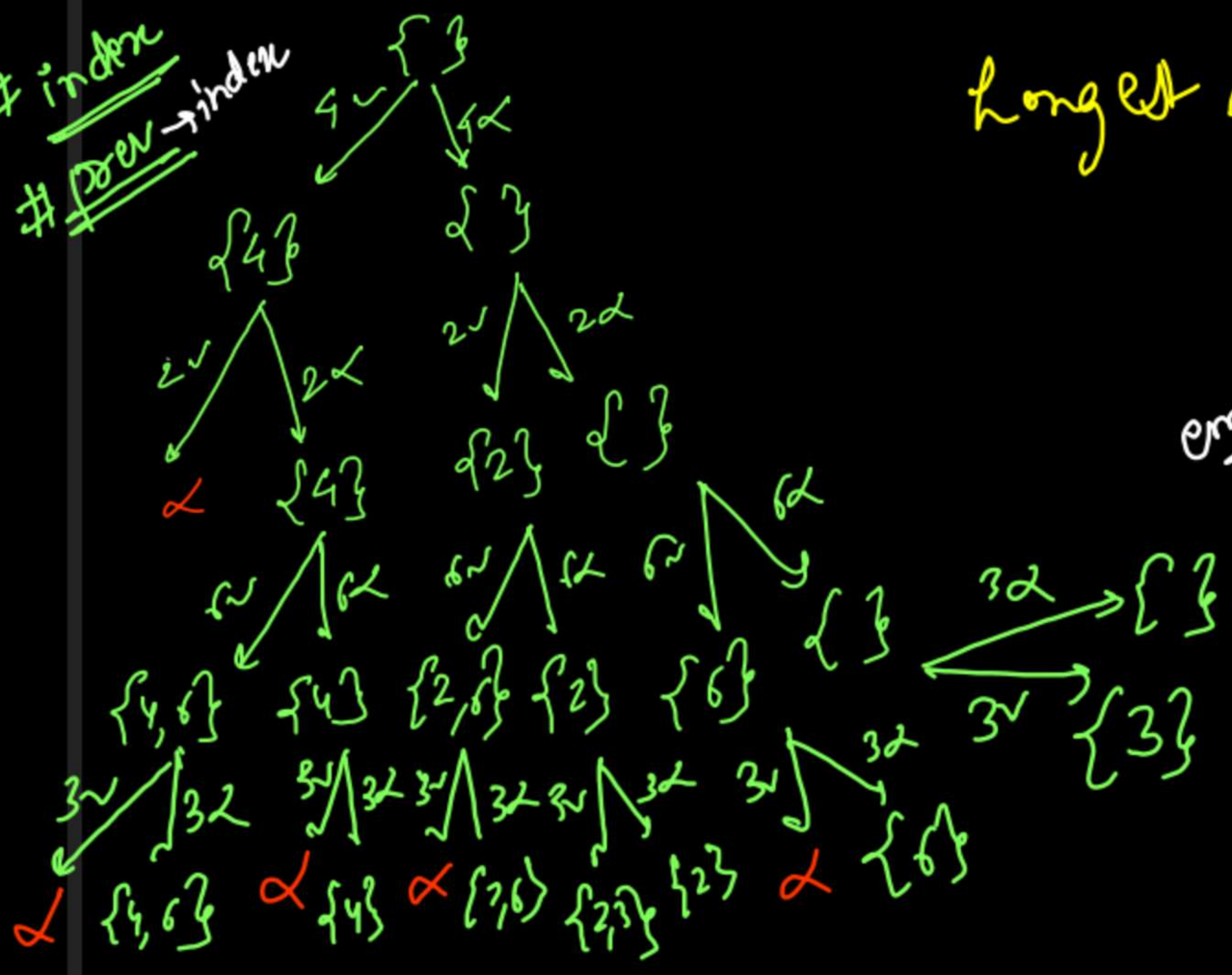
LIS(0, -1)
index prev_index

Longest Increasing (Strictly) Subsequence

↳ subset → 2^N total subsets

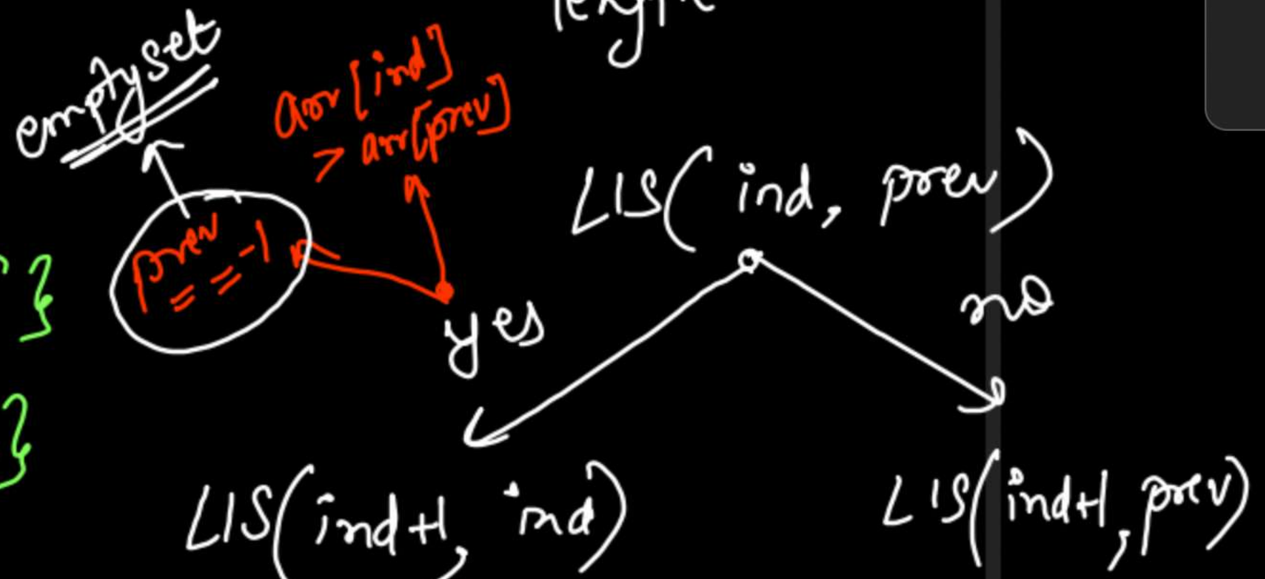
{ 4, 2, 6, 3, 5, 7, 1, 6, 8, 9, 2 }

index
prev → index



Longest LIS: ⇒ { 2, 3, 5, 6, 8, 9 }

length = 6



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

        int yes = (prev == -1 || nums[prev] < nums[curr])
                  ? memo(curr + 1, curr, nums, dp) + 1 : 0;

        int no = memo(curr + 1, prev, nums, dp);

        return dp[curr][prev + 1] = Math.max(yes, no);
    }

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

        return memo(0, -1, nums, dp);
    }
}
```

to store prev:-1 also in the DP

curr element is included ⇒ length ↑ses by 1

Memoization

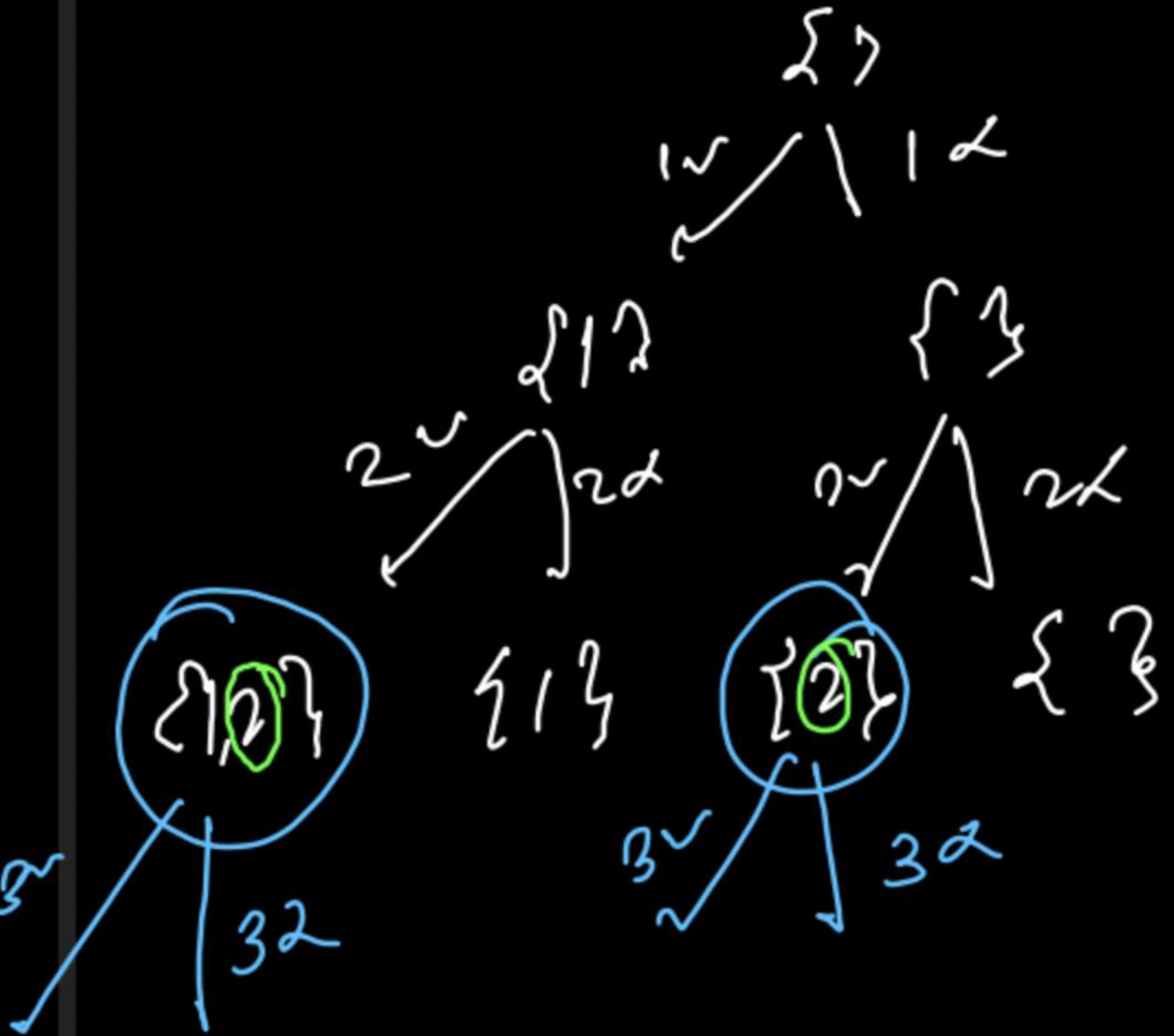
Time → O(N*N)

curr, prev

Space → O(N*N)

2D DP

{1, 2, 3, 4} Overlapping subproblems



Tabulation

1	1	1	1	1	1	1	1	1	1
{4}	{2}	{6}	{3}	{5}	{7}	{1}	{6}	{8}	{4}

4 2 6 3 5 7 1 6, 8 4

dp[i] = longest increasing subsequence length ending at index i

```java
public int lengthOfLIS(int[] nums) {
    int n = nums.length;
    int[] dp = new int[n];
    int maxLIS = 0;
    for(int i=0; i<nums.length; i++){
        dp[i] = 1; // If Prev Does not Exist, then current element can have yes

        for(int j=0; j<i; j++){
            if(nums[j] < nums[i]){
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }

        maxLIS = Math.max(maxLIS, dp[i]);
    }

    return maxLIS;
}
```

Tabulation

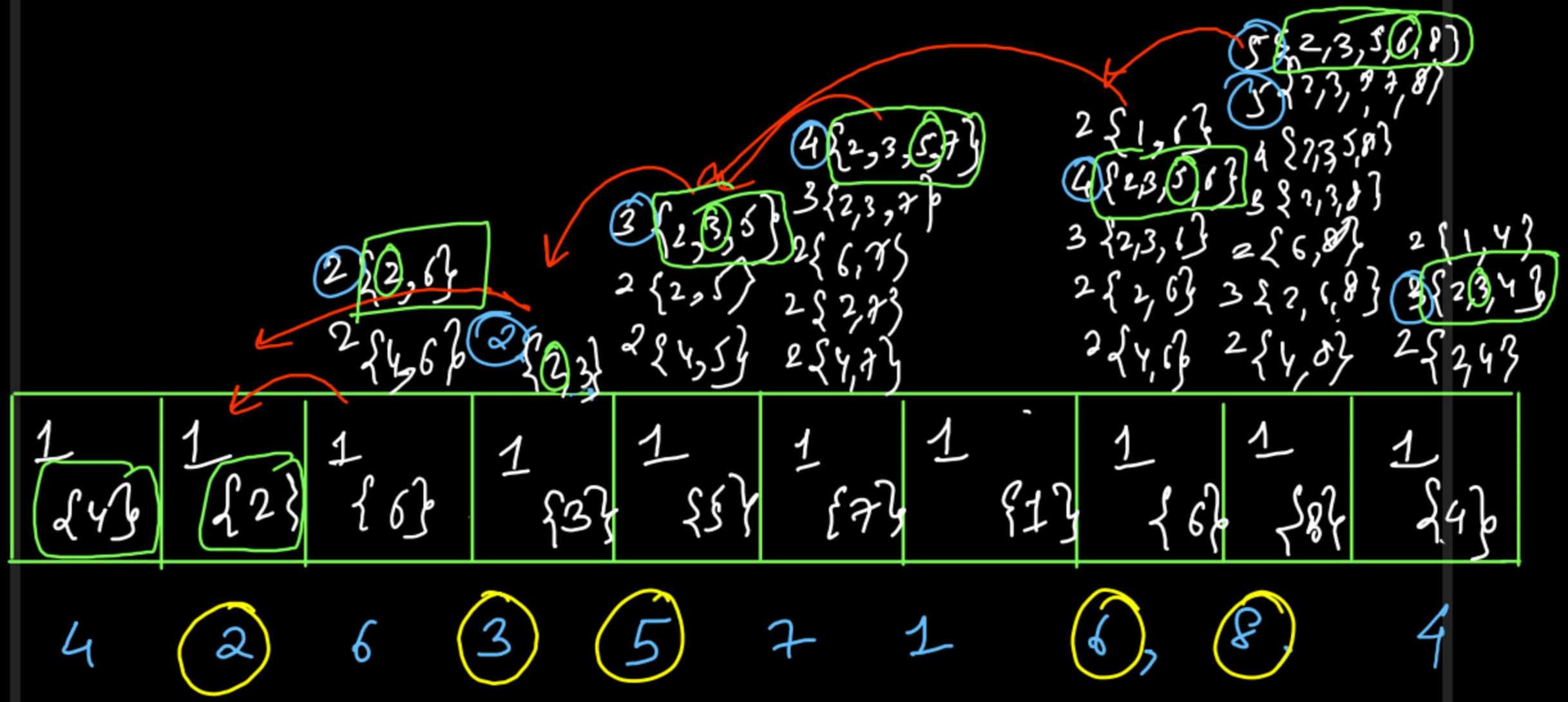
Time → O(N*N)

Space → O(N)

1D DP

Printing LIS

- Any one
  - Arraylist LIS store at each index
  - Prev state
- Print All → Backtracking



Print Any one LIS

```java
public static void solution(int[] nums){
    int n = nums.length;
    ArrayList<Integer>[] dp = new ArrayList[n];

    int maxLIS = 0, maxLISind = 0;

    for(int i=0; i<nums.length; i++){
        dp[i] = new ArrayList<>();
        dp[i].add(nums[i]);

        for(int j=0; j<i; j++){
            if(nums[j] < nums[i]){
                if(dp[j].size() + 1 > dp[i].size()){
                    dp[i] = new ArrayList<>(dp[j]);
                    dp[i].add(nums[i]);
                }
            }
        }

        if(dp[i].size() > maxLIS){
            maxLIS = dp[i].size();
            maxLISind = i;
        }

    }

    for(int val: dp[maxLISind]){
        System.out.print(val + " ");
    }
}
```

✓ Time

$O(N*N*N)$

$= O(N^3)$

✓ Space

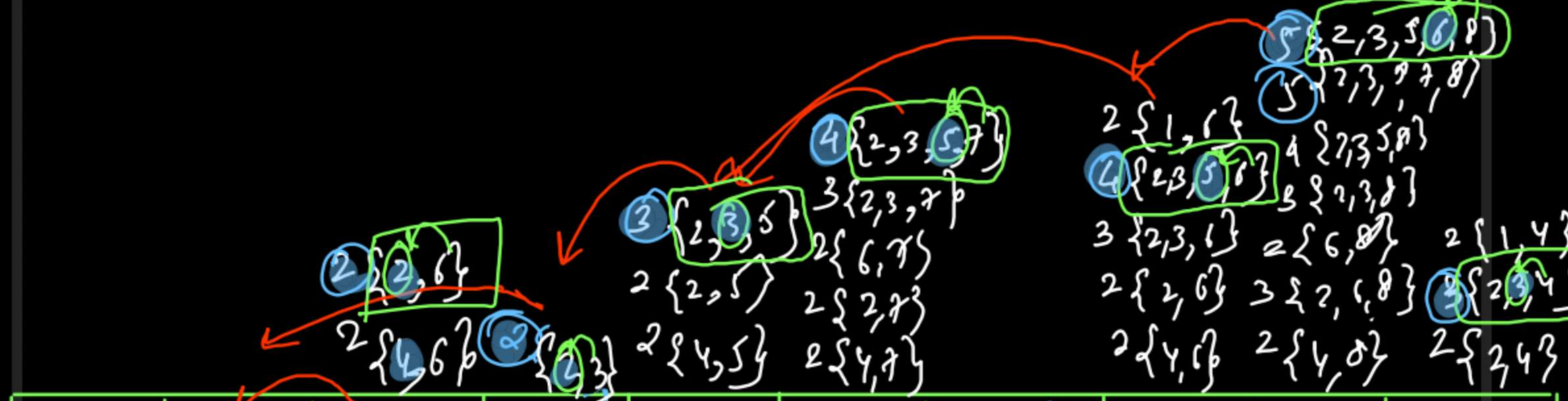
$O(N*N)$

storing AL at each index

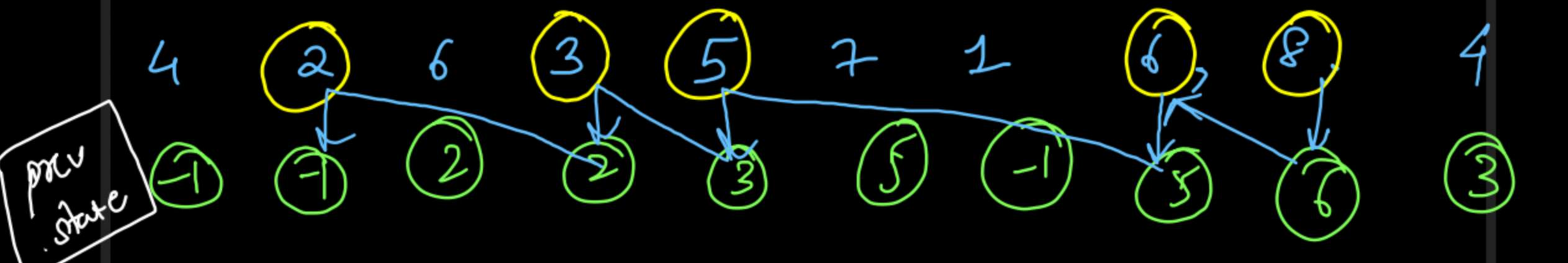
deep copy → $O(N)$

length → prev state

2, 3, 5, 6, 8



1	1	1	1	1	1	1	1	1	1
{4}	{2}	{6}	{3}	{5}	{7}	{1}	{6}	{8}	{4}



```java
public static void solution(int[] nums){
    int n = nums.length;
    int[] dp = new int[n];
    int[] prev = new int[n];

    int maxLIS = 0, lisidx = 0;

    for(int i=0; i<nums.length; i++){
        dp[i] = 1; // Length
        prev[i] = -1; // Empty Subset -> Current Element

        for(int j=0; j<i; j++){
            if(nums[j] < nums[i]){
                if(dp[j] + 1 > dp[i]){
                    dp[i] = dp[j] + 1;
                    prev[i] = j;
                }
            }
        }

        if(dp[i] > maxLIS){
            maxLIS = dp[i];
            lisidx = i;
        }

    }

    ArrayList<Integer> LIS = new ArrayList<>();

    while(lisidx != -1){
        LIS.add(nums[lisidx]);
        lisidx = prev[lisidx];
    }

    Collections.reverse(LIS);
    System.out.println(LIS);
}
```

Time $\geq O(N^2 + N)$
$\approx O(N^2)$

Space $\rightarrow O(2*N)$
$+ O(N)$
$= O(N)$

Backtracking $\rightarrow O(LIS\ length)$
$\approx O(N)$ in worst case

10

maxLIS = 5

Print ALL LIS

1 {4}	1 {2}	1 {6}	1 {3}	1 {5}	1 {7}	1 {1}	1 {6}	1 {8}	1 {7}

4 2 6 3 5 7 1 6, 8 7

LIS of length = 5

2, 3, 5, 6, 8

2, 3, 5, 7, 8

2, 3, 5, 6, 7

LIS

End at diff index

Same index ↓ more than 2 LIS is also possible

DFS(7) {7}

6

Increasing order

if nums[j] < nums[i]

dp[j]+1 = dp[i]

DFS(6) {6,7}

DFS(5) {5,6,7}

DFS(8)

DFS(6)

DFS(7)

DFS(5)

DFS(3)

{2,3,5,6,8}

{2,3,5,3,3}

DFS(3) {3,5,6,7}

DFS(2)

{2,3,5,6,7}

```java
public static void DFS(int curr, int[] nums, int[] dp, String psf) {
    if (dp[curr] == 1) {
        System.out.println(psf);
        return;
    }

    for (int prev = curr - 1; prev >= 0; prev--) {
        if (nums[prev] < nums[curr] && dp[curr] == dp[prev] + 1) {
            DFS(prev, nums, dp, nums[prev] + " -> " + psf);
        }
    }
}
```

Increasing

Longest

2 LIS can have same ending index

Time

Worst case → O(exponential)

Avg case → O(polynomial)

```java
public static void solution(int[] nums) {
    int n = nums.length;
    int[] dp = new int[n];

    int maxLIS = 0;

    for (int i = 0; i < nums.length; i++) {
        dp[i] = 1; // Length

        for (int j = 0; j < i; j++) {
            if (nums[j] < nums[i]) {
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }

        if (dp[i] > maxLIS) {
            maxLIS = dp[i];
        }
    }

    System.out.println(maxLIS);
```

```java
for (int i = n - 1; i >= 0; i--) {
    // Start DFS from each node at which Increasing Subset is of
    // Longest Length

    if (dp[i] == maxLIS) {
        DFS(i, nums, dp, "" + nums[i]);
    }
}
```

Multiple Sources ↳ LIS can end at diff indices

Space

O(N) → DP, DFS

LIS

- LIS → Time optimization → O(NlogN) (Binary Search)
  - LIS length
- Count LIS

Dynamic Programming

Lecture 22

→ Length of LIS → O(NlogN) optimized BS approach

→ Count of LIS

Longest Increasing Subsequence Variations

Longest Increasing Subsequence → BS Approach

{ 4, 2, 6, 3, 5, 7, 1, 6, 8, 2 }

{4} {2} {4,6} {3,3} {2,6} {3,5} {2,3,5} {1} {1,6} {1,8} {1,2} {2,3,5,7} {2,3,5,6} {2,3,5,8} {1,2,5,6,8} {2,3,5,7,8} {2,3,5,6,8}

Length

```java
public int lowerBound(ArrayList<Integer> nums, int target){
    int low = 0, high = nums.size() - 1;
    int idx = nums.size();

    while(low <= high){
        int mid = low + (high - low) / 2;

        if(nums.get(mid) < target){
            low = mid + 1;
        } else {
            high = mid - 1;
            idx = mid;
        }
    }

    return idx;
}
```

→ Lower Bound → $O(\log_2 N)$

```java
public int lengthOfLIS(int[] nums) {
    int n = nums.length;

    ArrayList<Integer> sorted = new ArrayList<>();

    for(int i=0; i<nums.length; i++){
        int lb = lowerBound(sorted, nums[i]);
        if(lb == sorted.size()){
            sorted.add(nums[i]);
            // Current Element larger than the largest
            // LIS of one length more
        } else {
            sorted.set(lb, nums[i]);
        }
    }

    return sorted.size(); // This Sorted Array has same size as LIS
}
```

→ $O(N * \log_2 N)$

↑ Binary Search Based Time Optimization!

Lower Bound

Element found → get that index in sorted array

Else just greater value's index

Corner case

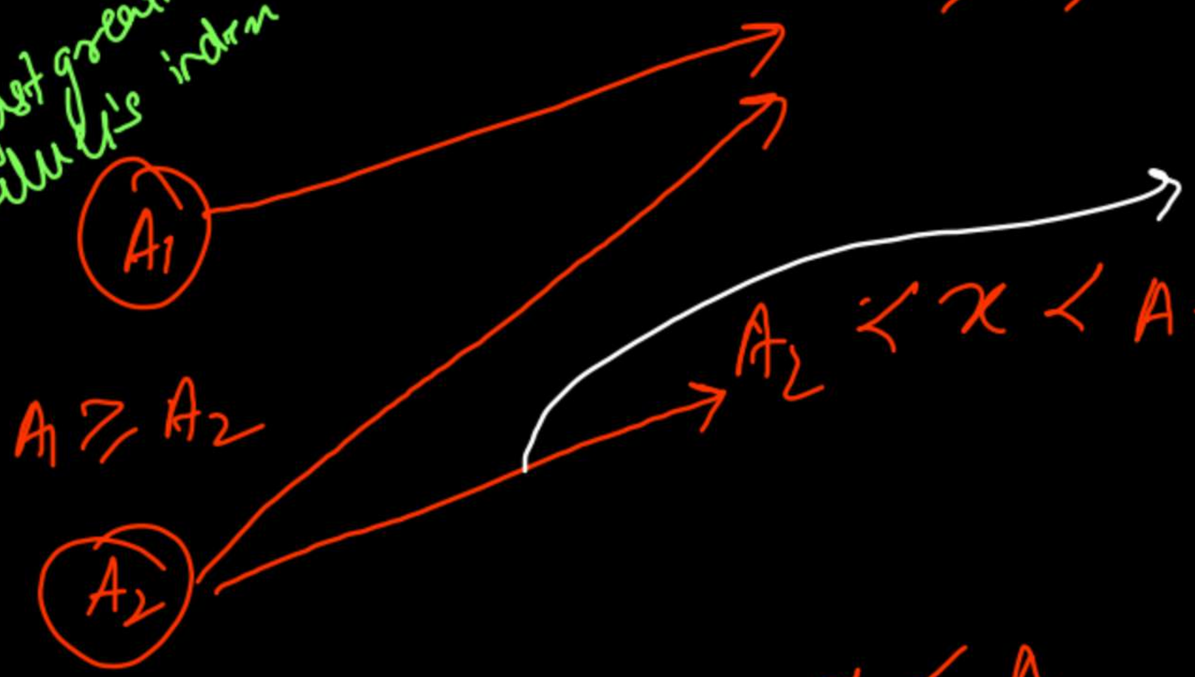
(2) {2,3}

(2) {2}

(3) {2,3}

(4) {3,3}

(4) The subsequence which you get at last is not the actual LIS → it just represents the length of LIS.



I will prefer smaller value over larger value when subsequences have same length

Count LIS

lc 673

→ There can be more than 1 LIS ending at given index

→ There can be more than 1 such indexes

$$\#\ count[i] = \sum_{j} count[j]$$

where $dp(i) = dp[j] + 1$ if $nums[i] > nums[j]$

Longest ① ① ①

increasing ③ + ②

5 + 5 = 10 LIS

count (DP)

{4} {2} {2,3} {2,3,6} {2,3,5} {2,3,4} {2,3,6,7} {2,3,5,7}

① ① ① ① ① ① ①

{1,3,4,7} {1,3,5,7} {1,3,6,7} {1,3,4,6}

{2,3,4,7,8} {1,3,5,7,8} {2,3,4,6,8} {2,3,5,6,8}

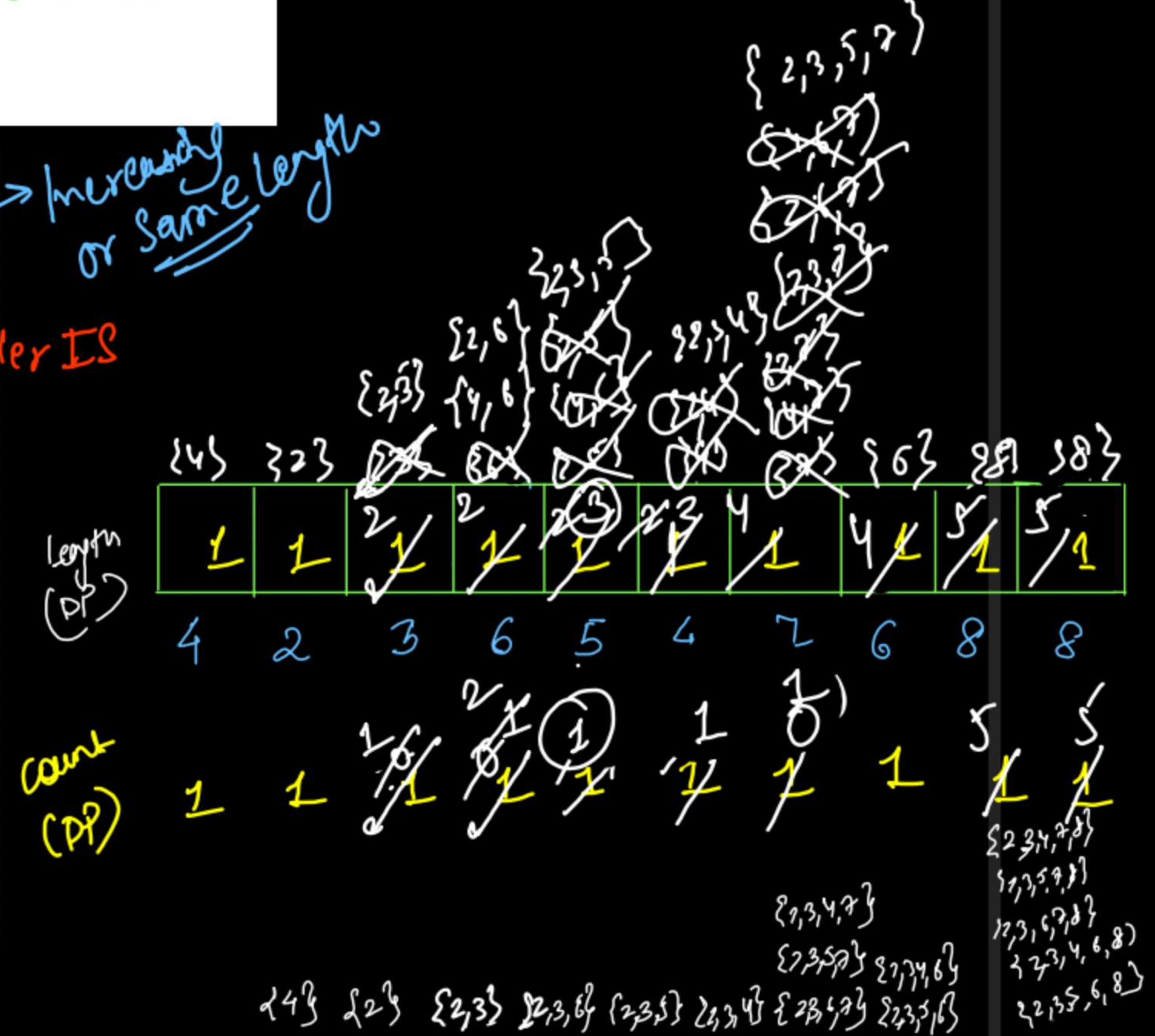
1	1	2	3	3	3	4	4	5	5

Length (DP)

4 2 3 6 5 4 7 6 8 8 1

```java
int n = nums.length;
int[] dp = new int[n]; // Length of LIS ending at index i
Arrays.fill(dp, 1);

int[] count = new int[n]; // Count of LIS ending at index i
Arrays.fill(count, 1);

int maxLIS = 0;
```

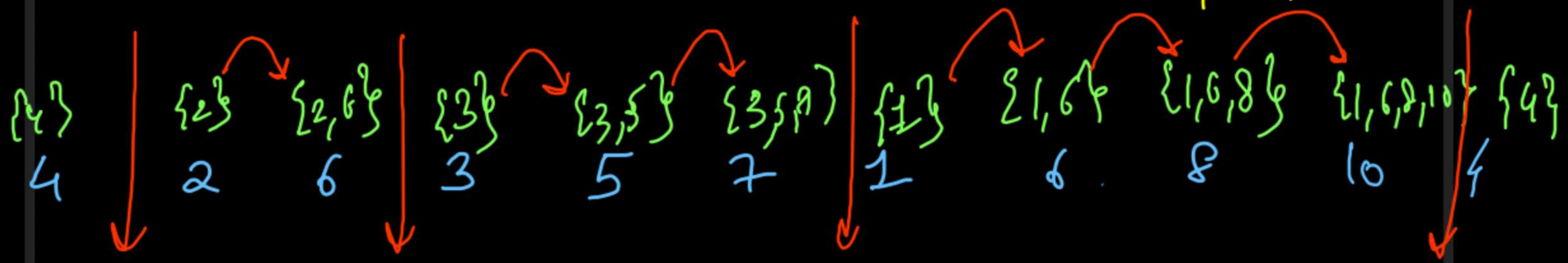
```java
for(int i=0; i<n; i++){

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

        if(nums[j] < nums[i] && dp[i] <= dp[j] + 1){
            if(dp[i] < dp[j] + 1)
                count[i] = 0;

            dp[i] = Math.max(dp[i], dp[j] + 1);
            count[i] += count[j];
        }

    }

    maxLIS = Math.max(maxLIS, dp[i]);
}

int countLIS = 0;
for(int i=0; i<n; i++){
    if(dp[i] == maxLIS)
        countLIS += count[i];
}

return countLIS;
```



LC 674 Longest Increasing Subarray

LIS + Kadane's

Subarray / Substring
↑ contiguous
Subsequence / Subset



current length = 1 1 2 1 2 3 1 2 3 4 1

max LIS = 1 1 2 3 4

```java
public int findLengthOfLCIS(int[] nums) {
    int curr = 0, max = 0;

    for(int i=0; i<nums.length; i++){

        if(i > 0 && nums[i-1] < nums[i]){
            curr++; // Extend the Previous Subarray
        } else {
            curr = 1; // Start New Increasing Subarray
        }

        max = Math.max(max, curr);
    }

    return max;
}
```

Time → O(N) linear

Space → O(1) Constant