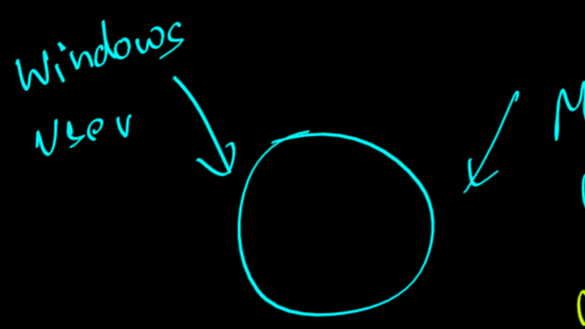
Time and Space Complexity

{ 200ms, 250ms, 1ms, 1000ms } → Run-time (Machine Dependent)



∝ Run time Analysis

Asymptotic Analysis

Algorithm ∝ input size → growth of Runtime wrt input size.

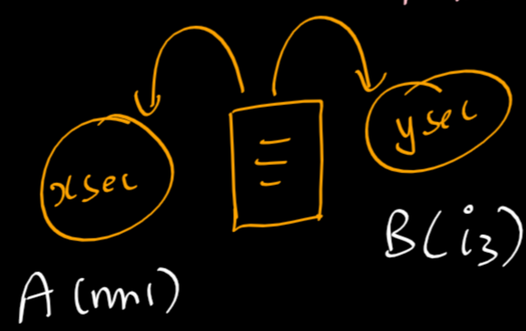
→ function of Runtime

T(N) → dependent on input size

ignore most all the factors.

↓

machine independent



for A

input N = 100 RunTime = 100ms

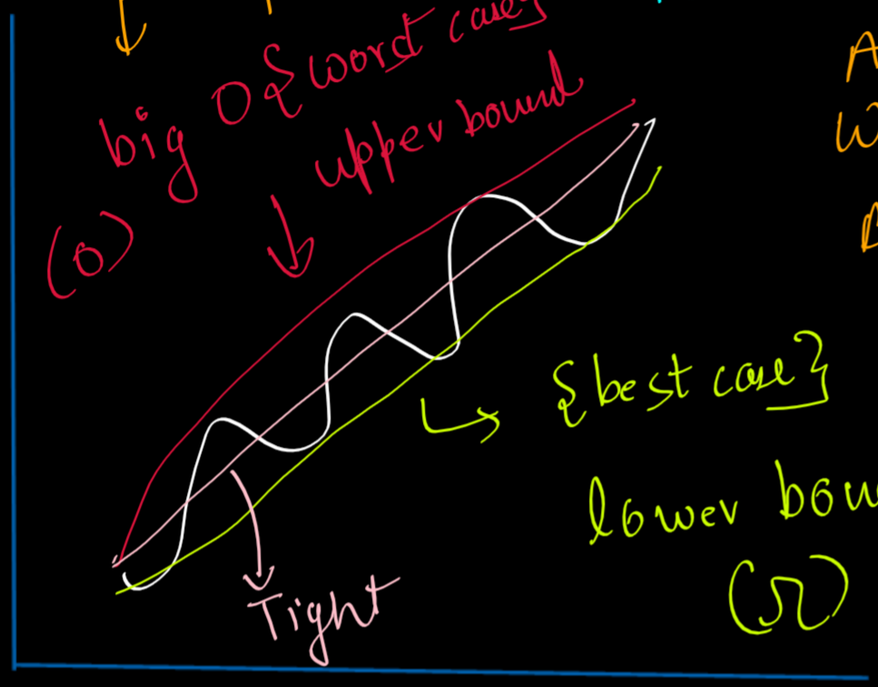
N = 1000 Runtime = 1000 ms } Linear Dependence

Bure se bura (In Reality, WC ≠ Big Oh ≠ Upper Bound)

↓ Time

↓ big O & worst case?

(O) ↓ upper bound



↳ & best case?

lower bound

(Ω)

bound

(Avg case) Θ

$T(n) = 5n^2 + 3n + 6$

Avg case: $\Theta(n^2)$

Worst case: $O(n^2)$

Best case: $\Omega(n^2)$

Acche se Accha TIME

Compilation + Linking + Loading

} Ignored in Run-Time Analysis

** To hum isa kab sakte hain ki

jab program process ban gaya hai

us k baad ka tym run-time hoga

** Best analysis is WORST CASE Scenario.

Bure se bure din me koi algo kaisa perform karti hai wahi comparison best rahega.

↳ Yani jab program RAM pe AA JATA HAI.

** Amortized Poochna Hai (during dion vector design)

Time Complexity Chart

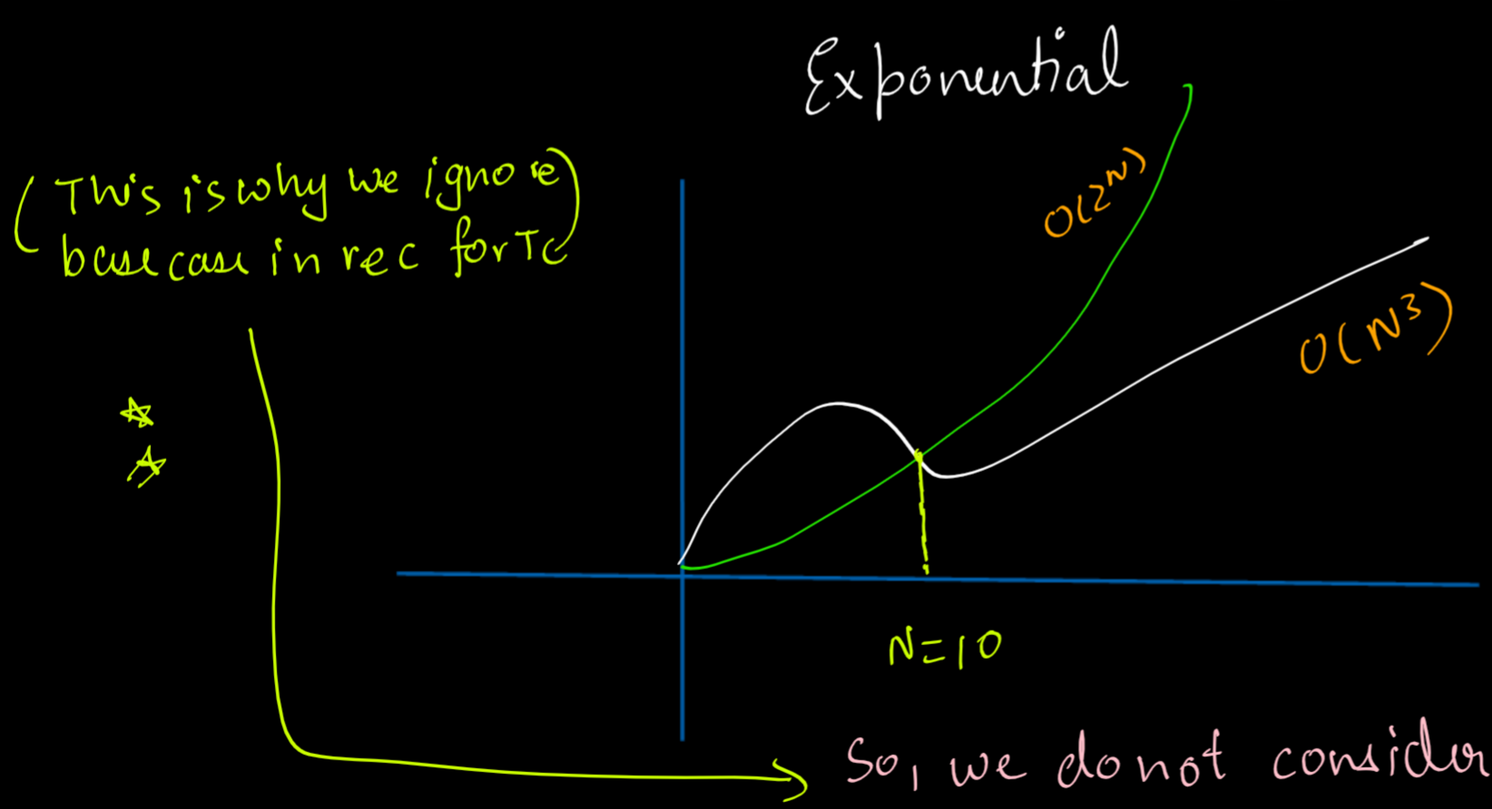
Polynomial

constant → Linear → logarithmic → Root → quadratic → cubic

$O(1) < O(\log_2 n) < O(\sqrt{n}) < O(n) < O(N \log N) < O(N^2) < O(N^3)$

$< < < O(2^N) < O(3^N) < O(N!)$ or $O(N^N)$

Exponential



Ignore constants as they just denote machine dependency

$O(N^3)$ $O(2^N)$

$O(2^3)$ < $O(2^2)$

$O(3^3)$ > $O(2^3)$

$O(4^3)$ < $O(2^4)$

for Loops Time Complexity

```java
for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
                System.out.println(i + " " + j);
        }
}
```

n=5

i=0 j= 0,1,2,3,4 (1+1+1+1+1) =O(n)

i= 1 j= 0,1,2,3,4 (5k) = O(n)

i=2 j= 0,1,2,3,4 (5k) = O(n)

i=3 j= 0,1,2,3,4 (5k) = O(n)

i=4 j= 0,1,2,3,4 (5k) = O(n)

Using Property

$f(n) * O(g(n)) = O(f(n) * g(n))$

$n * O(n) = O(n^2)$

```
for (int i = 0; i < n; i++) {
        for (int j = n - 1; j >= 0; j--) {

        }
}
```

n=5

i=0 j= 4,3,2,1,0 k+k+k+k+k → O(n)

i=1 j= 4,3,2,1,0 k+k+k+k+k → O(n)

i=2 j= 4,3,2,1,0 k+k+k+k+k → O(n)

i=3 j= 4,3,2,1,0 k+k+k+k+k → O(n)

i=4 j= 4,3,2,1,0 k+k+k+k+k → O(n)

n * O(n)

= O(n²)

```
for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {

        }
}
```

** Humesha nested-for $O(n^2)$

Nahi Hota

i=0 j= 0 → k

i=1 j= 0,1 → 2k

i=2 j= 0,1,2 → 3k

i=3 j= 0,1,2,3 → 4k

i=4 j= 0,1,2,3,4 → 5k

$k\left(\frac{n(n+1)}{2}\right)$

$O(n^2)$

```
for (int i = 1; i < n; i *= 2) {

}
```

$n = 35$

$i = 1 \rightarrow 1$

$i = 2 \rightarrow 2$

$i = 4 \rightarrow 3$

$i = 8 \rightarrow 4$

$i = 16 \rightarrow 5$

$i = 32 \rightarrow 6$

$O(\log_2 35) \approx O(\log_2 n + 1)$

iterations

$O(\log_2 n)$

```
for (int i = 0; i * i < n; i++) {

}
```

$n = 64$

(k) $i = 0$ $0*0 < 64$

(k) $i = 1$ $1*1 < 64$

(k) $i = 2$ $2*2 < 64$

(k) $i = 3$ $3*3 < 64$

(k) $i = 4$ $4*4 < 64$

(k) $i = 5$ $5*5 < 64$

(k) $i = 6$ $6*6 < 64$

(k) $i = 7$ $7*7 < 64$

(k) $i = 8$ $8*8 < 64$

for 64 → runs for 8 times

$\sqrt{n}$ times

$O(\sqrt{n})$

$O(\sqrt{n}+1) * k$

$\approx O(\sqrt{n})$

```java
for (int i = 0; i < n; i++) {

}
for (int j = 0; j < n; j++) {

}
```

→ O(n)

```java
for (int i = 0; i < n; i += 2) {

}
```

→ O(n)

```java
for (int i = 0; i < n; i++) {
        for (int j = 0; j < 4; j++) {

        }
}
```

→ O(n)

```java
for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j *= 2) {

        }
}
```

→ ∞

Algorithms - Competitive Programming Chart!

Input Size (N)	Time Complexity (worst case)	Hint of Algorithm
≤ 10	$O(n!)$ or $O(2^n)$	Backtracking {eg permutation, subsets}
≤ 18	$O(2^n \cdot n)$	Travelling Salesman
≤ 22	$O(2^n \cdot n^2)$	DP with Bitmasking
$\leq 10^2$	$O(n^4)$	Quadruplets / 4 Nested loops (4D DP)
$\leq 4 \times 10^2$	$O(n^3)$	3 nested loops, Floyd Warshall, (3D DP)
$\leq 2 \times 10^3$	$O(N^2 \log N)$	Nested loops & Binary Search
$\leq 10^4$	$O(N^2)$	2 Nested loops (Bubble, Insert, Select) (2D DP)
$\leq 10^5 - 10^6$	$O(N \log N)$	Sorting (QS, MS), BS on Answer, Greedy, etc.
$\leq 10^8$	$O(N)$	Array & Strings, (1D DP)
$\leq 10^{18}$	$O(\log_2 N)$ or $O(1)$	mathematical formula

leetcode
Codechef
Codeforces

→ 1s (10^8 operations)

for ex: if N is of the order 10^8
then max complex will be O(N)
as 10^8 max op will be performed in 1s.

Time Complexity = (No of calls)^{height} + (preorder + postorder) * height

Worst case

No of calls → Calls from one Node at a moment not the total No of calls

** Constant operations:

- input output
- arithmetic operations
- if/else

```java
package Recursion_In_Arrays.Codes;

import java.util.*;
public class DisplayArray {
    public static void main(String[] args) throws Exception {
        // write your code here
        Scanner scn = new Scanner(System.in);  → k1
        int n = scn.nextInt();  → k2

        int[] arr = new int[n];  → k3

        for(int i=0;i<n;i++) {
            arr[i] = scn.nextInt();
        }  } O(n)
        scn.close(); → k4
        displayArr(arr,0);  → TC
    }

    public static void displayArr(int[] arr, int idx){
        if(idx == arr.length) return;

        System.out.println(arr[idx]);  → k
        displayArr(arr,idx+1);
    }
}
```

Total TC = ~~k_1~~ + ~~k_2~~ + ~~k_3~~ + O(n) + TC

TC(displayArr) = $(1)^N$ + (pre + post) * N

= 1 + (k+0)N

= 1 + kN

Call Stack

5 → base case

4
3
2
1
0 } N

TC(displayArr) = O(N)

Total TC = O(n) + O(n) = O(n)

Power Linear

```java
import java.util.*;
public class PowerLinear {
    public static void main(String[] args) throws Exception {
        // write your code here
        Scanner scn=new Scanner(System.in); → k1
        int n=scn.nextInt(); → k2
        int p=scn.nextInt(); → k3
        scn.close(); → k4
        int ans=power(n,p); → TC(power)
        System.out.println(ans); → k5
    }

    public static int power(int x, int n){

        if(n==0)
        {
            return 1;
        }

        return x*power(x,n-1);
    }
}
```

callStack(x,n)

(2,5)
⇓
(2,4)
⇓
(2,3)
⇓
(2,2)
⇓
(2,1)
⇓
(2,0) → base case

N = height

No of calls = 1

Height = N

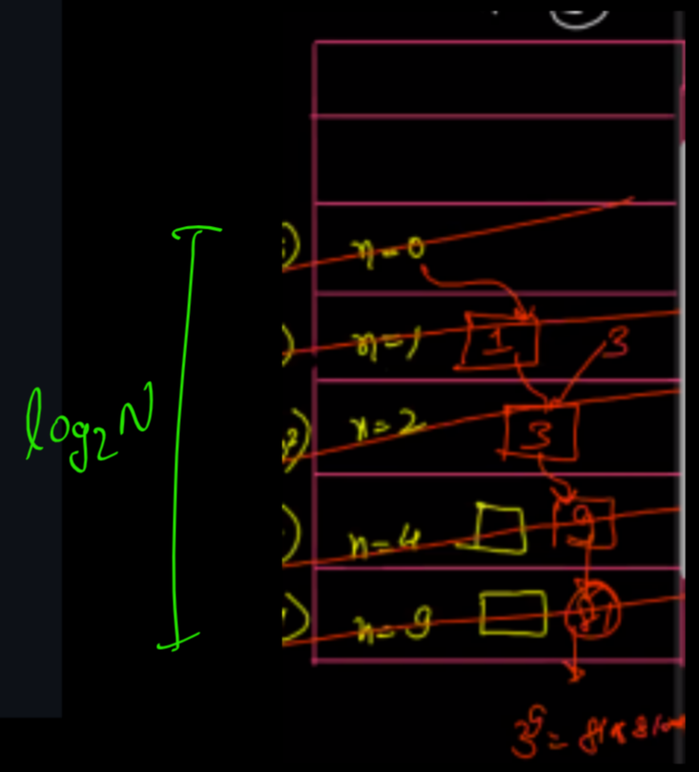
Multiplication done in pre = $(1)^N + (k+0)*N$

$= 1+kN = O(N)$

Power Logarithmic

```java
public class PowerLogarithmic {
    public static void main(String[] args) {
        Scanner scn=new Scanner(System.in);
        int n=scn.nextInt();
        int p=scn.nextInt();
        scn.close();
        int ans=power(n,p);
        System.out.println(ans);
    }

    public static int power(int x, int n) {

        if(n == 0) return 1;

        int pxnby2 = power(x, n/2); //call this only once else it will be O(n) only

        if(n%2 == 0) {
            return pxnby2*pxnby2;
        }

        return x*pxnby2*pxnby2;
    }
}
```

$(\text{No of calls})^h + (\text{pre} + \text{post}) * h$



→ If we remember the call stack, n goes to n/2 which is clear from the call as well

No of calls = 1

height = $\log_2 N$

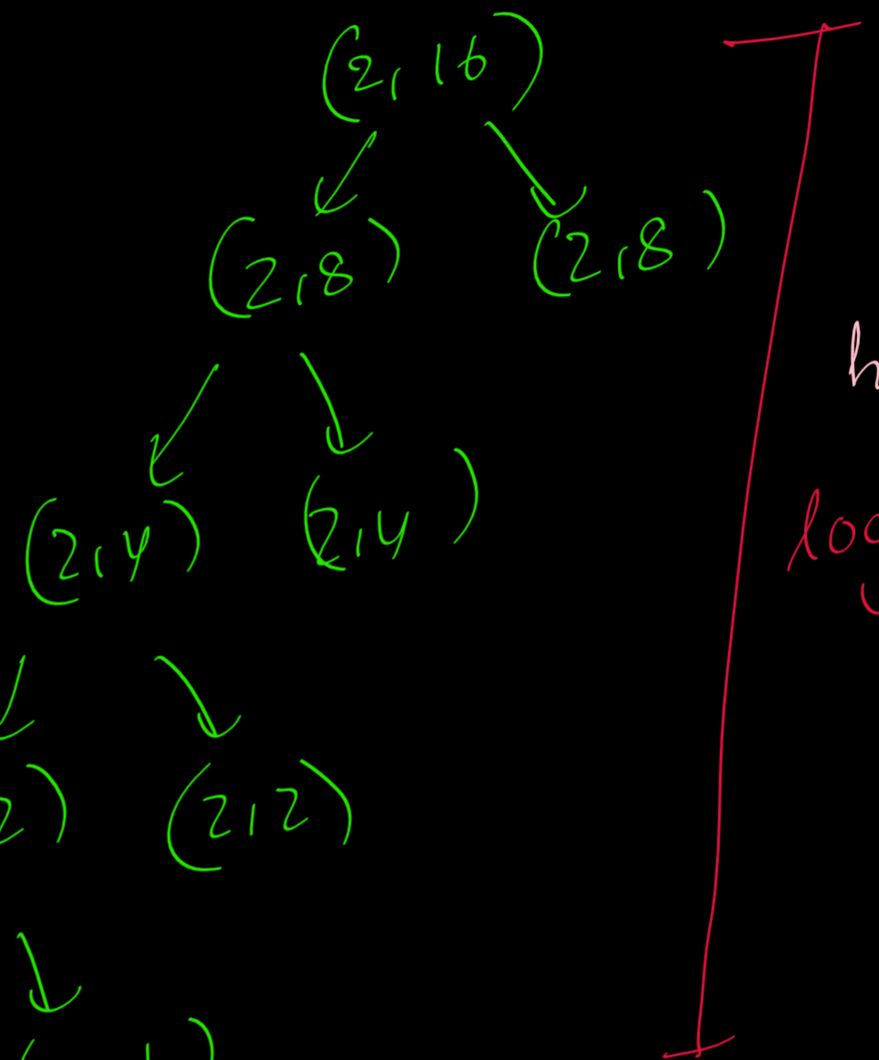
$(1)^{\log_2 N} + (k_1 + k_2) \log_2 N$

$O(\log_2 N)$

```java
public static int power(int x, int n) {
  if(n == 0){
    return 1;
  }
  int xpn = power(x, n/2) * power(x, n/2);

  if(n % 2 == 1){
    xpn = xpn * x;
  }

  return xpn;
}
```

L R

No of calls = 2



$\log_2 N$

$(2) + (k+k)(\log_2 N)$

$= (2)^{\log_2 N} + (2k)(\log_2 N)$

$= N + 2k\log_2 N$

$O(N)$

→ base case ignore

Print Zig Zag

```java
public class PrintZigZag {
    public static void main(String[] args) throws Exception {
        // write your code here
        Scanner scn = new Scanner(System.in);
        int n = scn.nextInt();
        scn.close();
        pzz(n);
    }

    public static void pzz(int n){
        if(n == 0) return;

        System.out.print(n + " ");
        pzz(n-1);
        System.out.print(n + " ");
        pzz(n-1);
        System.out.print(n + " ");
    }
}
```

Decreasing function

→ O(1)

Preorder = O(1)

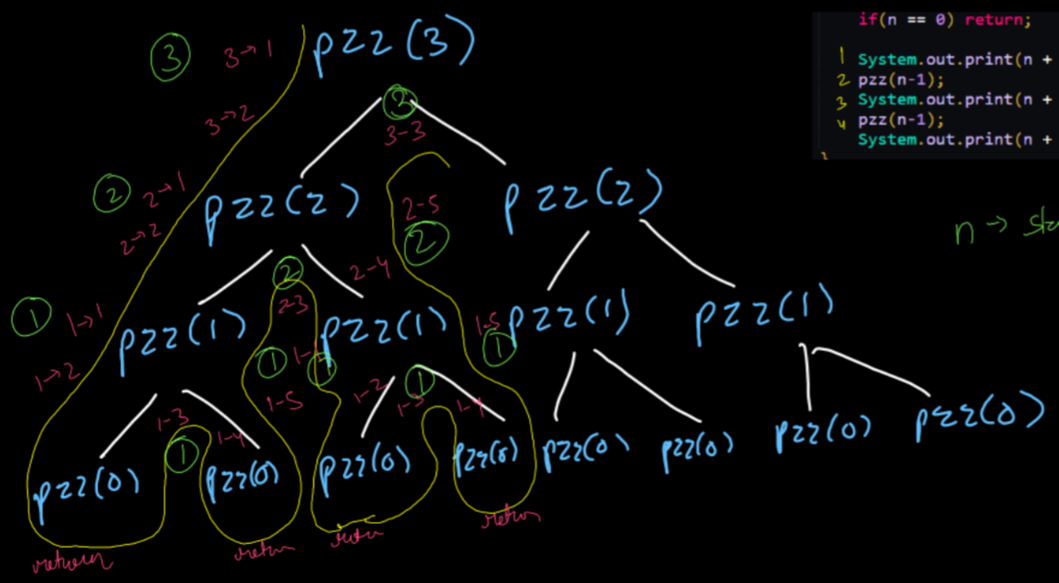
Postorder = O(1)

Calls = 2

Height = N

$(2)^N + (k_1 + k_2) * N$

$= 2^N + N = O(2^N)$




```java
if(n == 0) return;

System.out.print(n + " ")
pzz(n-1);
System.out.print(n + " ")
pzz(n-1);
System.out.print(n + " ")
```

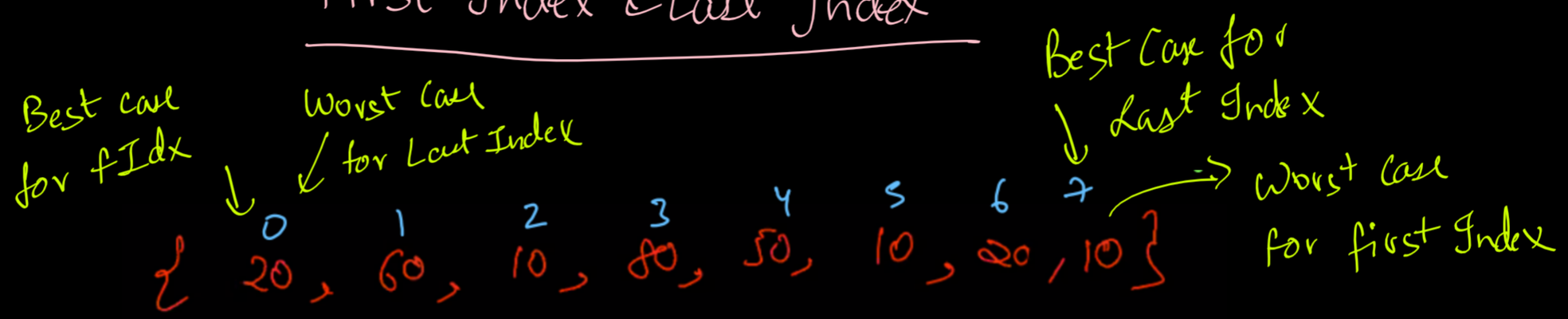
n → state

→ Height is N

** Height of dec fun (generally) = O(N)

Height of dividing fun (generally) = O(log N)

first Index & Last Index



Best Case : O(1)

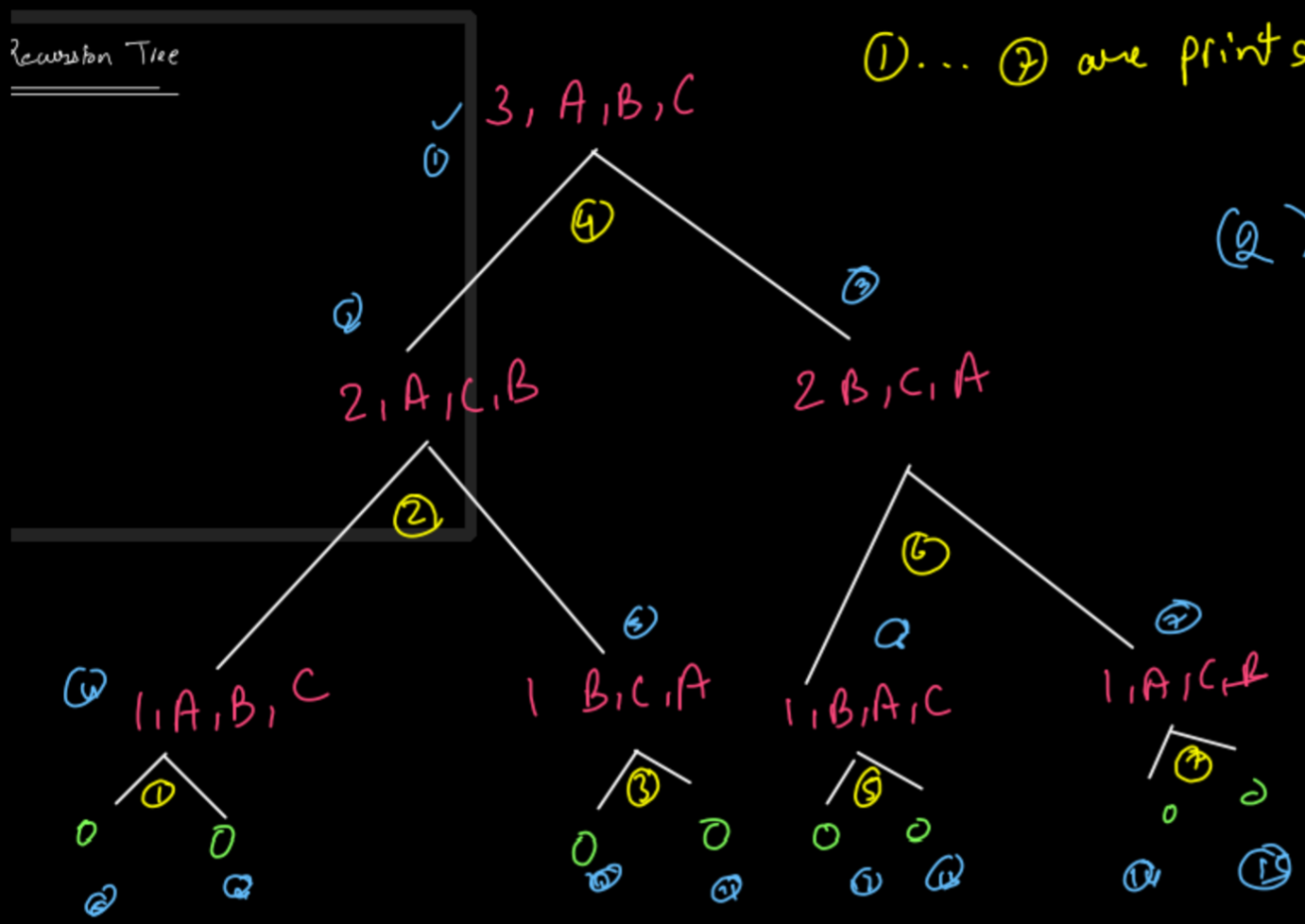
Worst Case : O(N)

Avg Case : O(N)

} for both first Index & last Index

(Though Complexities of first Idx & last Idx for every case is same, the cases however are different)

Tower of Hanoi



① ... ⑦ are print stmt

$(2)^n$

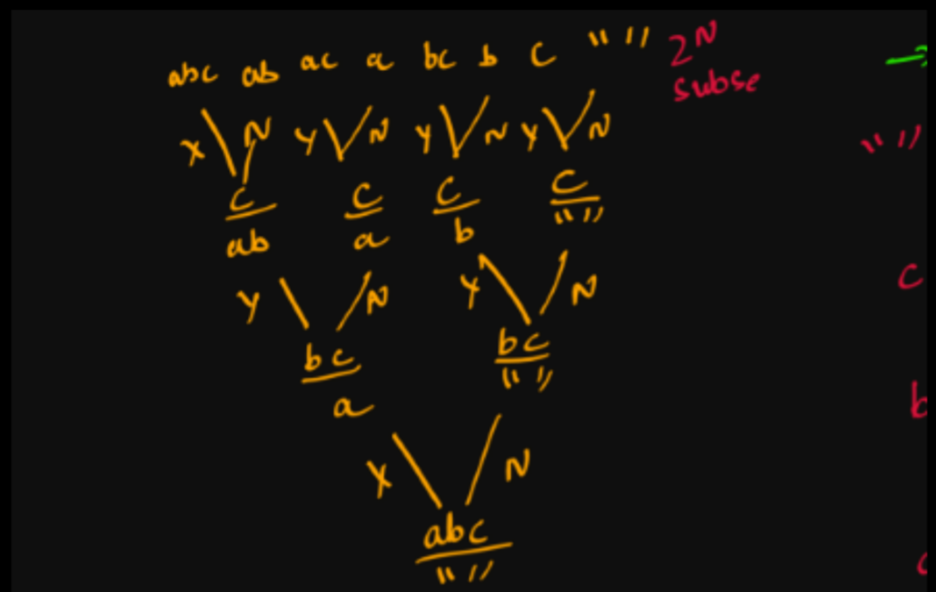
No of calls ≤ 2

Height $\leq N$

$(2)^N + (0) * N$

$= O(2^N)$

Print Subsequences




```java
public static void main(String[] args) throws Exception {
    Scanner scn = new Scanner(System.in);
    String str = scn.next();
    printSS(0,str,"");
}

public static void printSS(int idx,String str, String ans) {

    if(idx == str.length()) {
        System.out.println(ans);
        return;
    }

    printSS(idx + 1,str,ans + str.charAt(idx)); //yes call
    printSS(idx + 1,str,ans); //no call
}
```

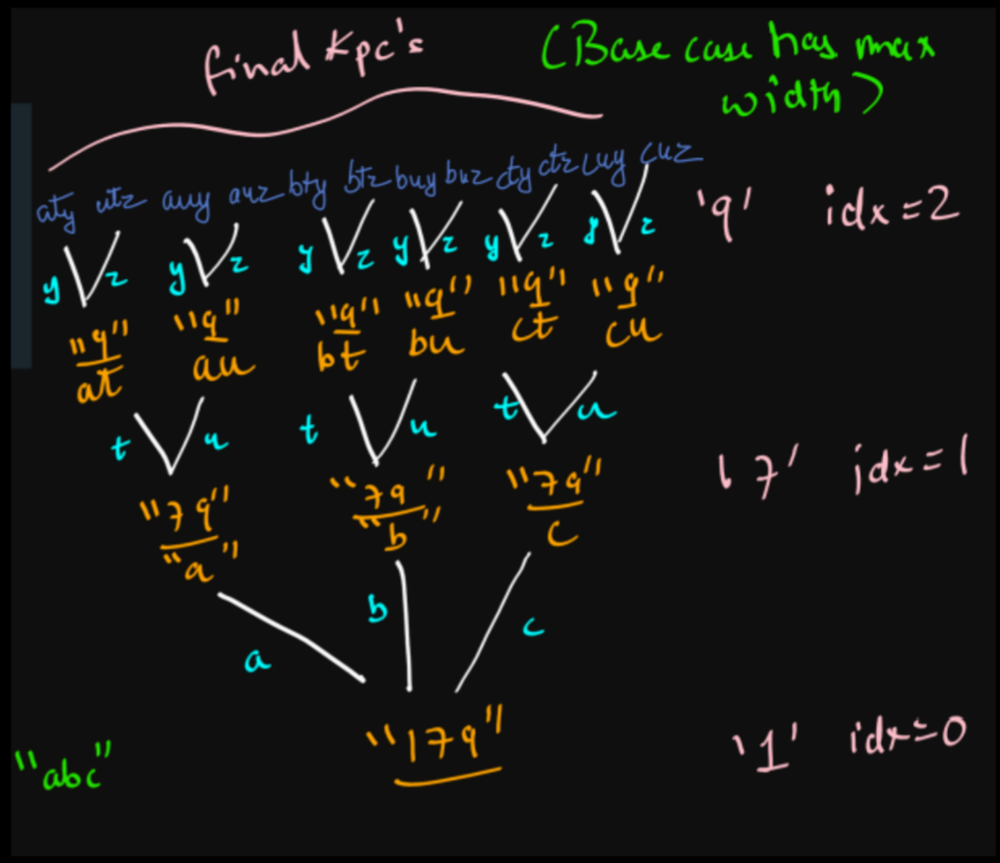
$$(2)^N + (0) * N$$

$$= O(2^N)$$

Print KPC

```java
public static void printKPC(int idx,String input, String output) {

    if(idx == input.length()) {
        System.out.println(output);
        return;
    }

    String str = keys[input.charAt(idx) - '0'];
    for(int i=0;i<str.length();i++) {
        printKPC(idx + 1,input,output + str.charAt(i));
    }
}
```

0 -> .;
1 -> abc
2 -> def
3 -> ghi
4 -> jkl
5 -> mno
6 -> pqrs
7 -> tu
8 -> vwx
9 -> yz



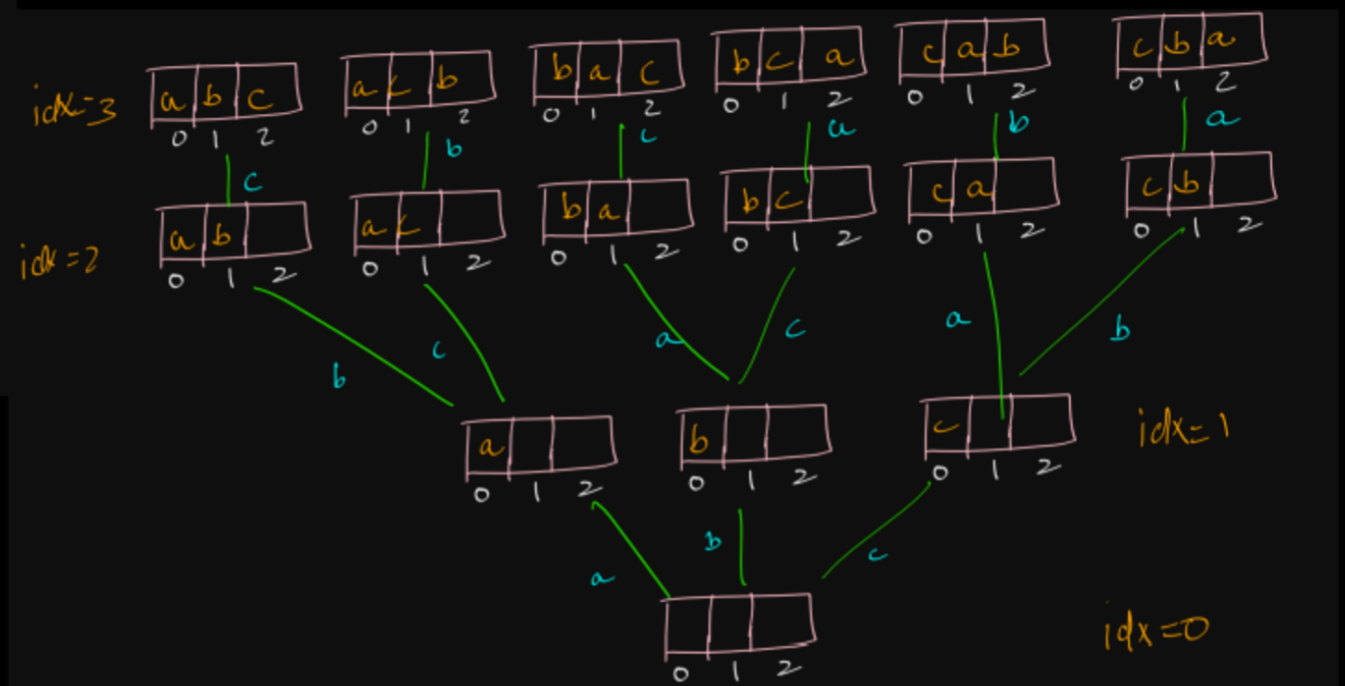
-> So, max calls can be 4.

$$(calls)^h + (pre + post) * h$$

$$(4)^N + (0) * N \leq O(4^N)$$

Print Permutations

```java
public static void printPermutations(String str, String asf)     {
    if(str.length() == 0) {
        System.out.println(asf);
        return;
    }

    for(int i=0;i<str.length();i++) {
        String newInput = str.substring(0,i) + str.substring(i+1);
        printPermutations(newInput,asf + str.charAt(i));
    }
}
```



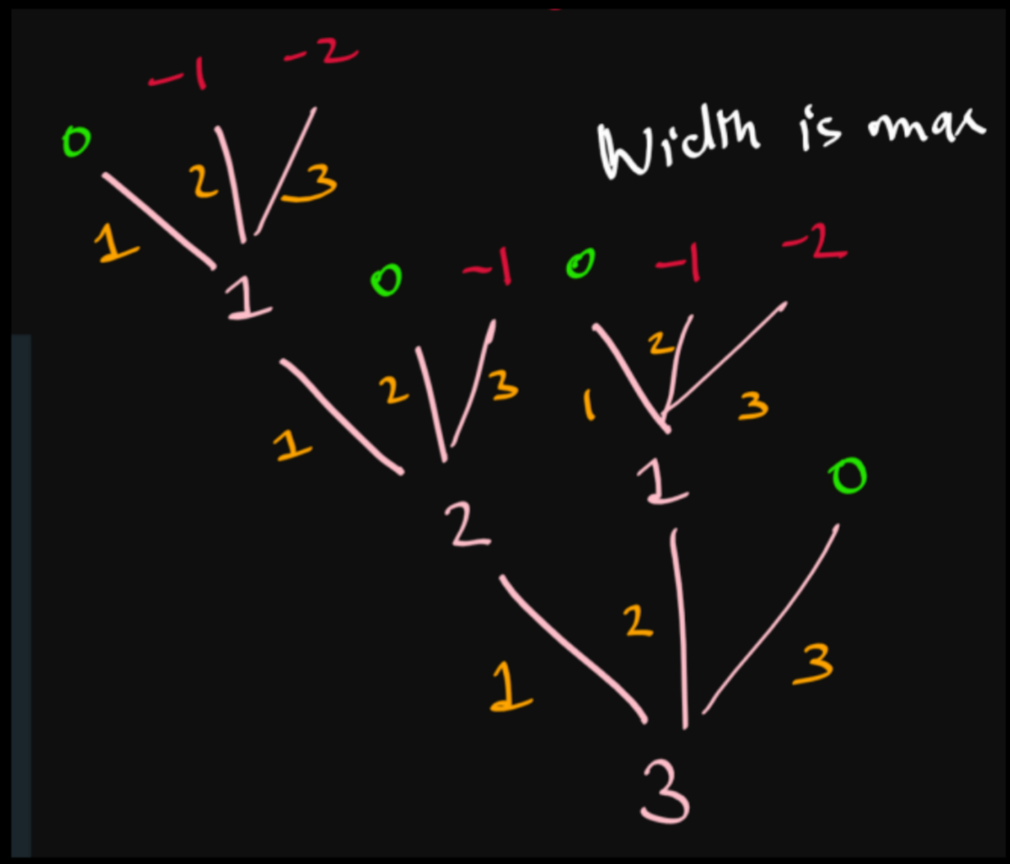
$$TC = (calls)^{height} + (pre + post) * height$$

$$= (N)^N + (k) * N$$

$$\approx O(N^N) \approx O(N!)$$

Print Stair Paths

```java
public static void printStairPaths(int n, String path) {

    if(n < 0) {
        return;
    }

    if(n == 0) {
        System.out.println(path);
        return;
    }

    printStairPaths(n-1 ,path + "1");
    printStairPaths(n-2 ,path + "2");
    printStairPaths(n-3 ,path + "3");
}
```



Calls = 3

height = N

Pre → say k

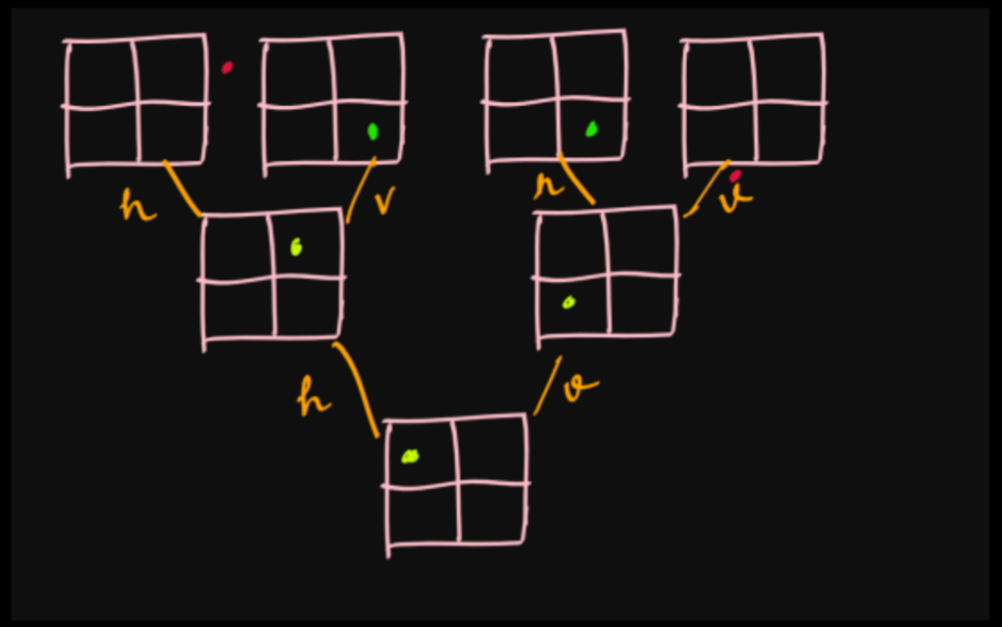
post → 0

$T(n) = (3)^N + (k+0) * N$

$= O(3^N)$

Print Maze Paths

```java
// sr - source row
// sc - source column
// dr - destination row
// dc - destination column
public static void printMazePaths(int sr, int sc, int dr, int dc, String psf) {

    if(sr > dr || sc > dc) {
        return;
    }

    if(sr == dr && sc == dc) {
        System.out.println(psf);
        return;
    }

    printMazePaths(sr,sc + 1,dr,dc,psf + "h");
    printMazePaths(sr + 1,sc,dr,dc,psf + "v");
}
```



No of calls = 2

height = m+n-2 ≈ m+n

↓

* Length of path = rows - 1 + cols - 1 (Greedy)
* No of paths = $\frac{(rows-1+cols-1)!}{(rows-1)!\,(cols-1)!}$ (DP)

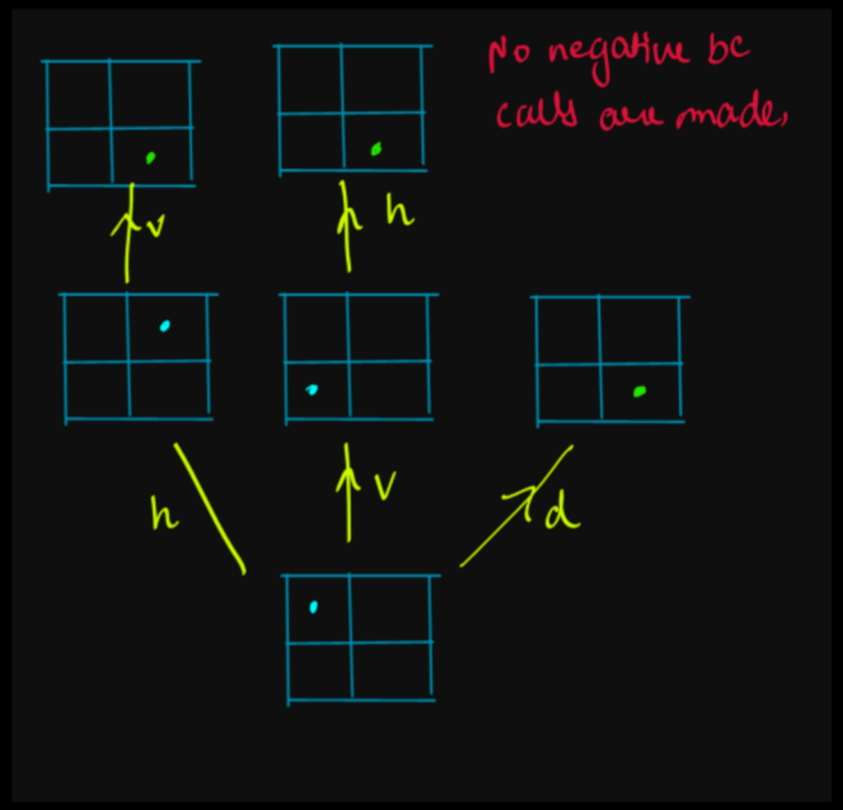
$TC = (2)^{m+n} + (k) * (m+n)$

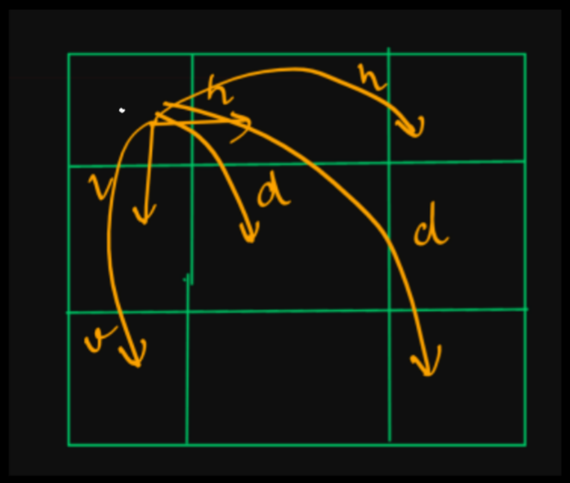
$TC = (2)^{m+n}$

$O(2^{m+n})$

Print Mare Paths With Jumps

```java
// sr - source row
// sc - source column
// dr - destination row
// dc - destination column
public static void printMazePaths(int sr, int sc, int dr, int dc, String psf) {

    if(sr == dr && sc == dc) {
        System.out.println(psf);
        return;
    }

    int jump = 1;

    while(sc + jump <= dc) {
        printMazePaths(sr,sc+jump,dr,dc,psf + "h" + jump);
        jump++;
    }

    jump = 1;
    while(sr + jump <= dr) {
        printMazePaths(sr+jump,sc,dr,dc,psf + "v" + jump);
        jump++;
    }

    jump=1;
    while(sr + jump <= dr && sc + jump <= dc) {
        printMazePaths(sr+jump,sc+jump,dr,dc,psf + "d" + jump);
        jump++;
    }
```





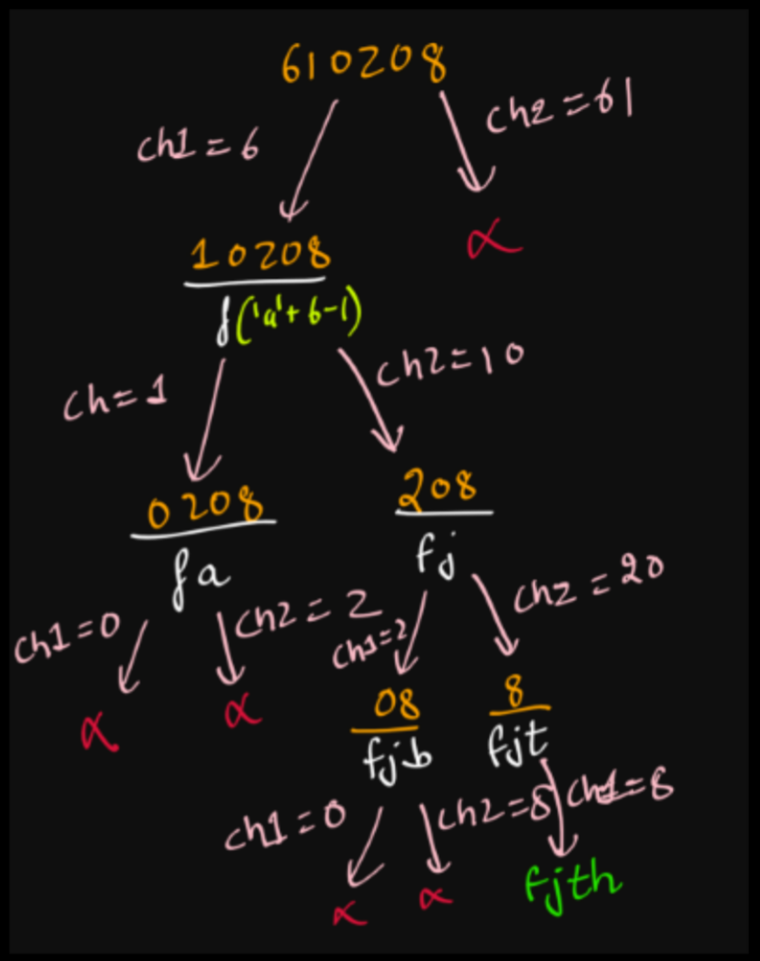
calls = $n-1+m-1+\min(m,n) \approx O(n+m) \approx n+m$

height = $(n+m)$

$T(n) = (n+m)^{(n+m)} \approx O((n+m)^{n+m}) \approx O((n+m)!)$

Print Encoding

```java
public static void printEncodings(int idx,String input,String output) {
    if(idx == input.length()) {
        //positve base case
        System.out.println(output);
        return;
    }

    int ch1 = input.charAt(idx) - '0';
    if(ch1 >=1 && ch1<=9) {
        printEncodings(idx + 1,input, output + (char)('a' + ch1 - 1));
    }

    if(idx + 1 < input.length()) {
        int ch2 = (input.charAt(idx) - '0') * 10 + (input.charAt(idx+1) - '0');

        if(ch2 >= 10 && ch2<= 26) {
            printEncodings(idx + 2,input,output + (char)('a' + ch2 -1));
        }
    }
}
```



calls = 2

height = n

$$TC = (2)^n$$

flood fill

Worst Case Height : n*m (all 0's wave path)

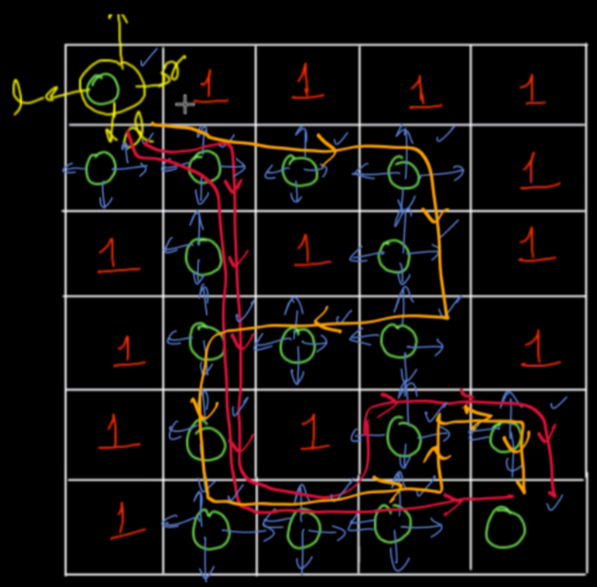
Calls = 4

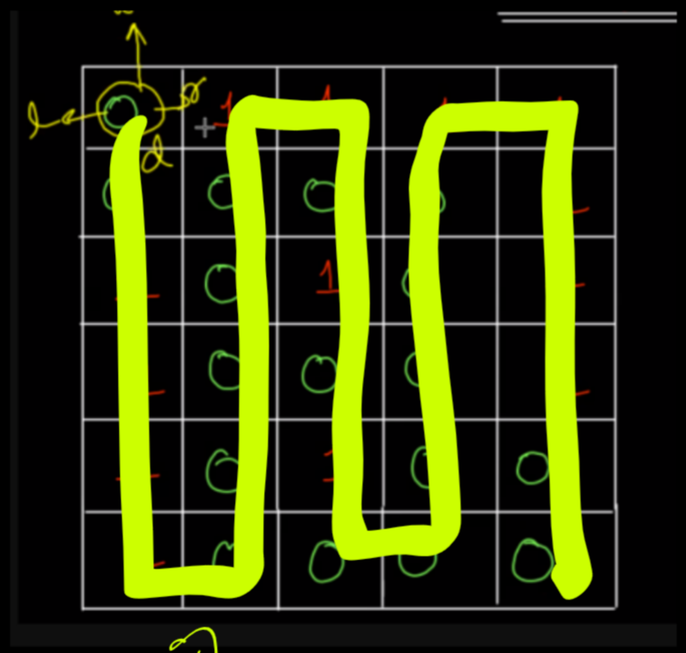
$$TC = (4)^{n*m} + (preorder) * (n*m)$$

can be ignored wrt 4^{n*m}

(it is approx $O(n+m)$)

$$TC = O(4^{nm})$$





↑ worst case height

if all 0's

N Queens

$TC = (columns)^{rows}$

max height = rows (place each queen at each row)

calls = no of columns (place queen in each row at every call)

Recurrence Relation

Power Linear

```java
import java.util.*;
public class PowerLinear {
    public static void main(String[] args) throws Exception {
        // write your code here
        Scanner scn=new Scanner(System.in);
        int n=scn.nextInt();
        int p=scn.nextInt();
        scn.close();
        int ans=power(n,p);
        System.out.println(ans);
    }

    public static int power(int x, int n){

        if(n==0)
        {
            return 1;
        }

        return x*power(x,n-1);
    }
}
```

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

$p(x,n) = p(x,n-1) + \text{constant}$

↑ Expectation ↑ faith ↑ constant work

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

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

$T(n-2) = T(n-3) + k$

⋮

$T(1) = T(0) + k$

n times

$$T(n) = T(0) + nk$$

$$T(n) = k_1 + nk \approx O(n)$$

Power Logarithmic

```java
public class PowerLogarithmic {
    public static void main(String[] args) {
        Scanner scn=new Scanner(System.in);
        int n=scn.nextInt();
        int p=scn.nextInt();
        scn.close();
        int ans=power(n,p);
        System.out.println(ans);
    }

    public static int power(int x, int n) {

        if(n == 0) return 1;

        int pxnby2 = power(x, n/2); //call this only once else it will be O(n) only

        if(n%2 == 0) {
            return pxnby2*pxnby2;
        }

        return x*pxnby2*pxnby2;
    }
```

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

Expectation — faith — Meeting Exp (constant work)

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

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

$T(n/4) = T(n/8) + k$

$T(n/8) = T(n/16) + k$

$\vdots$

$T(1) = T(0) + k$

$T(n) = T(0) + \log_2 N \, k$

$T(n) = O(\log_2 N)$

Power Third Method

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

* Solve further

```java
public static int power(int x, int n) {
  if(n == 0){
    return 1;
  }
  int xpn = power(x, n/2) * power(x, n/2);

  if(n % 2 == 1){
    xpn = xpn * x;
  }

  return xpn;
}
```

L R

HW

4. $T(n) = T(n-1) + O(n)$
5. $T(n) = T(n-1) + O(\log_2 N)$
6. $T(n) = 2T(n-1) + O(1)$
7. $T(n) = T(n/2) + O(N)$
8. $T(n) = T(\sqrt{N}) + O(1)$ {Root-function}

Recurrence Relation ?

Space Complexity

- Auxiliary Space (Extra Space)
  - Extra DS jo solve krne k liye bnaya hai jo na hume diye tha na humse mnaanga tha na he ye recursion call stack ki space hai.
- Input & Output Space
  - LL ki Graph ki tree ki apni space
- Recursion call Stack (Rec Overhead) Space (= height of tree / max height of call stack)
  - Pushing fun upon another in stack is very costly operation coz we have to store addresses, registers & much more.
  - → context switching

** Constant Extra Space

- primitive data types
- arr [26] → also constant

Leetcode adds up All auxiliary + I/O + Call stack space

Space Time trade-off

General Trend:

- Space ↑ Time ↓
- Space ↓ Time ↑ (This is more imp nowadays)