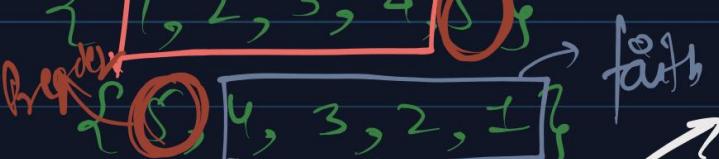


Recursion - level - 1 Revision

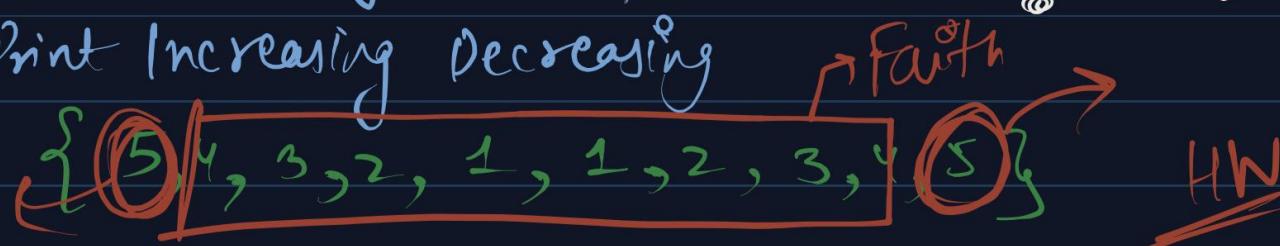
- Print Increasing



- Print Decreasing



- Print Increasing Decreasing



$$U^N + \{k^k\}^{N+k} = O(N)$$

①

Expectation : $\text{PI}(N) \rightarrow 1, 2, \dots, N$

Faith : $\text{PI}(N-1) \rightarrow 1, 2, 3, \dots, N-1$

Meeting Expectation : $\text{Syso}(N)$

Preorder

$\text{Syso}(N)$

Postorder

$\text{PI}(N-1)$

Preorder

Sys₀(n)

P I(N-1)

Postorder

P I(N-1)

Sys₀(n)

```
public class Main {
    public static void main(String[] args) throws Exception {
        Scanner scn = new Scanner(System.in);
        int n = scn.nextInt();
        printIncreasing(n);
    }

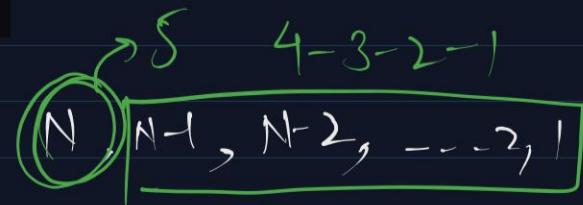
    public static void printIncreasing(int n){
        if(n == 0) return; // Base Case

        printIncreasing(n - 1); // Faith
        System.out.println(n); // Meeting Expectation with Faith
    }
}
```

Call → 1
height → N

$$(1)^N + \{0+k\}^N \Rightarrow \underline{\underline{O(N)}}$$

② Expectation: → P D(N) :→



Faith: → P D(N-1) :→ N-1, N-2, ... 1

Meeting Expectation: → Preorder: Sys₀(n)

$$\left. \begin{array}{l} \text{calls} = 1, \text{height} = N \\ (1)^N + \{k+0\}^N \\ = \underline{\underline{O(N)}} \end{array} \right\}$$

```
public static void main(String[] args) throws Exception {
    Scanner scn = new Scanner(System.in);
    int n = scn.nextInt();
    printDecreasing(n);
}

public static void printDecreasing(int n){
    if(n == 0) return;
    System.out.println(n);
    printDecreasing(n - 1);
}
```

Power Function

~~E~~

Expectation $\mapsto x^n \xleftarrow{\text{pow}(x,n)}$

$$x = 2.0, n = 5 \therefore 2^5 = 32$$

Faith $\rightarrow x^{n-1} * x$
 $\text{pow}(x, n-1)$ Meeting expectation

Generic Time Complexity of Recursion

\hookrightarrow $(\text{calls})^{\text{height}} + \{\text{Preorder} + \text{Postorder}\} * \text{height}$

Generic Time Complexity of Recursion

$$\text{Calls}^{\text{height}} + \{\text{Preorder+Postorder}\} * \text{height}$$

```

public double power(double x, int n){
    if(n == 0) return 1.0;
    double pxn1 = power(x, n - 1); // Faith
    return pxn1 * x; // Meeting Expectation
}

public double myPow(double x, int n) {
    if(x == 0) return 0.0;
    if(x == 1) return 1.0;

    if(n < 0){
        return 1.0 / power(x, -n);
    }

    return power(x, n);
}

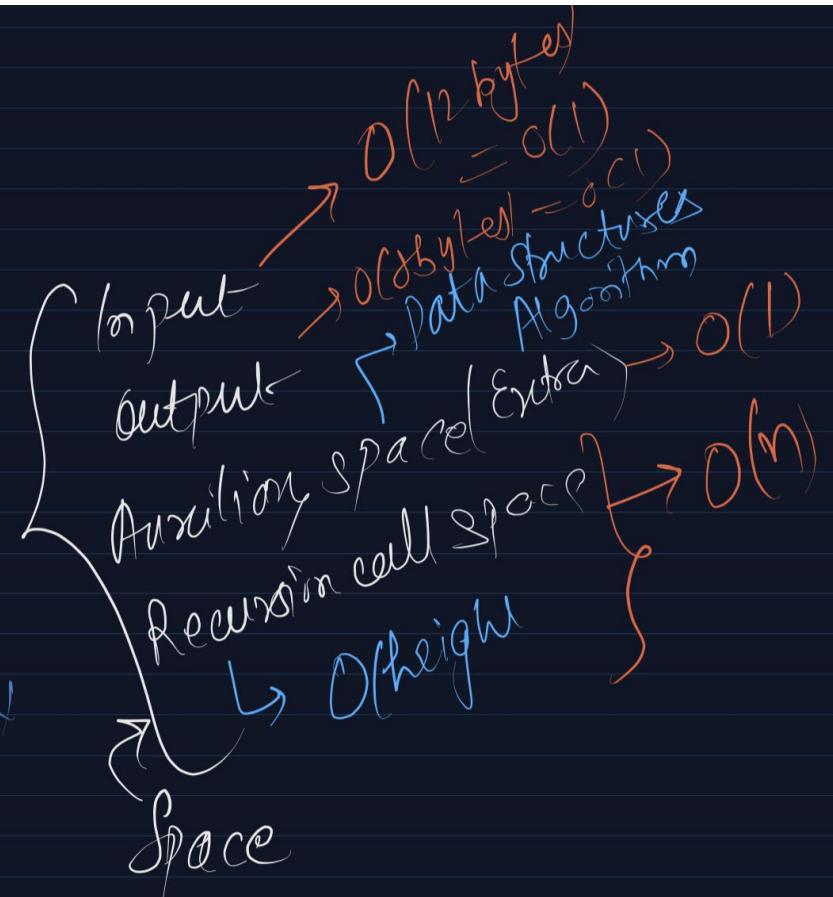
```

Brute force

$$\begin{aligned} \text{Calls} &= 1 \\ \text{height} &= N \end{aligned}$$

$$1^N + \{k+k\}^N$$

$$\approx \underline{\underline{O(N)}}$$



Power → Optimized

$$\begin{cases} \text{even} \\ \cancel{\text{odd}} \end{cases} \left\{ \begin{array}{l} x^n = x^{n/2} * x^{n/2} \\ 2^6 = 2^3 * 2^3 = 2^{3+3} = 2^6 \end{array} \right.$$

$$\left\{ \begin{array}{l} \text{odd} \\ \cancel{\text{even}} \end{array} \right. \left\{ \begin{array}{l} x^n = x^{n/2} * x^{n/2} * x \\ 2^7 = 2^3 * 2^3 * 2 = 2^{6+1} = 2^7 \end{array} \right.$$

```
class Solution {
    public double power(double x, int n){
        if(n == 0) return 1.0;

        if(n % 2 == 0)
            return power(x, n/2) * power(x, n/2); // Meeting Expectation
        else
            return power(x, n/2) * power(x, n/2) * x;
    }

    public double myPow(double x, int n) {
        if(x == 0) return 0.0;
        if(x == 1) return 1.0;

        if(n < 0){
            return 1.0 / power(x, -n);
        }

        return power(x, n);
    }
}
```

calls $\Rightarrow 2^{\log n}$ breadth

height $\Rightarrow \log n$

depth

$$\Rightarrow O(2^{\log n}) = O(n)$$

Input $\rightarrow O(1)$
Output $\rightarrow O(1)$
Extra $\rightarrow O(1)$

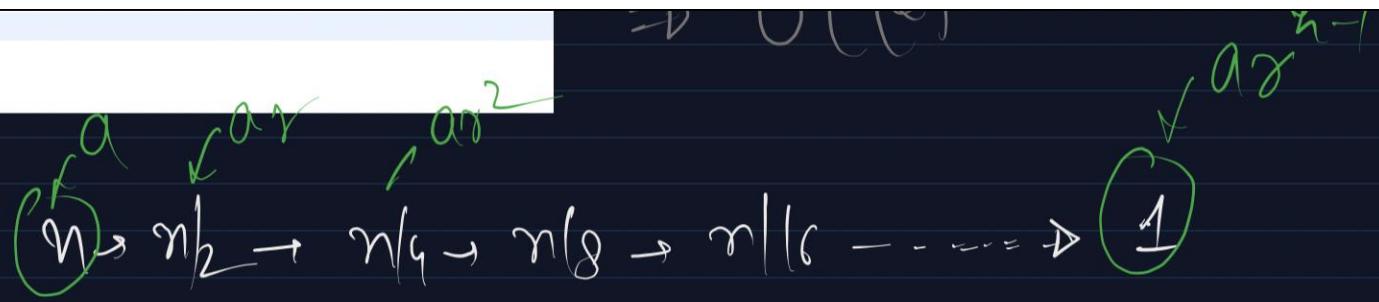
Rec. call stack $\rightarrow O(\log n)$

$n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow n/16 \dots \rightarrow 1$

h terms

```
    return power(x, n);  
}
```

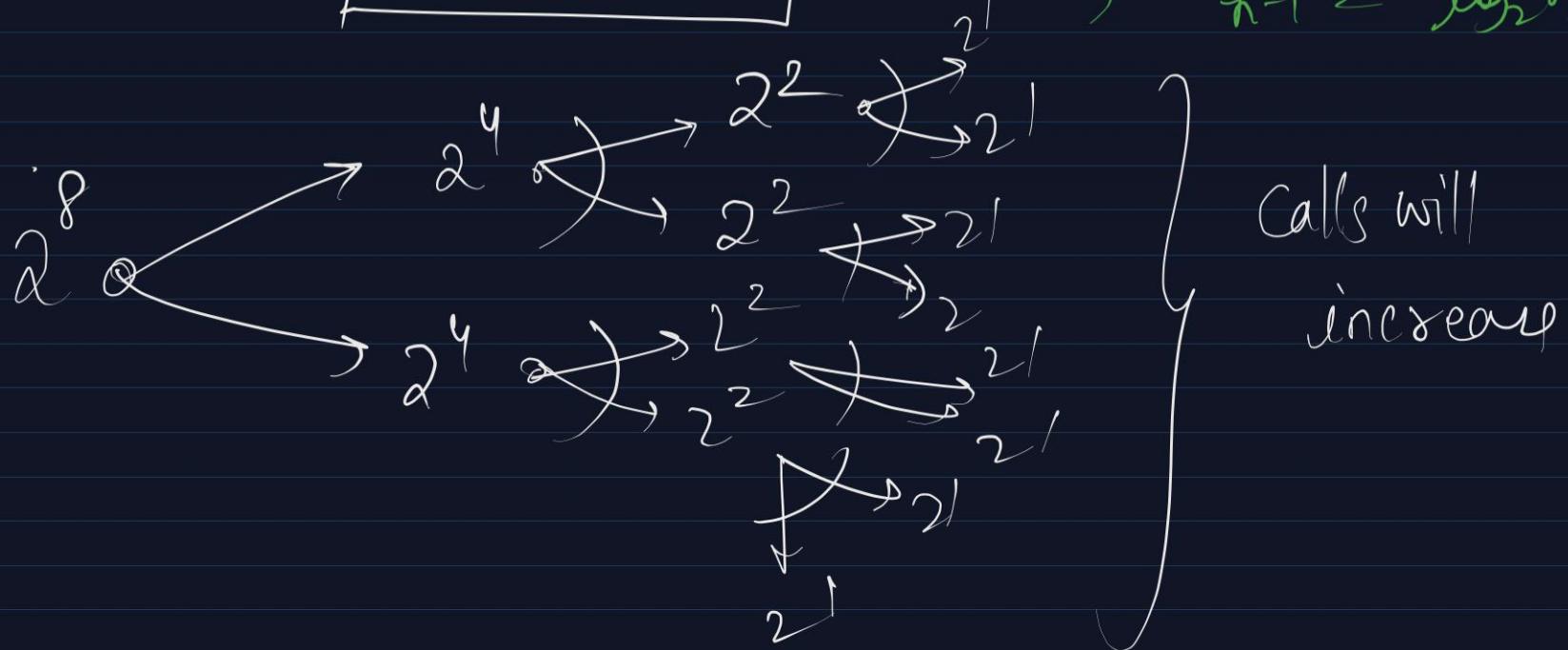
$\rightarrow O(1)$



h terms

$$1 = n \times (1/2)^{h-1} \Rightarrow 2^{h-1} = n$$

$$\boxed{h = O(\log n)}$$
$$\Rightarrow \log_2(2^{h-1}) = \log_2 n$$
$$\Rightarrow h-1 = \log n$$



```

class Solution {
    public double power(double x, int n){
        if(n == 0) return 1.0;

        double res = power(x, n/2); // Call 1
        if(n % 2 == 0)
            return res * res; // Meeting Expectation
        else
            return res * res * x;
    }

    public double myPow(double x, int n) {
        if(x == 0) return 0.0;
        if(x == 1) return 1.0;

        if(n < 0){
            return 1.0 / power(x, -n);
        }

        return power(x, n);
    }
}

```

Same S.C
as ~~Prec~~

$x^n \rightarrow x^{n/2} \rightarrow n^{\frac{n}{2}}$

height $\rightarrow h = O(\log_2 n)$

$$O\left(1^{\log_2 n} + k * \log n\right)$$

$$= O(\log_2 n) \approx O(1)$$

$\overbrace{2^{\log_2 n}}^{\text{B.L.}}$

$$O(\log_2 n) \ll O(n)$$

$$\begin{aligned} N &= 2^{16} \\ \log_2 n &= 16 \end{aligned}$$

Bit Manipulation \rightarrow Modular Exponentiation

\rightarrow TC: $O(\log_2 n)$

\rightarrow SC: $O(1)$

Dynamic Programming

→ "Those who can't remember their past are condemned to repeat it"

① Recursion

- Time Complexity Poor
- Space Complexity Poor

② Memoization

- Time Complexity Good
- Space Complexity Poor

③ Tabulation

- Time Complexity Good
- Space Complexity Good

Topic

Pre-requisites

★ → Dynamic Programming {Level 1 + 2} → Recursion

→ Hashmap & Heap {Level 1 + 2}

→ Graphs {Level 1 + Level 2} → Generic Tree {DFS, BFS}

→ Bit Manipulation → No System ↗ Binary Decimal

→ Array & String → Remaining Ques

Lecture ① Dynamic Programming {10:30 - 12:00} 19 Apr

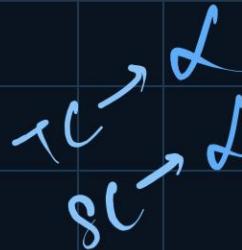
→ Fibonacci + Climb Stairs Module

"Those who can't remember their past
are condemned to repeat it"

Exponential

Backtracking

$$\left\{ \begin{array}{l} N \leq 18 \\ 20 \\ 30 \end{array} \right.$$



① Recursion { Brute force }



② Memoization { Top Down DP }

Recursive

Iterative

③ Tabulation { Bottom Up DP }



④ Space Optimization → limited previous states

Fibonacci Number

0, 1, 1, 2, 3, 5, 8, 13
0th 1st 2nd 3rd 4th 5th 6th 7th

Expectation: \rightarrow Nth Fibonacci No. $\{ \text{fib}(n) \}$

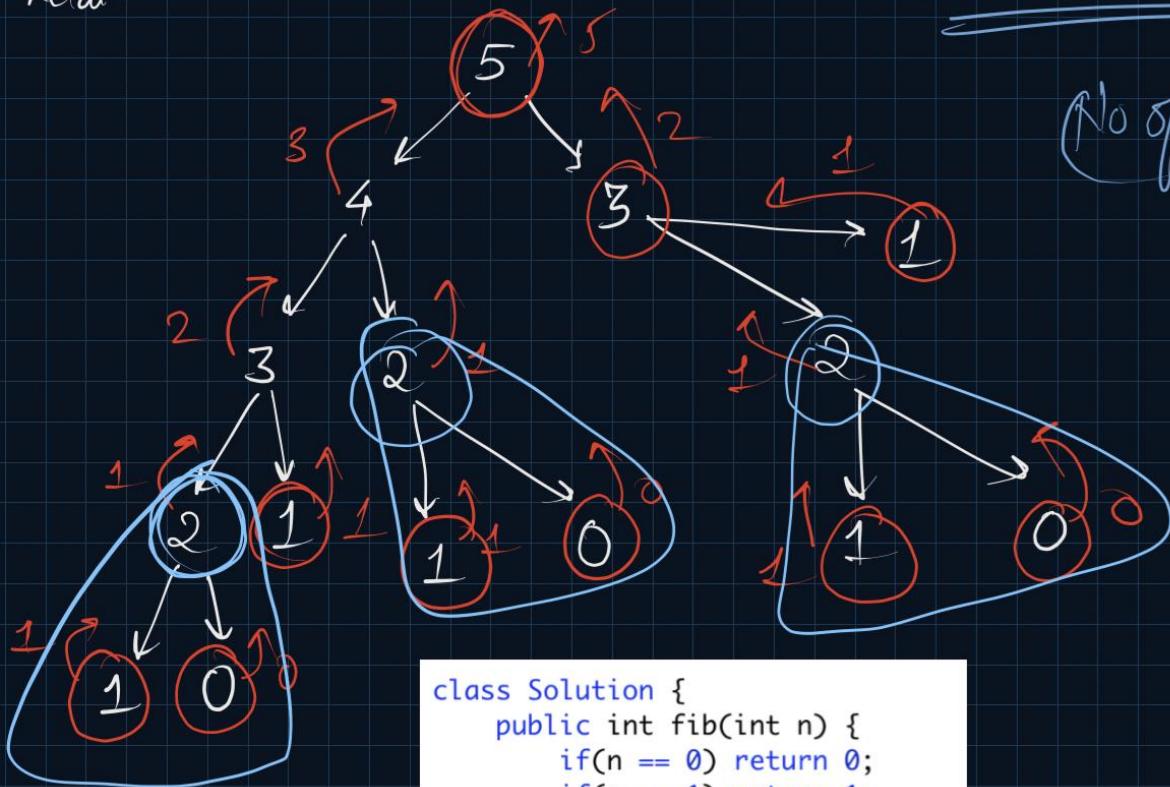
Faith: \rightarrow $\text{fib}(n-1)$, $\text{fib}(n-2)$

Recurrence Relation

$$\boxed{\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)}$$

Recurrence Relation

$$f_b(n) = f_b(n-1) + f_b(n-2)$$



worst case

height
 (No of calls) + (pre + post) * height

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

$\Rightarrow O(2^N)$ Time Complexity

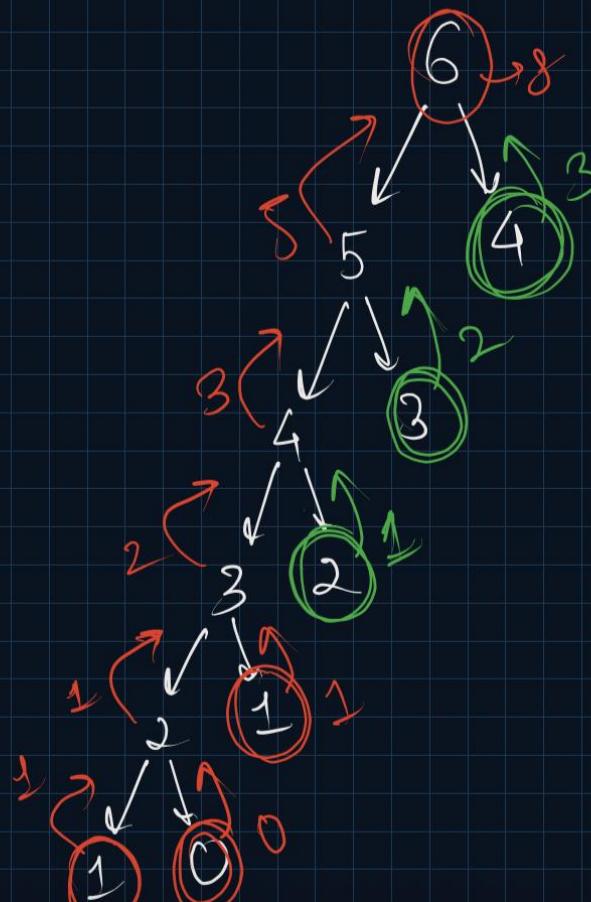
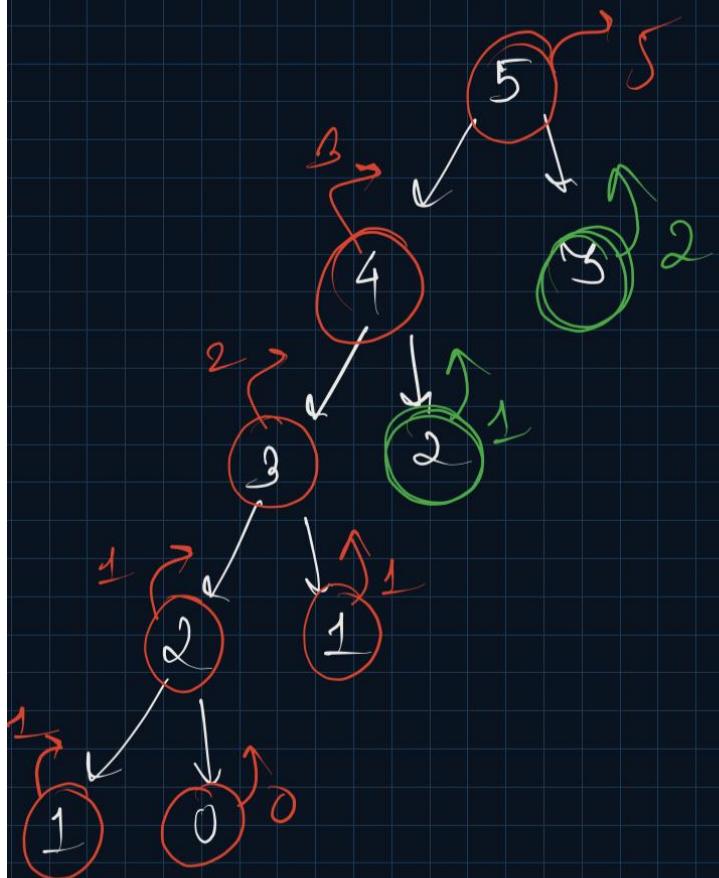
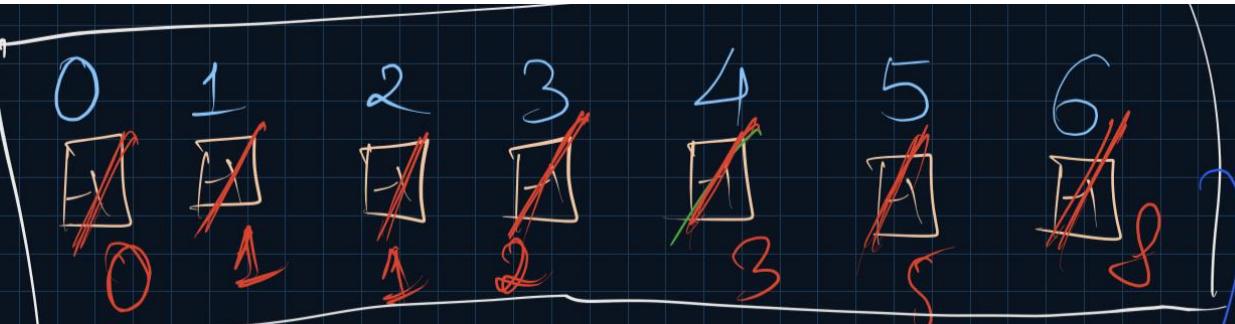
Space Complexity $\Rightarrow O(N)$
 $R \cdot C \cdot S \cdot S$

```
class Solution {
    public int fib(int n) {
        if(n == 0) return 0;
        if(n == 1) return 1;

        int prev1 = fib(n - 1);
        int prev2 = fib(n - 2);

        return prev1 + prev2;
    }
}
```

Memoization



$O(N)$ Time Complexity

```

class Solution {
    public int fib(int n, int[] dp){
        if(n == 0) return 0;
        if(n == 1) return 1;
        if(dp[n] != -1) return dp[n];
        // Already Calculated Value should be returned

        int prev1 = fib(n - 1, dp);
        int prev2 = fib(n - 2, dp);

        dp[n] = prev1 + prev2;
        // Before returning the calculated value, store it somewhere
        return prev1 + prev2;
    }

    public int fib(int n) {
        int[] dp = new int[n + 1];
        Arrays.fill(dp, -1);
        return fib(n, dp);
    }
}

```

Recursion Call stack

$\rightarrow O(N)$

Extra Space: $\rightarrow O(N) \underset{DP}{=}$

Time Complexity $\rightarrow O(N)$

Dynamic Programming Identification

\rightarrow ① Overlapping Subproblems { Repeated calls }

\rightarrow ② Optimal Substructure of Fath Z

③ Tabulation

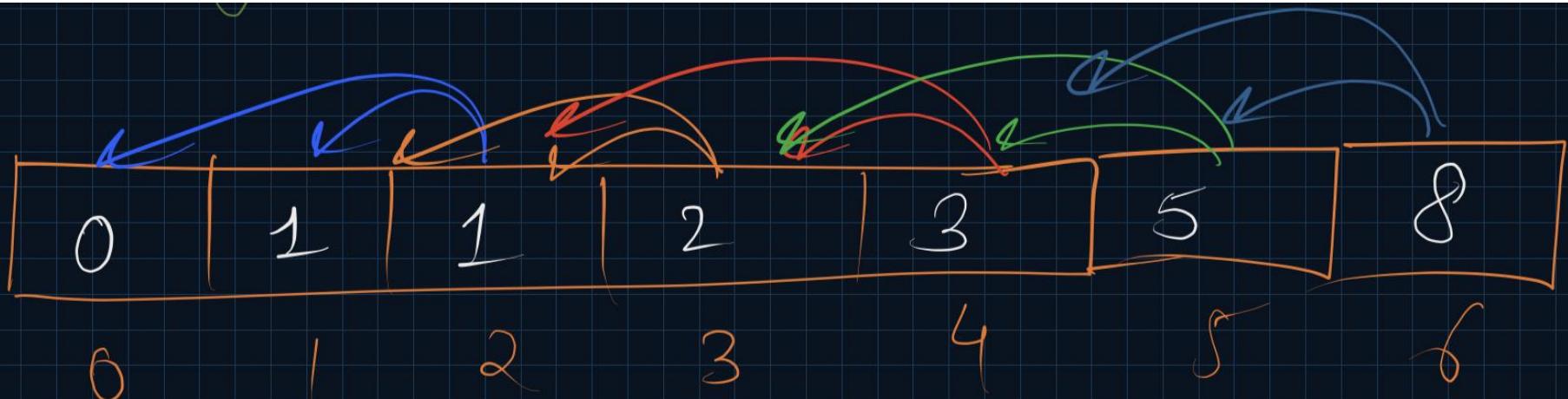
$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

$$DP[n] = DP[n-1] + DP[n-2]$$

① $DP[i] \rightarrow$ Meaning of the cell
↓
ith fibonacci no

② Smaller Problem $\{ DP[0], DP[1] \}$
↓
Bigger Problem $\{ DP[n] \}$

Iterative
soln



```
class Solution {  
    public int fib(int n) {  
        if(n <= 1) return n;  
  
        int[] dp = new int[n + 1];  
        dp[0] = 0; dp[1] = 1;  
  
        for(int i=2; i<=n; i++){  
            dp[i] = dp[i - 1] + dp[i - 2];  
        }  
  
        return dp[n];  
    }  
}
```

TC $\rightarrow O(N)$

SC $\rightarrow O(N)$
(extra space)

If R.C.S.S $\rightarrow O(1)$

W

```
class Solution {  
    public int fib(int n) {  
        if(n <= 1) return n;  
  
        int prev1 = 0, prev2 = 1;  
  
        for(int i=2; i<=n; i++){  
            int curr = prev1 + prev2;  
            prev1 = prev2;  
            prev2 = curr;  
        }  
  
        return prev2;  
    }  
}
```

}

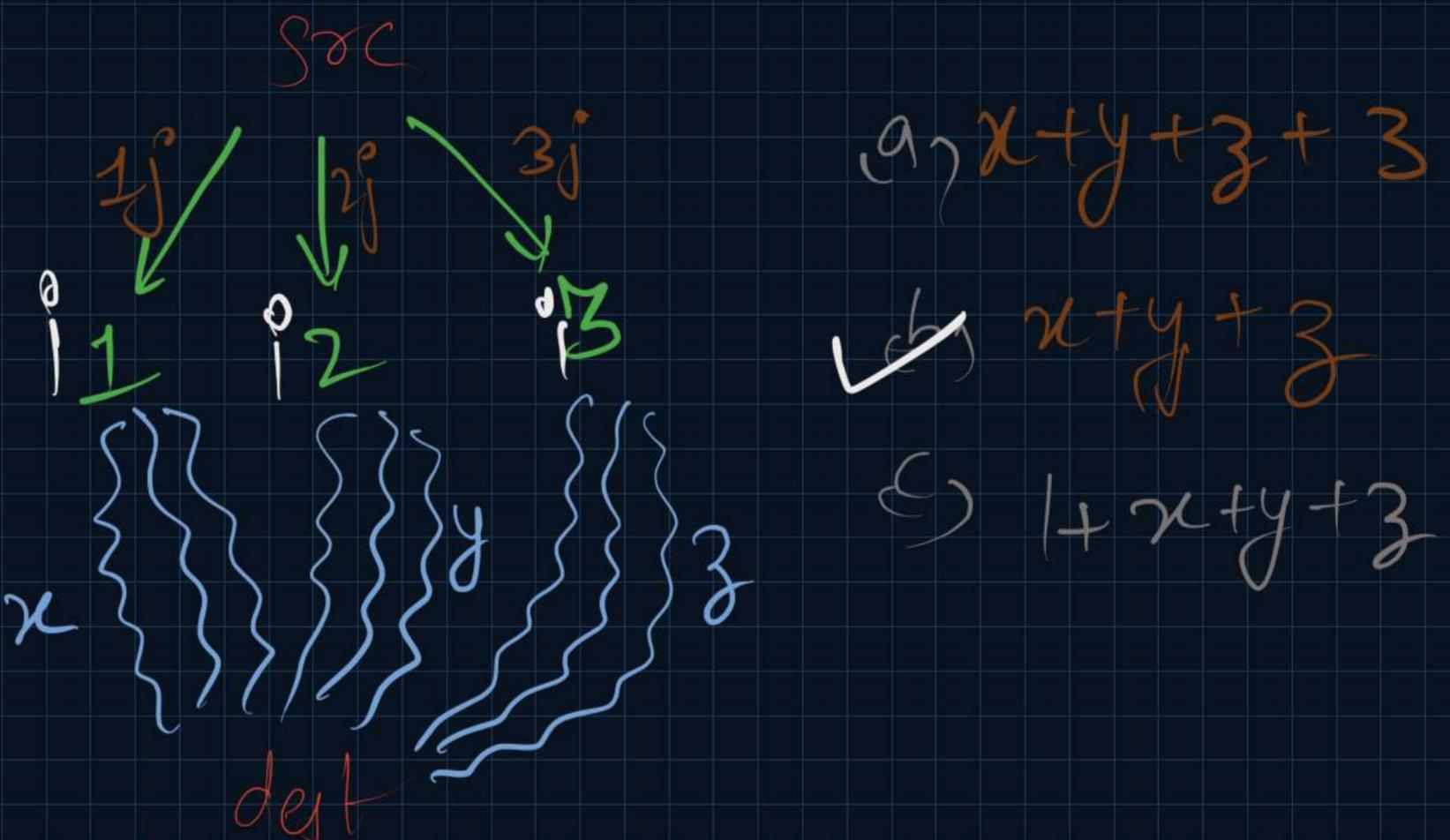
Time Complexity $\rightarrow \Theta(N)$
Space Complexity $\rightarrow O(1)$

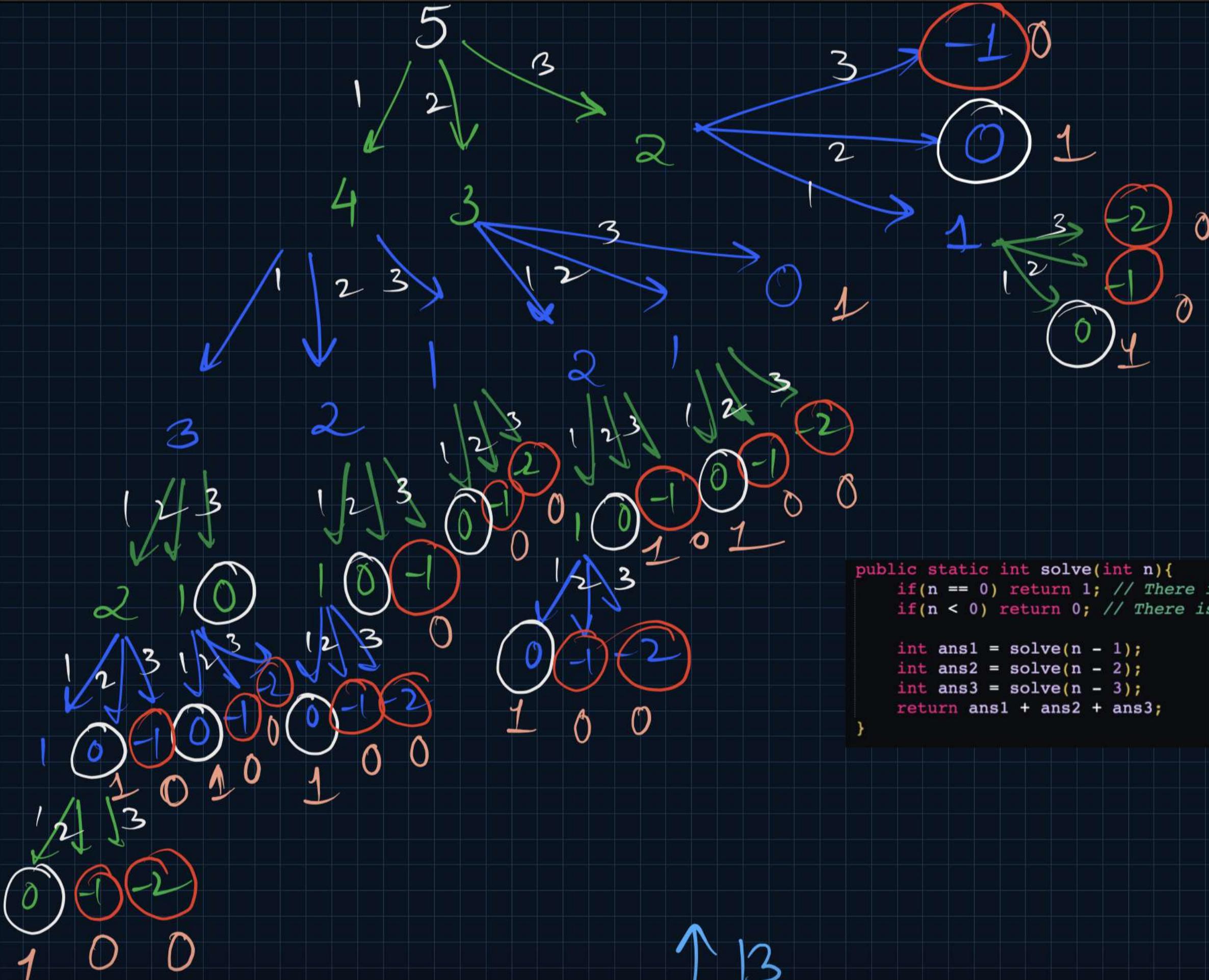
Dynamic Programming - Lecture ②

{ 23 April 2022
9 AM - 12 PM}

→ Climb Stairs & its Variations

① Climb Stairs



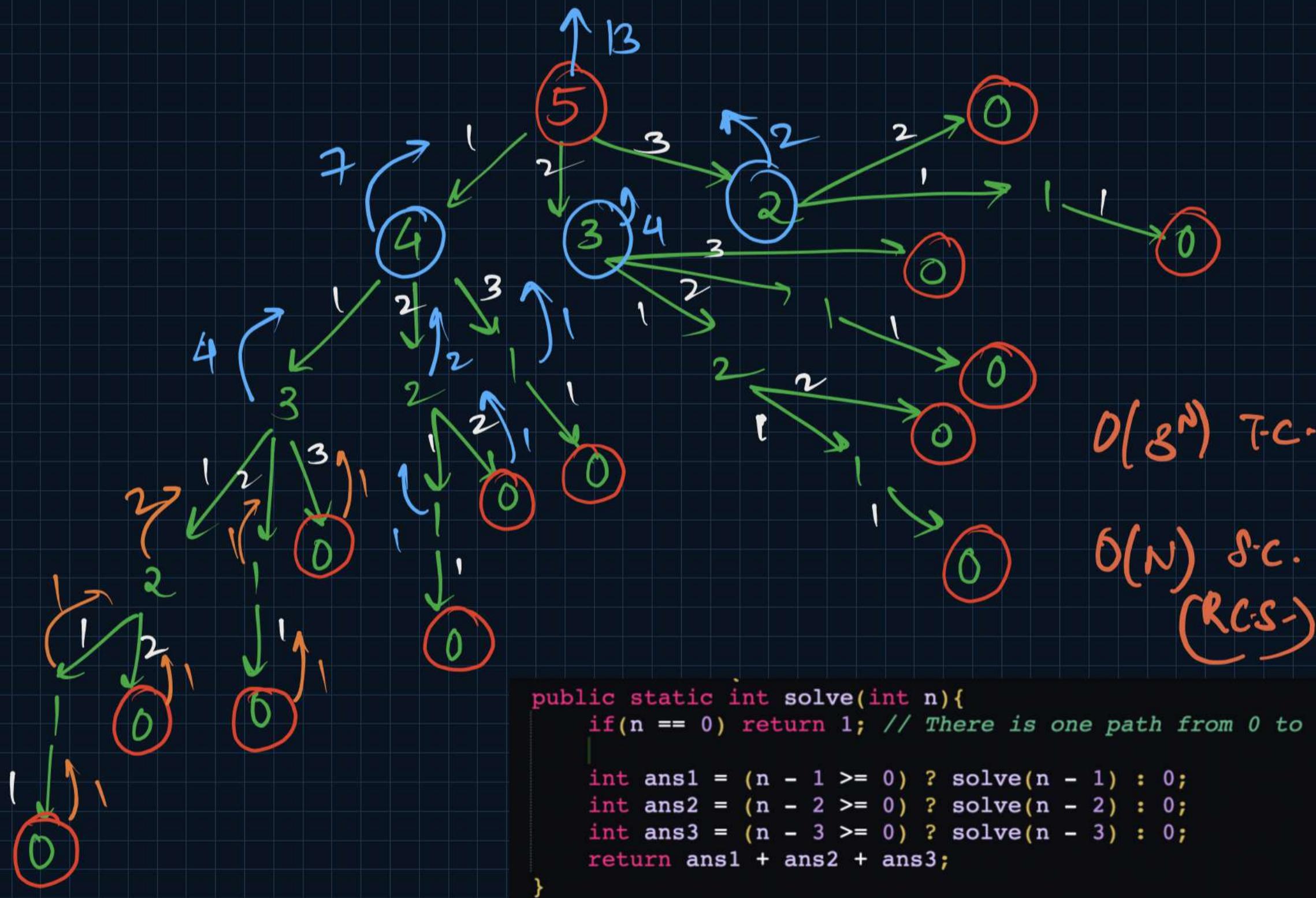


```

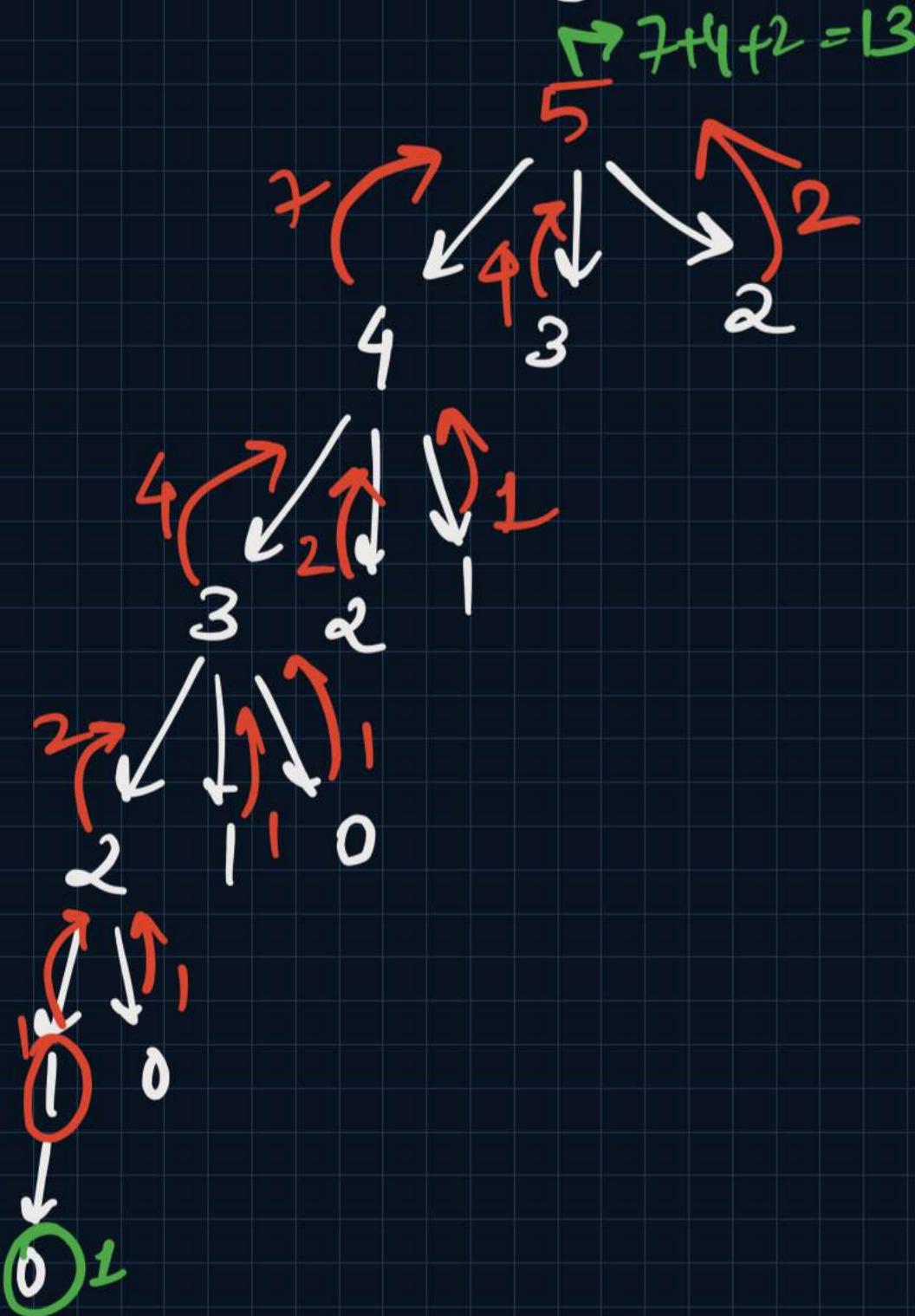
public static int solve(int n){
    if(n == 0) return 1; // There is one path from 0 to 0, i.e. "" empty string
    if(n < 0) return 0; // There is no path from -1 or -2 to 0

    int ans1 = solve(n - 1);
    int ans2 = solve(n - 2);
    int ans3 = solve(n - 3);
    return ans1 + ans2 + ans3;
}

```



Memoization



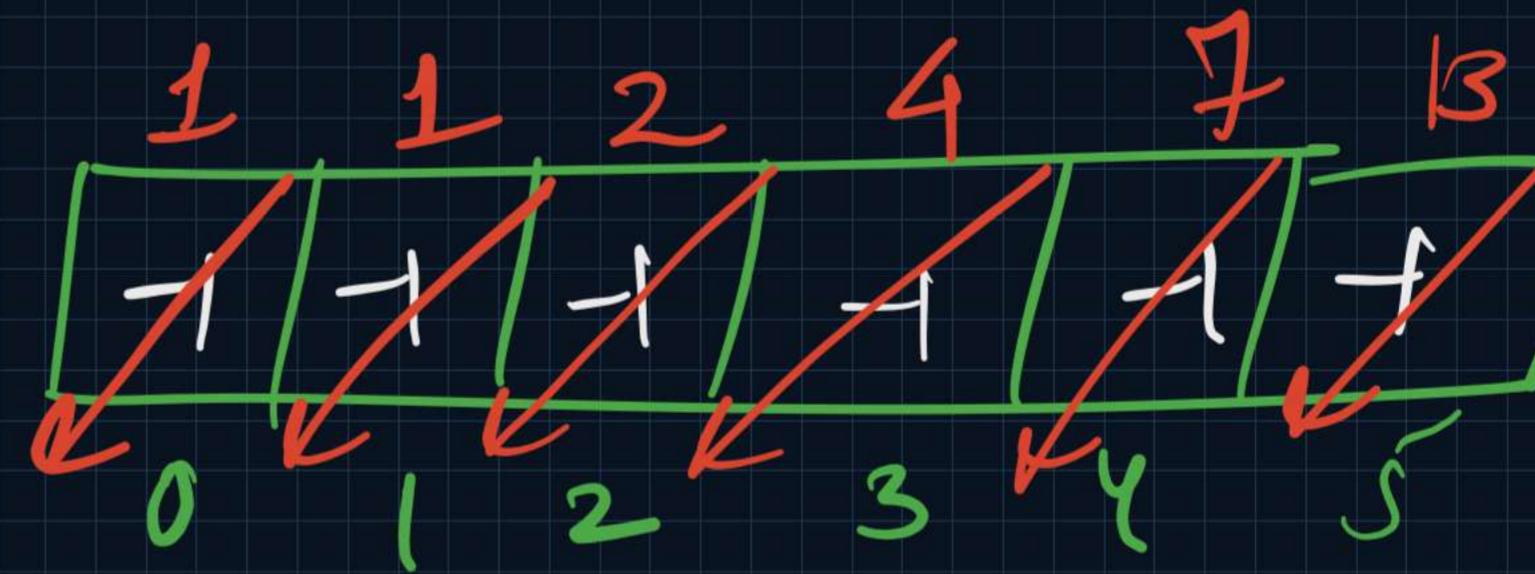
```

public static int solve(int n, int[] dp){
    if(n == 0) return 1; // There is one path from 0 to 0, i.e. "" empty string
    if(dp[n] != -1) return dp[n]; // Return the already calculated value (Preorder)

    int ans1 = (n - 1 >= 0) ? solve(n - 1, dp) : 0;
    int ans2 = (n - 2 >= 0) ? solve(n - 2, dp) : 0;
    int ans3 = (n - 3 >= 0) ? solve(n - 3, dp) : 0;

    dp[n] = ans1 + ans2 + ans3; // Store the Answer in DP Table (Postorder)
    return dp[n];
}

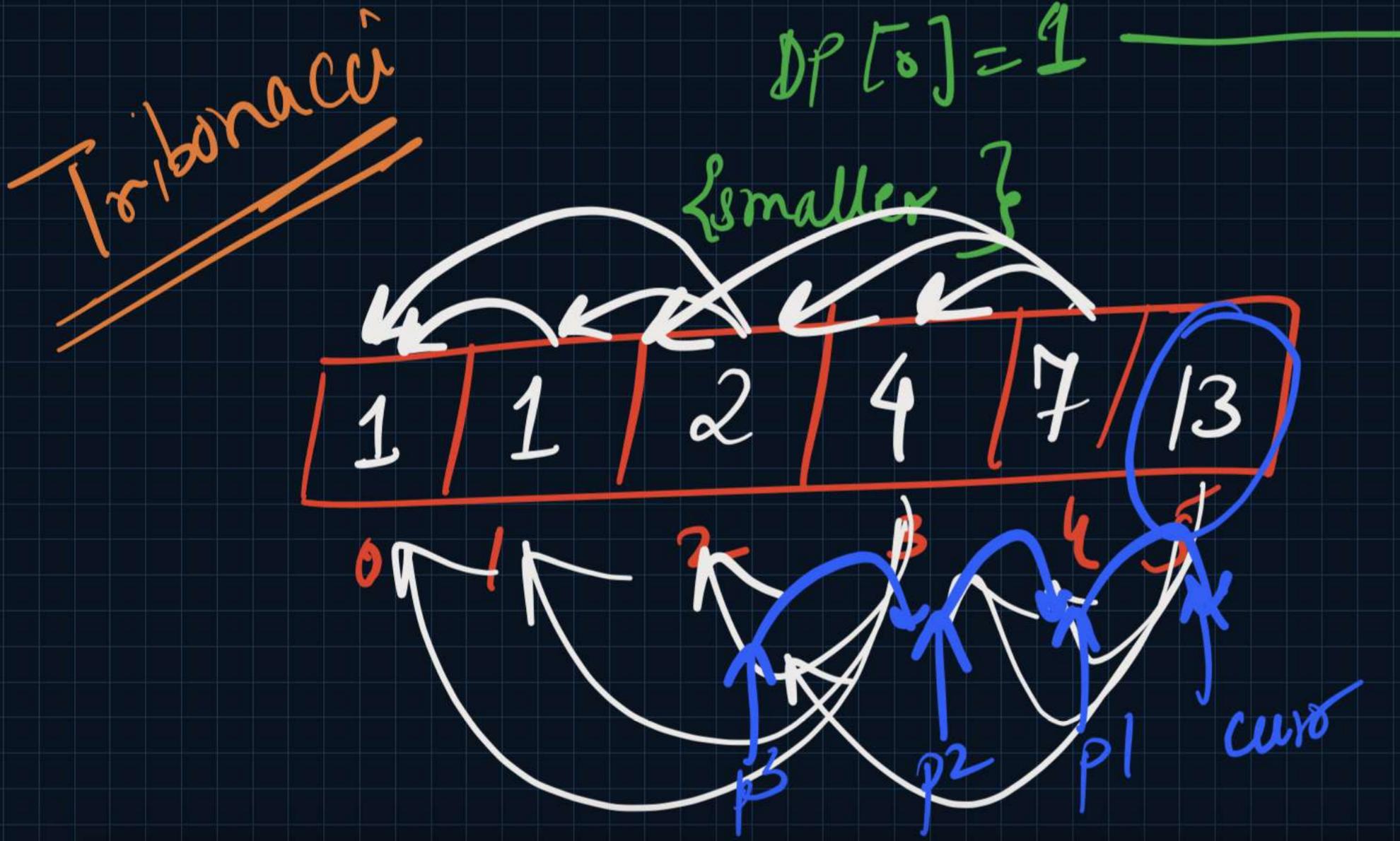
```



Time Complexity $\rightarrow O(N)$

Space Complexity $\rightarrow O(N) \xrightarrow{\text{R.C.S}} \text{DP Table}$

Tabulation



TC $\rightarrow O(N)$
SC $\rightarrow O(N)$
extra

Nth state
DP[N] = ?
{larger}

$$DP[N] = DP[N-1] + DP[N-2] + DP[N-3]$$

```
int n = scn.nextInt();
int[] dp = new int[n + 1];
dp[0] = 1;

for(int i=1; i<=n; i++){
    int ans1 = (i - 1 >= 0) ? dp[i - 1] : 0;
    int ans2 = (i - 2 >= 0) ? dp[i - 2] : 0;
    int ans3 = (i - 3 >= 0) ? dp[i - 3] : 0;
    dp[i] = ans1 + ans2 + ans3;
}

System.out.println(dp[n]);
```

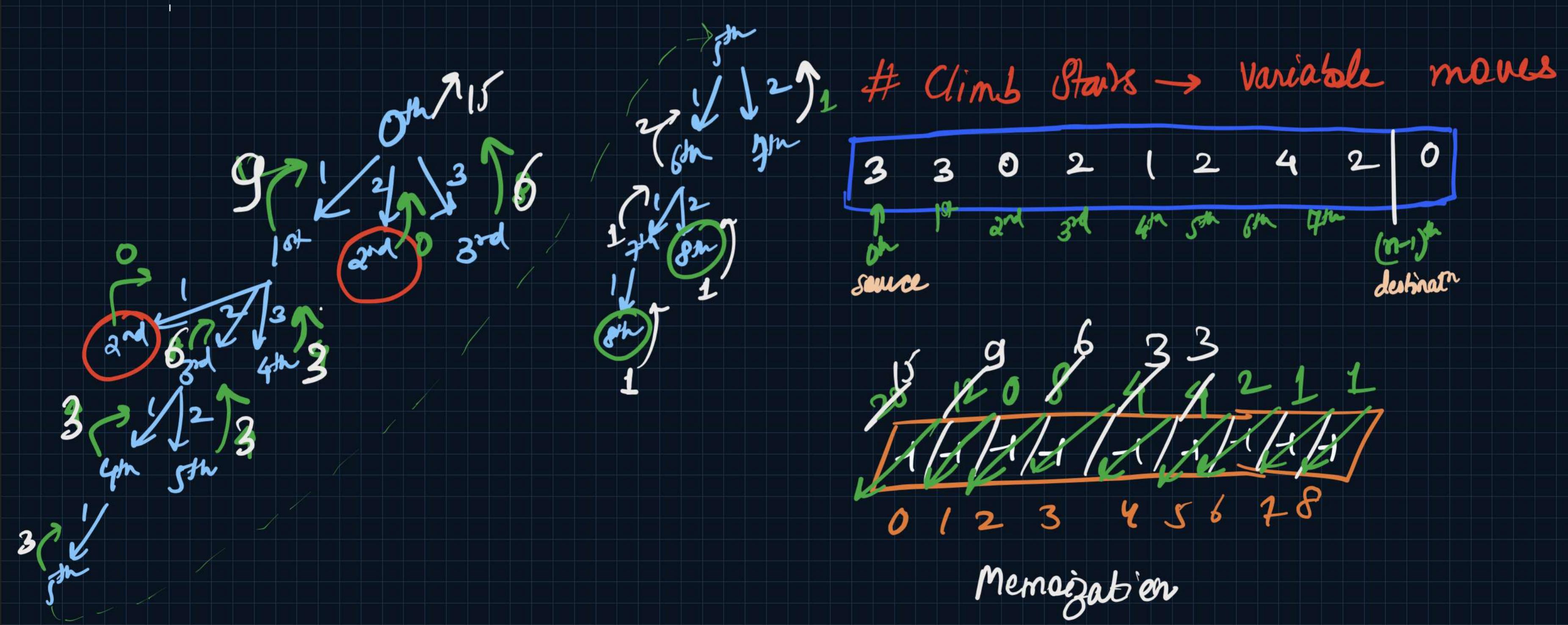
Extra Space
Optimization
O(n) Time
O(n) Space

```
public static void main(String[] args) throws Exception {
    Scanner scn = new Scanner(System.in);
    int n = scn.nextInt();

    int prev1 = 1, prev2 = 0, prev3 = 0;
    for(int i=1; i<=n; i++){
        int curr = prev1 + prev2 + prev3;
        prev3 = prev2;
        prev2 = prev1;
        prev1 = curr;
    }

    System.out.println(prev1);
}
```

4 pointer
Technique



```

public class Main {
    public static int memo(int n, int[] jumps, int[] dp){
        if(n == jumps.length) return 1;
        if(dp[n] != -1) return dp[n];

        int ans = 0;
        for(int i=1; i<=jumps[n]; i++){
            if(n + i <= jumps.length)
                ans += memo(n + i, jumps, dp);
        }

        dp[n] = ans;
        return ans;
    }
}

```

```

public static void main(String[] args) throws Exception {
    Scanner scn = new Scanner(System.in);
    int n = scn.nextInt();
    int[] jumps = new int[n];
    for(int i=0; i<n; i++){
        jumps[i] = scn.nextInt();
    }

    int[] dp = new int[n + 1];
    Arrays.fill(dp, -1);
    System.out.println(memo(0, jumps, dp));
}

```

recursion
 \downarrow
 $TC \Rightarrow O(jumps^N)$

$SC \Rightarrow O(N)$

R-C-S

Mem

$TC \Rightarrow O(N \times jumps)$

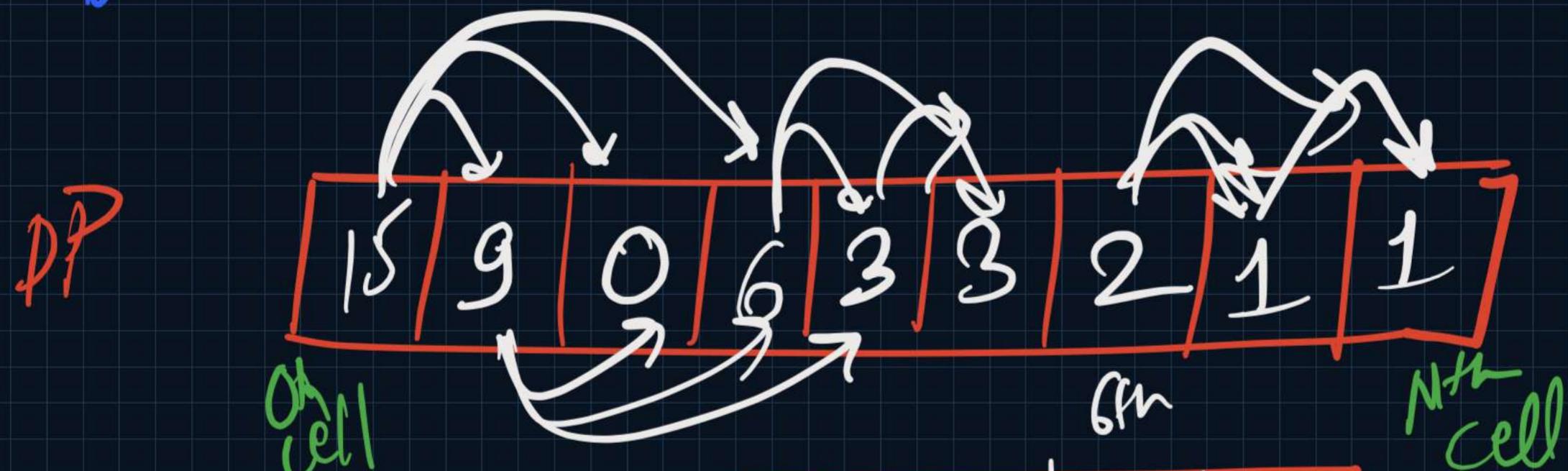
$SC \Rightarrow O(N)$

R-C-S, DT

~~Tabulation~~

hanger
Bth state

Smaller
Nth state



jumps

A diagram illustrating a dynamic programming table for jumps. The table consists of 8 columns, each representing a state. The values in the columns are: 3, 3, 0, 2, 1, 2, 4, 2. The table is enclosed in a red border.

3	3	0	2	1	2	4	2
---	---	---	---	---	---	---	---

```

public static void main(String[] args) throws Exception {
    Scanner scn = new Scanner(System.in);
    int n = scn.nextInt();
    int[] jumps = new int[n];
    for(int i=0; i<n; i++){
        jumps[i] = scn.nextInt();
    }

    int[] dp = new int[n + 1];
    dp[n] = 1;

    for(int cell=n-1; cell>=0; cell--){
        int ans = 0;
        for(int jump=1; jump<=jumps[cell]; jump++){
            if(cell + jump <= n){
                ans += dp[cell + jump];
            }
        }
        dp[cell] = ans;
    }
    System.out.println(dp[0]);
}

```

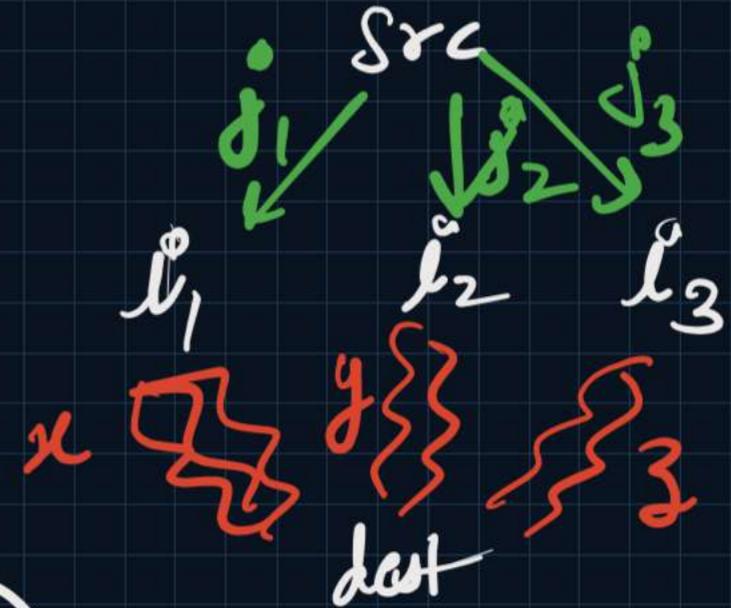
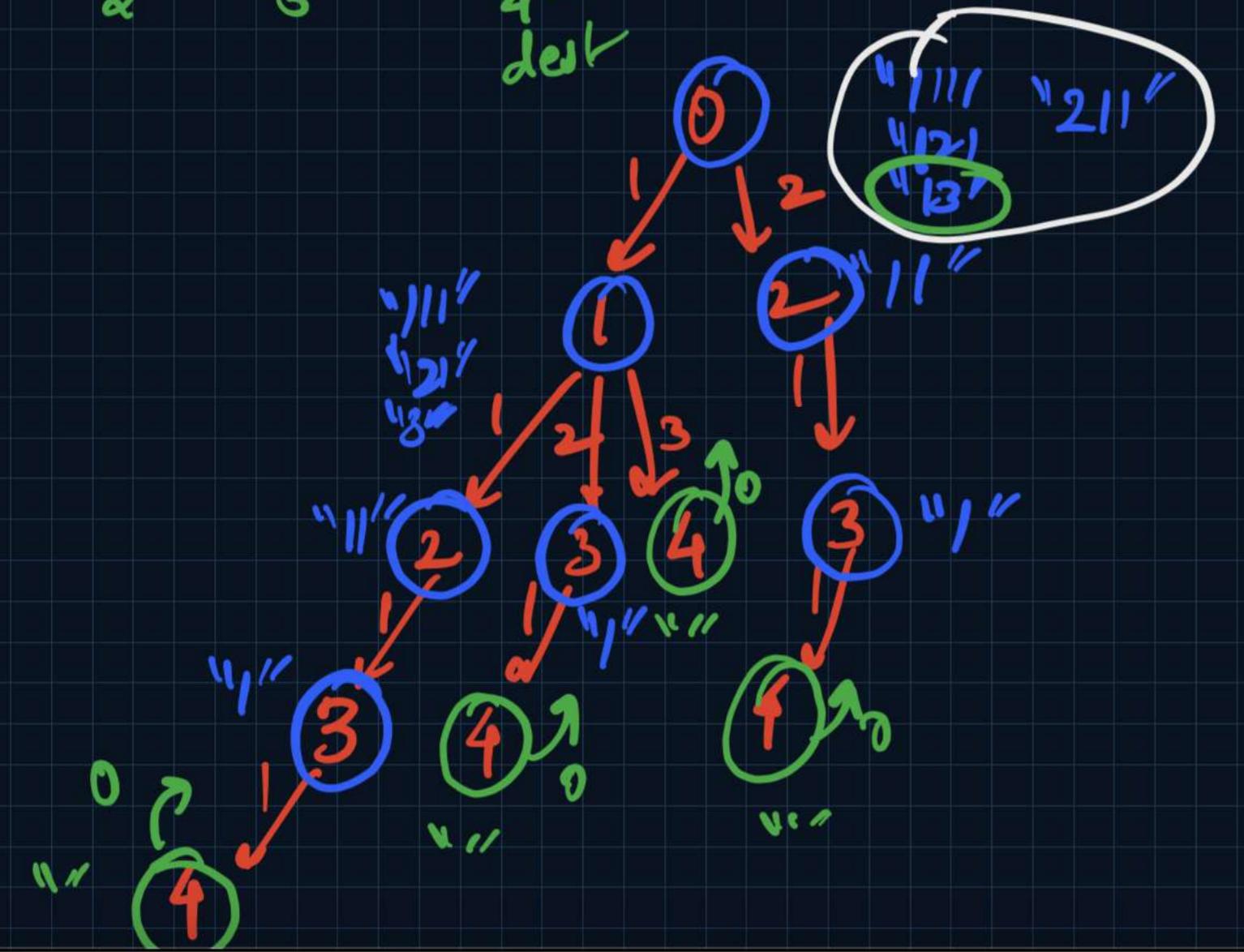
$$O(N * \text{jumps}) \quad \underline{TC}$$

$$O(N) \leq \{ \text{DP table} \}$$

Space optimization

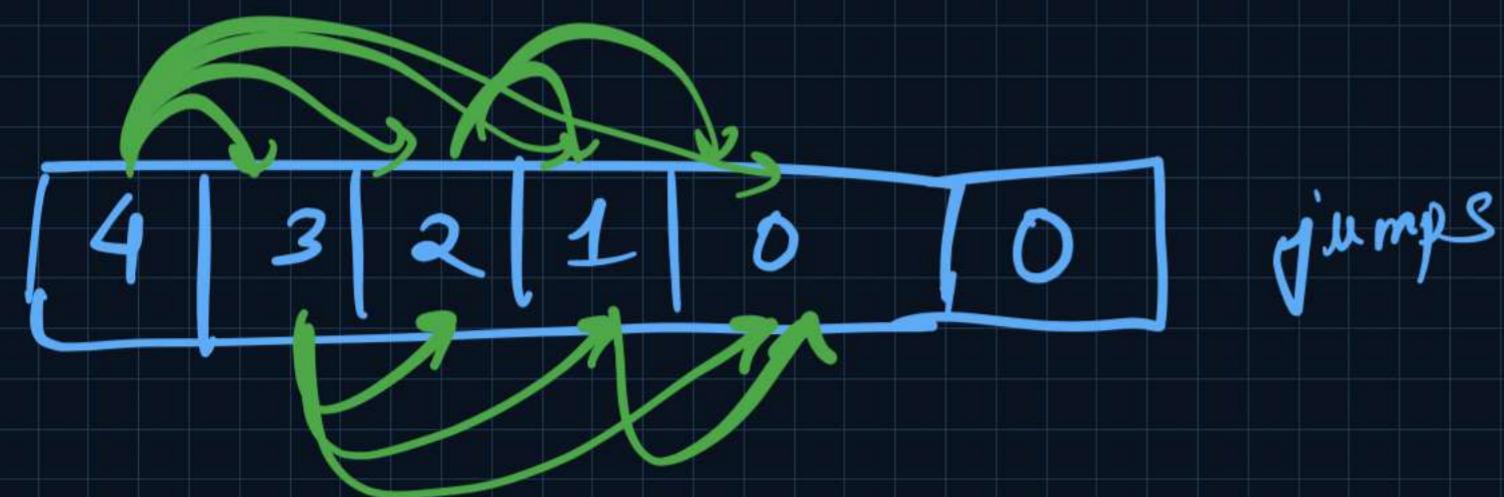
↳ Not possible { jumps value is not fixed }

climb stairs with minimum jumps



$$\text{ans} = \min(x, y, z)$$

eg²



Count of paths = 0

min length = +∞



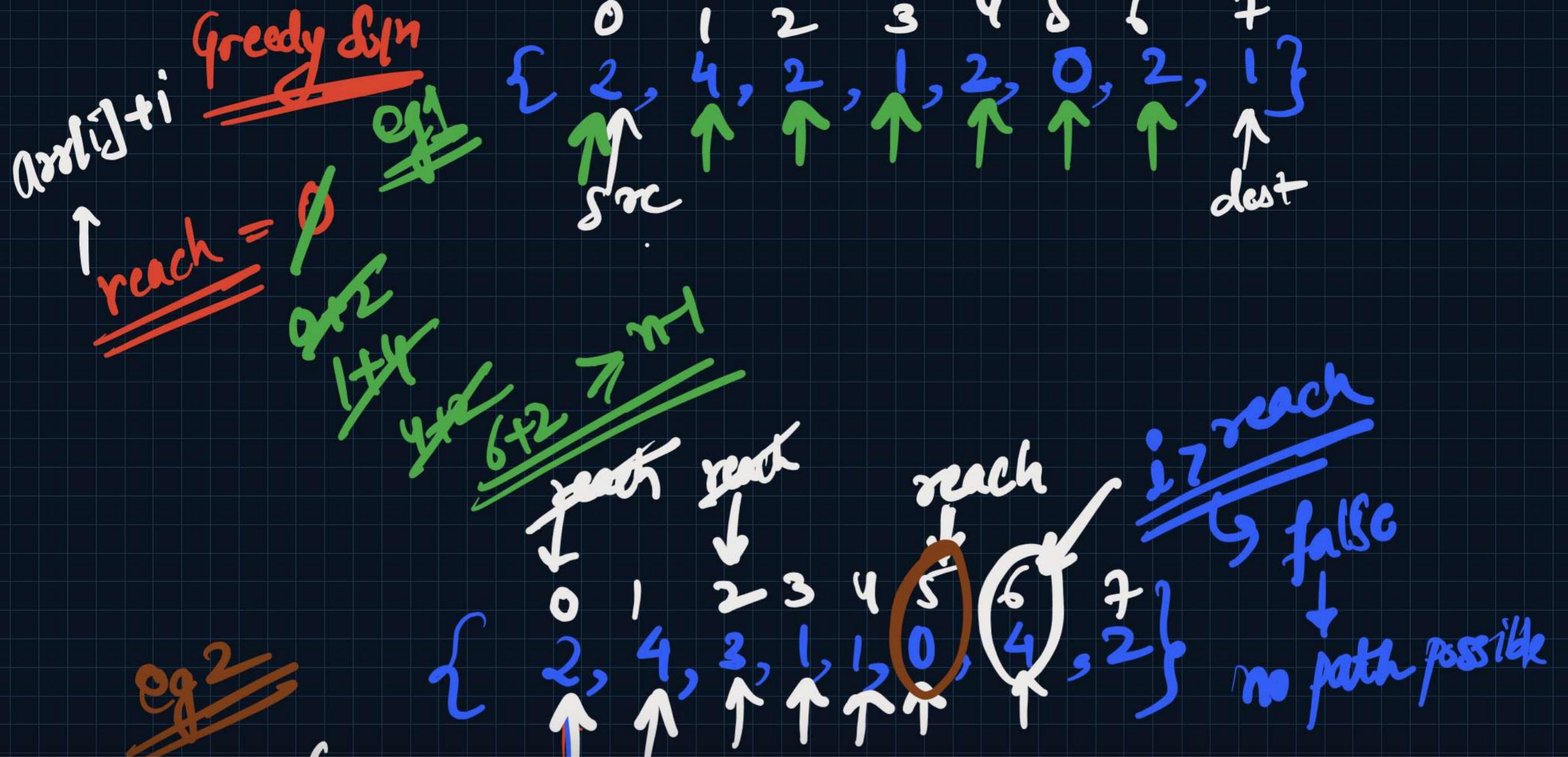
```
class Solution {  
    public long memo(int src, int[] jumps, long[] dp){  
        if(src == jumps.length - 1)  
            return 0; // min moves to go to dest from dest is 0 (empty string)  
        if(dp[src] != -1) return dp[src];  
  
        long min = Integer.MAX_VALUE;  
        for(int jump=1; jump<=jumps[src]; jump++){  
            if(src + jump < jumps.length){  
                min = Math.min(min, memo(src + jump, jumps, dp) + 1);  
            }  
        }  
  
        dp[src] = min;  
        return min;  
    }  
  
    public int jump(int[] jumps) {  
        long[] dp = new long[jumps.length + 1];  
        Arrays.fill(dp, -1);  
  
        return (int)memo(0, jumps, dp);  
    }  
}
```

TC $\Rightarrow O(N \times \text{jumps})$
SC $\Rightarrow O(N)$

* Tabulation \rightarrow HW \rightarrow TC $\Rightarrow O(n * j^{umps}) \approx O(N^2)$

* Space Optimization \rightarrow N of possible

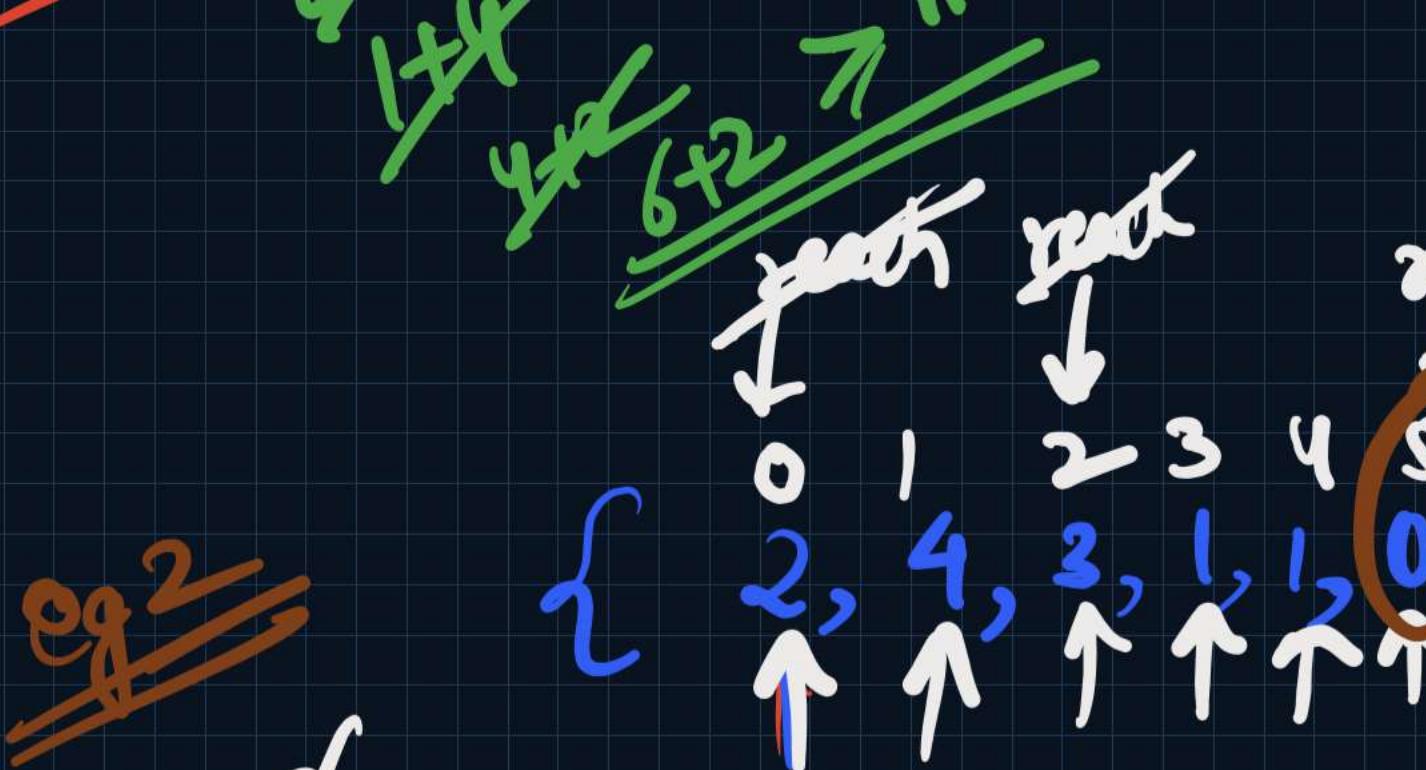
Jump Game - I { check if $+\infty$, \Rightarrow false { path not possible }
else \Rightarrow true { path possible }



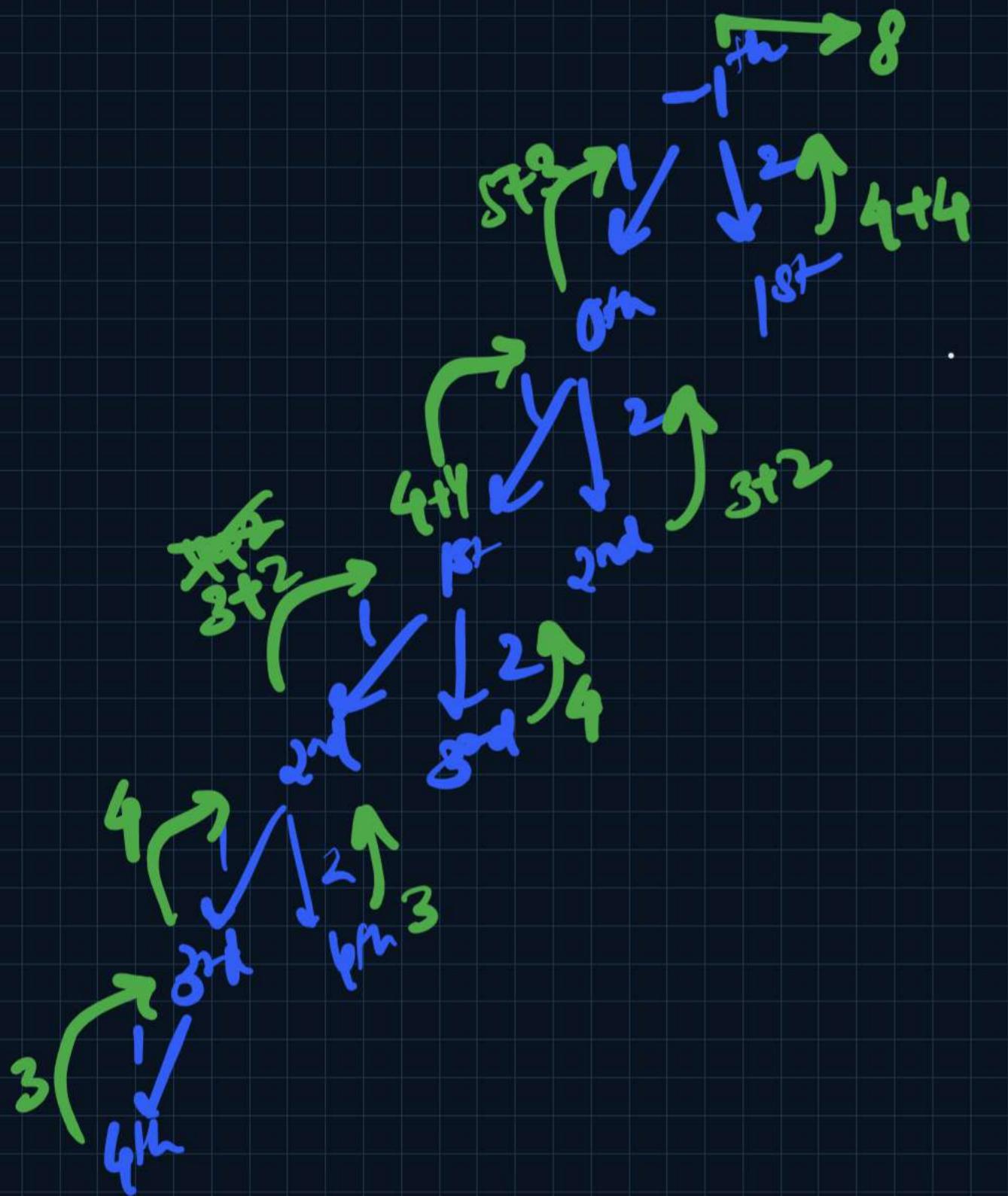
~~reach = 0~~
~~0, 4, 1, 4~~

{

```
class Solution {  
    public boolean canJump(int[] jumps) {  
        int reach = 0;  
        for(int i=0; i<jumps.length; i++){  
            if(i > reach) return false;  
  
            if(i + jumps[i] > reach){  
                reach = i + jumps[i];  
            }  
        }  
  
        return true;  
    }  
}
```



TC $\Rightarrow O(N)$
SC $\Rightarrow O(1)$



Climb Stairs with Minimum Cost

$$\{0, 3, 4, 2, 1, 3\}$$

-1 0 1 2 3 4

jumps

1
2

$SRC(x)$

i_1 i_2

$\{ \} p$ $\{ \} q$

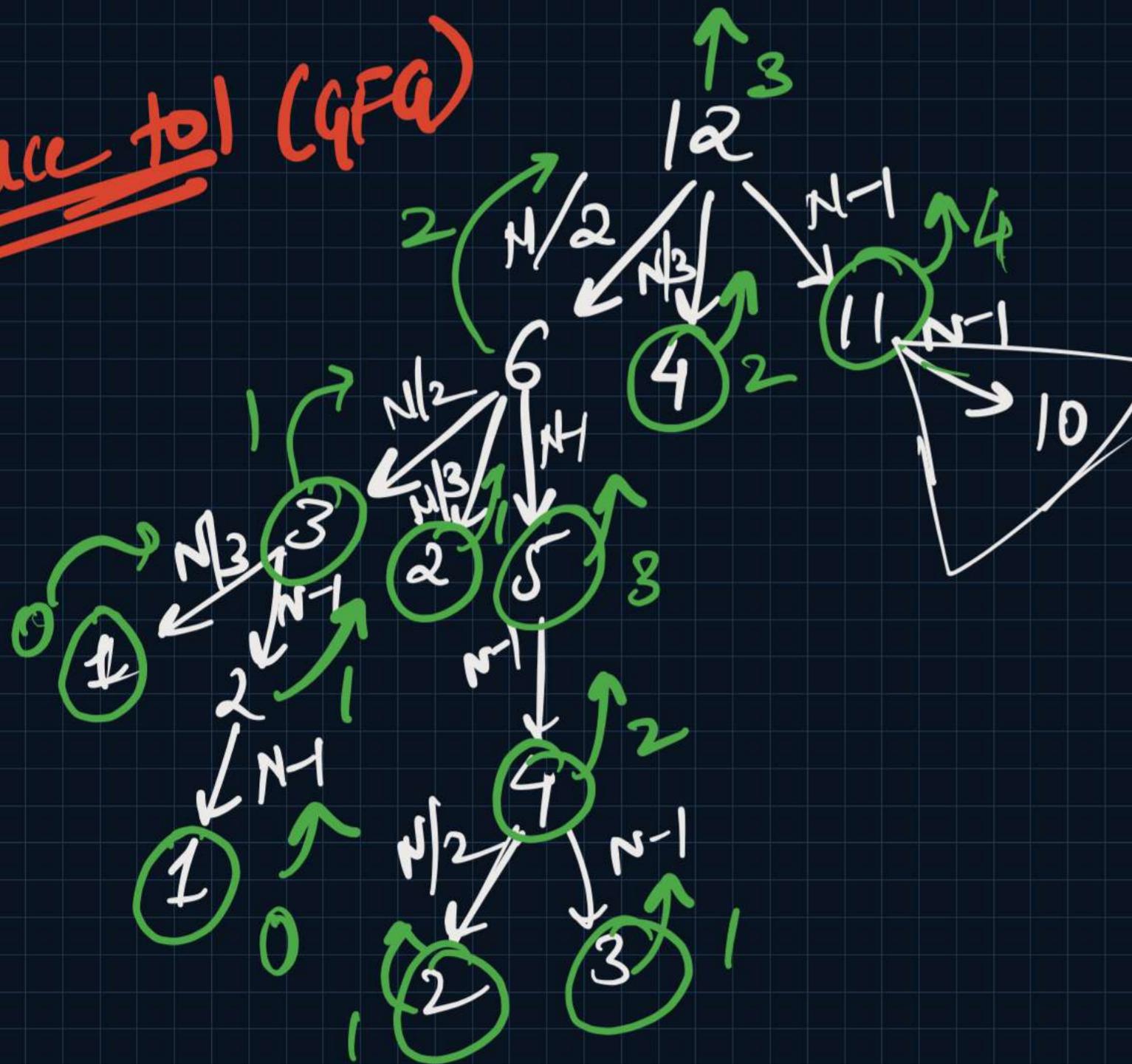
dp

$\min(p, q) + x$

```
class Solution {  
    public int memo(int src, int[] cost, int[] dp){  
        if(src >= cost.length) return 0;  
        if(dp[src] != -1) return dp[src];  
  
        int ans1 = memo(src + 1, cost, dp);  
        int ans2 = memo(src + 2, cost, dp);  
  
        return dp[src] = Math.min(ans1, ans2) + cost[src];  
    }  
  
    public int minCostClimbingStairs(int[] cost) {  
        int[] dp = new int[cost.length + 1];  
        Arrays.fill(dp, -1);  
  
        memo(0, cost, dp);  
        return Math.min(dp[0], dp[1]);  
    }  
}
```

Memoization
 $O(N)$ time
 $O(N)$ space

reduce to 1 (GFA)



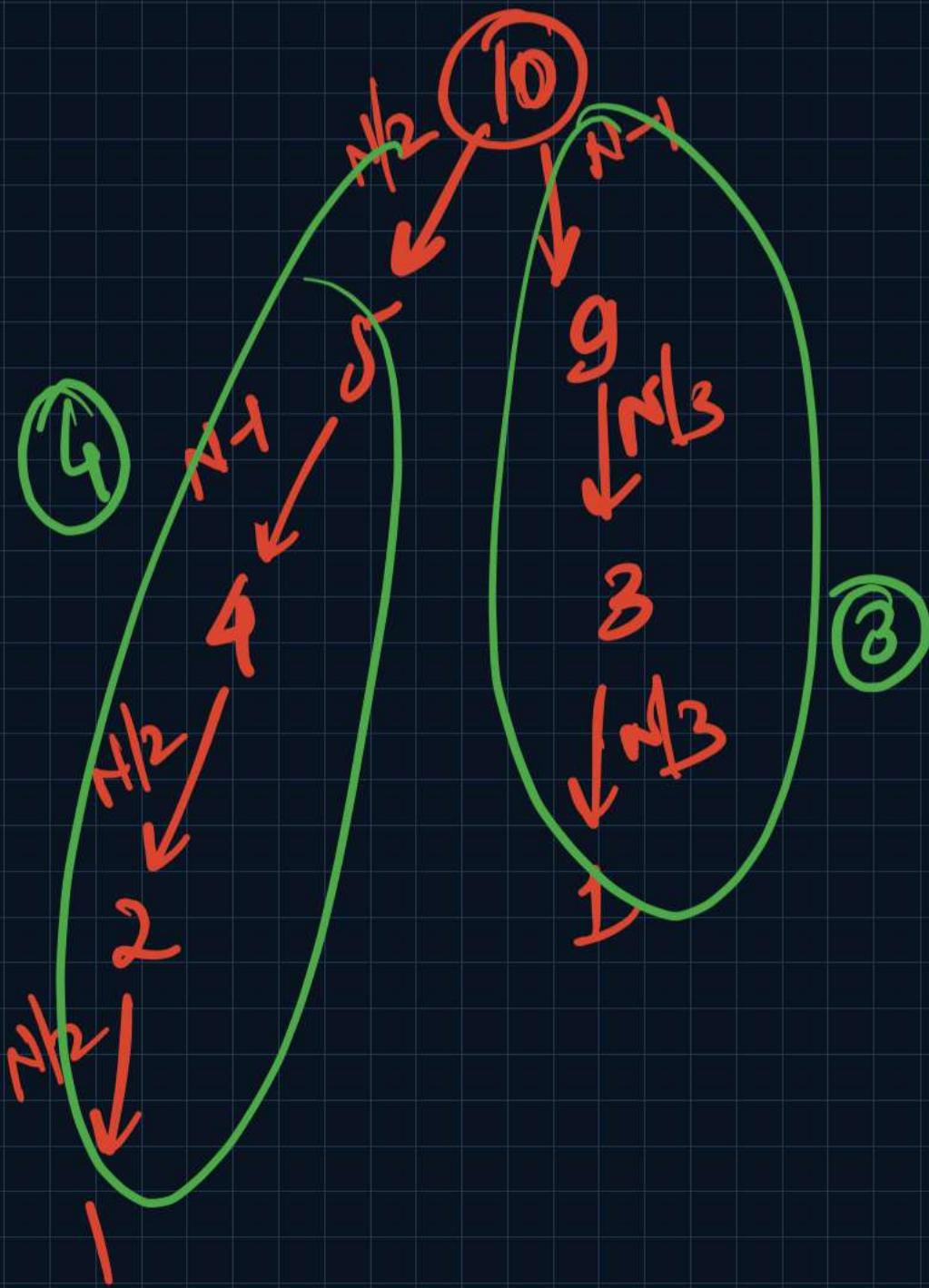
```
class Solution{
    public int memo(int N, int[] dp){
        if(N == 1) return 0;
        if(dp[N] != -1) return dp[N];

        int ans1 = (N % 2 == 0) ? memo(N/2, dp) : Integer.MAX_VALUE;
        int ans2 = (N % 3 == 0) ? memo(N/3, dp) : Integer.MAX_VALUE;
        int ans3 = memo(N - 1, dp);

        return dp[N] = Math.min(ans1, Math.min(ans2, ans3)) + 1;
    }

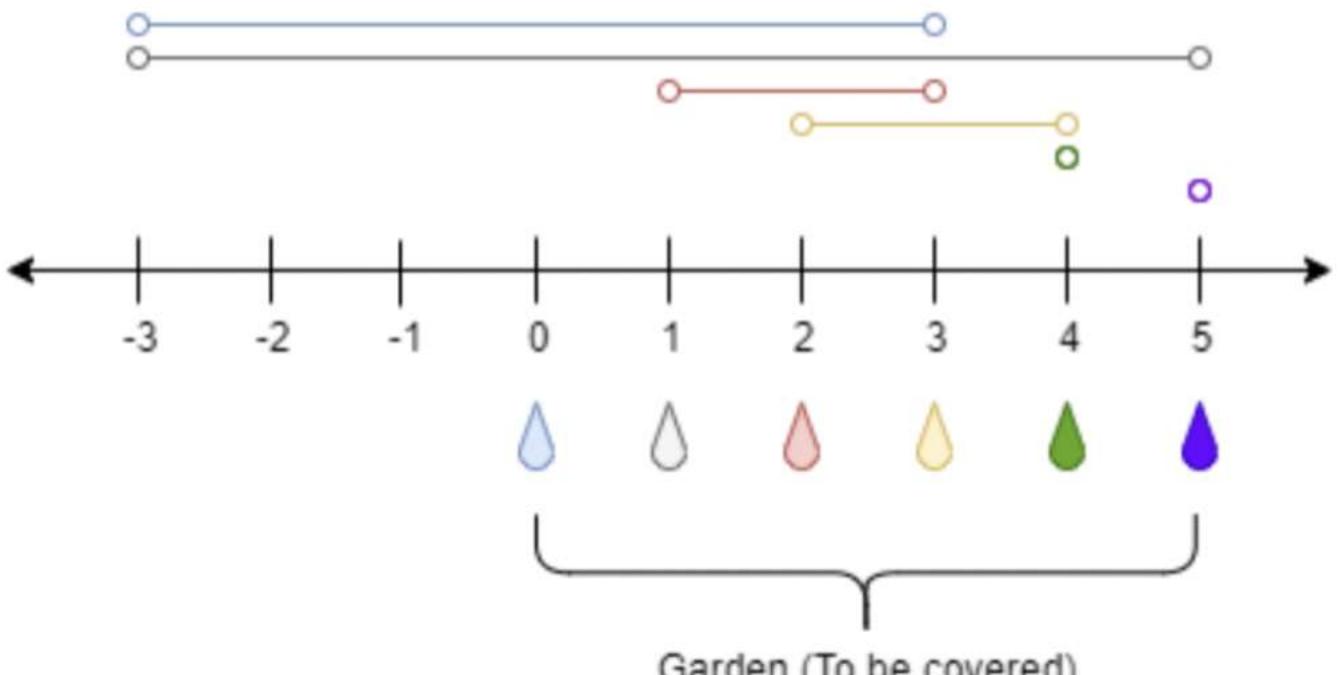
    public int minSteps(int N) {
        int[] dp = new int[N + 1];
        Arrays.fill(dp, -1);

        return memo(N, dp);
    }
}
```



Greedy fail
smaller the no, better the answer

~~eg)~~
Example 1:



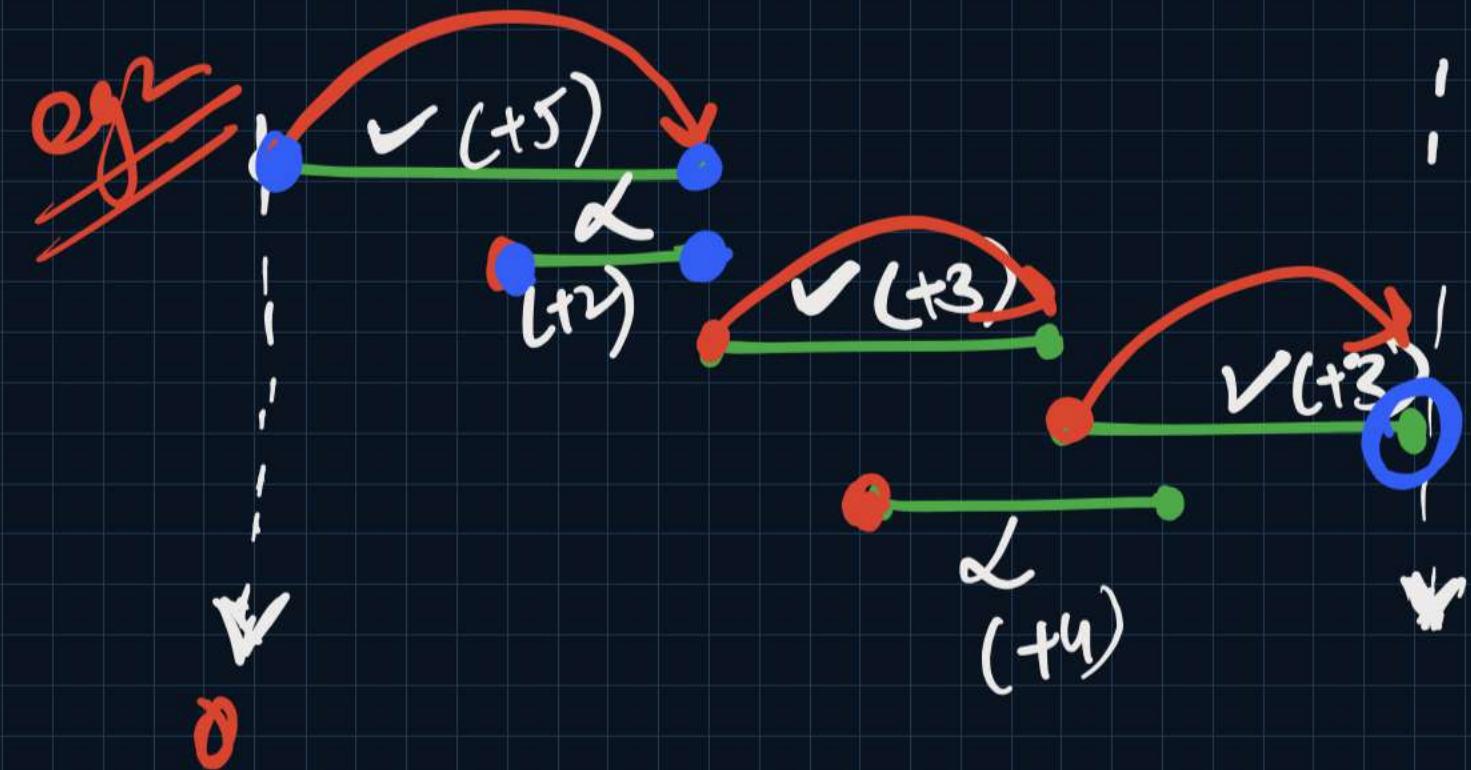
Input: $n = 5$, ranges = [3, 4, 1, 1, 0, 0]



1 → 2

2 → 3

Minimum Taps to Garden



Answer = ③

3 → 4

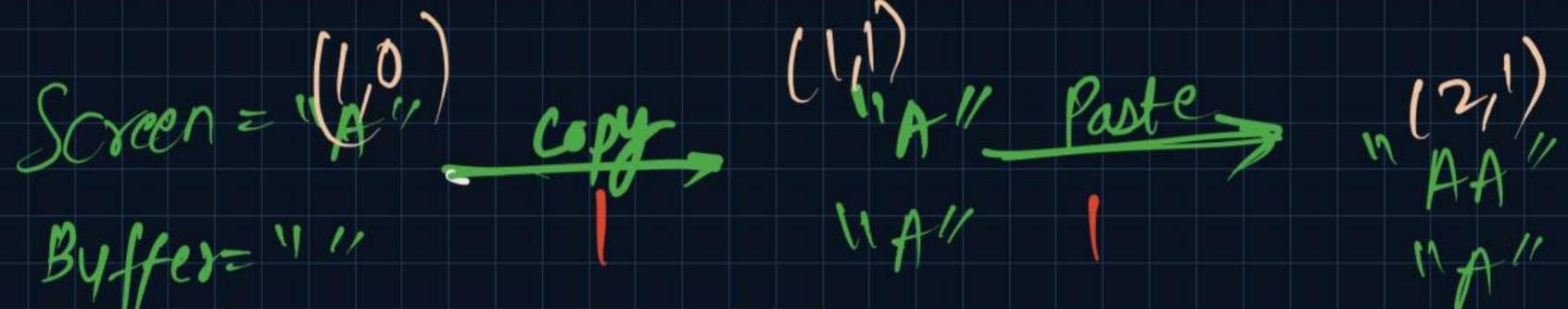
4 → 3



There is only one character '`'A'`' on the screen of a notepad. You can perform one of two operations on this notepad for each step:

- Copy All: You can copy all the characters present on the screen (a partial copy is not allowed).
- Paste: You can paste the characters which are copied last time.

Given an integer `n`, return *the minimum number of operations to get the character '`'A'`' exactly `n` times on the screen.*



```

class Solution {
    public long memo(int screen, int buffer, long dest, long[][] dp){
        if(screen > dest) return Integer.MAX_VALUE;
        if(screen == dest) return 0;
        if(dp[screen][buffer] != -1)
            return dp[screen][buffer];
        long copyPaste = 2l + memo(2 * screen, screen, dest, dp);
        long paste = 1l + memo(screen + buffer, buffer, dest, dp);

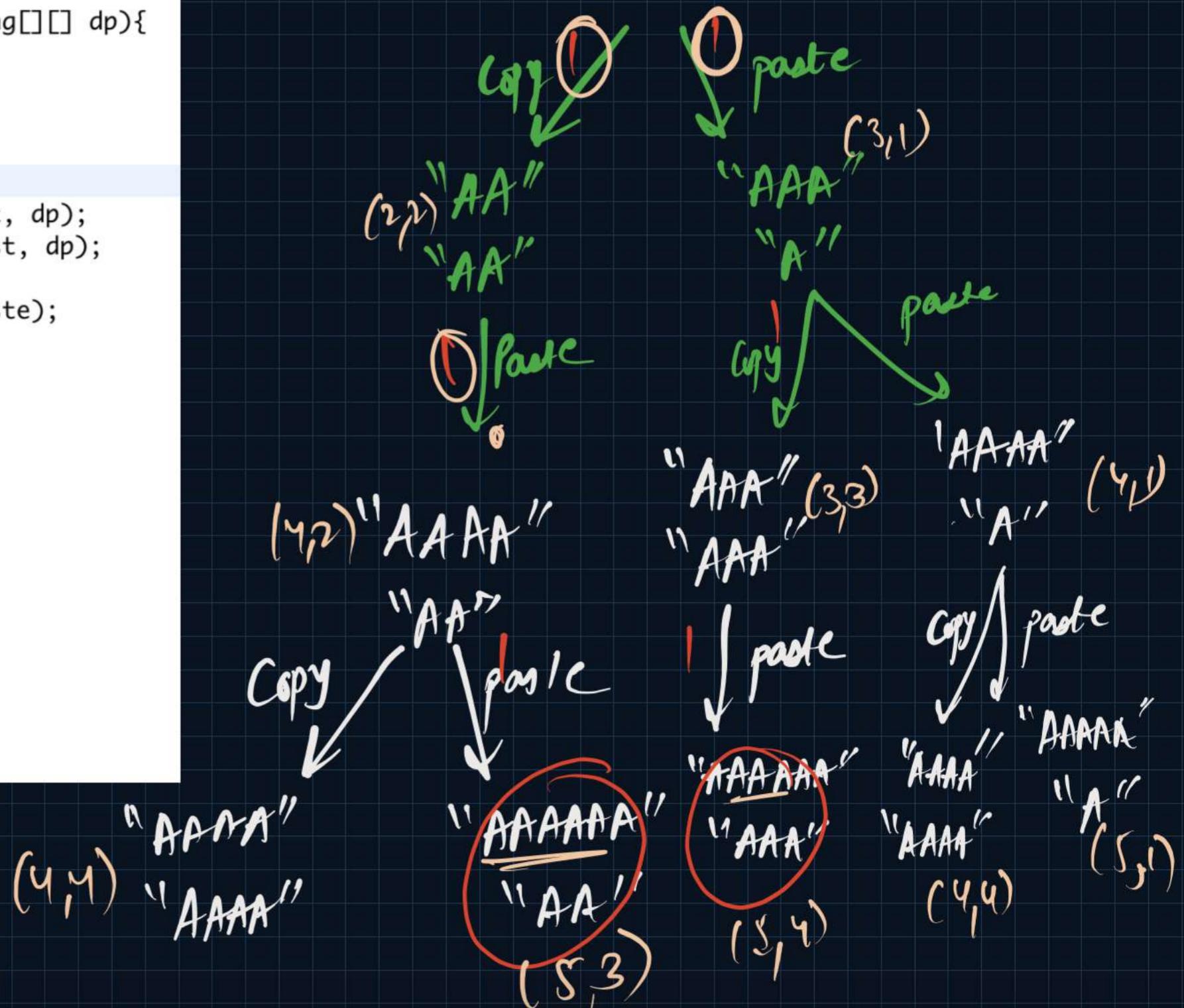
        return dp[screen][buffer] = Math.min(paste, copyPaste);
    }
    public int minSteps(int n) {
        if(n == 1) return 0;

        long[][] dp = new long[2 * n + 1][2 * n + 1];
        for(int i=0; i<=2*n; i++){
            for(int j=0; j<=2*n; j++){
                dp[i][j] = -1;
            }
        }

        return (int)(1l + memo(1, 0, n, dp));
    }
}

```

TC $\rightarrow O(N^2)$



Dynamic Programming - Lecture 4

{ 24 April, Sunday }
9 AM - 12 PM

- as Consecutive 1's not allowed X
- ↳ Decode Ways - LeetCode X
- as Ways To Tile A Floor | Pratik X
- as Count the number of ways X
- as Friends Pairing Problem | X
- ↳ Ugly Number II - LeetCode X

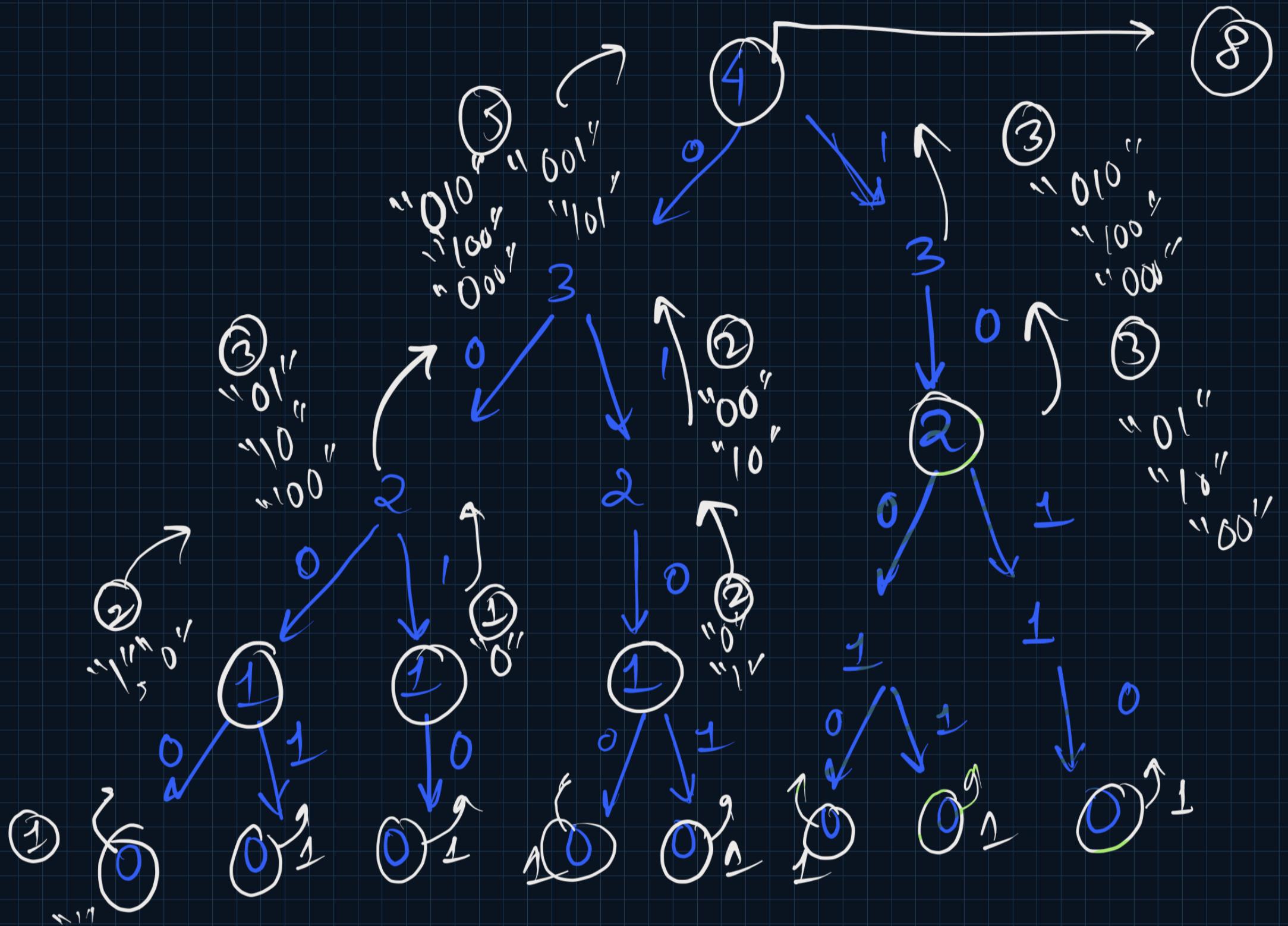
{ Fibonacci Variations }

① Count valid binary strings

(2) ① → "0", "1"

(3) ② → "00", "01", "10", "~~"11"~~

(5) ③ → "000", "001", "010", "~~"011"~~,
"100", "101", "~~"110"~~", "~~"111"~~



```

class Solution {
    int mod = 1000000007;

    long memo(int noOfDigits, int prevDigit, long[][] dp){
        if(noOfDigits == 0) return 1; // Empty String
        if(dp[noOfDigits][prevDigit] != 0)
            return dp[noOfDigits][prevDigit];

        long appending0 = memo(noOfDigits - 1, 0, dp);
        long appending1 = (prevDigit == 0) ? memo(noOfDigits - 1, 1, dp) : 0l;
        return dp[noOfDigits][prevDigit] = (appending0 + appending1) % mod;
    }

    long countStrings(int n) {
        long[][] dp = new long[n + 1][2];
        return memo(n, 0, dp);
    }
}

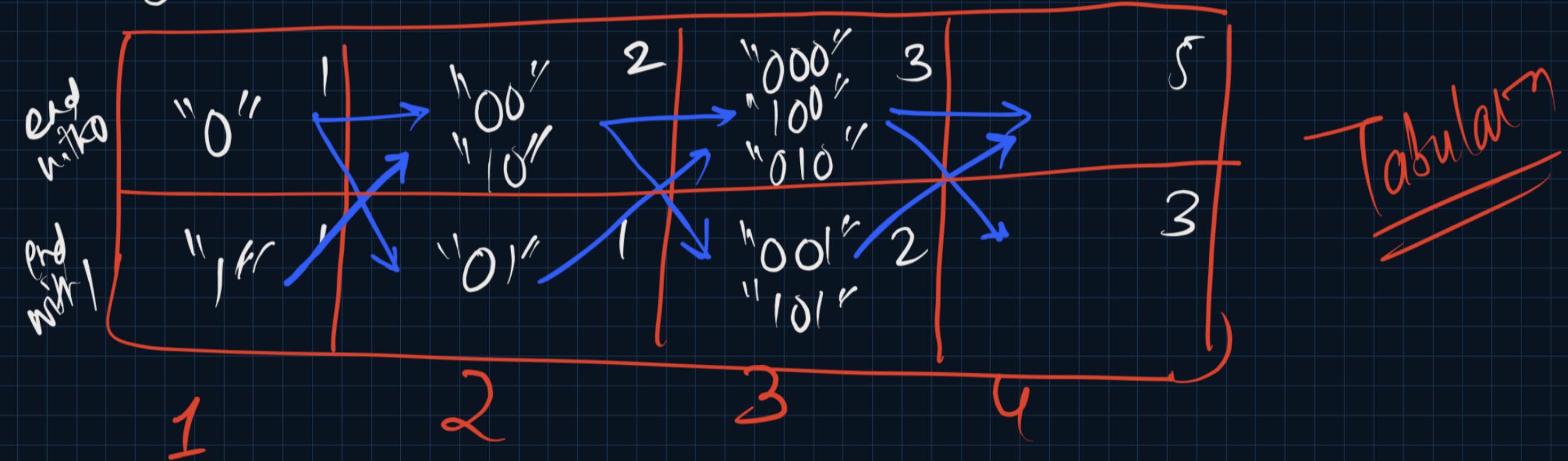
```

$T.C$
 $O(2 * N)$
 $\approx O(N)$

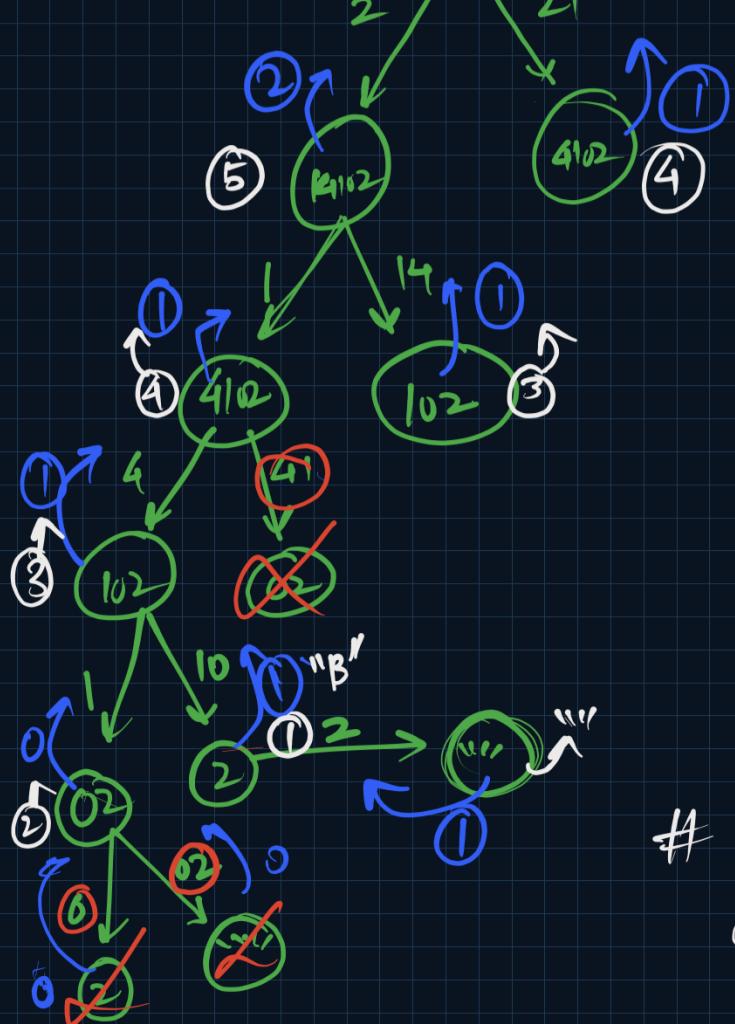
SC
 ↴ ↘
 Recs Extra Space
 ↓ ↓
 $O(N)$ $O(2 * N)$
 $= O(N)$

smaller Problem

Large Problem



⑥ "214102" $\nearrow_{2+1} = ③$



② Covert Encodings { Decode Ways }

1	10	20
2	11	21
3	12	22
4	1	23
;	;	24
9	19	25
		26

count
(ways)

deci-
↓
test
...
0

min →
steps
(length)

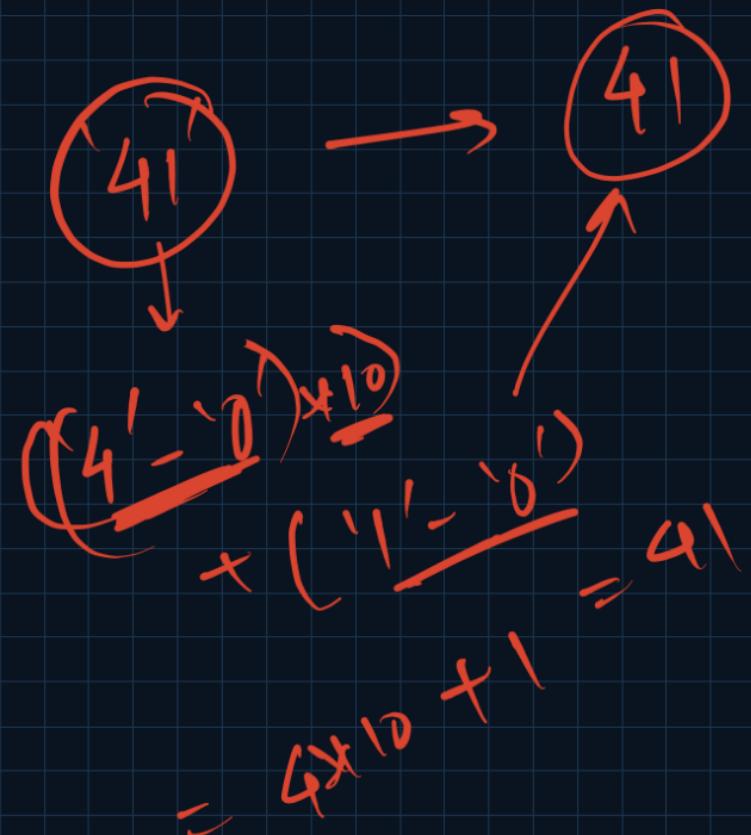
DP table can be made
by taking indexing as length
of input string

```
class Solution {
    public int memo(String s, int idx, int[] dp){
        if(idx == s.length()) return 1;
        if(dp[idx] != -1) return dp[idx];

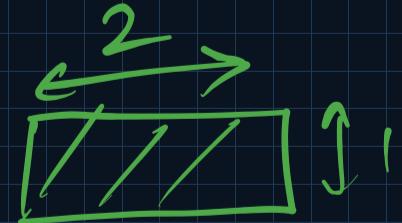
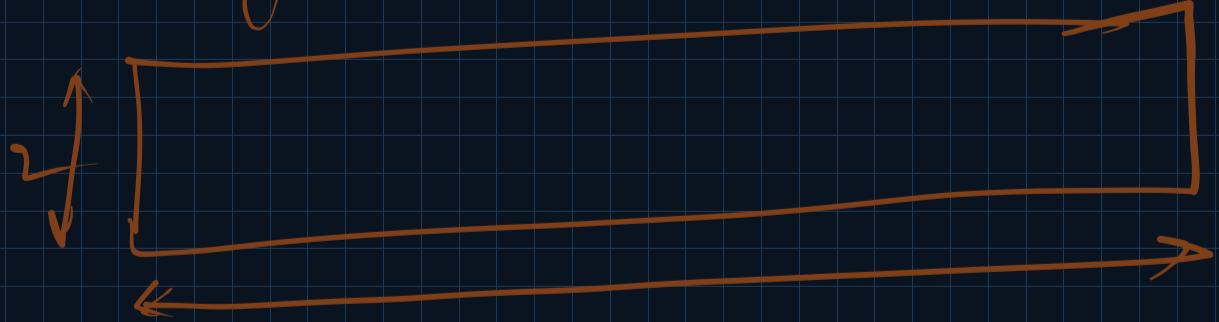
        int ans1 = 0, ans2 = 0;
        if(s.charAt(idx) != '0'){
            ans1 = memo(s, idx + 1, dp);
        }

        if(idx + 1 < s.length()){
            int code = (s.charAt(idx) - '0') * 10 + (s.charAt(idx + 1) - '0');
            if(code >= 10 && code <= 26){
                ans2 = memo(s, idx + 2, dp);
            }
        }

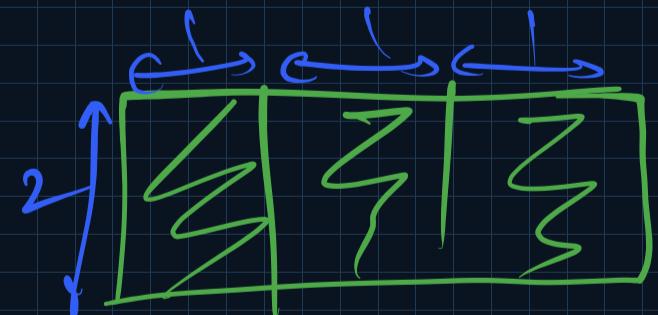
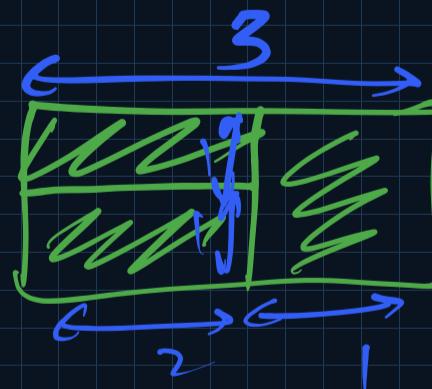
        return dp[idx] = ans1 + ans2;
    }
}
```

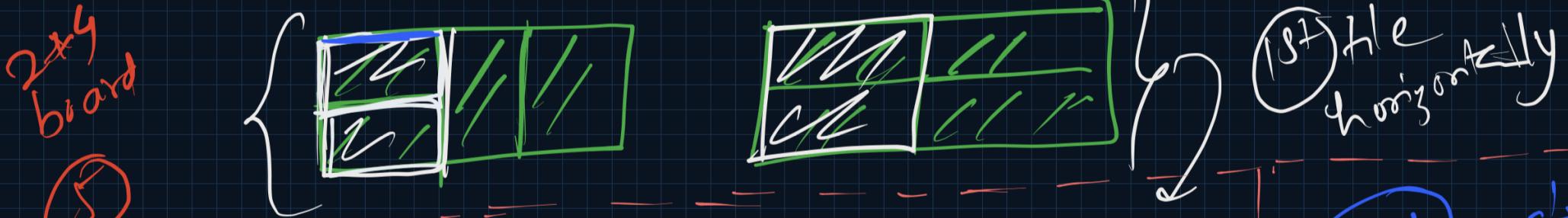
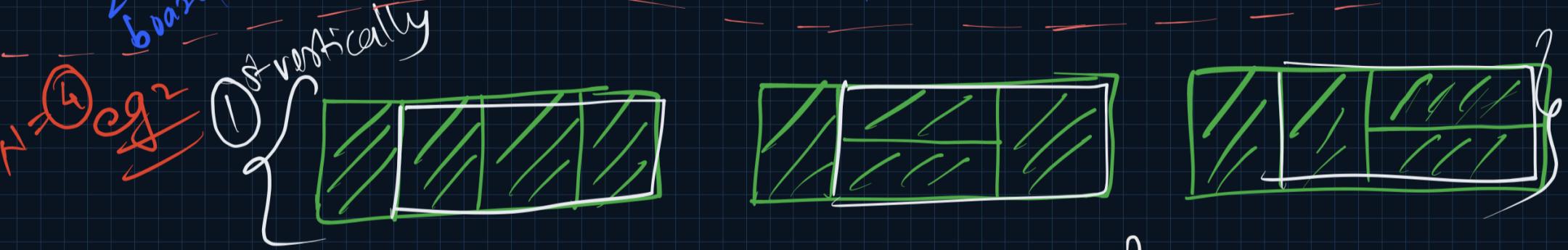


Tiling Problem - I



N = 3
cg
 2×3
board
isoptically





$N=4$

Fibonacci

⑤

$N=2$ eq³

2×2

②



$N=1$



Fibonacci

$N=4$

vertically

horizontally

$N=2$

$N=3$

vertically

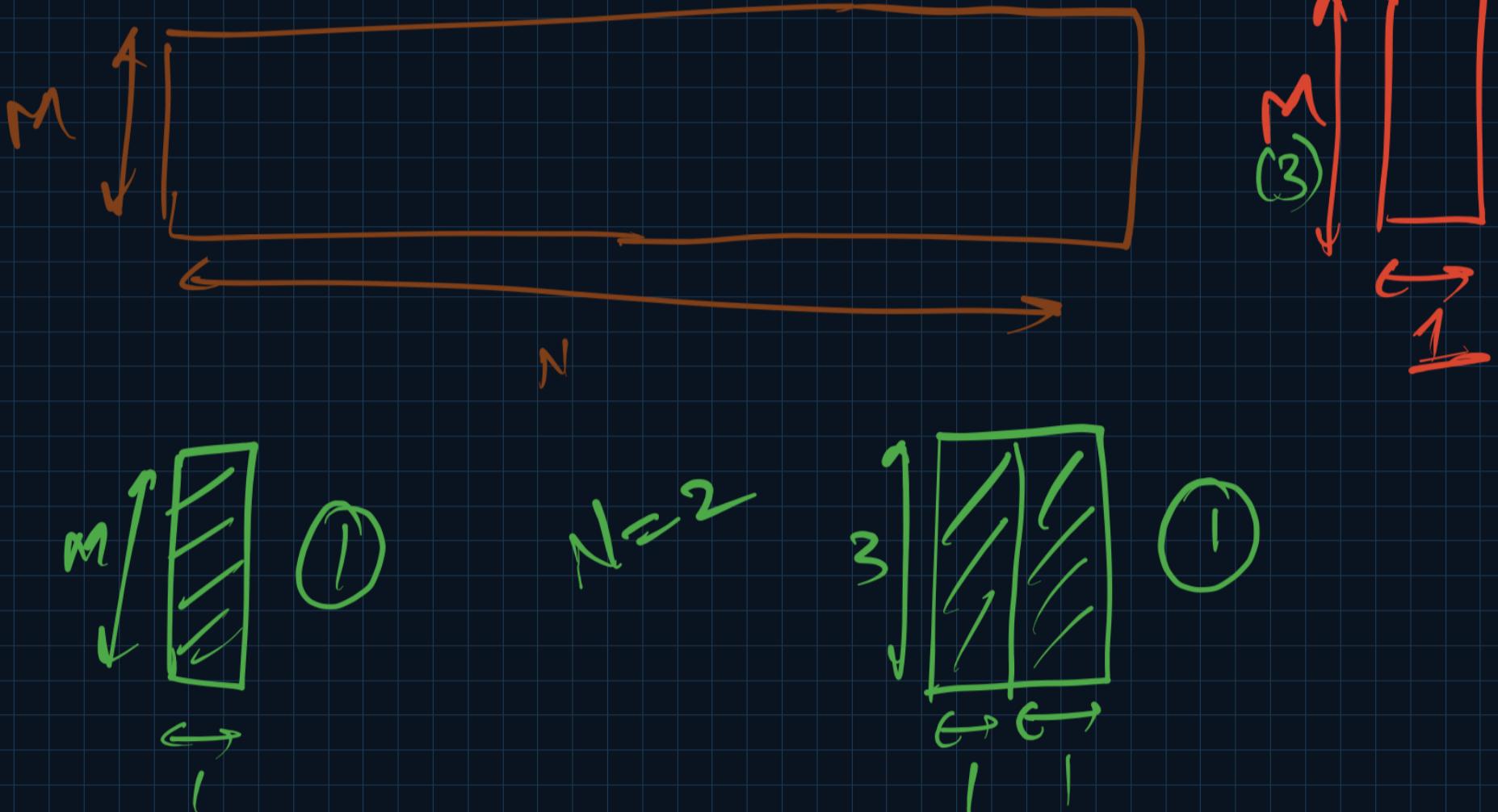
horizontally

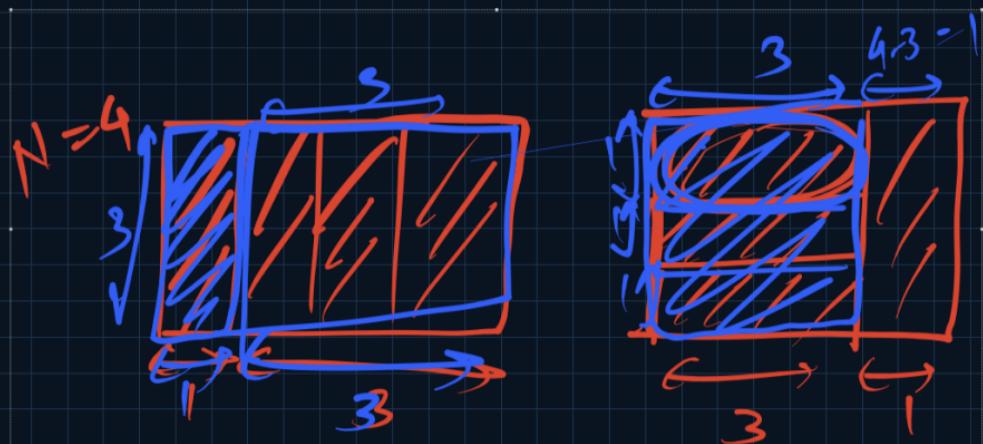
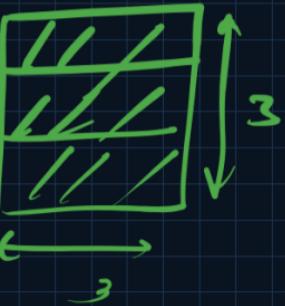
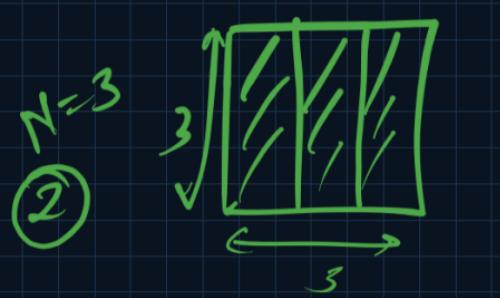
$N=1$

$N=2$

V H
 $N=1$ $N=0$

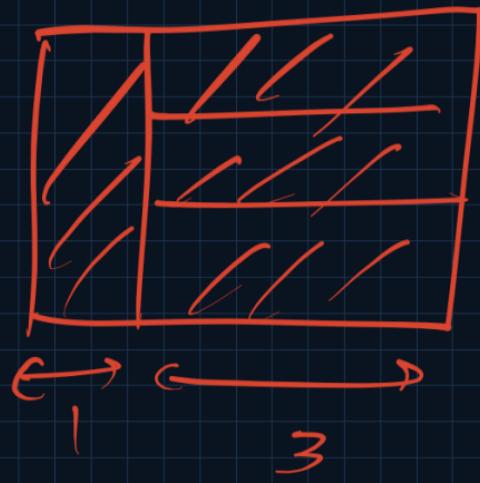
Tiling Problem - II





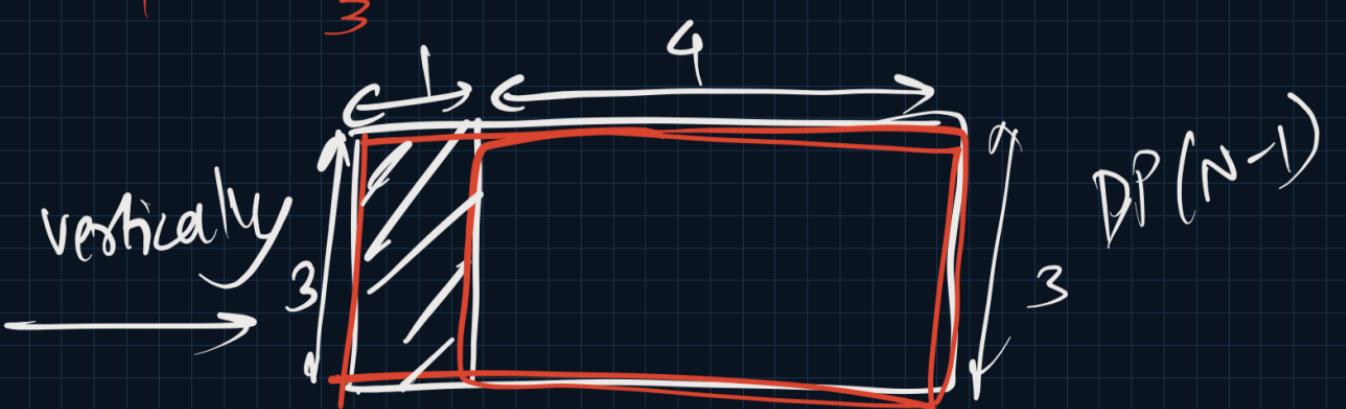
$$DP(N) = DP(N-1) + DP(N-2)$$

↑ ↑
vertically horizontally

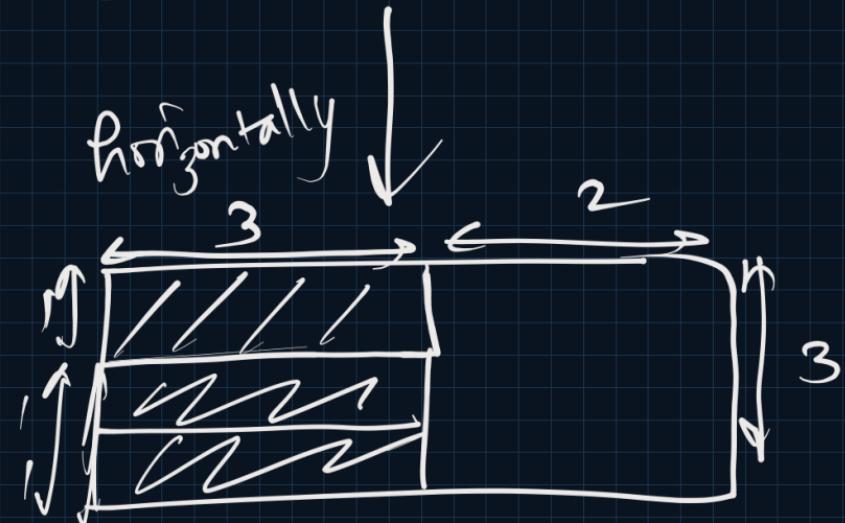


$$DP(N-4) = DP(N-3) + DP(N-1)$$

$N \times S$



$DP(N-M)$



$$DP(N) = \frac{DP(N-1)}{DP(N-M)}$$

$M=4$

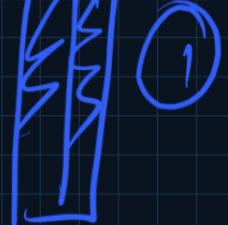
Base Cases

If $N < M$
count = ①

$N=1$



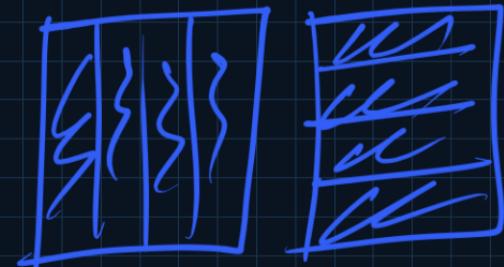
$N=2$



$N=3$



$N=4$



②

If $N = M$
count = ②

```

int mod = 1000000007;
public int memo(int n, int m, int[] dp){
    if(n < m) return 1;
    if(n == m) return 2;
    if(dp[n] != -1) return dp[n];

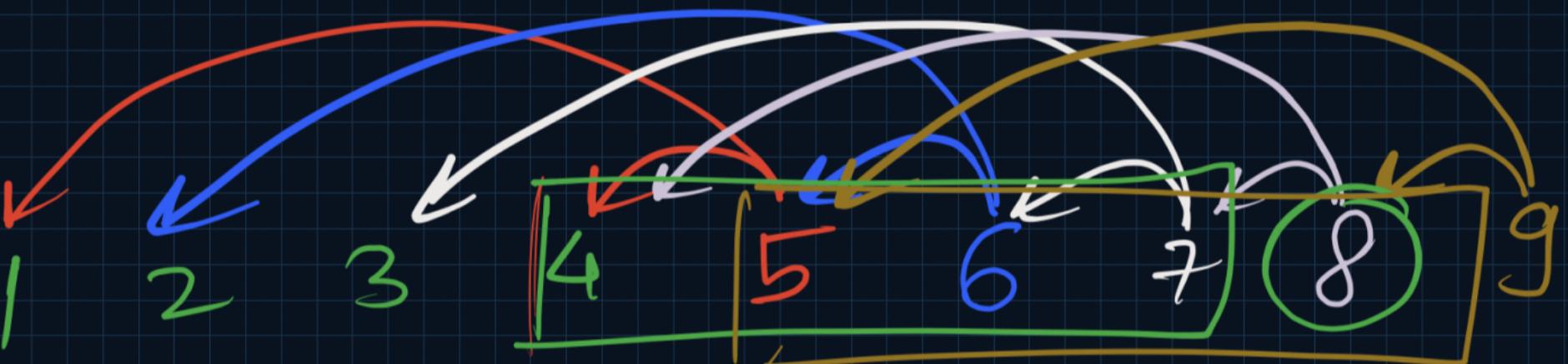
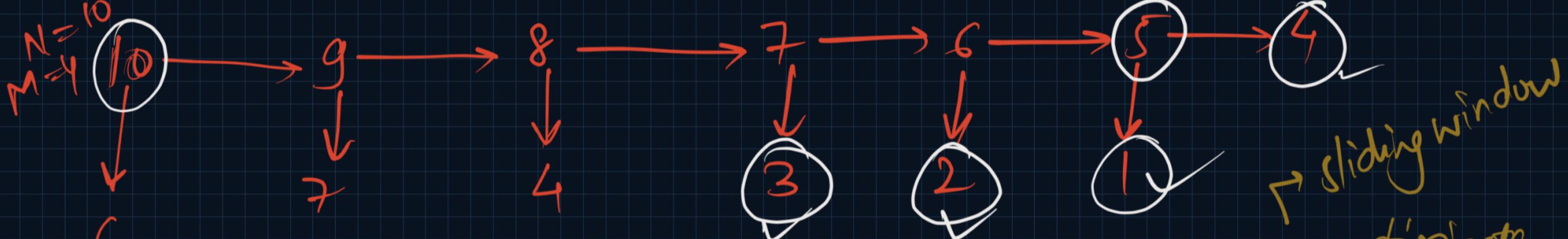
    int ans1 = memo(n - 1, m, dp);
    int ans2 = memo(n - m, m, dp);
    return dp[n] = (ans1 + ans2) % mod;
}

public int countWays(int n, int m)
{
    int[] dp = new int[n + 1];
    Arrays.fill(dp, -1);
    return memo(n, m, dp);
}

```

Time : $\rightarrow O(N)$

Space : $\rightarrow O(N)$
 ↴ R.C.S ↴ DP Table



↑ sliding window

space optimization



$O(M)$

```

public int countWays(int n, int m)
{
    if(n < m) return 1;
    if(n == m) return 2;
    // int[] dp = new int[n + 1];
    // Arrays.fill(dp, -1);
    // return memo(n, m, dp);

    Deque<Integer> dp = new ArrayDeque<>();
    for(int i=1; i<m; i++){
        dp.add(1); // DP[N < M]
    }
    dp.add(2); // DP[N == M]

    for(int i=m+1; i<=n; i++){
        int ans = (dp.getFirst() + dp.getLast()) % mod;
        dp.removeFirst();
        dp.addLast(ans);
    }

    return dp.getLast();
}

```

A hand-drawn diagram on a grid background. It shows two arrows pointing from handwritten text to mathematical notations. The top arrow points from the word "Time" to the notation $O(N)$. The bottom arrow points from the word "Space" to the notation $O(M)$.



House Robber

man sum

$$\{ \checkmark 10, \cancel{20}, \checkmark 60, \cancel{40}, \cancel{30}, \checkmark 100 \}$$

```
graph TD; Greedy[Greedy] --> Alternating[Alternating]; Alternating --> fail[fail]
```

```

class Solution {
    public int memo(int[] nums, int idx, int prev, int[][] dp){
        if(idx == nums.length) return 0;
        if(dp[idx][prev] != -1) return dp[idx][prev];

        int yes = (prev == 0) ? (memo(nums, idx + 1, 1, dp) + nums[idx]) : 0;
        int no = memo(nums, idx + 1, 0, dp);

        return dp[idx][prev] = Math.max(yes, no);
    }

    public int rob(int[] nums) {
        int n = nums.length;
        int[][] dp = new int[n + 1][2];
        for(int i=0; i<=n; i++){
            dp[i][0] = -1;
            dp[i][1] = -1;
        }

        return memo(nums, 0, 0, dp);
    }
}

```

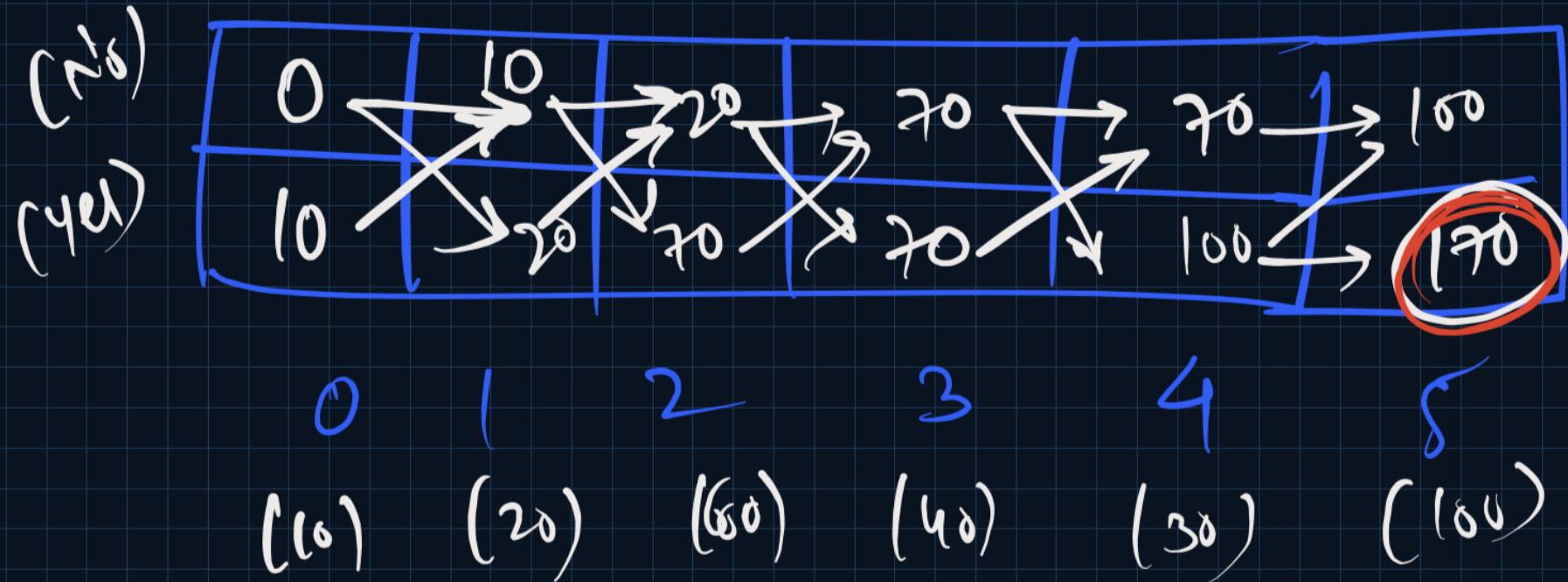
Memo

→ $O(N)$ Time Comp

→ $O(N)$ Space Comp

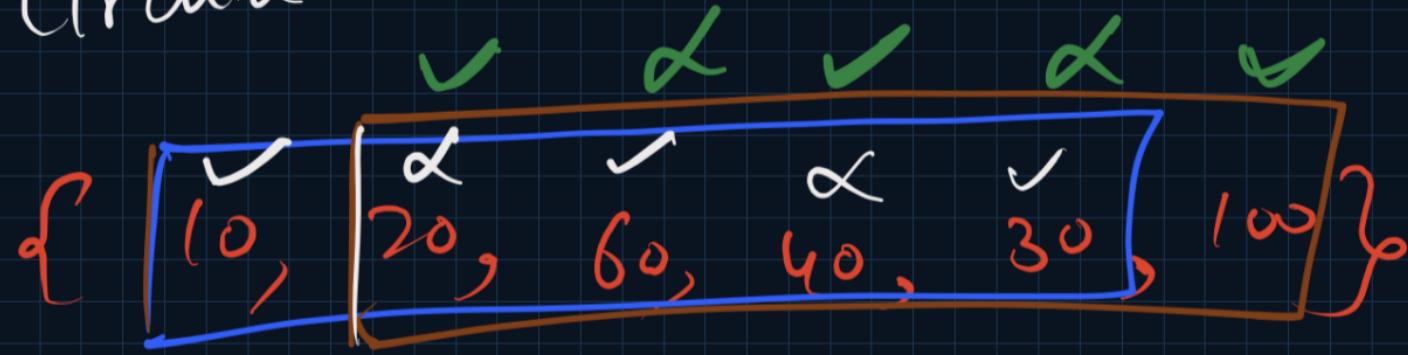
↙
Entry ↘
R-C-S

smaller \rightarrow larger



Homework: \Rightarrow Tabulation (Code)

House Robber - Circular



Extra constraint → 10 & 100 are
adjacent

arr[0] & arr[n-1]
are also adjacent

```

public int memo(int[] nums, int idx, int n, int prev, int[][] dp){
    if(idx == n + 1) return 0;
    if(dp[idx][prev] != -1) return dp[idx][prev];

    int yes = (prev == 0) ? (memo(nums, idx + 1, n, 1, dp) + nums[idx]) : 0;
    int no = memo(nums, idx + 1, n, 0, dp);

    return dp[idx][prev] = Math.max(yes, no);
}

public int rob(int[] nums, int start, int end){
    int n = nums.length;

    int[][] dp = new int[n + 1][2];
    for(int i=0; i<=n; i++){
        dp[i][0] = -1;
        dp[i][1] = -1;
    }

    return memo(nums, start, end, 0, dp);
}

```

```

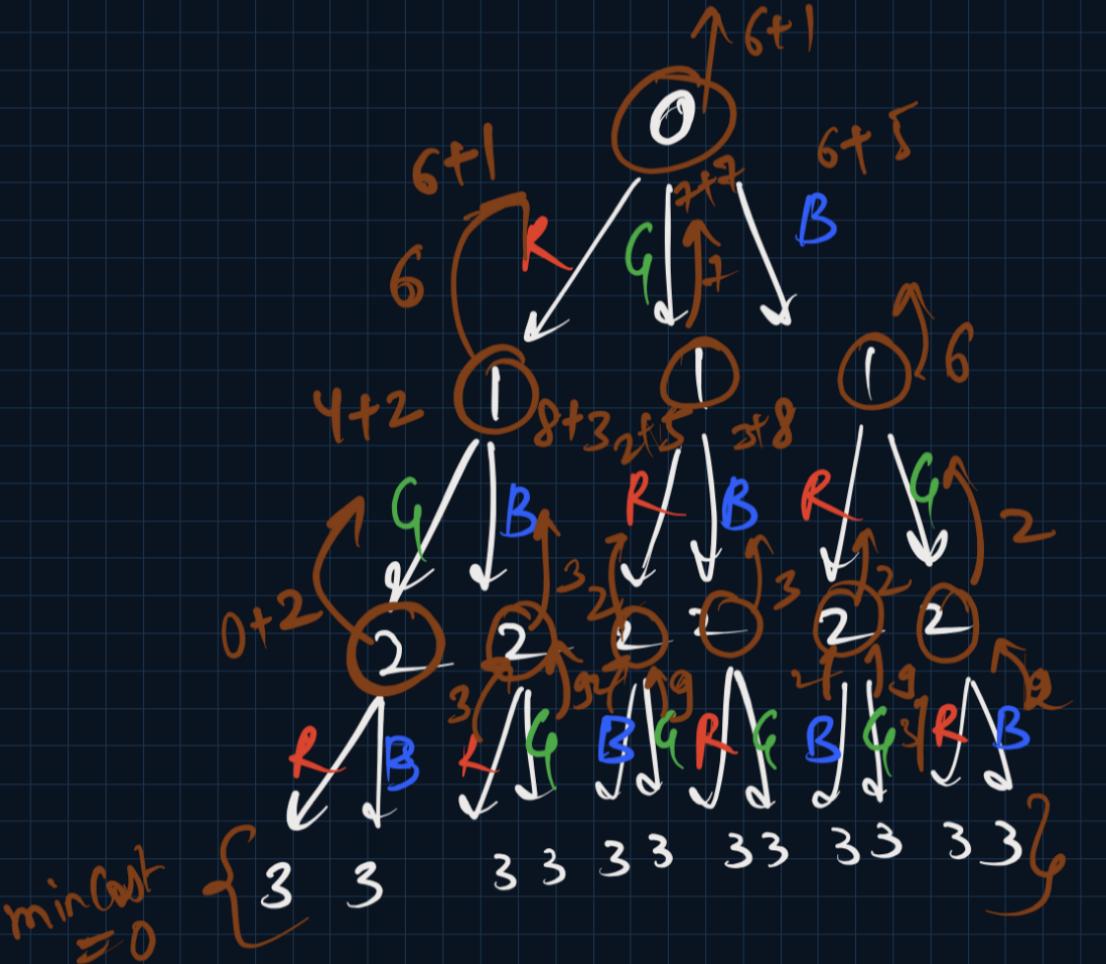
public int rob(int[] nums) {
    if(nums.length == 0) return 0;
    if(nums.length == 1) return nums[0];

    return Math.max(rob(nums, 0, nums.length - 2)
                    , rob(nums, 1, nums.length - 1));
}

```

TC $\Rightarrow O(N^2)$
SC $\Rightarrow O(n)$

(R)	1	5	3	
(B)	5	8	2	
(G)	7	4	9	
0	1	2	3	



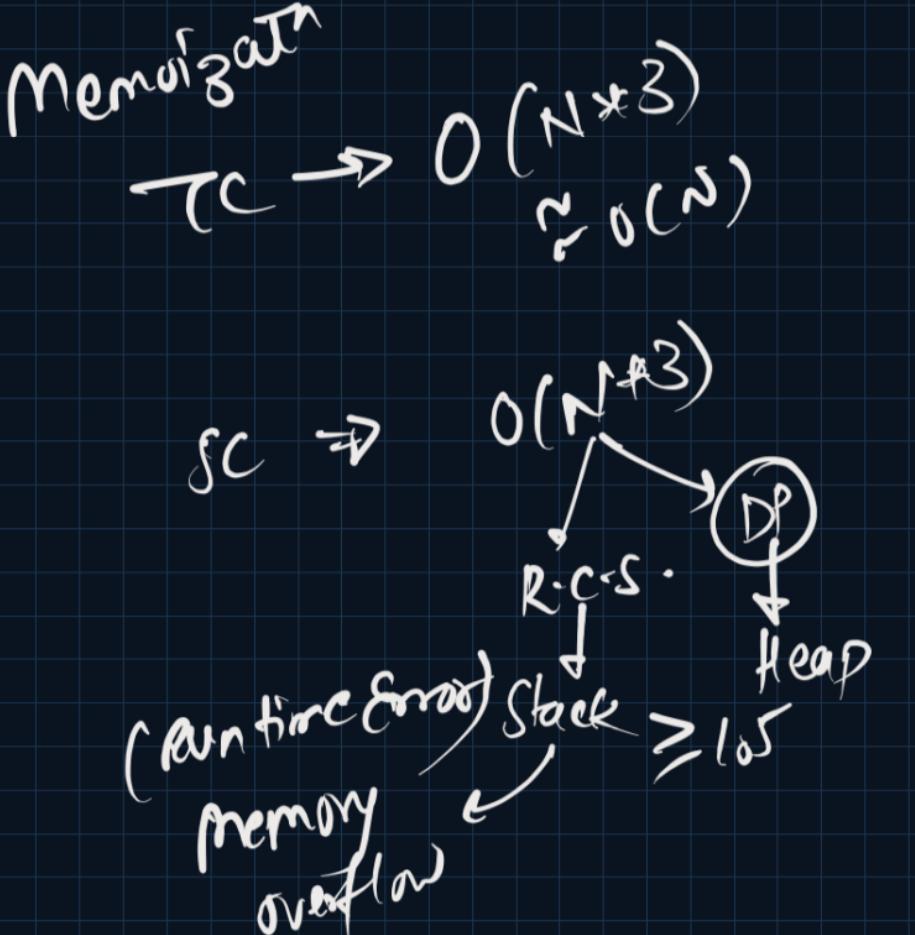
```

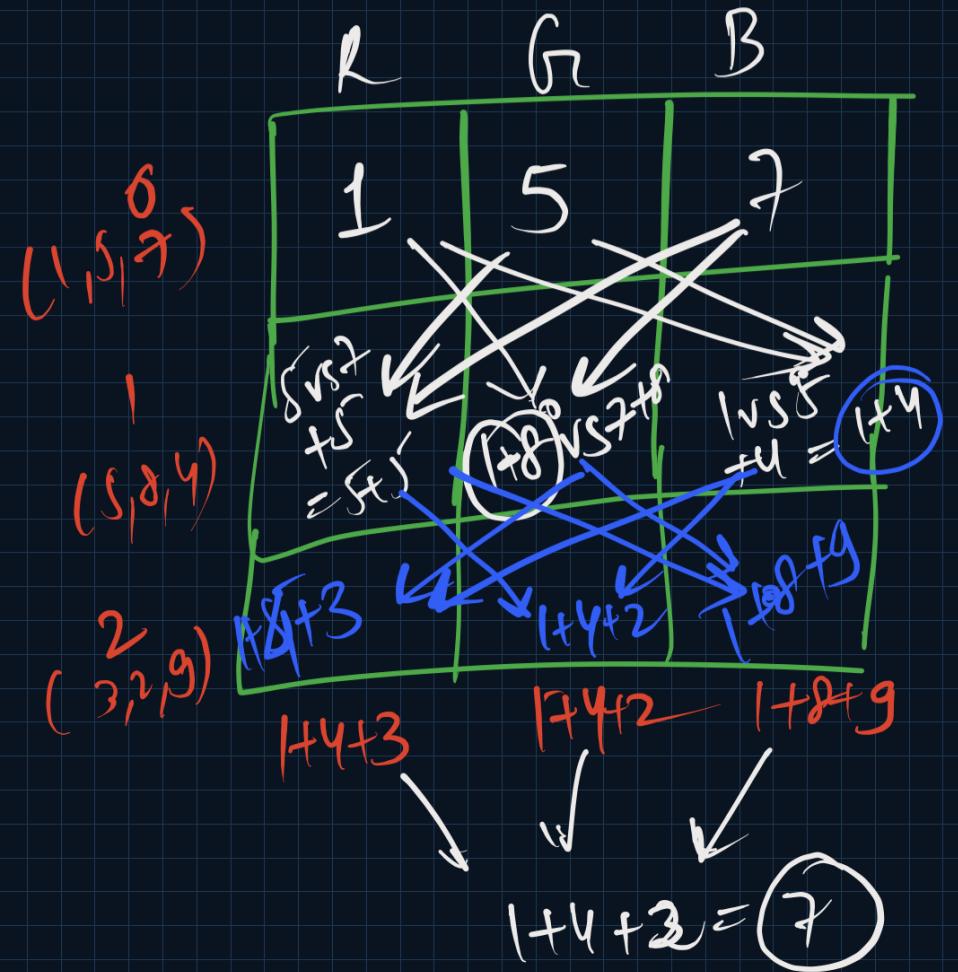
public int helper(int[][] costs, int idx, int prev, int[][] dp){
    if(idx == costs.length) return 0;
    if(prev >= 0 && dp[idx][prev] != -1) return dp[idx][prev];

    int ansR = (prev == 0) ? Integer.MAX_VALUE
        : helper(costs, idx + 1, 0, dp) + costs[idx][0];
    int ansB = (prev == 1) ? Integer.MAX_VALUE
        : helper(costs, idx + 1, 1, dp) + costs[idx][1];
    int ansG = (prev == 2) ? Integer.MAX_VALUE
        : helper(costs, idx + 1, 2, dp) + costs[idx][2];

    if(prev == -1)
        return Math.min(ansR, Math.min(ansB, ansG));
    return dp[idx][prev] = Math.min(ansR, Math.min(ansB, ansG));
}

```





$DP(i, j)$ = (0^{-i}) houses
 paint min cost
 such that i^{th}
 house has color j^o

- ① Storage & meaning
- ② Smaller, larger problem
- ③ Travel & solve
 - { Recurrence relation }

```
int[][] dp = new int[n + 1][3];
dp[0][0] = costs[0][0];
dp[0][1] = costs[0][1];
dp[0][2] = costs[0][2];

for(int i=1; i<n; i++){
    // (0 - i houses) -> ith House Red
    dp[i][0] = costs[i][0] + Math.min(dp[i - 1][1], dp[i - 1][2]);

    // (0 - i houses) -> ith House Blue
    dp[i][1] = costs[i][1] + Math.min(dp[i - 1][0], dp[i - 1][2]);

    // (0 - i houses) -> ith House Green
    dp[i][2] = costs[i][2] + Math.min(dp[i - 1][0], dp[i - 1][1]);
}

return Math.min(dp[n - 1][0], Math.min(dp[n - 1][1], dp[n - 1][2]));
}
```

$T_C \Rightarrow O(N^3) = O(n)$

$SC \Rightarrow O(N^3) = O(n)$

↓

“Stack overflow”
is not
occurring,

```

int prev0 = costs[0][0];
int prev1 = costs[0][1];
int prev2 = costs[0][2];

for(int i=1; i<n; i++){
    // (0 - i houses) -> ith House Red
    int curr0 = costs[i][0] + Math.min(prev1, prev2);

    // (0 - i houses) -> ith House Blue
    int curr1 = costs[i][1] + Math.min(prev0, prev2);

    // (0 - i houses) -> ith House Green
    int curr2 = costs[i][2] + Math.min(prev0, prev1);

    prev0 = curr0; prev1 = curr1; prev2 = curr2;
}

return Math.min(prev0, Math.min(prev1, prev2));

```

"Space Optimization"
 in Tabulation
 ↗ $O(n)$ Time
 $O(1)$ Space

Paint House - II

cost

DP

house
(N)

colors = $S \cup k$

A

0

1

2

3

4

B

6
 $1+6$

2
 $2+1$

3
 $2+3$

4
 $1+4$

1
 $1+1$

C

2
 $1+1+2$

2
 $1+1+2$

1
 $1+1+1$

1
 $1+1+1$

2
 $2+1+2$

D

3
 $1+1+1$
 $+3$

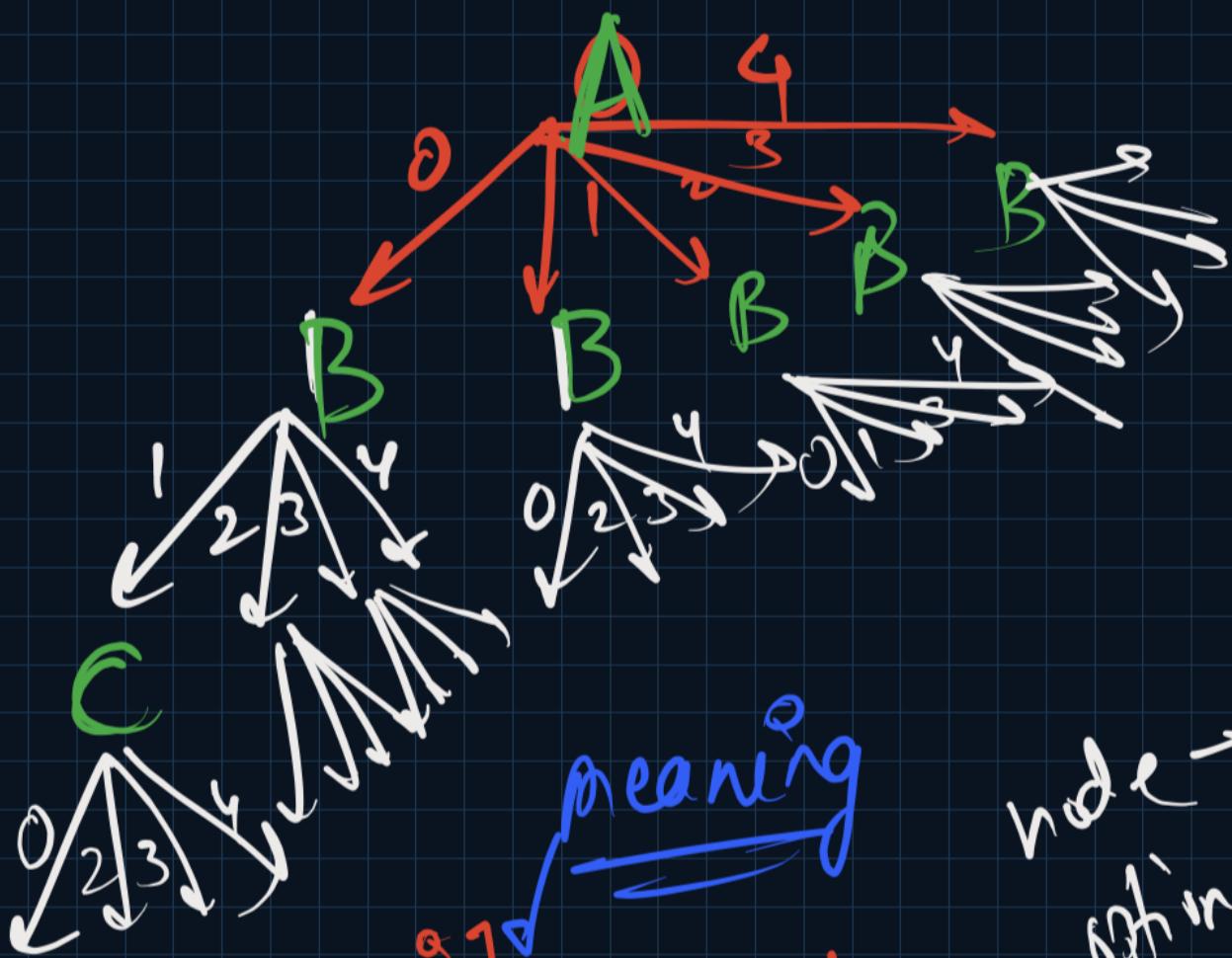
1
 $1+1+1$
 $+1$

3
 $1+1+1$
 $+3$

4
 $1+1+1$
 $+4$

4
 $1+1+1$
 $+4$

min cost



meaning

node \rightarrow house
edge \rightarrow color
graph \rightarrow cost

$P[i][j]$

$= P[i-1][j]$ house
min cost
such that
ith house
has j colors

```

int[][] dp = new int[costs.length + 1][k];
for(int c=0; c<k; c++){
    dp[0][c] = costs[0][c];
}

for(int i=1; i<n; i++){
    for(int c=0; c<k; c++){
        int min = Integer.MAX_VALUE;

        // Extracting Min of Previous Row Excluding Our Column
        for(int prev=0; prev<k; prev++){
            if(prev == c) continue;
            min = Math.min(min, dp[i - 1][prev]);
        }

        dp[i][c] = costs[i][c] + min;
    }

    int min = Integer.MAX_VALUE;
    for(int c=0; c<k; c++){
        min = Math.min(dp[n - 1][c], min);
    }
    return min;
}

```

Time Comp: $O(N * K^2)$

Space Comp: $O(n * k)$

For calculating every cell value we are looping on the previous row $O(k)$

0 1 2 3 4
 A 2 2 4 4 1 1 3 3 5 5
 1st min = 1, 2nd min = 2

B 6 2 3 4 1
 1+6 1+2 2+3 4+4 1+x
 1st min = 1+1, 2nd min = 1+2

C 2 2 1 1 1 2
 1+1+2 1+1+1 1+1+1 1+2+2
 1st min = 1+1+1
 2nd min = 1+1+1

D 3 1 3 4 4
 1+1+1 1+1+1 1+1+1 1+1+1
 1+3 1+3 1+4 1+4

```

for(int i=1; i<n; i++){
    int firstMin = Integer.MAX_VALUE;
    int secondMin = Integer.MAX_VALUE;
    for(int prev=0; prev<k; prev++){
        if(dp[i - 1][prev] <= firstMin){
            secondMin = firstMin;
            firstMin = dp[i - 1][prev];
        } else if(dp[i - 1][prev] < secondMin){
            secondMin = dp[i - 1][prev];
        }
    }

    for(int c=0; c<k; c++){
        int min = Integer.MAX_VALUE;
        // Extracting Min of Previous Row Excluding Our Column
        if(dp[i - 1][c] == firstMin)
            dp[i][c] = costs[i][c] + secondMin;
        else dp[i][c] = costs[i][c] + firstMin;
    }
}

```

finding $O(k)$

*1st min & 2nd min
of prev row*

*calculating current
dp row*

*TC $\Rightarrow O(n * (k + k))$
 $= O(n * 2k)$
 $\approx O(n * k)$*

Paint Fence

n , k colors
fence/houses

No more than 2 fences have the same color

$$\begin{array}{c} n=1 \rightarrow k \\ k=1 \end{array} \quad (1)$$

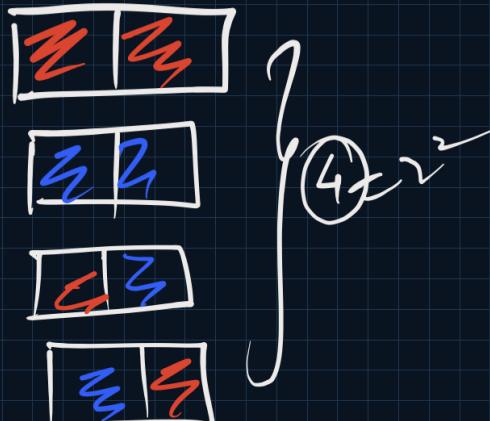
$$\begin{array}{c} n=2 \rightarrow k^2 \\ k=1 \end{array}$$

$$k=2 \quad (2)$$

$$k=2$$

$$k=3 \quad (3)$$

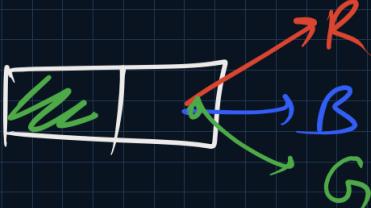
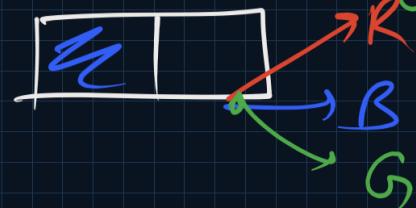
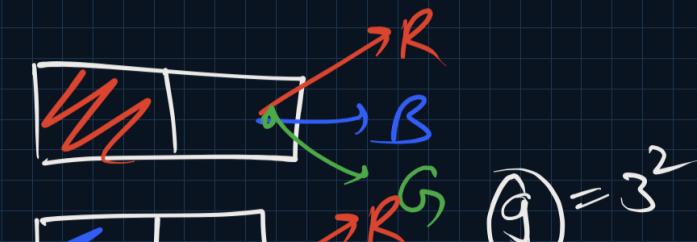
$$k=k \quad (k)$$



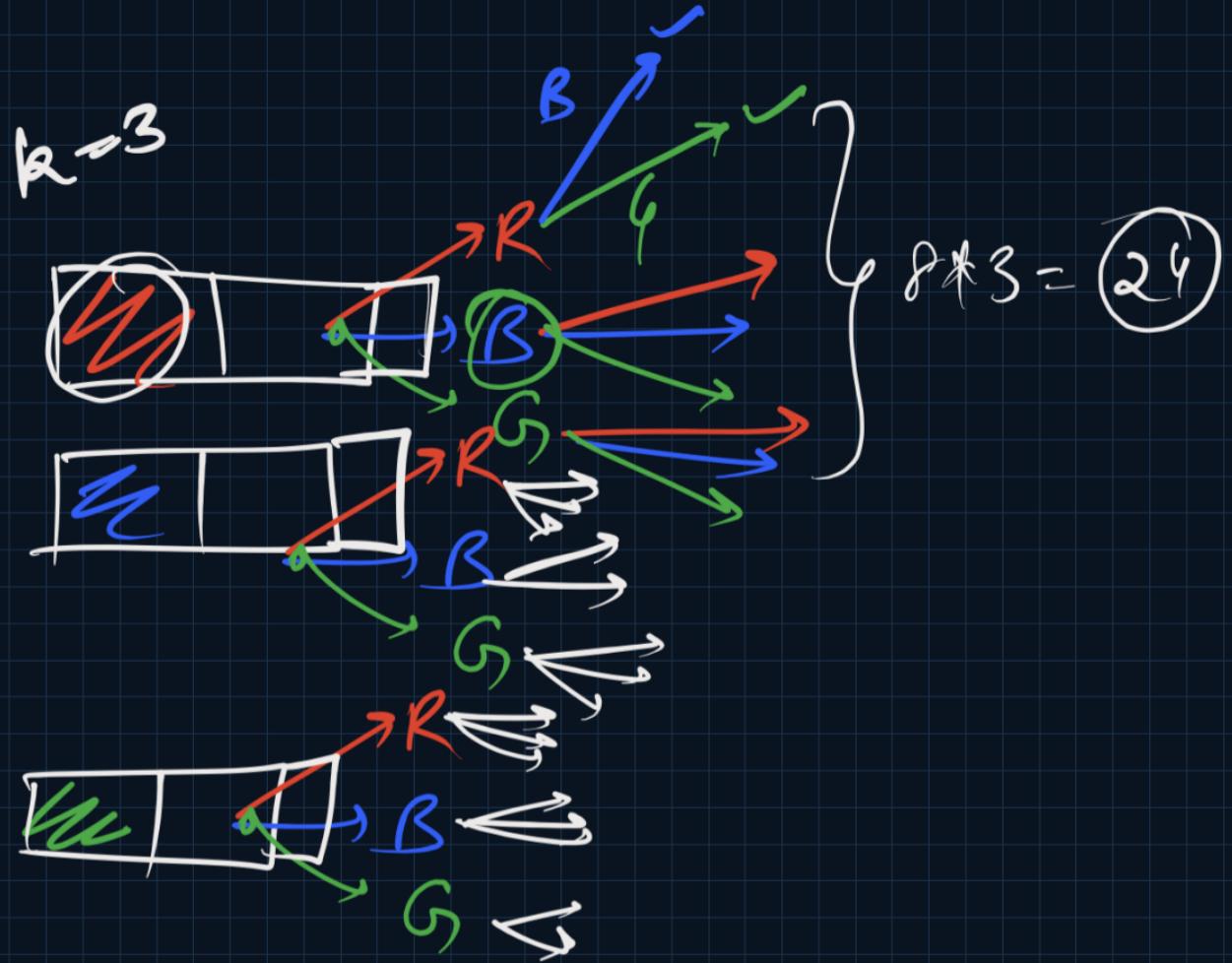
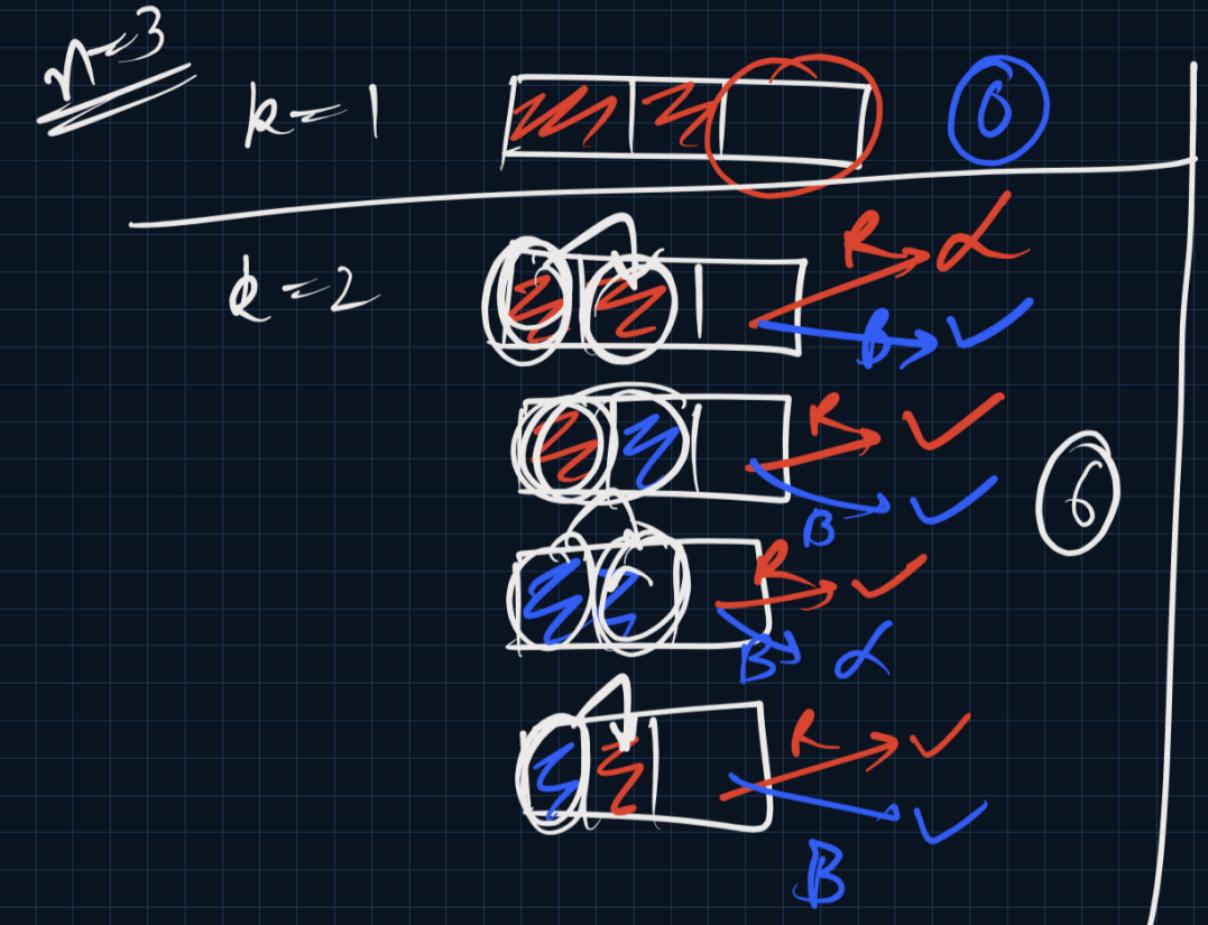
$$(1) = 1^2 \quad k=3$$



$$(4) = 2^2$$



$$k=k \quad \text{count of ways} = k^2 = k*k$$



$N=4$

single = 1 * $f(N-1)$

Diagram illustrating the recursive step for $N=4$. It shows all possible pairings of four people into two pairs. One specific pairing, where person 1 is paired with person 4 and person 2 is paired with person 3, is highlighted with a red circle.

Friends Pairing

$N=1$

① {1}

$N=2$

② {1, 2}, {1, 3}, {2, 3}

$N=3$

③ {1, 2, 3}, {1, 2}, {1, 3}, {2, 3}

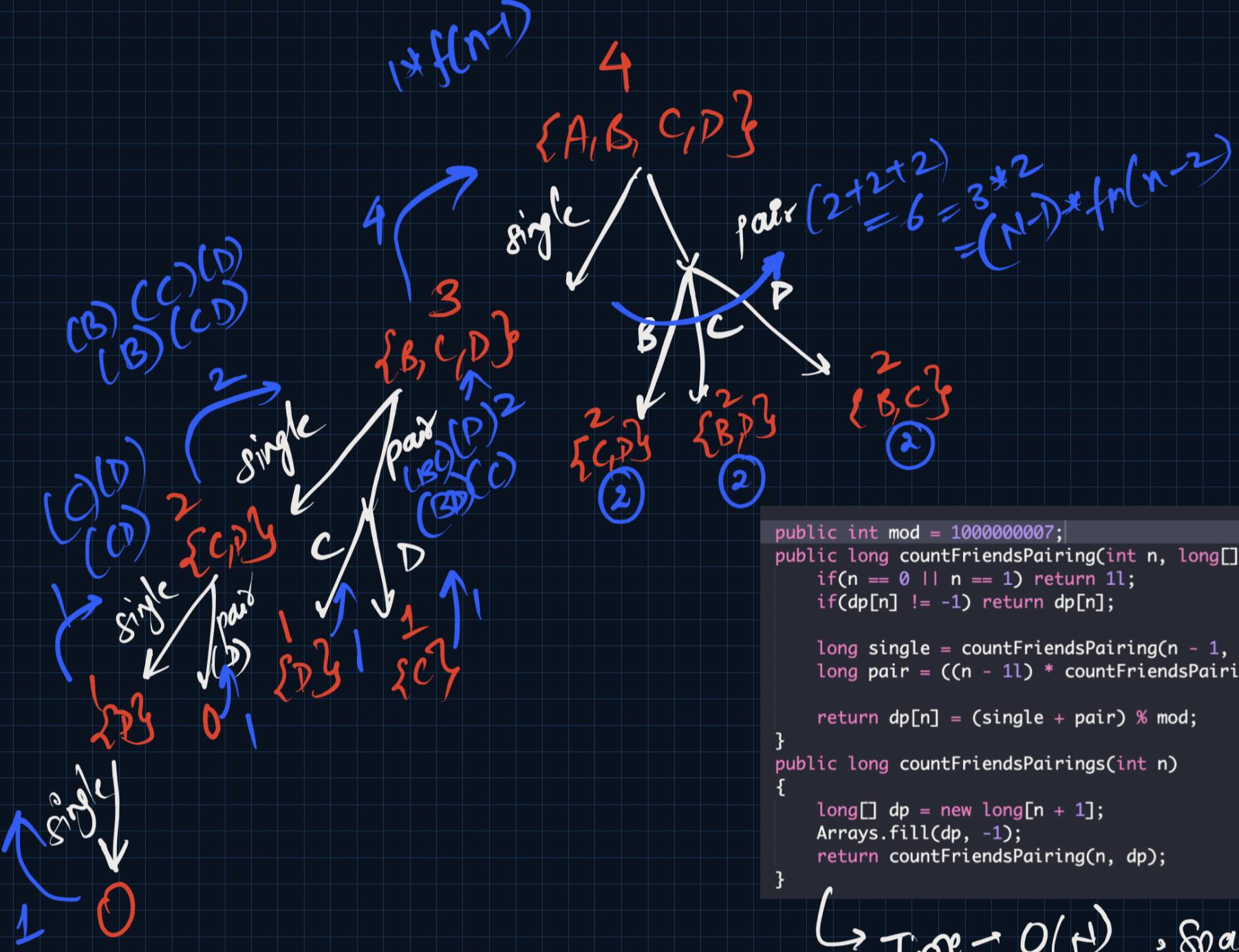
$f(N-1)$

$N=4$

④ {1, 2, 3, 4}, {1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}

$f(N-2)$

$f(N) = f(N-1) + (f(N-2) * (N-1))$



```

public int mod = 1000000007;
public long countFriendsPairing(int n, long[] dp){
    if(n == 0 || n == 1) return 1l;
    if(dp[n] != -1) return dp[n];

    long single = countFriendsPairing(n - 1, dp);
    long pair = ((n - 1l) * countFriendsPairing(n - 2, dp)) % mod;

    return dp[n] = (single + pair) % mod;
}
public long countFriendsPairings(int n)
{
    long[] dp = new long[n + 1];
    Arrays.fill(dp, -1);
    return countFriendsPairing(n, dp);
}

```

Time $\rightarrow O(N)$, Space $\rightarrow O(N)$

Space Optimizatr \rightarrow Possible \rightarrow Tabulation
{} Two Pointer }

Ugly No - |

$$N = p_1^a * b_2^b * b_3^c \dots$$

$$N = 2^a * 3^b * 5^c$$

where $\{ a, b, c \in [0, \infty) \}$

rotation

where $N_1 = 2^2 \times 3^2 \times 5^2 = 1$ ✓ true

$N_2 = 2^3 \times 3^4 \times 7^2 \neq 1$ false

Prime Factorization

Worst $O(\sqrt{N})$

Avg $O(\log_2 N + \log_3 N)$

Udy No - 11

loop * prime fact $\rightarrow O(N \times \sqrt{N})$ TC
 $N^{3/2}$

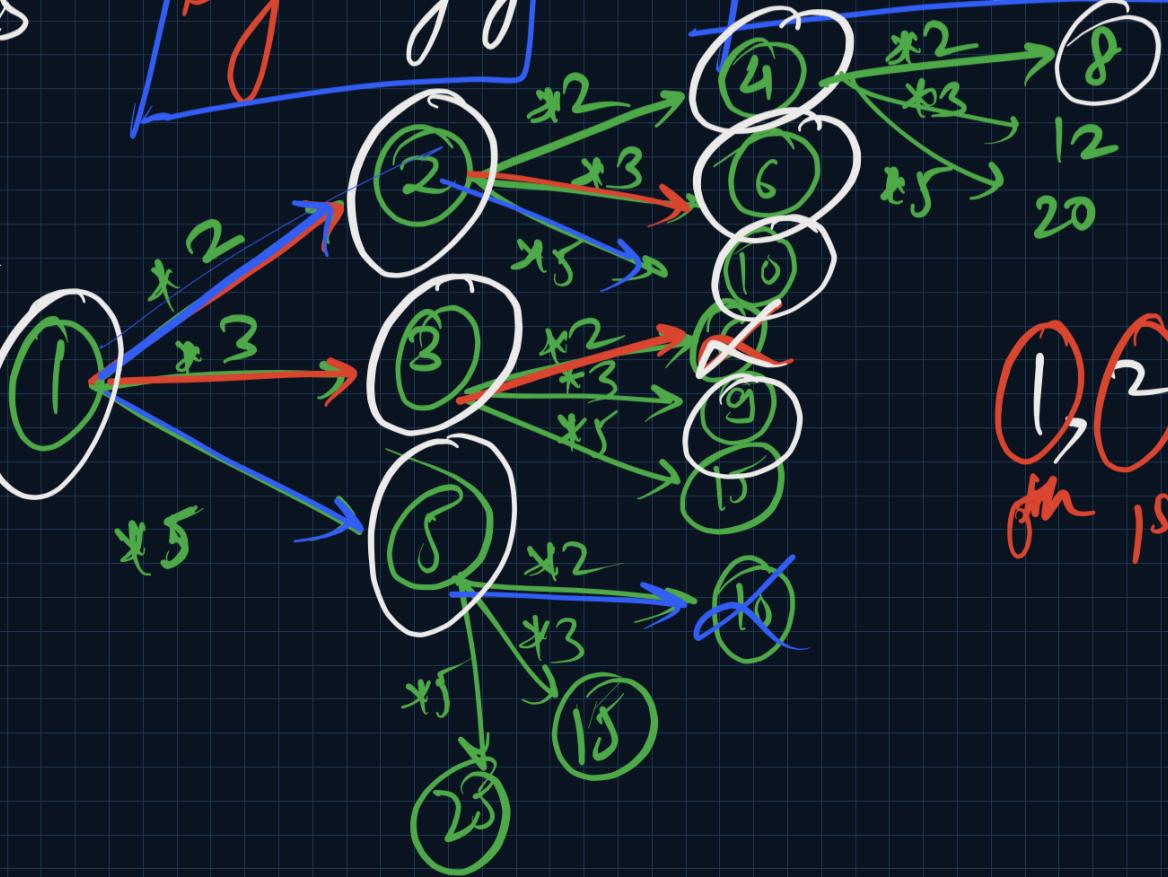
① Brute force

② 1, 2, 3, 4, 5, 6, 8, 10, 12, 15
 2×2 , 2×3 , $2 \times 2 \times 2$, 2×5 , $2 \times 2 \times 3$, 3×5

It is neither
BFS nor DFS

There can be
repetitions
 $2^2 \times 3 = 3 \times 2^2$

Big ugly = smaller ugly * ($2/3/5$)



①, ②, ③, ④, ⑤, ⑥, ⑦, ⑧, ⑨
on 1st 1st 2nd 1st 1st 2nd 3rd 2nd