Dynamic Programming :

- ① Recursion
- ② Memoisation
- ③ Tabulation

D.P.

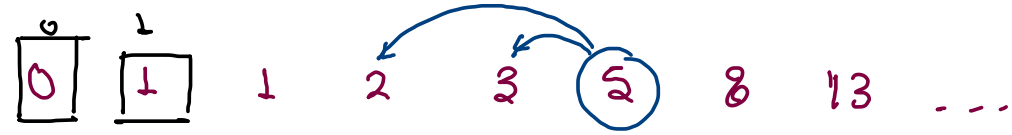
Recursion → Repitition of same problem,

→ In the recursion we are trying to solve a same problem again and again

→ Sacrifice with space } store the repitition of problem
and save time.

① Fibbonacci - series -] D.P.
Memoisation
Tabulation

Fibonacci - with Recursion



if(n==0 || n==1) return n;

```
public static int fib_rec(int n) {

    int fibnm1 = fib_rec(n - 1);
    int fibnm2 = fib_rec(n - 2);

    int fibn = fibnm1 + fibnm2;

    return fibn;
}
```

order of n

fib(5) fib(4) fib(3) fib(2) fib(1) fib(0) fib(1) fib(2) fib(1) fib(0) fib(1) fib(3) fib(2) fib(1) fib(0) fib(1) fib(3) fib(2) fib(1) fib(0) fib(1)

1+0=1

$2T(n-2) \leq T(n) \leq 2T(n-1)$

$T(n) \equiv O(2^n)$

Time: $O(2^n)$

Repitition point

→ Base case

→ fib(2)

→ fib(3)

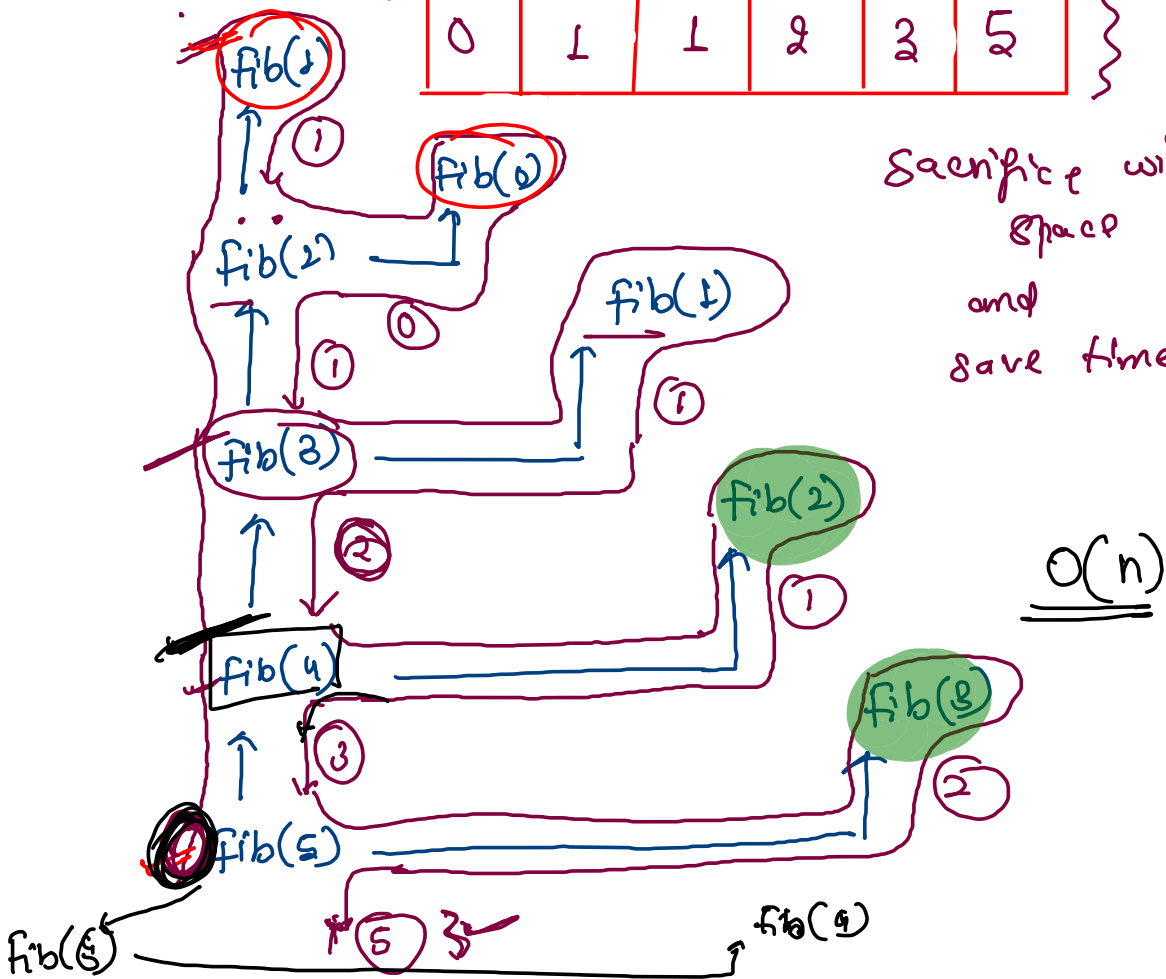
Fibbonacci - Memoisation

n = 5.

Size = n+1

dp →

0	1	2	3	4	5
0	1	1	2	3	5

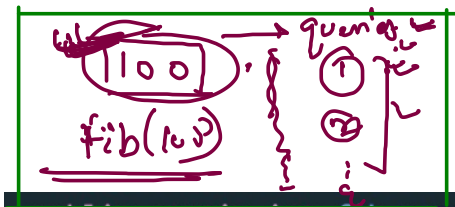


Sacrifice with space and save time.

O(n)

① DP: Storage to store the problem.

↳ Size of storage (n+1)




```java
public static int fib_memo(int n, int[] dp) {
    if(n == 0 || n == 1) {
        return dp[n] = n;
    }
    // 1. if Problem is already solved, then r
    if(dp[n] != 0) {
        return dp[n];
    }

    int fibnm1 = fib_memo(n - 1, dp);
    int fibnm2 = fib_memo(n - 2, dp);

    int fibn = fibnm1 + fibnm2;
    // 2. If not solved, then solve the proble
    return dp[n] = fibn;
}
```

Smallest problem

Fibbonacci - with Tabulation

Steps: (1) figure out repitition.

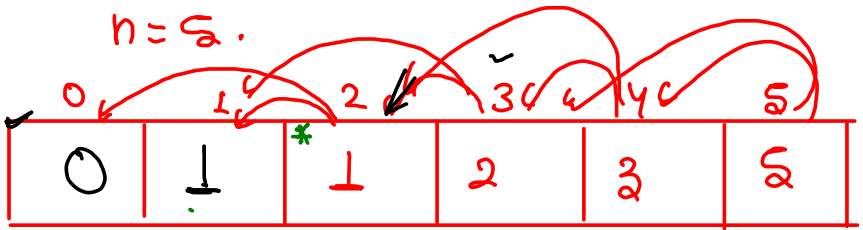
(2)* make a strage of n+1 size

(3) assign meaning to cell

(4) figure out the direction of smallest to largest problem

(4) Pre requisite marking on dp.] Base case of recursion

Repitition is on single variable.



Meaning of cell → * → dp[i] → fibonacci(i) = fib(i-1) + fib(i-2)

recurrence relation: dp[i] = dp[i-1] + dp[i-2]

dp[0] = 0

dp[1] = 1

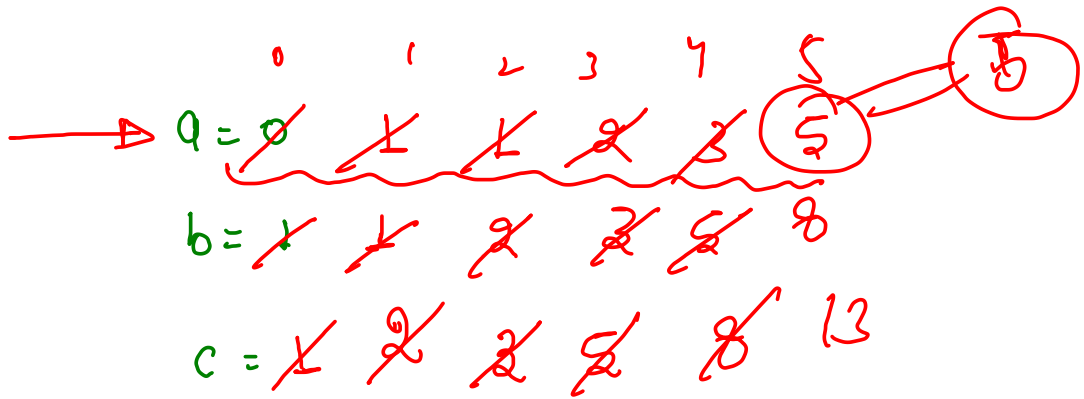
Smallest problem.

Largest problem.

direction of solving the problem.

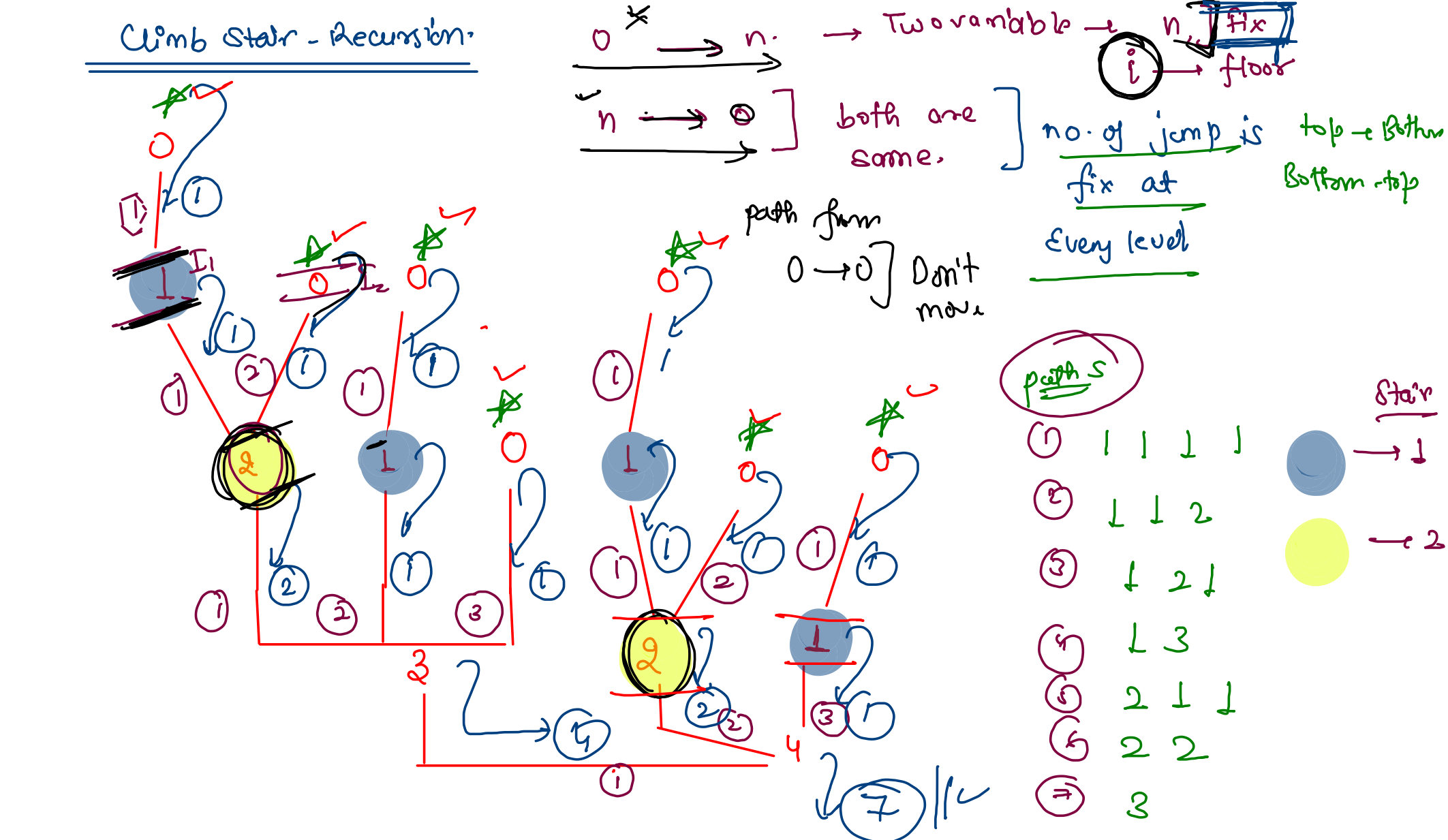
final result → dp[n]

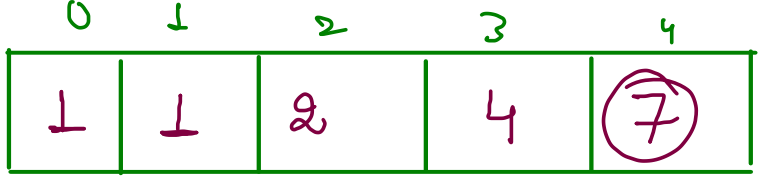
Fibonacci_btr :→ n = 5.



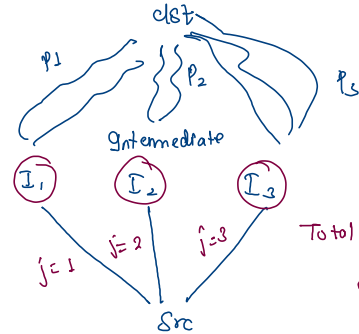

```
c = a+b;
a = b;
b = c;
```

Climb Stair - Recursion.

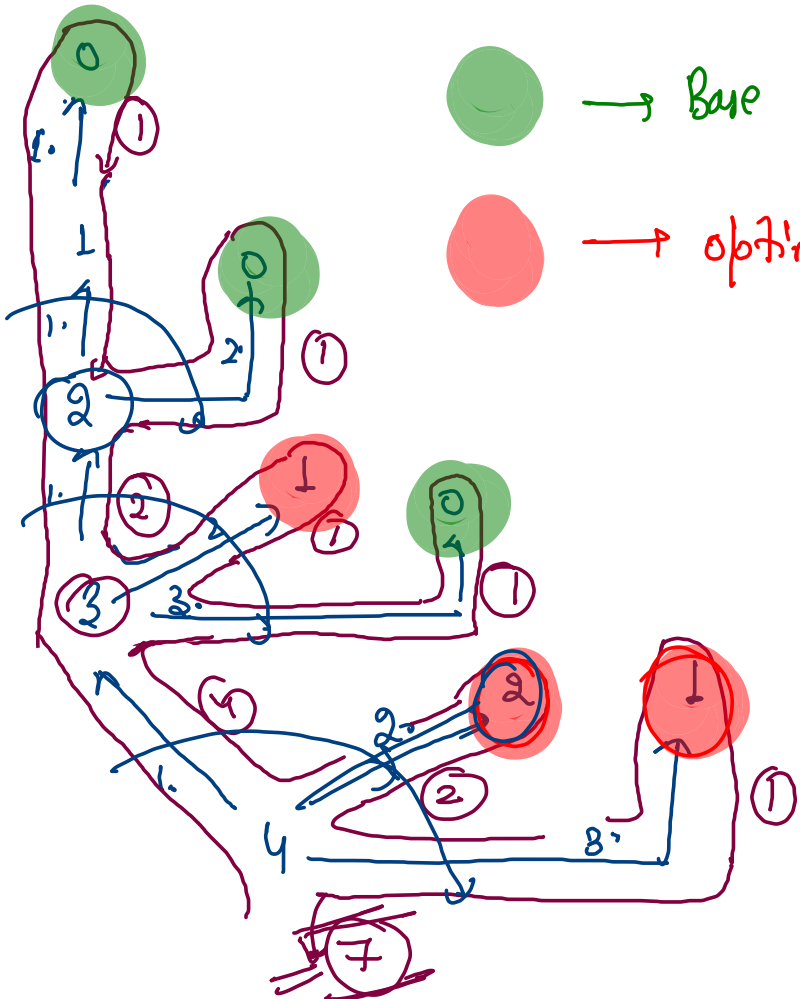




order of problem.
0 →



Total path from src to dst :-
Src → dst → P1 + P2 + P3



→ Base case.

→ optimisation

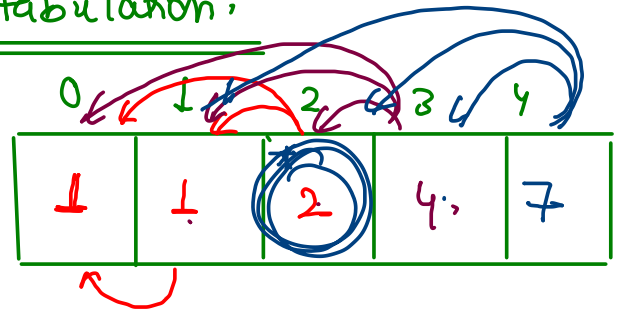
n = 4

```java
public static int climbStair_memo(int n, int[] dp) {
    if(n == 0) {
        return dp[0] = 1;
    }

    if(dp[n] != 0) {
        return dp[n];
    }

    int count = 0;
    for(int j = 1; j <= 3; j++) {
        if(n - j >= 0) {
            count += climbStair_memo(n - j, dp);
        }
    }
    return dp[n] = count;
}
```

Climbstair - tabulation.



(1) Storage.

(2) Meaning of cell

(3) Direction of traversal → smallest problem is at 0

(4) prerequisite.

result at - dp[4] → 7

largest is at 4

dp[0] = 1 → there is one path from 0 → 0 i.e. don't move.

* dp[i] → count of path from i^{th} src to destination(0)

jump / path

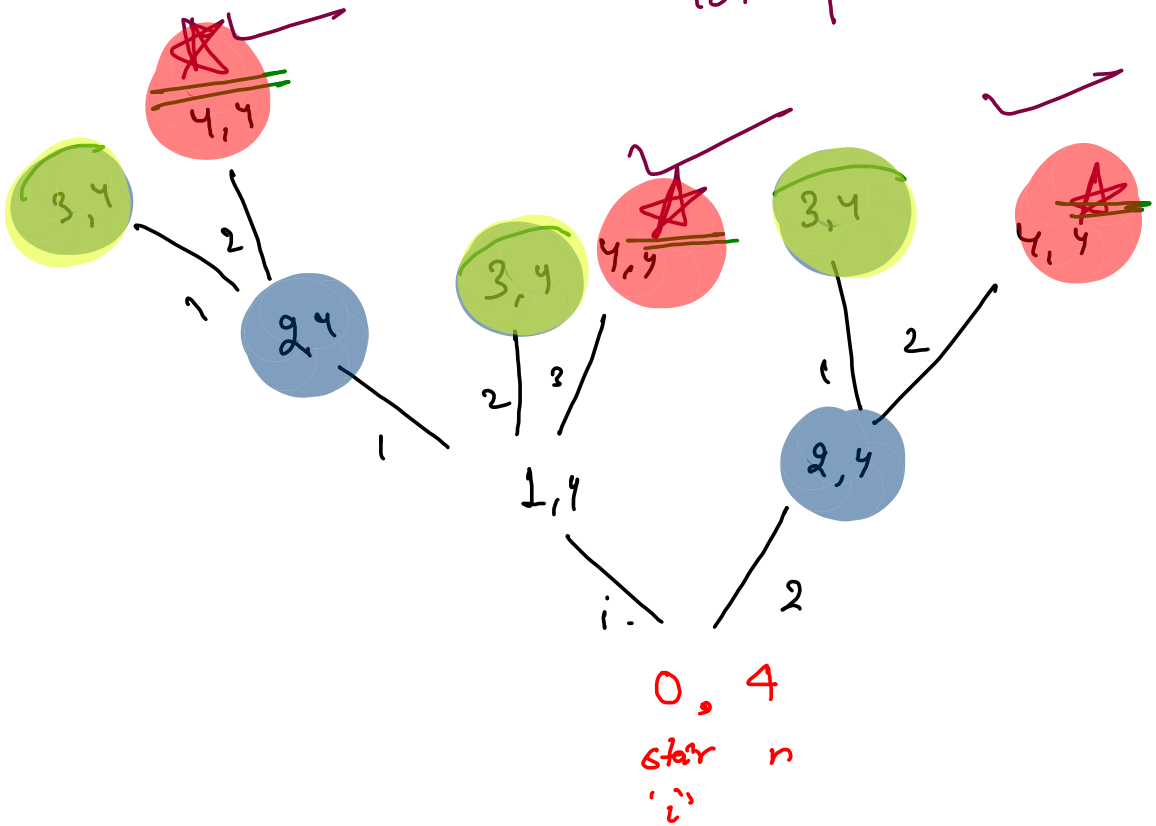
ClimbStairs with Variable Jumps.

0 → 4. Total no. of path.

0	1	2	3	4.
2	3	2	0	1

variables which participate in repitition is basically number of stair

Total path = 3



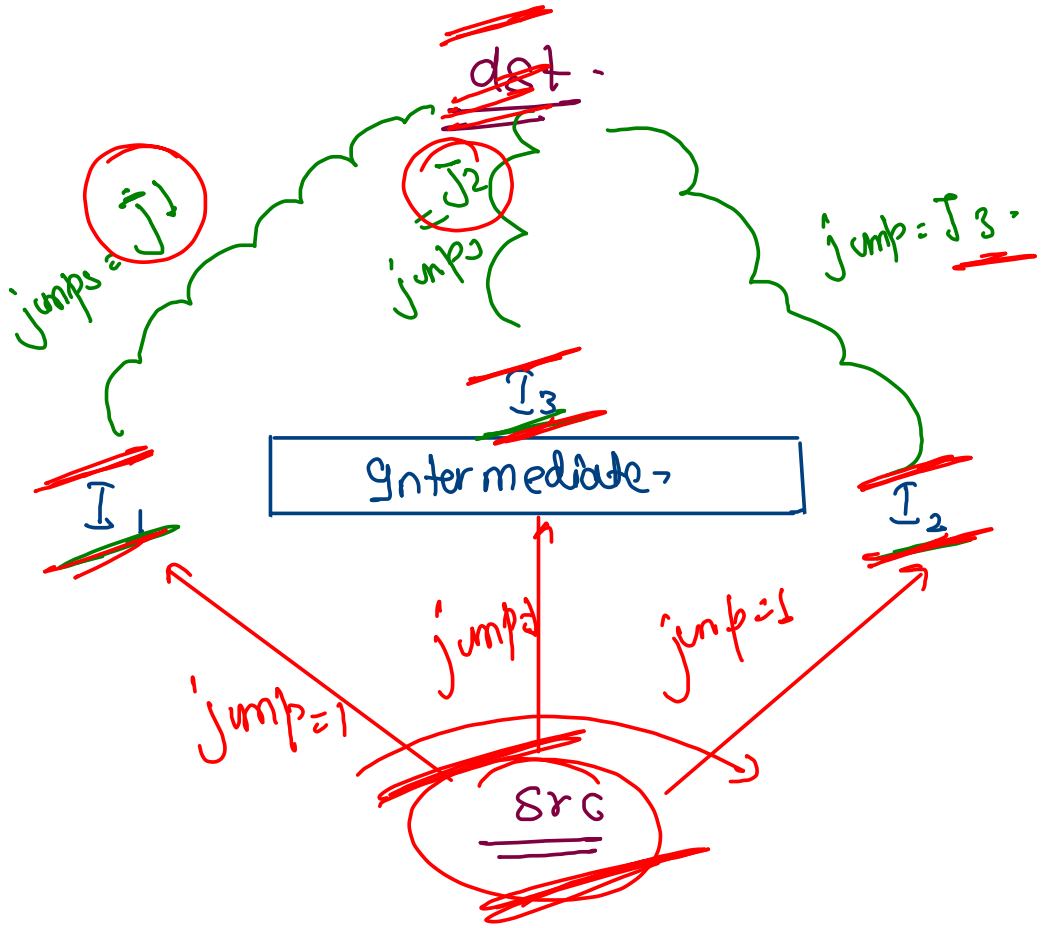
→ Base case

→ 3rd stair

→ 2nd stair.

moves = jumps.

path.



src → dst (-min jumps)

min (J1, J2, J3) + 1

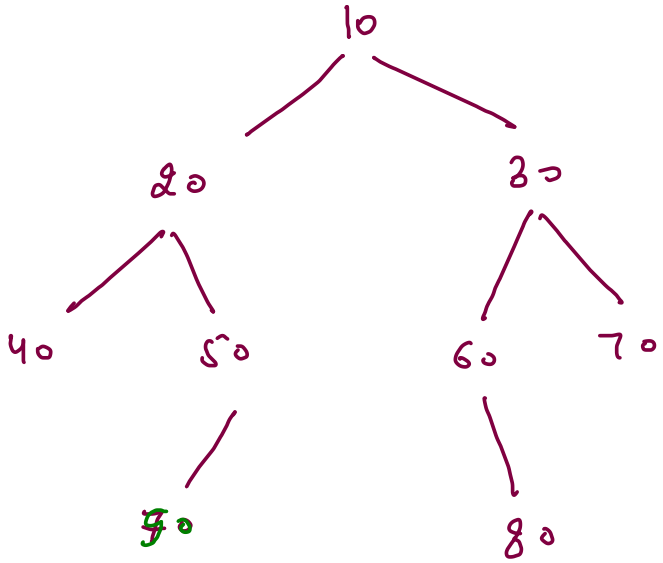
single path

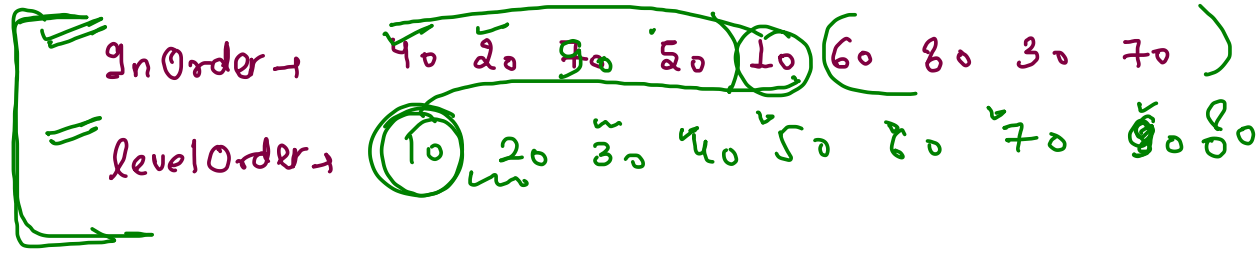
moves = J

jumps.

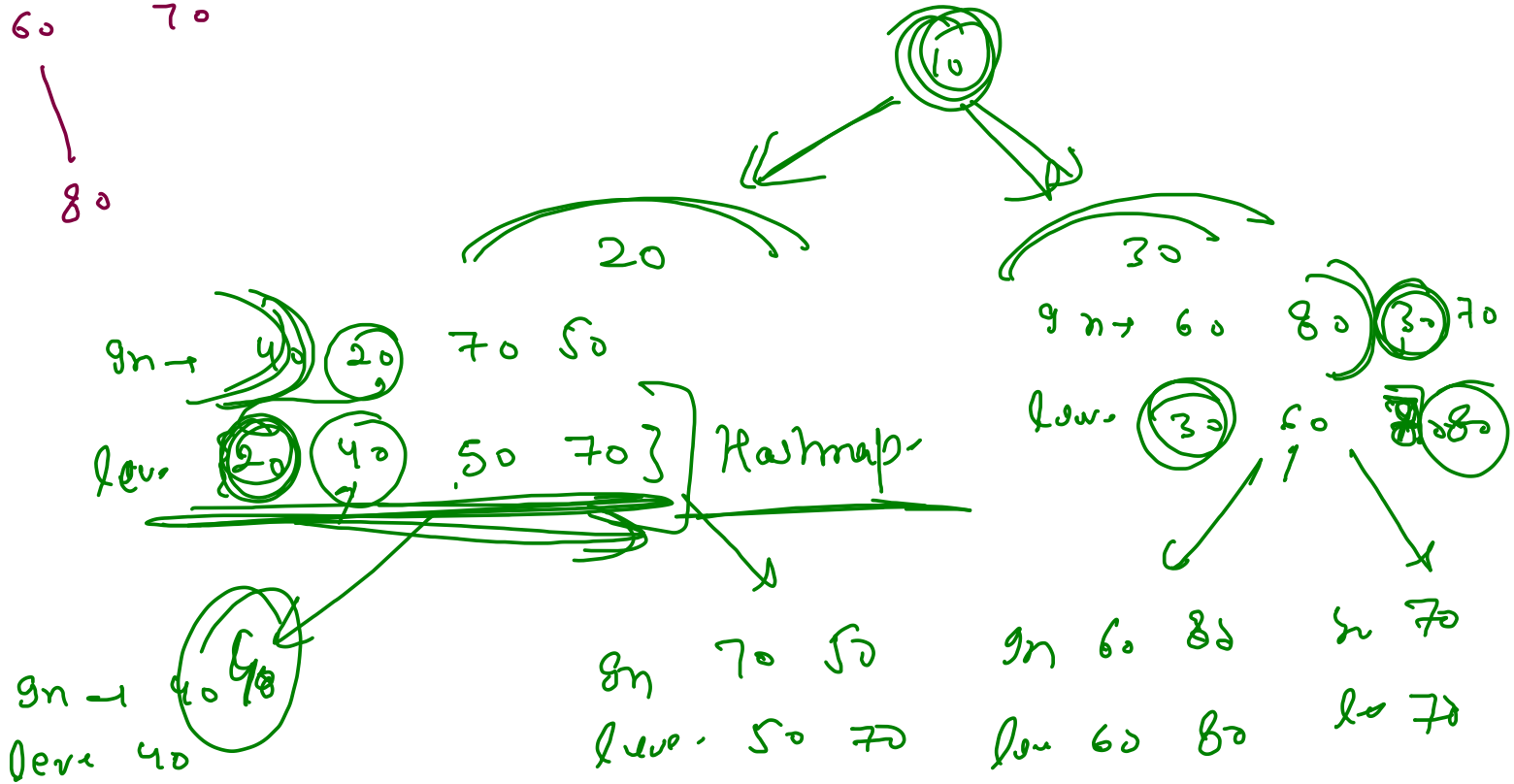
dst

src.


10
20
30
40
50
60
70
80


InOrder →
LevelOrder →

Binary Tree -


20
30
In →
lev.
Hashmap
In
lev.
In
lev.
In
lev.