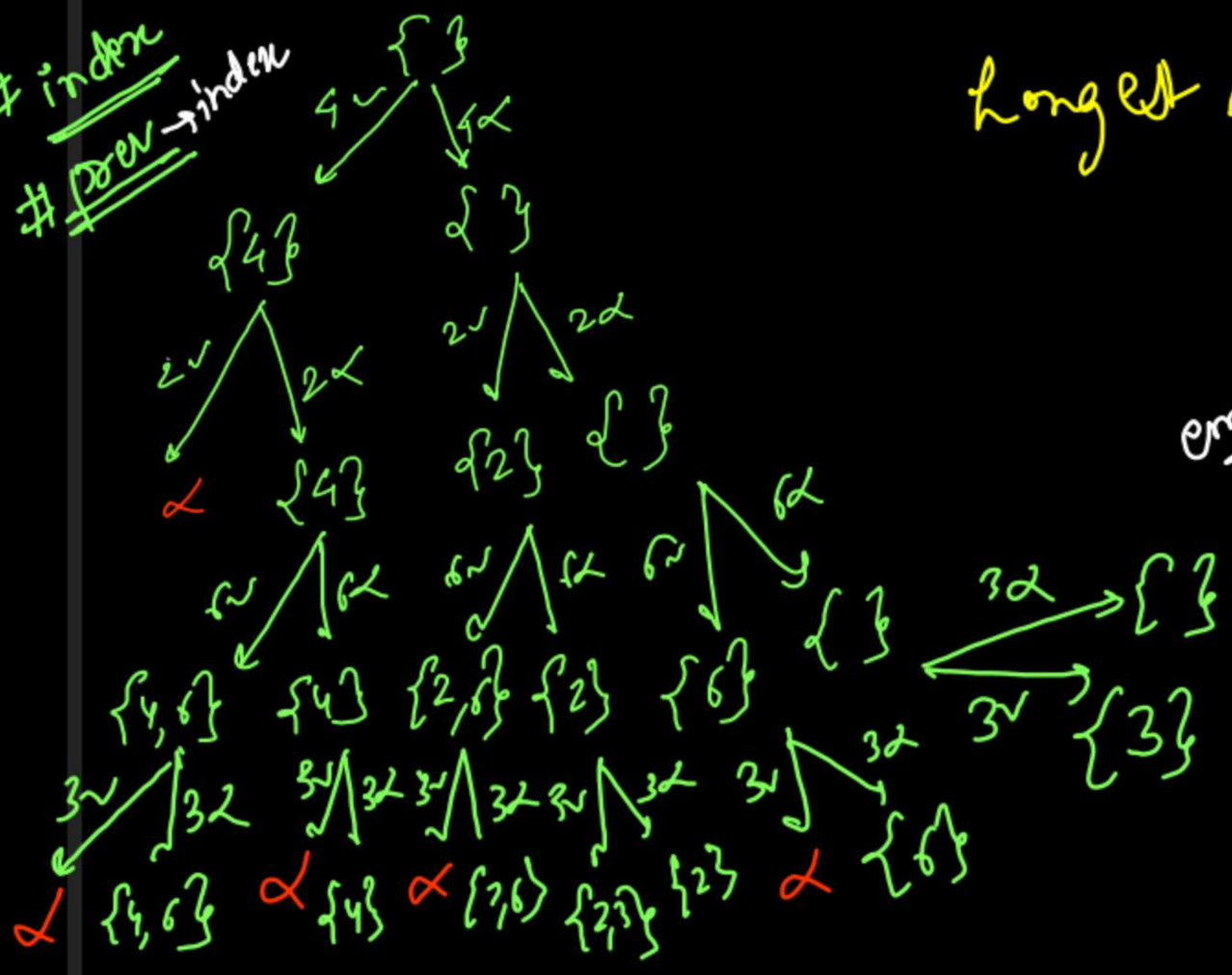
LIS(0, -1)
index prev_index

Longest Increasing (Strictly) Subsequence

↳ subset → 2^N total subsets

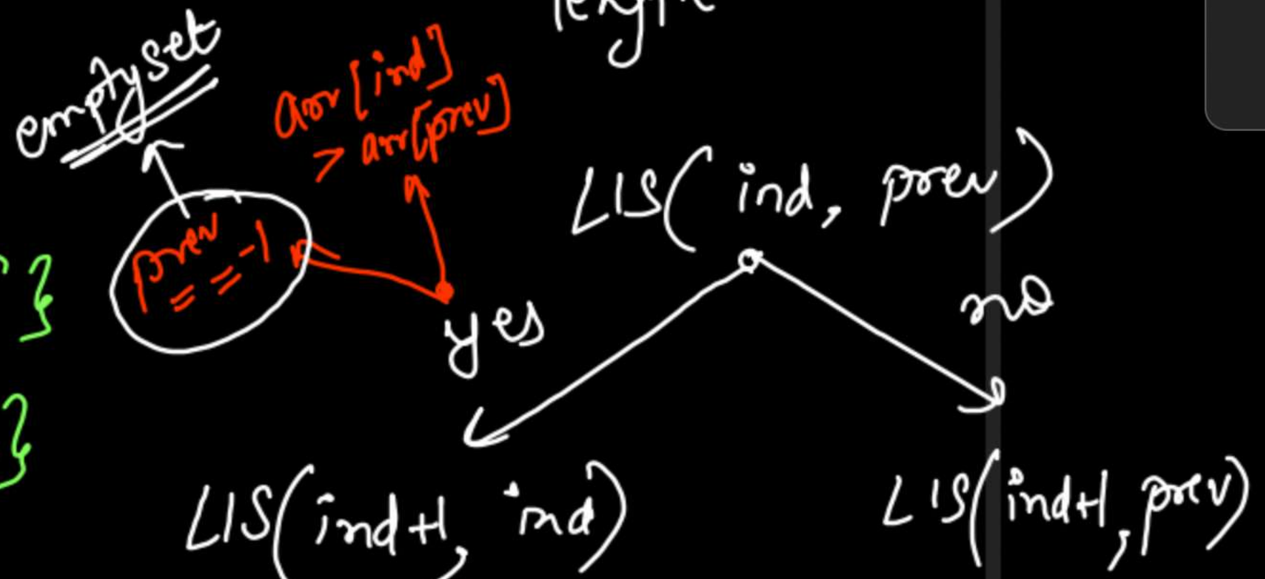
{ 4, 2, 6, 3, 5, 7, 1, 6, 8, 9, 2 }

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

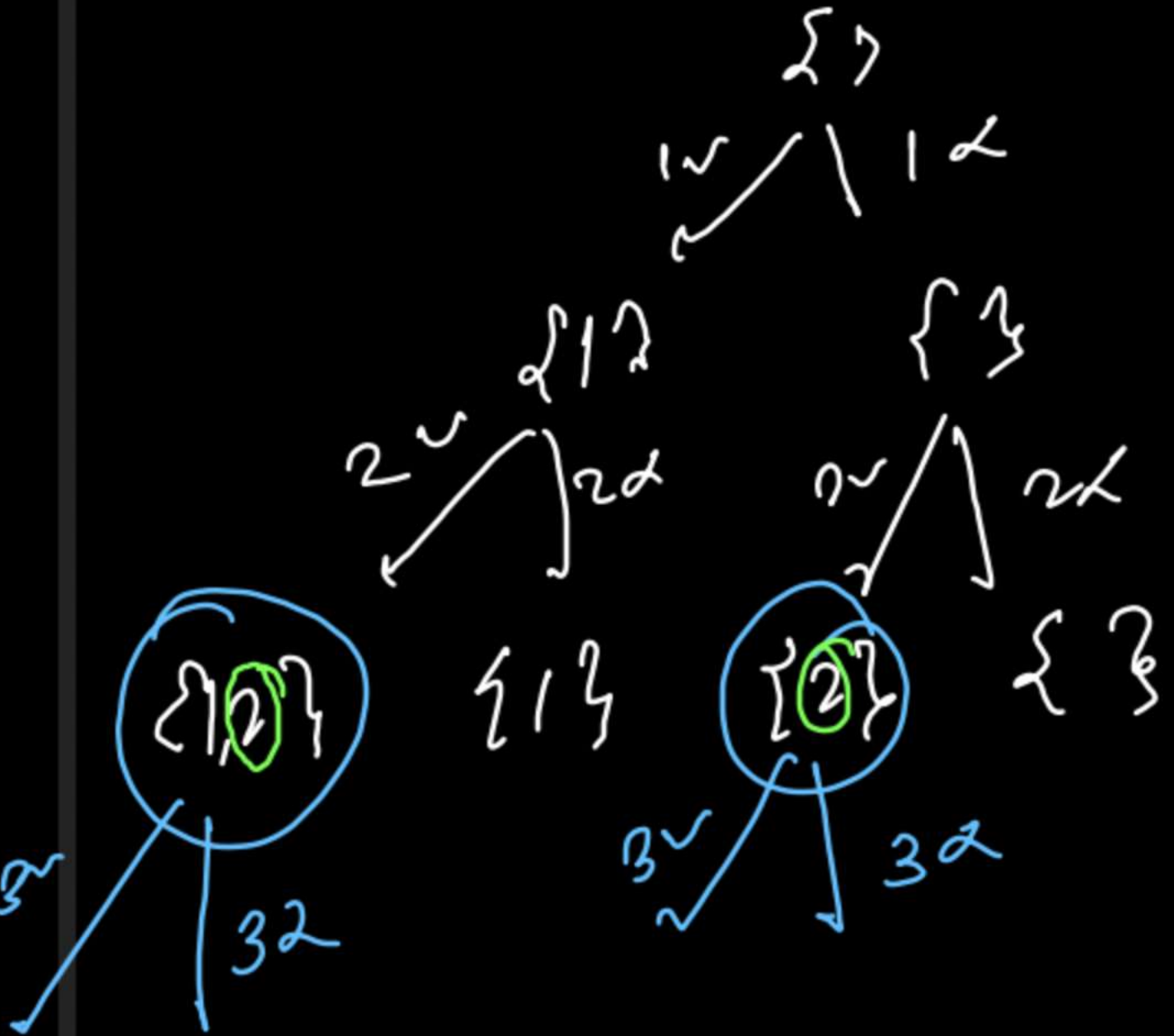
Time → O(N*N)

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

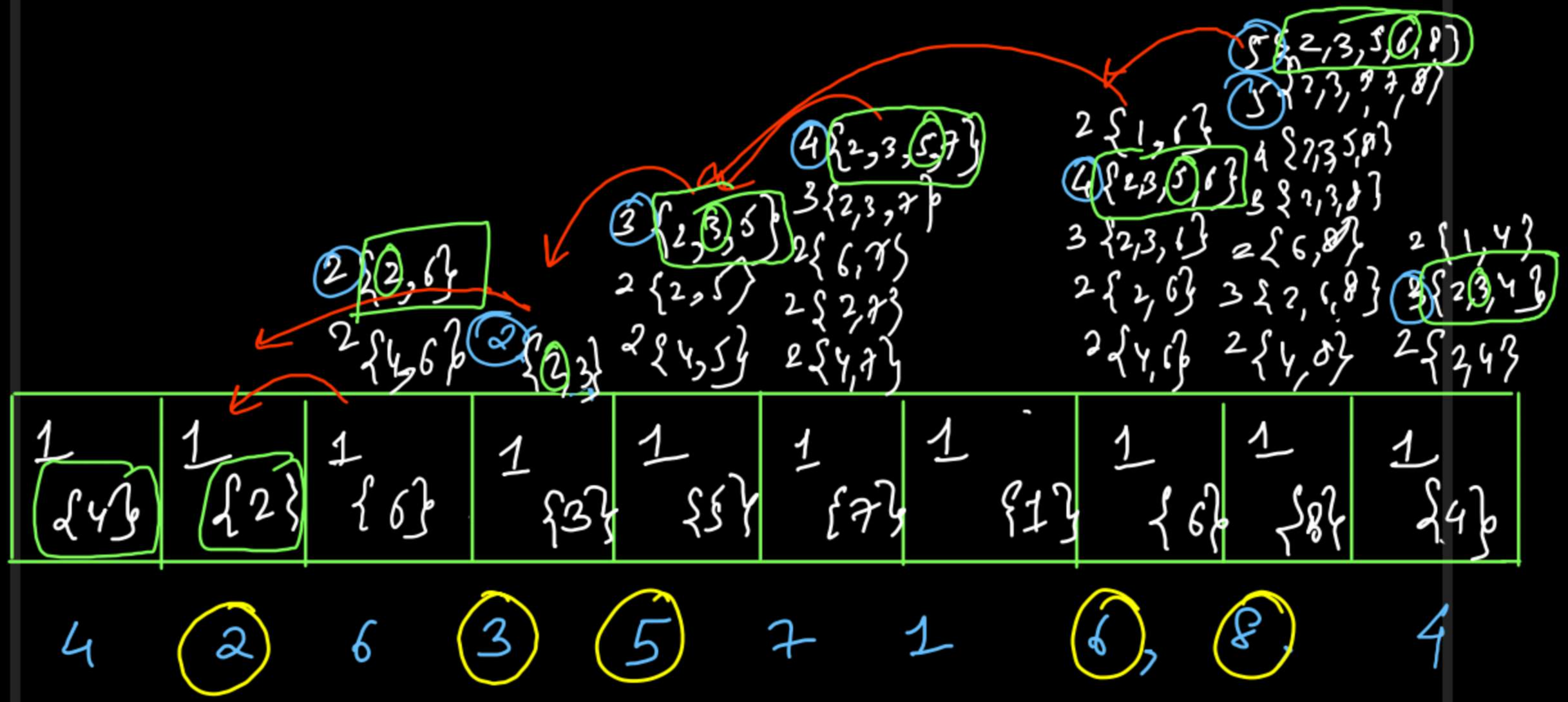
Time → O(N*N)

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

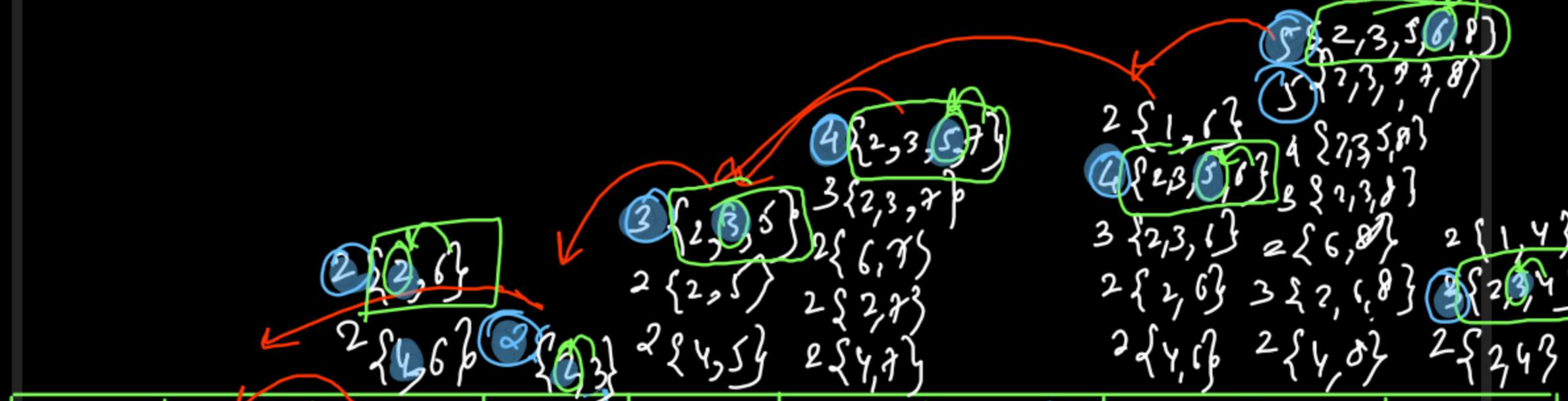
$O(N*N)$

storing AL at each index

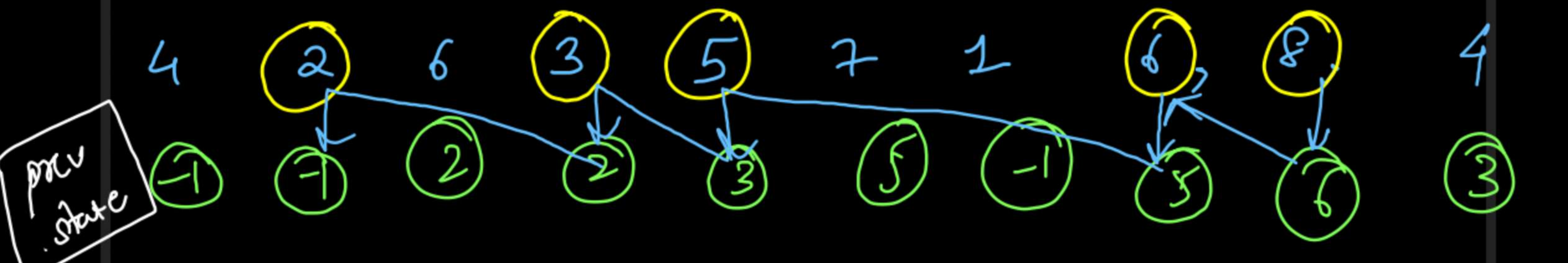
deep copy → $O(N)$

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

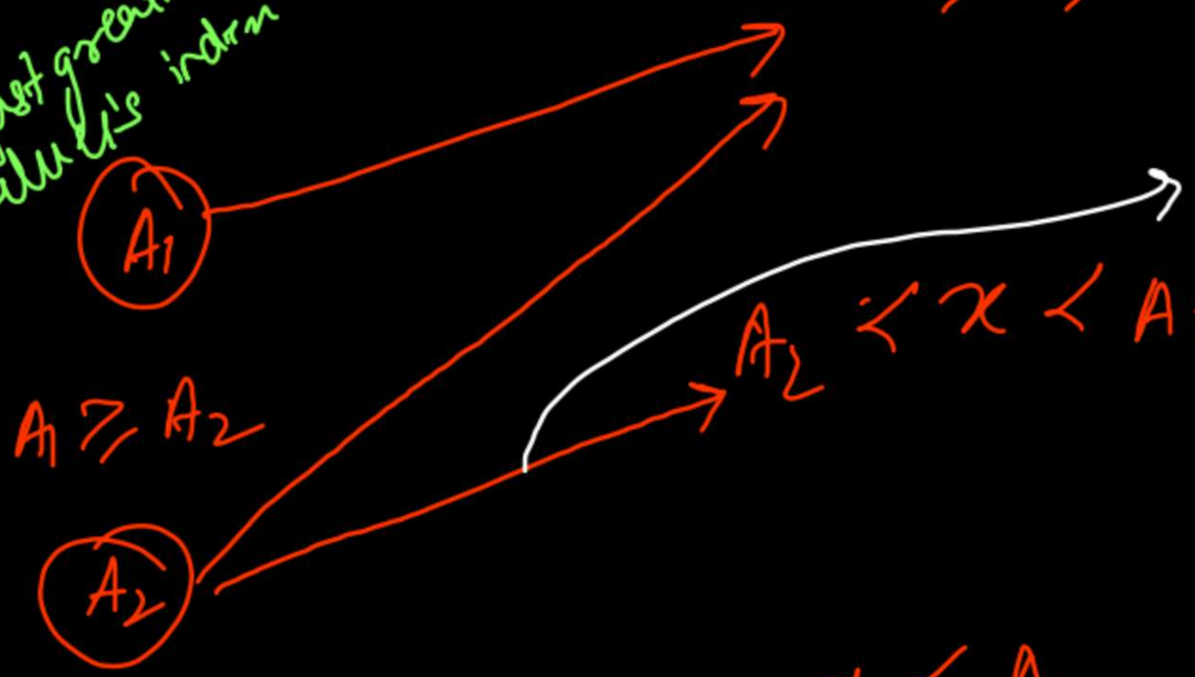
(2) {2,3}

(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}

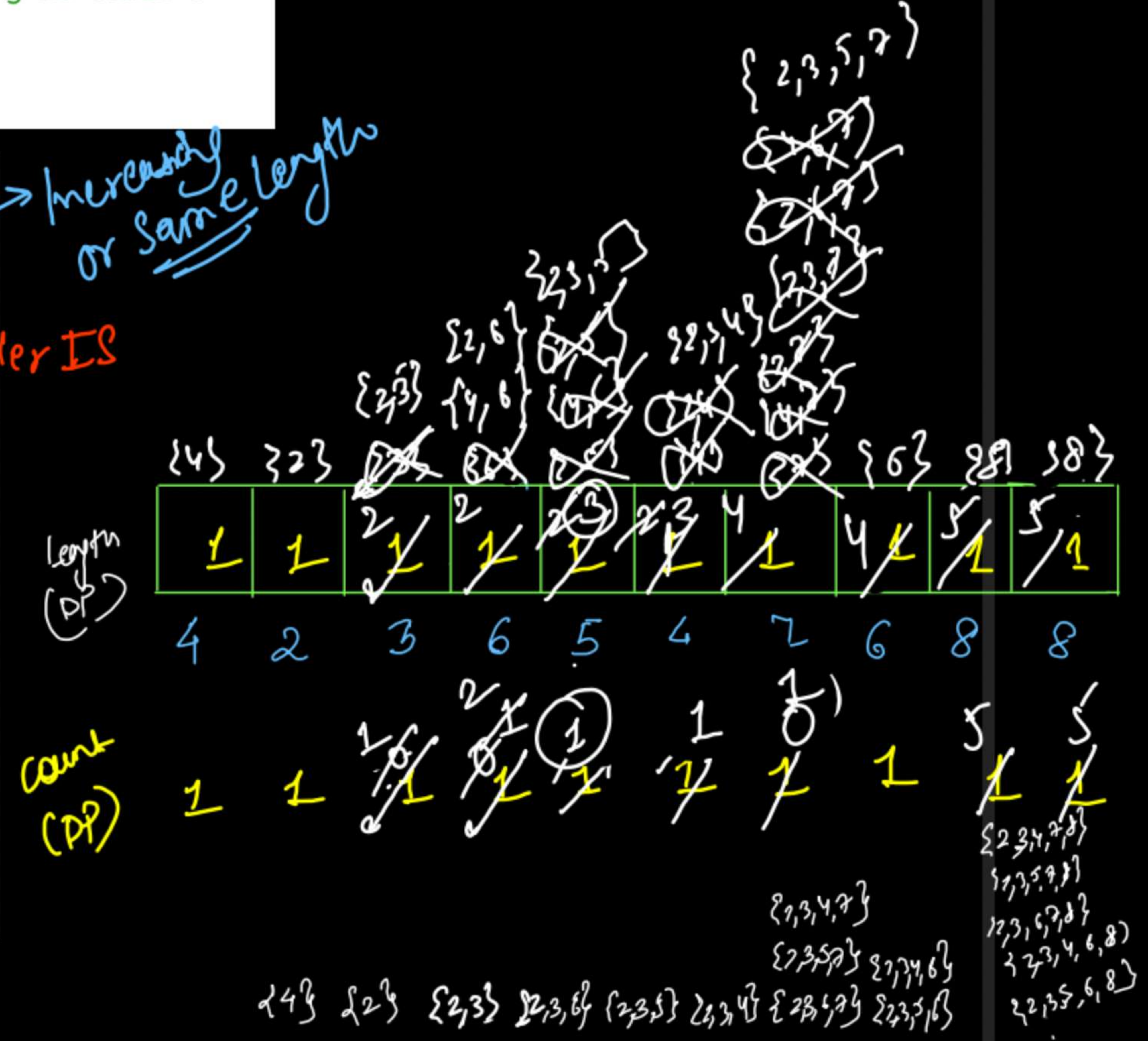
1	1	2	3	3	3	4	4	5	5

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;
```

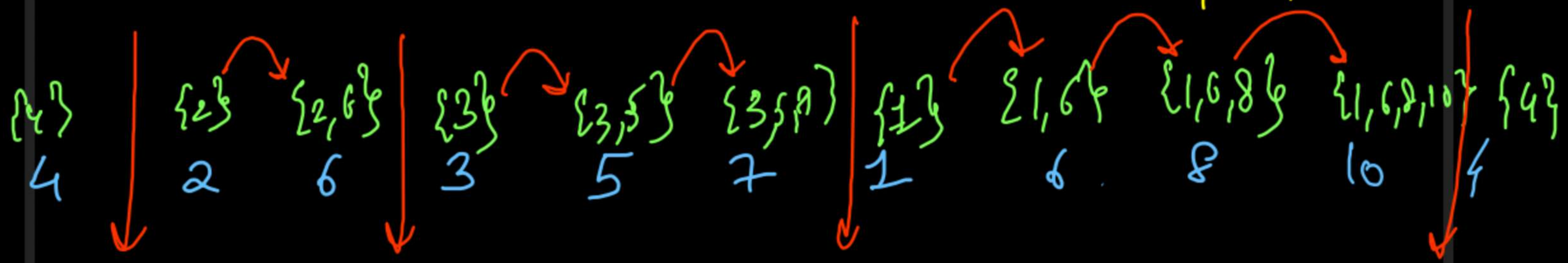
```java
for(int i=0; i<n; i++){

    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

GFG

Max Sum Increasing Subsequence → LIS

Longest Sum

{4}	{2}	{4,6}	{2,3}	{2,3,5}	{4,5,7}	{1}	{2,3,5,6}	{1,2}
4	2	6	3	5	7	1	6	2
(4)	(2)	(10)	(5)	(10)	(17)	(1)	(16)	(3)

LIS

① Ordering
Increasing, decreasing,
Inc-Dec (Peak-valley)
Bonlous studies
on priority

②

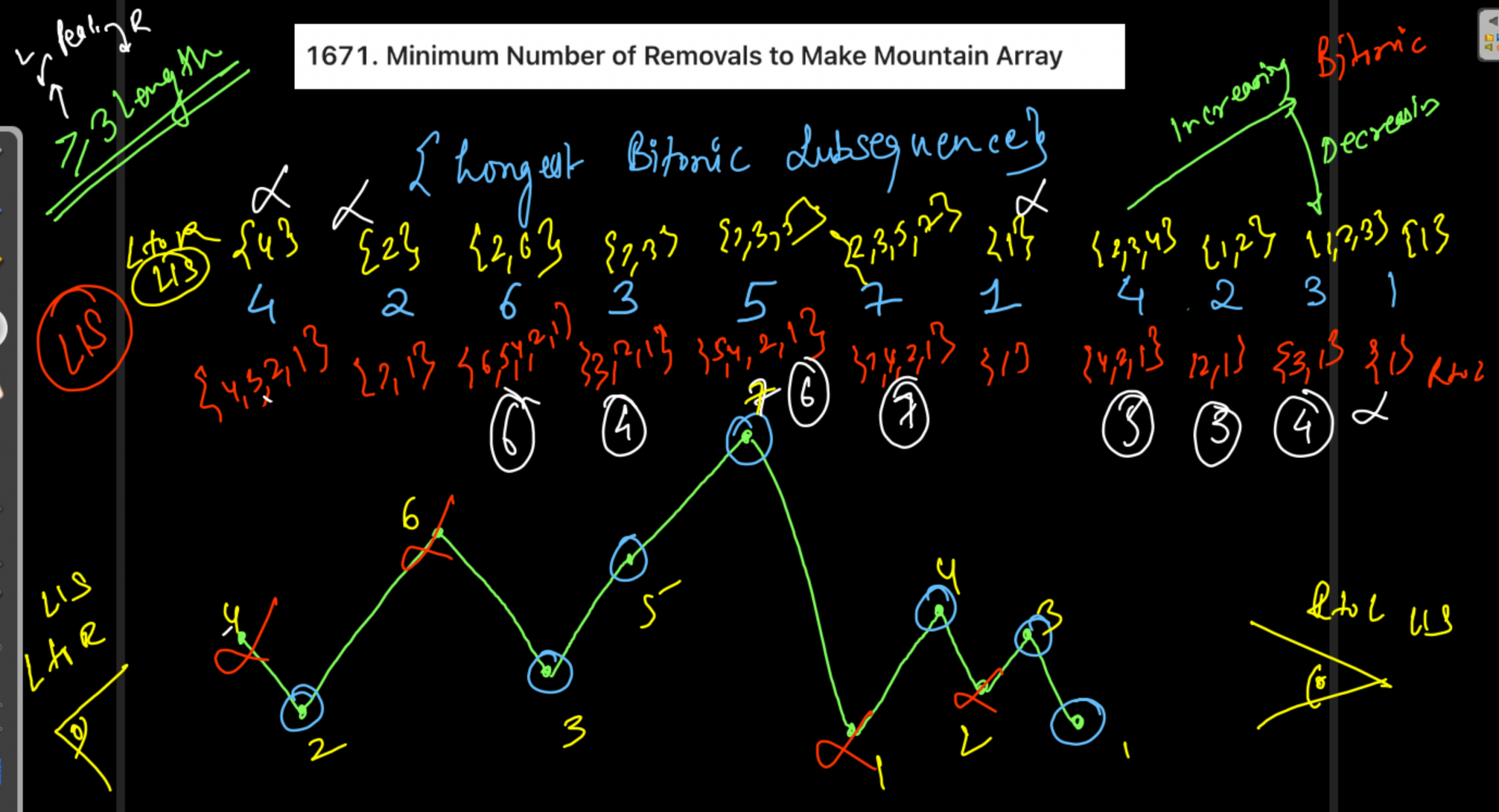
```java
public int maxSumIS(int nums[], int n)
{
    int[] dp = new int[n];
    int maxSum = 0;
    for (int i = 0; i < nums.length; i++) {
        dp[i] = nums[i]; // If Prev Does not Exist, then

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

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

    return maxSum;
}
```

Time → O(N×N)

Space → O(N)

1D DP

1671. Minimum Number of Removals to Make Mountain Array



```
int[] left = new int[nums.length];
// Longest Increasing Subsequence ending at index i

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

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


int[] right = new int[nums.length];
// Longest Decreasing Subsequence starting at index i

for(int i=nums.length-1; i>=0; i--){
    right[i] = 1;

    for(int j=nums.length-1; j>i; j--){
        if(nums[j] < nums[i]){
            right[i] = Math.max(right[i], right[j] + 1);
        }
    }
}
```

$O(n^2)$

$O(n^2)$

```
// Longest Bitonic Subsequence with peak at index i
// = LIS ending at i + LDS starting at i - 1
// -1 represents the peak element occuring in both LIS and LDS
// Constraint: LIS > 1 && LDS > 1 (Alteast 1 element to the left of peak
// and atleast 1 element to the right of peak)
int maxBitonic = 0;
for(int i=0; i<nums.length; i++){
    int curr = left[i] + right[i] - 1;
    if(left[i] > 1 && right[i] > 1){
        maxBitonic = Math.max(maxBitonic, curr);
    }
}

return nums.length - maxBitonic;
```

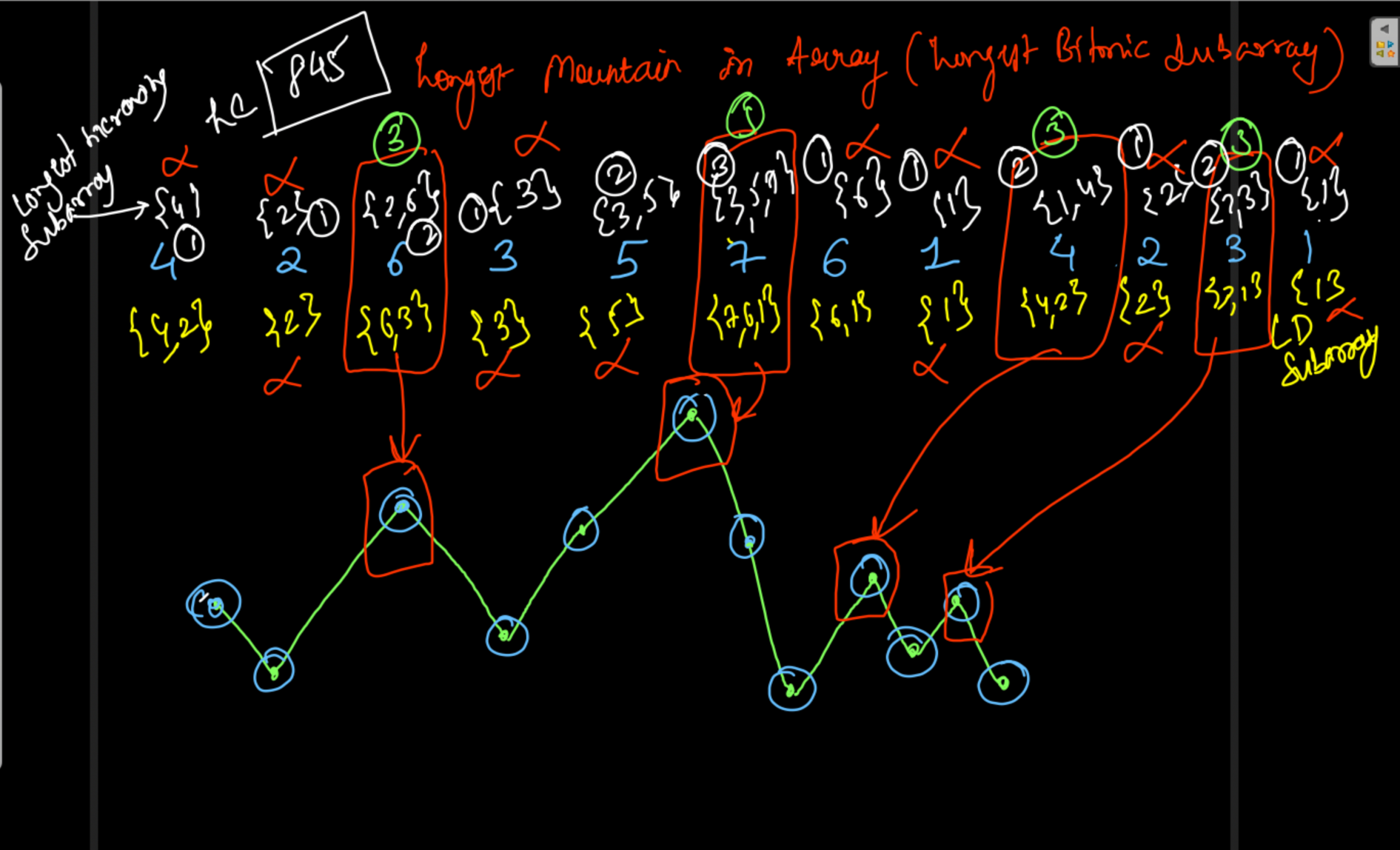
$O(n^2)$

Return the nr of elements to be removed to form longest Bitonic subsequence

longest Mountain in Array (longest Bitonic subarray)

LC 845

Longest increasing subarray



```java
public int longestMountain(int[] nums) {
    int[] left = new int[nums.length];
    // Longest increasing subarray ending at index i
    Arrays.fill(left, 1);

    for(int i=1; i<nums.length; i++){
        if(nums[i - 1] < nums[i]){
            left[i] = left[i - 1] + 1;
        }
    }


    int[] right = new int[nums.length];
    // Longest decreasing subarray starting at index i
    Arrays.fill(right, 1);

    for(int i=nums.length-2; i>=0; i--){
        if(nums[i] > nums[i + 1]){
            right[i] = right[i + 1] + 1;
        }
    }
```

```java
  // Longest Bitonic Subarray = LIS + LDS - 1
  int max = 0;
  for(int i=0; i<nums.length; i++){
      if(left[i] > 1 && right[i] > 1){
          max = Math.max(max, left[i] + right[i] - 1);
      }
  }

  return max;
```

→ O(N)

+

→ O(N)

+

→ O(N)

longest Bitonic Subarray

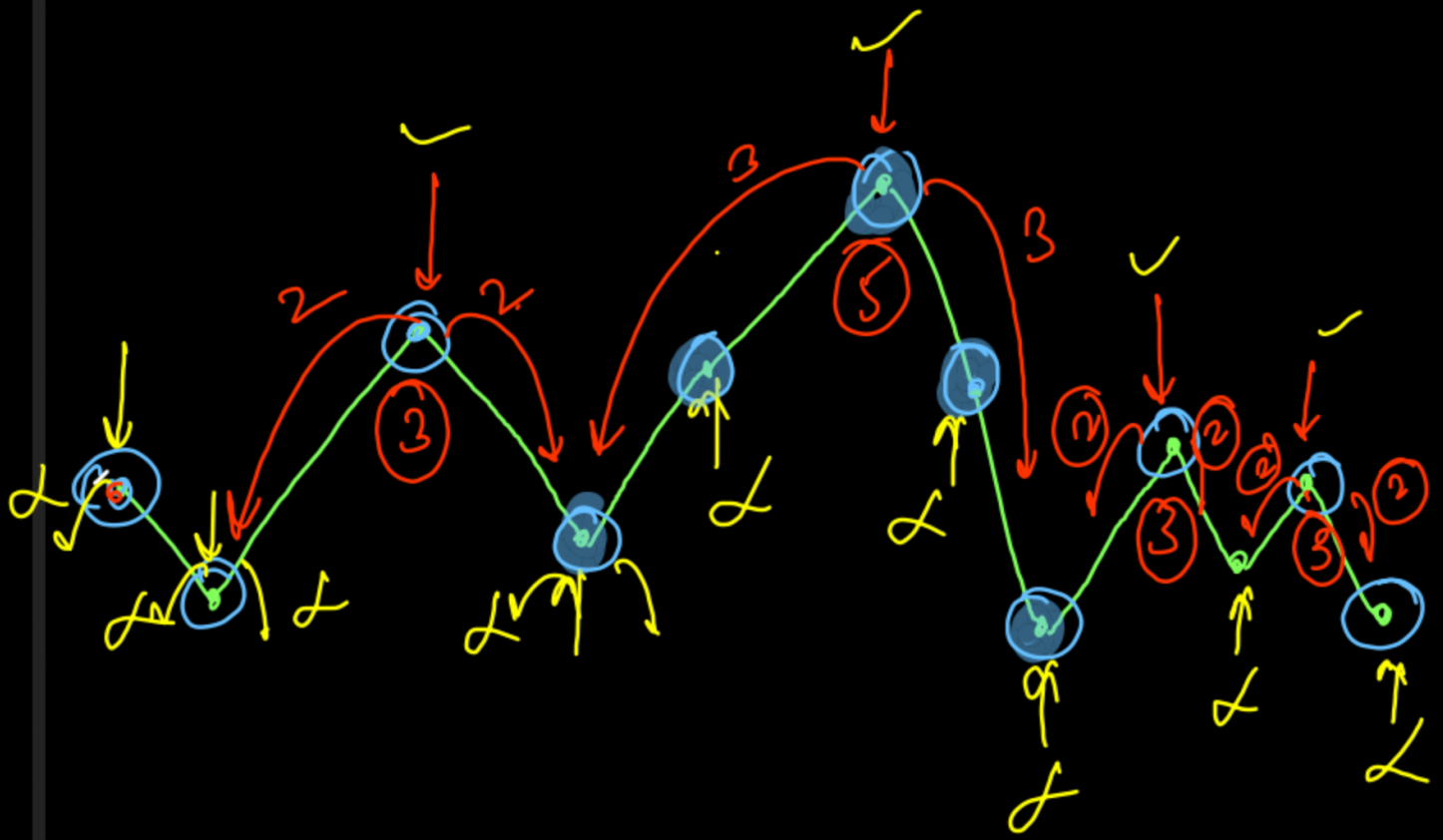
Time → O(N) (3N)

Space → O(N) (2N)

max = ϕβs

Greedy Soln

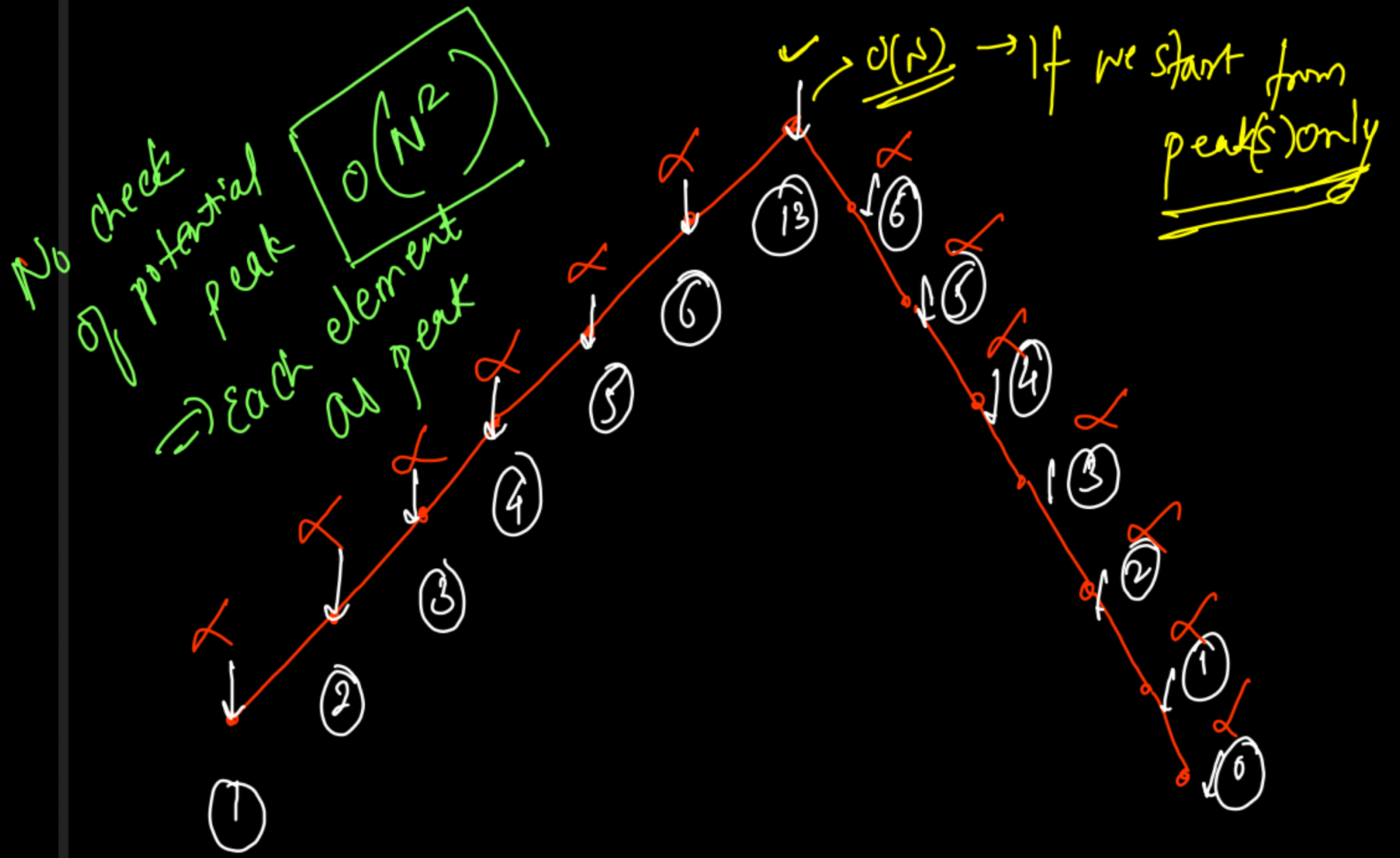
↳ Two pointer Technique



```
int max = 0;

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

    // Potential Peak Element
    if(i > 0 && i < nums.length - 1 && nums[i] > nums[i - 1] && nums[i] > nums[i + 1]){

        int curr = 1;

        int left = i - 1;
        while(left >= 0 && nums[left] < nums[left + 1]){
            left--;
            curr++;
        }

        int right = i + 1;
        while(right < nums.length && nums[right] < nums[right - 1]){
            right++;
            curr++;
        }

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

}
```

Time → O(N)
Linear

Space → O(1)
Two Pointers


No check of potential peak
O(N²)
⇒ Each element as peak
O(N) → If we start from peak(s) only
1
2
3
4
5
6
13
6
5
4
3
2
1
0

LC 368 Largest Divisible Subset

distinct positive

eg { 10, 2, 8, 3, 4, 1, 5, 16, 20, 9 }

(A[i], A[j])

A[i] % A[j] == 0

or

A[j] % A[i] == 0

sorted

i → A[i]

j → A[j]

(1,5) 10 % 5 == 0

(5,20) 20 % 5 = 0

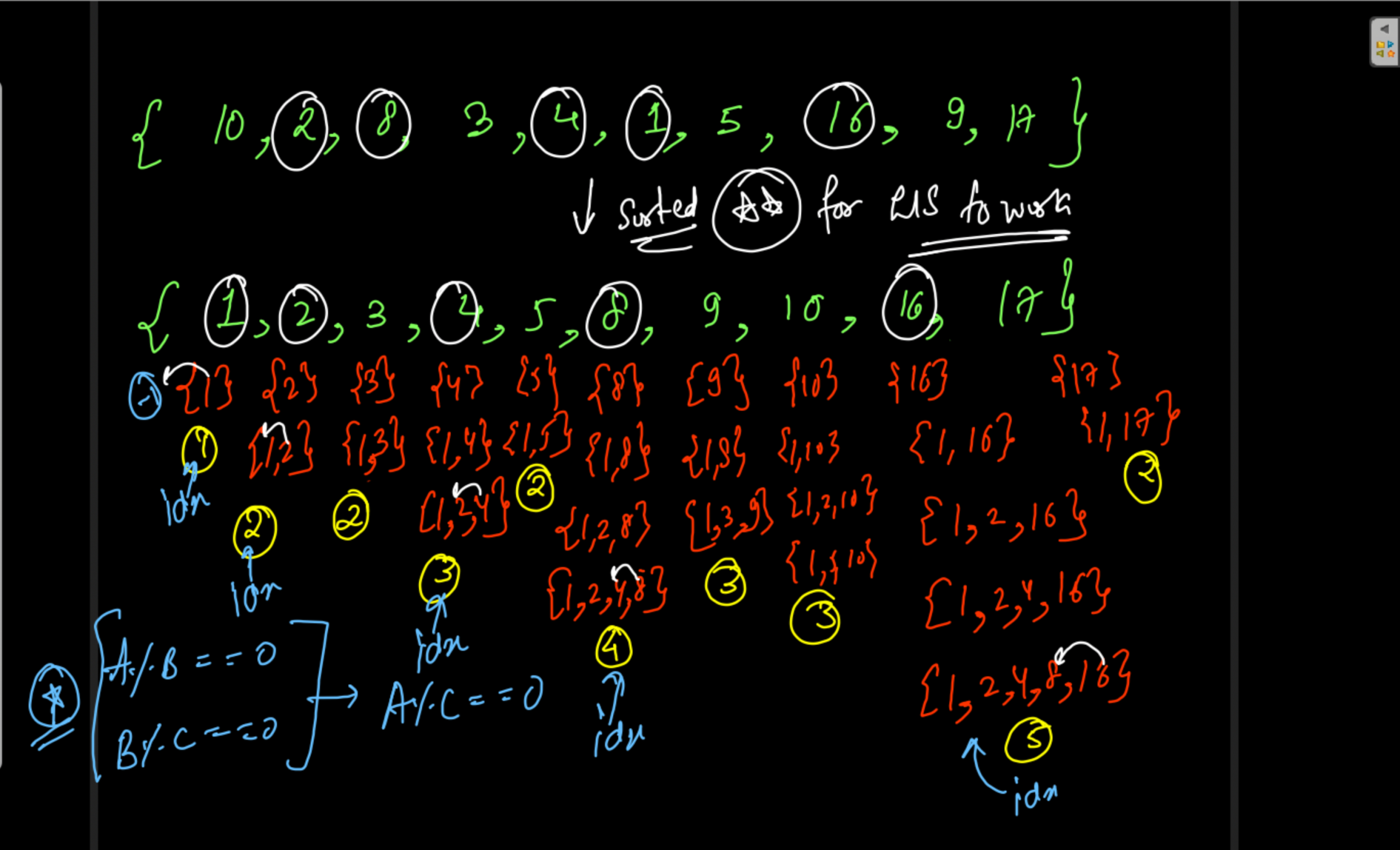
(10,20) 20 % 10 = 0

If any subset is valid, all its permutations are also valid

{10,20,5}
{10,5,20} ③
{5,20,10}
{20,5,10}
{20,10,5}
{5,10,20}

⑤
{2,8,4,1,16}

{1,2,4,8,16}
⑤


{ 10, 2, 8, 3, 4, 1, 5, 16, 9, 17 }
↓ Sorted ** for LIS to work
{ 1, 2, 3, 4, 5, 8, 9, 10, 16, 17 }
{1} {2} {3} {4} {5} {8} {9} {10} {16} {17}
{1,2} {1,3} {1,4} {1,5} {1,8} {1,9} {1,10} {1,16} {1,17}
{1,3,4}
{1,2,8} {1,3,9} {1,2,10}
{1,2,4,8} {1,3,10}
{1,2,16}
{1,2,4,16}
{1,2,4,8,16}
idx
* A%B == 0
B%C == 0
A%C == 0

```java
public List<Integer> largestDivisibleSubset(int[] nums) {
    // Sort the Array (Any Permutation of a valid subset is also valid)
    Arrays.sort(nums);

    int[] dp = new int[nums.length];
    Arrays.fill(dp, 1);

    int[] prev = new int[nums.length];
    Arrays.fill(prev, -1);

    int maxlen = 0, idx = 0;

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

        if(dp[i] > maxlen){
            maxlen = dp[i];
            idx = i;
        }
    }
```

Time → $O(N^2)$

Space → $O(N)$

$O(N^2)$

LIS

```java
    List<Integer> subset = new ArrayList<>();
    while(idx != -1){
        subset.add(nums[idx]);
        idx = prev[idx];
    }

    return subset;
}
```

→ Backtracking → Generating the subset

↳ $O(N)$

Tuesday, Thursday

9:45 PM
to
11:45 PM

Saturday

9 AM - 12 PM

3 PM - 5 PM

Sunday

9 AM - 12 PM

Dynamic Programming Lecture (23)

25th May Wednesday 10 PM - 12 AM

LIS Variations

→ Perfect Squares (279)

→ Arithmetic Slices
- → Subarray (413)
- → Subset (446)

→ Wiggle Subset

→ 2D/3D LIS
- → Russian Doll Envelope (354)
- → Box Stacking (1691)

Perfect Squares

$1 = 1^2$ (1)

$2 = 1^2 + 1^2$ (2)

$3 = 1^2 + 1^2 + 1^2$ (3)

$4 = 2^2$ (1)

$5 = 1^2 + 2^2$ (2)

$6 = 1^2 + 1^2 + 2^2$ (3)

$7 = 1^2 + 1^2 + 1^2 + 2^2$ (4)

$8 = 2^2 + 2^2$ (2)

$9 = 3^2$ (1)

$10 = 3^2 + 1^2$ (2)

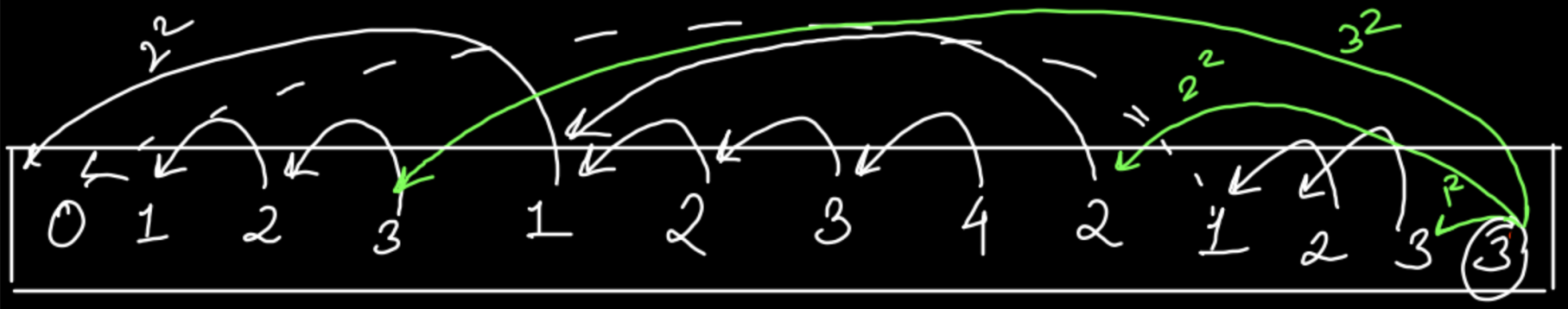
$11 = 1^2 + 1^2 + 3^2$ (3)

$12 = 2^2 + 2^2 + 2^2$ (3)

Greedy fails here

$13 = 2^2 + 3^2$ (2)

$12 = 3^2 + 1^2 + 1^2 + 1^2$

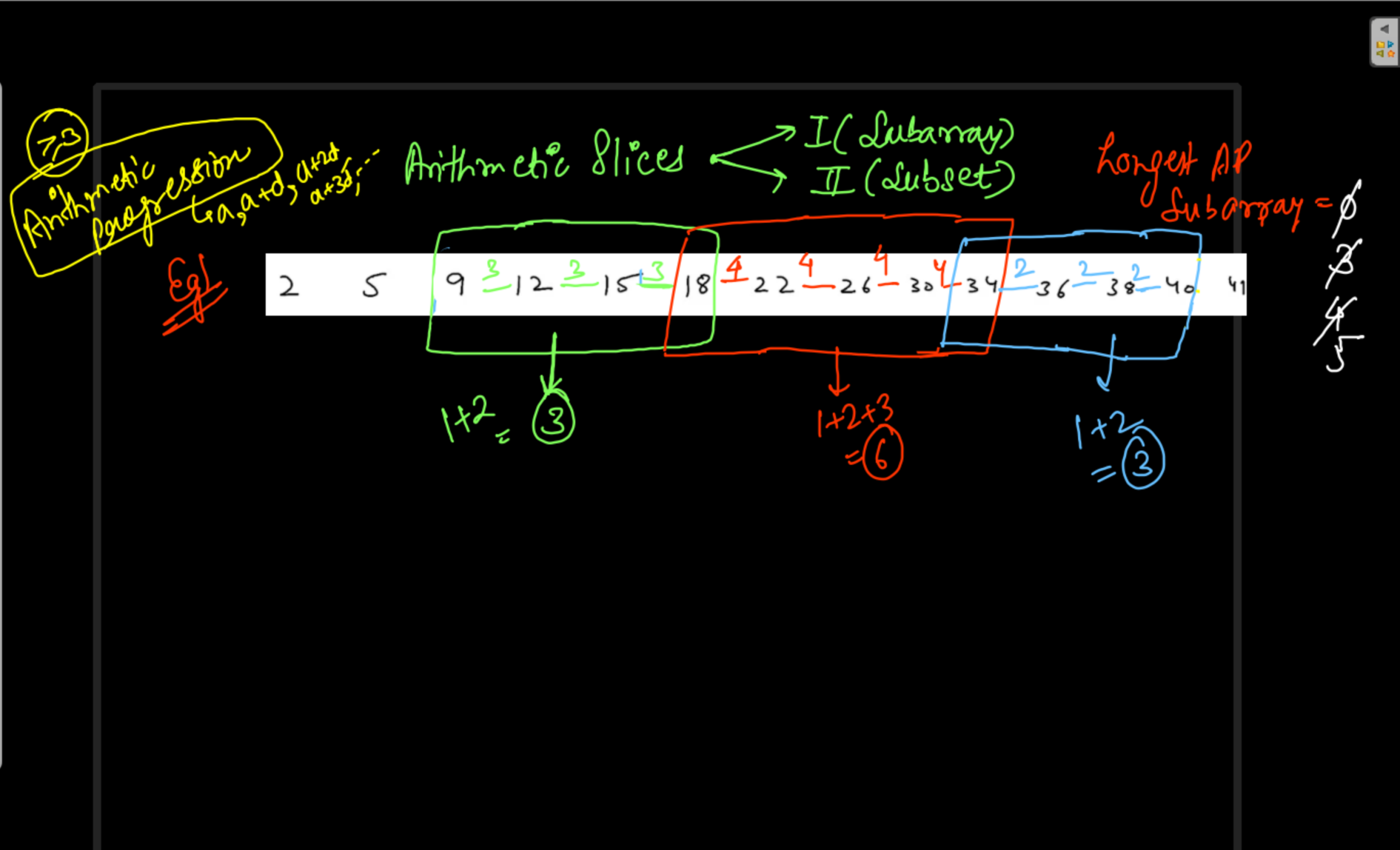


Time → $O(N^2)$

Space → $O(N)$ 1D DP

```
public int numSquares(int n) {
    int[] dp = new int[n + 1];

    for(int i=1; i<=n; i++){
        dp[i] = i;
        for(int j=1; j*j<=i; j++){
            dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
        }
    }

    return dp[n];
}
```

$1^2+1^2+1^2+3^2$
$2^2+2^2+2^2$
$3^2+1^2+1^2+1^2$

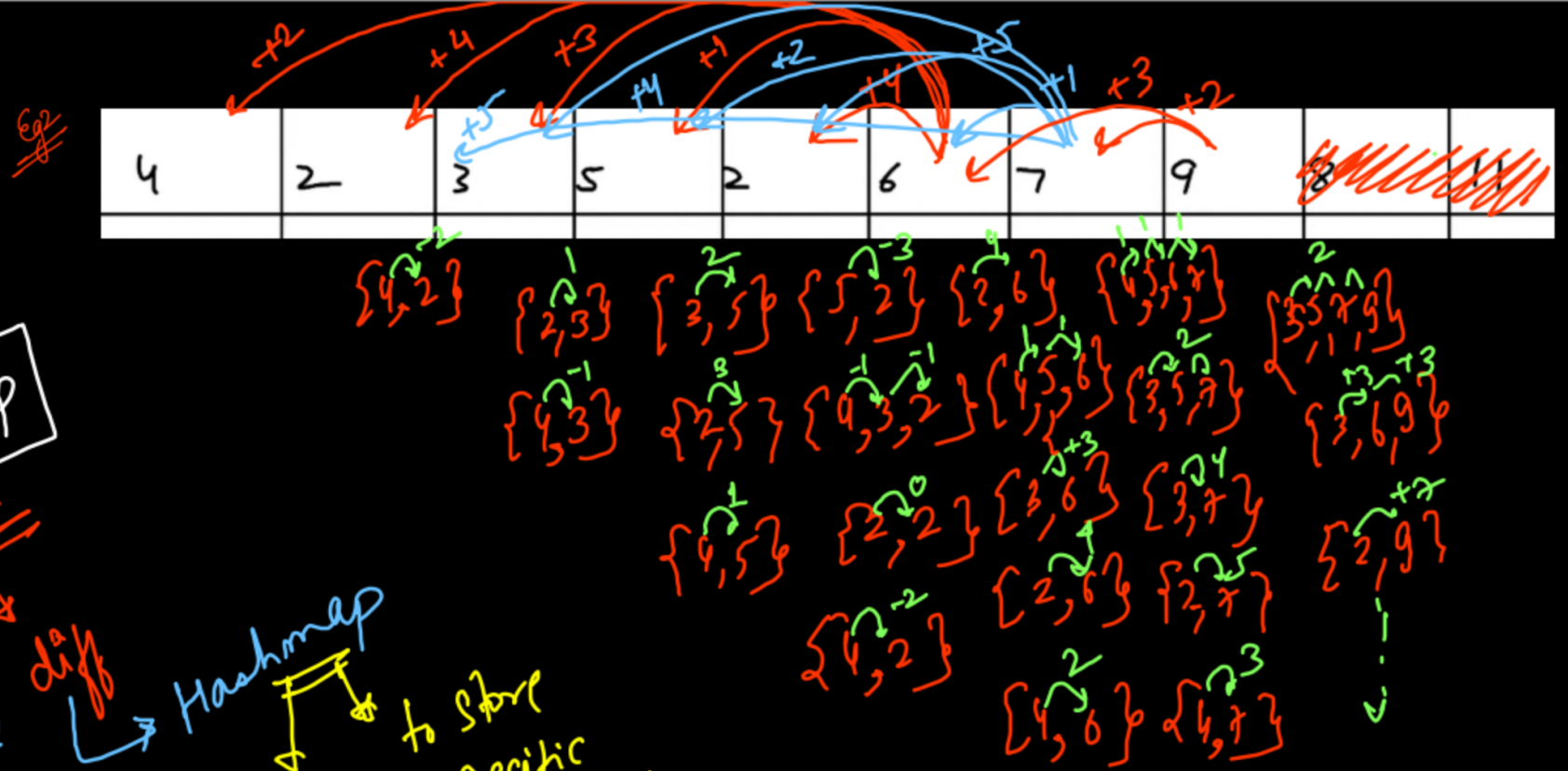

23
Arithmetic progression
[a, a+d, a+2d, a+3d, ...
Arithmetic Slices
I (Subarray)
II (Subset)
Longest AP Subarray = 5
Eg!
2 5 9 12 15 18 22 26 30 34 36 38 40 41
3 3 3 4 4 4 4 2 2 2
1+2 = 3
1+2+3 = 6
1+2 = 3

```java
public int numberOfArithmeticSlices(int[] nums) {
    if(nums.length < 3) return 0;

    int count = 0;
    int currLength = 2;
    int diff = nums[1] - nums[0];

    for(int i=2; i<nums.length; i++){
        int newdiff = nums[i] - nums[i - 1];

        if(newdiff == diff){
            currLength++;
        } else {
            currLength = 2;
            diff = newdiff;
        }

        if(currLength >= 3)
            count = count + (currLength - 2);
    }

    return count;
}
```

Greedy | Kadane

Time → O(N)

Space → O(1)



6g2

2D DP

State

9dx 2 diff

1DDP

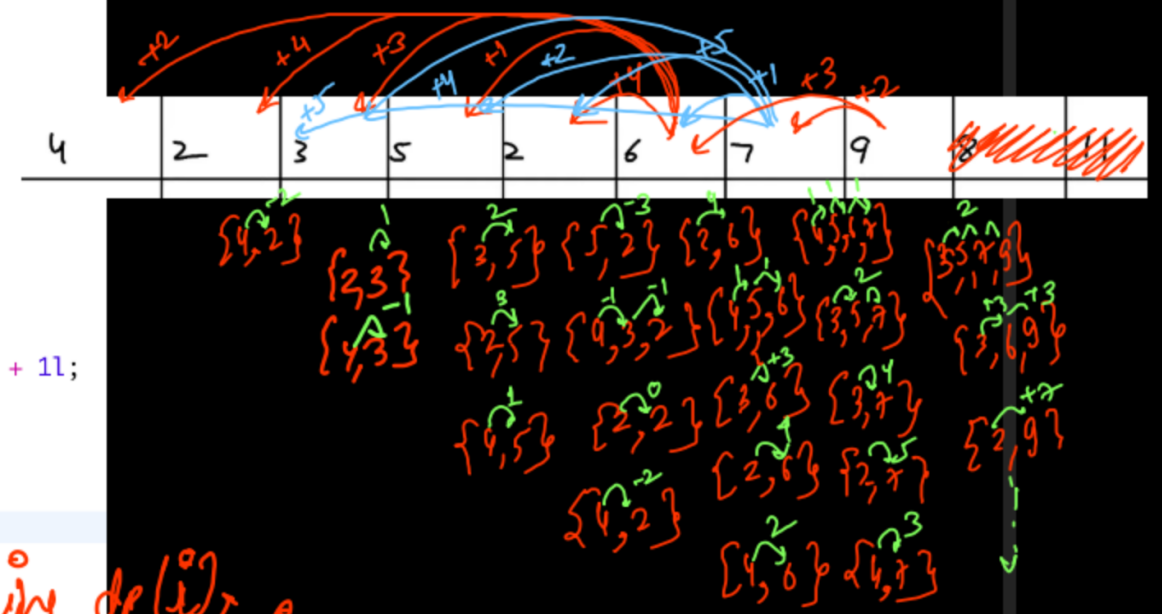
Hashmap

to store -ve cdiff

to store specific cdiff!!

Count = 1 2 4 8 ✗ 8

```java
public int numberOfArithmeticSlices(int[] nums) {
    if(nums.length < 3) return 0;

    // DP[IDX][COMMON DIFF] = COUNT OF AP SUBSETS
    HashMap<Long, Long>[] dp = new HashMap[nums.length];
    for(int i=0; i<nums.length; i++){
        dp[i] = new HashMap<>();
    }

    long count = 0;
    for(int i=1; i<nums.length; i++){
        for(int j=i-1; j>=0; j--){
            long diff = 1l * nums[i] - nums[j];

            long oldVal = dp[i].getOrDefault(diff, 0l);
            long newVal = oldVal + dp[j].getOrDefault(diff, 0l) + 1l;

            dp[i].put(diff, newVal);
            count += dp[j].getOrDefault(diff, 0l);
        }
    }

    return (int)count;
}
```



↳ Instead of adding dp[i], we are adding dp[j] ⇒ we are able to exclude subsets of 2 length

Count = ∅ / 2 - - - - -

LIS 376 Wiggle Subsequence

[1,17,5,10,13,15,10,5,16,8]

{1} {1,5} {1,13}

{1,17} {1,17,5} {1,17,13}

{1,10} {1,17,10} {1,13,5,10}

{1,15} {1,17,15} {1,17,5,15} {1,17,10,15} {1,17,13,15}

{1,17,5,10}

{1,17,10,13,10}

{1,17,5,10,13,15,10}

{1,17,5,10,13} {1,17,10,13}

longest = 1 2 3 4 (5)

{1,17,5,10,5,16,8}

{17,5,15,10,16,8}

{1,17,5,10,13} {17,5,15,10,8}