -1 : return -1
    if A[N] != B :
        return findeleB(A, B, N-1)
    else return i

N = len(A)
findeleB(A, B, N-1)
```