Dynamic Programming

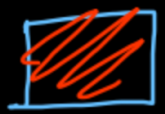
→ Lecture - 8 9 am to 12 pm Saturday
30 April

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

Paint Fence

(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)

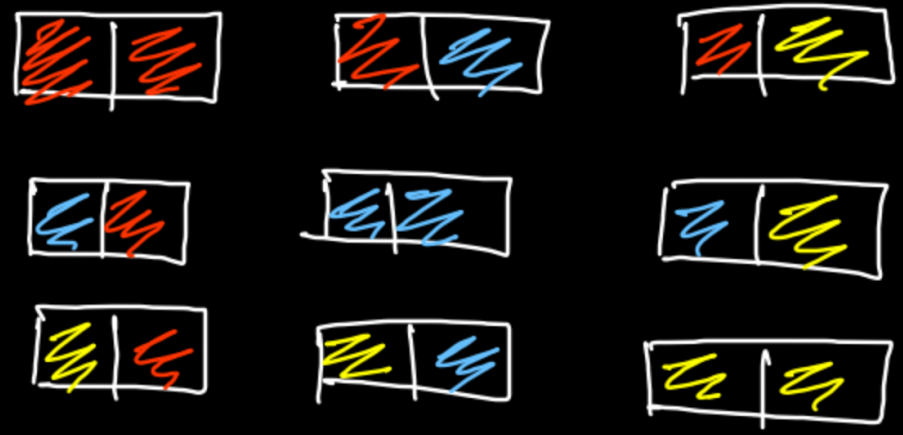
N=2

k=1  ①

k=2  ④

N = 2

k = 3



9

N = 2

k → k^2

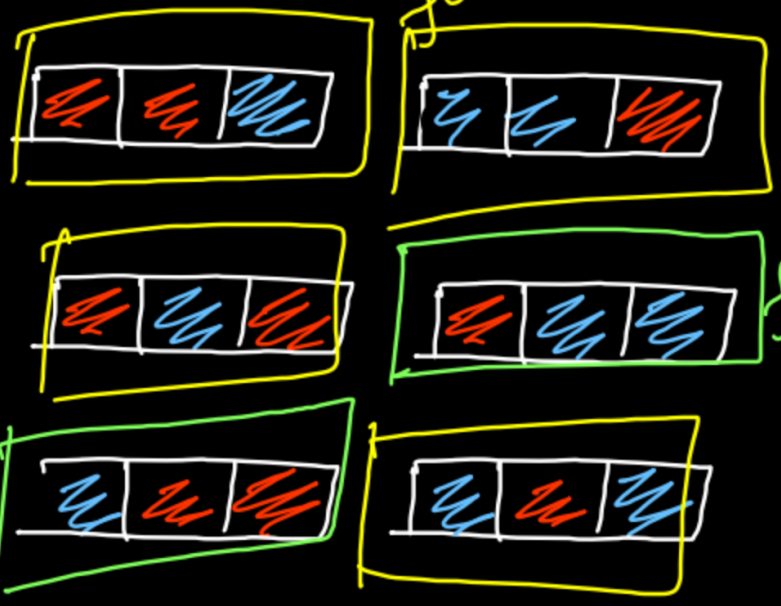
N=3

k=1

6

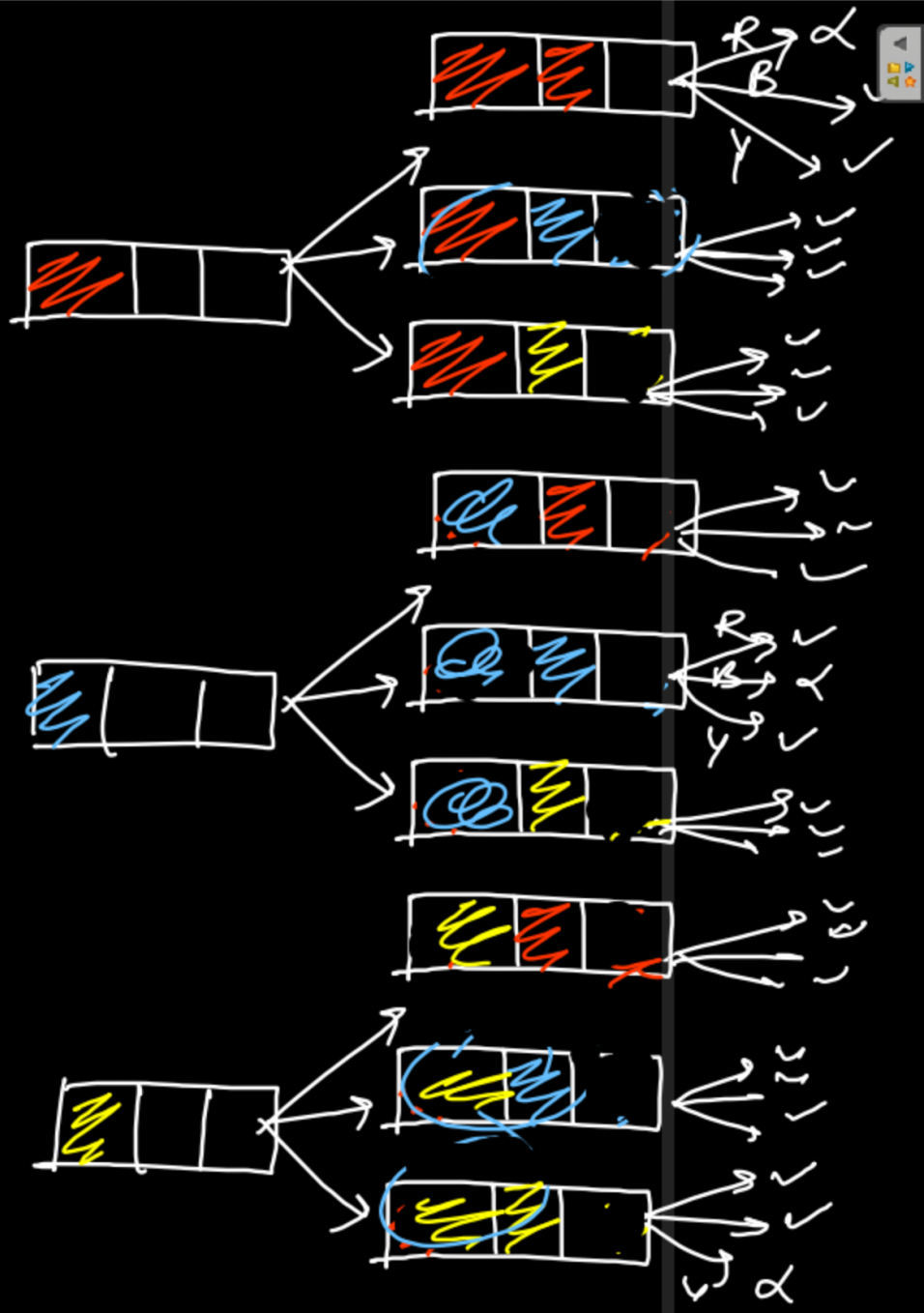
k=2

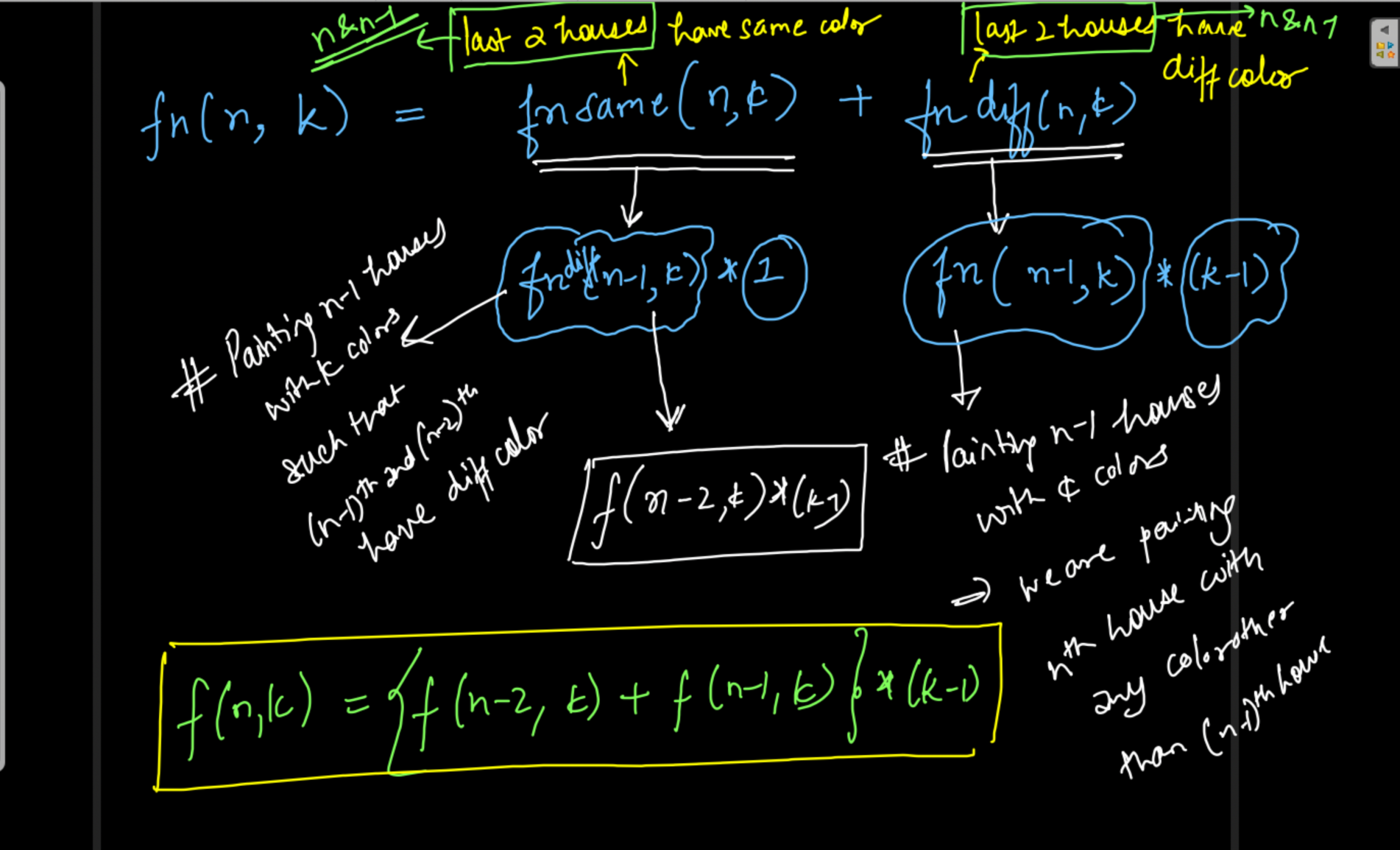
6



k=3

24




n & n-1
last 2 houses have same color
last 2 houses have n & n-1 diff color
fn(n, k) = fn same(n, k) + fn diff(n, k)
fn diff(n-1, 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
least 2 houses 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);
    }
}
```

0.7 x

```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];
}
```

0.7 x

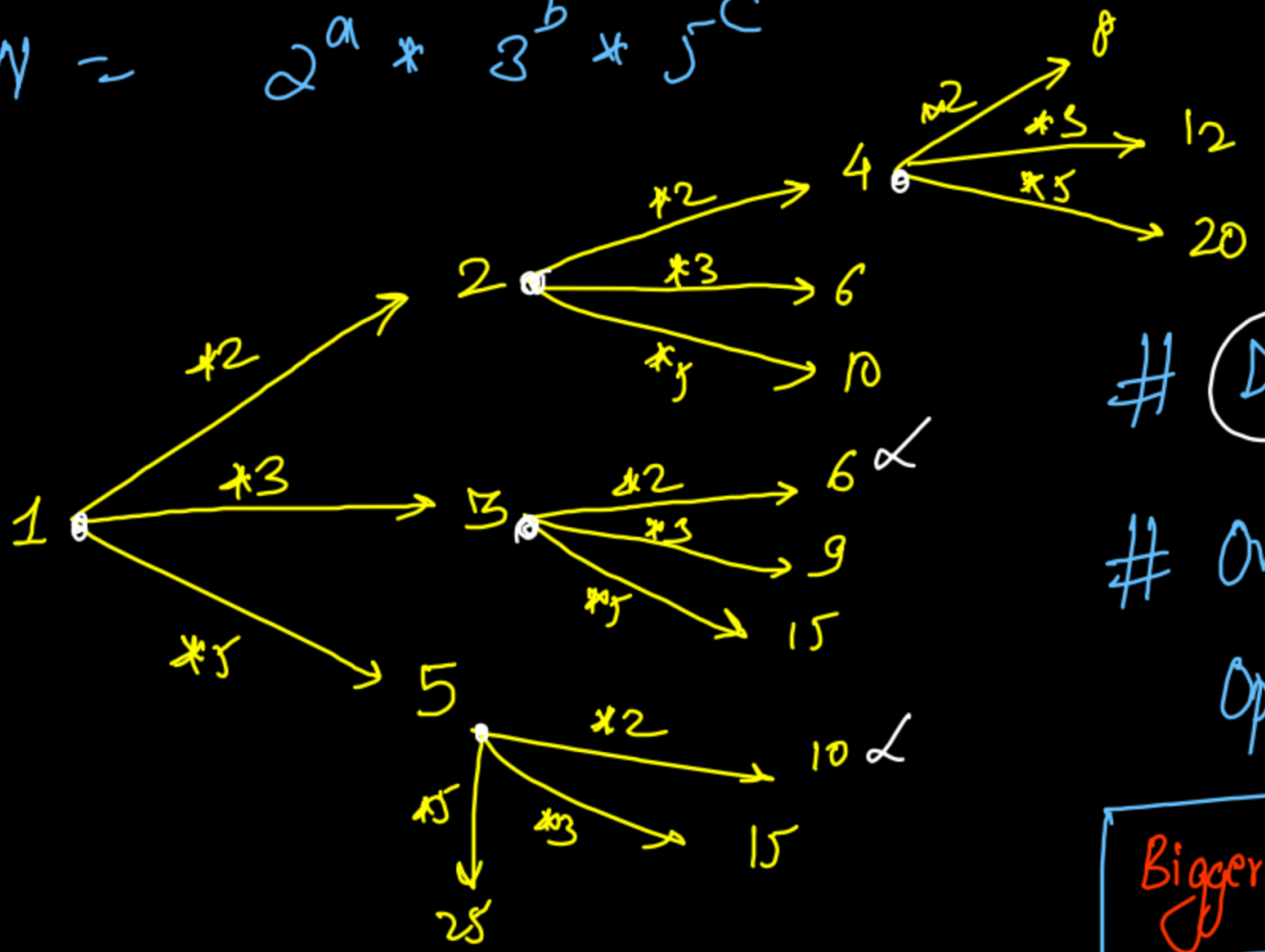
```java
public int numWays(int n, int k) {
    if(n == 1) return k;
    if(n == 2) return k * k;
    if(k == 1) return 0;

    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 →

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



DFS or BFS ?

Recursion

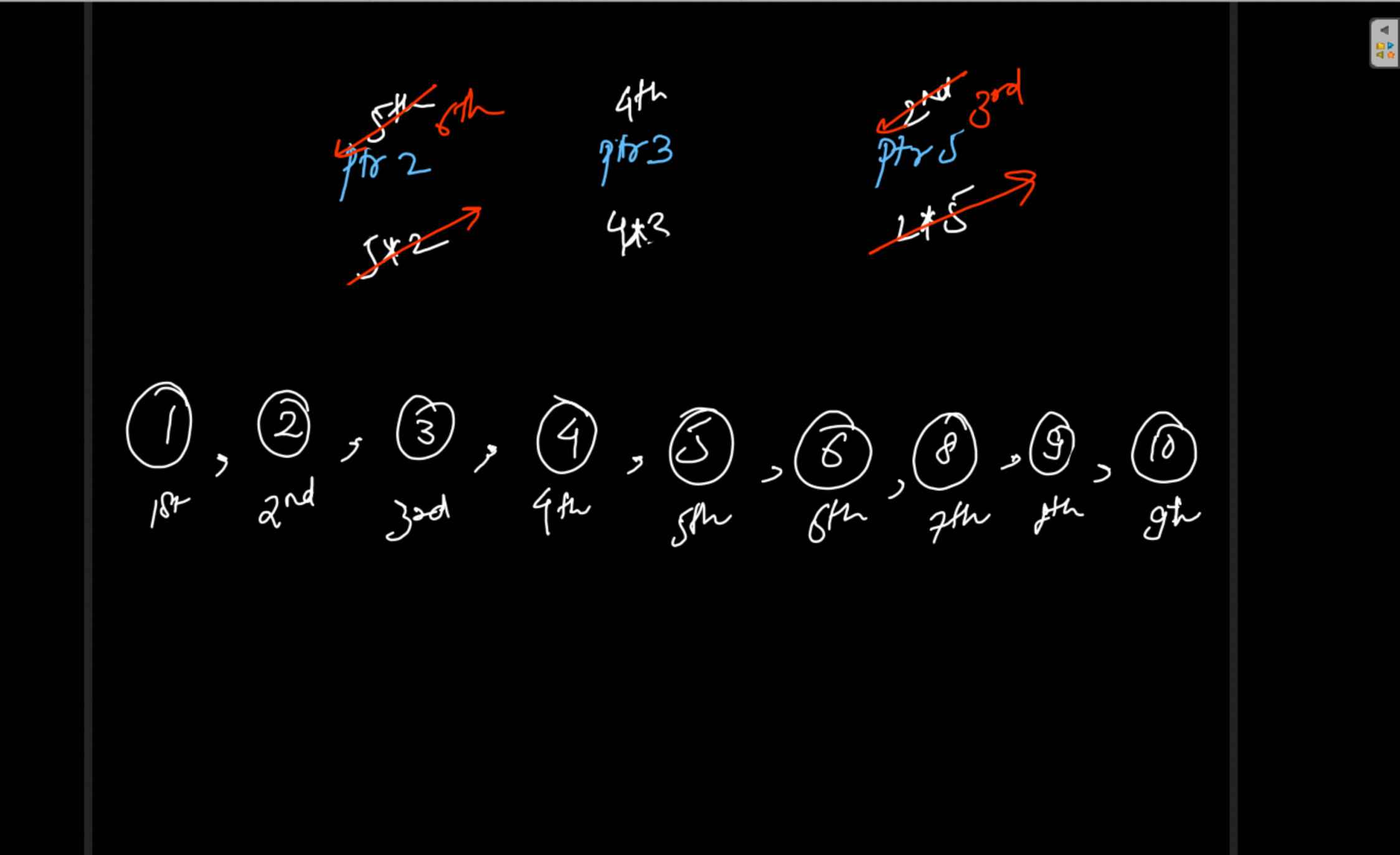
queue

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
1*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);
}
```

0.7 x

```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;
}
```

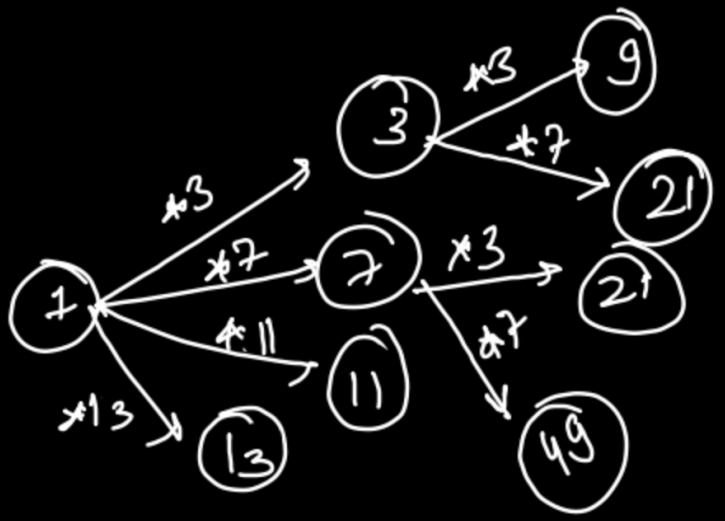
0.5 x

Super Ugly No

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

N=7 → 7th ugly ?

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



Res (Extra) = O(N)

1 (1st), 3 (2nd), 7 (3rd), 9 (4th), 11 (5th), 13 (6th), 21 (7th)

Space Comp O(N+K)

Res

pointers index

```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);
}
```

Dynamic Programming - Lecture (9)

9 AM - 12 PM

Sunday, 1 May

count of ways
- → Coin Change Permutations
- → Coin change Combinations

minimum steps
- → Minimum Coin change
- → Indian Coin change

Coin change → minimum coins (Indian)

eg1 {1, 2, 5, 10}

target = 79

Why greedy is working?

(10 * 7) + (5 * 1) + (2 * 2)

7 + 1 + 2 = 10 coins

eg2 uniformity coins multiples of 2, 5

{1, 2, 5, 10, 20, 50, 100, 500, 2000} ← Indian denomination ⇒ greedy will also work

target: 2499

2000 * 1 + 100 * 4 + 50 * 1 + 20 * 2 + 5 * 1 + 2 * 2

1 + 4 + 1 + 2 + 1 + 2 = 11 coins

```java
static List<Integer> minPartition(int target)
{
    List<Integer> res = new ArrayList<>();
    int[] coins = {1, 2, 5, 10, 20, 50, 100, 200, 500, 2000};
    int count = 0;

    for(int i=coins.length-1; i>=0; i--){

        while(target - coins[i] >= 0){
            res.add(coins[i]);
            target -= coins[i];
        }

        if(target == 0) break;
    }

    return res;
}
```

Time Comp

↳ $\frac{N}{2000} + \frac{N}{500} + \frac{N}{200} + \cdots \frac{N}{1}$

$\Rightarrow O(N + N + N + \cdots N) = O(N*10)$

$\approx O(N)$

Space Complexity → O(1) extra space

O(n) output space

322. Coin Change → Minimum

Coins = {2, 3, 7} target = 8

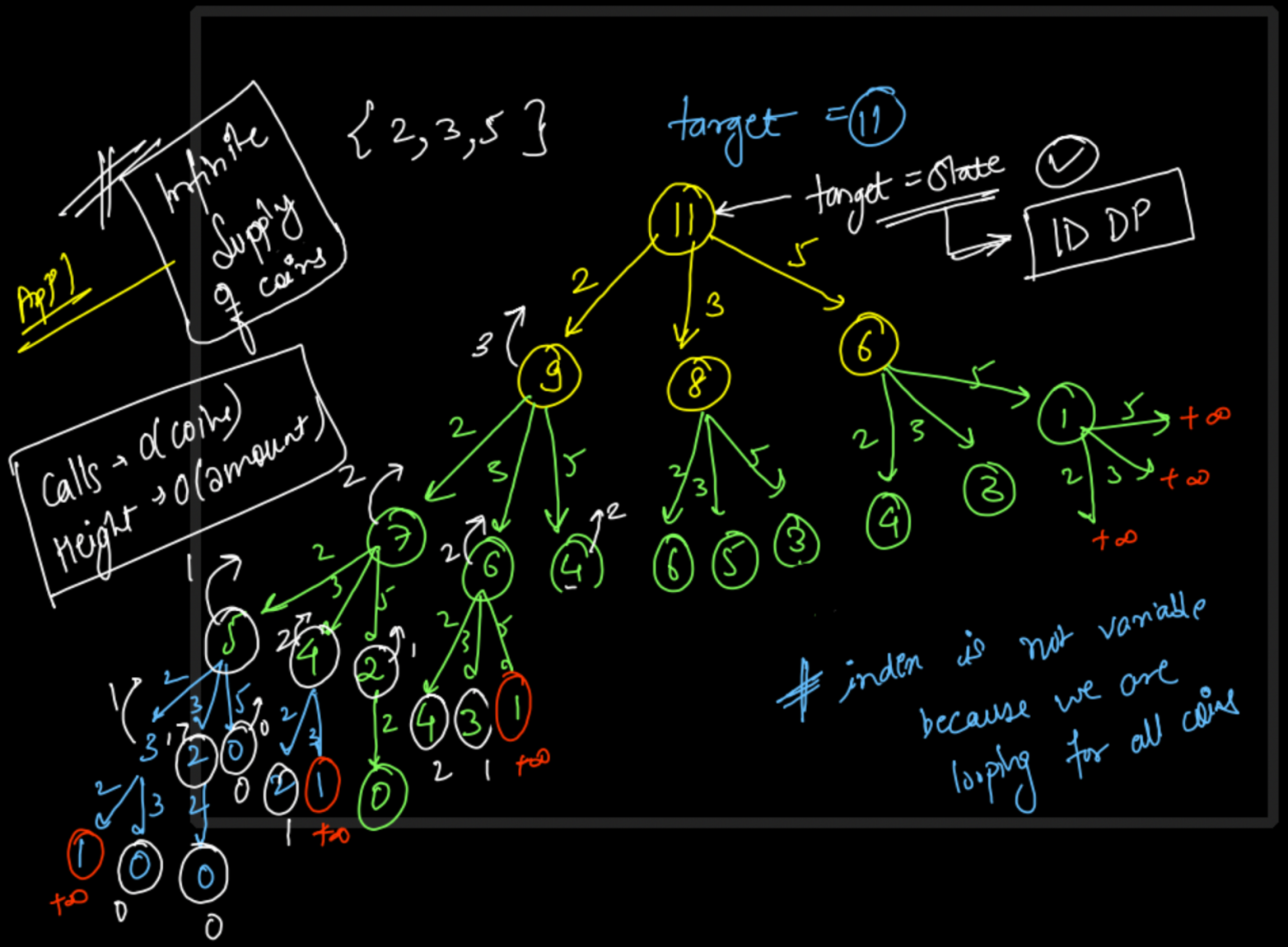
Greedy: target = 8 − 7 = 1 → target. } no answer possible

Actual res: target = 8 − 3 = 5 (not the biggest coin)

5 − 3 = 2

2 − 2 = 0 → target

ans = 3


Infinite supply of coins
{2,3,5}
target = 11
target = State
1D DP
Calls → O(coins)
Height → O(amount)
index is not variable
because we are
looping for all coins
0.7 x

```java
// Time Complexity: O(Amount * Coins), Space Complexity: O(Amount)
public int memo(int amount, int[] coins, int[] dp){
    if(amount == 0) return 0;
    if(dp[amount] != -1) return dp[amount];

    int minCoins = Integer.MAX_VALUE;
    for(int i=0; i<coins.length; i++){
        if(amount - coins[i] >= 0){
            minCoins = Math.min(minCoins, memo(amount - coins[i], coins, dp));
        }
    }

    if(minCoins < Integer.MAX_VALUE) minCoins += 1;
    return dp[amount]  = minCoins;
}

public int coinChange(int[] coins, int amount) {
    int[] dp = new int[amount + 1];
    Arrays.fill(dp, -1);

    int ans = memo(amount, coins, dp);
    return (ans == Integer.MAX_VALUE) ? -1 : ans;
}
```

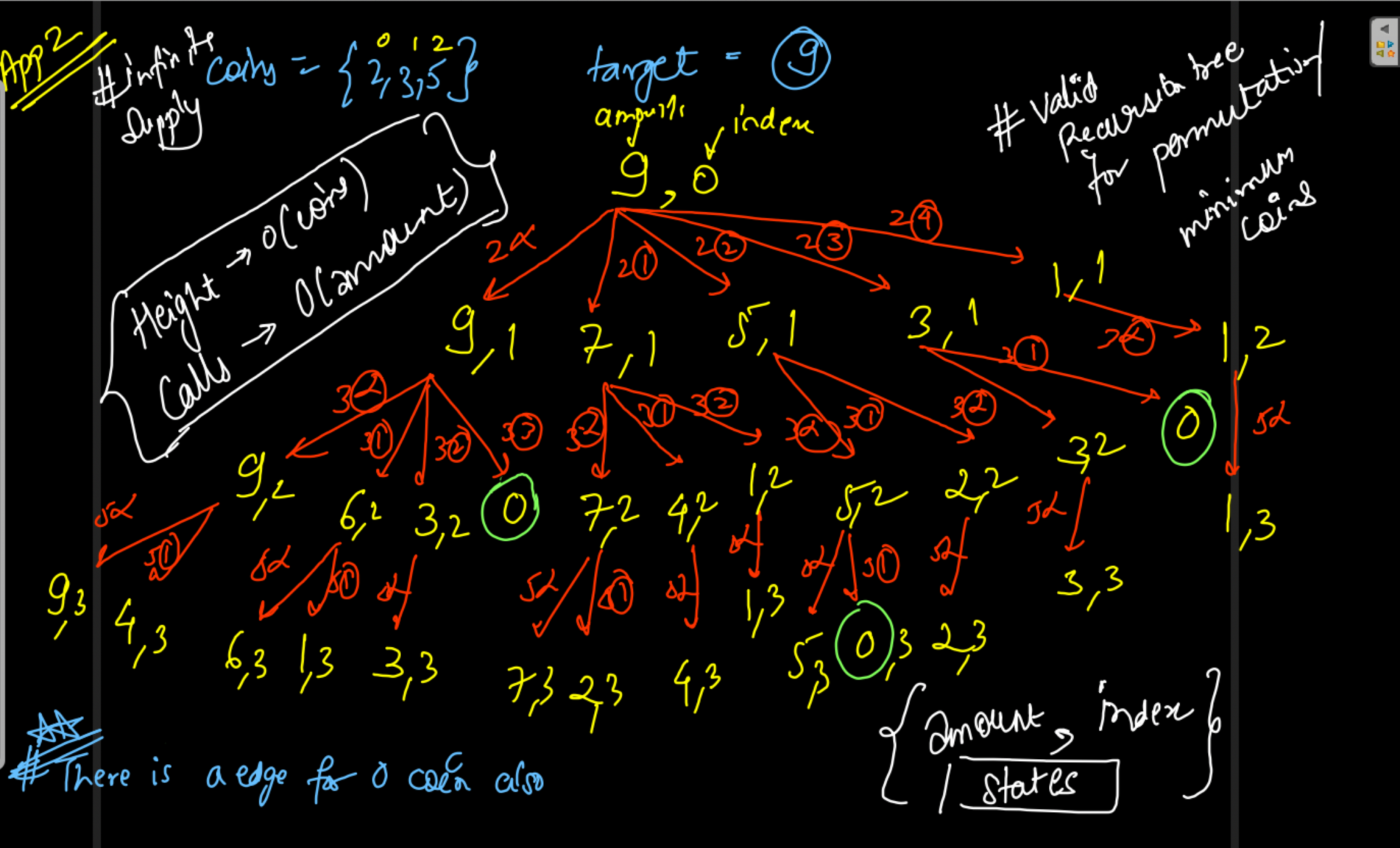
App 2

infinite supply

coins = {2, 3, 5}

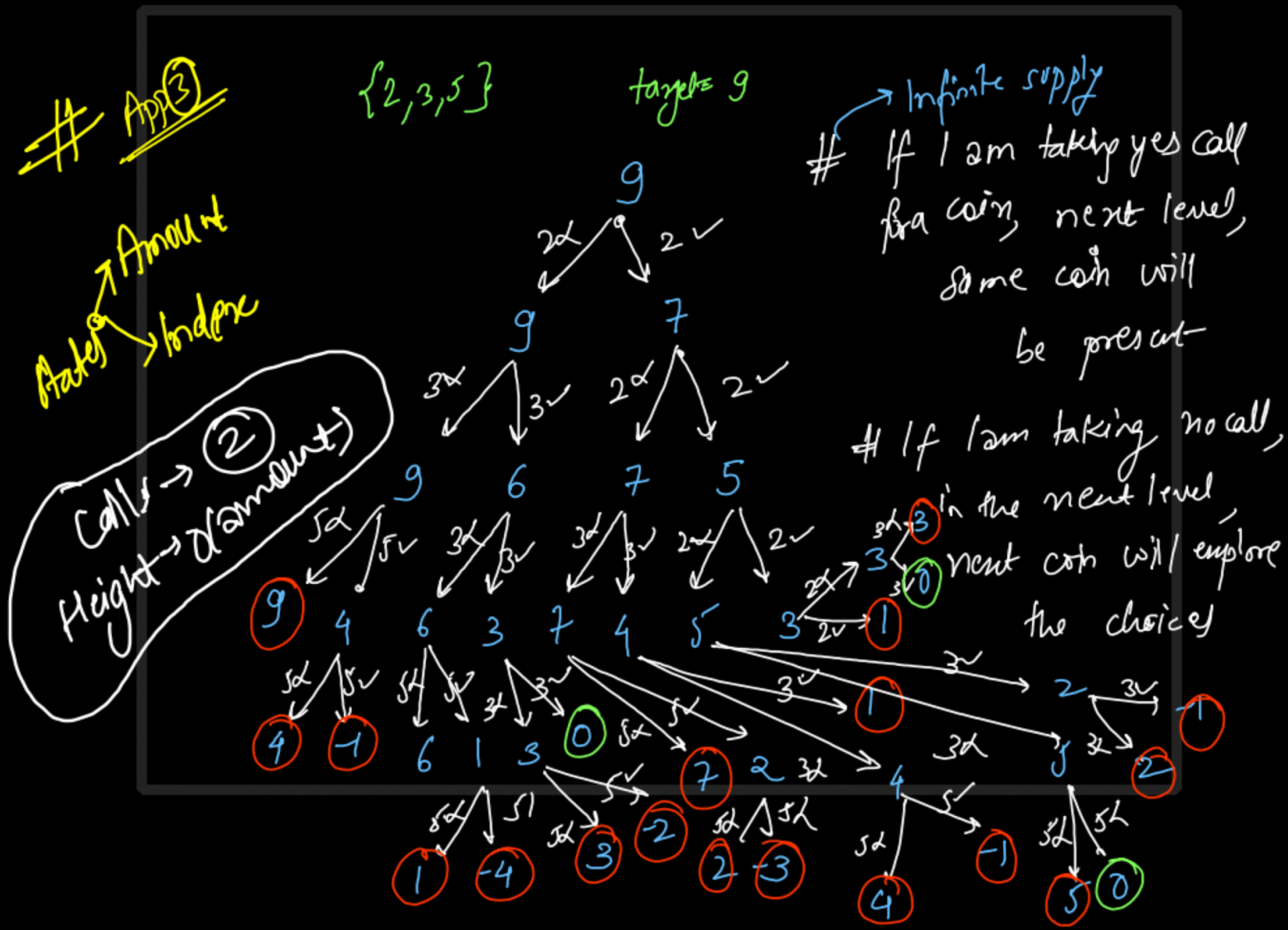
target = 9

amount, index



```java
// Time Complexity: O(Amount * Coins), Space Complexity: O(Amount * Coins)
public int memo(int amount, int idx, int[] coins, int[][] dp){
    if(amount == 0) return 0;
    if(idx == coins.length) return Integer.MAX_VALUE;
    if(dp[amount][idx] != -1) return dp[amount][idx];

    int minCoins = Integer.MAX_VALUE;
    for(int coin=0; amount >= coins[idx]*coin; coin++){
        int ans =  memo(amount - coins[idx] * coin, idx + 1, coins, dp);
        if(ans < Integer.MAX_VALUE) ans += coin;
        minCoins = Math.min(minCoins, ans);
    }

    return dp[amount][idx] = minCoins;
}

public int coinChange(int[] coins, int amount) {
    int[][] dp = new int[amount + 1][coins.length];
    for(int i=0; i<=amount; i++){
        for(int j=0; j<coins.length; j++){
            dp[i][j] = -1;
        }
    }

    int ans = memo(amount, 0, coins, dp);
    if(ans == Integer.MAX_VALUE) return -1;
    return ans;
}
```

0.5 x


App 3
{2,3,5}
target 9
States → Amount, Index
Calls → (2) Height → O(amount)
→ Infinite supply
If I am taking yes call for a coin, next level, same coin will be present
If I am taking no call, in the next level, next coin will explore the choices

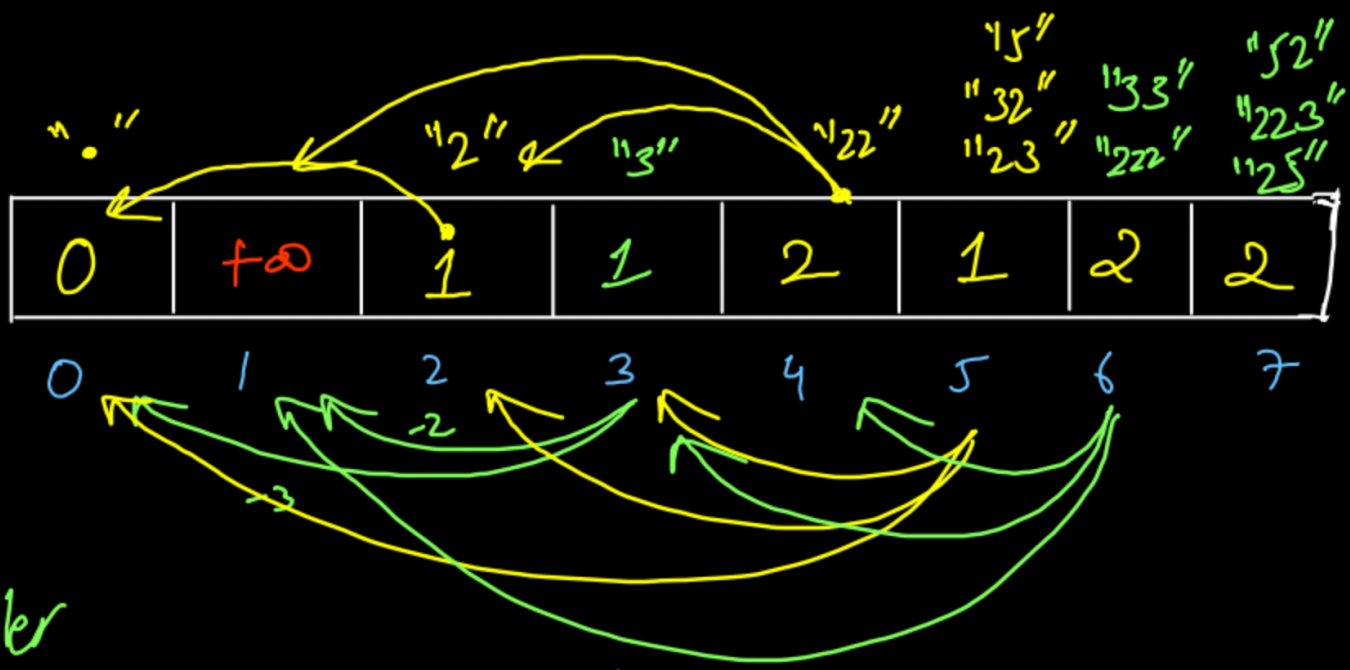
```java
// Time Complexity: O(Amount * Coins), Space Complexity: O(Amount * Coi
public int memo(int amount, int idx, int[] coins, int[][] dp){
    if(amount < 0) return Integer.MAX_VALUE;
    if(amount == 0) return 0;
    if(idx == coins.length) return Integer.MAX_VALUE;
    if(dp[amount][idx] != -1) return dp[amount][idx];

    int yes = memo(amount - coins[idx], idx, coins, dp);
    if(yes != Integer.MAX_VALUE) yes += 1;

    int no = memo(amount, idx + 1, coins, dp);

    return dp[amount][idx] = Math.min(yes, no);
}

public int coinChange(int[] coins, int amount) {
    int[][] dp = new int[amount + 1][coins.length];
    for(int i=0; i<=amount; i++){
        for(int j=0; j<coins.length; j++){
            dp[i][j] = -1;
        }
    }

    int ans = memo(amount, 0, coins, dp);
    if(ans == Integer.MAX_VALUE) return -1;
    return ans;
}
```

0.5 x

Tabulation

{2, 3, 5}

target = 7



1D array (amount) is a must

space opt
↓
Not possible

smaller problem

larger problem

dp[i] = min coins to form target i

```java
// Time -> O(Amount * Coin), Space -> O(Amount)
public int coinChange(int[] coins, int amount) {
    int[] dp = new int[amount + 1];
    Arrays.fill(dp, Integer.MAX_VALUE);
    dp[0] = 0;

    for(int i=1; i<=amount; i++){
        for(int coin: coins){
            if(i - coin >= 0 && dp[i - coin] != Integer.MAX_VALUE)
                dp[i] = Math.min(dp[i], dp[i - coin] + 1);
        }
    }

    return (dp[amount] == Integer.MAX_VALUE) ? -1 : dp[amount];
}
```

Coin change Permutations vs Combinations

coins = {2, 3, 5}

target = 10

Permutations

14

{2,2,2,2,2} {5,5}

{2,2,3,3} {2,3,3,2} {2,3,3,2}

{3,2,2,3} {3,2,3,2} {3,3,2,2}

{2,3,5} {2,5,3} {3,2,5}

{3,5,2} {5,2,3} {5,3,2}

Combinations

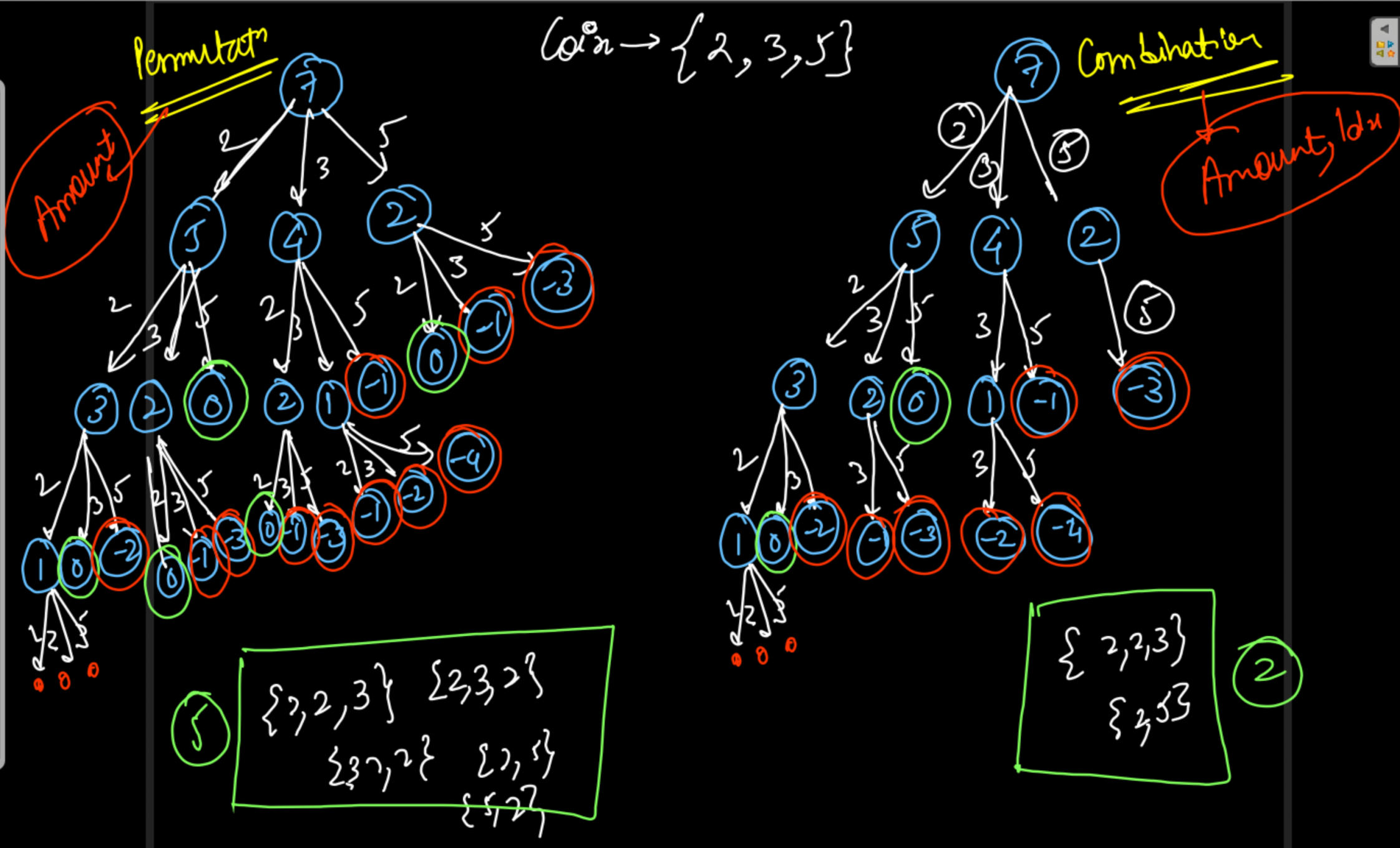
5

{2,2,2,2,2} {2,2,3,3}

{5,5} {2,3,5}

3! = 6

$\frac{4!}{2!2!} = \frac{24}{4} = 6$


Permutation
Coin → {2, 3, 5}
Amount
Combination
Amount, Idx
{2,2,3} {2,3,2}
{3,2,2} {2,5}
{5,2}
{2,2,3}
{3,5}
2

Dynamic Programming

lecture (10)

3 May 2022,
8 PM - 11 PM

Coin change

→ Permutations (377)

→ Combinations (518)

+

Knapsack Variations

→ 0/1 Knapsack

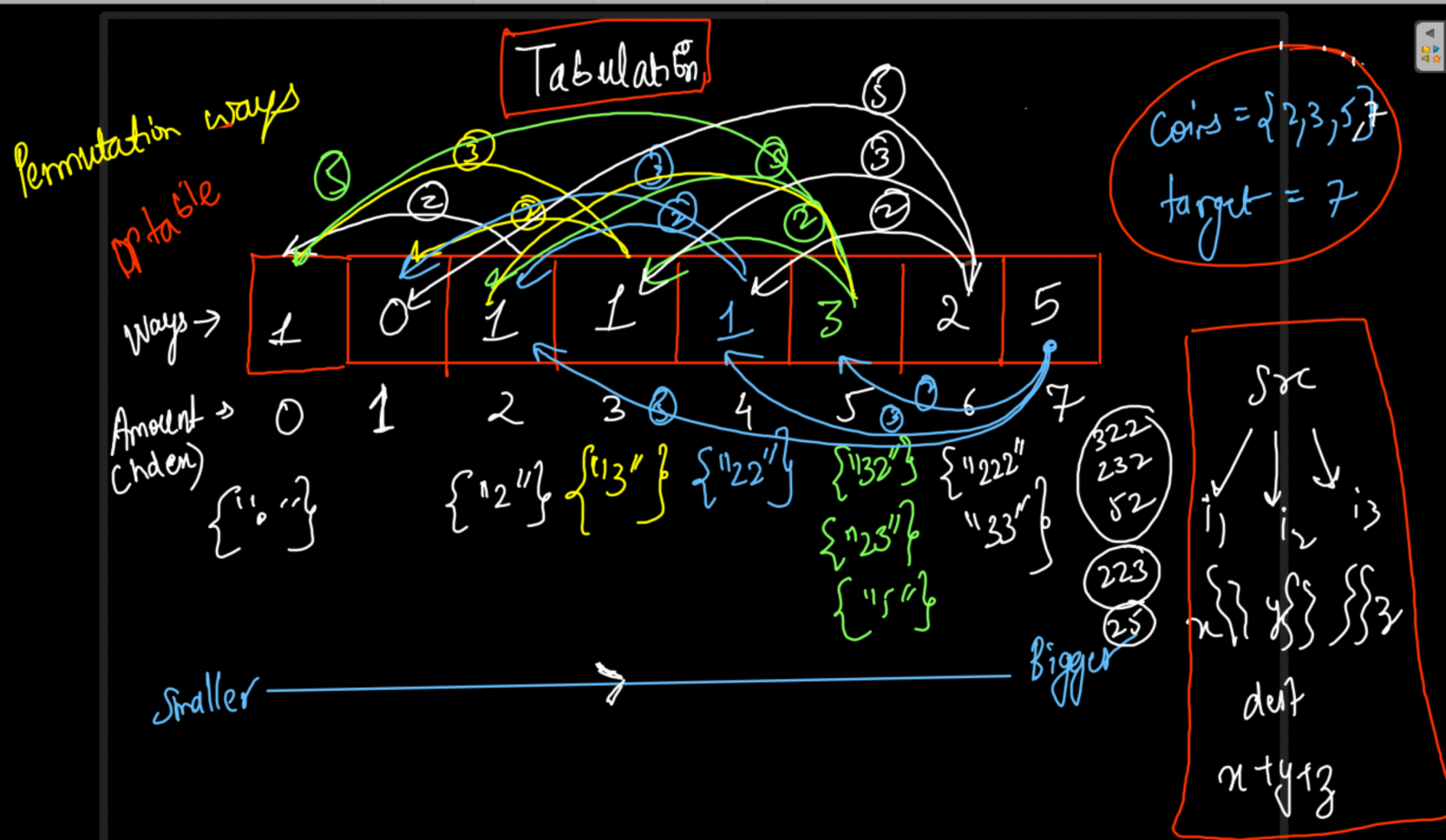
→ Unbounded Knapsack { GFG }

Permutation

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

        int perm = 0;
        for (int i = 0; i < coins.length; i++) {
                perm += memo(amount - coins[i], coins, dp);
        }

        return dp[amount] = perm;
    }

    public int combinationSum4(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, -1);

        return memo(amount, coins, dp);
    }
}
```

Combination

```java
// Time - O(Amount * Coins), Space - O(Amount * Coins): 2D DP + RCS
public int memo(int amount, int lastCoin, int[] coins, int[][] dp) {
    if (amount < 0) return 0;
    if (amount == 0) return 1;
    if (dp[amount][lastCoin] != -1) return dp[amount][lastCoin];

    int perm = 0;
    for (int coin = lastCoin; coin < coins.length; coin++) {
            perm += memo(amount - coins[coin], coin, coins, dp);
    }

    return dp[amount][lastCoin] = perm;
}

public int change(int amount, int[] coins) {
    int[][] dp = new int[amount + 1][coins.length + 1];
    for(int i=0; i<=amount; i++){
        for(int j=0; j<=coins.length; j++){
            dp[i][j] = -1;
        }
    }

    return memo(amount, 0, coins, dp);
}
```

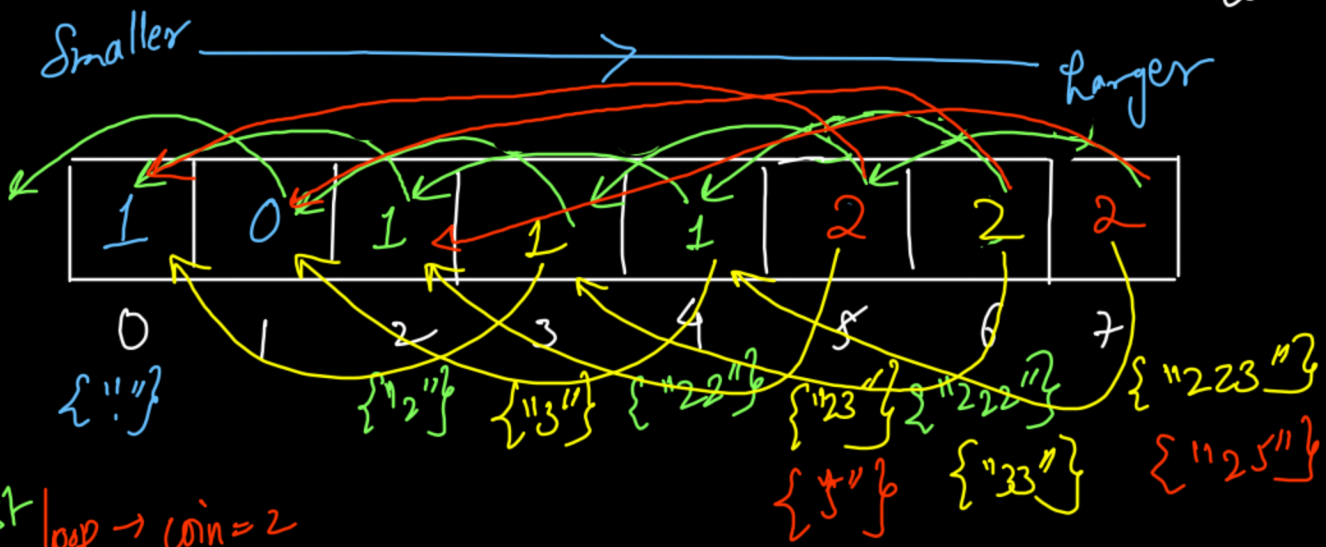

Tabulation
Permutation ways
DP table
Coins = {2,3,5}
target = 7
Ways →
1 0 1 1 1 3 2 5
Amount → 0 1 2 3 4 5 6 7
Choices
{"0"...}
{"2"}
{"3"}
{"22"}
{"32"}
{"25"}
{"5"}
{"222"
"33"}
322
232
223
25
Smaller
Bigger
Src
i1 i2 i3
x{{{ y{{{ {{{z
def
x+y+z

Tabulation

```
// Time - O(Amount * Coins), Space - O(Amount)
for(int i=1; i<= amount; i++){
    for(int coin: coins){
        if(i >= coin){
            dp[i] += dp[i - coin];
        }
    }
}
return dp[amount];
```

Combination (Tabulation)

target = 7
coins = {2, 3, 5}



1st loop → coin = 2

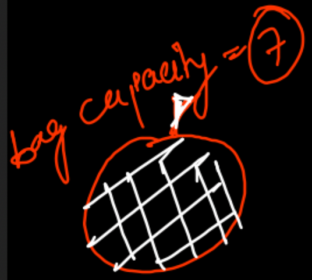
2nd loop → coin = 3

3rd loop: coin = 5

Tabulation

```
public int change(int amount, int[] coins) {
    int[] dp = new int[amount + 1];
    dp[0] = 1; // Ways to reach dest when src = dest is 1.

    // Time - O(Amount * Coins), Space - O(Amount)
    for(int coin: coins)
        for(int i=coin; i<= amount; i++)
                dp[i] += dp[i - coin];

    return dp[amount];
}
```

0-1 Knapsack



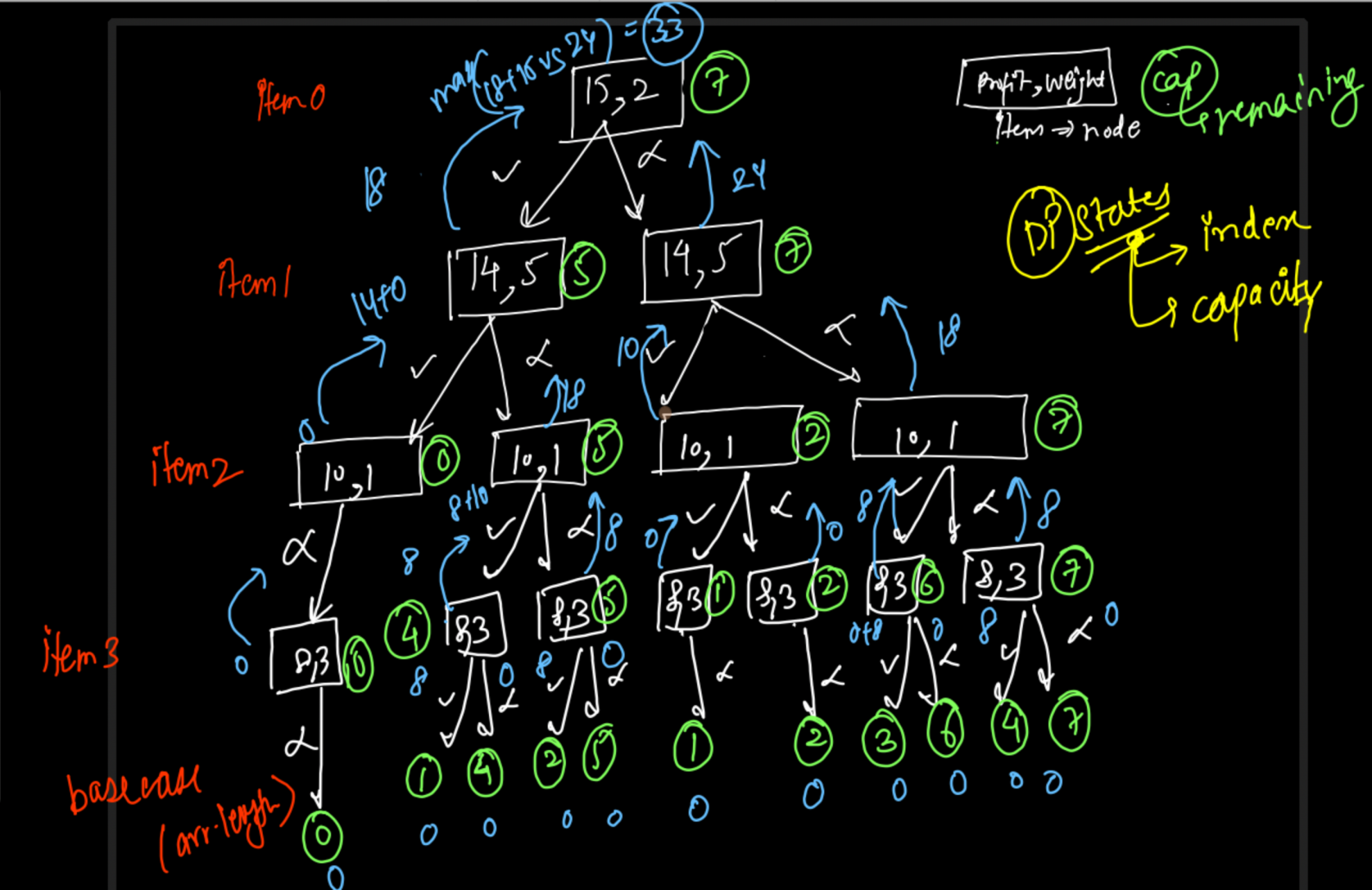
cost	15	14	10	8	20
weight	2	5	1	3	4
	i_0	i_1	i_2	i_3	i_4

Constraint

→ Maximum loot(profit) → sort in decreasing order of cost array (greedy ① fails)

→ items selected
weight sum ≤ capacity → sort in increasing order of weight array (greedy ② fails)

→ You can either the item in entirety or do not the item ⟷ (sort in $\frac{\text{cost}}{\text{weight}}$) (greedy ③ fails) because item is not divisible


item 0
item 1
item 2
item 3
base case
(arr.length)
max (18+15 vs 24) = 33
15, 2
14, 5
14, 5
10, 1
10, 1
10, 1
10, 1
8, 3
8, 3
8, 3
8, 3
8, 3
8, 3
8, 3
8, 3
Profit, Weight
item → node
cap
remaining
DP
states
index
capacity

```java
static int memo(int cap, int item, int[] wt, int[] cost, int[][] dp){
    if(item == cost.length) return 0; // No Item No Profit
    if(dp[cap][item] != -1) return dp[cap][item];

    int yes = (cap >= wt[item]) ?
        memo(cap - wt[item], item+1, wt, cost, dp) + cost[item]: -1;
    int no = memo(cap, item+1, wt, cost, dp);

    return dp[cap][item] = Math.max(yes, no);
}

static int knapSack(int cap, int wt[], int cost[], int n)
{
    int[][] dp = new int[cap + 1][cost.length];
    for(int i=0; i<=cap; i++){
        for(int j=0; j<cost.length; j++){
            dp[i][j] = -1;
        }
    }

    return memo(cap, 0, wt, cost, dp);
}
```

Time

$O(cap * item)$

Space

$O(cap * item)$

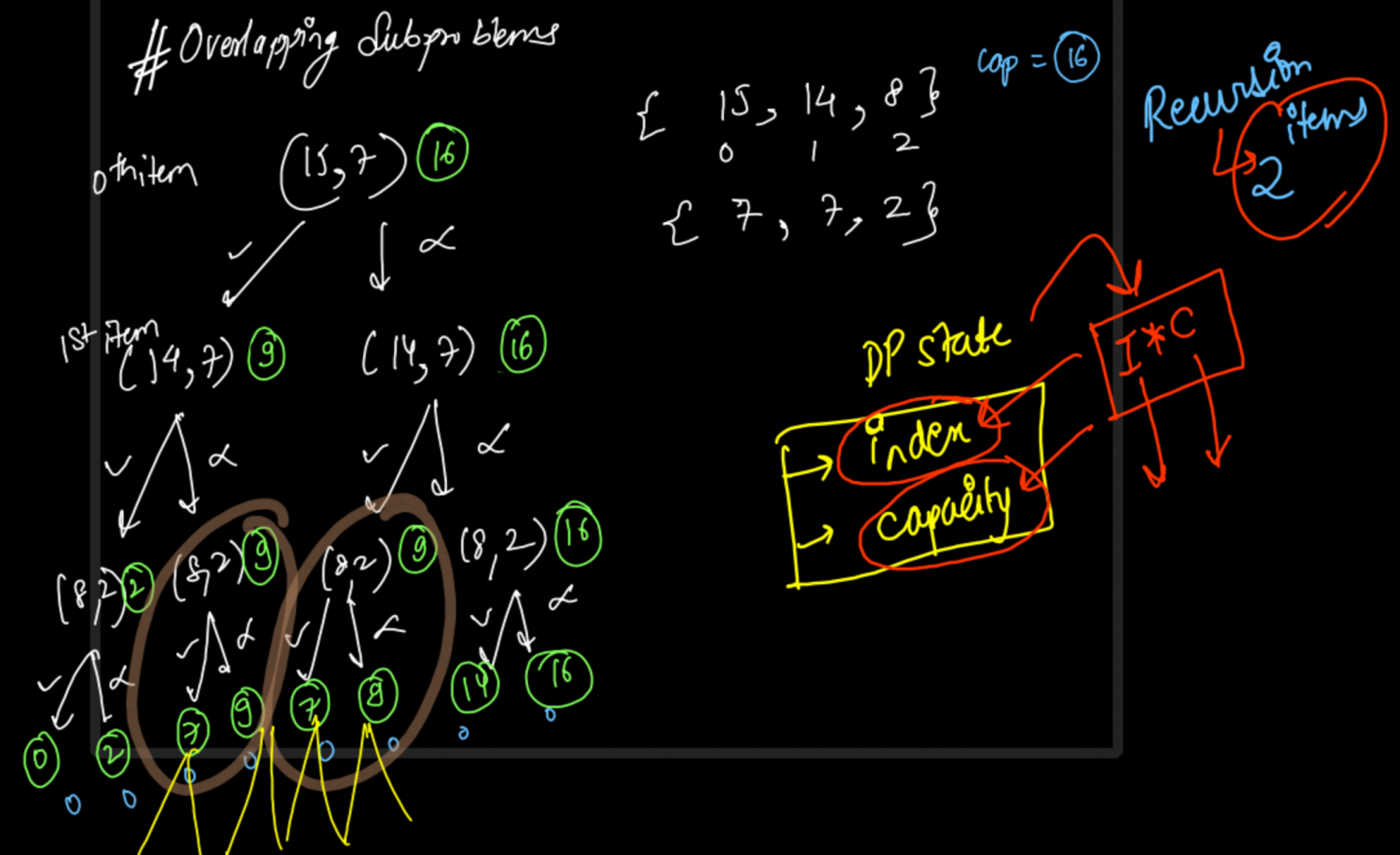
2D DP

K·C·S $\Rightarrow O(items)$

Space optimization ? yes

↳ Tabulation

↳ 1D DP ↳ $O(cap)$


Overlapping Subproblems
{ 15, 14, 8 }
0 1 2
{ 7, 7, 2 }
cap = 16
Recursion
items
2
0th item
(15,7) 16
1st item
(14,7) 9
(14,7) 16
(8,2) 2
(8,2) 9
(8,2) 9
(8,2) 16
0
2
7
9
7
8
14
16
DP state
index
capacity
I*C

Tabulation
DP Table

Follow up: Space Optimization?

Items = 4, capacity = 7

items (col)

rows (cap)	No items	(15, 2)	(14, 5)	(10, 1)	(8, 3)
0 (smaller)	0	0	0	0	0
1	0	0 vs -1 → 0	0 vs -1 → 0	0 vs 10+0 → 10	10 vs -1 → 10
2	0	0 vs 15+0 → 15	15 vs -1 → 15	15 vs 10+0 → 15	15 vs -1 → 15
3	0	0 vs 15+0 → 15	15 vs -1 → 15	15 vs 10+15 → 25	25 vs 8+0 → 25
4	0	0 vs 15+0 → 15	15 vs -1 → 15	15 vs 10+15 → 25	25 vs 8+10 → 25
5	0	0 vs 15+0 → 15	15 vs 14+0 → 15	15 vs 10+15 → 25	25 vs 8+15 → 25
6	0	0 vs 15+0 → 15	15 vs 14+0 → 15	15 vs 10+15 → 25	25 vs 8+25 → 33
7 (larger)	0	0 vs 15+0 → 15	15 vs 14+15 → 29	29 vs 10+15 → 29	29 vs 8+25 → 33

```java
static int knapSack(int caps, int wt[], int cost[], int n)
{
    int[] dp = new int[caps + 1];

    for(int item=1; item<=cost.length; item++){
        int[] newDp = new int[caps + 1];

        for(int cap=1; cap<=caps; cap++){

            int no = dp[cap];
            int yes = (cap >= wt[item - 1])
                    ? cost[item - 1] + dp[cap - wt[item - 1]]
                    : -1;

            newDp[cap] = Math.max(yes, no);
        }

        dp = newDp;
    }

    return dp[caps];
}
```

Space opt

TC → O(I * C)

SC → O(cap)

1D

```
int[][] dp = new int[caps + 1][cost.length + 1];

for(int item=1; item<=cost.length; item++){
    for(int cap=1; cap<=caps; cap++){

        int no = dp[cap][item - 1];
        int yes = (cap >= wt[item - 1])
                ? cost[item - 1] + dp[cap - wt[item - 1]][item - 1]
                : -1;

        dp[cap][item] = Math.max(yes, no);
    }
}

return dp[caps][cost.length];
```

TC ⇒ O(Items * Cap)

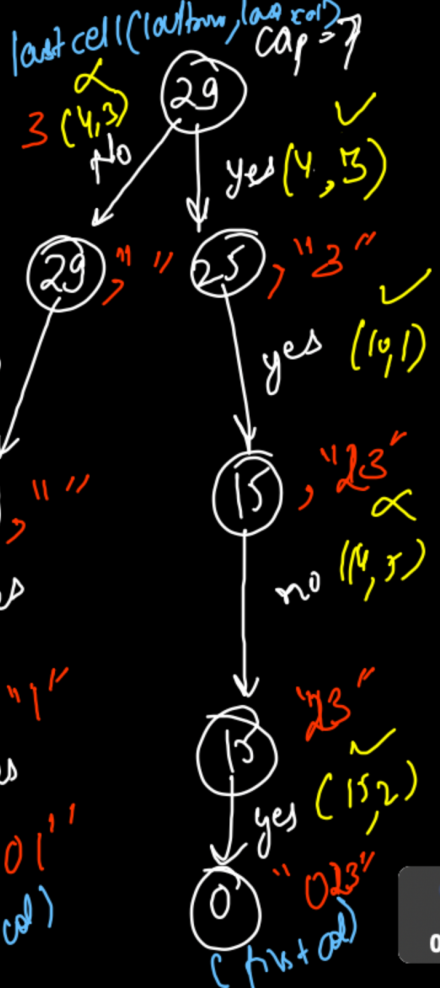
SC ⇒ O(Items * Cap) {2D}

Print All ways to select items in Knapsack such that maximum profit is achieved

	No items	(15,2)	(14,5)	(10,1)	(4,3)
Smaller 0	0	0	0	0	0
1	0	0 vs -1 0	0 vs -1 0	0 vs 10+0 10	10 vs -1 10
2	0	0 vs 15+0 15	15 vs -1 15	15 vs 10+0 15	15 vs -1 15
3	0	0 vs 15+0 15	15 vs -1 15	15 vs 10+15 25	25 vs 4+0 25
4	0	0 vs 15+0 15	15 vs -1 15	15 vs 10+15 25	25 vs 4+10 25
5	0	0 vs 15+0 15	15 vs 14+0 15	15 vs 10+15 25	25 vs 4+15 25
6	0	0 vs 15+0 15	15 vs 14+0 15	15 vs 10+15 25	25 vs 4+25 29
larger 7	0	0 vs 15+0 15	15 vs 14+15 29	29 vs 10+15 25	29 vs 4+25 29

Items (wt)

For Backtracking
src → last cell
dest → first col



BFS

```
Queue<Pair> q = new ArrayDeque<>();
q.add(new Pair(caps, cost.length, ""));

while(q.size() > 0){
    Pair top = q.remove();
    int row = top.row;
    int col = top.col;
    String psf = top.psf;

    if(top.col == 0){
        System.out.println(top.psf);
        continue;
    }

    // If Item Can be Included and It gives Maximum Profit, then explore this
    if(row >= wt[col - 1] &&
        dp[row][col] == cost[col - 1] + dp[row - wt[col - 1]][col - 1]){
        q.add(new Pair(row - wt[col - 1], col - 1, (col - 1) + " " + psf));
    }

    // If no call gives maximum profit, then only explore this edge
    if(dp[row][col] == dp[row][col - 1]){
        q.add(new Pair(row, col - 1, psf));
    }
}
```

DFS

TC ⇒ O(exponential) in worst case
O(V+E) BFS in avg case

SC ⇒ O(Cap * N) DP + O(I-items * Cap) queue

Dynamic Programming

lecture (12)

Saturday, 9 AM – 12 PM, 7 May

Knapsack
- → Unbounded Knapsack
- → Rod Cutting
- → Buy & Sell Stocks

Unbounded Knapsack

bag cap = (7)

Items

cost	15	14	10	8	20
weight	2	5	1	3	4
	i_0	i_1	i_2	i_3	i_4

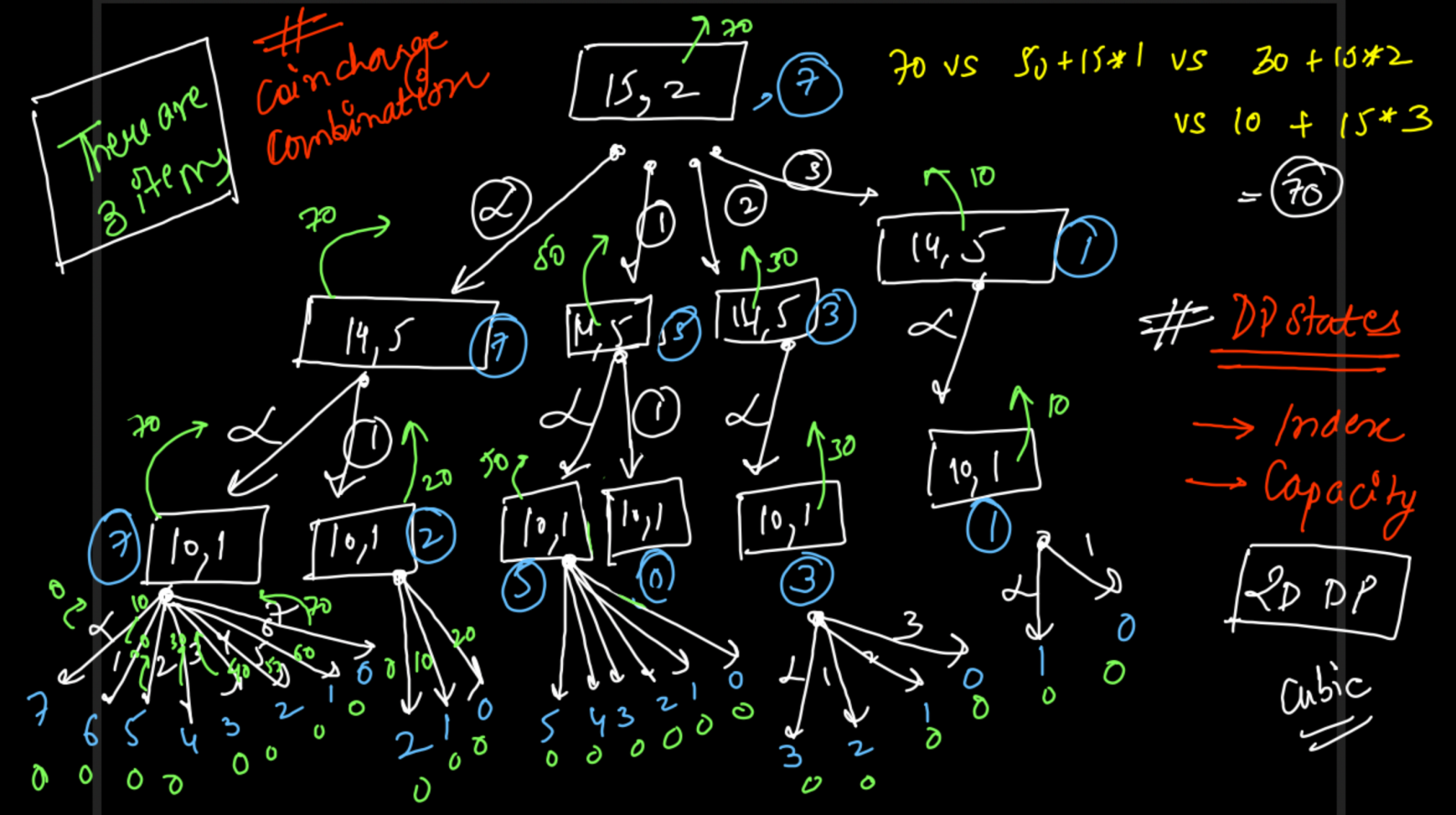
Constraint

→ maximum loot(profit)

→ items selected
weight sum ≤ capacity

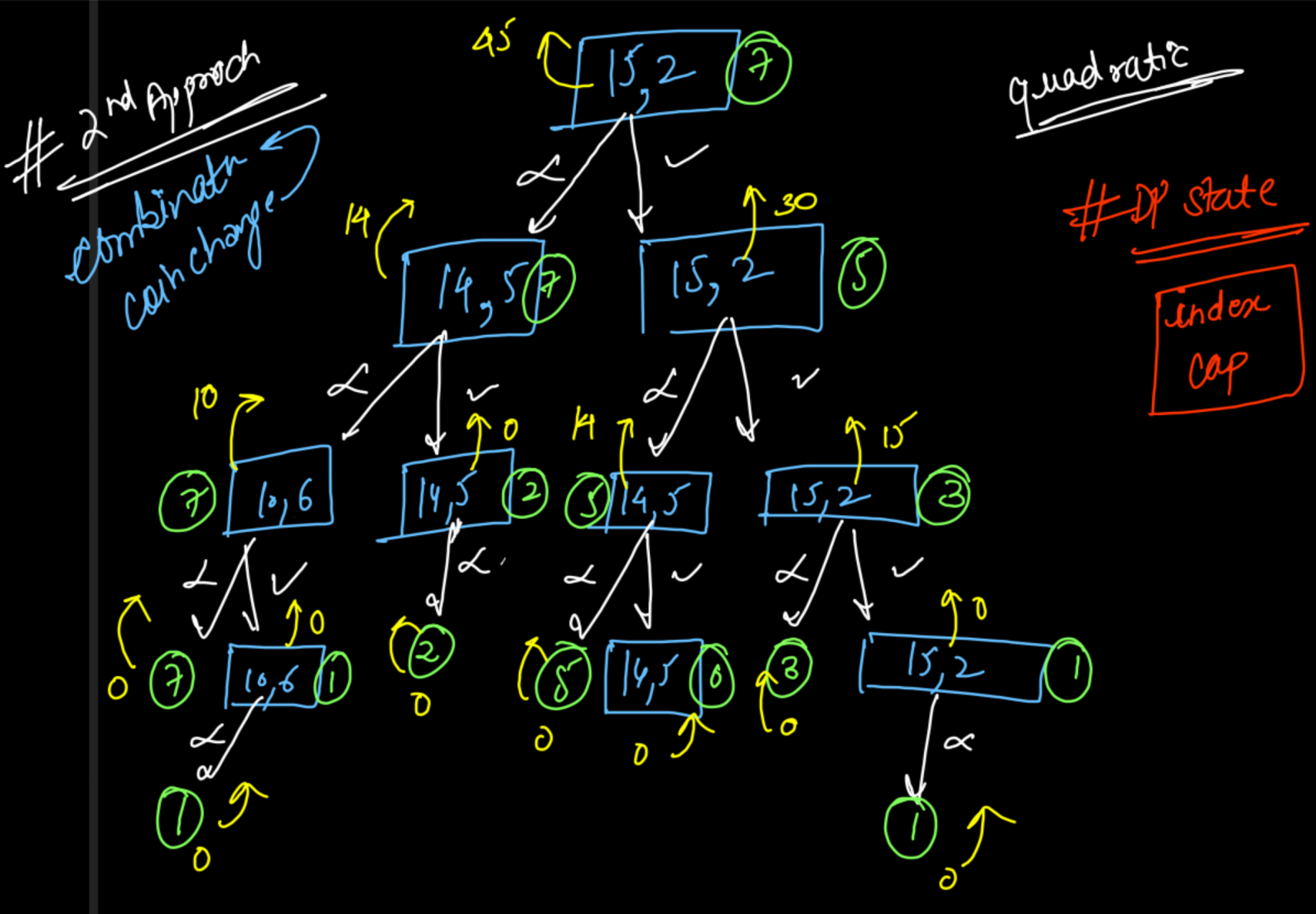
→ Items can not be divided

→ You have infinite supply
of each items


There are 3 items
Coin change 2 Combination
15, 2
70
7
70 vs 50 + 15*1 vs 30 + 10*2
vs 10 + 15*3
= 70
14, 5
10, 1
DP states
→ Index
→ Capacity
2D DP
Cubic

```java
static int memo(int index, int cap, int cost[], int wt[], int N, int[][] dp){
    if(index == N || cap == 0) return 0;
    if(dp[index][cap] != -1) return dp[index][cap];

    int ans = -1;
    for(int freq=0; cap >= freq * wt[index]; freq++){
        int temp = memo(index + 1, cap - freq * wt[index], cost, wt, N, dp)
                   + freq * cost[index];
        ans = Math.max(ans, temp);
    }

    return dp[index][cap] = ans;
}
```

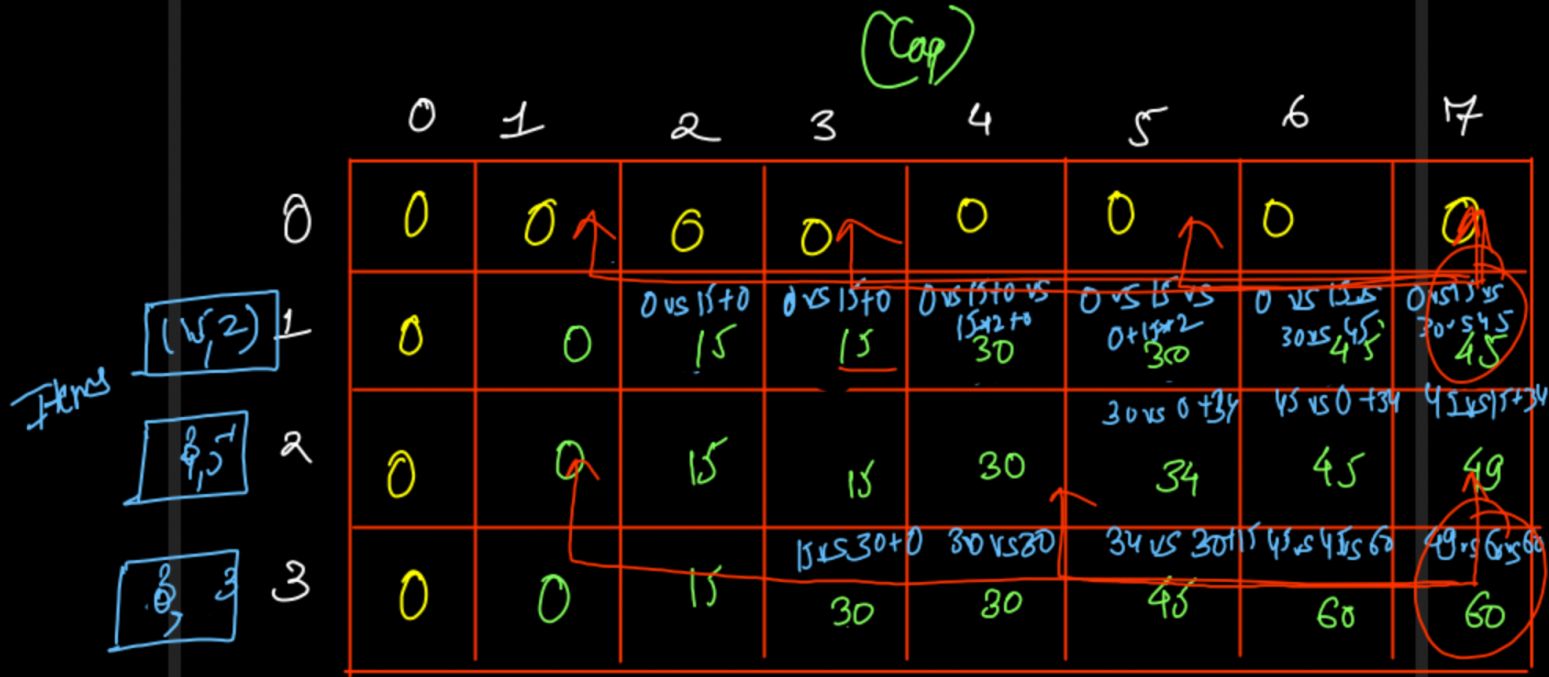
```java
static int knapSack(int N, int cap, int cost[], int wt[])
{
    int[][] dp = new int[N + 1][cap + 1];
    for(int i=0; i<=N; i++){
        for(int j=0; j<=cap; j++){
            dp[i][j] = -1;
        }
    }

    return memo(0, cap, cost, wt, N, dp);
}
```

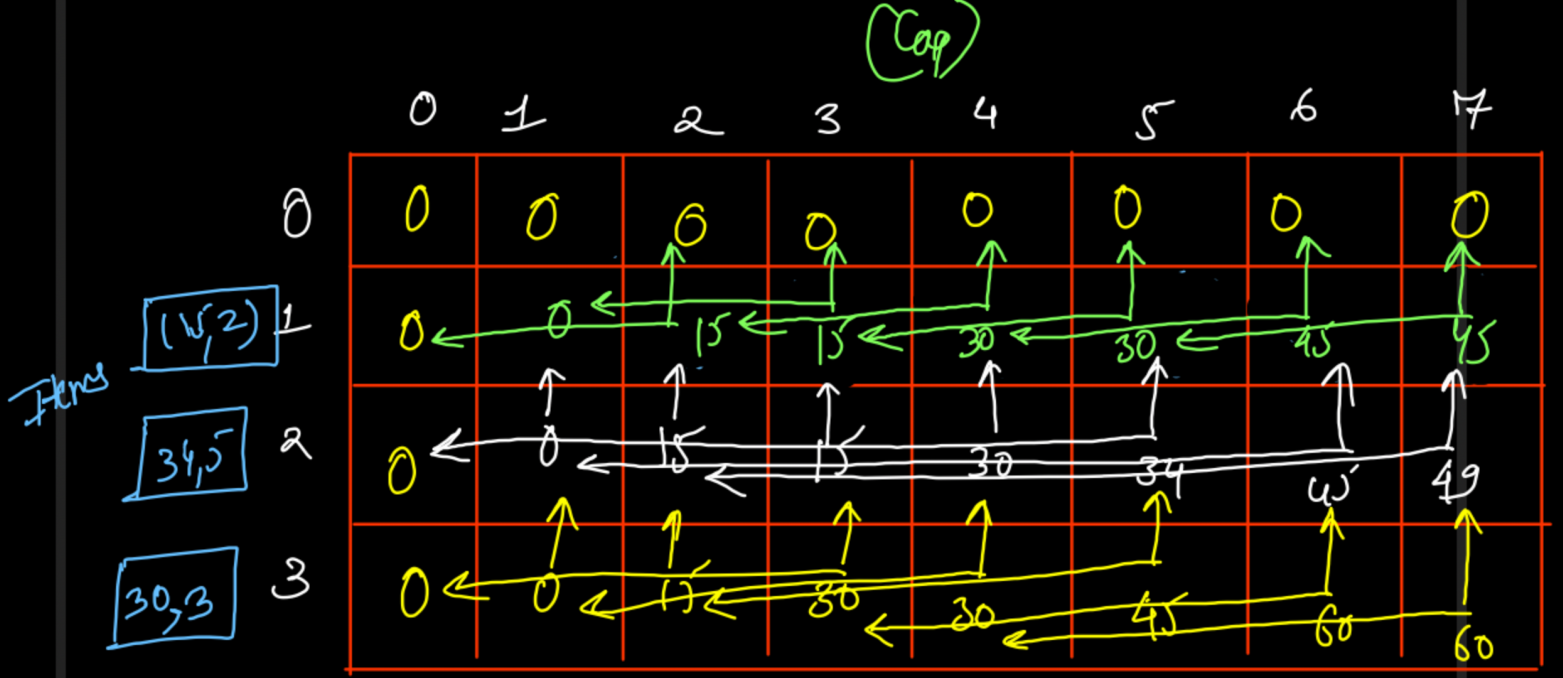

2nd Approach
combination
coin change
quadratic
DP state
index
cap
15, 2
14, 5
15, 2
10, 6
14, 5
14, 5
15, 2
10, 6
15, 2
14, 5
15, 2

```java
static int memo(int index, int cap, int cost[], int wt[], int N, int[][] dp){
    if(index == N || cap == 0) return 0;
    if(dp[index][cap] != -1) return dp[index][cap];

    int no = memo(index + 1, cap, cost, wt, N, dp);
    int yes = (cap >= wt[index])
        ? cost[index] + memo(index, cap - wt[index], cost, wt, N, dp) : -1;

    return dp[index][cap] = Math.max(yes, no);
}

static int knapSack(int N, int cap, int cost[], int wt[])
{
    int[][] dp = new int[N + 1][cap + 1];
    for(int i=0; i<=N; i++){
        for(int j=0; j<=cap; j++){
            dp[i][j] = -1;
        }
    }

    return memo(0, cap, cost, wt, N, dp);
}
```

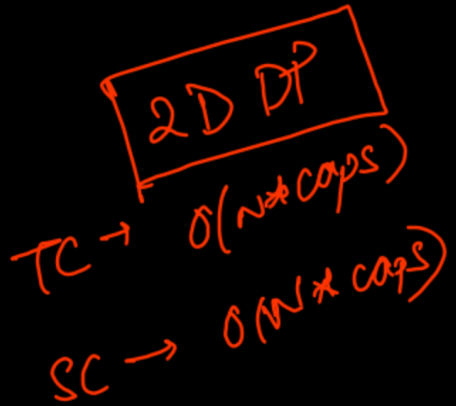
TC
$O(N * cap)$

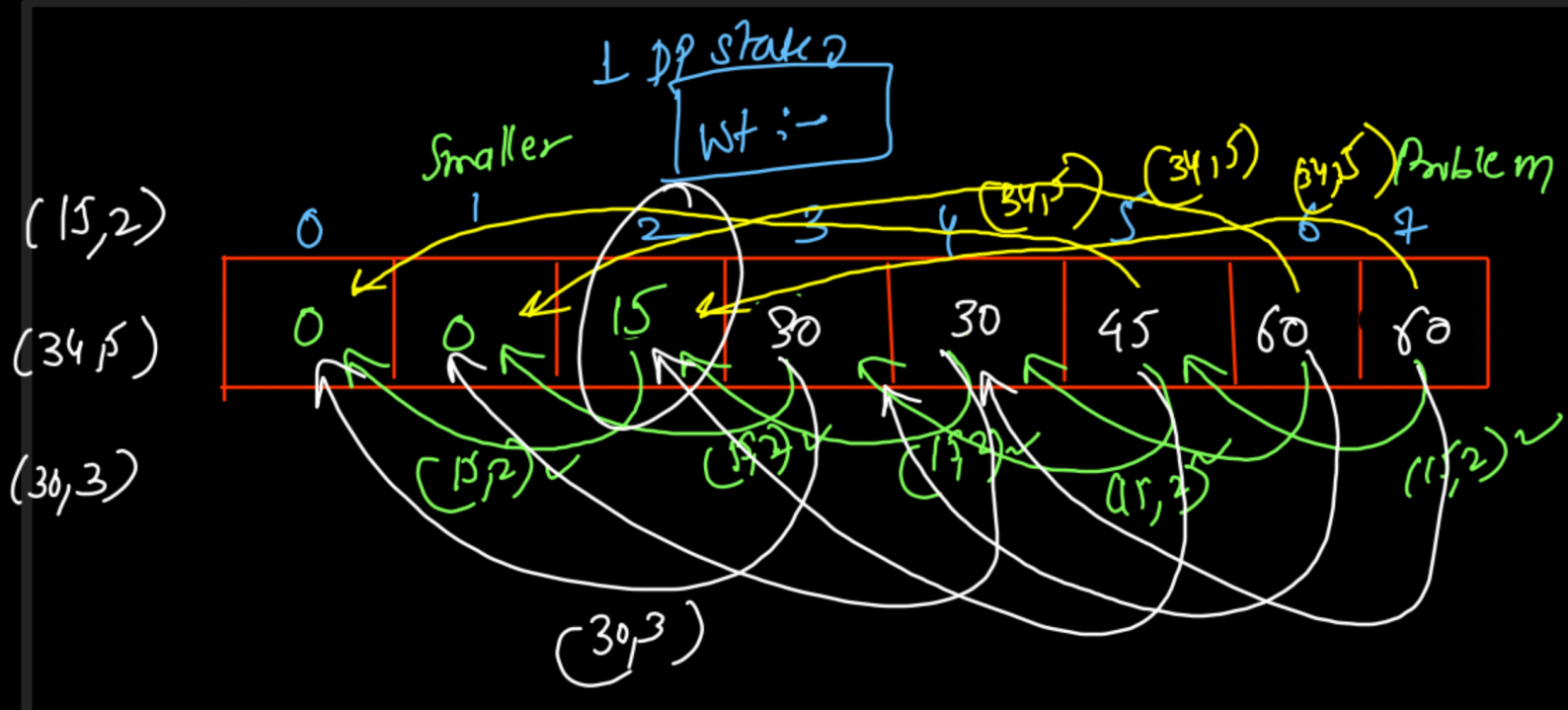
SC
2D DP $\rightarrow O(N * cap)$


(Cap)
0 1 2 3 4 5 6 7
Items
(1,2) 1
(4,5) 2
(3,4) 3
0 0 0 0 0 0 0 0
0 0 0 vs 15+0 15 0 vs 15+0 15 0 vs 15+0 vs 15+2+0 30 0 vs 15+vs 0+15+2 30 0 vs 15+vs 30+45 45 0 vs 15+vs 30+45 45
0 0 15 15 30 30 vs 0+34 34 45 vs 0+34 45 49 vs 15+34 49
0 0 15 15 vs 30+0 30 30 vs 30 30 34 vs 30 vs 15 45 45 vs 45 vs 60 60 49 vs 60 vs 60 60
0 0 15 30 30 45 60 60


(Cap)
0 1 2 3 4 5 6 7
Items
(W,2)
3,4,5
3,0,3
0
1
2
3

```java
static int knapSack(int N, int caps, int cost[], int wt[])
{
    int[][] dp = new int[N + 1][caps + 1];

    for(int item=1; item<=N; item++){
        for(int cap=1; cap<=caps; cap++){
            int no = dp[item - 1][cap];
            int yes = (cap >= wt[item - 1])
                ? dp[item][cap - wt[item - 1]] + cost[item - 1]
                : -1;

            dp[item][cap] = Math.max(no, yes);
        }
    }

    return dp[N][caps];
}
```




1 DP state 2
Wt :-
Smaller
Problem
(15,2)
(34,5)
(30,3)
0
1
2
3
4
5
6
7
0
0
15
30
30
45
60
60
(34,5)
(34,5)
(34,5)
(15,2)
(15,2)
(15,2)
(15,2)
(30,3)

```java
static int knapSack(int N, int caps, int cost[], int wt[])
{
    int[] dp = new int[caps + 1];

    for(int item=1; item<=N; item++){
        for(int cap=1; cap<=caps; cap++){
            int no = dp[cap];
            int yes = (cap >= wt[item - 1])
                ? dp[cap - wt[item - 1]] + cost[item - 1]
                : -1;

            dp[cap] = Math.max(no, yes);
        }
    }

    return dp[caps];
}
```

1D DP

TC → O(Cap * N)

SC → O(Cap)

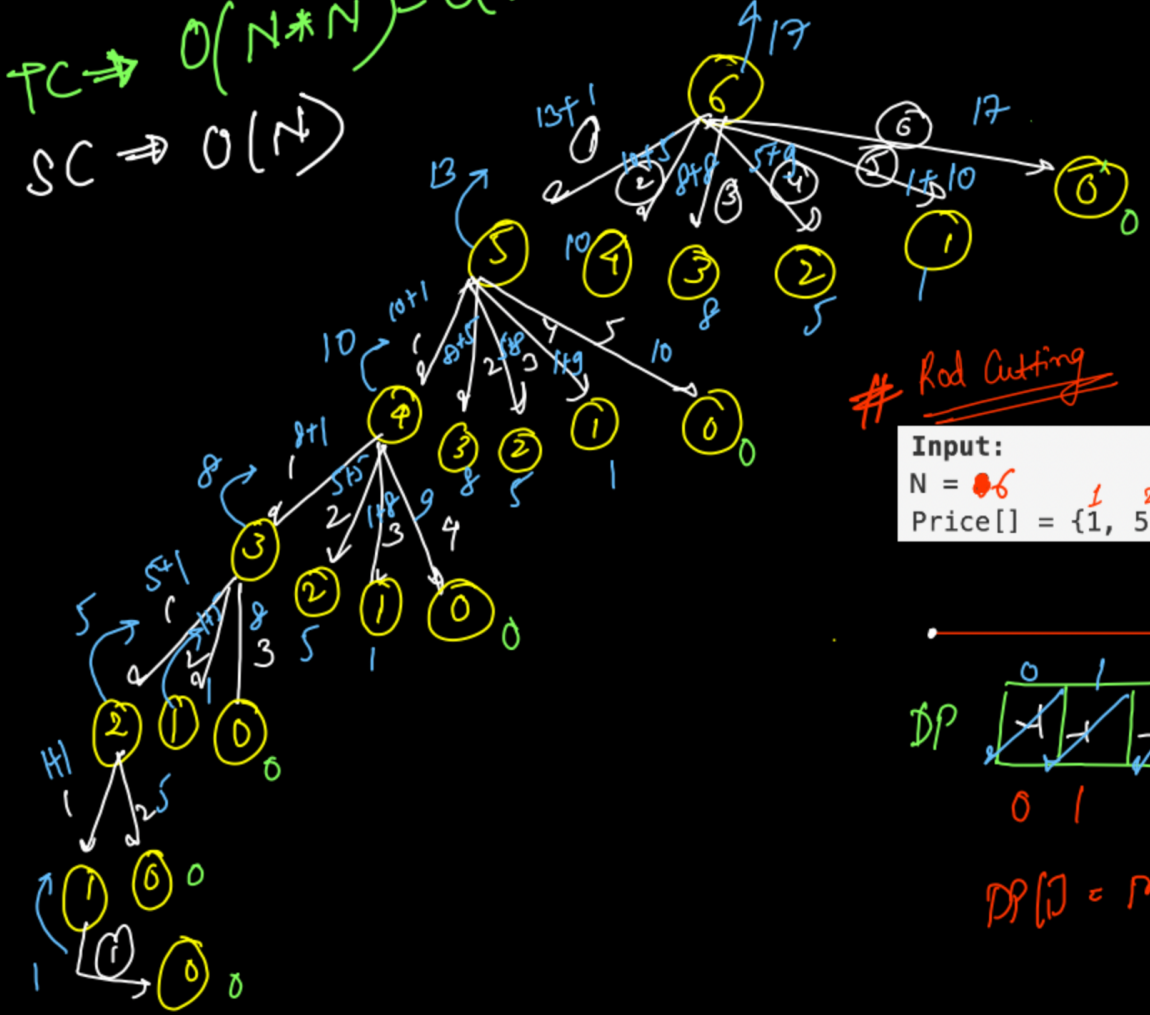
Infinite Supply → yes → same row

Limited Supply → yes → previous row

No call infinite row row

TC ⇒ $O(N*N) = O(N^2)$

SC ⇒ $O(N)$



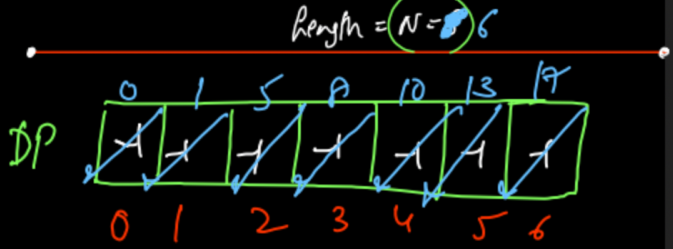
Rod Cutting

N → length

Input:

N = 6

Price[] = {1, 5, 8, 9, 10, 17, ...}



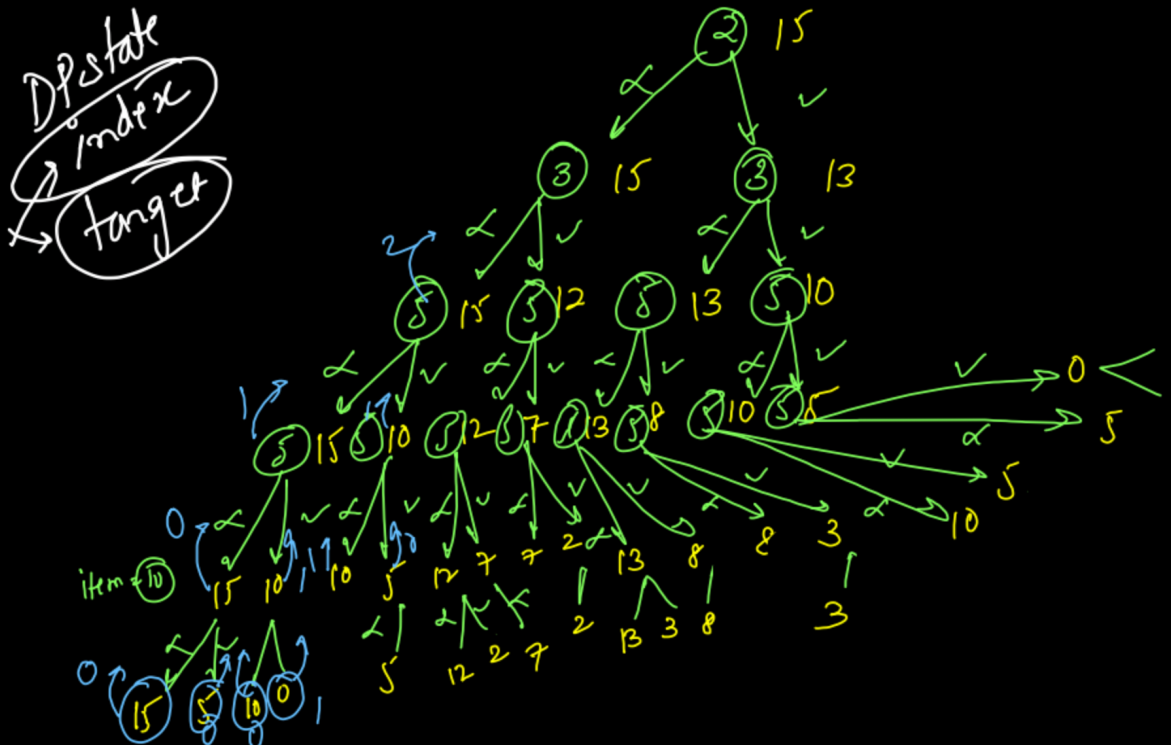
DP[i] = Max profit of rod length

Variation of # Climb Stairs with minimum cost

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

        int ans = 0;
        for(int cut=1; cut<=n; cut++){
            ans = Math.max(ans, price[cut - 1] + memo(n - cut, price, dp));
        }

        return dp[n] = ans;
    }
    public int cutRod(int price[], int n) {
        int[] dp = new int[n + 1];
        Arrays.fill(dp, -1);
        return memo(n, price, dp);
    }
```

Target sum subset (variation of 0-1 knapsack)

{2, 3, 5, 5, 10} target = 15



Dynamic Programming - Lecture 13

Sunday, 8 May, 9 AM - 12 PM

Buy & Sell Stocks

Variations

- Fibonacci
- House Robber
- Knapsack

(121) Buy & Sell Stocks - 1 Transaction

You are given an array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

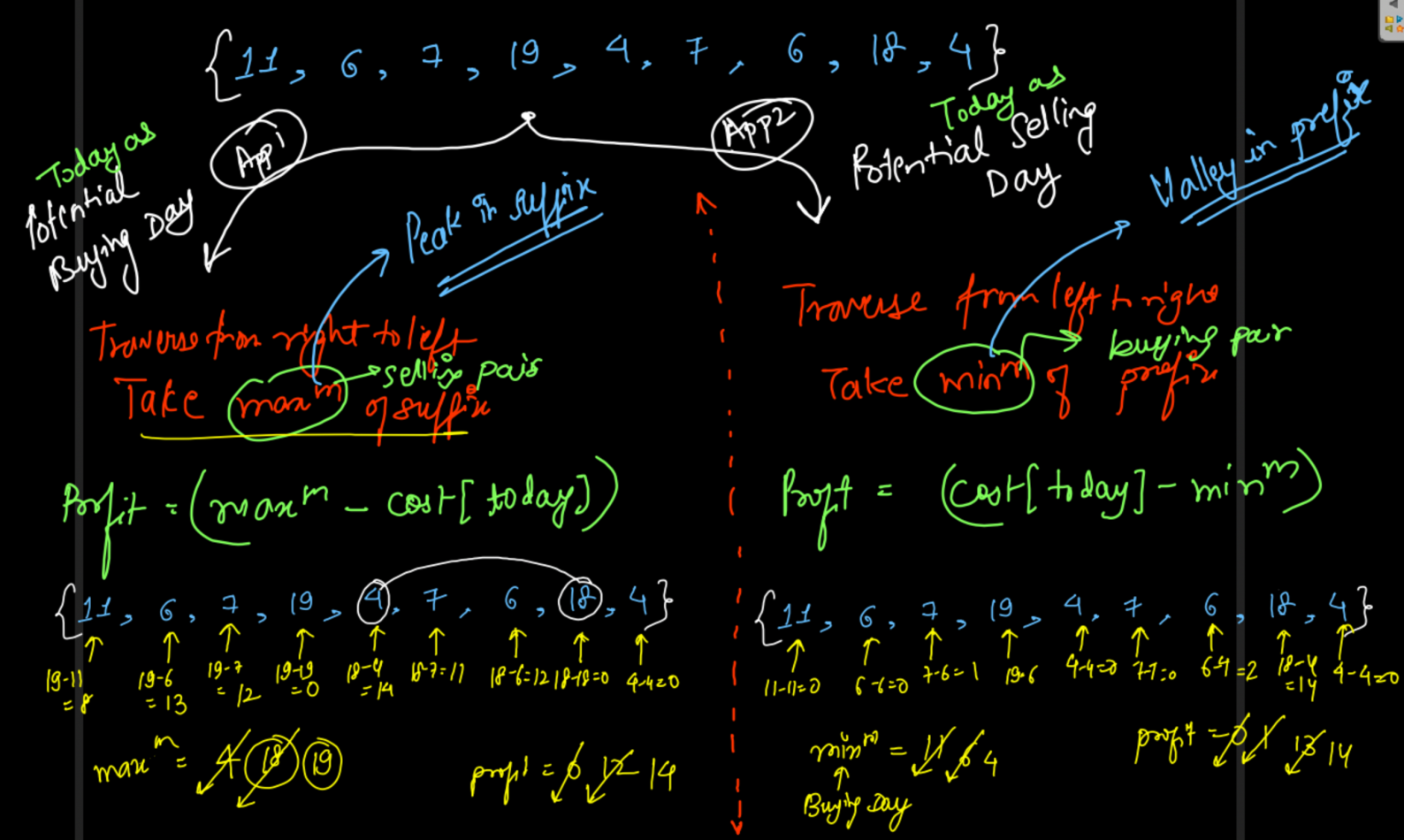
Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

Constraint

Buy → Sell | 1 transaction

BBSS ∝

{11, 6, 7, 19, 4, 1, 6, 18, 4}


{11, 6, 7, 19, 4, 7, 6, 18, 4}
Today as Potential Buying Day
App1
App2
Today as Potential Selling Day
Peak in suffix
Valley in prefix
Traverse from right to left
Take max^m of suffix
selling pair
Traverse from left to right
Take min^m of prefix
buying pair
Profit = (max^m - cost[today])
Profit = (cost[today] - min^m)
19-11 = 8
19-6 = 13
19-7 = 12
19-19 = 0
18-4 = 14
18-7 = 11
18-6 = 12
18-18 = 0
4-4 = 0
11-11 = 0
6-6 = 0
7-6 = 1
19-6
4-4 = 0
7-7 = 0
6-4 = 2
18-4 = 14
4-4 = 0
max^m = 19
profit = 14
min^m = 4
Buying day
profit = 14

```java
class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;

        int selling = prices[n - 1], profit = 0;

        for(int i=n-1; i>=0; i--){
            selling = Math.max(selling, prices[i]);

            int buying = prices[i]; // Today - Potential Buying Day

            profit = Math.max(profit, selling - buying);
        }

        return profit;
    }
}
```

App 1

O(N) Time
O(1) Space

```java
public int maxProfit(int[] prices) {
    int n = prices.length;

    int buying = prices[0], profit = 0;

    for(int i=0; i<n; i++){
        buying = Math.min(buying, prices[i]);

        int selling = prices[i]; // Today - Potential Selling Day

        profit = Math.max(profit, selling - buying);
    }

    return profit;
}
```

App 2

122. Best Time to Buy and Sell Stock II

Infinite Transaction

#BBS is not possible
#BSBS is possible

(top) buying day = 11 6 7 19 4 1 6 18 4 12 2

profit = 0+1+12+5+12+8

11	6	7	19	4	1	6	18	4	12	2

```java
class Solution {
    public int maxProfit(int[] prices) {
        Stack<Integer> stk = new Stack<>();
        int profit = 0;

        for(int i=0; i<prices.length; i++){
            if(stk.size() > 0 && stk.peek() < prices[i]){
                profit += prices[i] - stk.pop();
            }

            stk.push(prices[i]);
        }

        return profit;
    }
}
```

O(N) time
O(N) space

Greedy Approach

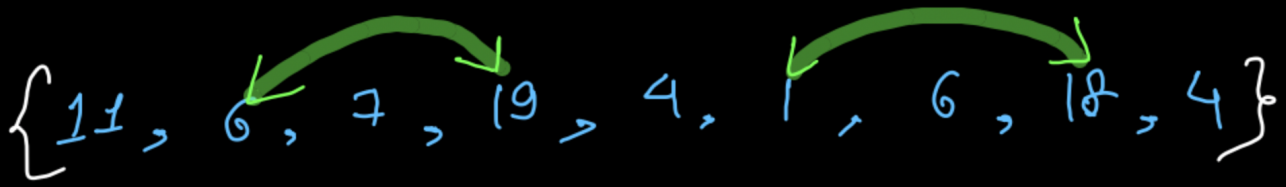
```java
class Solution {
    public int maxProfit(int[] prices) {
        int buying = prices[0];
        int profit = 0;

        for(int i=0; i<prices.length; i++){
            if(buying < prices[i]){
                profit += prices[i] - buying;
            }

            buying = prices[i];
        }

        return profit;
    }
}
```

O(N) time
O(1) space

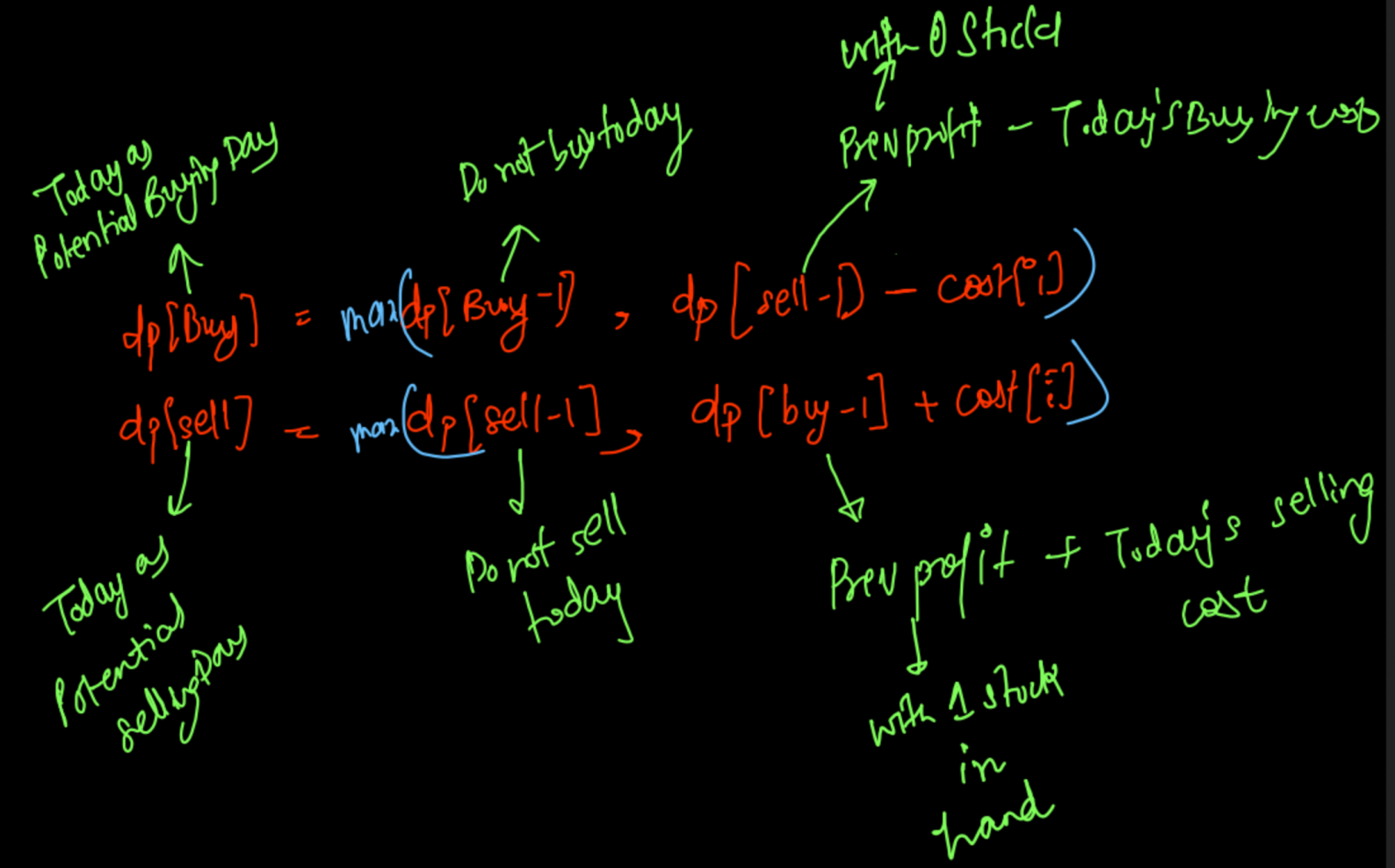
DP

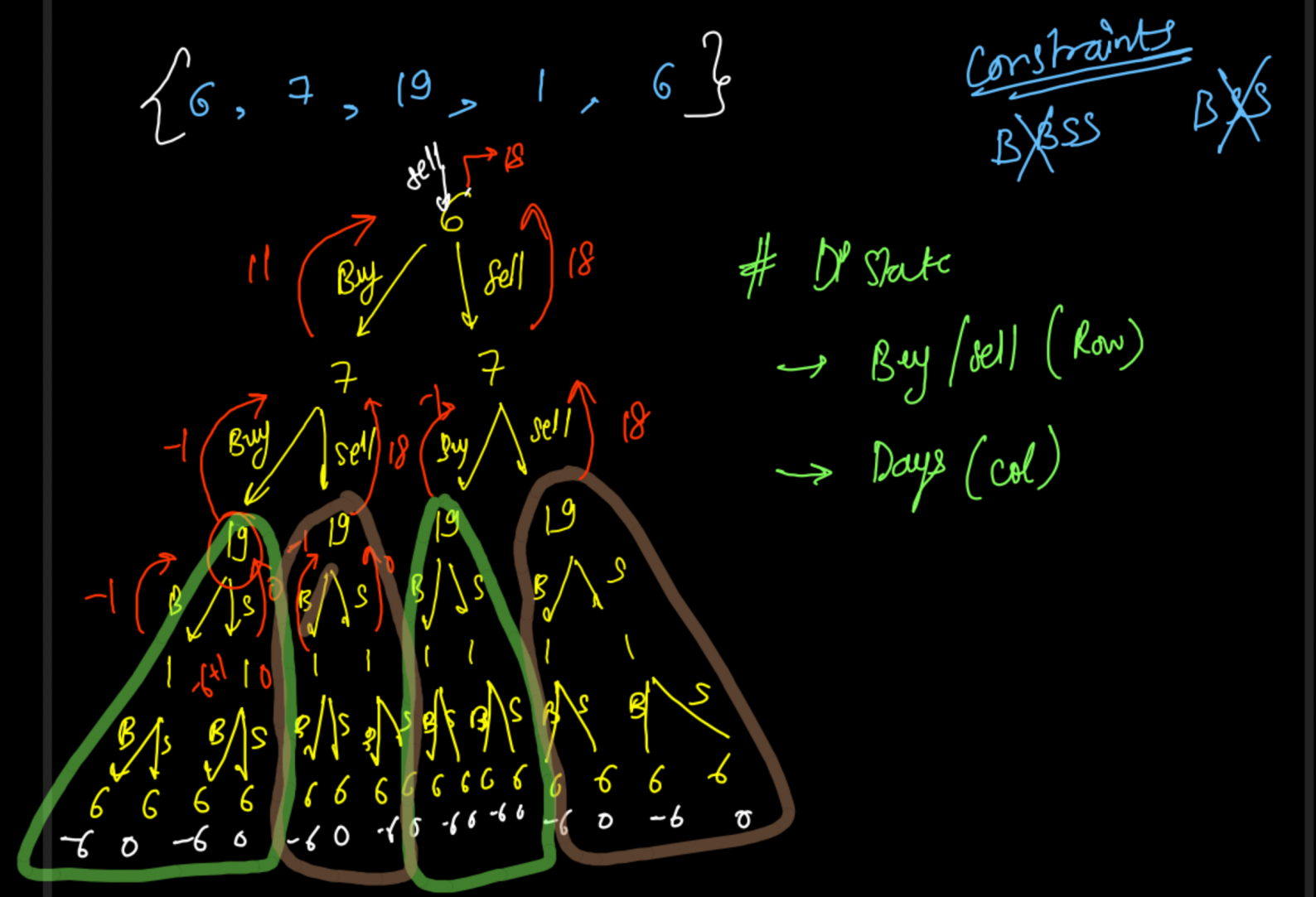
Buy & Sell Stock - Infinite Transaction

- Transaction fee
- Cooldown



(1 Extra Stock) Buy	-11	-11 vs -6 → -6	-7 vs -6 → -6	-6 vs -18 → -6	-6 vs -4+13 → 9	9 vs 13-1 → 12	13-6 vs 12 → 12	18-18 vs 12 → 12	12 vs 30-4 → 26
(Count Stock) Sell	0	0 vs -11+6 → 0	1 vs 0 → 1	13 vs 1 → 13	-2 vs 13 → 13	13 vs 9+1 → 13	13 vs 12+6 → 18	18 vs 12+18 → 30	30 vs 12+14 → 30


Today as Potential Buy today
Do not buy today
With 0 Stock
Prev profit - Today's Buy by cost
dp[Buy] = max(dp[Buy-1], dp[sell-1] - cost[i])
dp[sell] = max(dp[sell-1], dp[buy-1] + cost[i])
Today as Potential sell today
Do not sell today
Prev profit + Today's selling cost
With 1 stock in hand


{6, 7, 19, 1, 6}
Constraints
B S S
B S
DP State
→ Buy / Sell (Row)
→ Days (col)
sell
18
6
11
Buy
Sell
18
7
7
-1
Buy
sell
18
Buy
sell
18
19
19
19
19
-1
B
S
0
1
1
1
1
-6+1
1
0
B
S
B
S
B
S
B
S
B
S
B
S
B
S
6
6
6
6
6
6
6
6
6
6
6
6
-6
0
-6
0
-6
0
-6
-6
-6
-6
-6
0
-6
0

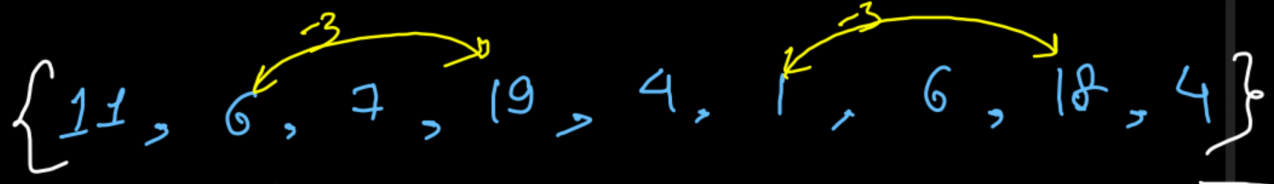
```java
public int maxProfit(int[] prices) {
    int[] buy = new int[prices.length];
    int[] sell = new int[prices.length];

    buy[0] = -prices[0];
    sell[0] = 0;

    for(int i=1; i<prices.length; i++){

        // Treat today as Buying Day
        buy[i] = Math.max(buy[i - 1], sell[i - 1] - prices[i]);

        // Treat today as Selling Day
        sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]);
    }

    return sell[prices.length - 1];
}
```

O(N) DP
Time
O(N)
Space 1D DP

```java
public int maxProfit(int[] prices) {
    int buy = -prices[0];
    int sell = 0;

    for(int i=1; i<prices.length; i++){

        // Treat today as Buying Day
        int newBuy = Math.max(buy, sell - prices[i]);

        // Treat today as Selling Day
        int newSell = Math.max(sell, buy + prices[i]);

        buy = newBuy; sell = newSell;
    }

    return sell;
}
```

O(N) DP Time
O(1) Constant Space

transaction fee ⇒ (+3) → on completion of every transaction (B→S)



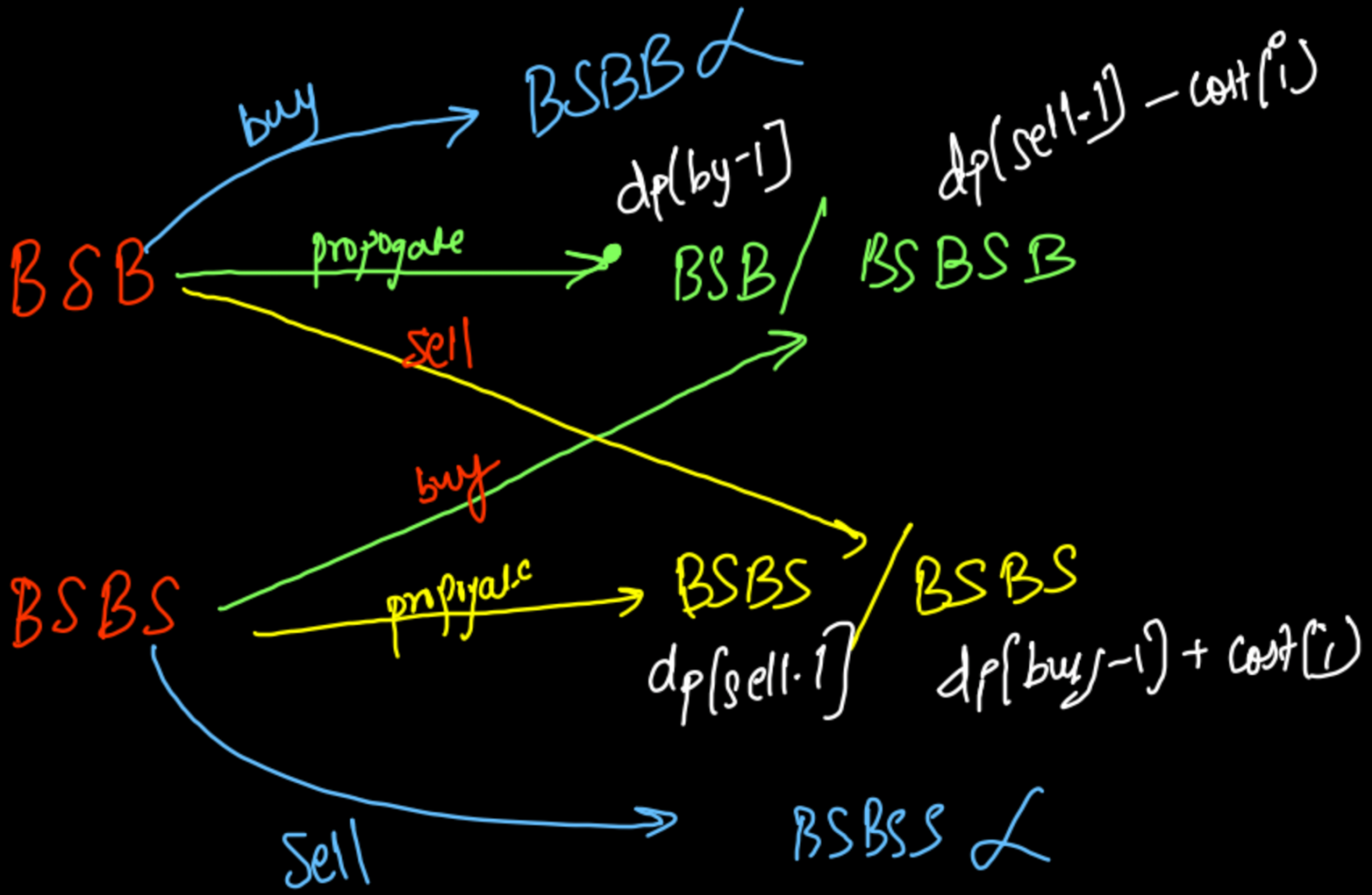
(1 extra stock) Buy	-11	-11 vs -6 -6	-6 vs 0-7 -6	-6 vs 0-19 -6	-6 vs 10-4 6	6 vs 10-1 9	9 vs 10-1 9	9 vs 12-18 9	9 vs 24-4 20
Sell (0 extra stock)	0	-11+6-3 vs 0 0	0 vs 7-6-3 0	0 vs -6+19-3 10	10 vs -6+4-3 10	10 vs 6-1-3 10	10 vs 9+6-3 12	12 vs 9+18-3 24	9+4-3 vs 24 24

$$dp[buy] = dp[buy-1],\ (dp[sell-1] - cost[i])$$

$$dp[sell] = dp[sell-1],\ (dp[buy-1] + cost[i] - fee)$$

```java
public int maxProfit(int[] prices, int fee) {
    int buy = -prices[0];
    int sell = 0;

    for(int i=1; i<prices.length; i++){

        // Treat today as Buying Day
        int newBuy = Math.max(buy, sell - prices[i]);

        // Treat today as Selling Day
        int newSell = Math.max(sell, buy + prices[i] - fee);

        buy = newBuy; sell = newSell;
    }

    return sell;
}
```

O(N) time, O(1) constant space


buy
BSBB
BSB
propogate
dp(by-1]
BSB
dp(sell-1] - cost(i)
BSBSB
sell
buy
BSBS
propgate
BSBS
dp[sell-1]
BSBS
dp[buy-1] + cost(i)
Sell
BSBSS