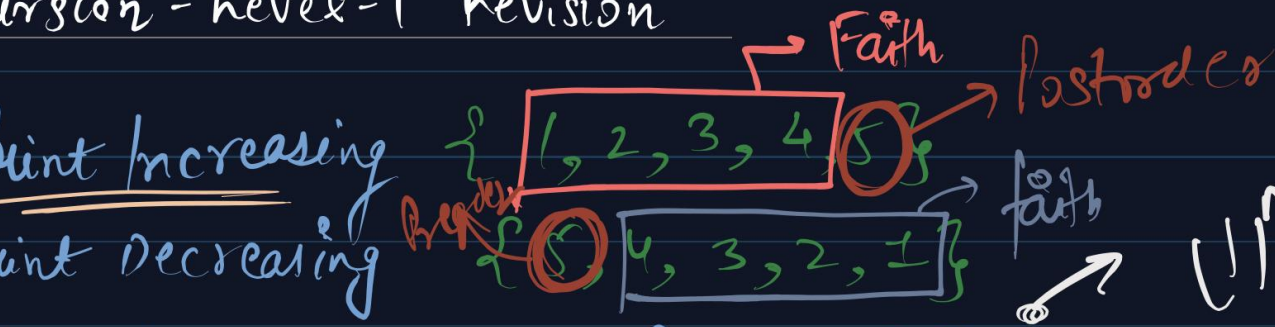
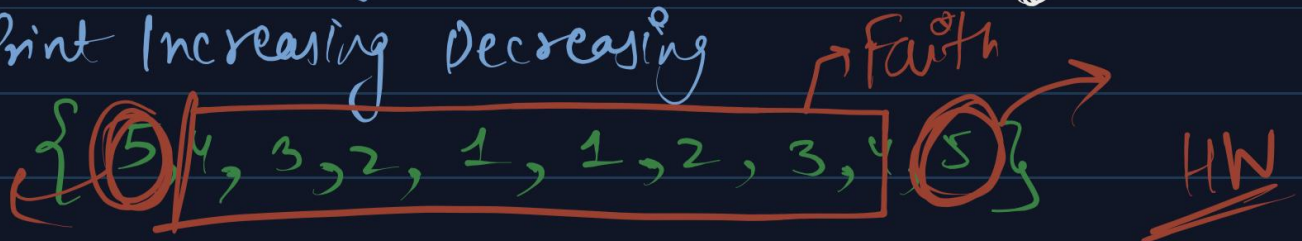
Recursion - level-1 Revision

- Print Increasing
- Print Decreasing



$$1^N + \{k + k\}_{+N} = O(N)$$

- Print Increasing Decreasing



HW

① Expectation :→ pI(N) :→ 1, 2, ... N

Faith :→ pI(N-1) :→ 1, 2, 3, .. N-1

Meeting Expectation :→ Syso(N)

Preorder	Postorder
Syso(N)	pI(N-1)

Preorder	Postorder
Syso(N)	pI(N-1)
pI(N-1)	Syso(N)

```java
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

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

② Expectation :→ pD(N) :→ N, N-1, N-2, ---2, 1 (5) 4-3-2-1

Faith :→ pD(N-1) :→ N-1, N-2, ---1

Meeting Expectation :→ Preorder : Syso(N)

```java
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);
}
```

calls = 1, height = N

$(1)^N + \{k+0\} * N$

$= O(N)$

Power Function

Expectation :- x^n ← pow(x,n)

$x = 2.0, \; n = 5 \;\; : \; 2^5 = 32$

Faith → $x^{n-1} * x$

pow(x,n-1) → x^{n-1}

→ x → Meeting Expectation

Generic Time Complexity of Recursion

$$(Calls)^{height} + \{Preorder + Postorder\} * height$$

Generic Time Complexity of Recursion

$$(Calls)^{height} + \{preorder + postorder\} * height$$

```
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

Calls = 1

Height = N

$(1)^N + \{k + k\} * N$

$= O(N)$

Space

- Input → O(12 bytes) = O(1)
- Output → O(8 bytes) = O(1)
- Auxiliary space (Extra Data Structures Algorithm) → O(1)
- Recursion call space → O(N)
  - ↳ O(height)

Power -> Optimized

even $\begin{cases} x^n = x^{n/2} * x^{n/2} \\ 2^6 = 2^3 * 2^3 = 2^{3+3} = 2^6 \end{cases}$

odd $\begin{cases} x^n = x^{n/2} * x^{n/2} * x \\ 2^7 = 2^3 * 2^3 * 2 = 2^6 * 2 = 2^7 \end{cases}$

```
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 $\Rightarrow 2$ → breadth

height $\Rightarrow \log n$

↳ depth

$\Rightarrow O(2^{\log_2 n}) = O(n)$

Input → O(1)
output → O(1)
Extra → O(1)
Rec. call stack → $O(\log_2 n)$

$n \to n/2 \to n/4 \to n/8 \to n/16 \cdots \to 1$ (a, ar, ar^2, ar^{h-1})

h terms

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

$a, ar, ar^2, \ldots, ar^{n-1}$

$n \to n/2 \to n/4 \to n/8 \to n/16 \cdots \to 1$

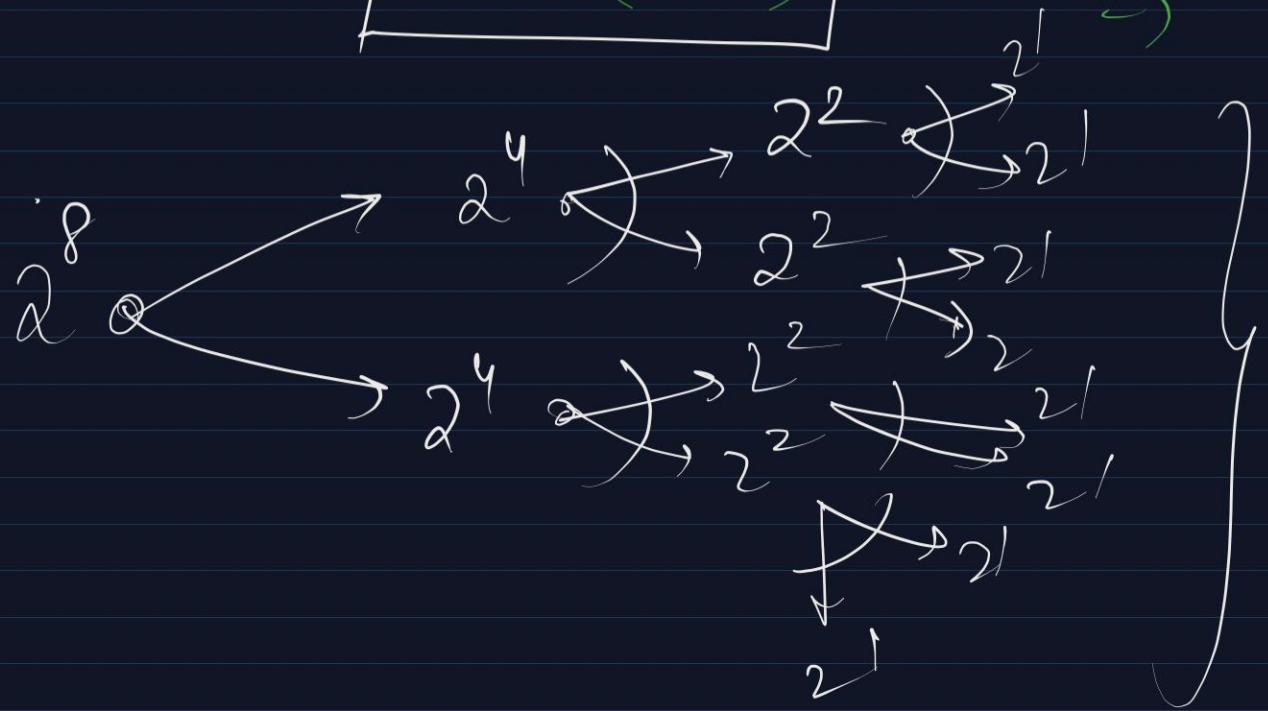
h terms

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

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

$$\Rightarrow h - 1 = \log_2 n$$

$$\boxed{h = O(\log n)}$$



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

        double res = power(x, n/2);

        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);
    }
}
```

call = 1

Same S.C as prev

$x^n \to x^{n/2} \to x^{n/4} \to x^{n/8}$

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

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

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

$2^{31}-1$

$O(\log_2 n) << O(n)$

$N = 2^{16}$

$\log_2 n = 16$

Bit Manipulation $\to$ Modular Exponentiation

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

$\to$ 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

Pre-requisites

Topics

- 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

19 Apr

Lecture (1) Dynamic Programming {10:30 - 12:00}

→ Fibonacci + Climb Stairs Module

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

Exponential

Backtracking

N < 18
20
30

TC → α
SC → α

① Recursion { Brute force }

Recursive

TC → ✓
SC → α

② Memoization { Top Down DP }

Iterative

TC → ✓
SC → ✓

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 :→ Nth Fibonacci No. {fib(N)}

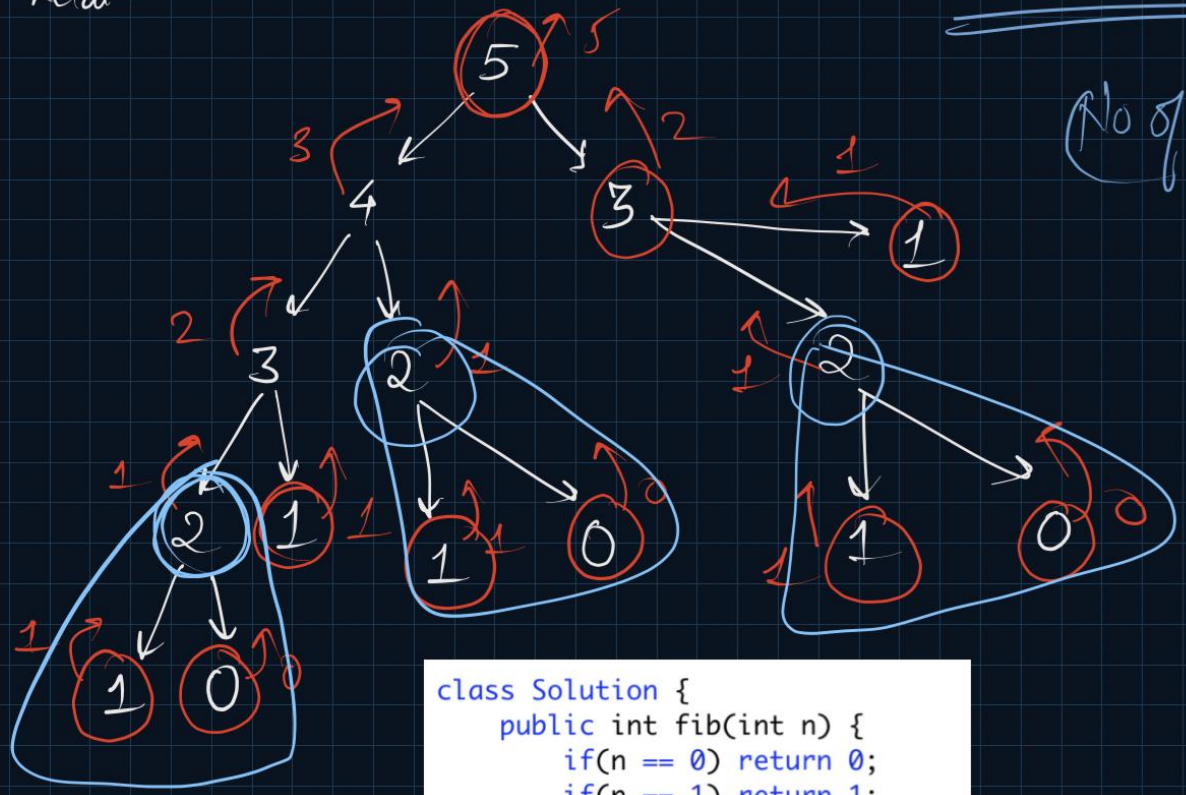
Faith :→ fib(N-1) → fib(N-2)

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

Recurrence Relation

Recurrence Relation

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



Worst case

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

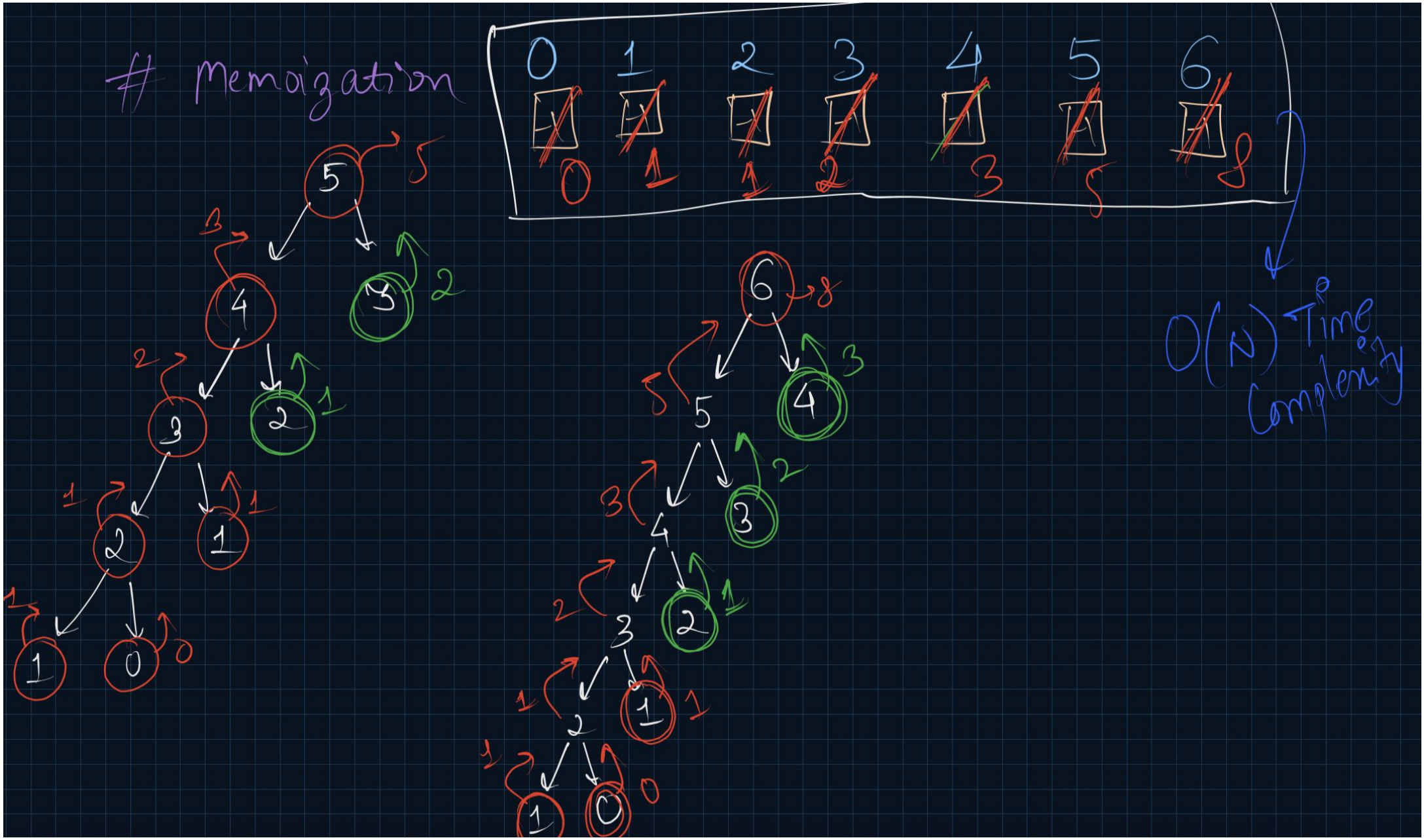
$$(2)^N + (k + k) * N$$

$$\Rightarrow O(2^N) \text{ Time Complexity}$$

Space Complexity $\Rightarrow O(N)$

R · C · S · S

```java
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
0 1 2 3 4 5 6
0 1 1 2 3 5 8
O(N) Time Complexity

```java
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
⋮ → O(N)

Extra Space: → O(N) DP

Time Complexity → O(N)

Dynamic Programming Identification

→ ① Overlapping Subproblems { Repeated calls }

→ ② Optimal Substructure { Faith }

② Tabulation

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

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

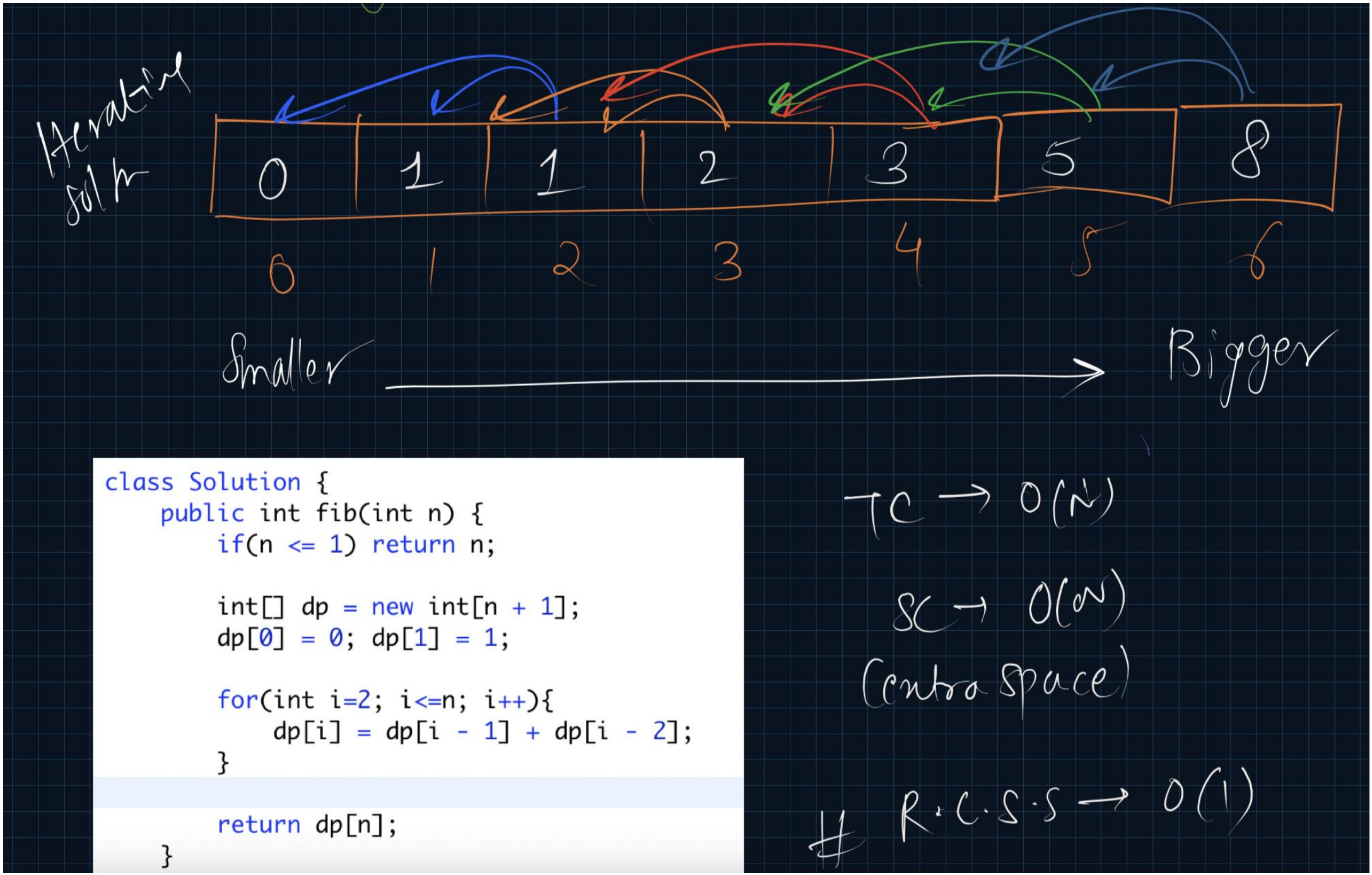
① DP[i] ⟶ Storage & meaning of the cell

↳ ith fibonacci no

② Smaller Problem { DP[0], DP[1] }

↓

Bigger Problem { DP[N] }




```java
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 → O(N)

SC → O(N)
(extra space)

R·C·S·S → O(1)

4

```java
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 → O(N)

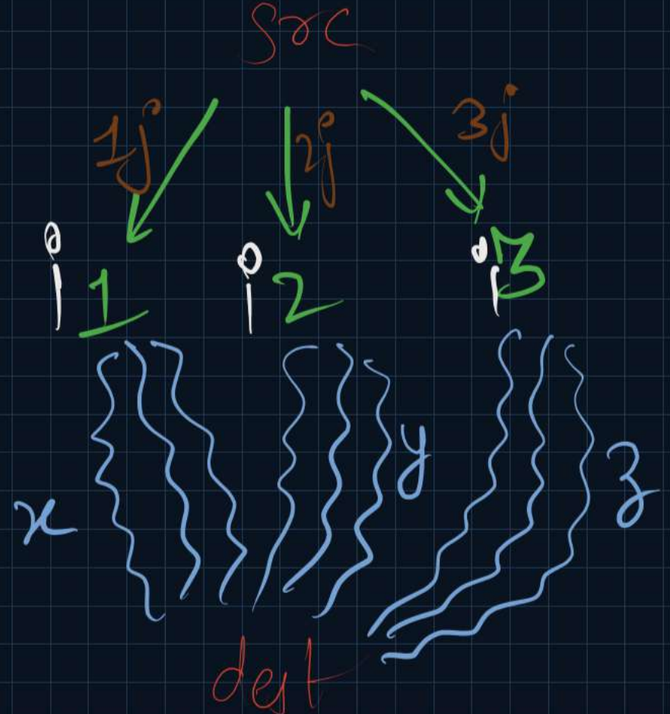
Space Complexity → O(1)

Dynamic Programming - lecture ②

{ 23 April 2022
9 AM - 12 PM }

→ Climb Stairs & its Variations

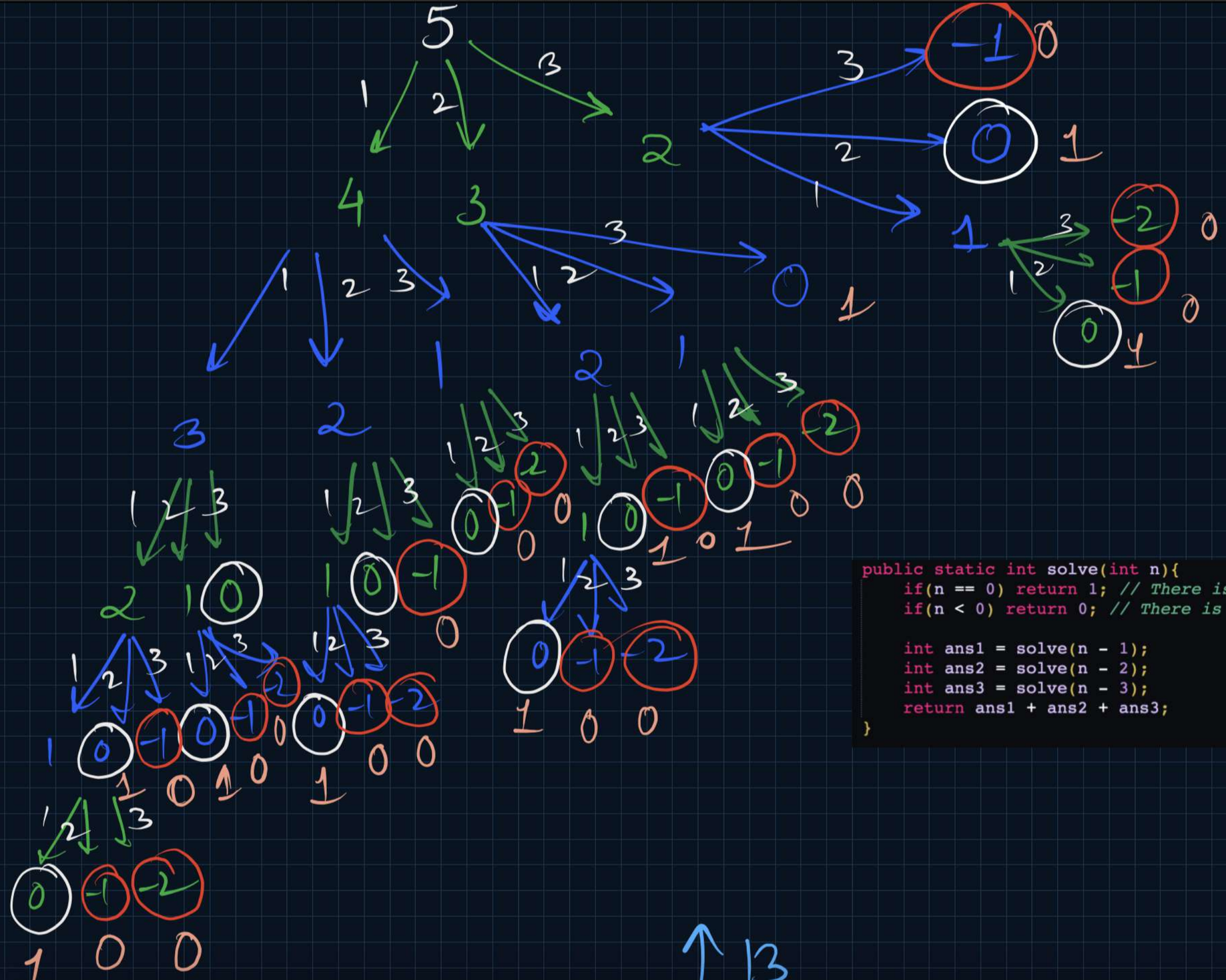
① Climb Stairs



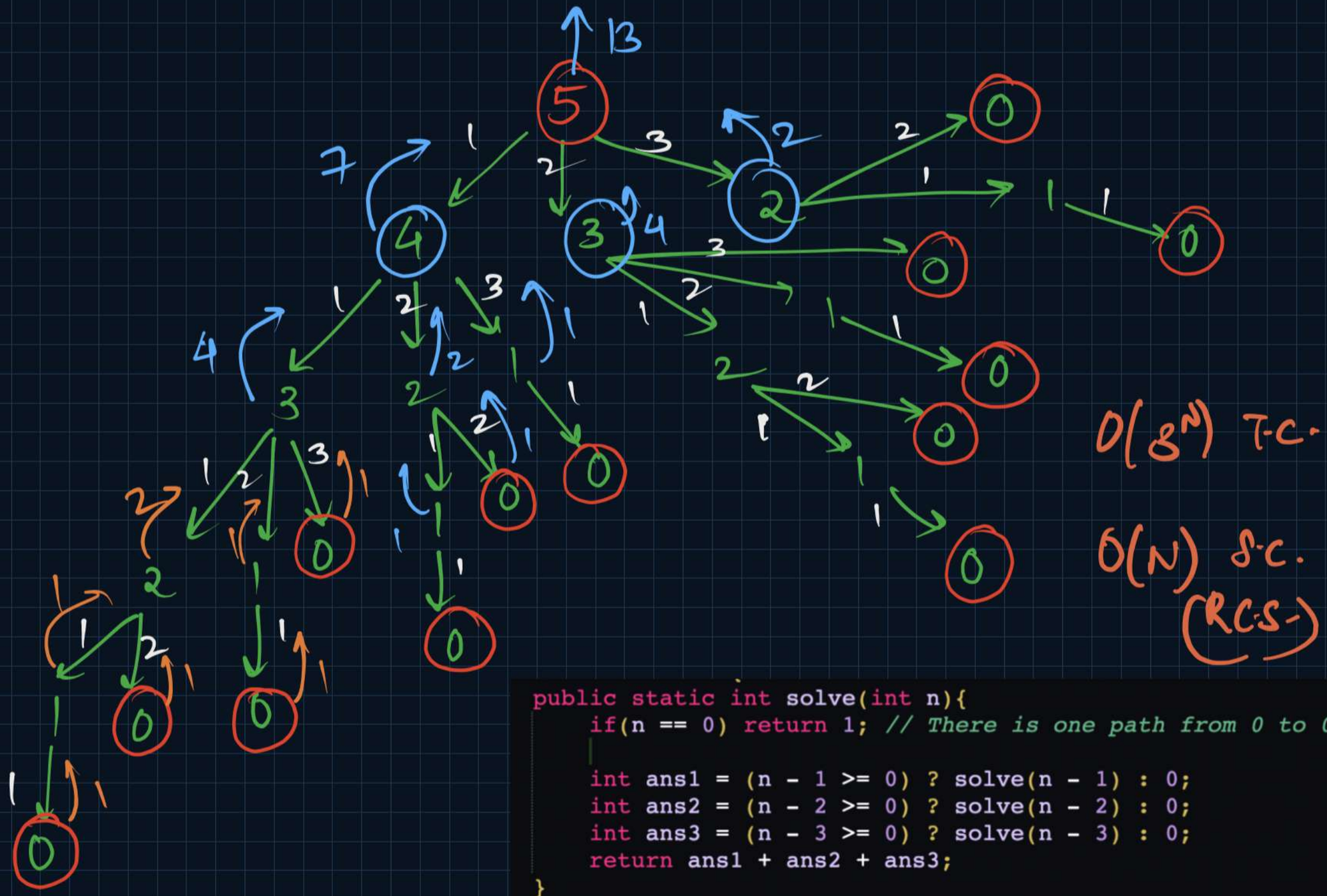
(a) $x+y+z+3$

(b) $x+y+z$

(c) $1+x+y+z$

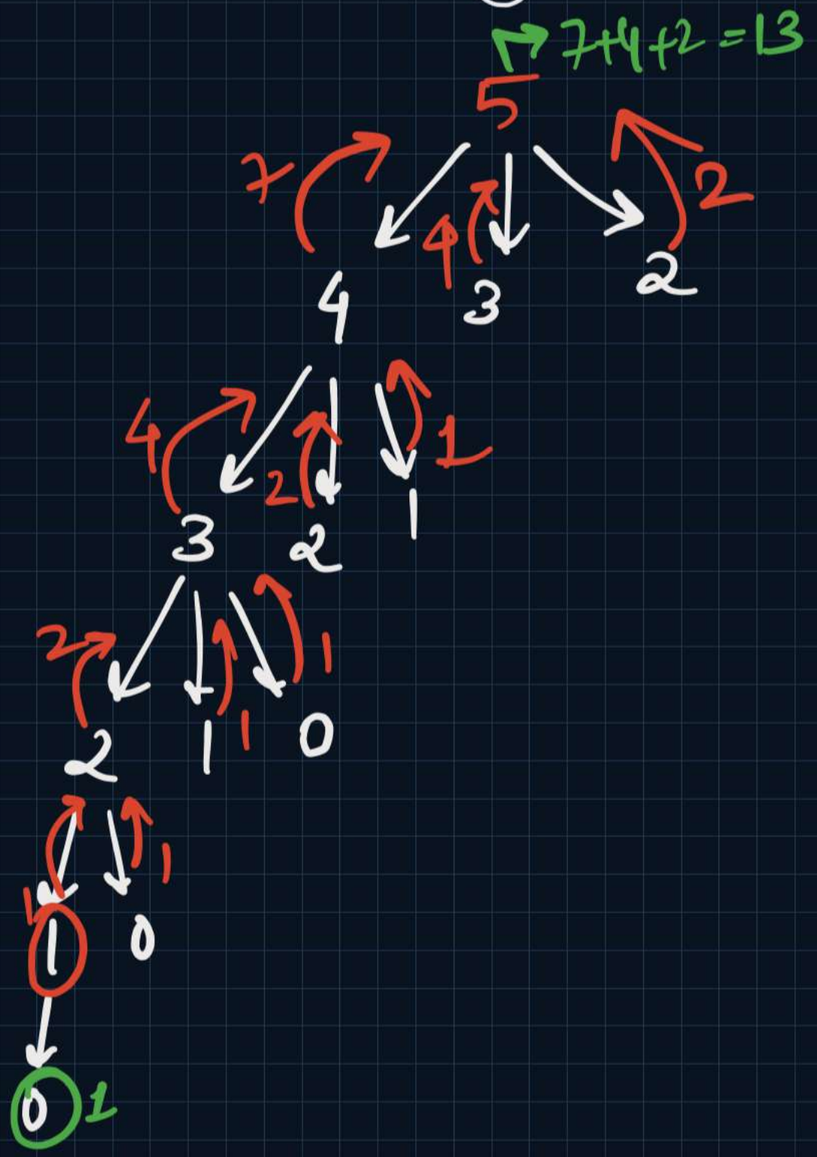


```java
public static int solve(int n){
    if(n == 0) return 1; // There is one path from 0 to 0, i.e. "" empty string
    if(n < 0) return 0; // There is no path from -1 or -2 to 0

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

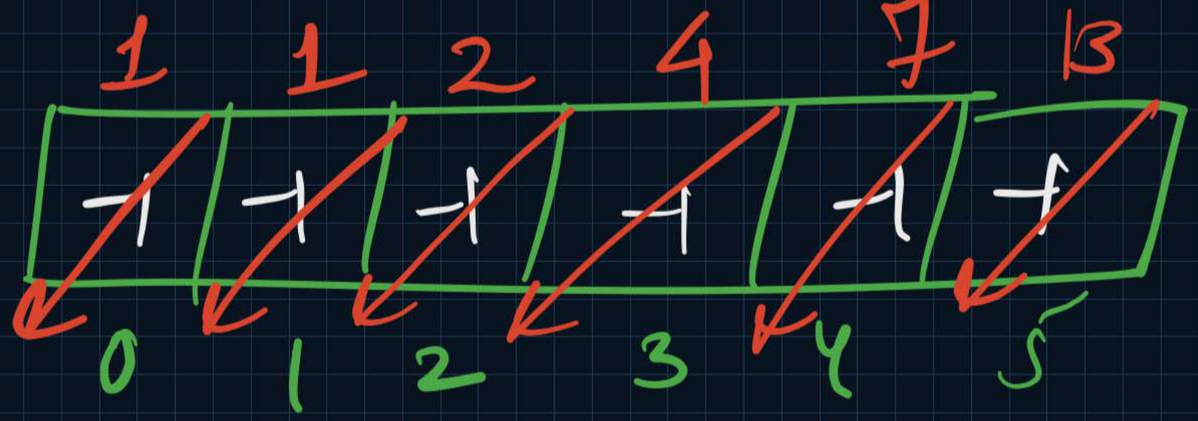



```java
public static int solve(int n){
    if(n == 0) return 1; // There is one path from 0 to 0, i.e. "" empty string

    int ans1 = (n - 1 >= 0) ? solve(n - 1) : 0;
    int ans2 = (n - 2 >= 0) ? solve(n - 2) : 0;
    int ans3 = (n - 3 >= 0) ? solve(n - 3) : 0;
    return ans1 + ans2 + ans3;
}
```

Memorization




```java
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 → O(N)

Space Complexity → O(N) → R.C.S, DP table

Tabulation

TC → O(N)

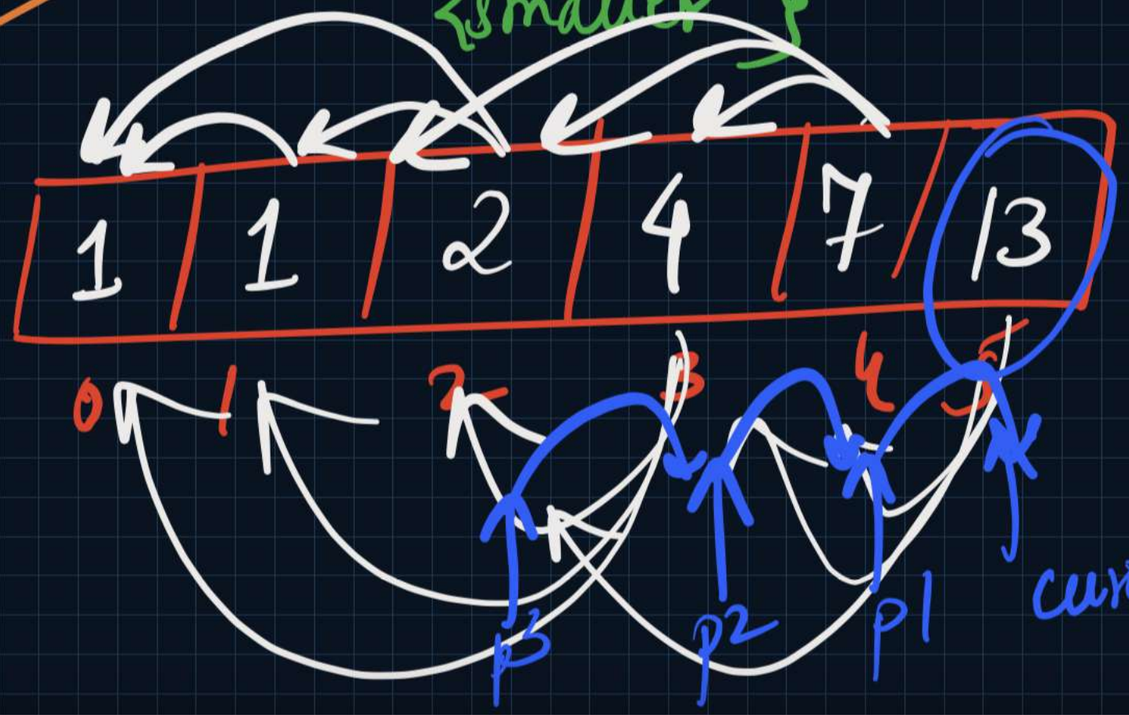
SC → O(N) extra

Tribonacci

0th state

DP[0] = 1

{smaller}



Nth state

DP[N] = ?

{larger}

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

```java
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]);
```

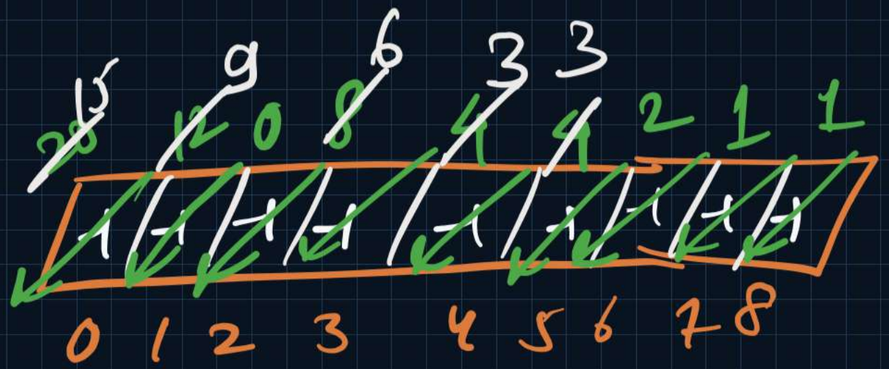
Extra space optimizatn

O(N) time
O(1) space

```java
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

Climb Stairs → Variable moves



Memoization

```java
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

↓

TC ⇒ $O(jumps^N)$

SC ⇒ $O(N)$

R-C-S:

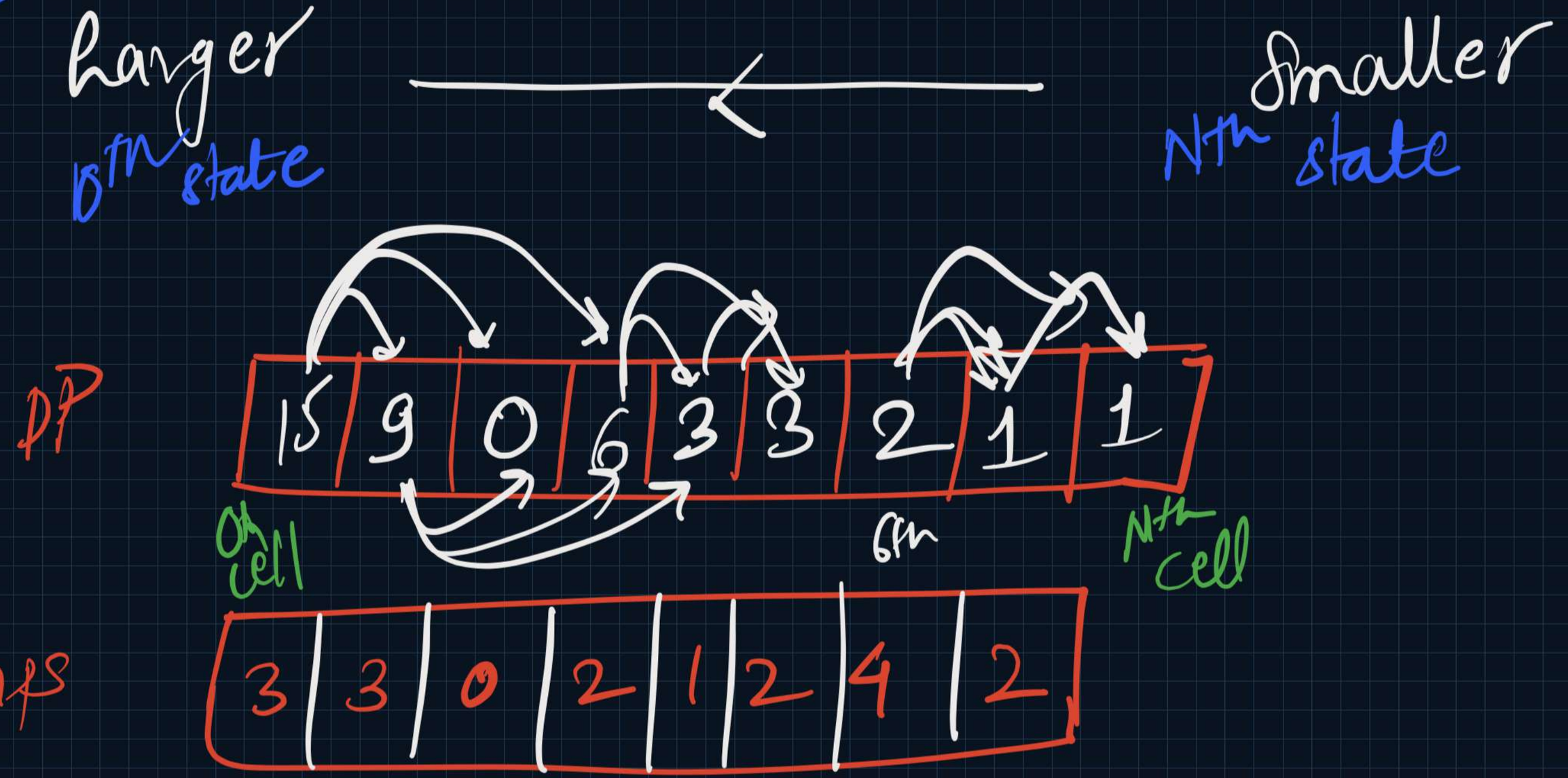
Memo

TC ⇒ $O(N * jumps)$

SC ⇒ $O(N)$

R-C-S, DT

Tabulation


larger
0th state
Nth smaller state
DP
15 9 0 6 3 3 2 1 1
0th cell
6th
Nth cell
jumps
3 3 0 2 1 2 4 2

```java
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*jumps) TC

O(n) SC {DP table}

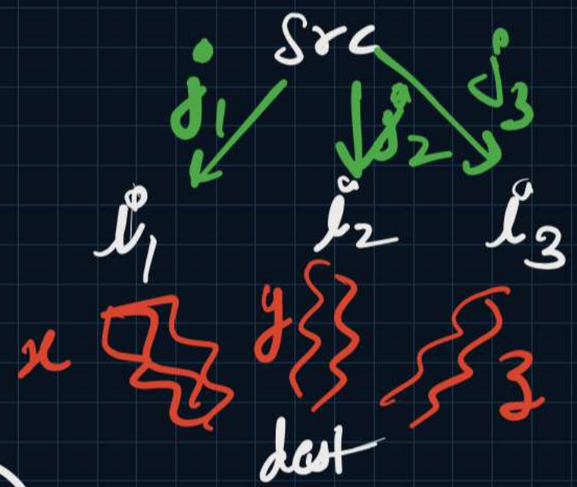
Space optimization

↳ Not possible {jumps value is not fixed}

Climb Stairs with minimum Jumps

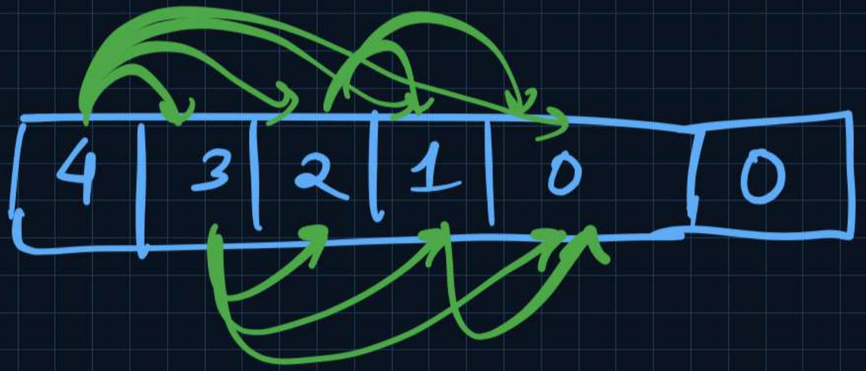
2	3	1	1	4
0th	1st	2nd	3rd	4th
src				dest

jumps



$$ans = \min(x, y, 3) + 1$$

eg2



count of paths = 0

minm length = $+\infty$


```java
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) + 1l);
            }
        }

        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 ⇒ $O(N \times jumps)$

SC ⇒ $O(N)$

* Tabulation ⇒ HW → TC ⇒ O(n * jumps) ≈ O(N²)
                  → SC ⇒ O(n)

* Space Optimization → Not possible.

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

arr[i]+i

Greedy Soln

reach = 0

eg1

{ 2, 4, 2, 1, 2, 0, 2, 1 }

0 1 2 3 4 5 6 7

src

dest

i > reach

6+2 > n-1

eg2

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

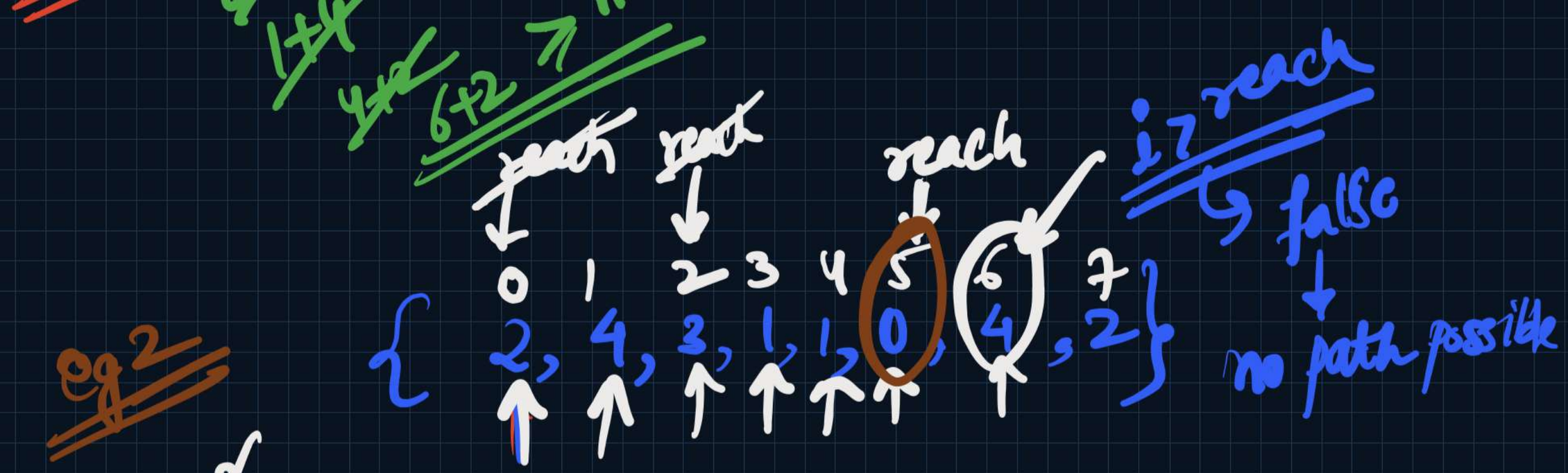
0 1 2 3 4 5 6 7

reach

reach

reach

if reach ↳ false ↓ no path possible



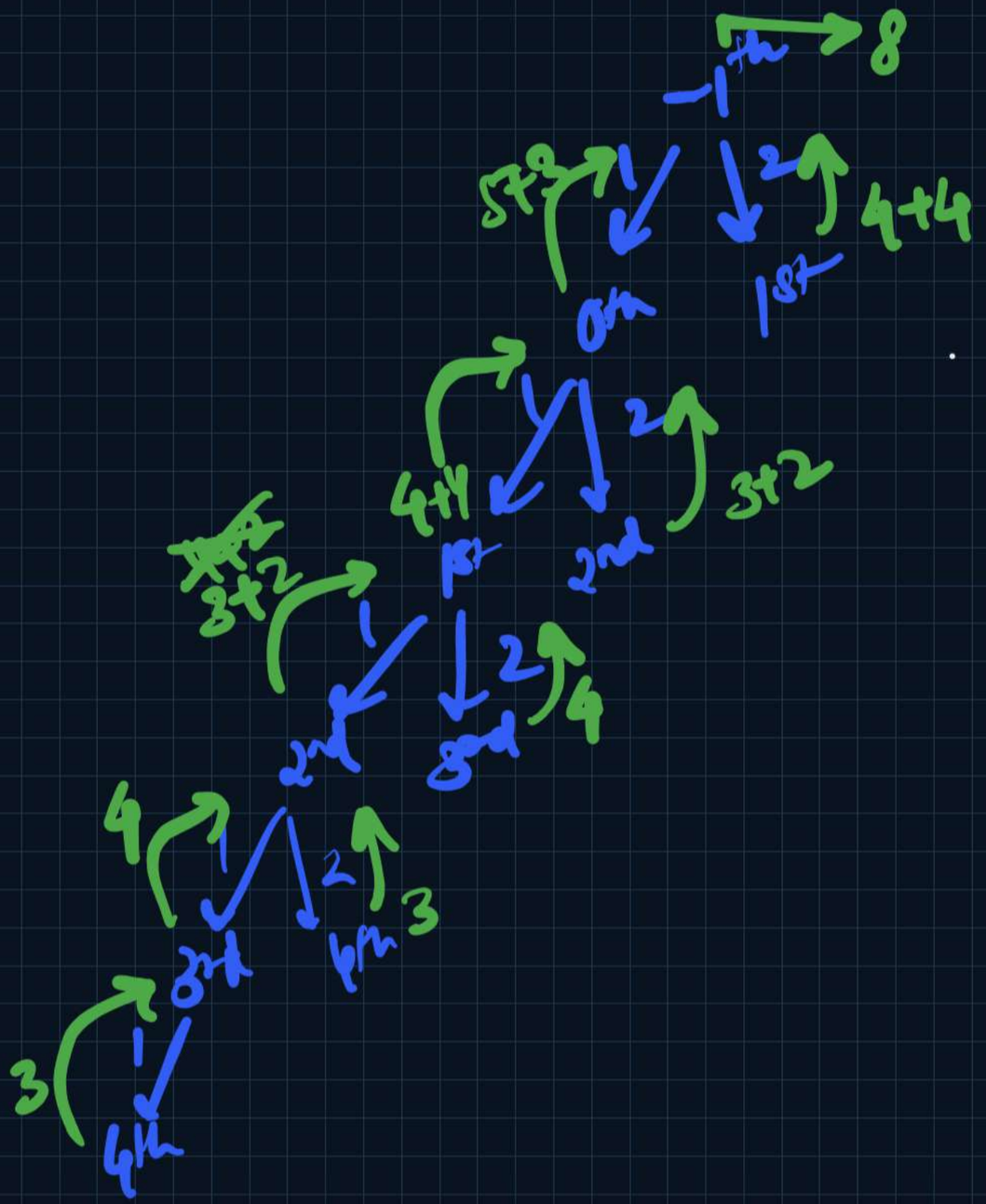
eg 2

reach = 0

```java
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 ⇒ O(N)

SC ⇒ O(1)

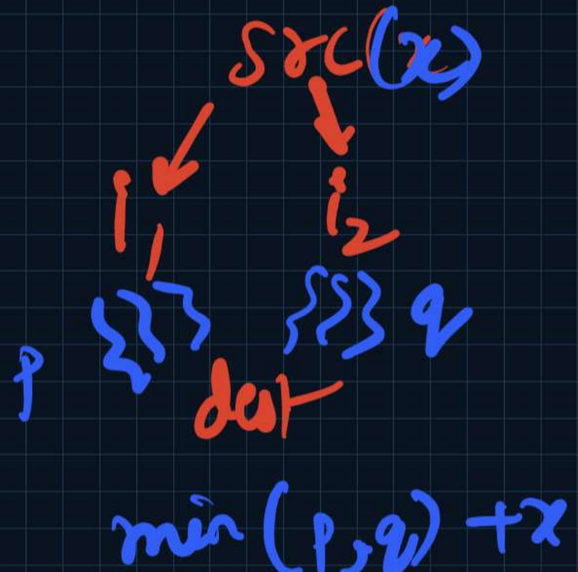


Climb Stairs with Minimum Cost

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

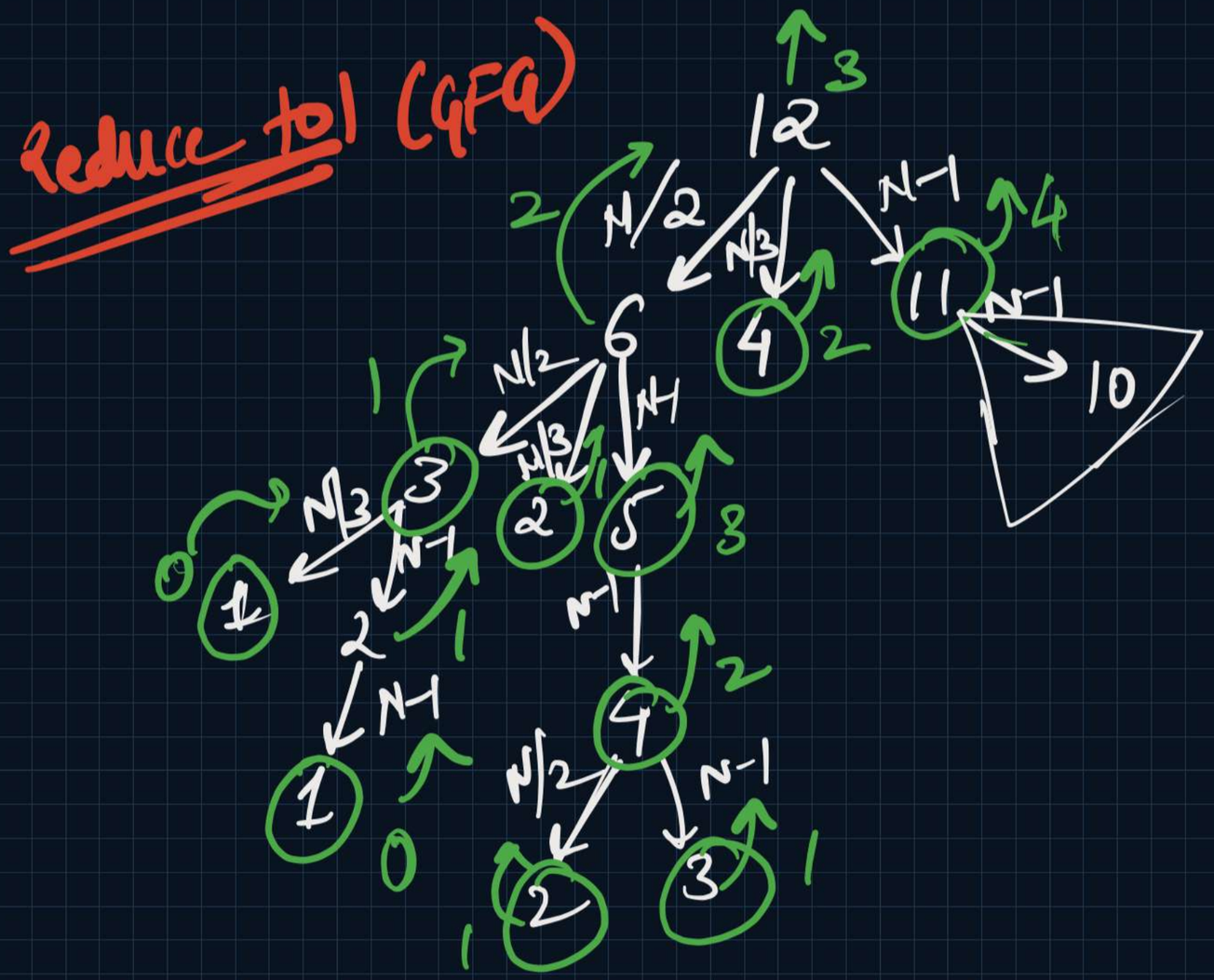
-1 0 1 2 3 4

jumps → 1, 2

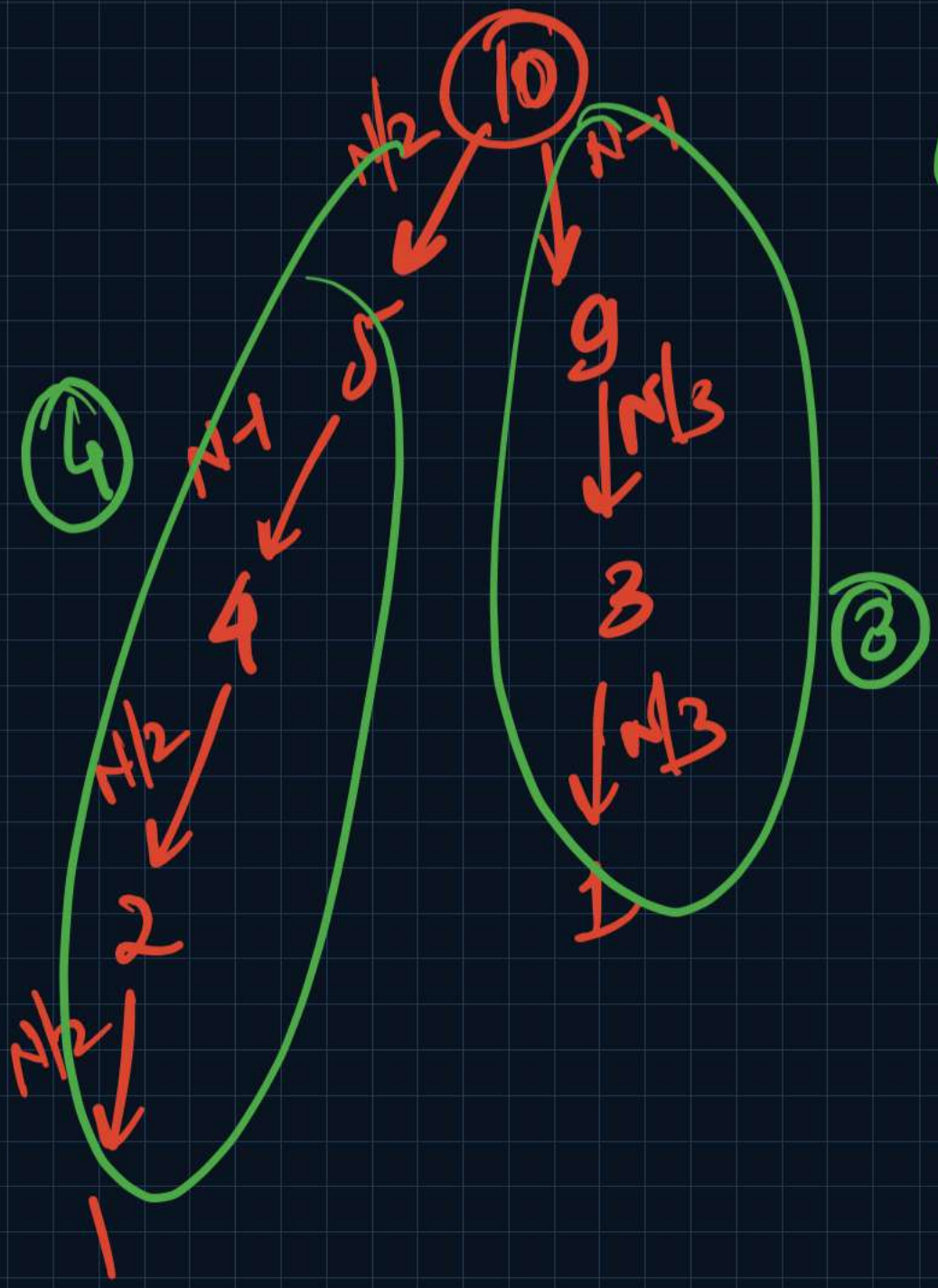



```java
class Solution {
    public int memo(int src, int[] cost, int[] dp){
        if(src >= cost.length) return 0;
        if(dp[src] != -1) return dp[src];

        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]);
    }
}
```

Memoizatn
O(N) time
O(N) space




```java
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

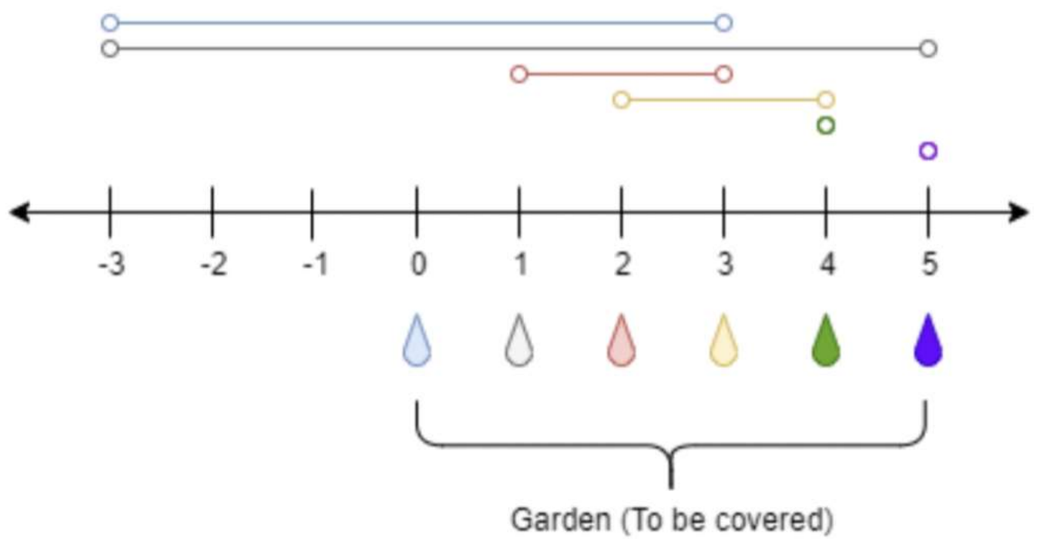
↓

Smaller the no, better the answer

Minimum Taps to Garden

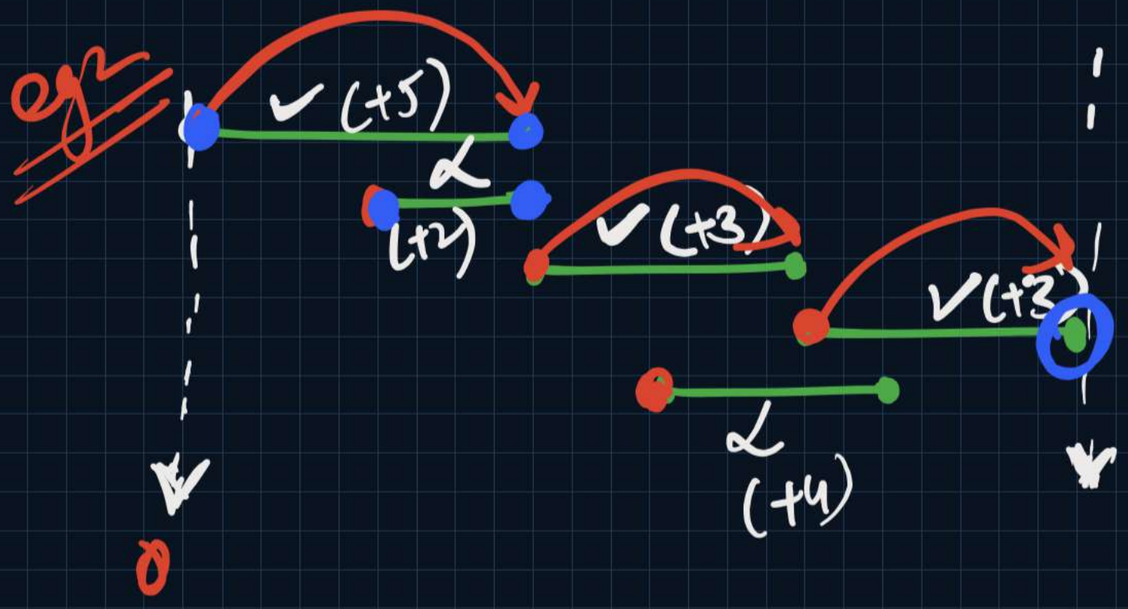
eg1

Example 1:



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

eg2



0 → 5

1 → 2

2 → 3

3 → 4

4 → 3

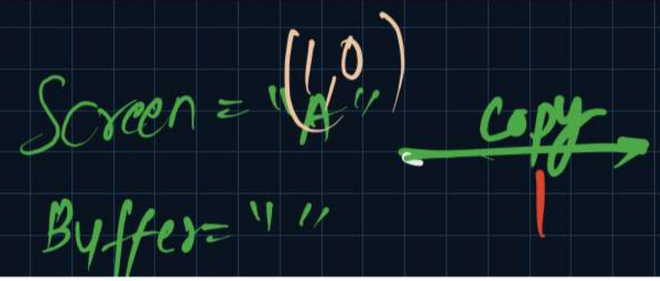
answer = (3)

2 Key keyboard

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:

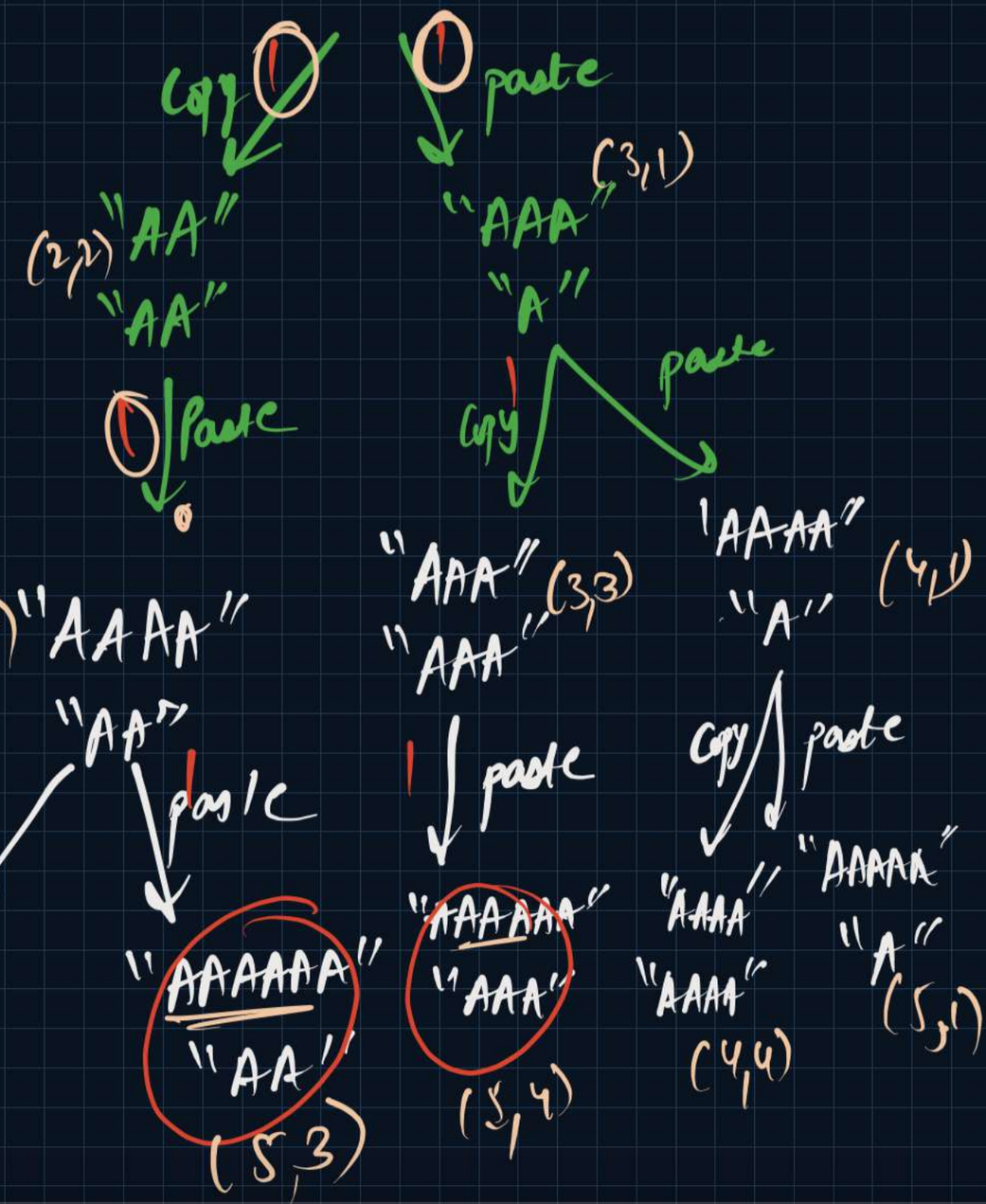
- Copy All: You can copy all the characters present on the screen (a partial copy is not allowed).
- Paste: You can paste the characters which are copied last time.

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

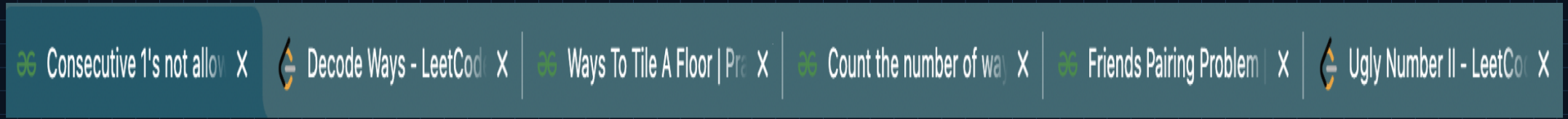



```java
class Solution {
    public long memo(int screen, int buffer, long dest, long[][] dp){
        if(screen > dest) return Integer.MAX_VALUE;
        if(screen == dest) return 0;
        if(dp[screen][buffer] != -1l)
            return dp[screen][buffer];

        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 → $O(N^2)$



Dynamic Programming - lecture 4 { 24 April, Sunday }
9 AM - 12 PM



{ Fiboracci Variations }

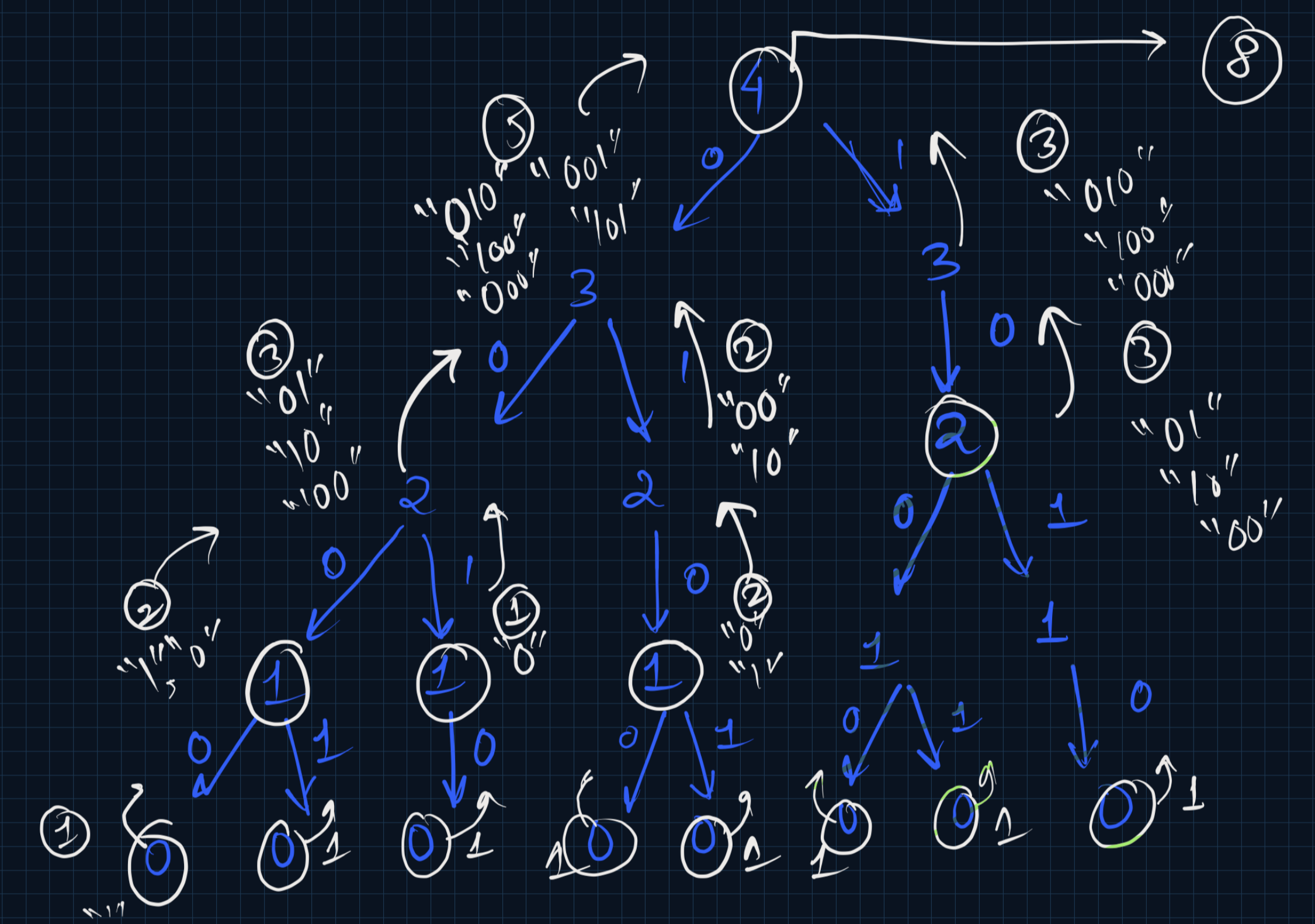
① Count valid binary strings

(2) ① → "0", "1"

(3) ② → "00", "01", "10", ~~"11"~~

(5) ③ → "000", "001", "010", ~~"011"~~,

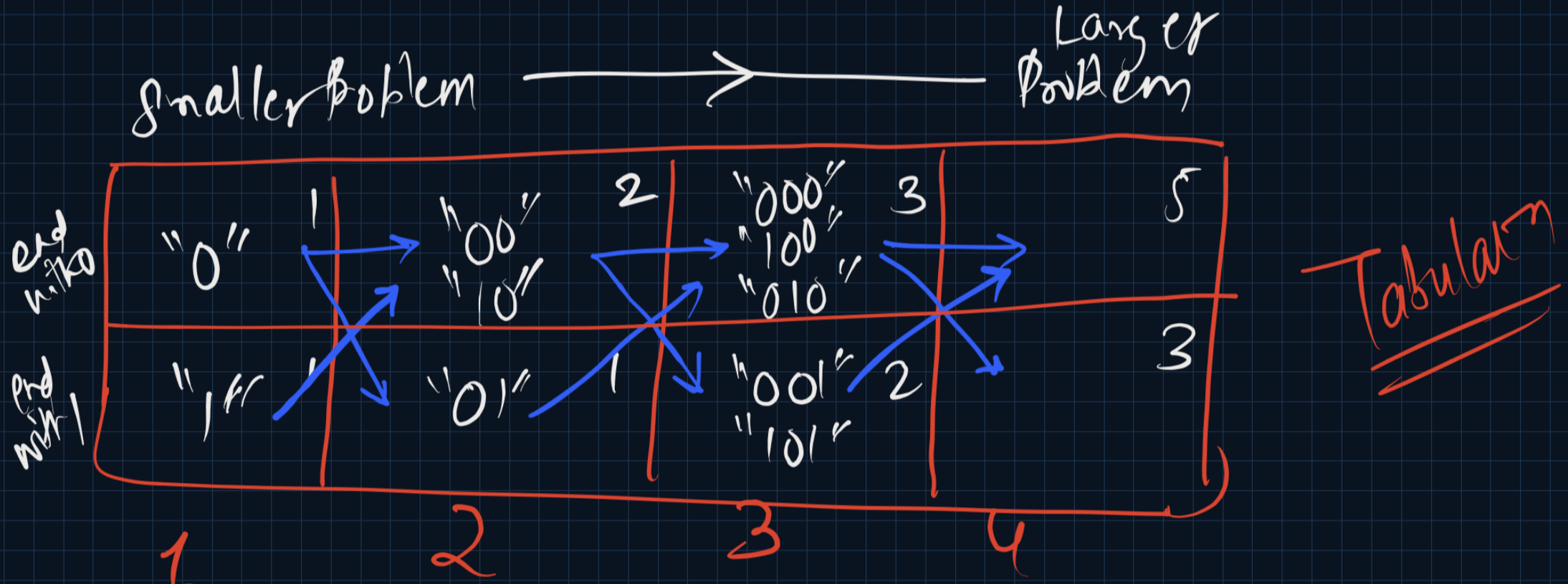
"100", "101", ~~"110"~~, ~~"111"~~

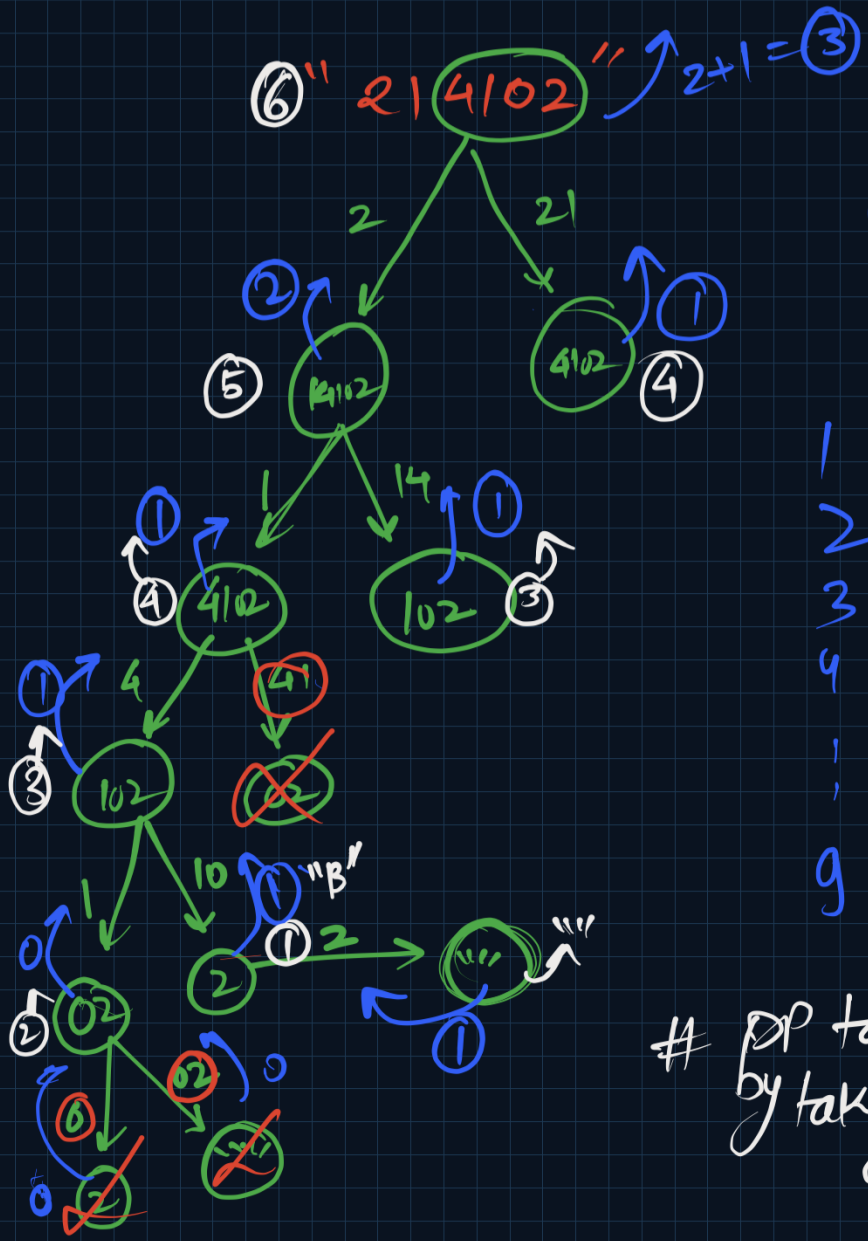


```java
class Solution {
    int mod = 1000000007;

    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);
    }

}
```

TC
O(2*N)
≈ O(N)

SC
RCS → O(N)
ExtraSpace → O(2*N) = O(N)


Smaller Problem
Larger Problem
end with 0
end with 1
"0"
"1"
"00"
"10"
"000"
"100"
"010"
"001"
"101"
1
2
3
5
3
2
1
1
2
3
4
Tabular



② Count Encodings { Decode Ways }

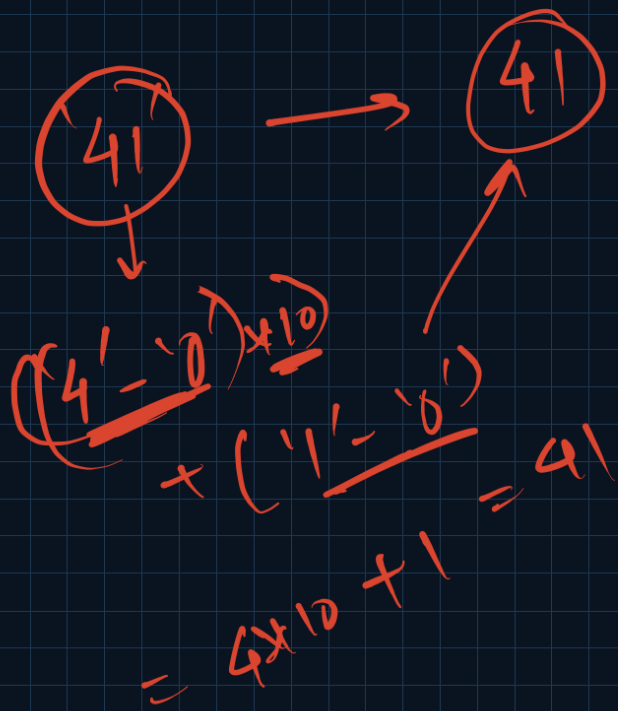
1	10	20
2	11	21
3	12	22
4	1	23
⋮	⋮	24
9	19	25
		26

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

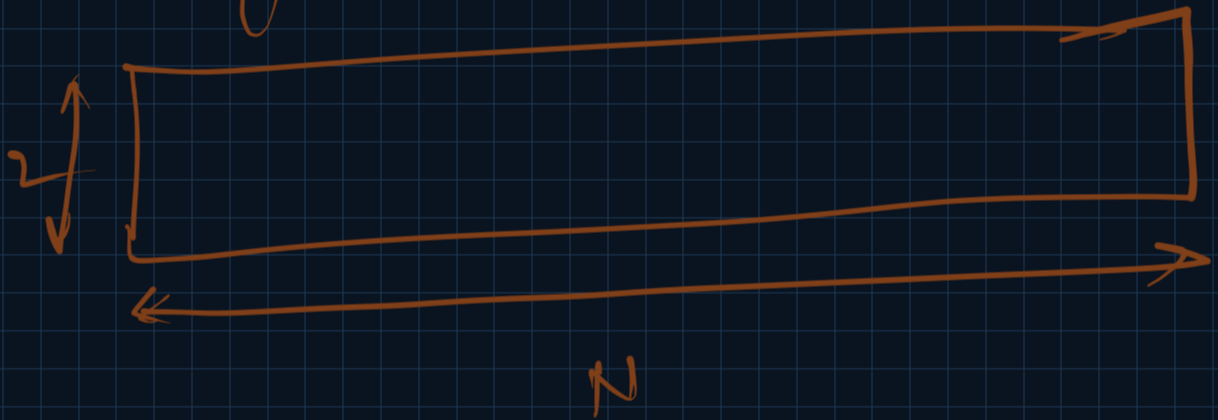
count (ways) → des → dest ①

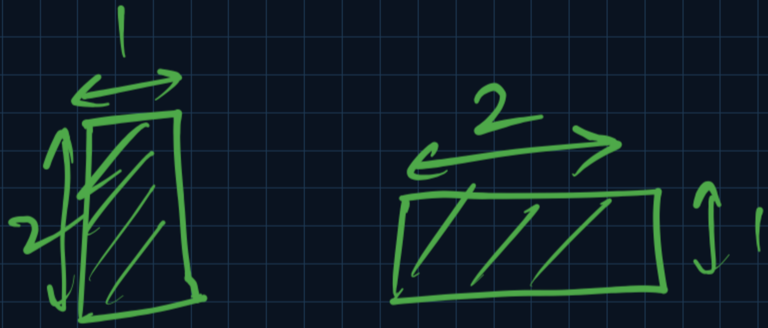
min steps (length) → des → dest ⓪

```java
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;
    }
```

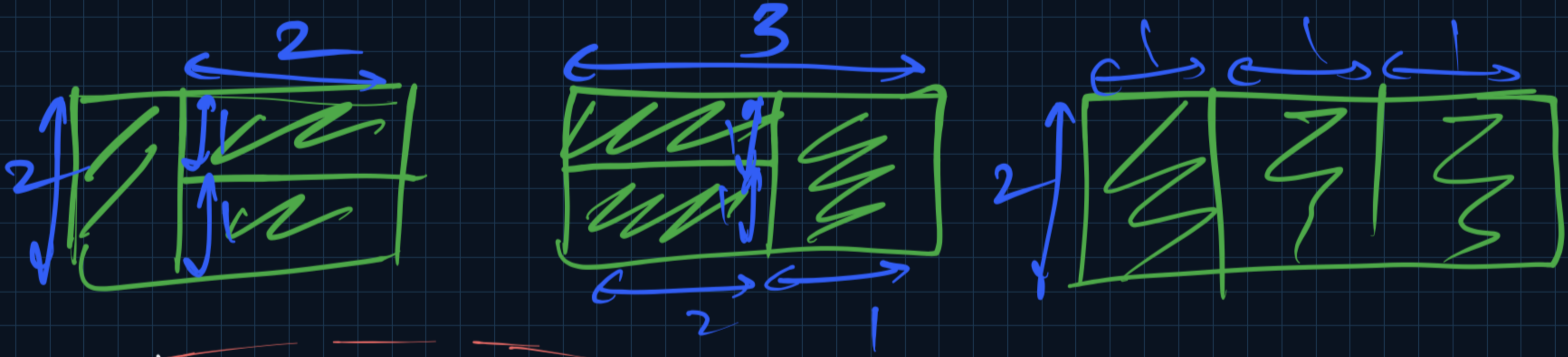


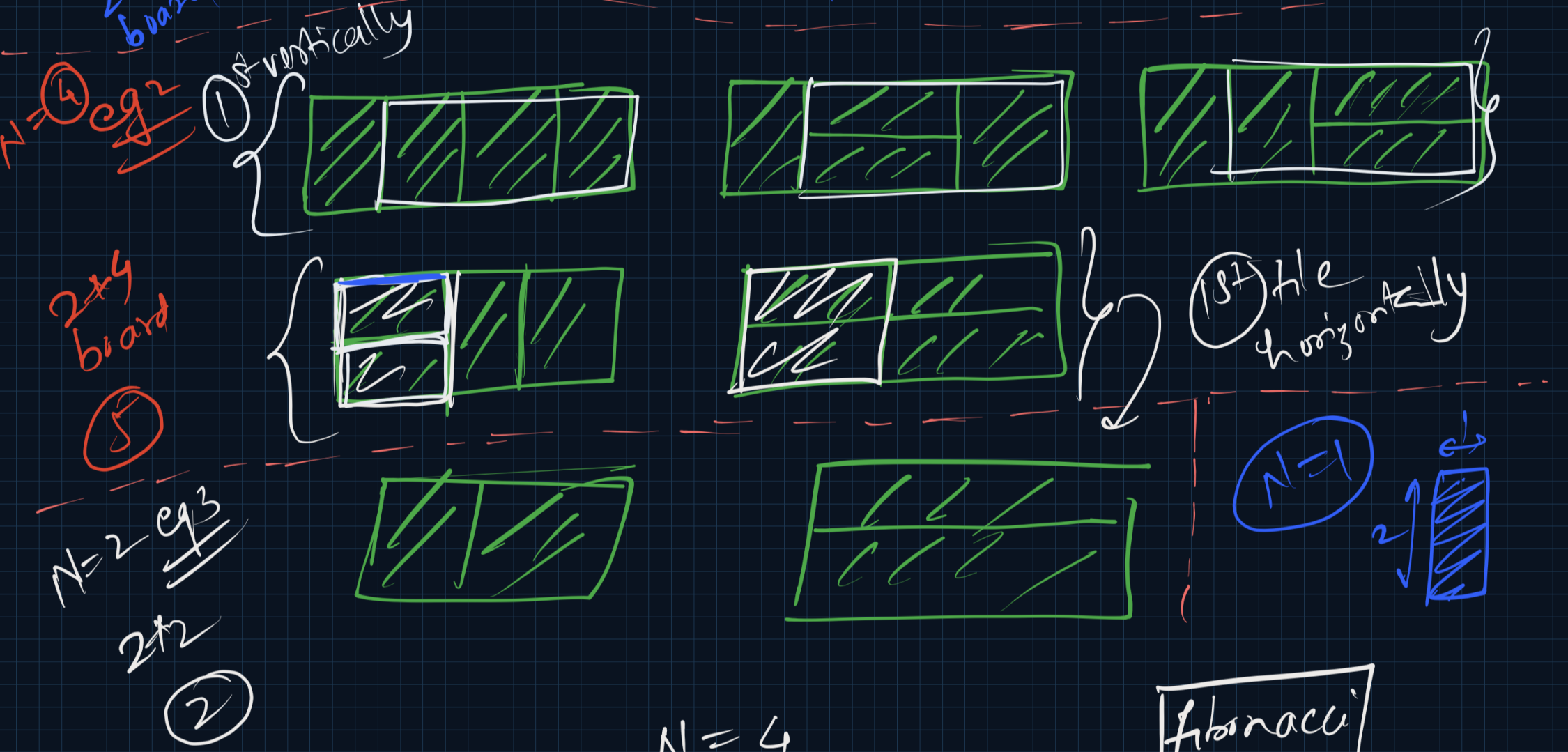
Tiling Problem – (I)

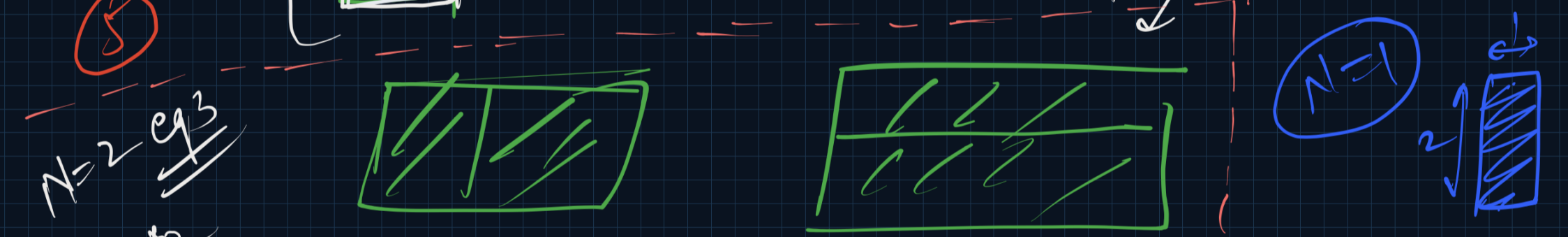




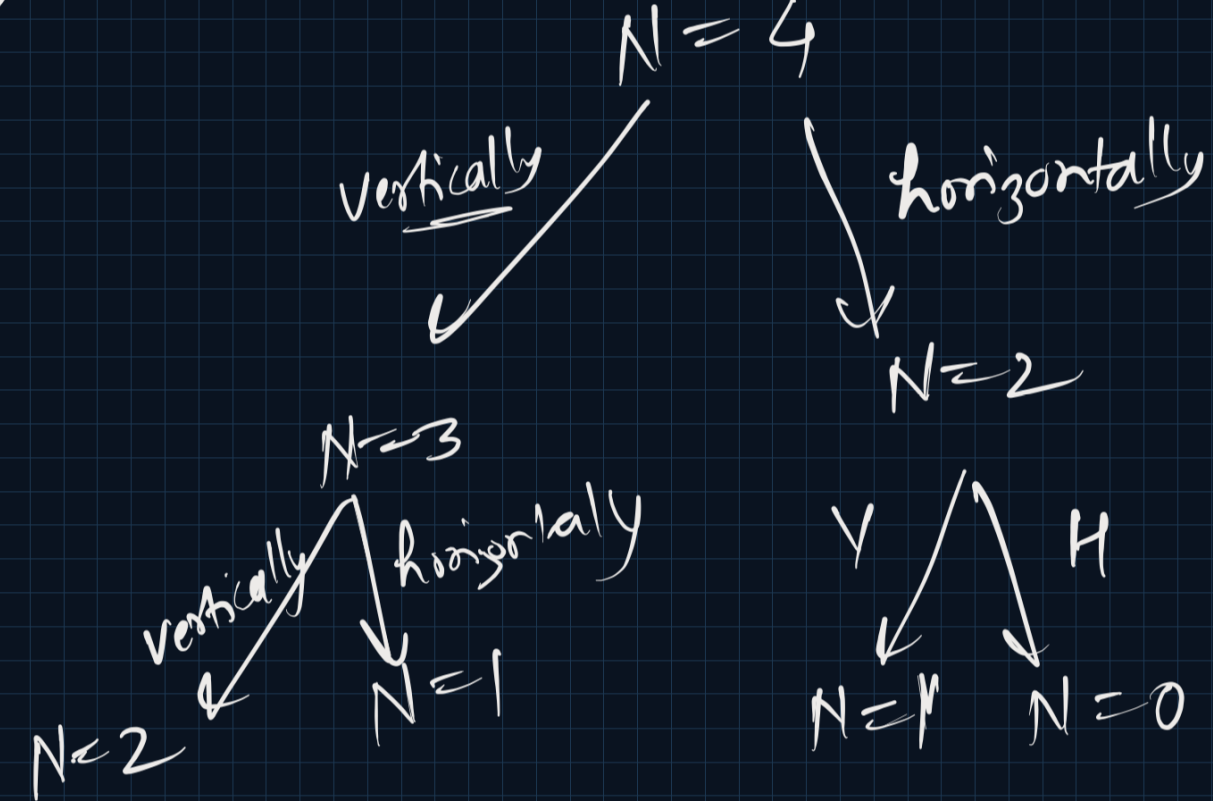
N = (3) eg!
2 x 3 board



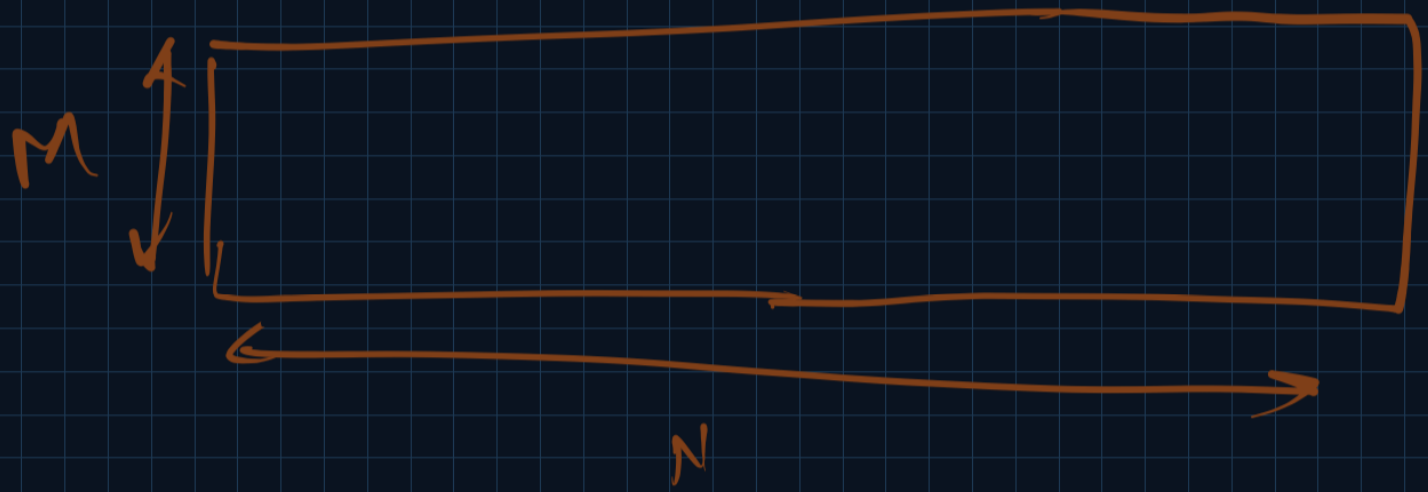

N=4 eg2
4 vertically
2*4 board
5
N=2 eg3
2*2
2
1 vertically
1 horizontally
N=1
N = 4
Fibonacci


N=2 eg3
2+2
2
N=1
2

Fibonacci


N=4
vertically
horizontally
N=2
N=3
vertically
horizontally
N=2
N=1
V
H
N=1
N=0

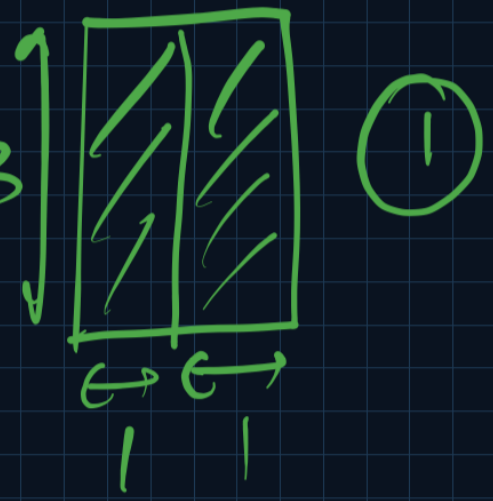
Tiling Problem - II

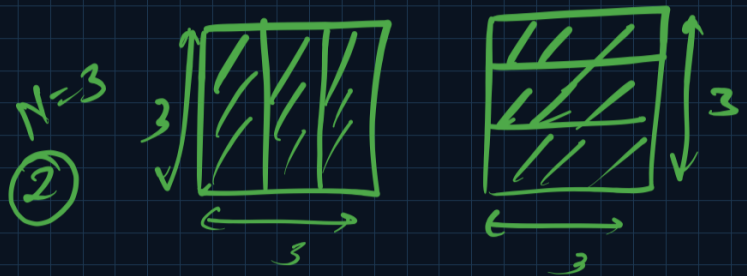


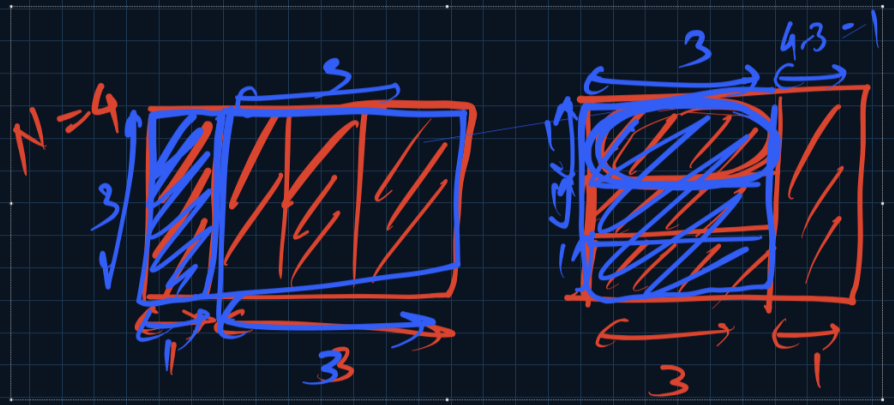
M=3

N=1

N=2

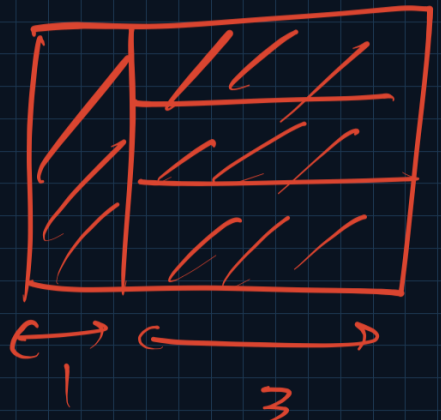




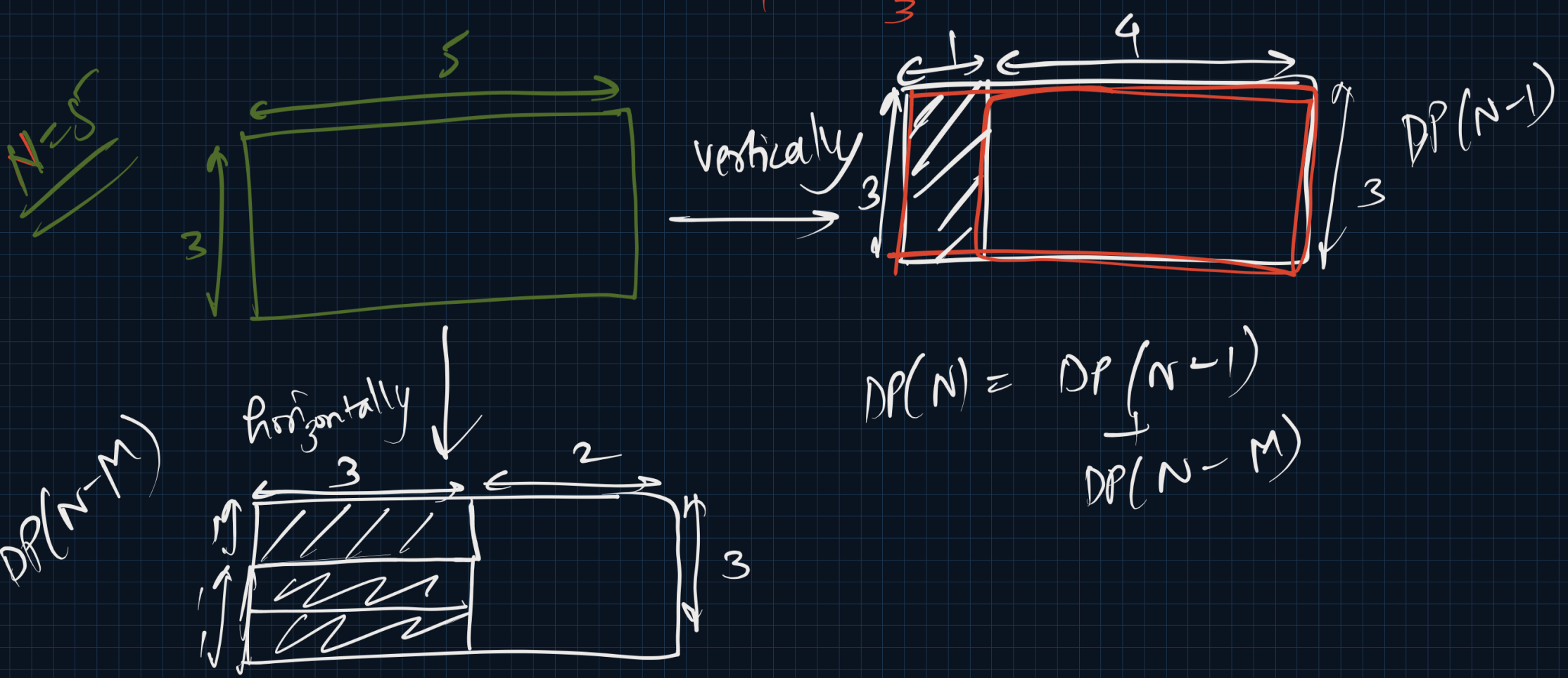


$DP(N) = DP(N-1) + DP(N-M)$

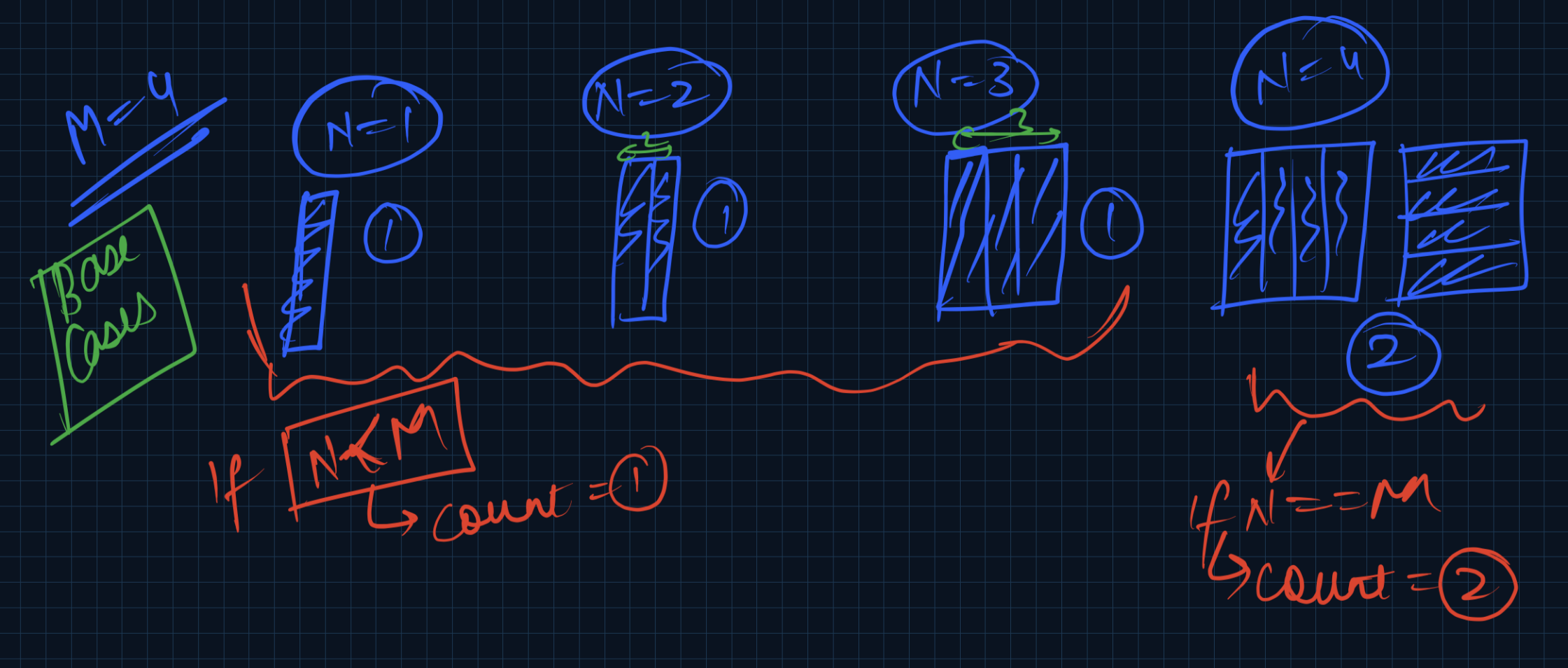
↑ vertically ↑ horizontally



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



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

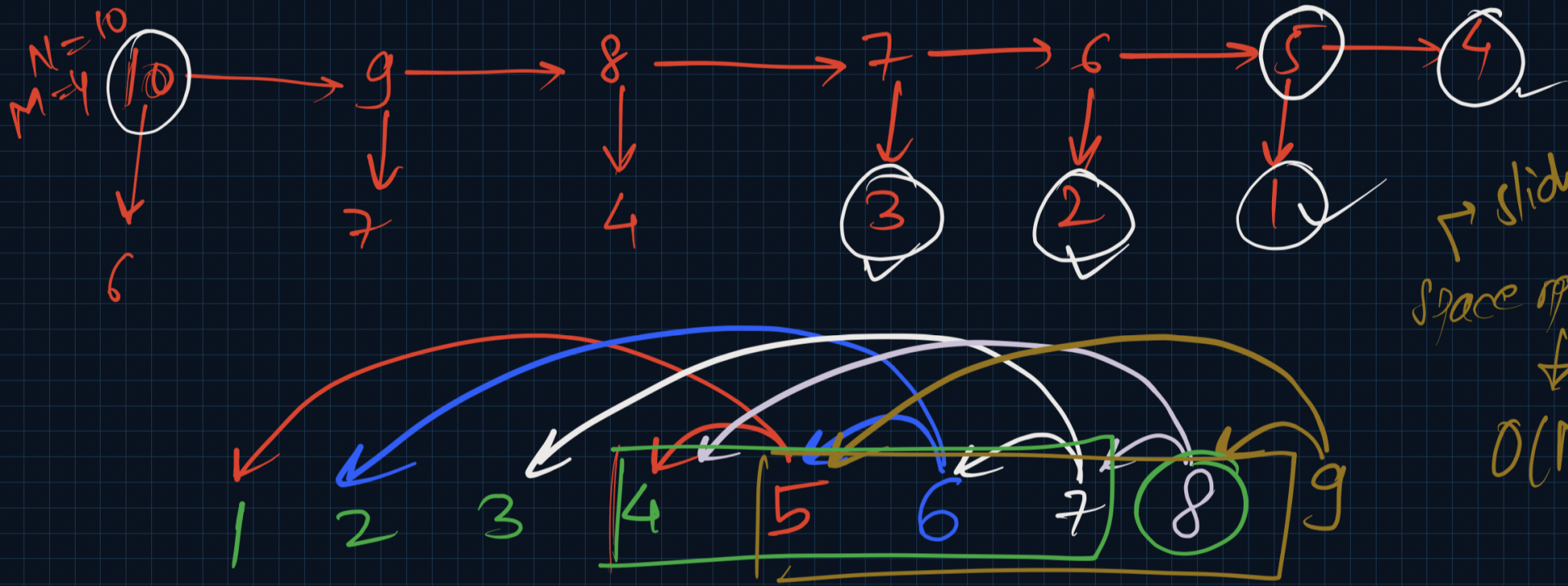

M=4
Base Case
N=1
1
N=2
1
N=3
1
N=4
2
If N<M
count = 1
if N==M
count = 2

```java
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 :→ $O(N)$

Space :→ $O(N)$

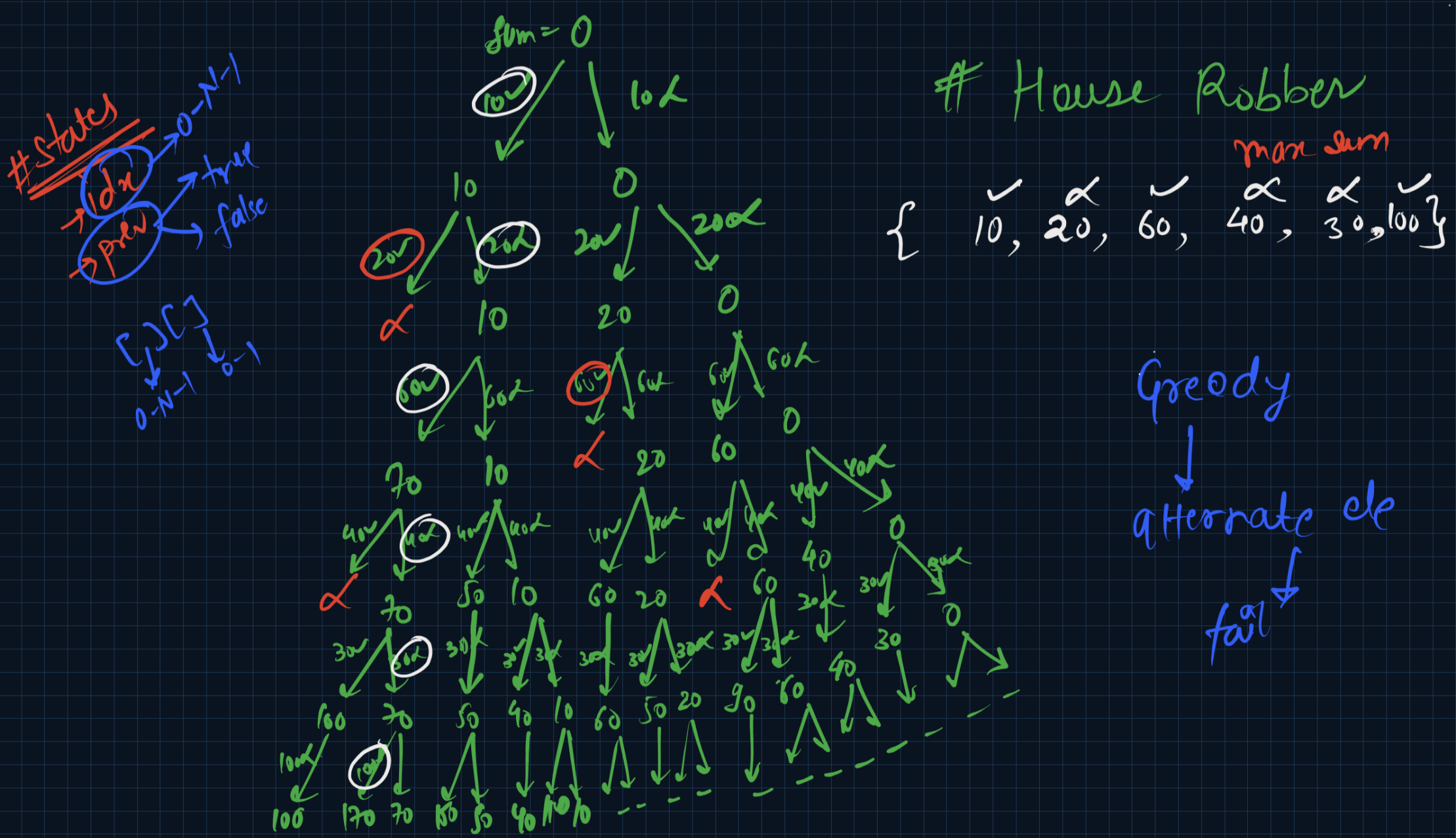
↙ R.C.S ↓ DP table


N=10
M=4
10
9
8
7
6
5
4
6
7
4
3
2
1
sliding window
space optimization
O(M)
1
2
3
4
5
6
7
8
9

```java
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();
}
```

Time → O(N)

Space → O(M)


sum = 0
# House Robber
max sum
{ 10, 20, 60, 40, 30, 100 }
Greedy
alternate de
fail
#States
idx
prev
0-N-1
true
false
[ ][ ]
0-N-1
0-1

```java
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);
    }
```

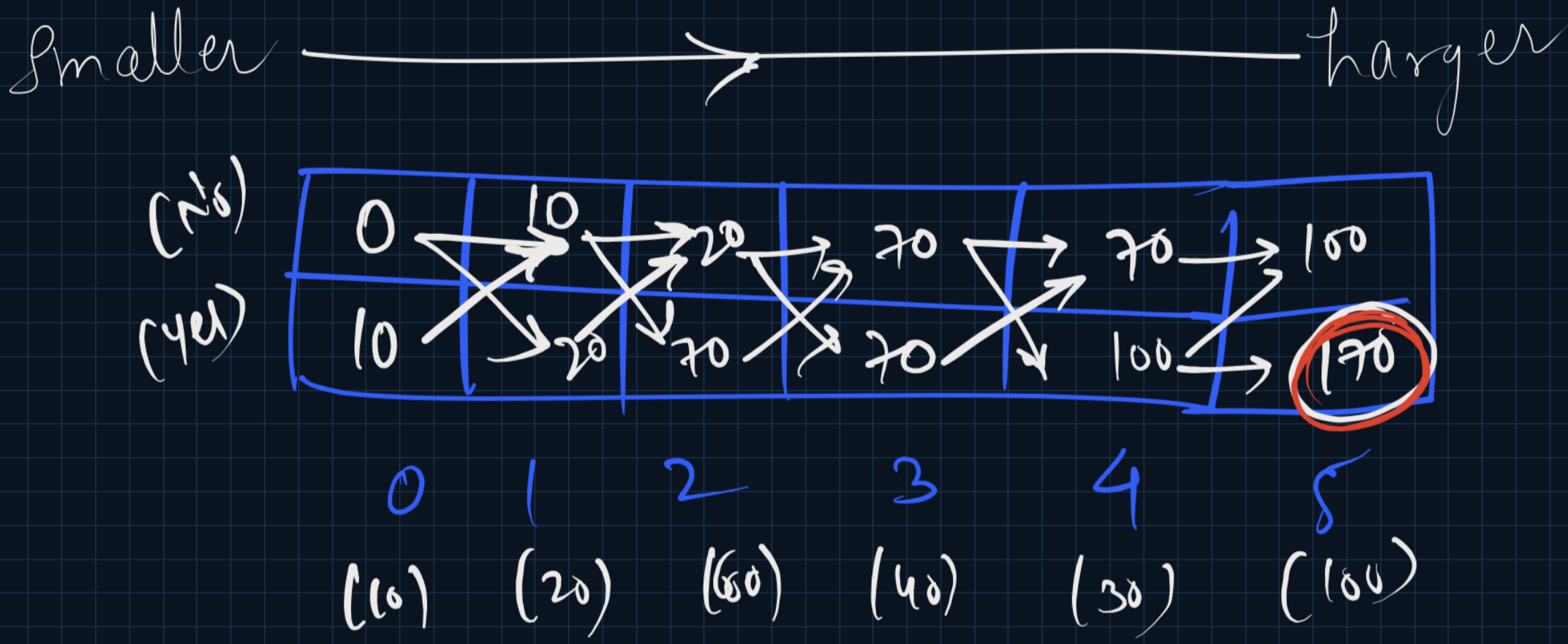
Memo

→ O(N) Time Comp

→ O(N) Space Comp

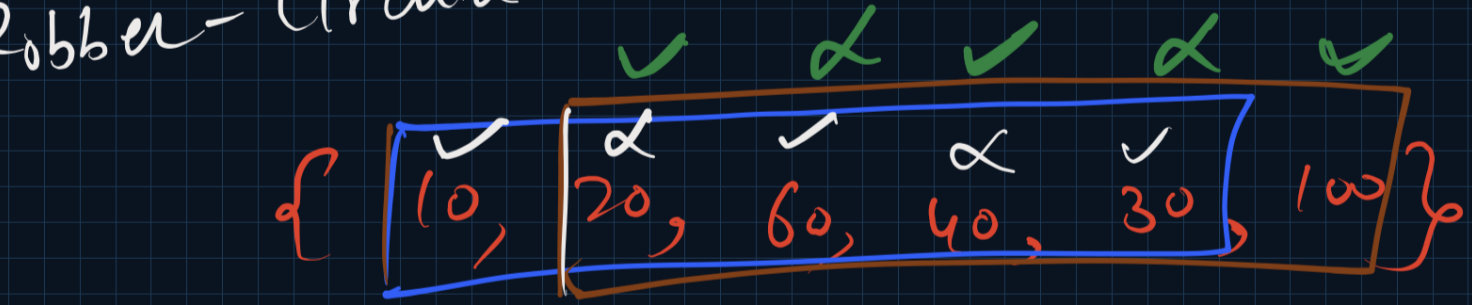
↓ Extra

↓ R.C.S



Homework :⇒ Tabulation (Code)

House Robber - Circular



Extra Constraint → 10 & 100 are adjacent
arr[0] & arr[n-1] are also adjacent !

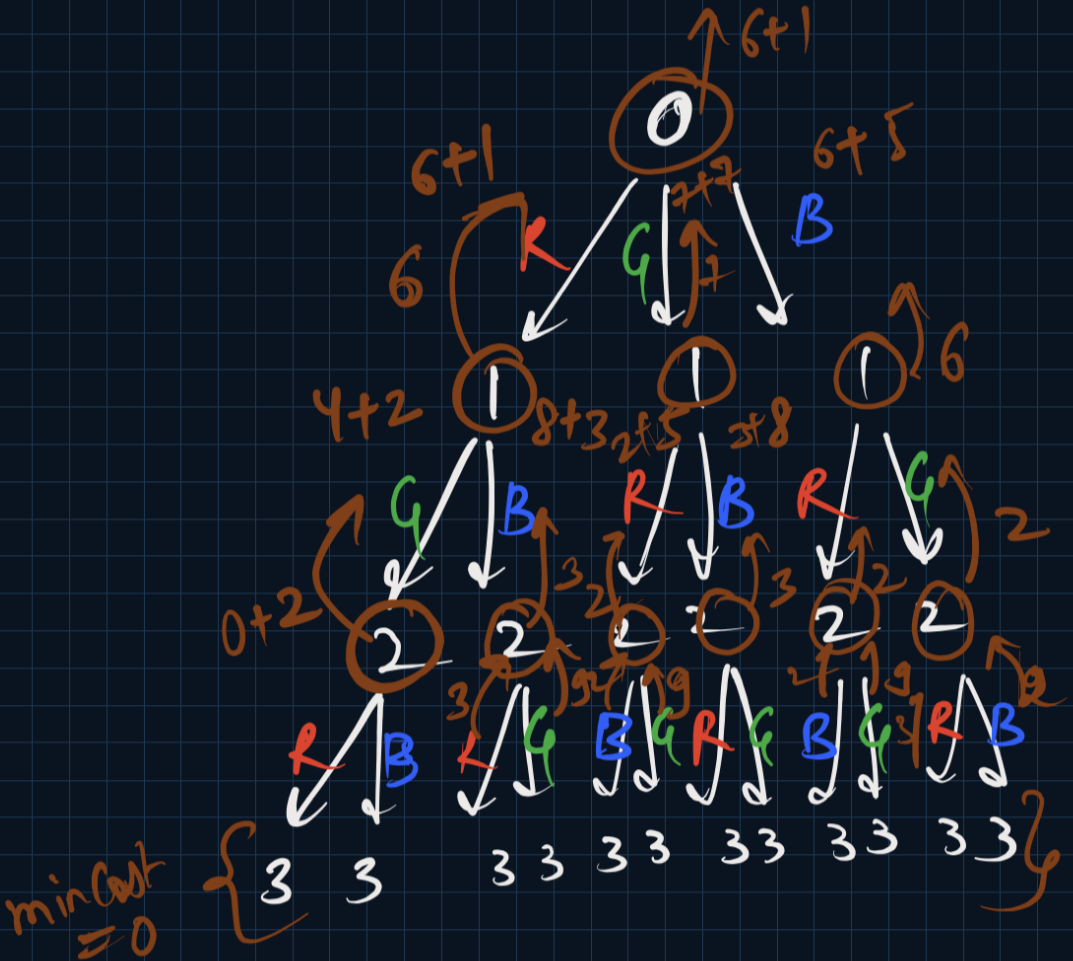
```java
public int memo(int[] nums, int idx, int n, int prev, int[][] dp){
    if(idx == n + 1) return 0;
    if(dp[idx][prev] != -1) return dp[idx][prev];

    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);
}
```

```java
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));
}
```

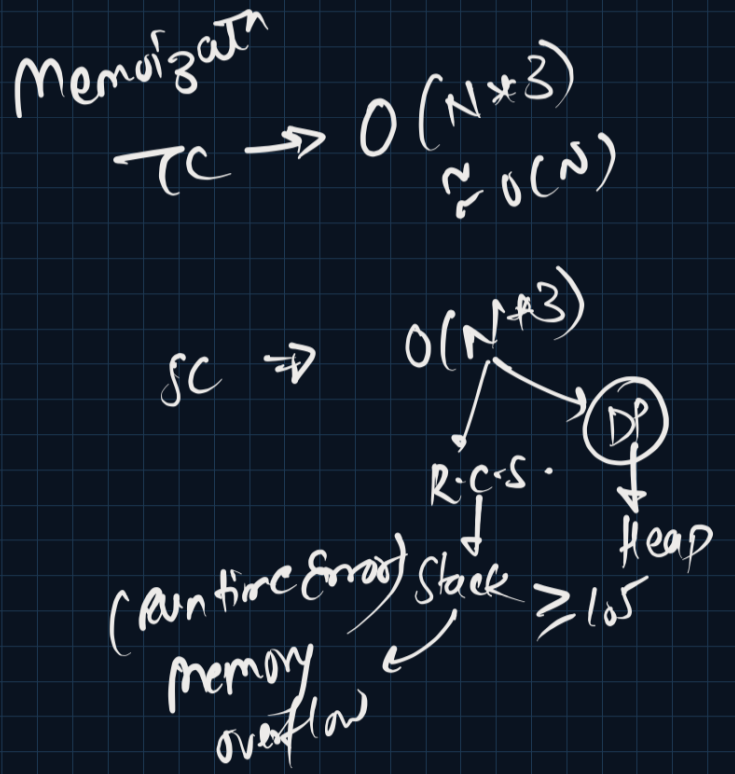
TC ⇒ O(N*2)

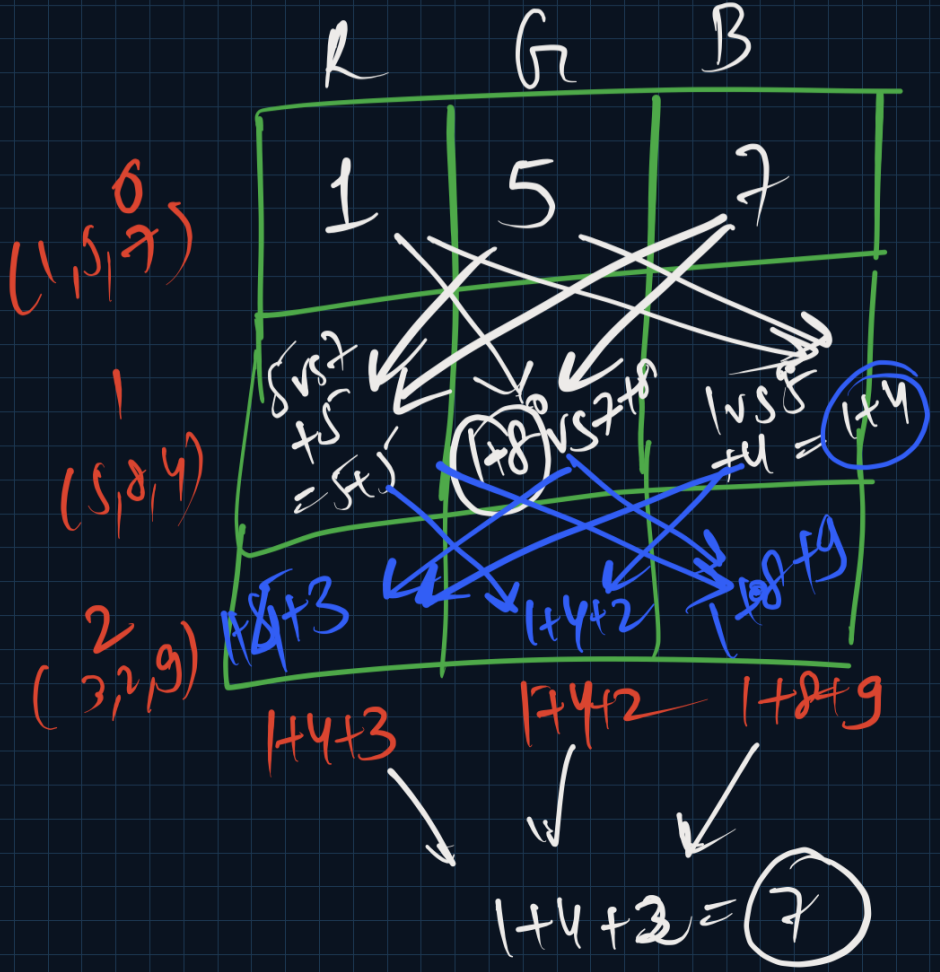
SC ⇒ O(N)

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



```java
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));
}
```





$DP(\underset{0 \to N-1}{i})(\underset{0 \to 2}{j}) = (0 \to i)$ houses paint-ing cost such that i^{th} house has color j

① Storage & meaning

② Smaller, larger problem

③ Travel & Solve (Recurrence Relation

```java
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]));
```

TC ⇒ O(N*3) = O(N)

SC ⇒ O(N*3) = O(N)

↓

DP

↓

"Stack overflow is not occuring"

```java
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

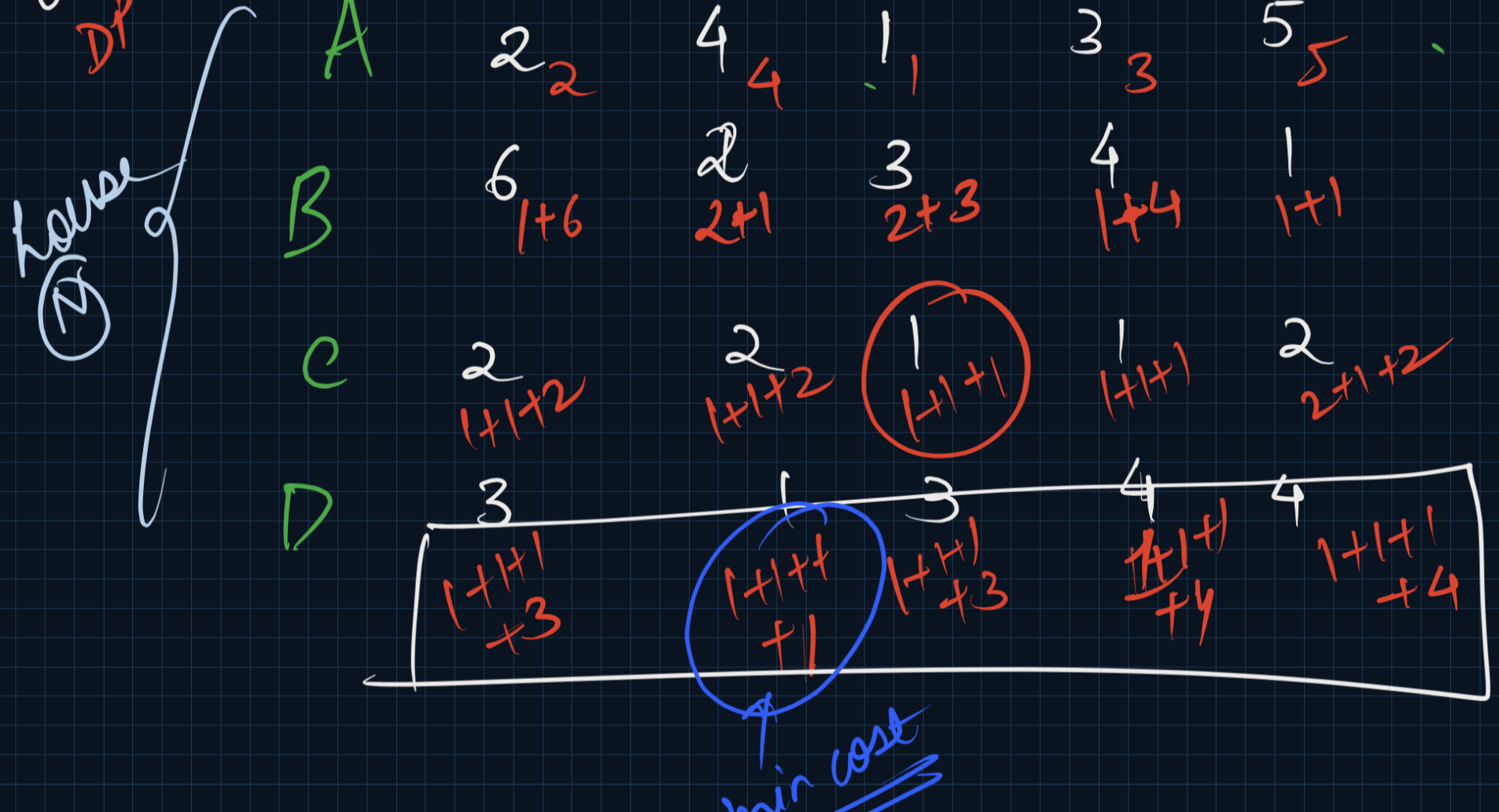
↳ O(1) Space

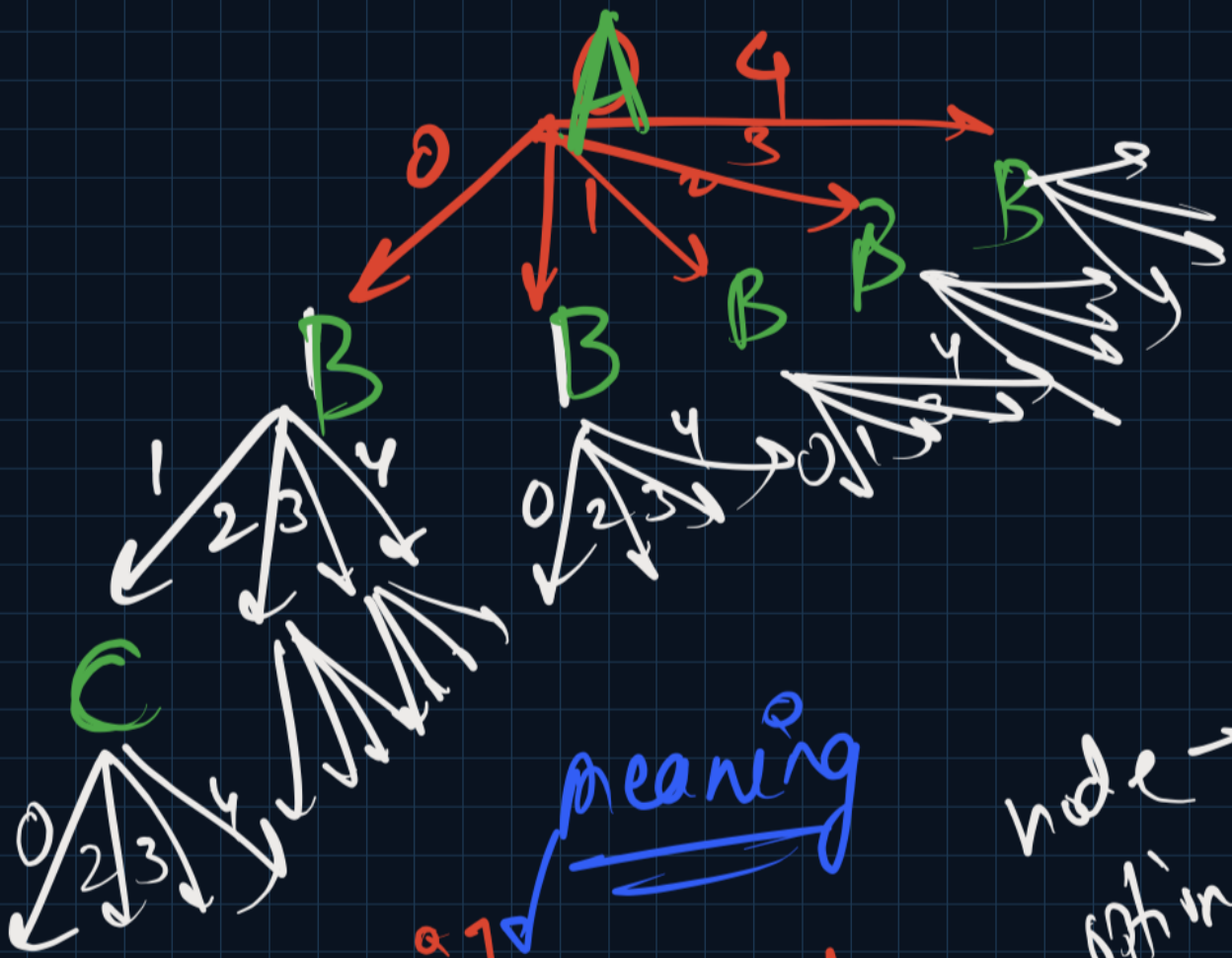
Paint House - II

colors = S → K

cost

DP





meaning

node → house

origin → color

dp[i][j]

= (0-i) houses

min cost

such that

ith house

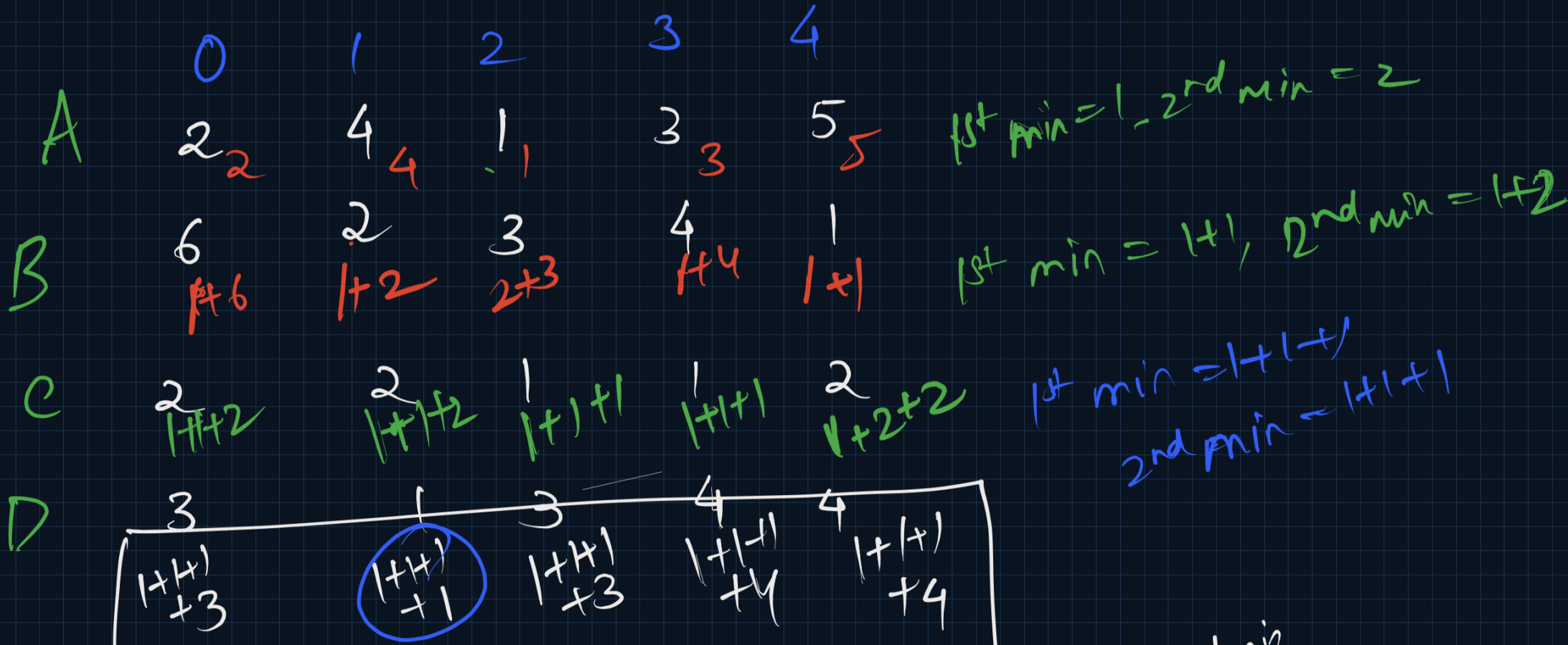
has j color

```java
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 4 1 3 5
2 4 1 3 5
1st min = 1, 2nd min = 2
B
6 2 3 4 1
6 1+2 2+3 1+4 1+1
1st min = 1+1, 2nd min = 1+2
C
2 2 1 1 2
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 +3
1+1+1 +1
1+1+1 +3
1+1+1 +4
1+1+1 +4

```java
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

O(k)

TC ⇒ O(n * (k + k)) = O(n * 2k) ≈ O(n * k)

S ⇒ O(N * k)

Paint Fence

n , k → colors

↓ Fence/houses

No more than 2 fences have the same color

n=1 → k

k=1 ①

k=2 ②

k=3 ③

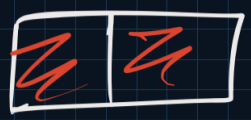
⋮

k=k (k)

n=2 → k^2

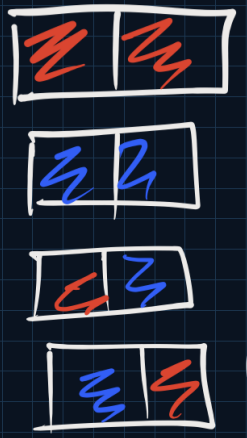
k=1

k=2

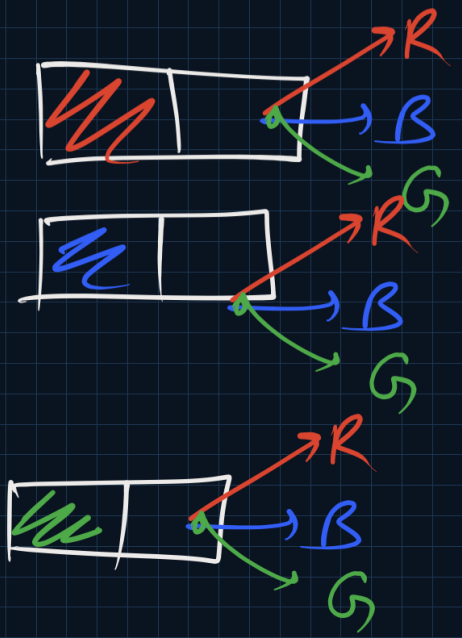


① = 1^2

k=3



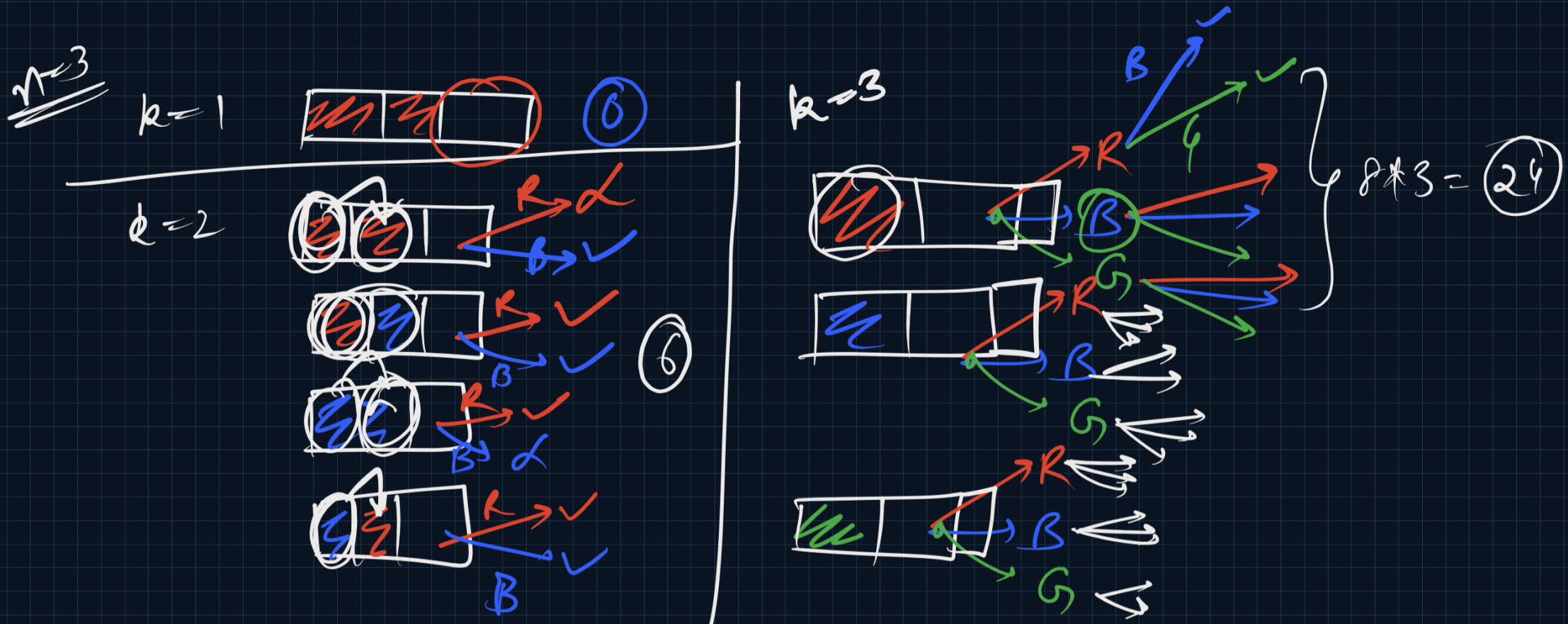
④ = 2^2



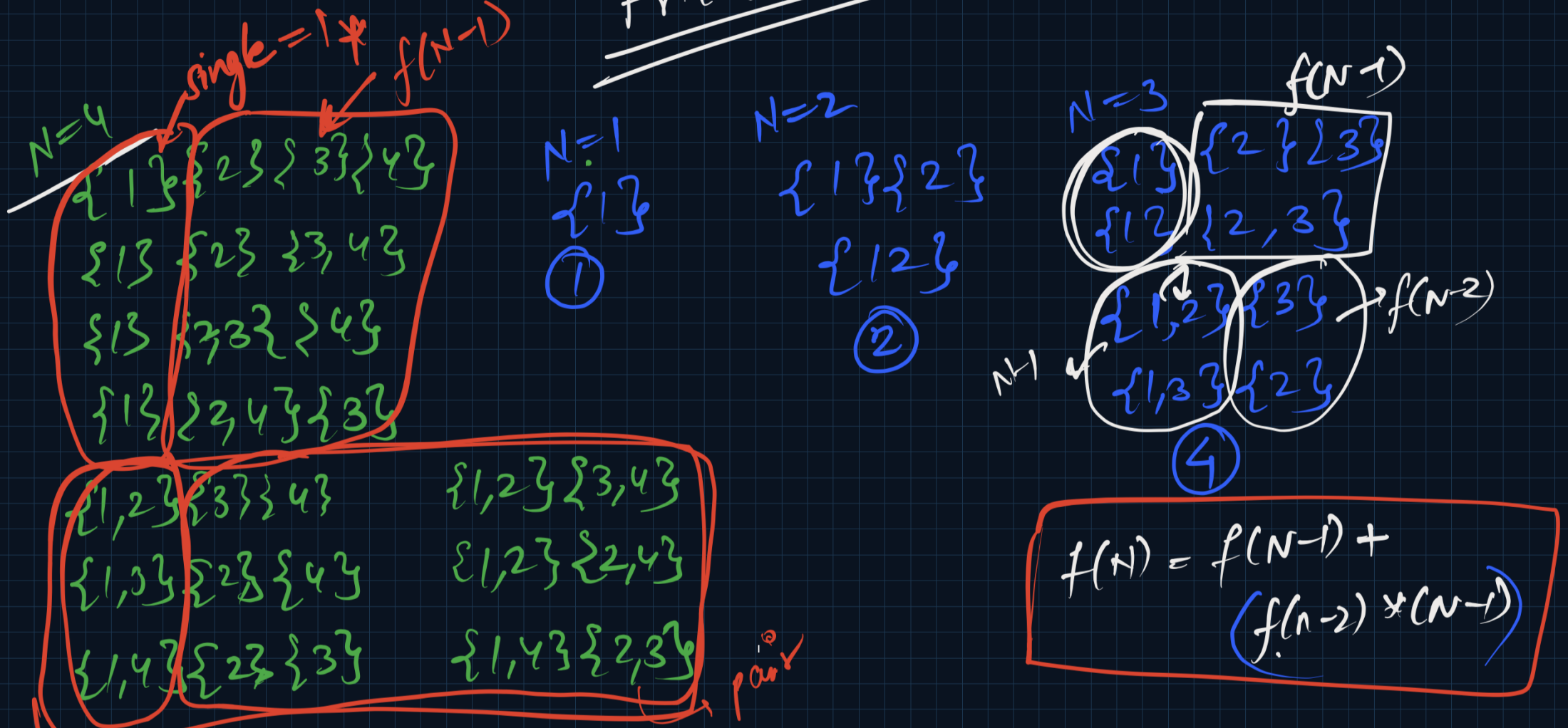
⑨ = 3^2

k=k

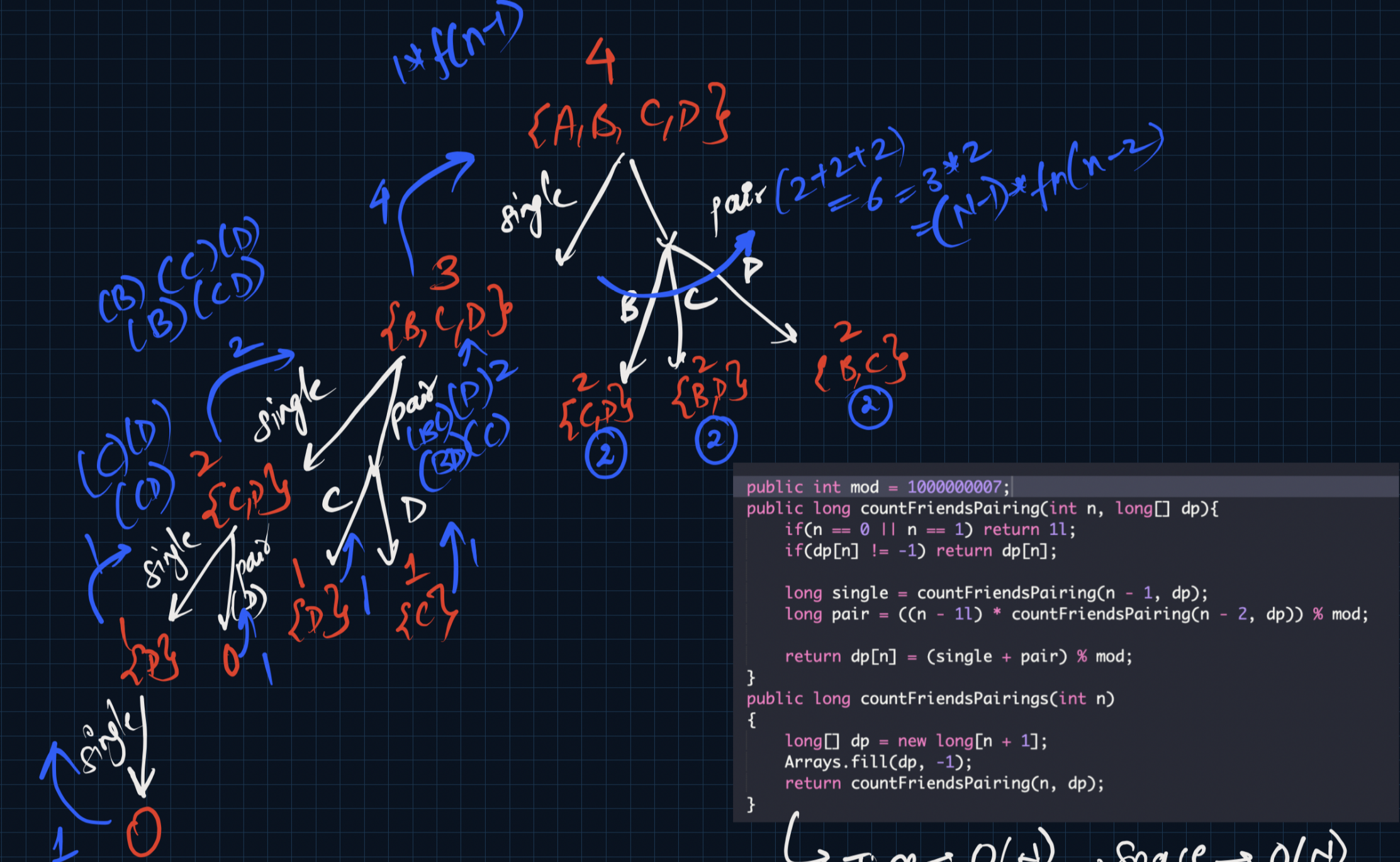
count of ways = k^2 = k*k


n=3
k=1
6
k=2
R
B
6
k=3
R
B
G
8*3=24

Friends pairing



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




```java
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 → O(N) , Space → O(N)

Space Optimizatn → Possible → Tabulation
{ Two Pointer }

Ugly No. - 1

$$N = p_1^a * p_2^b * p_3^c ----$$

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

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

$N_1 = 2^0 * 3^1 * 5^0 = 1$ ✓ true

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

Prime Factorization

Worst $O(\sqrt{N})$

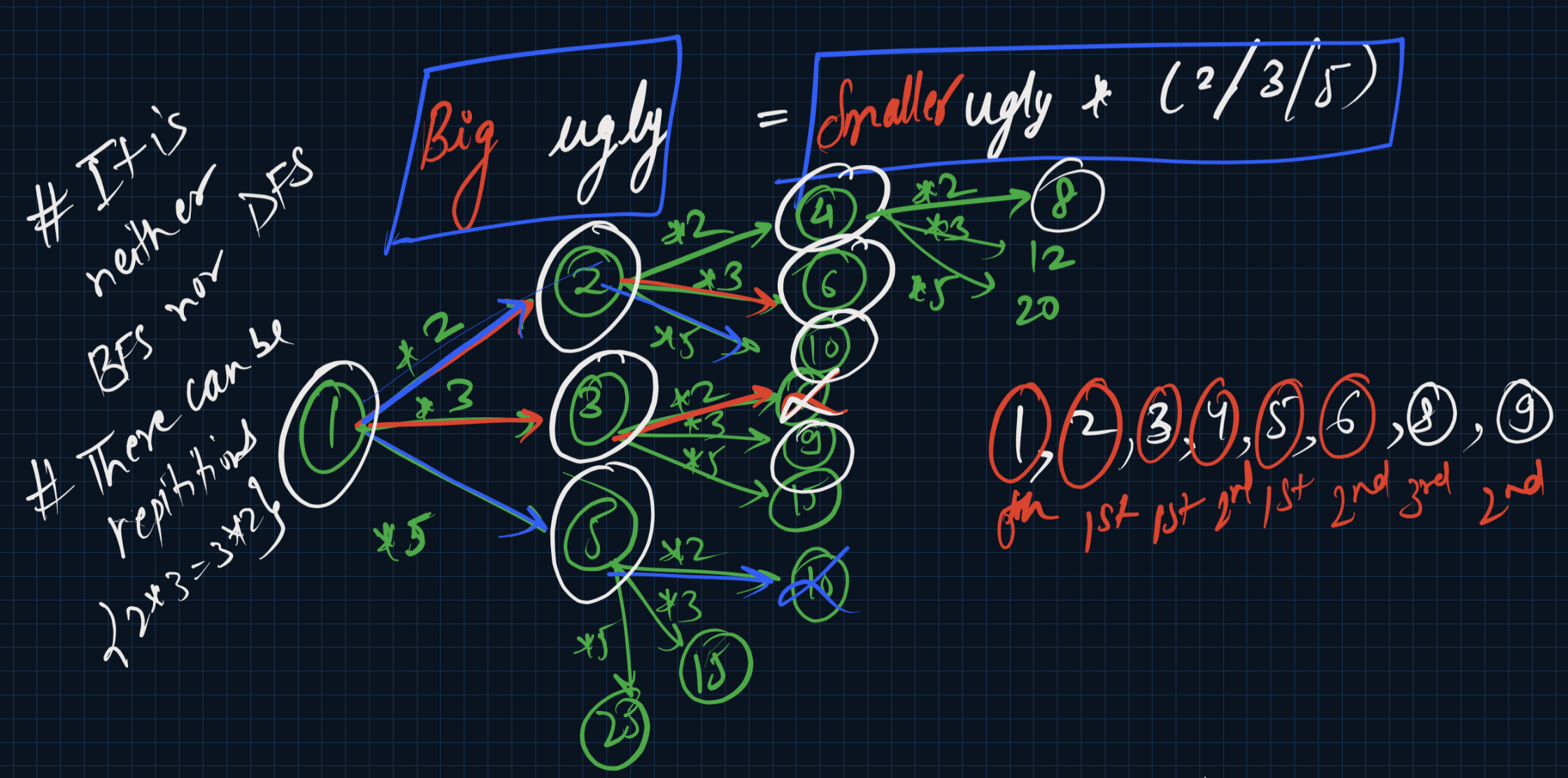
Avg $O(\log_2 N + \log_3 N + \log_5 N)$

Ugly No - II

① Brute force → loop * primefact → $O(N * \sqrt{N})$ TC

$N^{3/2}$

② 1, 2, 3, 4 (2*2), 5, 6 (2*3), 8 (2*2*2), 10 (2*5), 12 (2*2*3), 15 (3*5)


Big ugly = Smaller ugly * (2/3/5)
It is neither BFS nor DFS
There can be repitations {2x3=3*2}
*2
*3
*5
12
20
1, 2, 3, 4, 5, 6, 8, 9, 10
0th 1st 1st 2nd 1st 2nd 3rd 2nd

Dynamic Programming

→ Lecture - 8 9pm to 12PM Saturday
30 April

1. Paint Fence
2. Ugly No - II
3. Ugly No - III
4. Frog Jump
5. Jump Game - All Paths

(n houses, k colors) no more than 2 houses have the same color

N=1

k=1  (1)

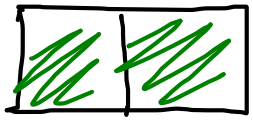
k=2  , (2)

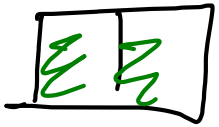
k=3  ,  ,  (3)

⋮

k=D (k)

N=2

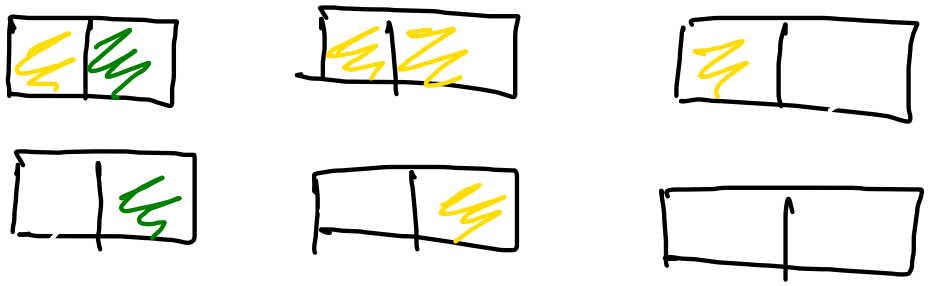
k=1  (1)

k=2     (4)

$N=2$

$k=3$

9

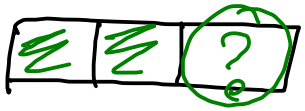


$N=2$

k → k^2

$N=3$

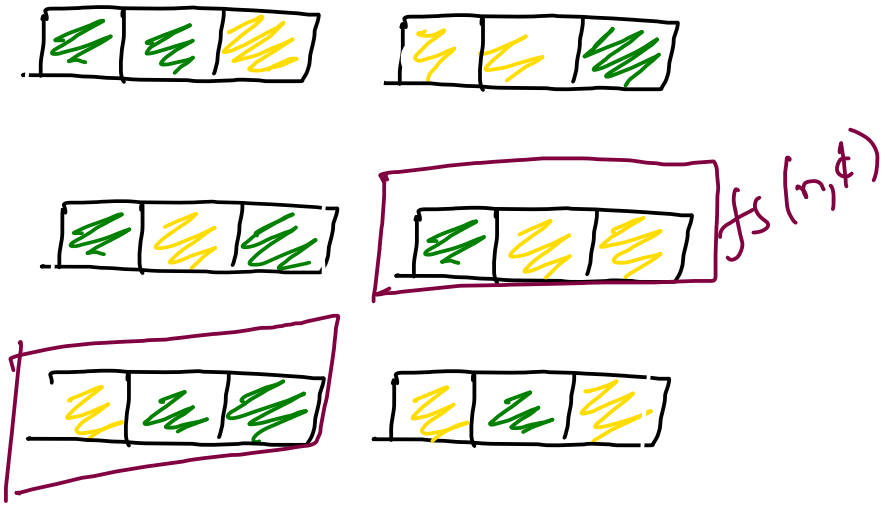
$k=1$



6

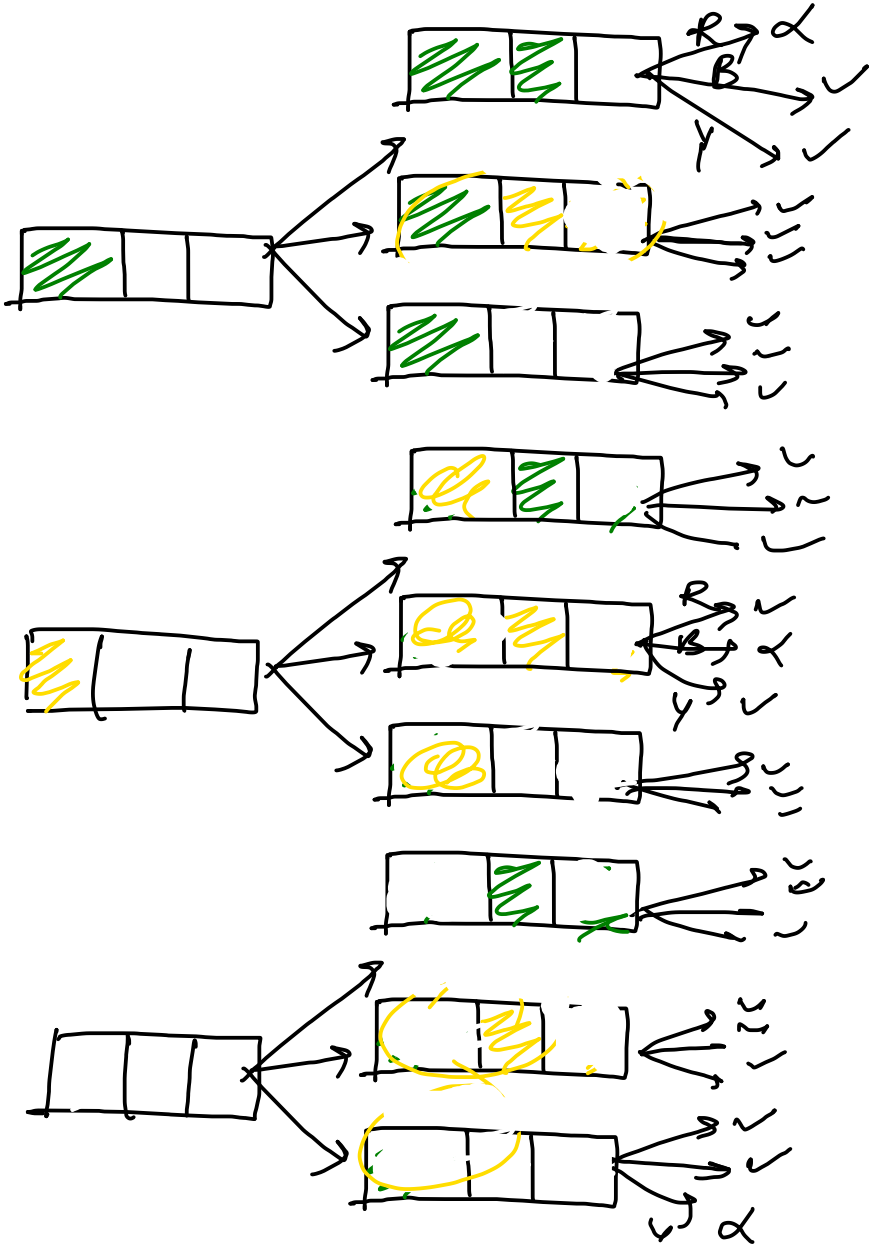
$k=2$

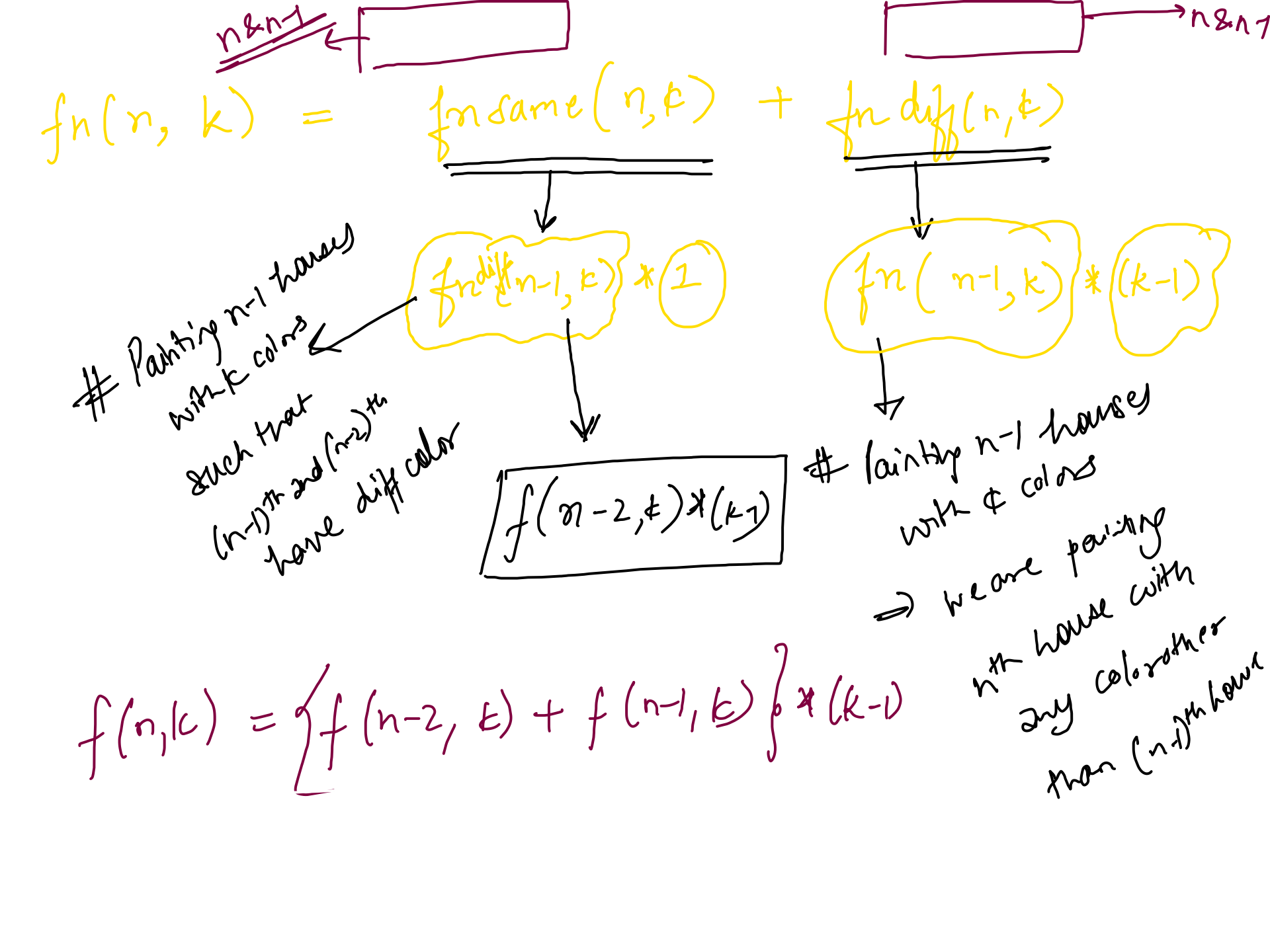
6



$k=3$

24




n&n-1
n&n-1
fn(n, k) = fn same(n,k) + fn diff(n,k)
fn diff(n-1, k) * (k-1)
fn(n-1, k) * (k-1)
# Painting n-1 houses with k colors such that (n-1)th and (n-2)th have diff color
f(n-2, k) * (k-1)
# painting n-1 houses with k colors
⇒ we are painting nth house with any color other than (n-1)th house
f(n,k) = f(n-2, k) + f(n-1, k) * (k-1)

Tabulation

$k = 3$

	N=1	N=2	N=3	N=4
last 2 same	1	3	3*2	18
diff	1	3*2	(3+3*2)*2 = 18	(18+6)*2 = 48

TC ⇒ O(N)

SC ⇒ O(N) → Space can be optimized

```java
public class Solution {
    public int memo(int n, int k, int[] dp){
        if(n == 1) return k;
        if(n == 2) return k * k;
        if(dp[n] != -1) return dp[n];

        int ans = (memo(n - 1, k, dp) + memo(n - 2, k, dp)) * (k - 1);
        return dp[n] = ans;
    }

    public int numWays(int n, int k) {
        if(n == 1) return k;
        if(n == 2) return k * k;
        if(k == 1) return 0;

        int[] dp = new int[n + 1];
        Arrays.fill(dp, -1);
        return memo(n, k, dp);
    }
}
```

```java
public int numWays(int n, int k) {
    if(n == 1) return k;
    if(n == 2) return k * k;
    if(k == 1) return 0;

    int[] same = new int[n + 1];
    int[] diff = new int[n + 1];
    same[2] = k; diff[2] = k * (k - 1);

    for(int i=3; i<=n; i++){
        same[i] = diff[i - 1];
        diff[i] = (same[i - 1] + diff[i - 1]) * (k - 1);
    }

    return same[n] + diff[n];
}
```

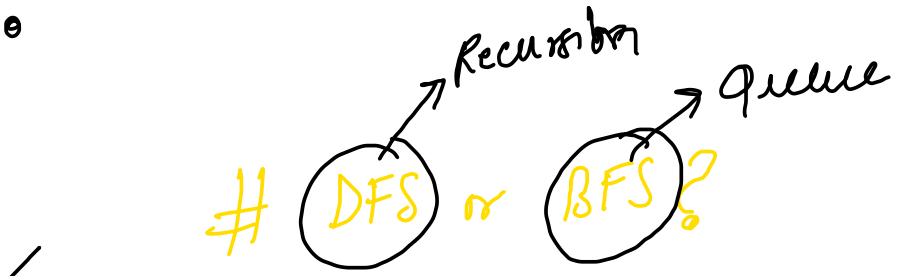
```java
public int numWays(int n, int k) {
    if(n == 1) return k;
    if(n == 2) return k * k;
    if(k == 1) return 0;

    int same = k;
    int diff = k * (k - 1);

    for(int i=3; i<=n; i++){
        int newSame = diff;
        int newDiff = (same + diff) * (k - 1);

        same = newSame; diff = newDiff;
    }

    return same + diff;
}
```

Ugly No - 11

0 1 2 3 4 5 6 8

1, 2, 3, 4, 5, 6, 8, 10 →

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

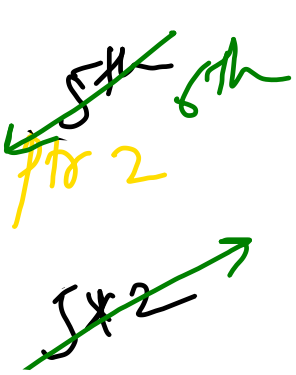


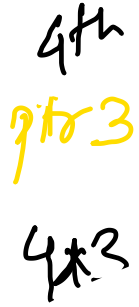
Overlapping subproblems (DP)

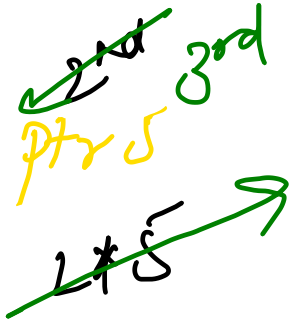
↳ 2*3 = 3*2

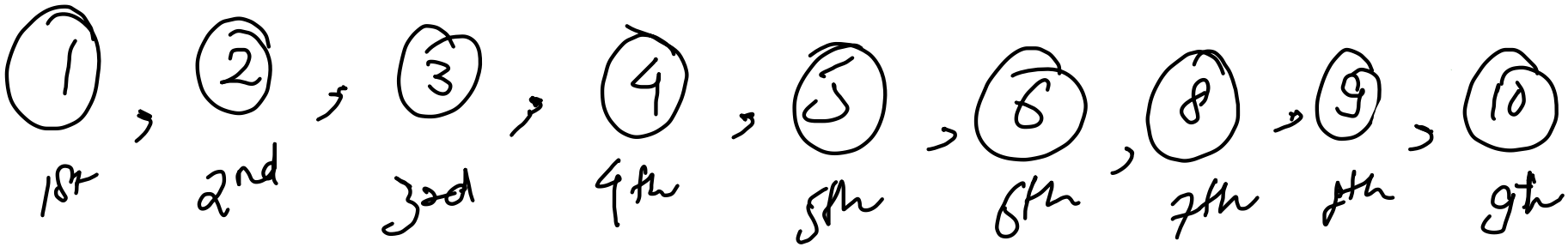
Optimal substructure (Faith)

Bigger Ugly = smaller Ugly * (2/3/5)


5th
6th
ptr 2
5*2


4th
ptr 3
4*3


2nd
3rd
ptr 5
2*5


1
2
3
4
5
6
8
9
10
1st
2nd
3rd
4th
5th
6th
7th
8th
9th

```java
// O(N) Time, O(N) Space
public int nthUglyNumber(int n) {
    if(n == 1) return 1;

    // Pointers Pointing to Indices not Values
    int ptr2 = 0, ptr3 = 0, ptr5 = 0;

    ArrayList<Integer> ugly = new ArrayList<>(); // visited array
    ugly.add(1); // to add the 1st ugly no at index 0

    for(int i=1; i<n; i++){
        int a = ugly.get(ptr2) * 2;
        int b = ugly.get(ptr3) * 3;
        int c = ugly.get(ptr5) * 5;

        int min = Math.min(a, Math.min(b, c));
        ugly.add(min);

        if(min == a) ptr2++;
        if(min == b) ptr3++;
        if(min == c) ptr5++;
    }

    return ugly.get(n - 1);
}
```

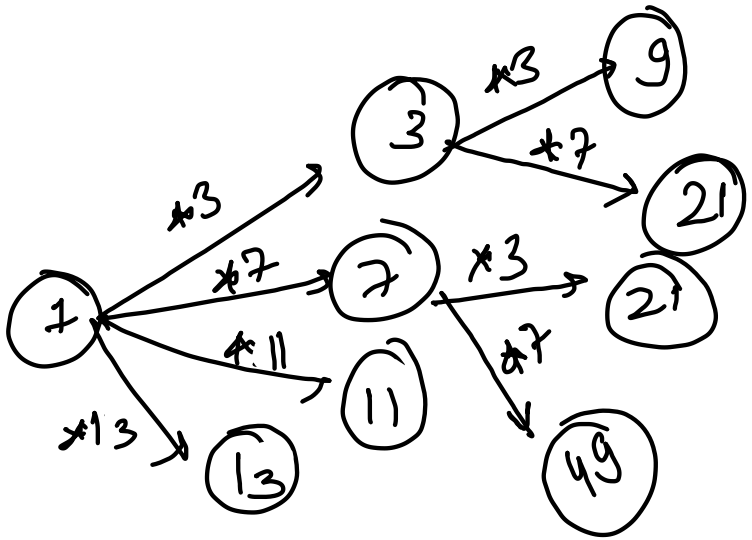
```java
// O(N * Log N) Time, O(N) Space
public int nthUglyNumber(int n) {
    if(n == 1) return 1;

    PriorityQueue<Long> q = new PriorityQueue<>();
    q.add(1l);
    HashSet<Long> vis = new HashSet<>();

    int idx = 0;
    while(q.size() > 0){
        long min = q.remove();
        if(vis.contains(min) == true)
            continue;

        idx++;
        if(idx == n) return (int)min;

        vis.add(min);
        q.add(min * 2l);
        q.add(min * 3l);
        q.add(min * 5l);
    }

    return 1;
}
```

K (Input) primes = { 3, 7, 11, 13 }

N=7

7th ugly ?

(Extra =O(K)) ptr = { 4, 3, 2, 2 }



Res (Extra) = O(N)

1 , 3 , 7 , 9 , 11 , 13 , 21

1st 2nd 3rd 4th 5th 6th 7th

Space Comp

O(N+K)

Res

pointers in arr

```java
// O(N * K) Time, O(N + K) Space
public int nthSuperUglyNumber(int n, int[] primes) {
    int[] ptr = new int[primes.length];

    ArrayList<Integer> ugly = new ArrayList<>();
    ugly.add(1); // Add 1st Ugly No at Index 0

    for(int i=1; i<n; i++){
        // Finding the next Smallest Ugly No
        int min = Integer.MAX_VALUE;
        for(int j=0; j<primes.length; j++)
            min = Math.min(min, ugly.get(ptr[j]) * primes[j]);

        ugly.add(min);

        // Updating All Pointers Pointing to Min
        for(int j=0; j<primes.length; j++)
            if(ugly.get(ptr[j]) * primes[j] == min)
                ptr[j]++;
    }

    return ugly.get(n - 1);
}
```