Lecture 17 : Recursion - 2

(68) Sum of digits using Recursion

```
def sumDigits(num):
    if num <= 0:
        return 0
    dig = num % 10
    return dig + sumDigits(num//10)
```

NOTE :

1. no other variable was declared
2. no. of function calls : len(str(x)) or $\log_{10} x + 1$

SC : O(log N)

3. How to calculate time complexity

For a number N,

$T(N) = T(N//10) + O(1)$

time taken for whole fn to execute

$T(N//100) + 2 * O(1)$

$\vdots$

$T(N) = T\left(\frac{N}{10^k}\right) + k * O(1)$

$\frac{N}{10^k} = 0$

What will T(0) return (in O(1))

when? $\Rightarrow$

$10^k > N$

$k > \log_{10} N$

TC: $\log_{10} N * O(1)$

TC: $O(\log_{10} N)$

(69) Power function

$2^{10} \rightarrow 2^9 \times 2 \rightarrow 2^8 \times 2 \times 2$ .....

```
def power(a, n):
    if n==0:
        return 1
    return a * power(a, n-1)
```

Finding T.C.

assume time taken as $T(a, n)$

$$T(a,n) = T(a, n-1) + O(1)$$
$$= T(a, n-2) + 2 * O(1)$$
$$\vdots$$
$$= T(a, 1) + (n-1) * O(1)$$
$$= T(a, 0) + n * O(1)$$

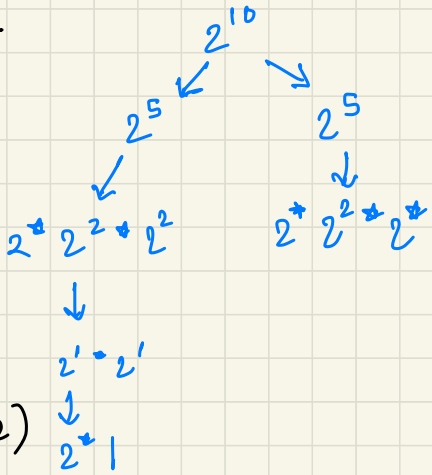
almost instantly returns 1

TC: O(N)

SC: O(N)

Optimized Power Function

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



T.C.

$$T(a,n) = 2^* T(a, n//2) + O(1)$$

$$T(a,n) = 2^k \cdot T\left(a, \frac{n}{2^k}\right) + (2^k - 1)O(1)$$

$T\left(a, \frac{n}{2^k}\right) \rightarrow O(1)$

$2^k > n$

$k > \log_2 n$

$= n \cdot O(1) + (n-1)O(1)$

TC ~ O(N) No improvement

SC: O(logN) Improved

Time-optimized Power Function

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

This time, we store the function call in a var & its being called only once thus reducing time for us.

TC

$$T(a,n) = T(a, n/2) + O(1)$$

$$\downarrow$$

$$T(a, n/4) + O(1)$$

$$\vdots$$

$$T(a,n) = T(a, n/2^k) + k * O(1)$$

$k > \log_2 n$

$$= O(1) + \log_2 n * O(1)$$

$$= O(\log_2 N)$$

TC : $O(\log N)$

SC : $O(\log N)$

(70) Finding remainder for a division by a large number ex. $2^{100} \% 7$

Using Modular Arithmetic Property

```
def mod-power(a, n, M):
    if n == 0:
        return 1
    x = mod-power(a, n//2, M)
    if n % 2 == 0:
        return (x * x) % M
    else:
        return (a * (x * x % M)) % M
```

✱ Calculating TC for a recursive relation

Q.1. $T(N) = 2 * 2T(N-1) + O(1)$

$\downarrow$

$2^2 * T(N-2) + 2 * O(1) + O(1)$

$\downarrow$

$2^3 * T(N-3) + 2^2 O(1) + 2^1 O(1) + 2^0 O(1)$

$= 2^K T(N-K) + (2^K - 1) O(1)$

$O(1)$ when $K = N$

T.C. $= 2^N O(1) + (2^N - 1) O(1)$

TC : $O(2^N)$

Q.2.

$$T(N) = 2T(N/2) + O(1)$$

$$\vdots$$

$$2^K T(N/2^K) + (2^K - 1)O(1)$$

$$\downarrow$$

$$2^K > N$$

$$K > \log_2 N$$

$$N * O(1) + (N-1)O(1)$$

TC : $O(N)$

Q.3 $T(N) = 2T(N/2) + O(N)$

$\downarrow$

$[2^0 T(N/4) + O(N/2)]$

$= 2^2 T(N/4) + 2^1 O(N/2) + 2^0 O(N)$

$\vdots$

$= 2^K T(N/2^K) + 2^{K-1} O(N/2^{K-1}) +$

$\vdots$

$2^1 O(N/2) +$

$2^0 O(N)$

$\sum_{0}^{K-1} 2^i O(N/2^i)$

$O(N)$ is added K times

K ~ $\log_2 N$ times

i.e. $N \cdot \log N$

$= 2^K T(N/2^K) + \sum_{0}^{K-1} 2^i O(N/2^i)$

$N/2^K \to O(1)$

$K > \log_2 N$

$= N \cdot O(1) + N^0 \log N$

TC : $N + N \log N$

(71) Return an array of integers representing gray code sequence

Ex. 2 [0 1 3 2]

00 01 11 10

```
def code(A):
    if A == 1
        return [0, 1]
    value = code(A-1)
    ans = value.copy()
    for i in range(len(value)-1, -1, -1):
        ans.append((1 << (A-1)) + value[i])
    return ans
return code(A)
```

code(2)

value = code(1) = [0, 1]

ans = [0, 1]

i : 1 to 0

i = 1 ans.append(2 + value[1]) → [0, 1, 3]

i = 0 ans.append(2 + value[0]) → [0, 1, 3, 2]