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

$a^n = \underbrace{a \times a}_{} \times \underbrace{a \times a}_{} \ldots a$

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:

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

$= \underline{2^3} \times \underline{2^3} \times 2^1$

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):
    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, 10

32 × 32

64

96

1024

32

same exact func calls gets trigged

```
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
```

2, 5

4 × 4 × 2

4

exact same func calls gets Trigged

```
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
```

2, 2

2 × 2

2

exact same func calls Trigged

```
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
```

2, 1

1 × 1 × 2

1

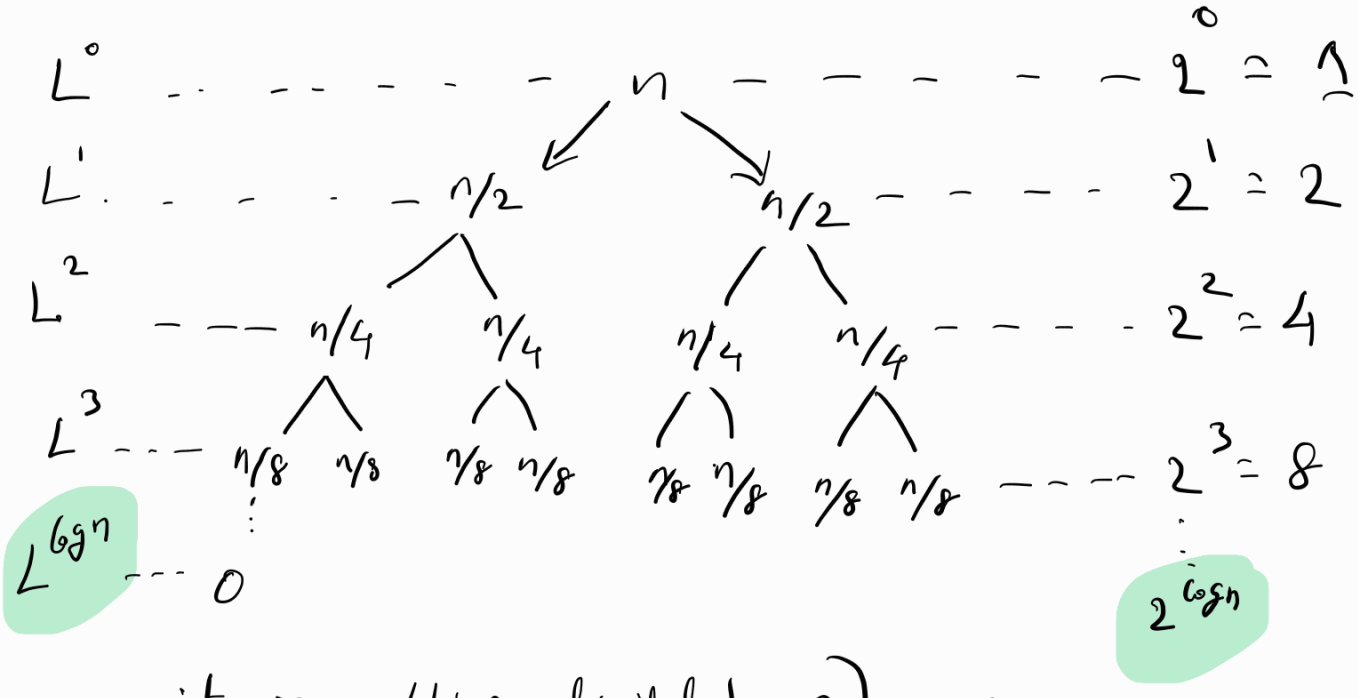
```
def fun(a, n):
    if n == 0: return 1
    if n%2 == 0:  #even
        return fun(a, n/2) * fun(a, n/2)
    else:
        return fun(a, n/2) * fun(a, n/2) * a
```

2, 0

```
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
```

2, 0

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

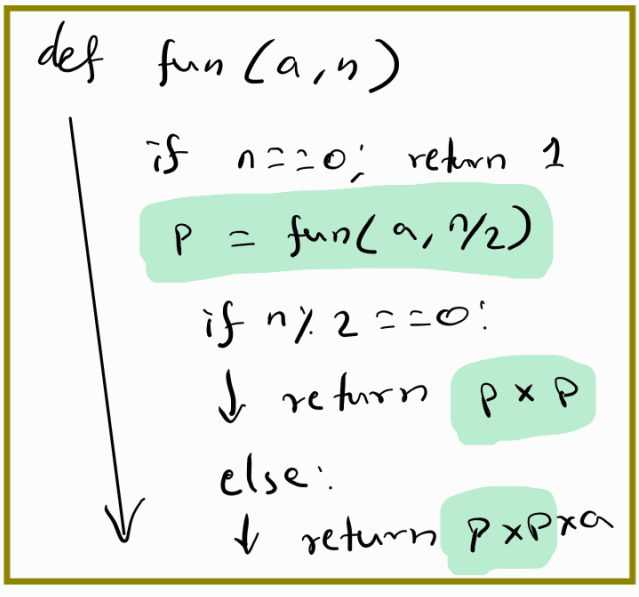
$$SC = O(\log N)$$

space is reduced from Previous method

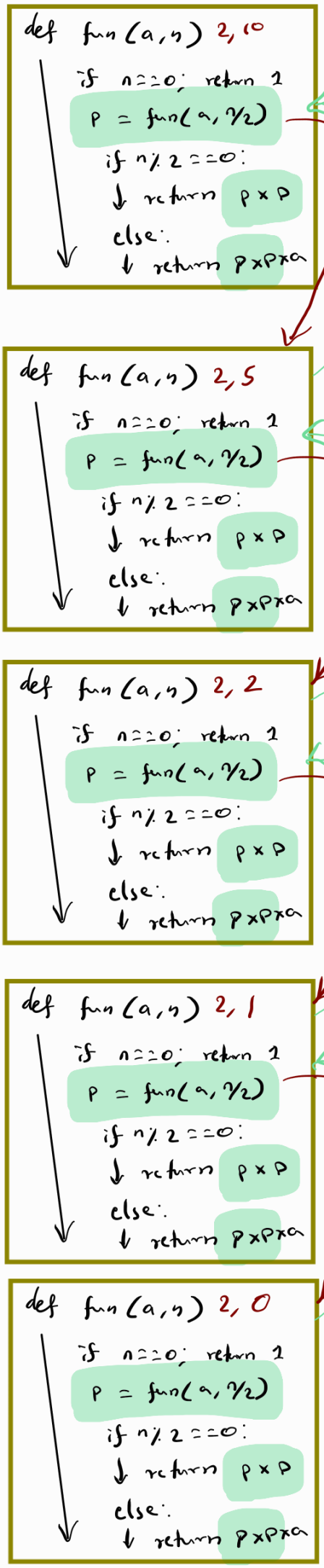
Optimzed approch: Fast Power method

Code:

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



a = 2 , n = 10



=> 1024

32

4

2

1

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 = $O(1)$

$\Rightarrow \log(n)$

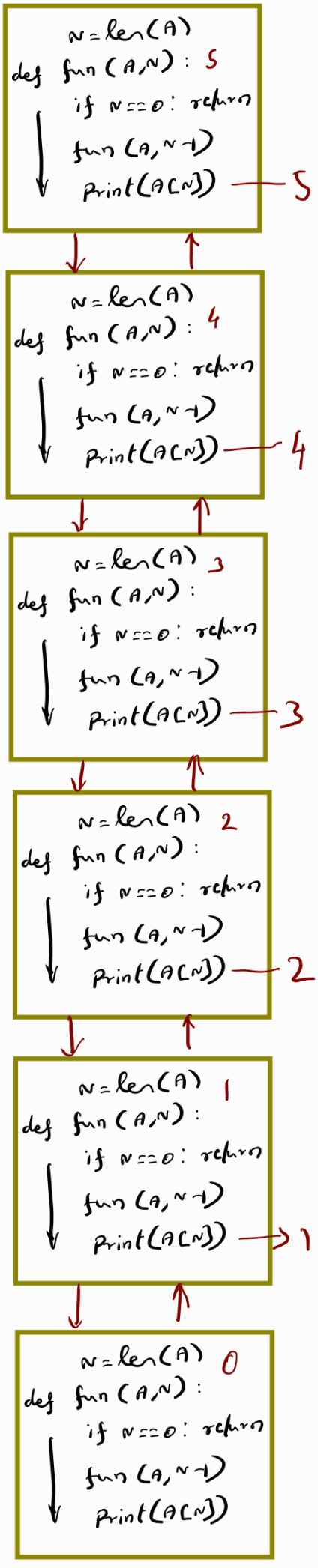
Problem 2 : Print array using recursion

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)
```

$\Rightarrow 0 - (N-1)$

$\Rightarrow N$

Time per func = Constant

TC = O(N)

SC = max depth of function calls + space per function

$= N + N$

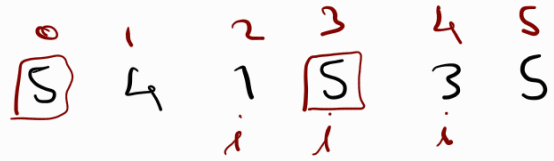
$\Rightarrow O(N)$

one method

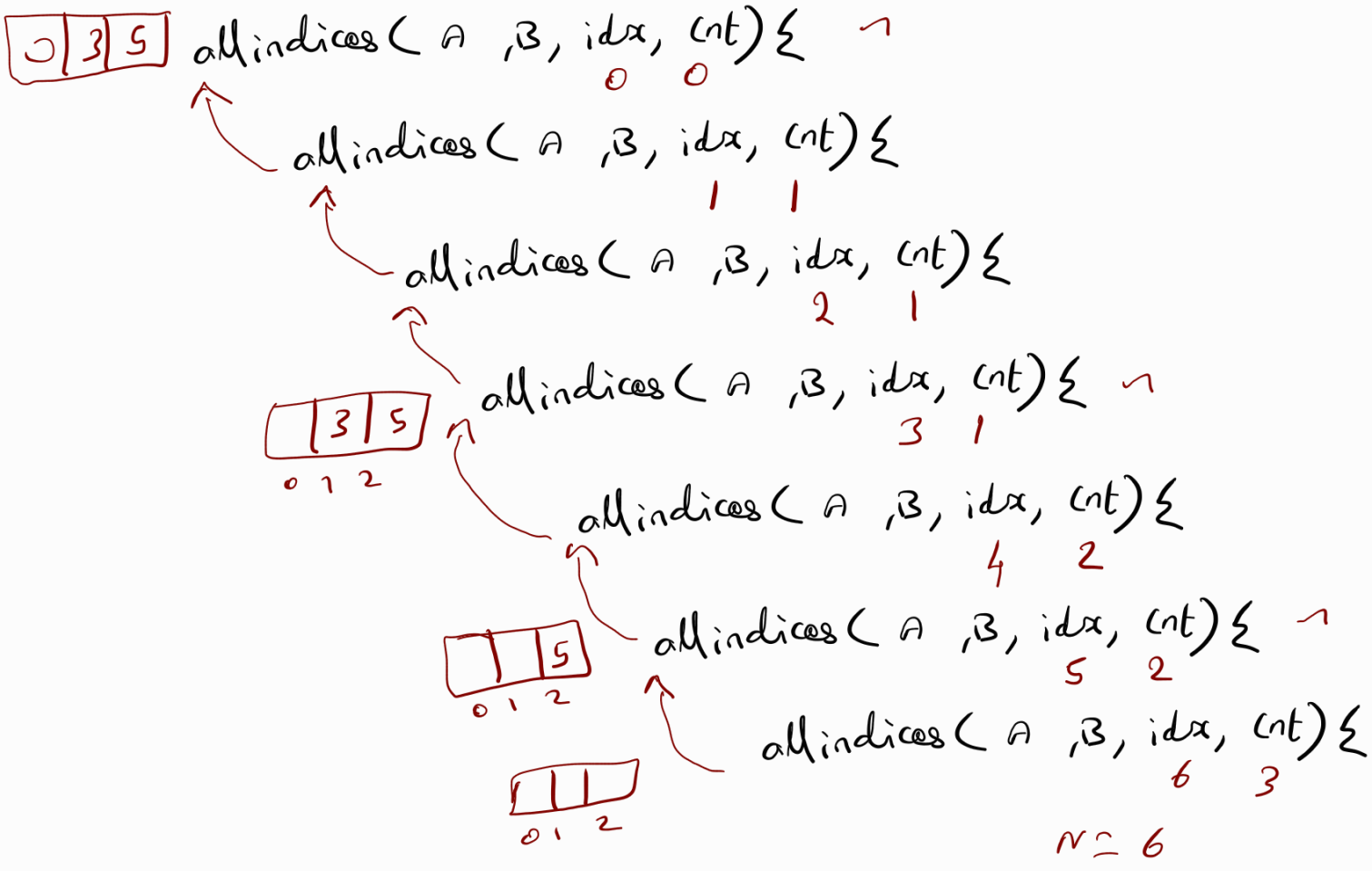
```
int [] allindices(A, B, idx, cnt){
    if (idx == N)
        return int[cnt]
    if (A[idx] == B) {
        int[] arr = allindices(A, B, idx+1, cnt+1)
        arr[cnt] = A[idx]
    }
    if (A[idx] != B) {
        int[] arr = allindices(A, B, idx+1, cnt)
        return arr
    }
}
```

```
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)

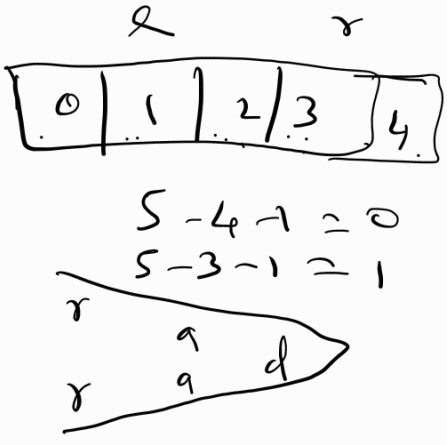
Palindrom or not

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 = 2N; 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

$= O(N)$