# Searching & Sorting : Sorting Algorithms

(1) Comparison Based

eg Bubble Sort, Insertion Sort, Selection Sort } → Time ⇒ $O(n^2)$, Space ⇒ $O(1)$

(2) Divide & Conquer Based / Optimized Approach

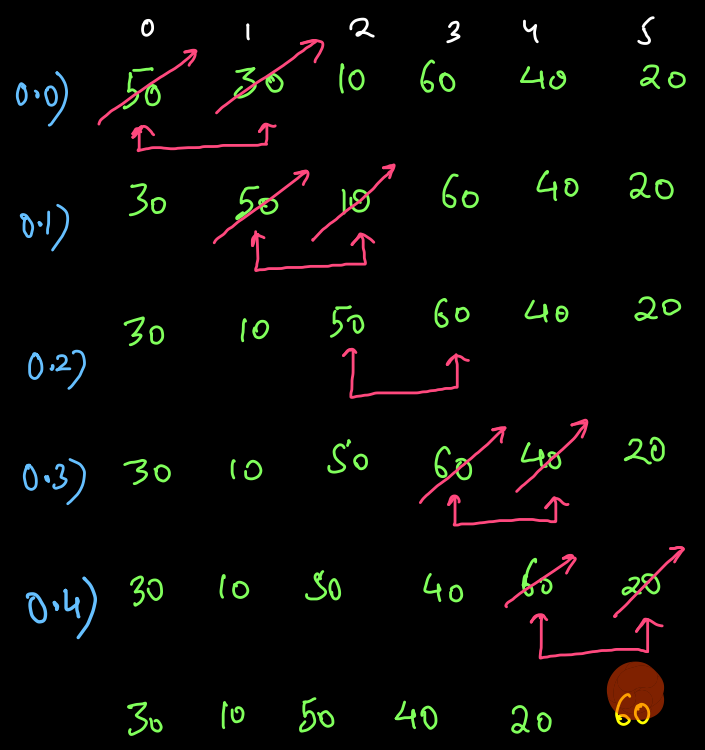
eg merge Sort, Quick Sort, Heap Sort

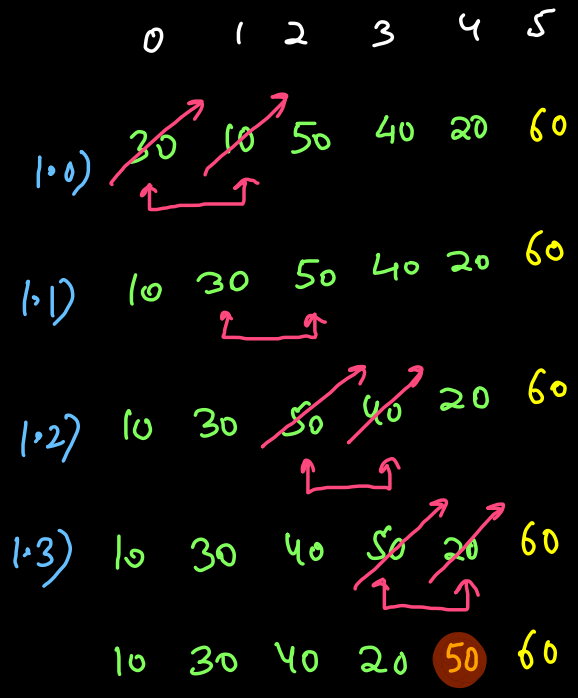
(3) Counting Based

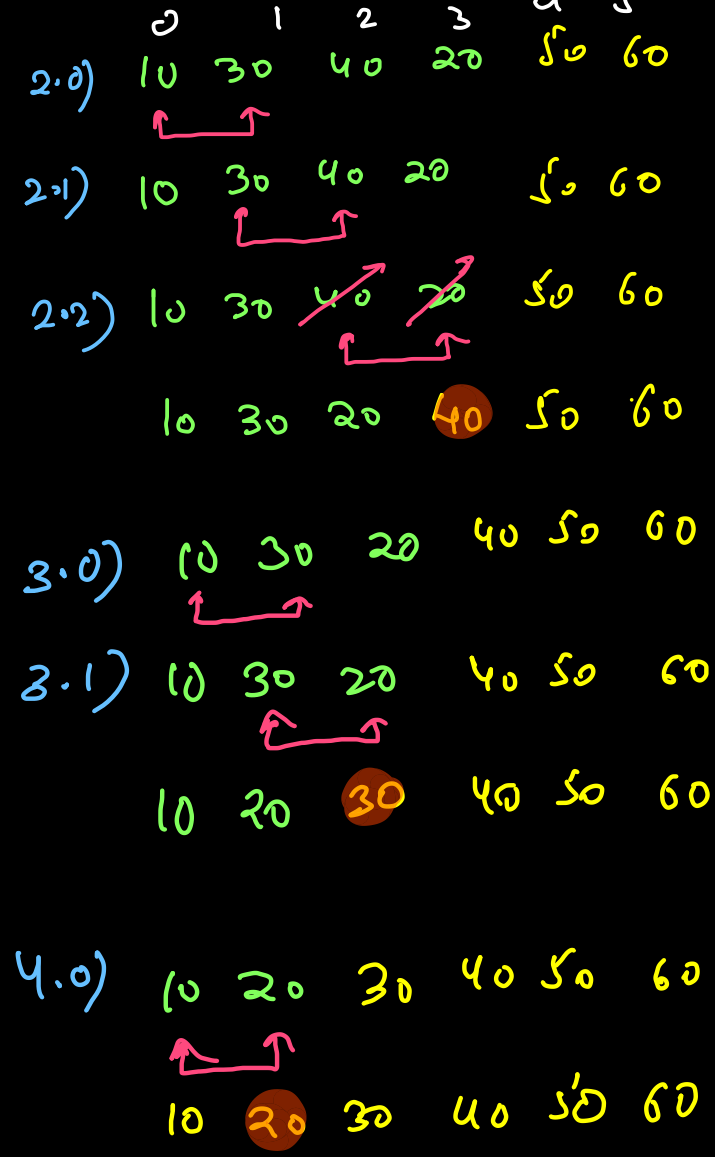
eg Count Sort, Radix Sort, Bucket Sort

# Bubble Sort

After every outer loop iteration, heaviest element settles at last index of unsorted array




Outer loop → (N-1) iteration

```java
public static void swap(int[] arr, int i1, int i2){
    int temp = arr[i1];
    arr[i1] = arr[i2];
    arr[i2] = temp;
}

public static void bubbleSort(int arr[], int n)
{
    for(int i = 0; i < n - 1; i++){
        for(int j = 0; j < n - 1 - i; j++){
            if(arr[j] > arr[j + 1]){
                swap(arr, j, j + 1);
            }
        }
    }
}
```

Best case
↳ Already sorted : $O(n^2)$

Avg Case / Worst case → Reverse Sorted : $O(n^2)$

Time Complexity

$O(n^2)$ quadratic

↳ $N_{max} = 10^4$

```java
public static void swap(int[] arr, int i1, int i2){
    int temp = arr[i1];
    arr[i1] = arr[i2];
    arr[i2] = temp;
}

public static void bubbleSort(int arr[], int n)
{
    for(int i = 0; i < n - 1; i++){
        int count = 0;
        for(int j = 0; j < n - 1 - i; j++){
            if(arr[j] > arr[j + 1]){
                count++;
                swap(arr, j, j + 1);
            }
        }
        if(count == 0) break;
    }
}
```

Best Case Optimization

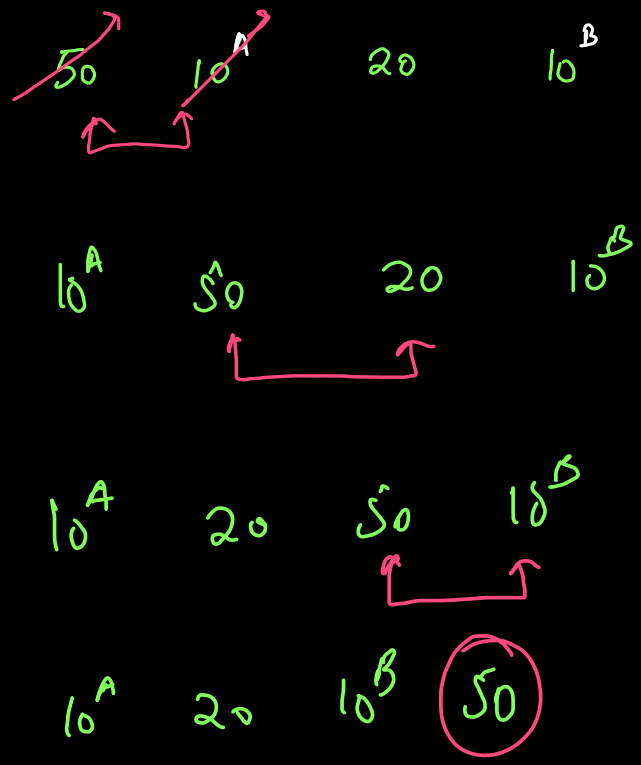
→ If already sorted

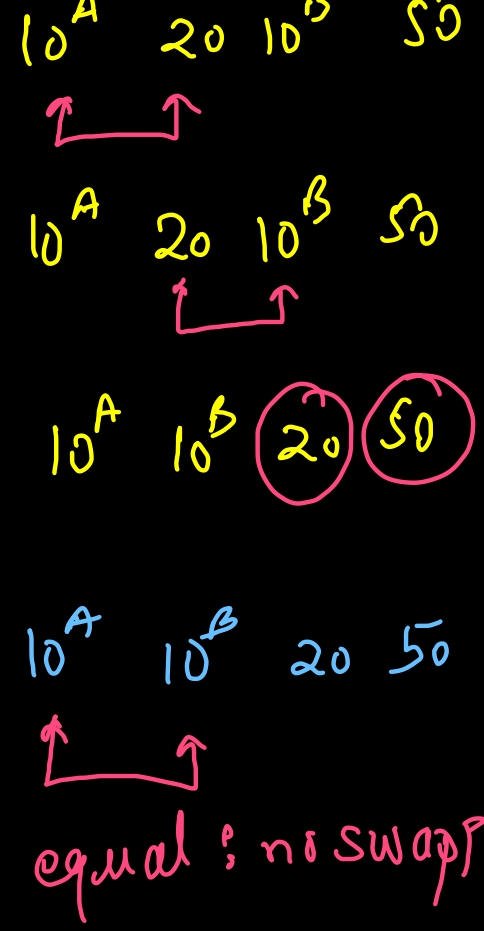
→ stop after no swappings

→ O(n) time in best case

but no change in avg/ worst case

# Stability in Bubble Sort



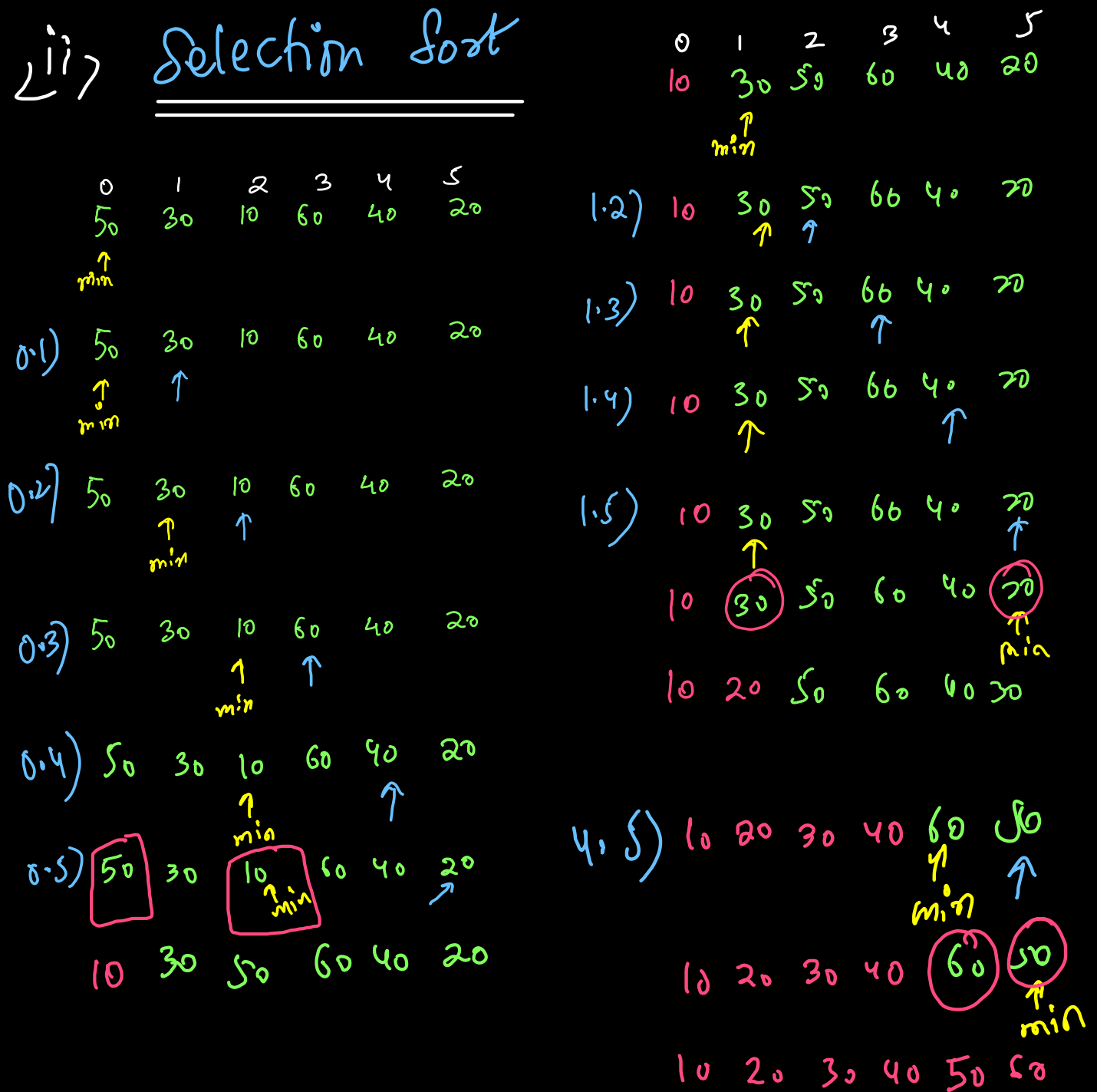

Insert order :

$10^A < 10^B$

Insert order should be preserved

Sorted order ⇒ then bubble sort is stable

# Selection Sort



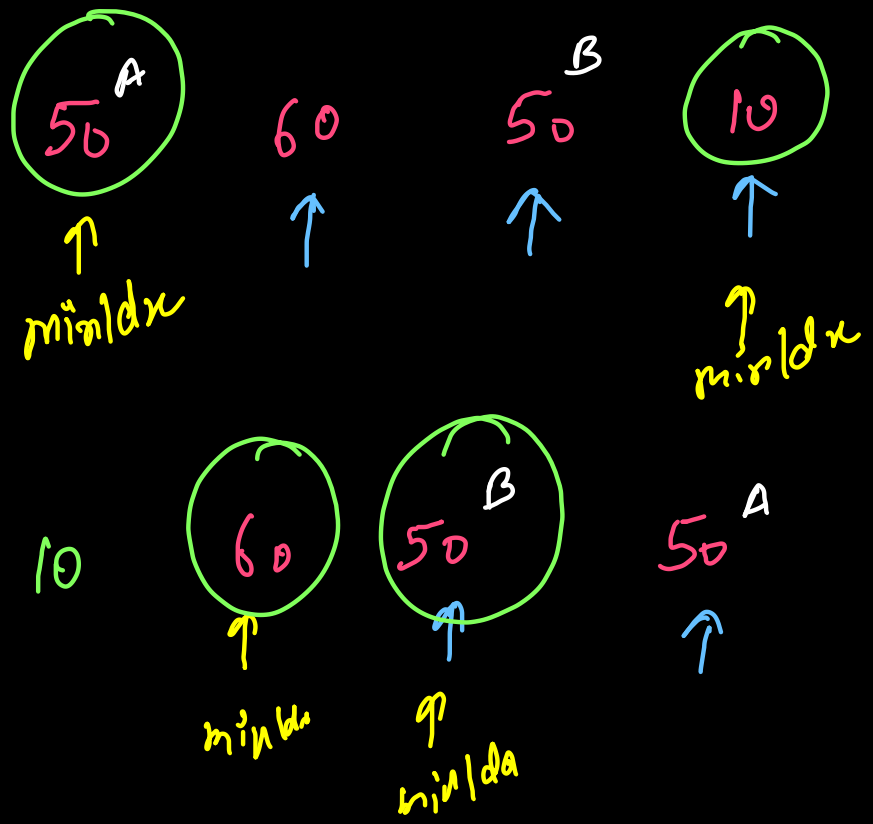
```java
void swap(int[] arr, int i1, int i2){
    int temp = arr[i1];
    arr[i1] = arr[i2];
    arr[i2] = temp;
}
void selectionSort(int arr[], int n)
{
    for(int i = 0; i < n - 1; i++){
        int minIdx = i;
        for(int j = i + 1; j < n; j++){
            if(arr[j] < arr[minIdx]){
                minIdx = j;
            }
        }
        swap(arr, i, minIdx);
    }
}
```

⇒ Best Case
⇒ Avg Case
Worst Case
→ $O(n^2)$ Time Complexity

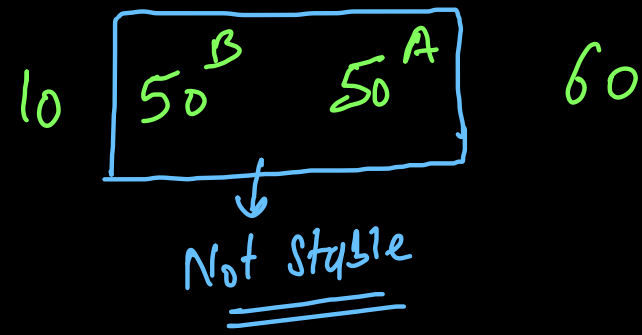
⇒ Space Complexity ⇒ $O(1)$
no extra space

⇒ Not Stable

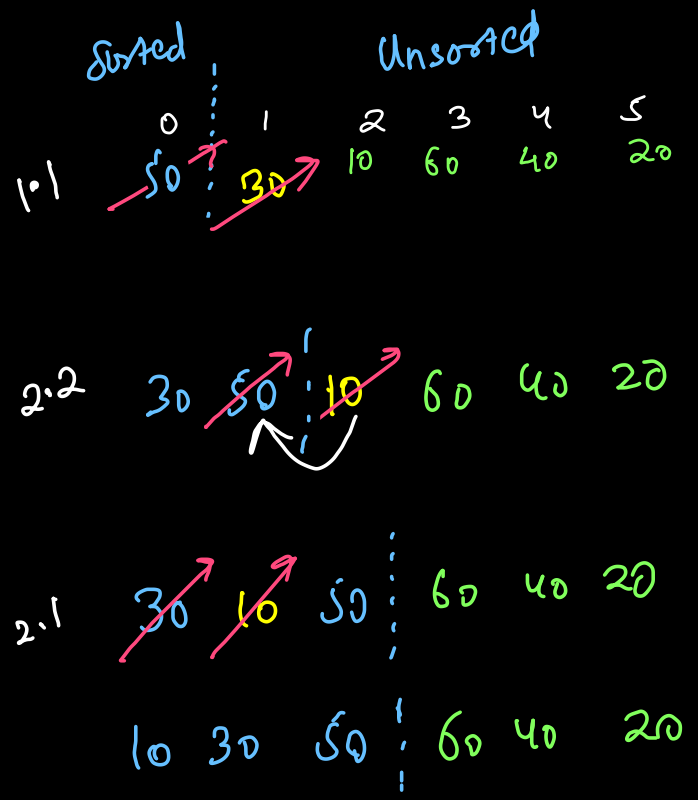
# Stability of Selection Sort

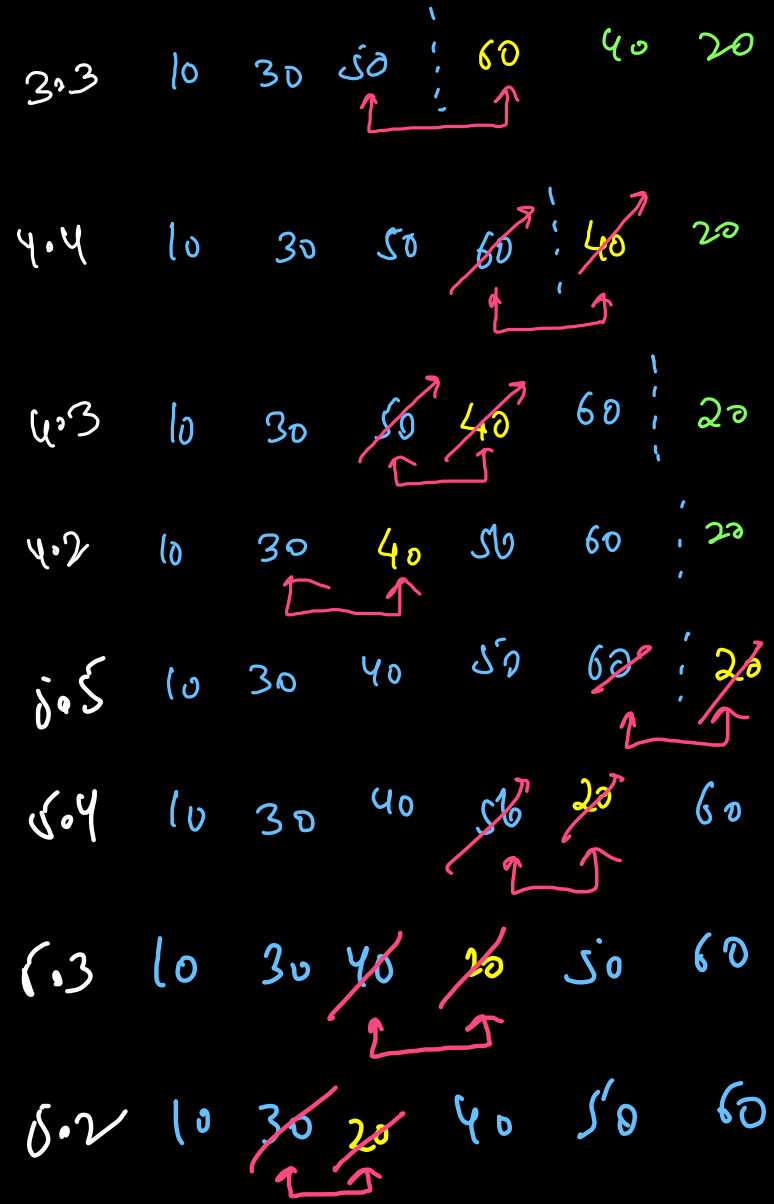


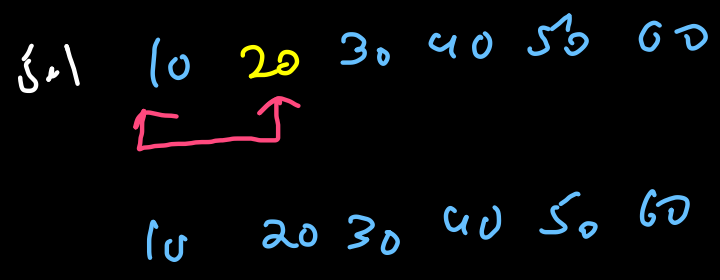




# Insertion sort



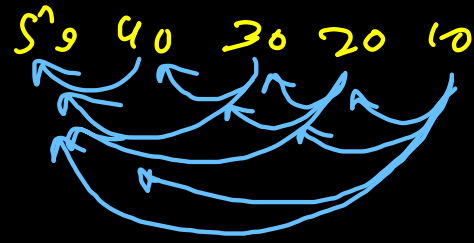



```java
public void swap(int[] arr, int i1, int i2){
        int temp = arr[i1];
        arr[i1] = arr[i2];
        arr[i2] = temp;
}

public void insertionSort(int arr[], int n)
{
    for(int i = 1; i < n; i++){
        for(int j = i; j > 0; j--){
            if(arr[j - 1] > arr[j]){
                swap(arr, j - 1, j);
            } else break;
        }
    }
}
```

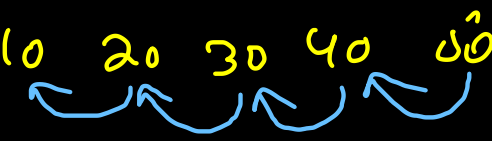
Avg/worst case
↳ $O(n^2)$

Reverse Sorted : 50 40 30 20 10



Best case → O(n)

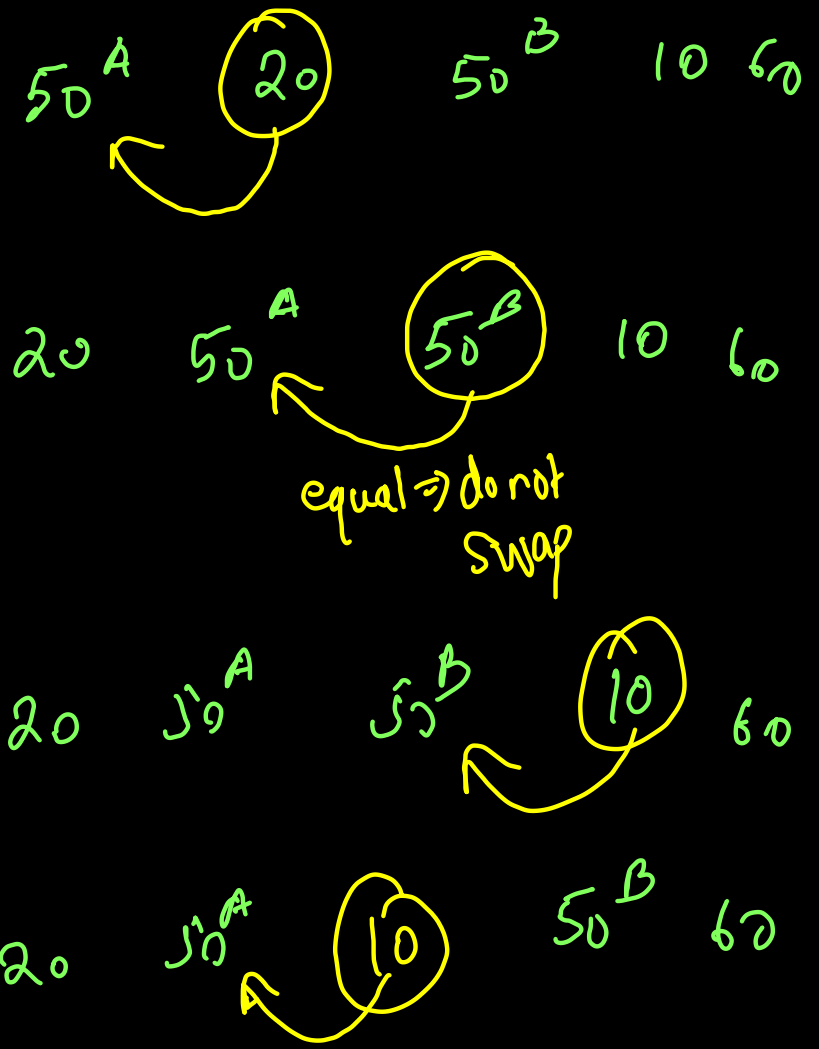
Already sorted : 10 20 30 40 50

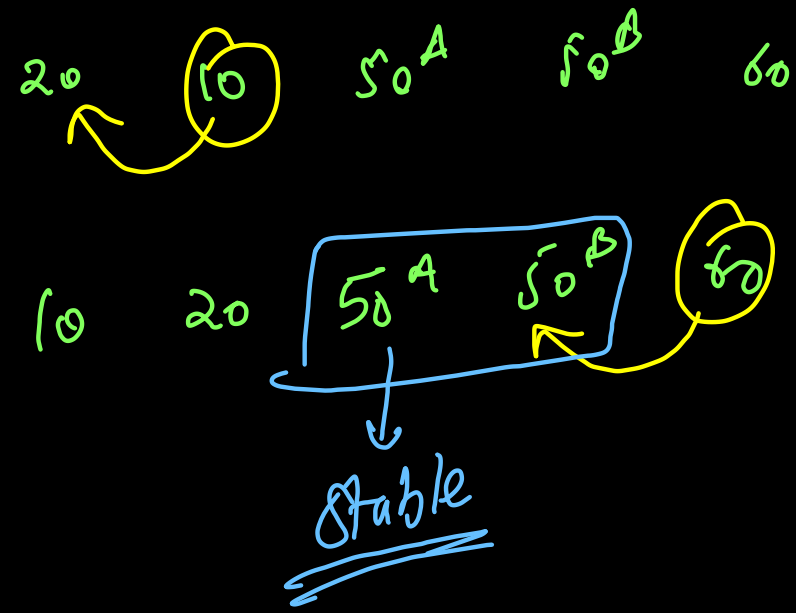


Space Complexity ⇒ O(1) constant extra space (Inplace)

Stable Algo

# Stability of Insertion Sort



| | Best Case | Avg Case | Worst Case | Space Comp | Stability |
|---|---|---|---|---|---|
| Bubble | $O(n)$ using count | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Yes |
| Selectn | $O(n)$ using check Sorted | $O(n^2)$ | $O(n^2)$ | $O(1)$ | No |
| Insertn | $O(n)$ using else break | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Yes |

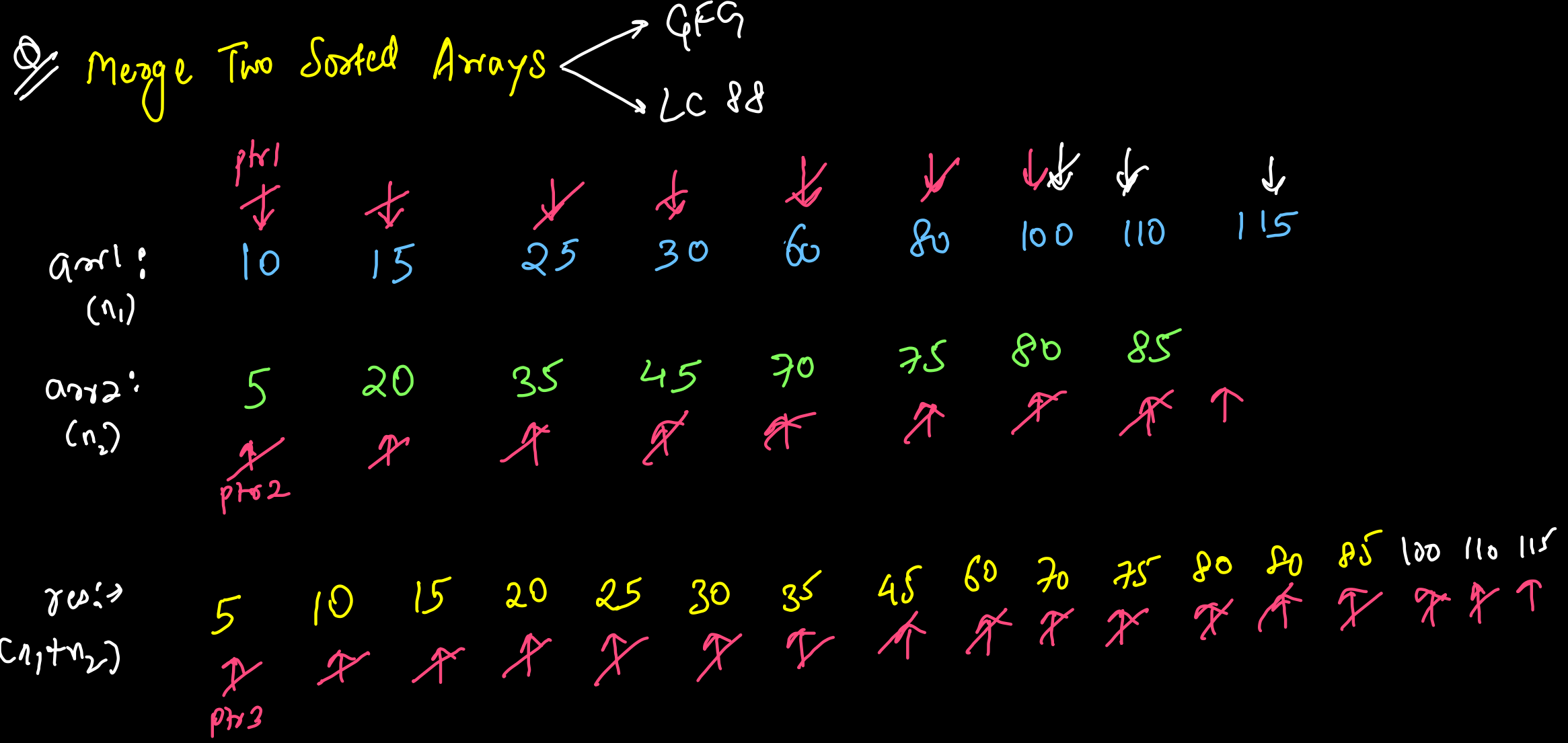
Already Sorted Array

Reverse Sorted Array

Constant Extra Space or Inplace

# Divide & Conquer Based

## Q. Merge Two Sorted Arrays → GFG, LC 88



input

arr1

20 40 50 0 0 0

$n_1 = 3$

arr2

10 30 60

$n_2 = 3$

arr1 :→ 10 20 30 40 50 60

```java
public void merge(int[] nums1, int n1, int[] nums2, int n2) {
    int ptr1 = 0, ptr2 = 0, ptr3 = 0;
    int[] res = new int[n1 + n2];→ extra

    while(ptr1 < n1 && ptr2 < n2){
        if(nums1[ptr1] <= nums2[ptr2]){
            res[ptr3] = nums1[ptr1];
            ptr1++; ptr3++;
        } else {
            res[ptr3] = nums2[ptr2];
            ptr2++; ptr3++;
        }
    }

    while(ptr1 < n1){
        res[ptr3] = nums1[ptr1];
        ptr1++; ptr3++;
    }

    while(ptr2 < n2){
        res[ptr3] = nums2[ptr2];
        ptr2++; ptr3++;
    }

    for(int i = 0; i < res.length; i++){
        nums1[i] = res[i];
    }
}
```

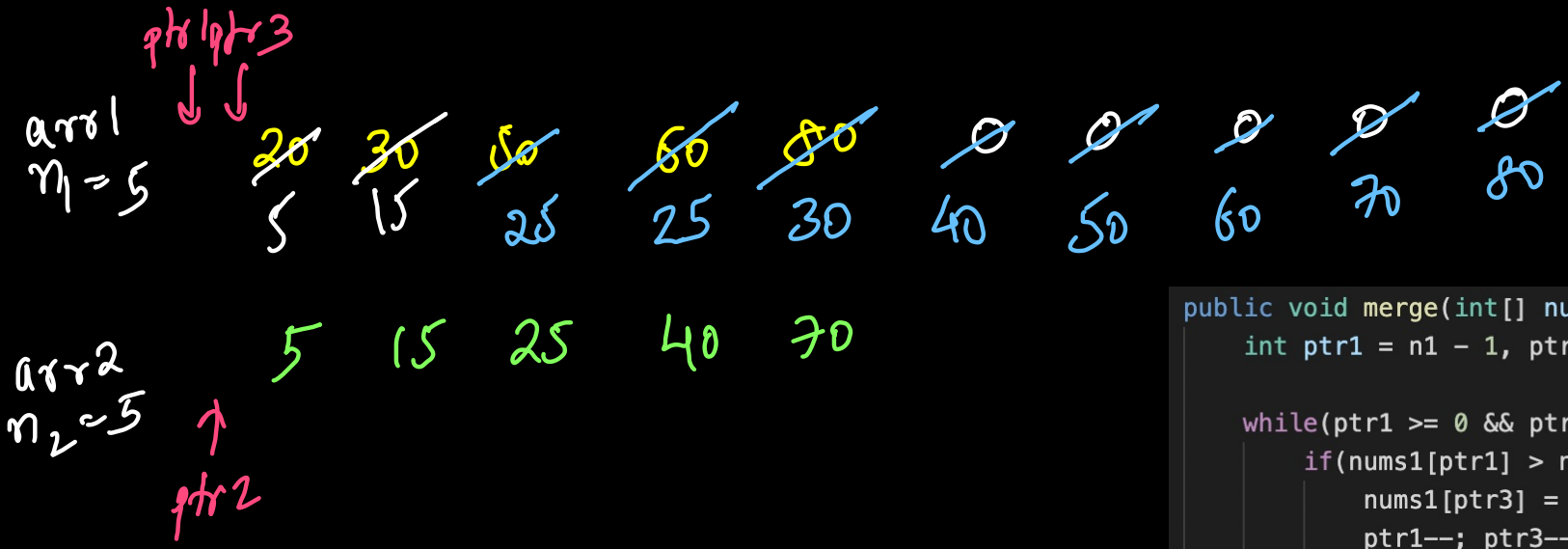
# Two Pointer Technique

Best Case / Avg Case / Worst Case

↳ $O(n_1+n_2)$ Linear

Space Complexity :→ $O(n_1+n_2)$ extra space

Stability : Yes

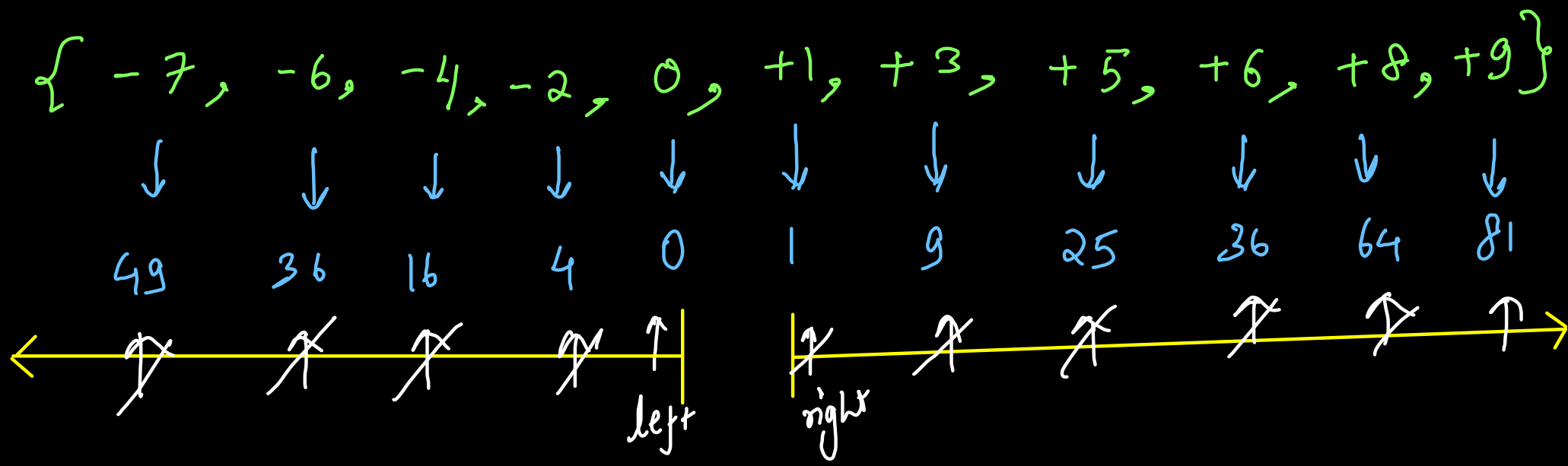



Time → $O(n_1 + n_2)$

Space → $O(1)$ constant extra space

```java
public void merge(int[] nums1, int n1, int[] nums2, int n2) {
    int ptr1 = n1 - 1, ptr2 = n2 - 1, ptr3 = n1 + n2 - 1;

    while(ptr1 >= 0 && ptr2 >= 0){
        if(nums1[ptr1] > nums2[ptr2]){
            nums1[ptr3] = nums1[ptr1];
            ptr1--; ptr3--;
        } else {
            nums1[ptr3] = nums2[ptr2];
            ptr2--; ptr3--;
        }
    }

    while(ptr1 >= 0){
        nums1[ptr3] = nums1[ptr1];
        ptr1--; ptr3--;
    }

    while(ptr2 >= 0){
        nums1[ptr3] = nums2[ptr2];
        ptr2--; ptr3--;
    }
}
```

# Leet 977 Squares of Sorted Array




res :→ 0 1 4 9 16 25 36 36 49 64 81

```java
public int middlePoint(int[] nums){
    for(int i = 0; i < nums.length; i++){
        if(nums[i] >= 0) return i;
    }
    return nums.length;
}
```

Two Pointer Technique

Time → O(n) linear

Space → O(n) extra space

```java
public int[] sortedSquares(int[] nums) {
    int right = middlePoint(nums);
    int left = right - 1, idx = 0;

    int[] res = new int[nums.length];

    while(left >= 0 && right < nums.length){
        if(nums[left] * nums[left] <= nums[right] * nums[right]){
            res[idx] = nums[left] * nums[left];
            idx++; left--;
        } else {
            res[idx] = nums[right] * nums[right];
            idx++; right++;
        }
    }

    while(left >= 0){
        res[idx] = nums[left] * nums[left];
        idx++; left--;
    }

    while(right < nums.length){
        res[idx] = nums[right] * nums[right];
        idx++; right++;
    }

    return res;
}
```

# Intersection → (I) (II)

LC 349 , LC 350

arr1: sorted

| 10 | 20 | 40 | 40 | 60 | 70 | 90 | 90 | 90 |
|---|---|---|---|---|---|---|---|---|
| ptr1 | ptr1 | | | | | | | ↑ |

arr2: sorted

| 20 | 40 | 40 | 70 | 70 | 70 | 90 | 90 | 120 |
|---|---|---|---|---|---|---|---|---|
| ptr2 | | | | | | | | ↑ |

LC 349) 20 40 70 90


LC 350) 20 40 40 70 90 90

```java
public int[] intersection(int[] nums1, int[] nums2) {
    Arrays.sort(nums1);
    Arrays.sort(nums2);

    ArrayList<Integer> common = new ArrayList<>();
    int ptr1 = 0, ptr2 = 0;

    while(ptr1 < nums1.length && ptr2 < nums2.length){
        if(nums1[ptr1] < nums2[ptr2]){
            ptr1++;
        }
        else if(nums1[ptr1] > nums2[ptr2]){
            ptr2++;
        }
        else {
            int val = nums1[ptr1];
            common.add(val);

            while(ptr1 < nums1.length && nums1[ptr1] == val){
                ptr1++;
            }
            while(ptr2 < nums2.length && nums2[ptr2] == val){
                ptr2++;
            }
        }
    }

    int[] res = new int[common.size()];
    for(int idx = 0; idx < res.length; idx++){
        res[idx] = common.get(idx);
    }
    return res;
}
```

$N \log N$

LC 349

$O(n_1 + n_2)$

# lc 350

```java
Arrays.sort(nums1);
Arrays.sort(nums2);

ArrayList<Integer> common = new ArrayList<>();
int ptr1 = 0, ptr2 = 0;

while(ptr1 < nums1.length && ptr2 < nums2.length){
    if(nums1[ptr1] < nums2[ptr2]){
        ptr1++;
    }
    else if(nums1[ptr1] > nums2[ptr2]){
        ptr2++;
    }
    else {
        int val = nums1[ptr1];

        while(ptr1 < nums1.length && nums1[ptr1] == val
        && ptr2 < nums2.length && nums2[ptr2] == val){
            common.add(val);
            ptr1++; ptr2++;
        }
        while(ptr1 < nums1.length && nums1[ptr1] == val){
            ptr1++;
        }
        while(ptr2 < nums2.length && nums2[ptr2] == val){
            ptr2++;
        }
    }
}

int[] res = new int[common.size()];
for(int idx = 0; idx < res.length; idx++){
    res[idx] = common.get(idx);
}
return res;
```

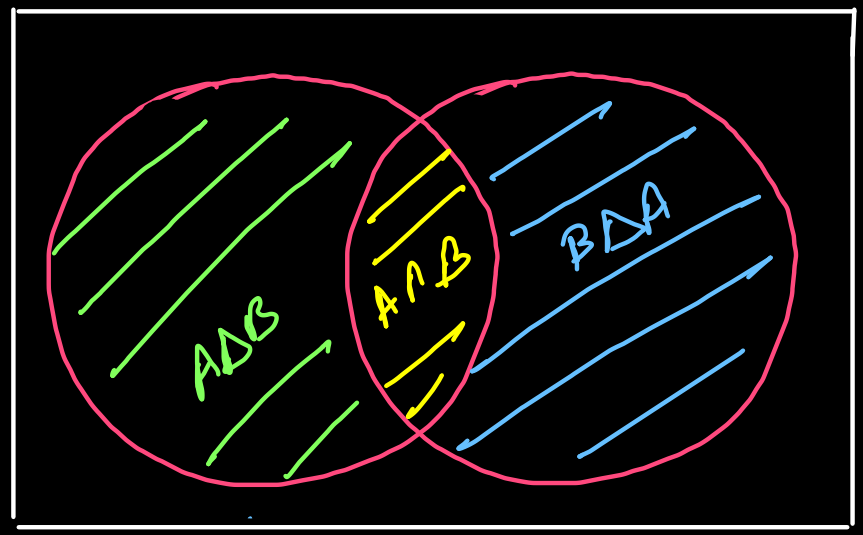
→ NlogN

Common

Uncommon

→ $O(n_1+n_2)$

# Symmetric Difference ( Lc 2215)




arr1: 10 20 40 50 70

arr2: 20 30 60 70 90

A Δ B :→ { 10, 40, 50 }

B Δ A :→ { 30, 60, 90 }

output :→ { { 10, 40, 50 }, { 30, 60, 90 } }

# Merge 2 Sorted Arrays w/o Extra Space

Input

arr1:

| 10 | 30 | 40 | 60 | 80 | 110 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

arr2:

| 20 | 50 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

Sorted

| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 80 | 90 | 100 | 110 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| | | | | | | 6+0 | 6+1 | 6+2 | 6+3 | 6+4 | 6+5 |

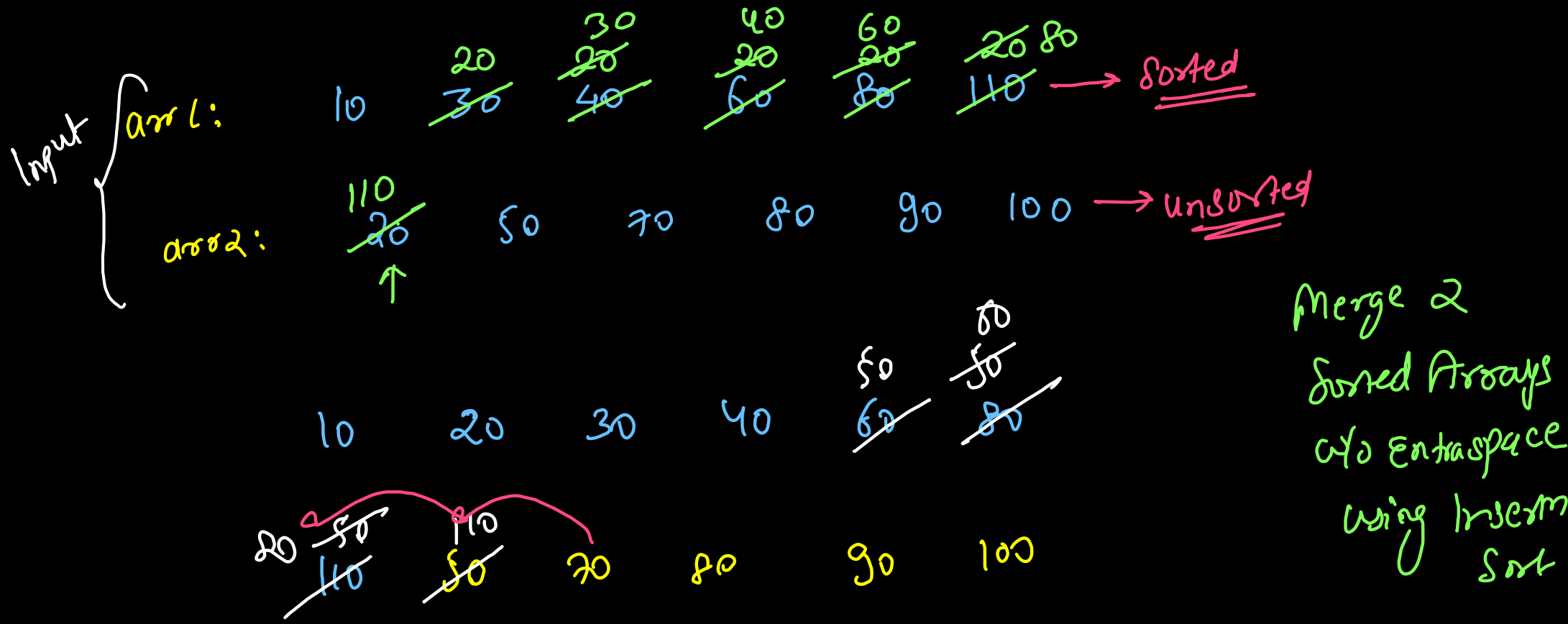
Expected output

arr1:

| 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|

arr2:

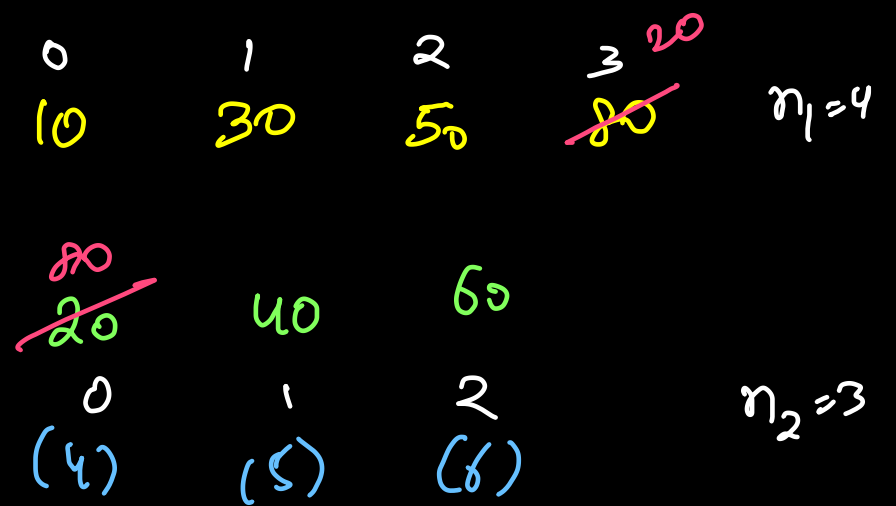
| 70 | 80 | 80 | 90 | 100 | 110 |
|---|---|---|---|---|---|




Merge 2 Sorted Arrays w/o Extra space using Insertion Sort

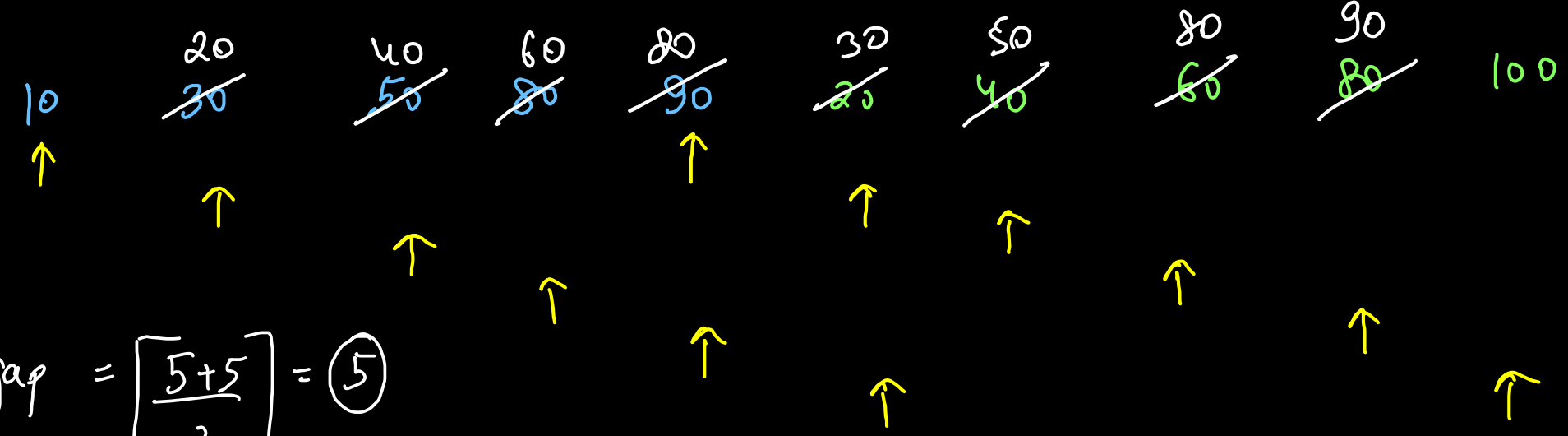
TLE → $O(n^2)$ quadratic  Insertion Sort

```java
public static void swap(long[] arr1, long[] arr2, int i1, int i2){
    long temp1 = get(arr1, arr2, i1);
    long temp2 = get(arr1, arr2, i2);

    set(arr1, arr2, i1, temp2);
    set(arr1, arr2, i2, temp1);
}

public static void set(long[] arr1, long[] arr2, int idx, long val){
    if(idx < arr1.length) arr1[idx] = val;
    else arr2[idx - arr1.length] = val;
}

public static long get(long[] arr1, long[] arr2, int idx){
    if(idx < arr1.length) return arr1[idx];
    return arr2[idx - arr1.length];
}

public static void merge(long arr1[], long arr2[], int n, int m) {
    for(int i = arr1.length; i < arr1.length + arr2.length; i++){
        for(int j = i; j > 0; j--){
            if(get(arr1, arr2, j) < get(arr1, arr2, j - 1)){
                swap(arr1, arr2, j, j - 1);
            } else break;
        }
    }
}
```

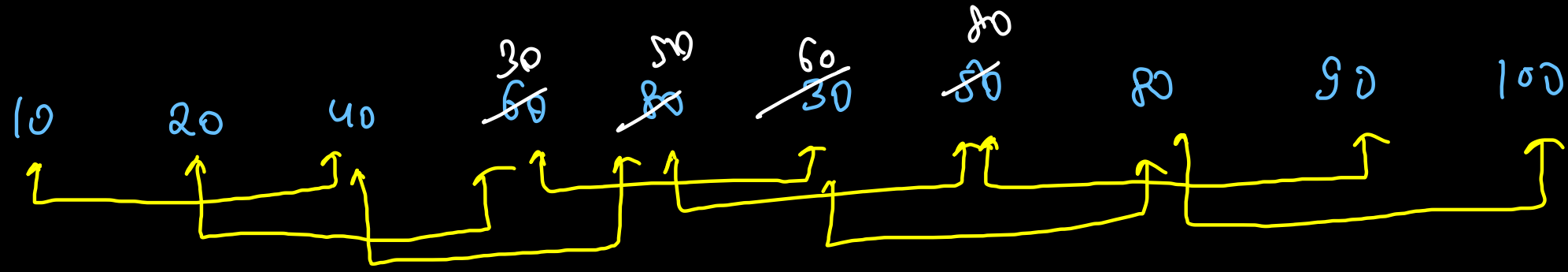


# Merge 2 sorted Arrays w/o extra Space

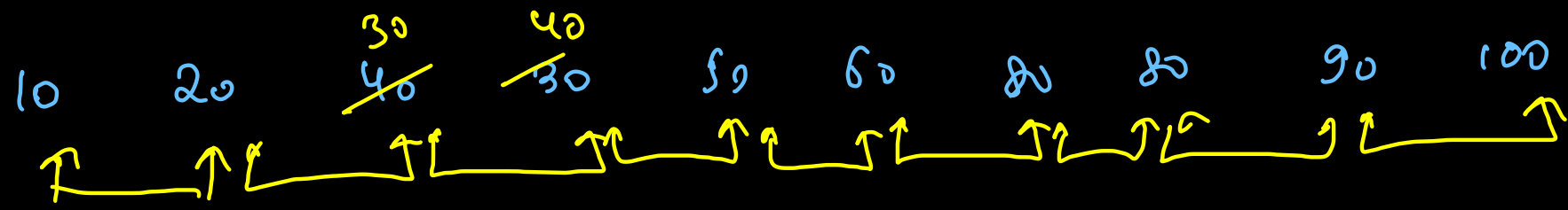
## ↳ using Shell Sort




$$gap = \left\lceil \frac{5+5}{2} \right\rceil = 5$$




$$gap = \left\lceil \frac{5}{2} \right\rceil = 3$$




$gap = \lceil 3/2 \rceil = 2$

10 20 30 40 50 60 70 80 90 100

shell sort → Insert sort → Adjacent → (n-1) iterations

Gap strategy → gap/2

```java
public void swap(int[] arr, int i1, int i2){
    int temp = arr[i1];
    arr[i1] = arr[i2];
    arr[i2] = temp;
}

public void merge(int[] nums1, int m, int[] nums2, int n) {
    // fill second array elements in empty blocks in first array
    for(int idx = 0; idx < nums2.length; idx++){
        nums1[m + idx] = nums2[idx];
    }

    // Shell Sort (Gap Strategy)
    for(double gap = (m + n) / 2.0; gap >= 1.0; gap = gap / 2.0){
        int ceil = (int)Math.ceil(gap);
        gap = ceil;

        for(int i = 0; i + ceil < m + n; i++){
            if(nums1[i] > nums1[i + ceil]){
                swap(nums1, i, i + ceil);
            }
        }
    }
}
```

→ logn

→ n

ShellSort
TC ⇒ O(nlogn)
SC ⇒ O(1)

① Two Pointer
Time ⇒ O(m+n)
space ⇒ O(m+n)

② Shell Sort
Time ⇒ O(nlogn)
Space ⇒ O(1)

③ Insertion Sort
Time ⇒ O($n^2$)
space ⇒ O(1)

$n = 5$ floor(5) = 5 ceil(5) = 5

$n = -5$ floor(-5) = -5 ceil(-5) = -5

$n = 5.5$ floor(5.5) = 5 ceil(5.5) = 6

$n = -5.5$ floor(-5.5) = -6 ceil(-5.5) = -4

$n = 6$

gap = 3 → 3/2 = 1.5 → 1.5/2 = 0.75

3 2 1

## Amazon

SDE-1 2 years

SDE-II 2 years

## Salesforce

AMTS - 1 year

Lower SDE-I

MTS - 1 year

Higher SDE-I & Lower SDE-II

SMTS

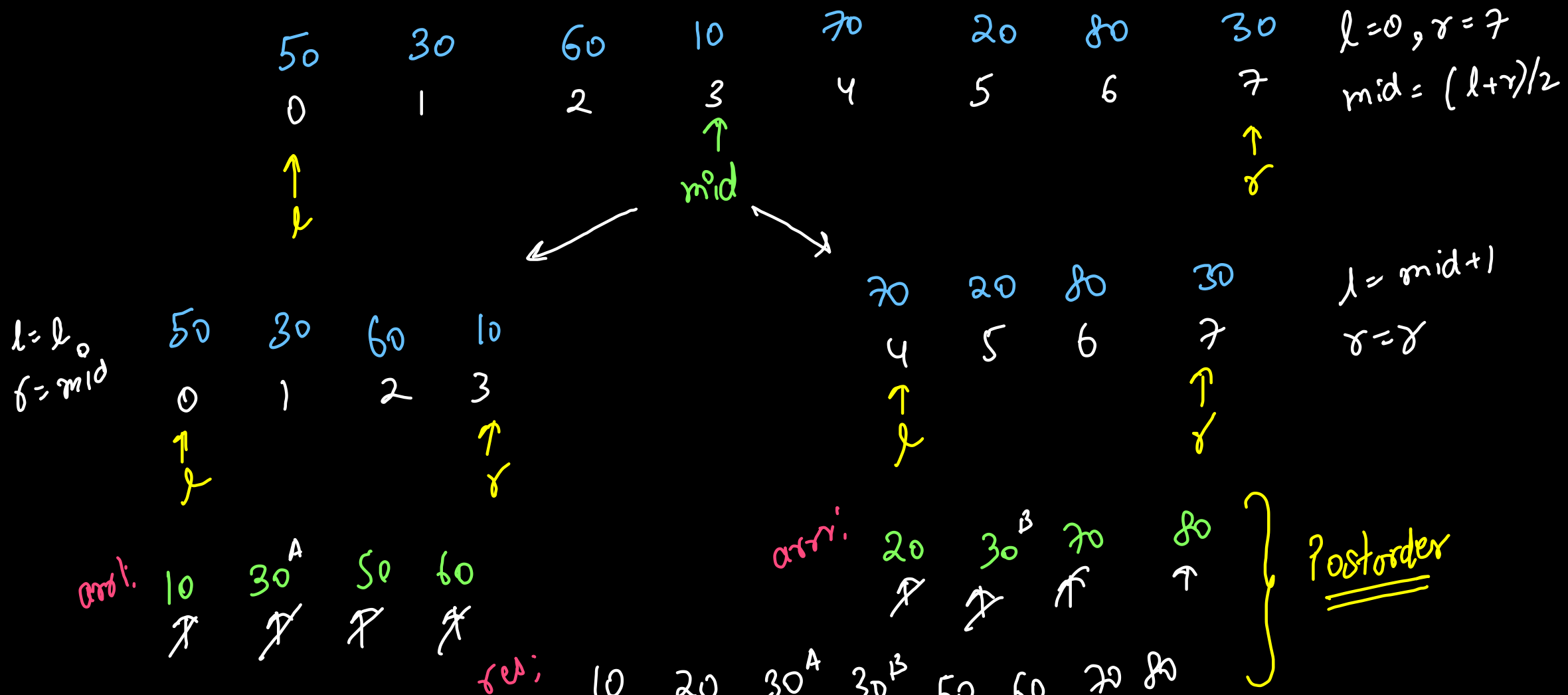
Higher SDE-II

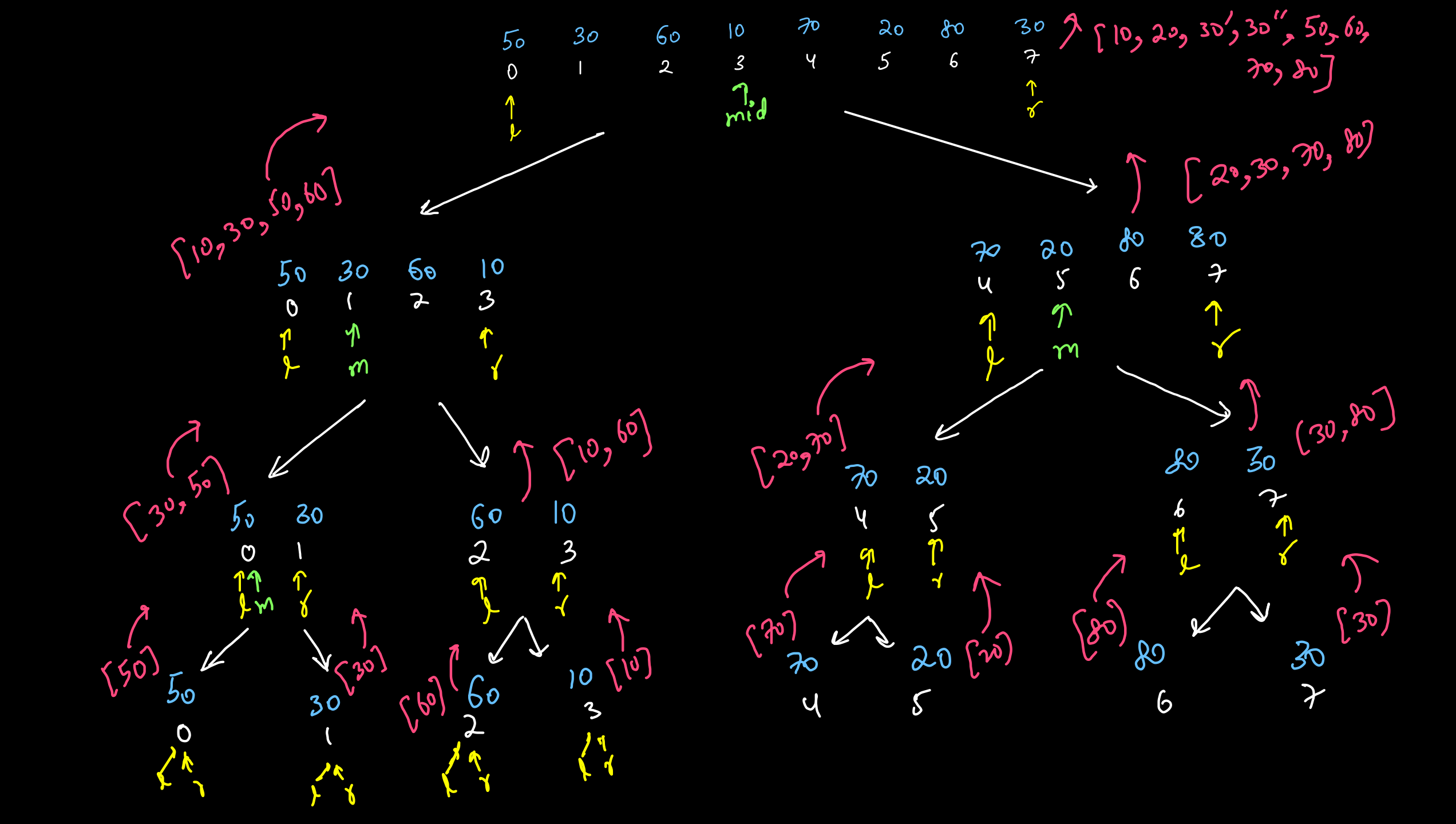
## Google / Netflix / Walmart

SWE-II ⇒ 2 year

SWE-III

# Merge sort :→ Divide & Conquer




50 30 60 10 70 20 80 30
0 1 2 3 4 5 6 7
l mid r
[10, 20, 30', 30'', 50, 60, 70, 80]
[10, 30, 50, 60]
50 30 60 10
0 1 2 3
l m r
[30, 50]
50 30
0 1
l m r
[50]
50
0
l r
[30]
30
1
l r
[10, 60]
60 10
2 3
l r
[60]
60
2
l r
[10]
10
3
l r
70 20 80 30
4 5 6 7
l m r
[20, 30, 70, 80]
[20, 70]
70 20
4 5
l r
[70]
70
4
l r
[20]
20
5
l r
80 30
6 7
l r
[80]
80
6
l r
[30, 80]
30
7
l r
[30]
30
7
l r

```java
// Recursive Code (Divide & Conquer)
public int[] mergeSort(int[] nums, int l, int r){
    if(l == r){
        int[] ans = new int[1];
        ans[0] = nums[l];
        return ans;
    }

    int mid = (l + r) / 2;
    int[] left = mergeSort(nums, l, mid);
    int[] right = mergeSort(nums, mid + 1, r);
    return merge(left, right);
}
```

→ divide (Preorder)

→ conquer (Postorder)

```java
public int[] sortArray(int[] nums) {
    return mergeSort(nums, 0, nums.length - 1);
}
```

Recursion Depth → $O(\log_2 n)$

Merge :- $O(n)$

} Time ⇒ $n \log n$

Best, Avg, Worst

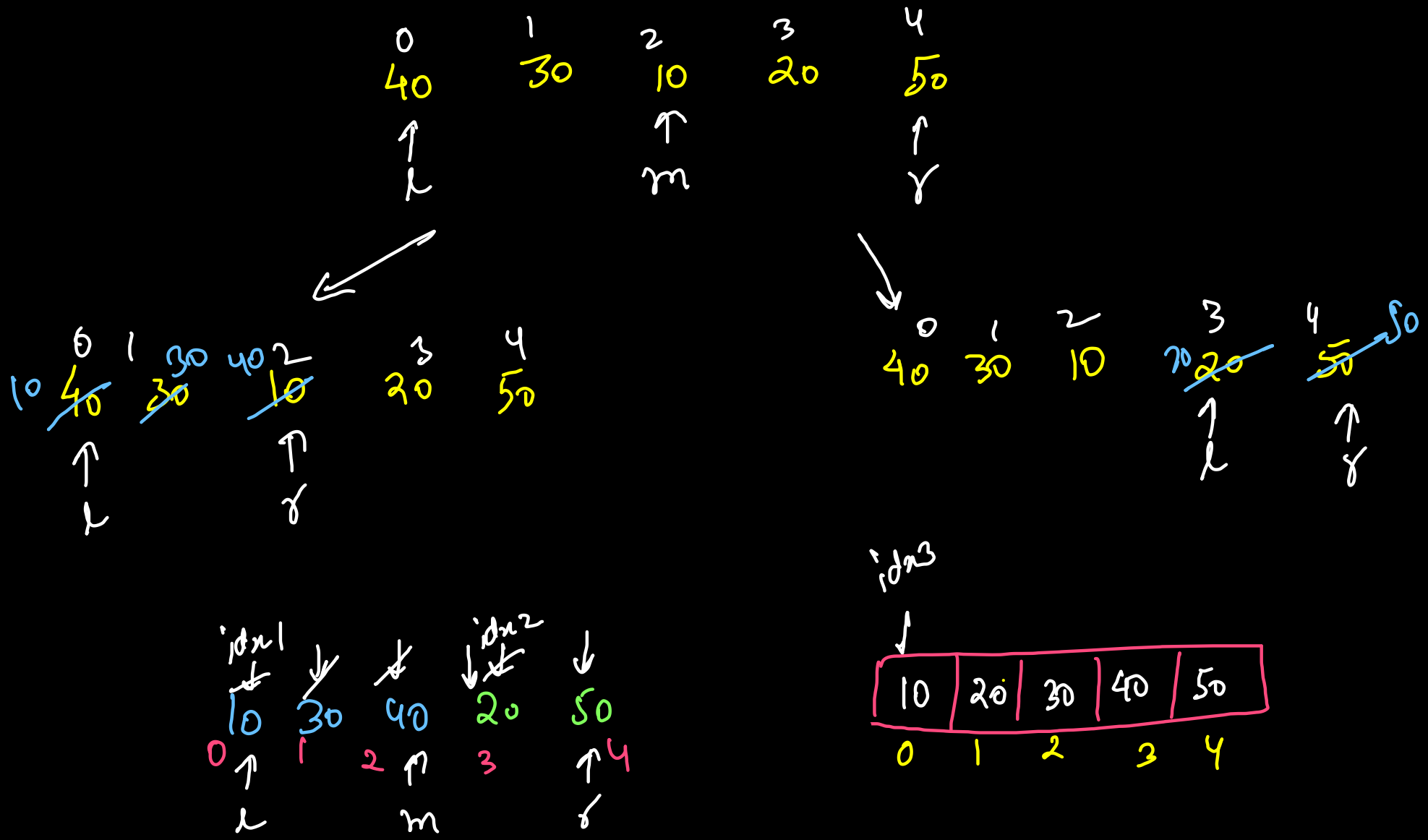
```java
public int[] merge(int[] arr1, int[] arr2){
    int[] res = new int[arr1.length + arr2.length];
    int idx1 = 0, idx2 = 0, idx3 = 0;

    while(idx1 < arr1.length && idx2 < arr2.length){
        if(arr1[idx1] <= arr2[idx2]){
            res[idx3] = arr1[idx1];
            idx1++; idx3++;
        } else {
            res[idx3] = arr2[idx2];
            idx2++; idx3++;
        }
    }

    while(idx1 < arr1.length){
        res[idx3] = arr1[idx1];
        idx1++; idx3++;
    }

    while(idx2 < arr2.length){
        res[idx3] = arr2[idx2];
        idx2++; idx3++;
    }

    return res;
}
```

* Stable ✓

* Inplace ✗

Space ⇒ $O(n)$ extra space

# MergeSort (with void return type)



[10, 15]

including both

10, 11, 12, 13, 14, 15

(6)

15 - 10 + 1

r - l + 1

(10, 15]

only r included

11, 12, 13, 14, 15

(5)

15 - 10

r - l

[10, 15)

only l included

10, 11, 12, 13, 14

(5)

15 - 10

r - l

(10, 15)

excluding both

11, 12, 13, 14

(4)

15 - 10 - 1

r - l - 1

```java
public void merge(int[] arr, int left, int mid, int right){
    int[] res = new int[right - left + 1]; ← extra space
    int idx1 = left, idx2 = mid + 1, idx3 = 0;

    while(idx1 <= mid && idx2 <= right){
        if(arr[idx1] <= arr[idx2]){
            res[idx3] = arr[idx1];
            idx1++; idx3++;
        } else {
            res[idx3] = arr[idx2];
            idx2++; idx3++;
        }
    }

    while(idx1 <= mid){
        res[idx3] = arr[idx1];
        idx1++; idx3++;
    }

    while(idx2 <= right){
        res[idx3] = arr[idx2];
        idx2++; idx3++;
    }

    for(int idx = left; idx <= right; idx++){
        arr[idx] = res[idx - left];
    }
}
```

Time → NlogN

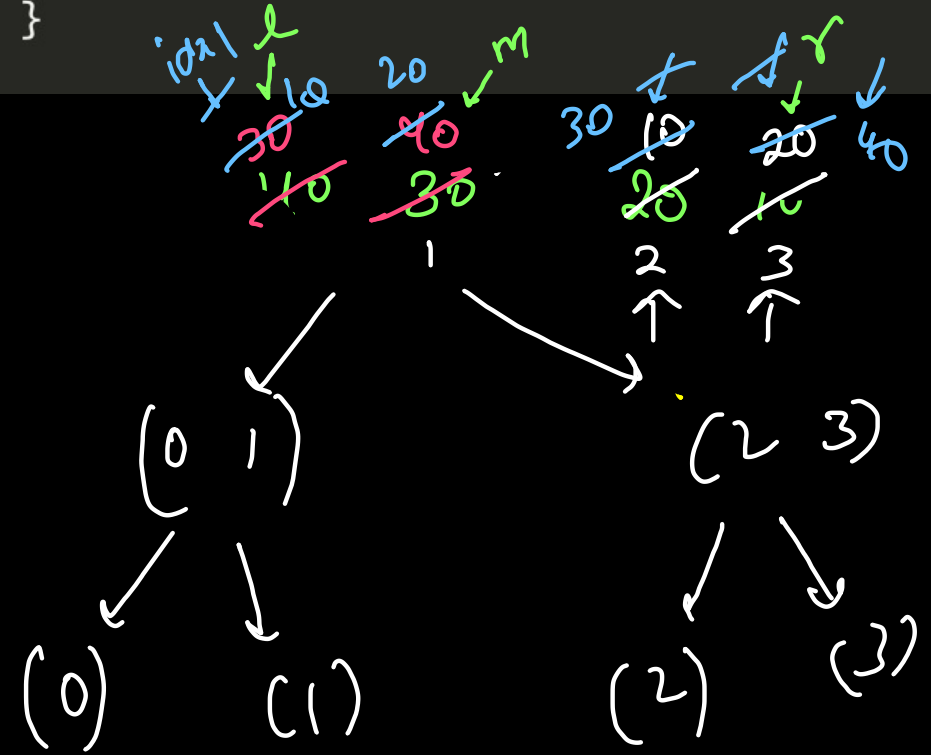
Space → N

extra space

* implace ✗

⇒ stable ✓

```java
void mergeSort(int nums[], int l, int r)
{
    if(l == r) return;

    int mid = (l + r) / 2;
    mergeSort(nums, l, mid);
    mergeSort(nums, mid + 1, r);
    merge(nums, l, mid, r);
}
```



# Inversion Count

| 30 | 90 | 40 | 80 | 70 | 20 | 10 | 50 | 60 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Inversion Pairs $(i, j)$

$i < j$

$arr[i] > arr[j]$

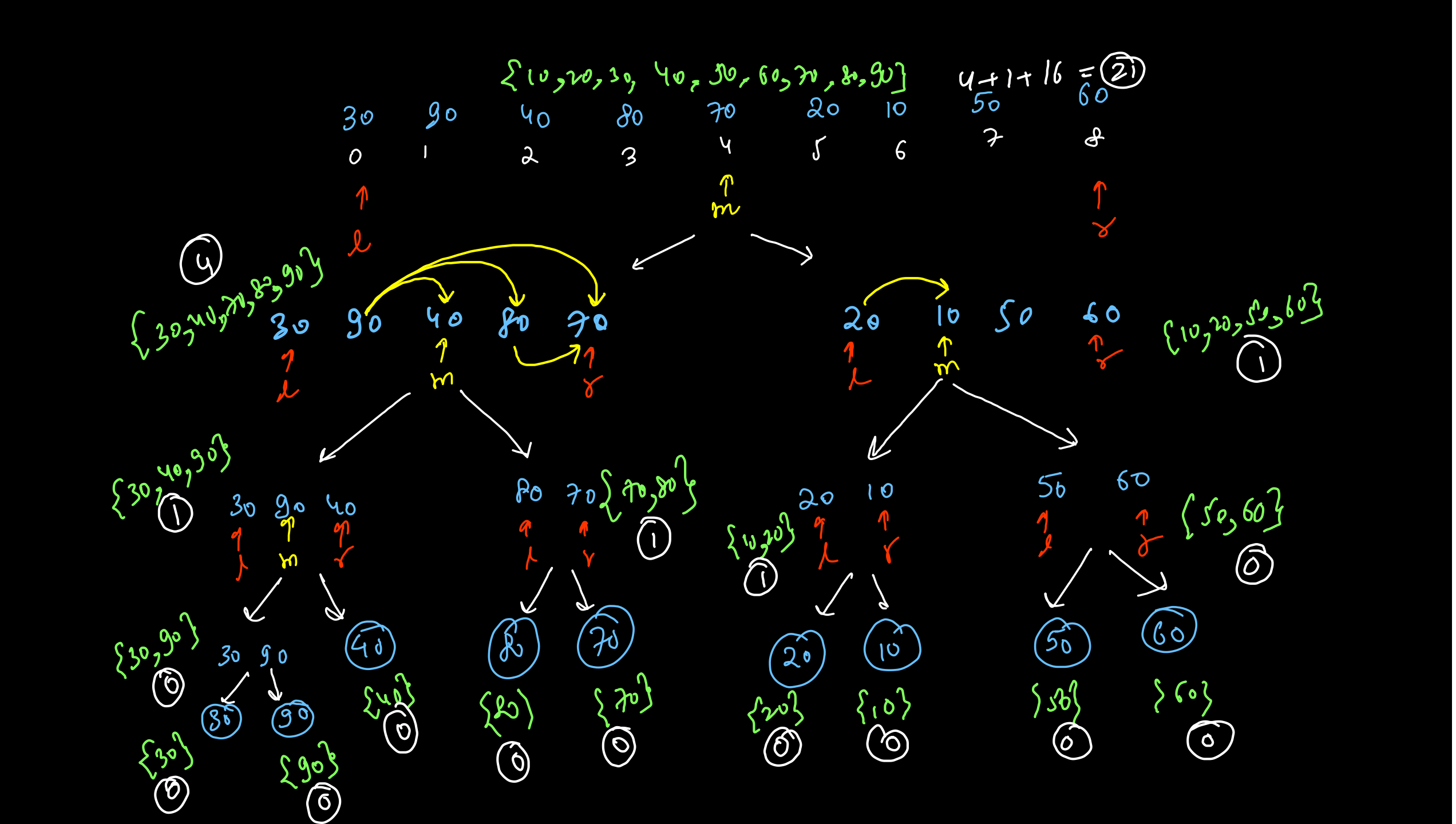
(30, 20)
(30, 10)
(90, 40)
(90, 80)
(90, 70)
(90, 20)
(90, 10)
(90, 50)
(90, 60)
(40, 20)
(40, 10)
(80, 70)
(80, 20)
(80, 10)
(80, 50)
(80, 60)
(70, 20)
(70, 10)
(70, 50)
(70, 60)
(20, 10)

# Brute force

↓

$O(n^2)$ quadratic

↳ $N_{max}$ : $10^4$


{10,20,30,40,50,60,70,80,90}
4+1+16 = 21
30 90 40 80 70 20 10 50 60
0 1 2 3 4 5 6 7 8
{30,40,70,80,90}
{10,20,50,60}
{30,40,90}
{70,80}
{10,20}
{50,60}
{30,90}
{30}
{90}
{40}
{80}
{70}
{20}
{10}
{50}
{60}

| 30 | 40 | 70 | 80 | 90 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| p1 | p1 | p1 | | |

| 10 | 20 | 50 | 60 | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | |
| p2 | p2 | p2 | p2 | p2 |

| ⑤ | ⑤ | ③ | ③ |
|---|---|---|---|
| (30,10) | (30,20) | (70,50) | (70,60) |
| (40,10) | (40,20) | (80,50) | (80,60) |
| (70,10) | (70,20) | (90,50) | (90,60) |
| (80,10) | (80,20) | | |
| (90,10) | (90,20) | | |

res:

| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |
|---|---|---|---|---|---|---|---|---|
| p3 | p3 | p3 | p3 | p3 | p3 | p3 | | |

count = 0+5+5+3+3

= (16)

```java
static long merge(long[] arr, int left, int mid, int right){
    long[] res = new long[right - left + 1];
    int idx1 = left, idx2 = mid + 1, idx3 = 0;

    long invCount = 0;
    while(idx1 <= mid && idx2 <= right){
        if(arr[idx1] <= arr[idx2]){
            res[idx3] = arr[idx1];
            idx1++; idx3++;
        } else {
            invCount += (mid - idx1 + 1);
            res[idx3] = arr[idx2];
            idx2++; idx3++;
        }
    }

    while(idx1 <= mid){
        res[idx3] = arr[idx1];
        idx1++; idx3++;
    }

    while(idx2 <= right){
        res[idx3] = arr[idx2];
        idx2++; idx3++;
    }

    for(int idx = left; idx <= right; idx++){
        arr[idx] = res[idx - left];
    }

    return invCount;
}
```

```java
static long mergeSort(long nums[], int l, int r)
{
    if(l == r) return 0;

    int mid = (l + r) / 2;
    long left = mergeSort(nums, l, mid);
    long right = mergeSort(nums, mid + 1, r);
    long curr = merge(nums, l, mid, r);
    return left + curr + right;
}

static long inversionCount(long nums[], long N) {
    return mergeSort(nums, 0, nums.length - 1);
}
```

Applicatn of mergeSort

Time → O(nlogn)

Space → O(n)

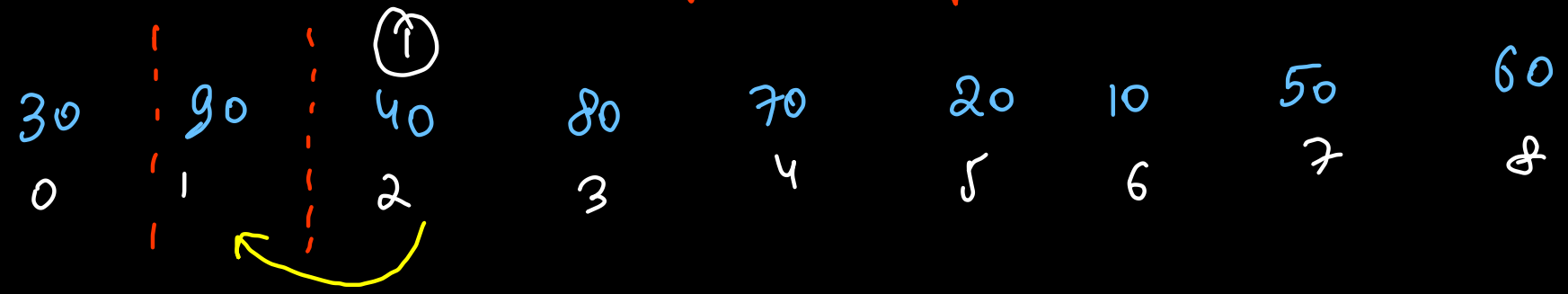
1st Nov

# OOPS

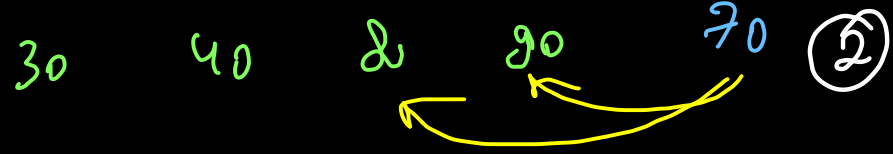
- geeksforgeeks → hackerrank (practical)
  - ↳ Netflix
- Scratch to Advanced
- 6 days (M-S : 7:30 pm to 9:30 pm)

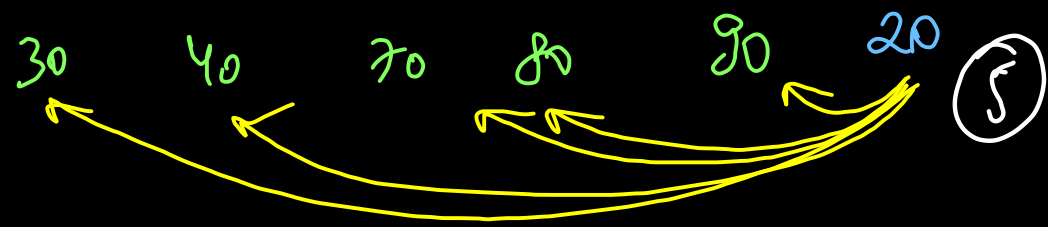
# Sort :→ Adjacent Swap

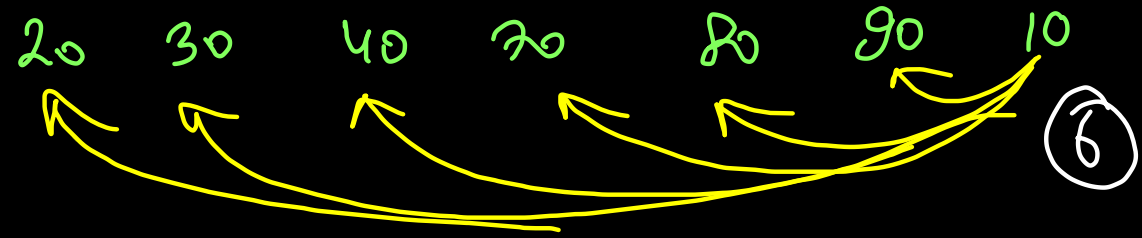
min swaps

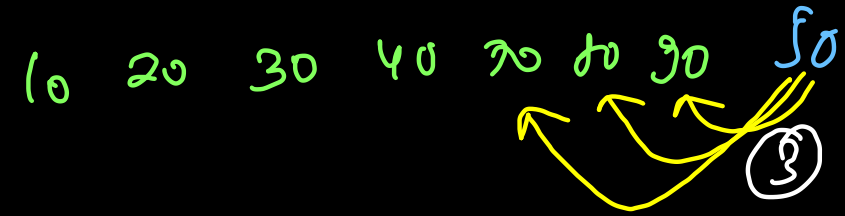


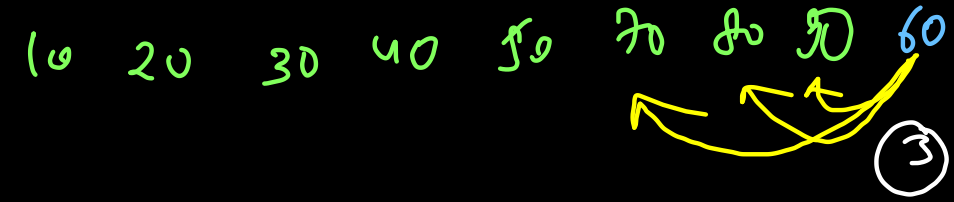




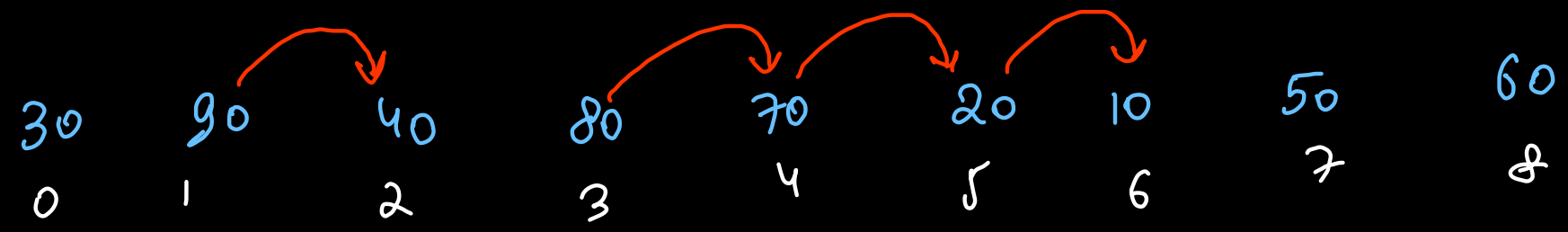




Inversion Pairs :→ 21

```java
int merge(int[] arr, int left, int mid, int righ
    int[] res = new int[right - left + 1];
    int idx1 = left, idx2 = mid + 1, idx3 = 0;

    int invCount = 0;
    while (idx1 <= mid && idx2 <= right) {
        if (arr[idx1] <= arr[idx2]) {
            res[idx3] = arr[idx1];
            idx1++;
            idx3++;
        } else {
            invCount += (mid - idx1 + 1);
            res[idx3] = arr[idx2];
            idx2++;
            idx3++;
        }
    }

    while (idx1 <= mid) {
        res[idx3] = arr[idx1];
        idx1++;
        idx3++;
    }

    while (idx2 <= right) {
        res[idx3] = arr[idx2];
        idx2++;
        idx3++;
    }

    for (int idx = left; idx <= right; idx++) {
        arr[idx] = res[idx - left];
    }

    return invCount;
}
```

```java
int mergeSort(int nums[], int l, int r) {
    if (l == r)
        return 0;
    int mid = (l + r) / 2;
    int left = mergeSort(nums, l, mid);
    int right = mergeSort(nums, mid + 1, r);
    int curr = merge(nums, l, mid, r);
    return left + curr + right;
}

int countSwaps(int nums[], int n) {
    return mergeSort(nums, l: 0, nums.length - 1);
}
```

# Global & Local Inversions




You are given an integer array `nums` of length `n` which represents a permutation of all the integers in the range `[0, n - 1]`.

The number of **global inversions** is the number of the different pairs `(i, j)` where:

- `0 <= i < j < n`
- `nums[i] > nums[j]`

The number of **local inversions** is the number of indices `i` where:

- `0 <= i < n - 1`
- `nums[i] > nums[i + 1]`

Return `true` *if the number of* ***global inversions*** *is equal to the number of* ***local inversions***.

inversion pairs

inversion pairs with adjacent indices

mergeSort
- Time → nlogn
- Space → O(n)

Brute force ; linear search
- Time → O(n)
- Space → O(1)

```java
public boolean isIdealPermutation(int[] nums) {
    int local = 0;
    for(int idx = 0; idx < nums.length - 1; idx++){
        if(nums[idx] > nums[idx + 1])
            local++;
    }

    int global = mergeSort(nums, 0, nums.length - 1);

    System.out.println(global + " " + local);
    return global == local;
}
```

Leetcode 775

Time ⇒ O(nlogn)

Space ⇒ O(n)

# Reverse Pairs

(LC 493 - Hard)

19 25 2 9 6 40 12

$(i, j)$

$i < j$

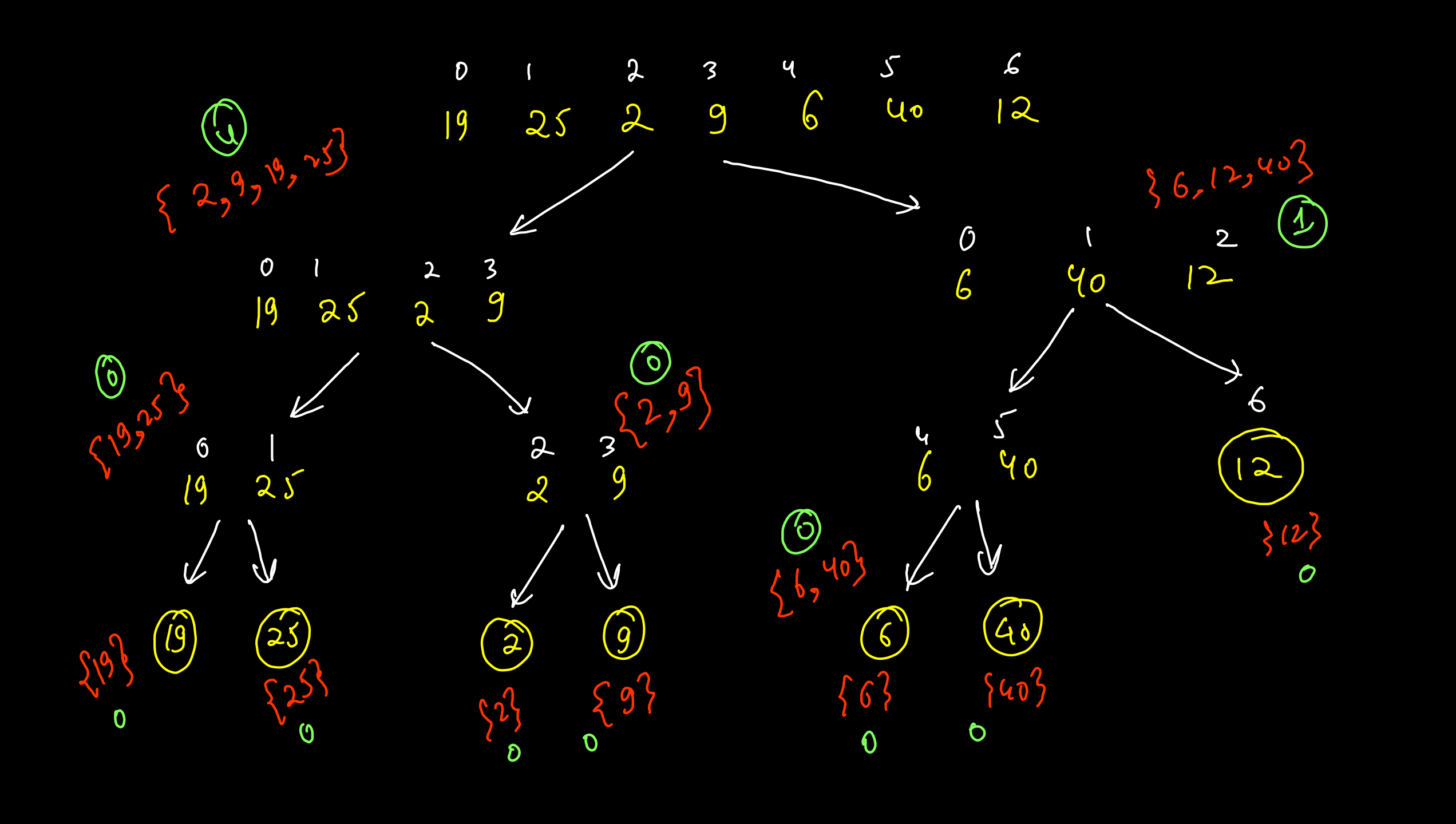
$arr[i] > 2 * arr[j]$

(19, 2*2) (25, 2*2)

(19, 2*9) (25, 2*9)

(19, 2*6) (25, 2*6)

(40, 2*12) (25, 2*12)


0 1 2 3 4 5 6
19 25 2 9 6 40 12
1
{2, 9, 19, 25}
{6, 12, 40}
1
0 1 2 3
19 25 2 9
0 1 2
6 40 12
0
{19, 25}
0
{2, 9}
0 1 2 3
19 25 2 9
4 5 6
6 40 12
0
{6, 40}
{12}
0
{19}
19
25
{25}
0
2
{2}
0
9
{9}
0
6
{6}
0
40
{40}
0

0 1

19, 25

↑

2 3

2, 9

~~↑~~ ~~↑~~ ↑

```
if (arr[p1] > 2*arr[p2])
    count += (mid-idx+1)
    res.add(arr[p2]);
    p2++
else
    res.add(arr[p1])
    p1++;
```

res: (2) (9) (19) (25)

count: 0+2+2

(19, 2*2) (19, 2*9)

(25, 2*2) (25, 2*9)

6 40

p1 p1

12

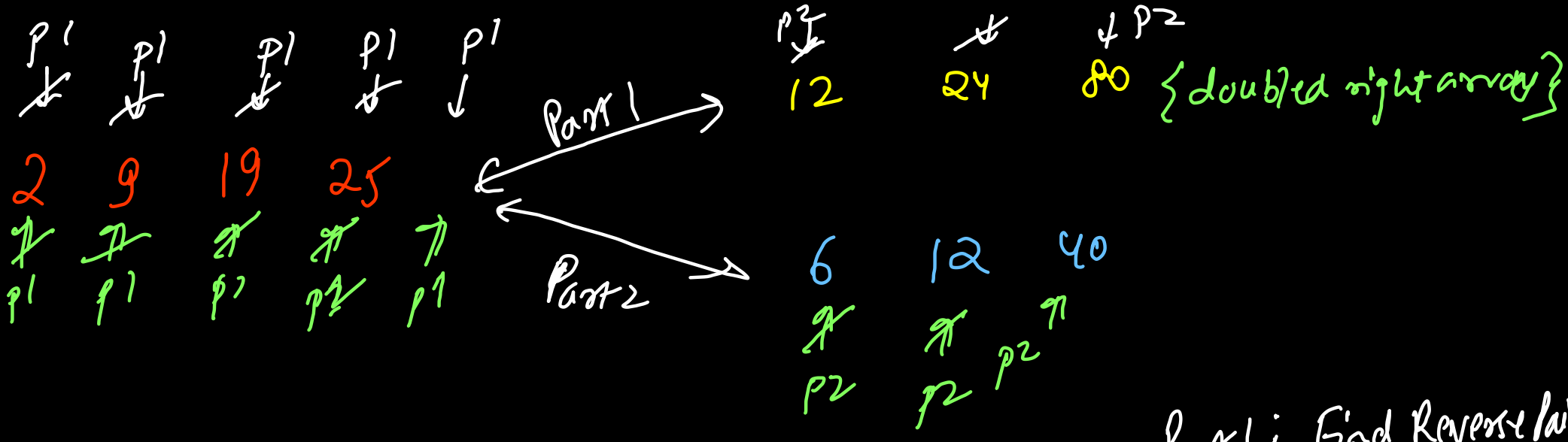
p2 p2

res: 6 12 40

Count = 0+1

(40,12)

```
if (arr[p1] > 2*arr[p2])
    count += (mid-idx+1)
    res.add(arr[p2]);
    p2++
else
    res.add(arr[p1])
    p1++;
```




count = 0+2+1 = (3)

(19, 2*6)

(25, 2*6)

(25, 2*12)

ru: 2 6 9 12 19 25 40

merge

Part 1: Find Reverse Pairs

```
if (arr[p1] > 2*arr[p2])
    count += (mid-idx+1)
    p2++
else
    p1++;
```

Part 2: Merge 2 Sorted Arrays

```java
int calcInv(int[] arr, int left, int mid, int right){
    int invCount = 0;
    int idx1 = left, idx2 = mid + 1, idx3 = 0;
    while(idx1 <= mid && idx2 <= right){
        if(arr[idx1] <= 2l * arr[idx2]){
            idx1++; idx3++;
        } else {
            invCount += (mid - idx1 + 1);
            idx2++; idx3++;
        }
    }
    return invCount;
}
```

twice the second array

```java
int mergeSort(int nums[], int l, int r)
{
    if(l == r) return 0;
    int mid = (l + r) / 2;
    int left = mergeSort(nums, l, mid);
    int right = mergeSort(nums, mid + 1, r);
    int curr = merge(nums, l, mid, r);
    return left + curr + right;
}

public int reversePairs(int[] nums) {
    return mergeSort(nums, 0, nums.length - 1);
}
```

```java
int merge(int[] arr, int left, int mid, int right){
    int invCount = calcInv(arr, left, mid, right);

    int[] res = new int[right - left + 1];
    int idx1 = left, idx2 = mid + 1, idx3 = 0;
    while(idx1 <= mid && idx2 <= right){
        if(arr[idx1] <= arr[idx2]){
            res[idx3] = arr[idx1];
            idx1++; idx3++;
        } else {
            res[idx3] = arr[idx2];
            idx2++; idx3++;
        }
    }

    while(idx1 <= mid){
        res[idx3] = arr[idx1];
        idx1++; idx3++;
    }
    while(idx2 <= right){
        res[idx3] = arr[idx2];
        idx2++; idx3++;
    }
    for(int idx = left; idx <= right; idx++){
        arr[idx] = res[idx - left];
    }

    return invCount;
}
```

Normal merge a Sorted Array

Time ⇒ O(nlogn)
Space ⇒ O(n)

Coder → 20-25 k monthly

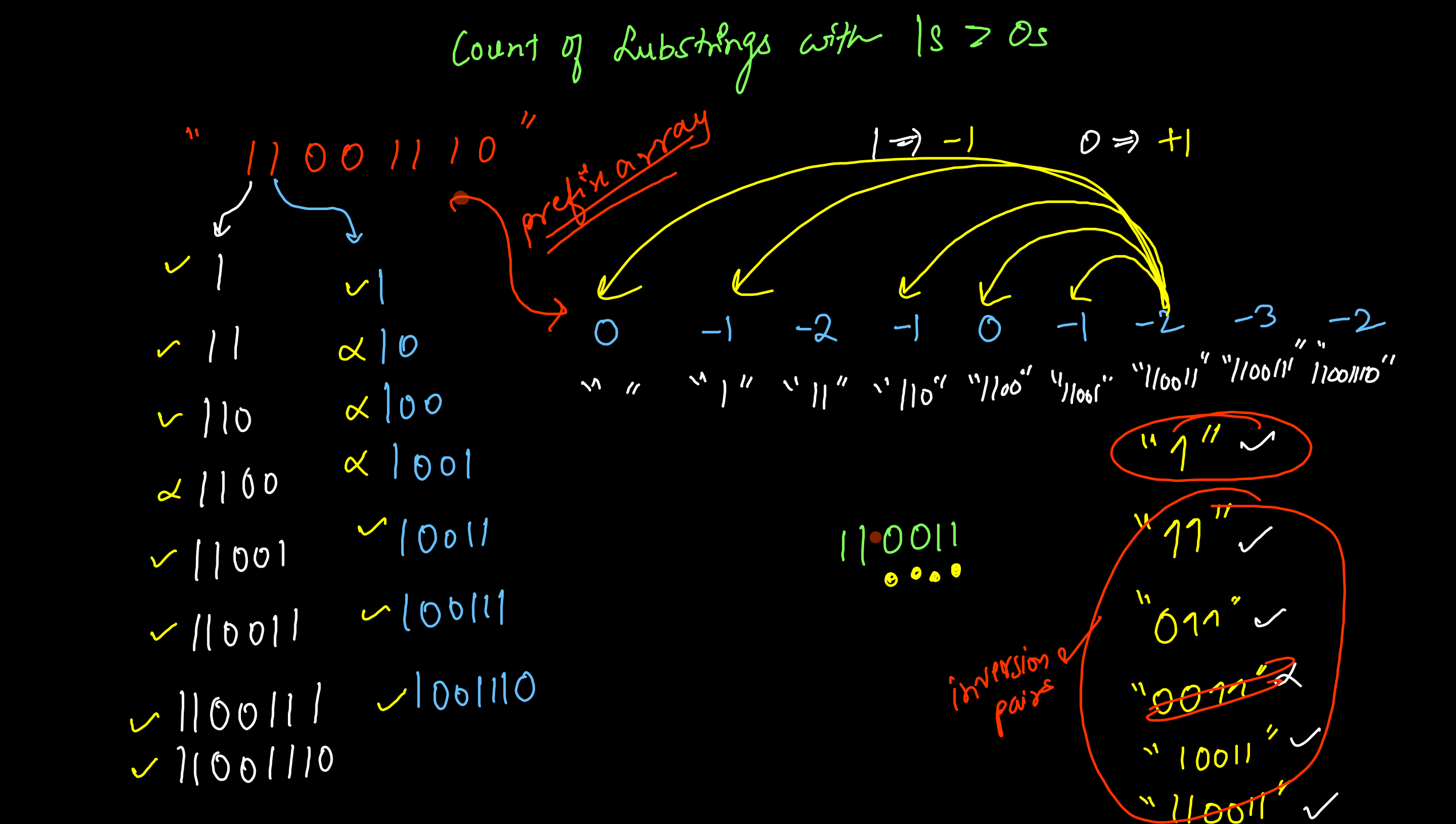
↳ code
  ↳ debugger
  ↳ tester
  ↳ support
  ↳ data analyst

So LIA ∈ programming
  ↳ architecture (HLD + LLD)
  ↳ Coding
  ↳ problem solving {DSA}
    ↳ dry run
    ↳ pen & paper
    ↳ insta

# procastination
  ↳ distractions ↓
  ↳ pair programming ↑
    ↳ competition
  ↳ achievement ↑ celebration

Software developer
  ↳ devops
  ↳ webapps | mobileapps | desktop apps
  ↳

# Count of Substrings with 1s ≥ 0s



Total answer = Total Nos of 1s

+

Inversion pairs in Prefix Array

(0 ⇒ +1, 1 ⇒ -1)

# Partitioning Algorithms

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| arr :- | 40 | 90 → 20 | 70 → 30 | 80 → 10 | 20 → 50 | 30 → 70 | 60 | 10 → 80 | 50 → 90 |

pivot :- 50

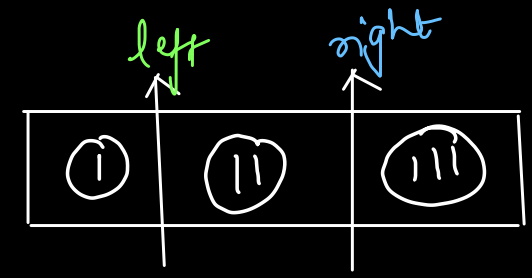
left

right

40 20 30 10 | 50 | 70 60 80 90

< 50 | pivot | > 50

```
while( right < arr.length){
if(arr[right] ≤ pivot) { swap(arr,l,r); l++; r++; }
else { right++; }}
```




[0, left) ⇒ smaller region (≤ pivot)

[left, right) ⇒ greater region (> pivot)

[right, n) ⇒ unexplored region

```java
public static void partition(int[] arr, int pivot){
  int left = 0, right = 0;
  while(right < arr.length){
      if(arr[right] <= pivot){
          swap(arr, right, left);
          left++; right++;
      } else {
          right++;
      }
  }
}

// used for swapping ith and jth elements of array
public static void swap(int[] arr, int i, int j) {
  System.out.println("Swapping " + arr[i] + " and " + arr[j]);
  int temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}
```

Note { not stable }

① left → relative order maintained ✓

② right → relative order maintained ✗

Time

↳ O(n)

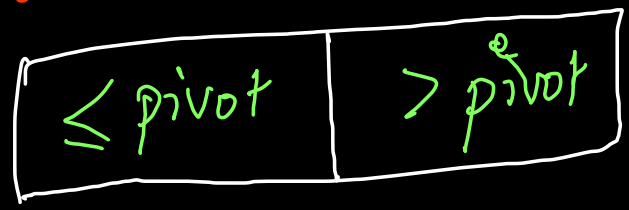
Space

↳ O(1)

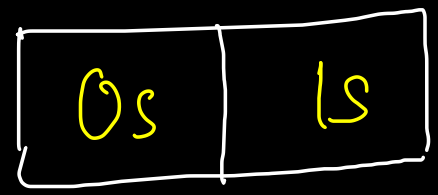
(constant (inplace)

Variations

(1)




(2) Binary Array / String :→ (Sort 0 1)




eg 0 1 1 0 0 1 1 0

↓

0 0 0 0 1 1 1 1 1

(1) Brute force → Sorting → $O(n \log n)$

(2) Linear Two Traversal

→ (2.1) Count zeros, ones

→ (2.2) Place all zeros in left, place all one's in right

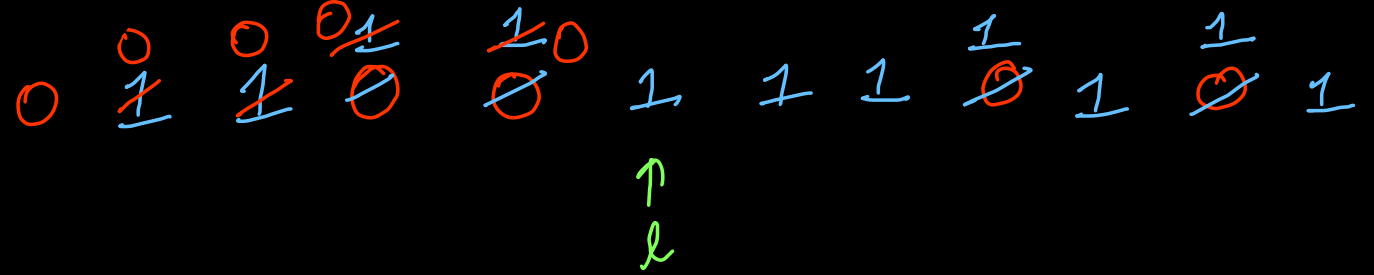
Time ↳ $O(n+n) = O(n)$

Space ↳ $O(1)$

(3) Linear Single Traversal → Two Pointer Technique

Time $= O(n)$

Space $= O(1)$




```
static void swap(int[] arr, int left, int right){
    int temp = arr[left];
    arr[left] = arr[right];
    arr[right] = temp;
}
static void binSort(int arr[], int N)
{
    int left = 0, right = 0;
    while(right < N){
        if(arr[right] == 0){
            swap(arr, left, right);
            left++; right++;
        } else {
            right++;
        }
    }
}
```

Sort Binary Array

Linear Single Traversal

Time ⇒ O(n)

Space ⇒ O(1)

≤ pivot ⇒ 0's

> pivot ⇒ 1's

③ Segregate odd & even {leetcode 905}
↳ Sort Array by Parity

odd → <pivot
even → >pivot

④ Move zeros to End (leetcode 283)

→ nonzero left (maintained)
→ zeros right

20 −30 0 0 40 10 0 60 0

↓

20 −30 40 10 60 0 0 0 0

if (arr[right] != 0) (swap l++ r++) else r++

```java
class Solution {
    public void swap(int[] nums, int left, int right){
        int temp = nums[left];
        nums[left] = nums[right];
        nums[right] = temp;
    }

    public int[] sortArrayByParity(int[] nums) {
        int left = 0, right = 0;

        while(right < nums.length){
            if(nums[right] % 2 == 0){
                swap(nums, left, right);
                left++; right++;
            } else {
                right++;
            }
        }

        return nums;
    }
}
```

Leetcode 905

Segregate Odd & Even

Sort Array by Parity

```java
public void swap(int[] nums, int left, int right){
    int temp = nums[left];
    nums[left] = nums[right];
    nums[right] = temp;
}

public void moveZeroes(int[] nums) {
    int left = 0, right = 0;

    while(right < nums.length){
        if(nums[right] != 0){
            swap(nums, left, right);
            left++; right++;
        } else {
            right++;
        }
    }
}
```

Leetcode 283

Time => O(n)

Space => O(1)

Relative order maintained!

⑤ Segregate +ves -ves

⑥ Segregate primeno non prime

⑦ String → Segregate 'x' & 'y'

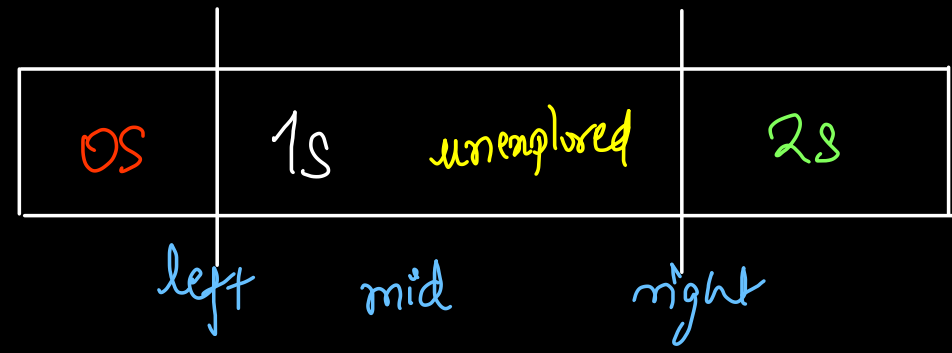
# Dutch National Flag Algorithm

DNF Sort

Sort 0s 1s 2s

Sort Colors (R/G/B)

Leetcode 75




0 1 1 2 0 2 1 2 0 1 1 2 0

⇓

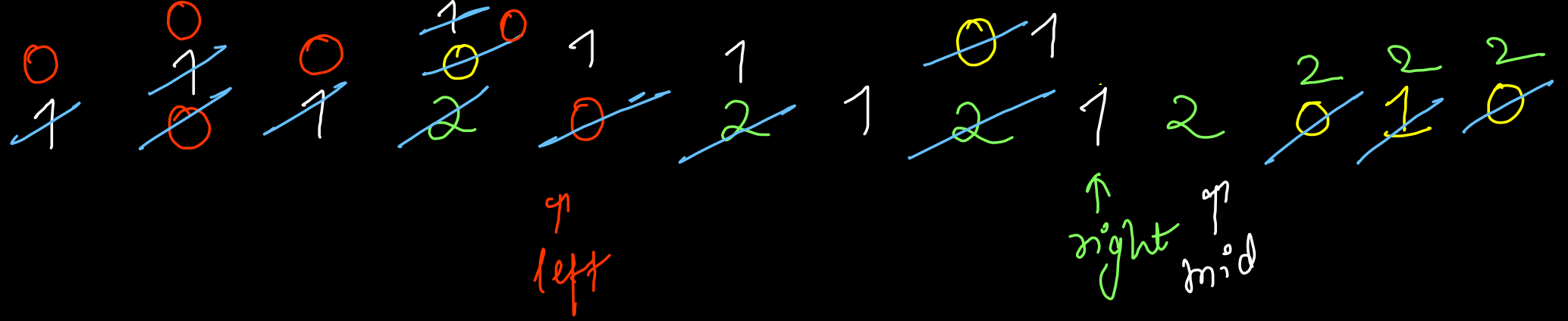
0 0 0 0 1 1 1 1 1 2 2 2 2

[0, left) ⇒ 0s

[left, mid) ⇒ 1s

[mid, right] ⇒ unexplored

(right, n) ⇒ 2s




```
while ( mid <= right) {
  if (arr[mid]==0) { swap(l,m); l++; m++; }
  else if (arr[mid]==1)  { mid++; }
  else  { swap(m,r); right--; }
}
```

```java
public void swap(int[] nums, int p1, int p2){
    int temp = nums[p1];
    nums[p1] = nums[p2];
    nums[p2] = temp;
}

public void sortColors(int[] nums) {
    int left = 0, mid = 0, right = nums.length - 1;

    while(mid <= right){
        if(nums[mid] == 0){
            swap(nums, left, mid);
            left++; mid++;
        } else if(nums[mid] == 1){
            mid++;
        } else {
            swap(nums, right, mid);
            right--;
        }
    }
}
```

Time Complexity
↳ O(n)

Space Complexity
↳ O(1) (constant)
inplace

# Variations of Sort 012

## ① Three Way partitioning

< pivot  = pivot  > pivot

eg  50 30 40 20 40 60 40 50 20 70 10

pivot = 40

10 20 30 20  40 40 40  50 70 60

< 40  = 40  > 40

## ② Dual pivot partitioning

pivot = a, b  $a < b$

| $< a$ | $[a, b]$ | $> b$ |
|---|---|---|

eg 40 10 30 50 70 80 20 30 50 60

pivot → a = 30, b = 60

⇓

10 20 | 40 30 50 30 50 60 | 70 80

$< 30$ | $[30, 60)$ | $\geq 60$

```java
public void swap(int[] nums, int p1, int p2){
    int temp = nums[p1];
    nums[p1] = nums[p2];
    nums[p2] = temp;
}

public void threeWayPartition(int arr[], int a, int b)
{
    int left = 0, mid = 0, right = arr.length - 1;

    while(mid <= right){
        if(arr[mid] < a){
            swap(arr, left, mid);
            left++; mid++;
        } else if(arr[mid] >= a && arr[mid] <= b){
            mid++;
        } else {
            swap(arr, right, mid);
            right--;
        }
    }
}
```

dualpivot

Dual Pivot Partitioning

<a [a,b] >b

```java
public void swap(int[] nums, int p1, int p2){
    int temp = nums[p1];
    nums[p1] = nums[p2];
    nums[p2] = temp;
}

public void threeWayPartition(int arr[], int a, int b)
{
    int left = 0, mid = 0, right = arr.length - 1;

    while(mid <= right){
        if(arr[mid] < a){
            swap(arr, left, mid);
            left++; mid++;
        } else if(arr[mid] == a){
            mid++;
        } else {
            swap(arr, right, mid);
            right--;
        }
    }
}
```

Three way Partitioning

<a =a >a

# Other Variations of Sort 0-1-2

③ Segregate -ves, 0s, +ves

-ves → 0s
0s → 1s
+ves → 2s

④ String → Segregate

| Red | Green | Blue |
|---|---|---|
| R | G | B |
| P | Q | R |
| X | Y | Z |

⑤ Segregate nos

| %3=0 | %3=1 | %3=2 |
|---|---|---|
| 3n | 3n+1 | 3n+2 |

# Three way Partitioning (with relative order)

Leetcode 2161

Input:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 60 | 80 | 30 | 10 | 50 | 90 | 70 | 20 | 60 | 50 |

pivot = 50

< pivot = pivot > pivot

| 40 | 30 | 10 | 20 | 50 | 50 | 50 | 80 | 90 | 70 | 60 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

extra space

| 40 | 30 | 10 | 20 | 50 | 50 | 50 | 80 | 90 | 70 | 60 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

```java
public int[] pivotArray(int[] nums, int pivot) {
    int[] res = new int[nums.length];
    int left = 0, right = nums.length - 1;

    // < pivot      left to right
    for(int idx = 0; idx < nums.length; idx++){
        if(nums[idx] < pivot) {
            res[left] = nums[idx];
            left++;
        }
    }

    // > pivot      right to left
    for(int idx = nums.length - 1; idx >= 0; idx--){
        if(nums[idx] > pivot){
            res[right] = nums[idx];
            right--;
        }
    }

    // = pivot      in the left & right
    for(int idx = left; idx <= right; idx++){
        res[idx] = pivot;
    }
    return res;
}
```

Time ⇒ O(n) → Two Traversal Linear

Space ⇒ O(n) extra space

leetcode 2161

# Segregate +ves & -ves

relative order maintained

| 30 | -20 | 40 | 10 | -50 | -10 | 20 | 50 | -30 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| 30 | 40 | 10 | 20 | 50 | -20 | -50 | -10 | -30 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

+ves

-ves

```java
public void approach1(int[] arr){
    int[] res = new int[arr.length];

    int left = 0;
    for(int idx = 0; idx < arr.length; idx++){
        if(arr[idx] >= 0){
            res[left] = arr[idx];
            left++;
        }
    }

    int right = arr.length - 1;
    for(int idx = arr.length - 1; idx >= 0; idx--){
        if(arr[idx] < 0){
            res[right] = arr[idx];
            right--;
        }
    }

    for(int idx = 0; idx < arr.length; idx++){
        arr[idx] = res[idx];
    }
}
```

Time → O(n) → Linear → Three Traversal → Most optimized

Space → O(n) → extra space → Inplace ×

# Segregate +ves & -ves

relative order maintained

without extra space

| 30 | 40 | 10 | 20 | 50 | -20 | -50 | -10 | -30 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

↑ idx

```java
public void swap(int[] arr, int p1, int p2){
    int temp = arr[p1];
    arr[p1] = arr[p2];
    arr[p2] = temp;
}

public void approach2(int[] arr){
    for(int i = 1; i < arr.length; i++){
        if(arr[i] < 0) continue;

        for(int j = i - 1; j >= 0; j--){
            if(arr[j] < 0){
                swap(arr, j, j + 1);
            } else break;
        }
    }
}
```

Insertion Sort

Time ⇒ $O(n^2)$ quadratic (TLE)

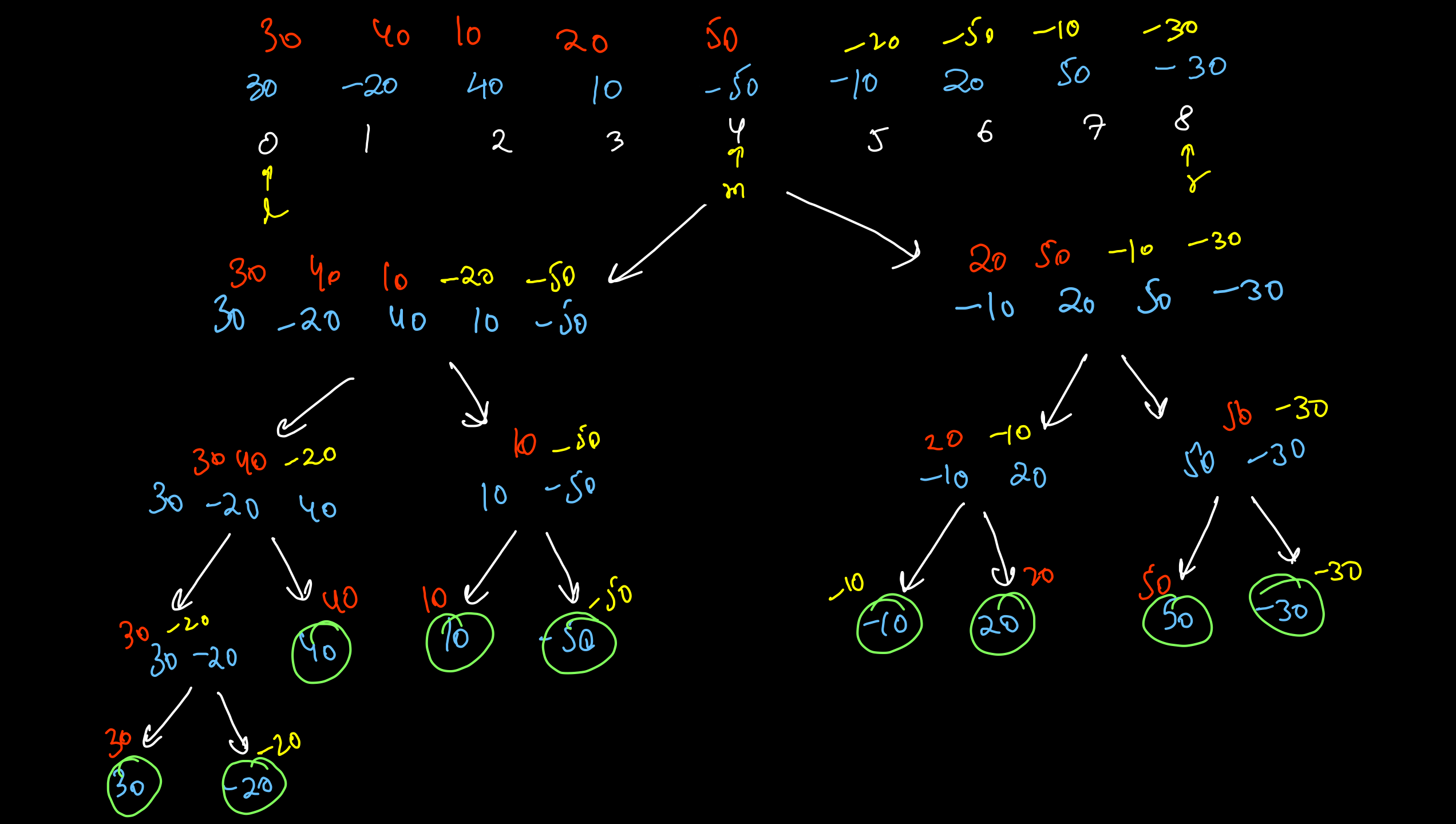
Space ⇒ $O(1)$ inplace

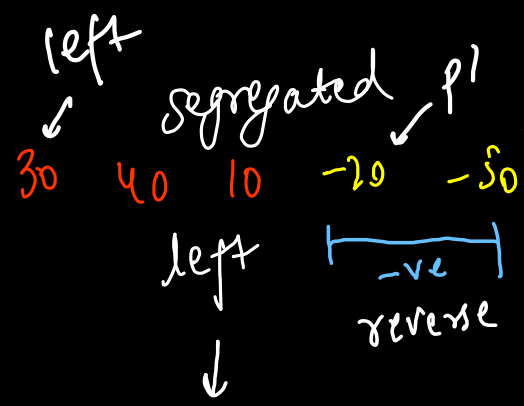
# Segregate +ves & -ves
relative order maintained
without extra space

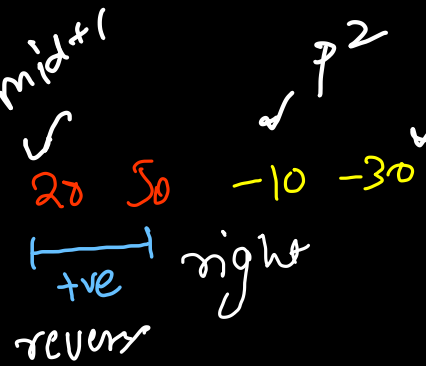
Approach 3 ; Time ⇒ O(nlogn) using merge sort (Divide & Conquer)

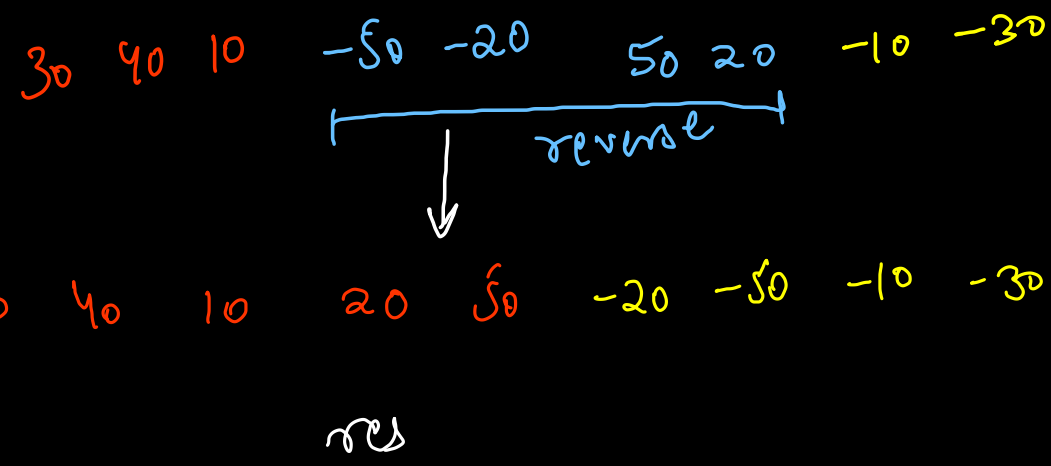
Space ⇒ O(1) inplace (extra)
⇒ O(logn) recursive call stack space

| 30 | -20 | 40 | 10 | -50 | -10 | 20 | 50 | -30 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |


30 40 10 20 50 -20 -50 -10 -30
30 -20 40 10 -50 -10 20 50 -30
0 1 2 3 4 5 6 7 8
l m r
30 40 10 -20 -50
30 -20 40 10 -50
20 50 -10 -30
-10 20 50 -30
30 40 -20
30 -20 40
10 -50
10 -50
20 -10
-10 20
50 -30
50 -30
30 -20
30 -20
30
-20
-10
20
50
-30
30
-20



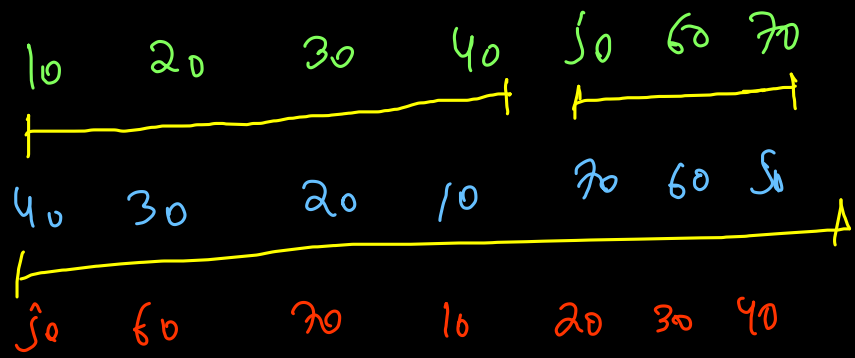

① reverse (p1, mid)

② reverse (mid+1, p2-1)

③ reverse (p1, p2-1)

## Rotate Array (Reversal Algo)

k=3




reverse(left)

reverse(right)

reverse(arr).

```java
public void reverse(int[] arr, int left, int right){
    while(left < right){
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
        left++; right--;
    }
}

public void merge(int[] arr, int left, int mid, int right){
    int p1 = left; // p1 is first negative element in left array
    while(p1 <= mid && arr[p1] >= 0) p1++;

    int p2 = mid + 1; // p2 is first negative element in right array
    while(p2 <= right && arr[p2] >= 0) p2++;

    reverse(arr, p1, mid);
    reverse(arr, mid + 1, p2 - 1);
    reverse(arr, p1, p2 - 1);
}

public void mergeSort(int[] arr, int left, int right){
    if(left == right) return;

    int mid = (left + right) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```

Time ⇒ O(n log n)

Space ⇒ O(1) extra space auxiliary

O(log n) recursion extra space

Youth: Segregate +ves & -ves with relative order maintained w/o extra space

mergeSort(arr, 0, n-1);

Continue at 12:00PM

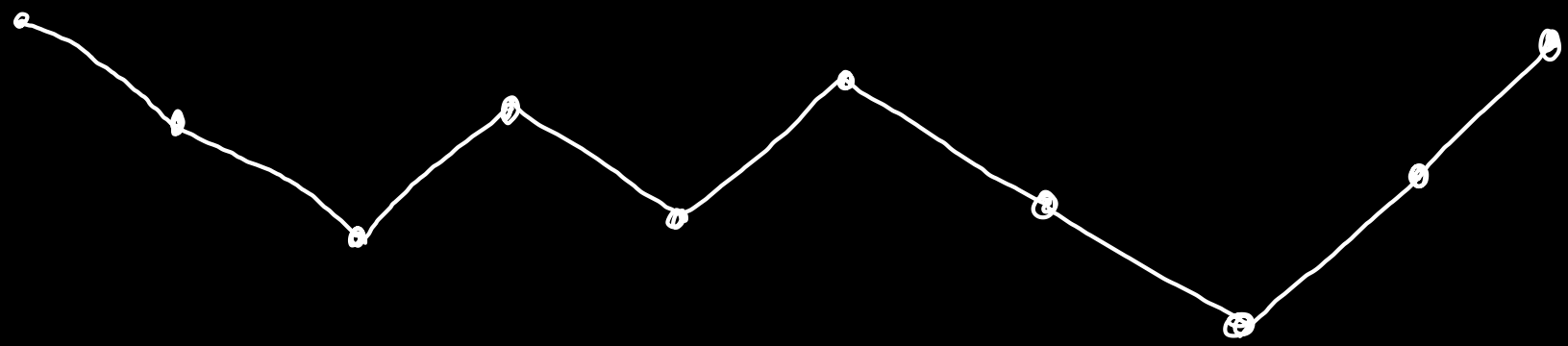
# Wave Sort / Wiggle Sort

1. Without duplicates → any order (convert array into zigzag GFG)
2. With duplicates (with equal adjacent) → any order (wiggle sort codestudio)
3. With duplicates (with unequal adjacent) → any order (wiggle sort II → LC324) (Hard / Mathematics)
4. Lexicographically minimum order (wave array → GFG)
5. Minimum-Maximum form or Peak-valley form (Rearrange Array Alternately - GFG)
6. Inverse Permutation (Rearrange Array - GFG)

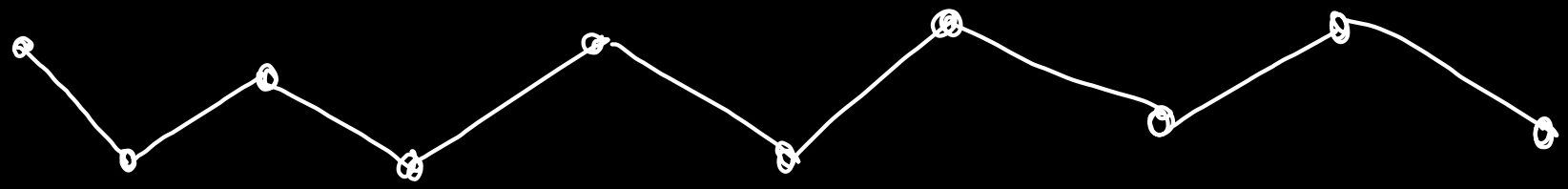
① without duplicates → any order

Zig Zag order

40 20 10 60 50 80 70 30 90 100



40 20 60 10 80 70 90 50 100 30



## (5) Minimum-Maximum form or Peak-valley form (Rearrange Array Alternately -GFG)

40 20 10 60 50 80 70 30 90 100

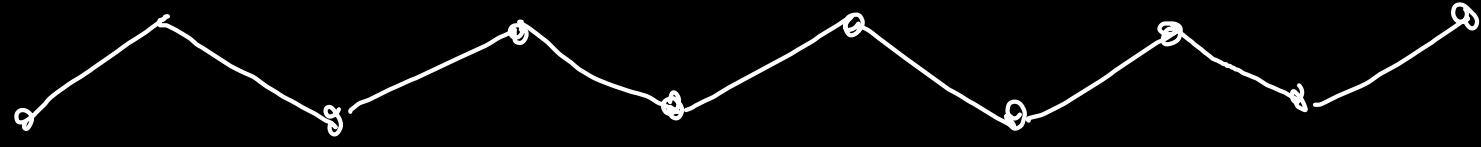
↓ sort :→ Time ⇒ O(nlogn)

10 20 30 40 50 60 70 80 90 100

zigzag

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| extra array | 10 | 100 | 20 | 90 | 30 | 80 | 40 | 70 | 50 | 60 |

O(n)



# Approach (1) with extra space

```java
public static void minMaxForm(long arr[], int n){
    long[] sorted = new long[n];
    for(int idx = 0; idx < n; idx++) sorted[idx] = arr[idx];

    int left = 0, right = arr.length - 1;

    for(int idx = 0; idx < n; idx++){
        if(idx % 2 == 1){
            arr[idx] = sorted[left];
            left++;
        } else {
            arr[idx] = sorted[right];
            right--;
        }
    }
}
```

## Time

# If already sorted
↳ O(n)

# If not already sorted
↳ O(nlogn)

## Space

O(n) extra space
inplace ✗

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| arr: | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| res: | 100 | 10 | 90 | 20 | 80 | 30 | 70 | 40 | 60 | 50 |

max = 1000

Quotient - Remainder Method

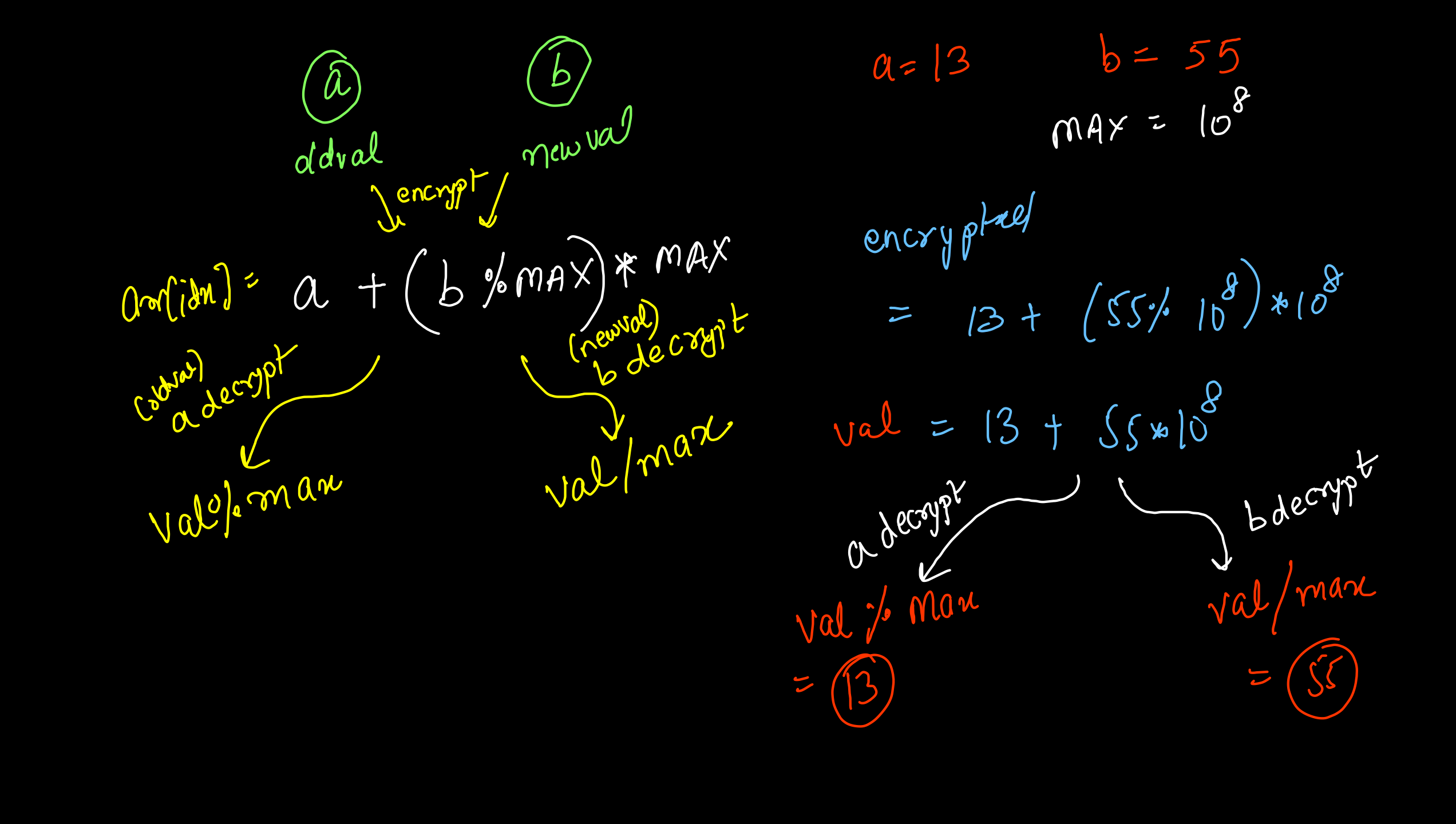
Constraint

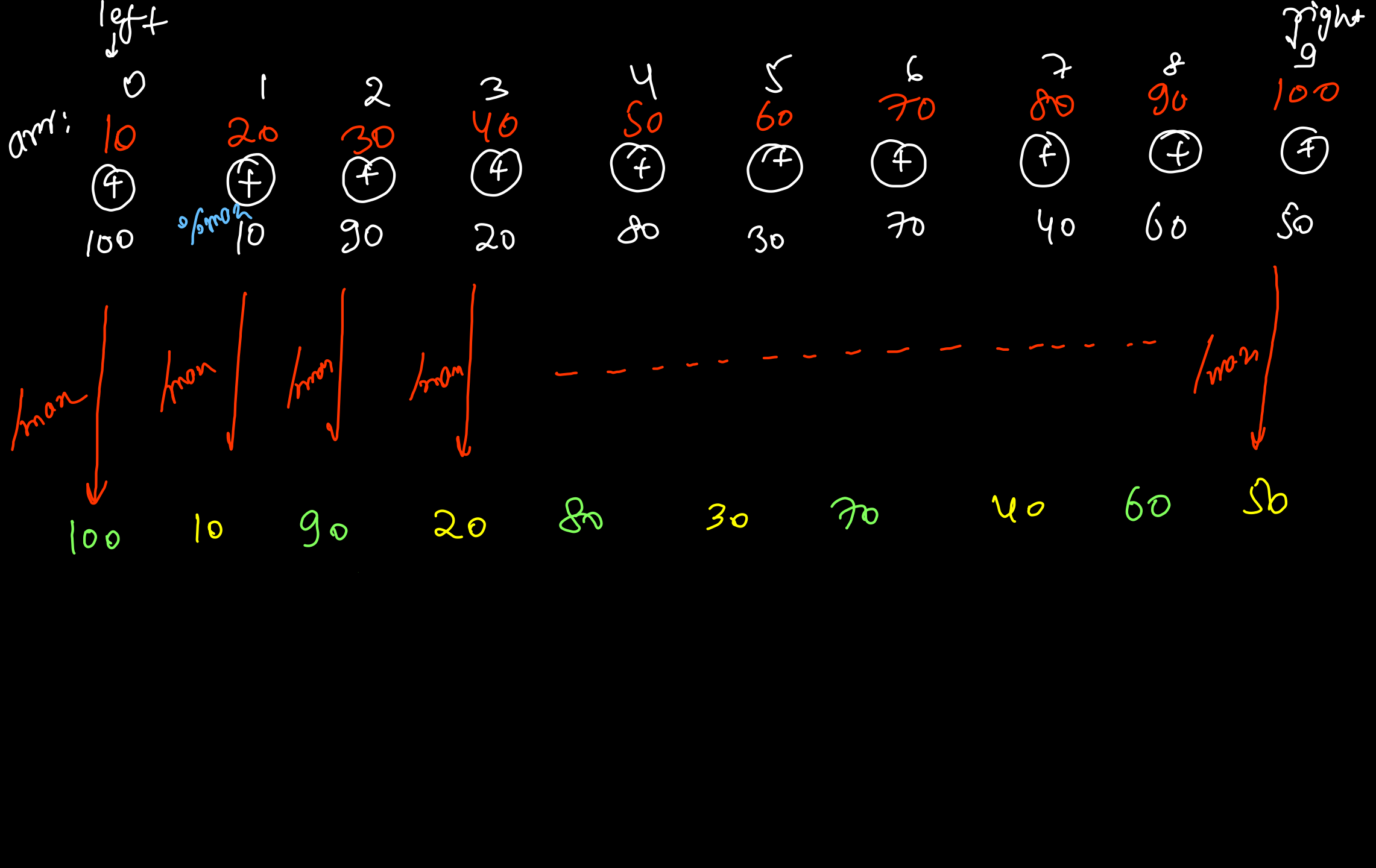
$arr[i] \leq 10^7$ integer

long datatype

we can store long

$10^{18}$


a
b
oldval
newval
encrypt
arr[idx] = a + (b % MAX) * MAX
(oldval) a decrypt
(newval) b decrypt
val % max
val / max
a = 13
b = 55
MAX = 10^8
encrypted
= 13 + (55 % 10^8) * 10^8
val = 13 + 55 * 10^8
a decrypt
b decrypt
val % max
= 13
val / max
= 55


left
right
0 1 2 3 4 5 6 7 8 9
arr:
10 20 30 40 50 60 70 80 90 100
100 10 90 20 80 30 70 40 60 50
max
100 10 90 20 80 30 70 40 60 50

```java
public static void quotientRemainder(long[] arr, int n){
    long max = 10000001;
    int left = 0, right = n - 1;

    // Encryption
    for(int idx = 0; idx < n; idx++){
        long oldVal = arr[idx];
        long newVal = (idx % 2 == 0)
                    ? arr[right--] % max
                    : arr[left++] % max;

        arr[idx] = oldVal + (newVal % max) * max;
    }

    // Decryption
    for(int idx = 0; idx < n; idx++){
        arr[idx] = arr[idx] / max;
    }
}
```

remainder

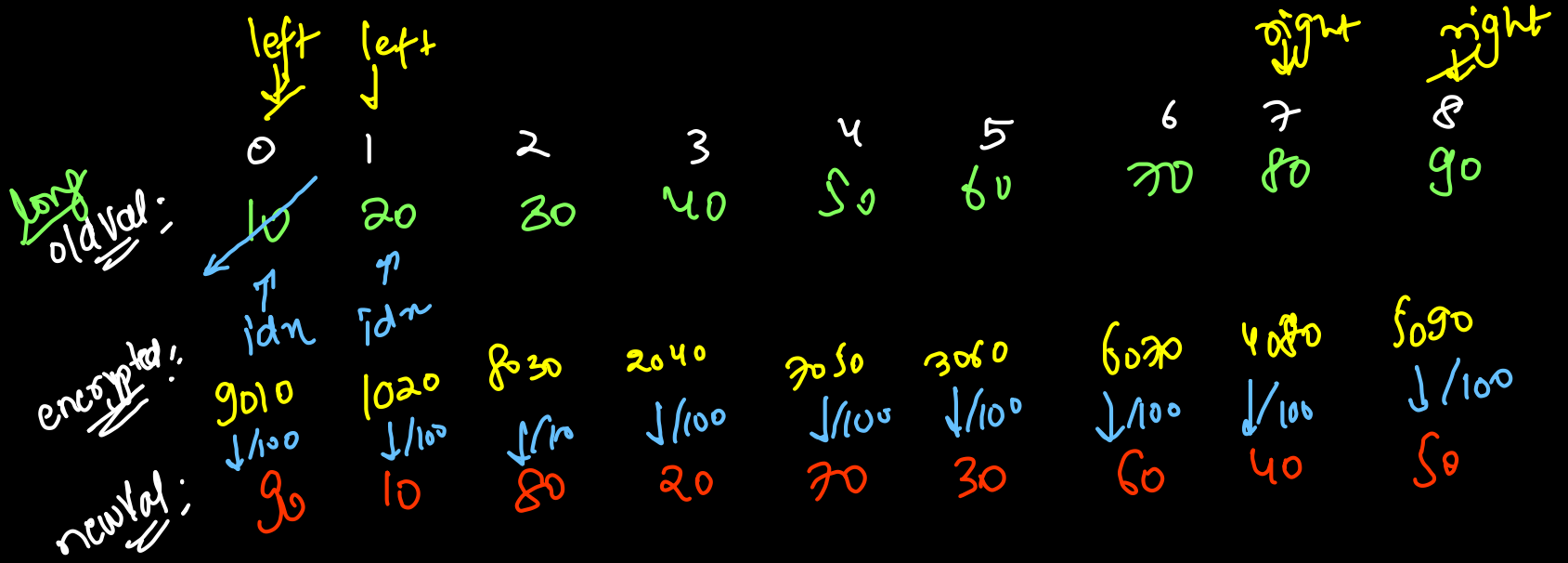
quotient

Time ⇒ O(n) (Already sorted)

Space ⇒ O(1)

Wiggle Sort-
min-max form

quotient-remainder method




```java
public static void quotientRemainder(long[] arr, int n){
    long max = 10000001;
    int left = 0, right = n - 1;

    // Encryption
    for(int idx = 0; idx < n; idx++){
        long oldVal = arr[idx];
        long newVal = (idx % 2 == 0)
                    ? arr[right--] % max
                    : arr[left++] % max;

        arr[idx] = oldVal + (newVal % max) * max;
    }

    // Decryption
    for(int idx = 0; idx < n; idx++){
        arr[idx] = arr[idx] / max;
    }
}
```

Zig-Zag: min-max Form

for idx=0

a = 10

b = 90 % 100 = 90

10 + 90 * 100 = 9010

for idx=1

a = 20

b = 9010 % 100 = 10

20 + 10 * 100 = 1020

max = 100

↑ greater than largest value in array

a + (b % max) * max

Inverse permutation

n = 10

Arr:

| input: | 3 | 7 | 9 | 2 | 5 | 0 | 1 | 4 | 6 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

| res: | 2 | 4 | 8 | 9 | 0 | 3 | 7 | 5 | 1 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|

arr[arr[i]]

## Rearrange an array with O(1) extra space

Medium Accuracy: **56.34%** Submissions: **72078** Points: **4**

Given an array **arr[]** of size **N** where every element is in the range from **0 to n-1**. Rearrange the given array so that **arr[i]** becomes **arr[arr[i]]**.

```java
static void approach1(long arr[], int n){
    long[] inv = new long[n];

    for(int idx = 0; idx < n; idx++){
        inv[idx] = arr[(int)arr[idx]];
    }

    for(int idx = 0; idx < n; idx++){
        arr[idx] = inv[idx];
    }
}
```

Using extra space

{ constructive algo }

Time ⇒ O(n)

Space ⇒ O(n) extra space

# Wave Array

(lexicographically minimum)

↳ dictionary order

"archit" < "arpit"

first uncommon character

"abcdzzzzz" < "abce"

d < e

I < II

199 < 200

integer

(numerically)

"199" < "200"

'1' < '2'

I < II

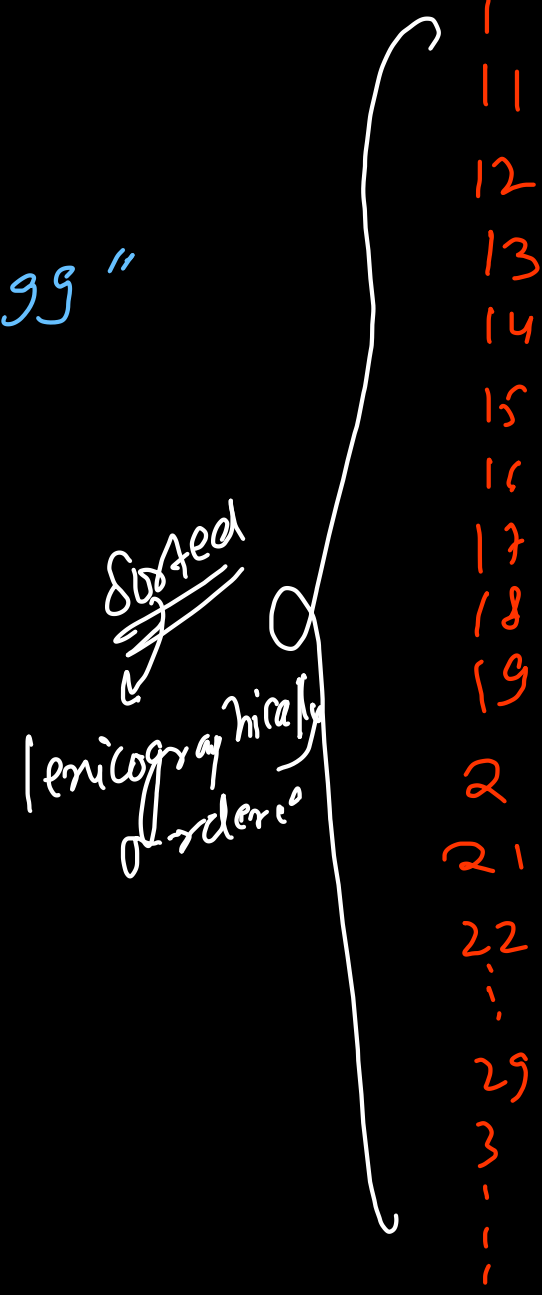
"8534" "299999999"

As integers 8534 < 299999999

As string (Lexicographical) "8534" > "29999999"

"archit" < "architaggarwal"

prefix will be smaller



(sorted) already lexicographical minimum

① Wave Array

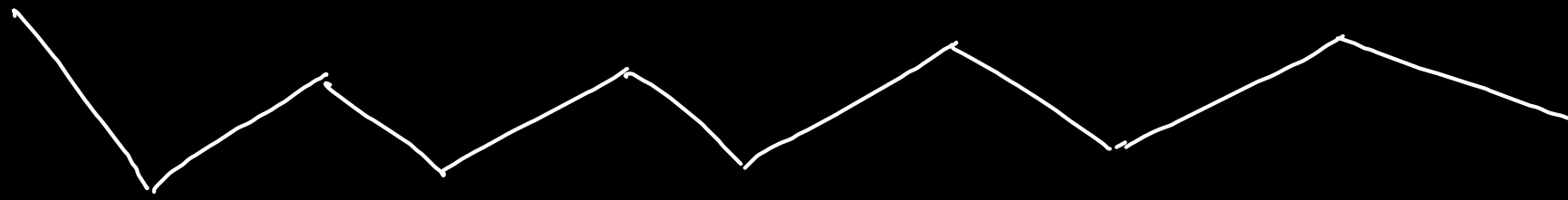
② (lexicographically minimum)

② min changes

10 20 30 40 50 60 70 80 90 100

20 10 40 30 60 50 80 70 100 90

① wave array



```java
public static void convertToWave(int n, int[] arr) {
    for(int i = 0; i < n - 1; i = i + 2){
        int temp = arr[i];
        arr[i] = arr[i + 1];
        arr[i + 1] = temp;
    }
}
```

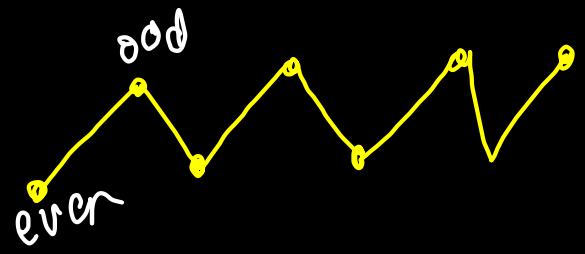
# Wiggle Sort - I

GFG

arr : distinct/unique

wave array

**arr[0] < arr[1] > arr[2] < arr[3] > arr[4] < . . . .**
**arr[n-2] < arr[n-1] > arr[n].**




Expected

Time → O(n)

Space → O(1)

inplace

Arr[] = {4, 3, 7, 8, 6, 2, 1}

**Output:** 3 7 4 8 2 6 1

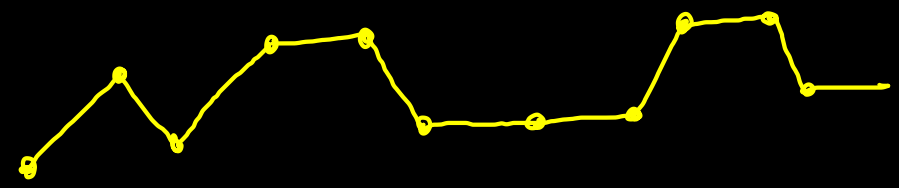
**Explanation:** 3 < 7 > 4 < 8 > 2 < 6 > 1

hC Locked (Codingninjas)

arr : duplicates allowed

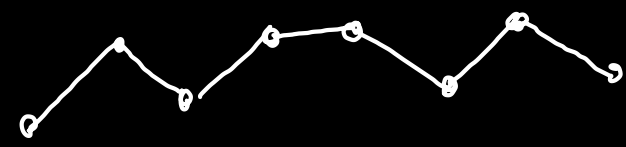
arr[0] ≤ arr[1] ≥ arr[2]

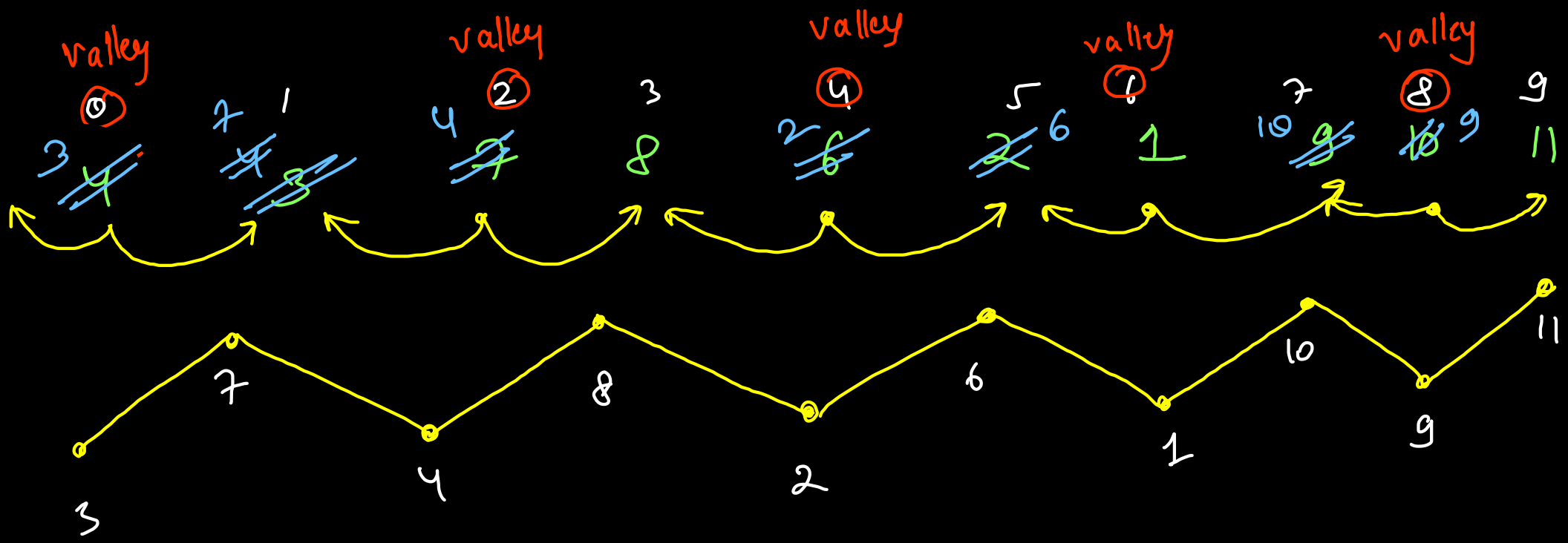
≤ arr[3] ≥ arr[4] ≤ arr[5]..



Arr : { 4, 3, 3, 8, 8, 6, 2, 2 }

3 4 3 8 8 2 6 2






Intuition

→ Make all even indices as valley

→ make all odd indices as peak

```java
public static void swap(int[] arr, int p1, int p2){
    int temp = arr[p1];
    arr[p1] = arr[p2];
    arr[p2] = temp;
}

public static int[] wiggleSort(int n, int[] arr) {
    for(int idx = 0; idx < n; idx = idx + 2){ // valley indices
        if(idx - 1 >= 0 && arr[idx - 1] < arr[idx]){
            swap(arr, idx - 1, idx);
        }

        if(idx + 1 < n && arr[idx + 1] < arr[idx]){
            swap(arr, idx + 1, idx);
        }
    }

    return arr;
}
```

Time ⇒ O(n)

Space ⇒ O(1)
inplace

# Quick Sort

→ Normal quick sort

Optimizations :→ ① Randomized quick sort

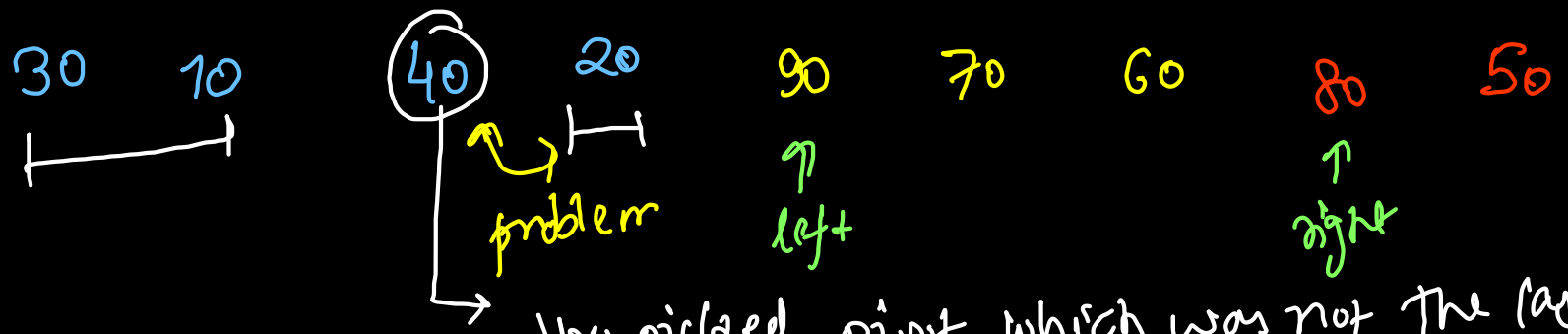
② Three way quicksort

③ Dual pivot quick sort

```java
public static void partition(int[] arr, int pivot) {
    int left = 0, right = 0;
    while (right < arr.length) {
        if (arr[right] <= pivot) {
            swap(arr, right, left);
            left++;
            right++;
        } else {
            right++;
        }
    }
}      You, 4 days ago • Added Partitioning
```

# Quick Sort {divide & conquer}

(Partitioning)




You picked pivot which was not the last index of array, hence partitioning logic is wrong

≤ pivot → left

> pivot → right

30 10 20 (40) 70 60 90 80 50

30 10 20 < 40

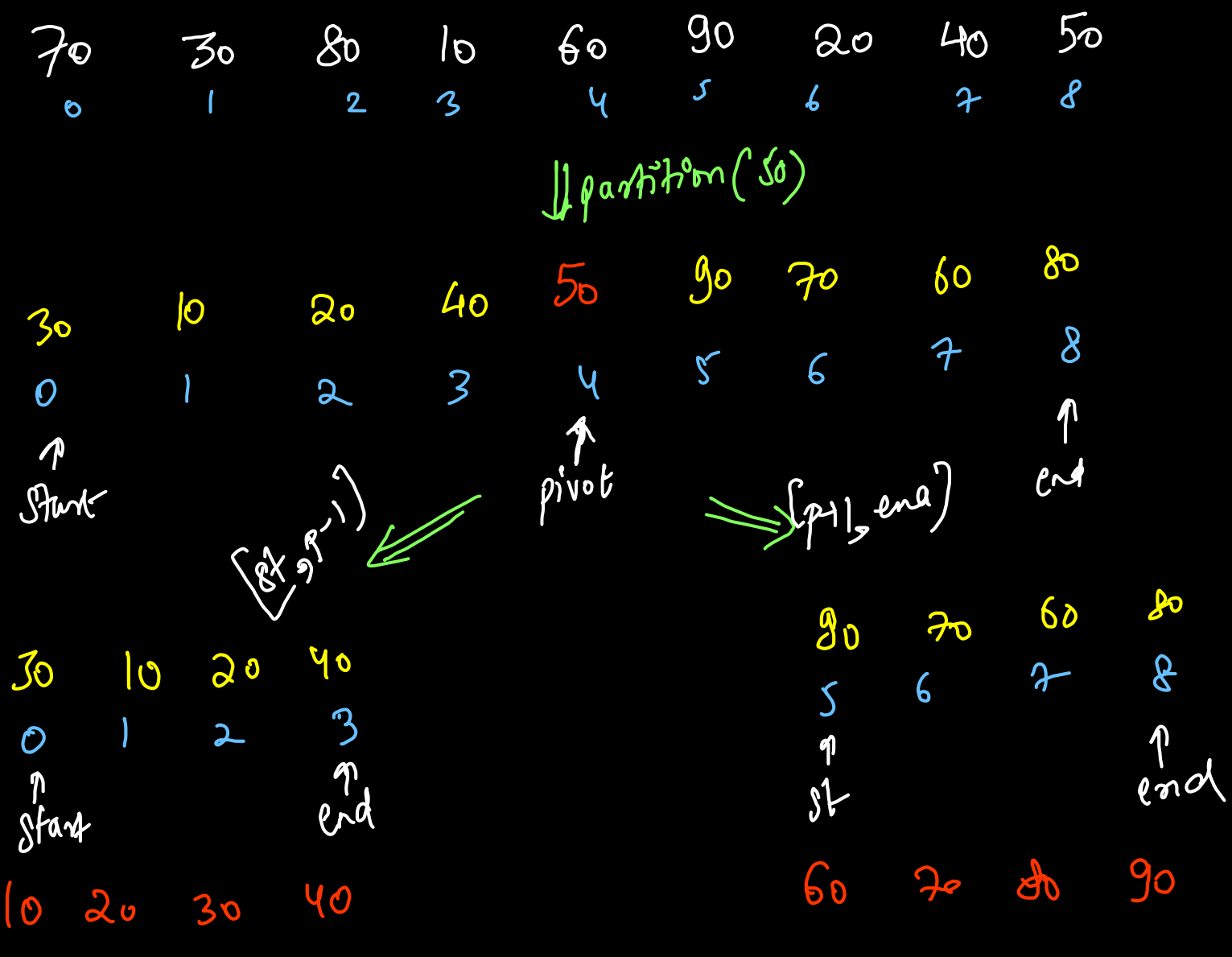
70 60 90 80 50 > 40

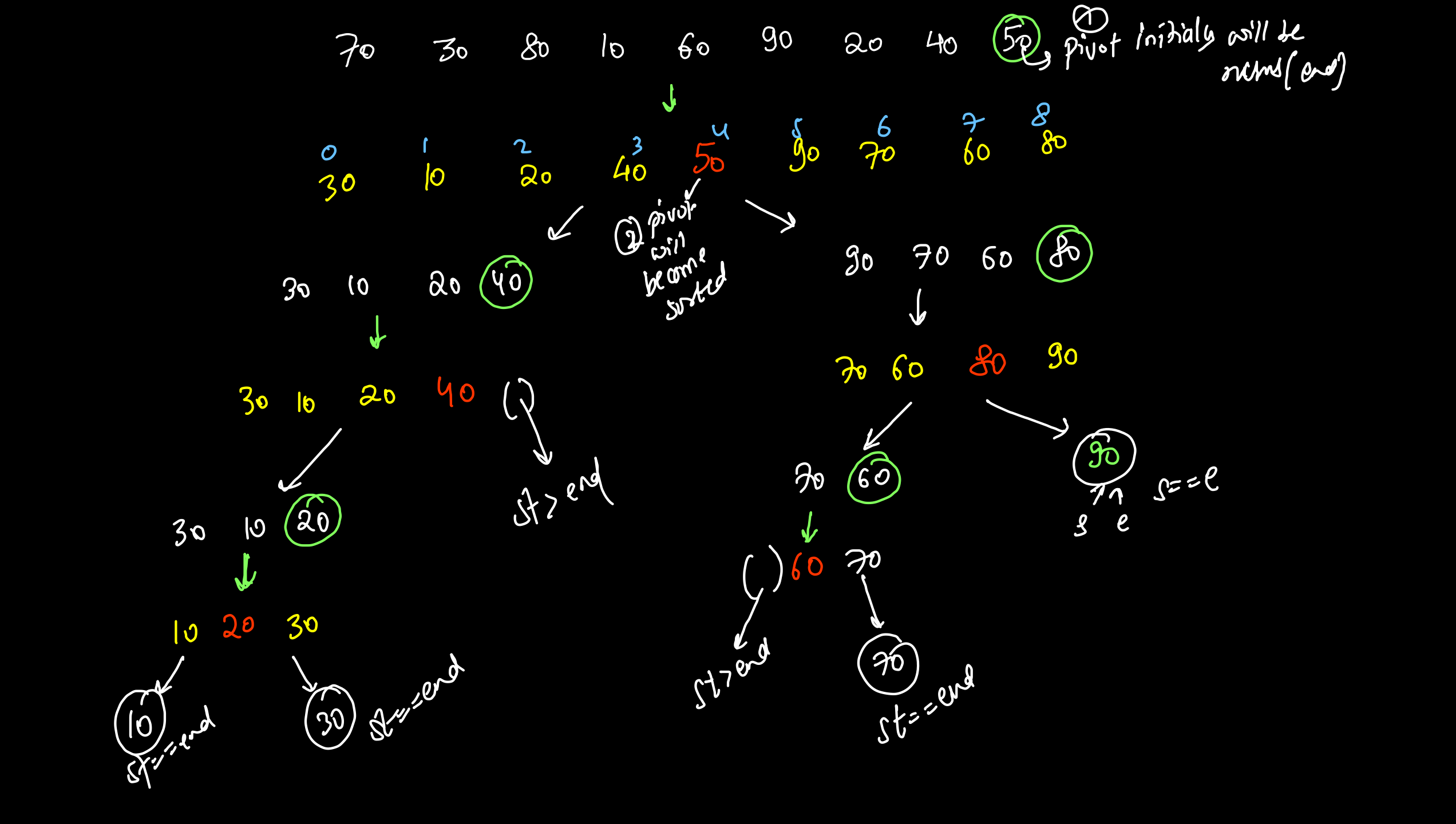
pivot must be the rightmost element!

```java
public static void partition(int[] arr, int pivot) {
    int left = 0, right = 0;
    while (right < arr.length) {
        if (arr[right] <= pivot) {
            swap(arr, right, left);
            left++;
            right++;
        } else {
            right++;
        }
    }
}
```

You, 4 days ago • Added Partitioning

# Quick Sort


70 30 80 10 60 90 20 40 50
0 1 2 3 4 5 6 7 8
partition(50)
30 10 20 40 50 90 70 60 80
0 1 2 3 4 5 6 7 8
start
pivot
end
[st, p-1)
[p+1, end]
30 10 20 40
0 1 2 3
start
end
10 20 30 40
90 70 60 80
5 6 7 8
st
end
60 70 80 90


70 30 80 10 60 90 20 40 50
pivot initially will be arr[end]
0 1 2 3 4 5 6 7 8
30 10 20 40 50 90 70 60 80
pivot will become sorted
30 10 20 40
30 10 20 40
30 10 20
30 10 20
10 20 30
10
st==end
30
st==end
st>end
90 70 60 80
70 60 80 90
70 60
70 60
60 70
st>end
70
st==end
90
s e
s==e

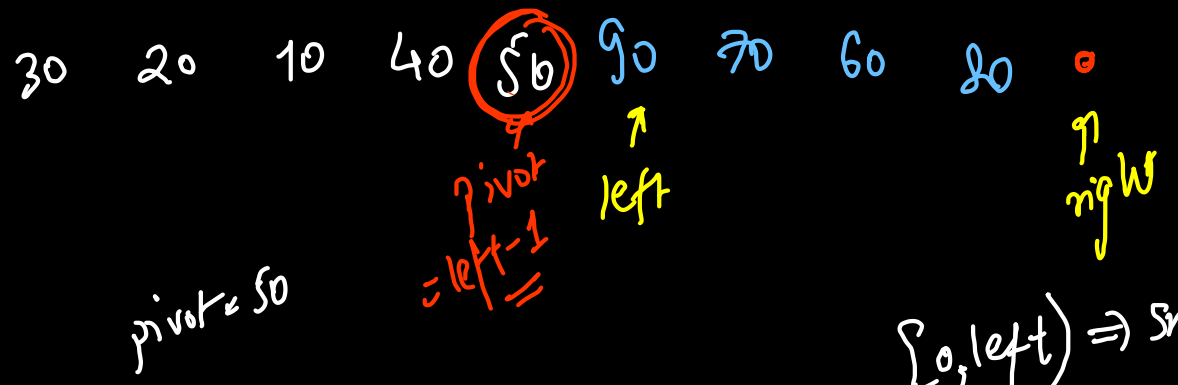
```java
public int partition(int[] nums, int start, int end){
    int pivot = nums[end];

    int left = start, right = start;
    while(right <= end){
        if(nums[right] <= pivot){
            int temp = nums[left];
            nums[left] = nums[right];
            nums[right] = temp;
            left++; right++;
        } else {
            right++;
        }
    }

    return left - 1; // this is the last index of <= region
}
```

swap(left, right)

```java
public void quickSort(int[] nums, int start, int end){
    if(start >= end) return;
    // either 0 elements or 1 array: already sorted

    int pivot = partition(nums, start, end);
    quickSort(nums, start, pivot - 1);
    quickSort(nums, pivot + 1, end);
}

public int[] sortArray(int[] nums) {
    quickSort(nums, 0, nums.length - 1);
    return nums;
}
```

preorder

divide & conquer




[0, left) ⇒ smaller

[left, right) ⇒ larger

[right, n) ⇒ unexplored

# Time Complexity

## Average Case or Best Case (pivot == mid)

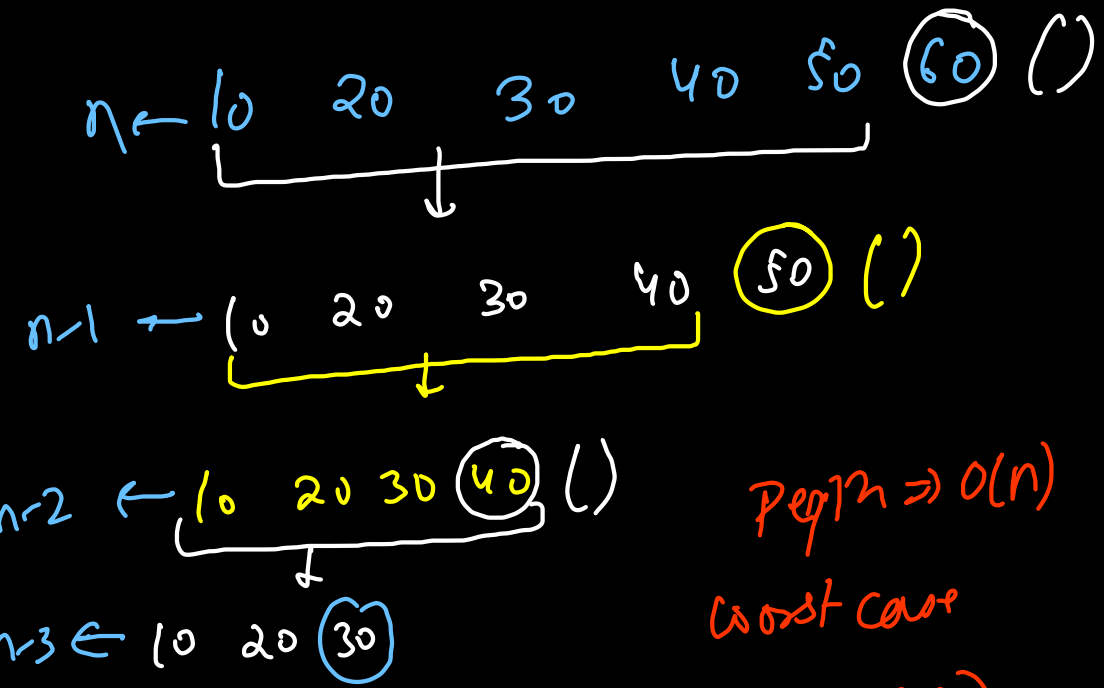

Depth $\Rightarrow O(\log_2 n)$

Avg case $\Rightarrow \theta(n \log_2 n)$

Best case $\Rightarrow \Omega(n \log n)$

## Worst case (pivot == min$^m$/max$^m$)

- already sorted
- (almost sorted)
- reverse sorted




Depth $\Rightarrow O(n)$

Worst case

$\Rightarrow O(n^2)$

TLE $\Rightarrow n = 10^5$

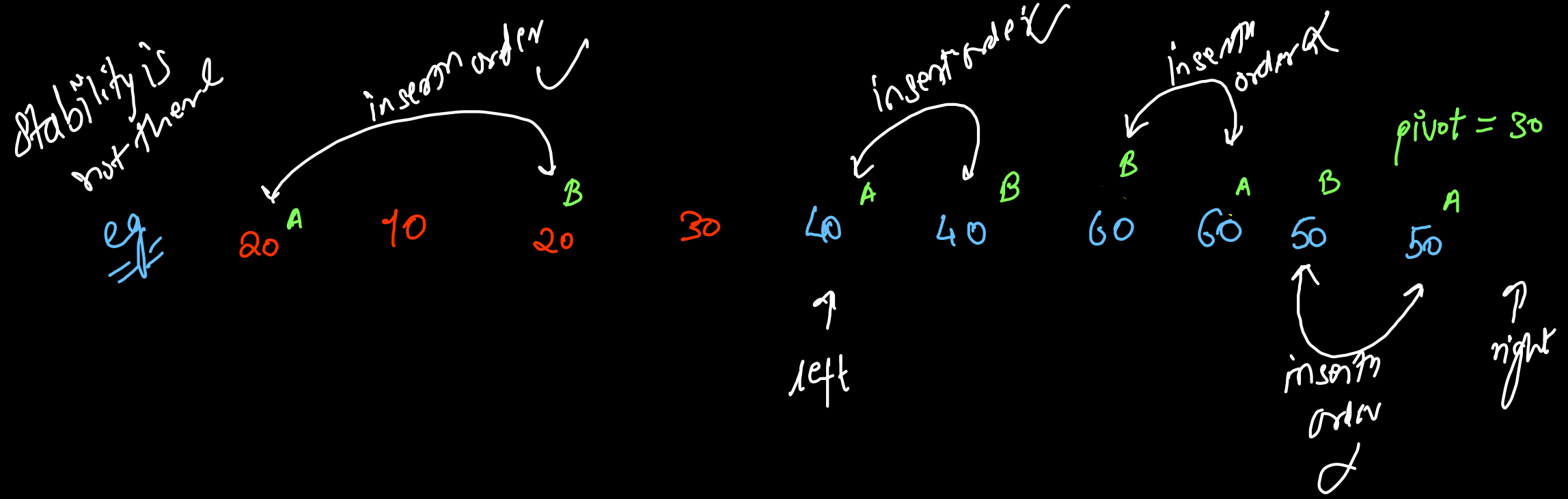
Space Complexity :→ Auxiliary/Extra space ⇒ O(1) → Inplace ↳ Sorted the inputted array

Recursion Callstack ⇒ $O(\log_2 n)$ avg case

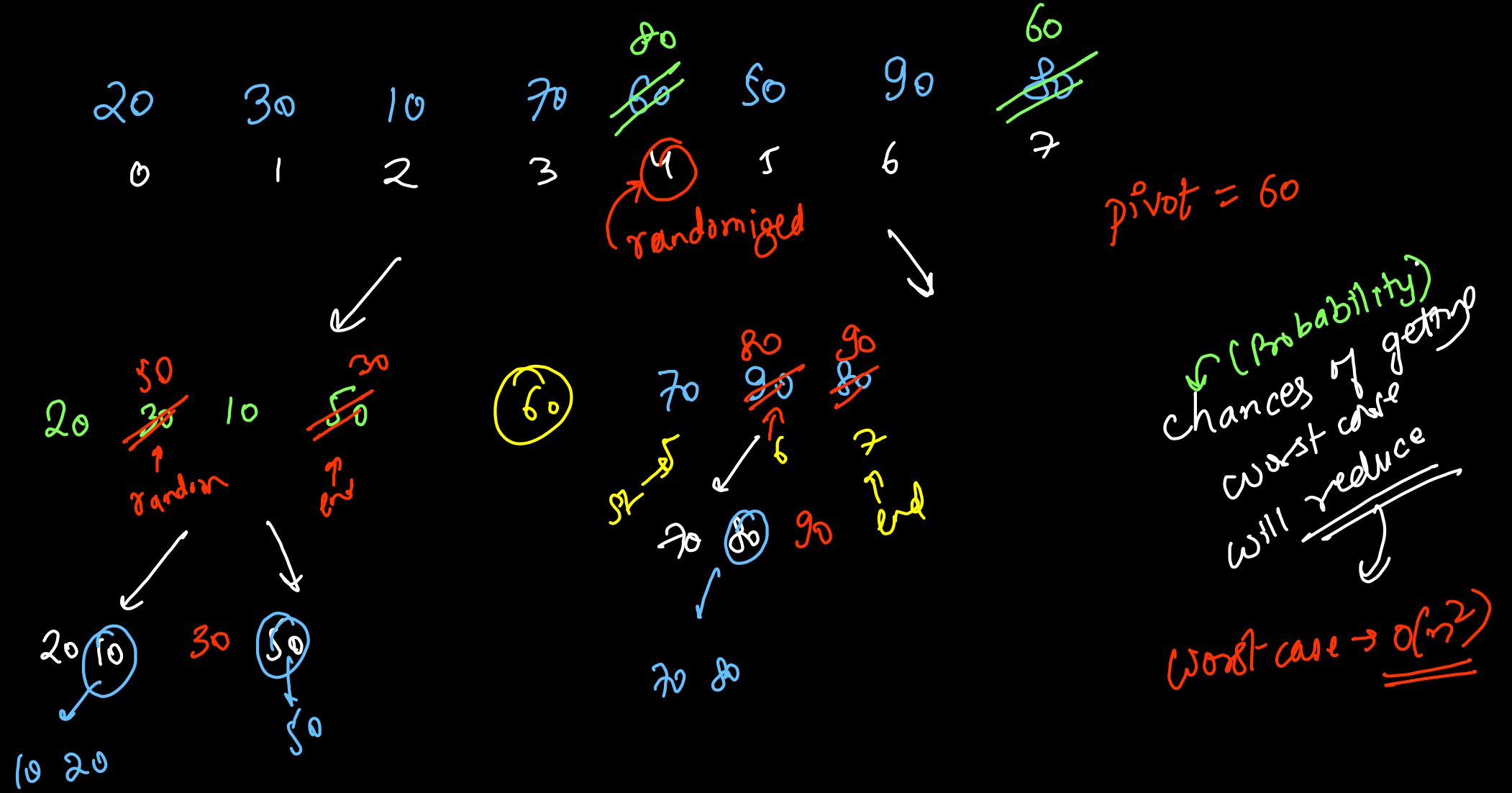
$O(n)$ worst case !

Stability :→ Partitioning Logic → does not maintain relative order

↳ Quick Sort → not stable


Stability is not there
eg
insertion order
20 A
10
20 B
30
insertion order
40 A
40 B
left
insertion order
60 B
60 A
50 B
50 A
pivot = 30
insertion order
right

# Randomized Quick Sort



5 6 7 8 9

st end

double [0, 1)

Math.random() * ( end - start + 1) + start

[0, 5) + 5 ⇒ [5, 10)

```java
public void swap(int[] nums, int p1, int p2){
    int temp = nums[p1];
    nums[p1] = nums[p2];
    nums[p2] = temp;
}

public void randomize(int[] nums, int start, int end){
    int randomIdx = (int)(Math.random() * (end-start+1) + start);
    swap(nums, randomIdx, end);
}
```

```java
public int partition(int[] nums, int start, int end){
    randomize(nums, start, end);

    int pivot = nums[end];
    int left = start, right = start;
    while(right <= end){
        if(nums[right] <= pivot){
            swap(nums, left, right);
            left++; right++;
        } else {
            right++;
        }
    }

    return left - 1; // this is the last index of <= region
}
```

```java
public void quickSort(int[] nums, int start, int end){
    if(start >= end) return;
    // either 0 elements or 1 array: already sorted

    int pivot = partition(nums, start, end);
    quickSort(nums, start, pivot - 1);
    quickSort(nums, pivot + 1, end);
}

public int[] sortArray(int[] nums) {
    quickSort(nums, 0, nums.length - 1);
    return nums;
}
```

Randomized Quick Sort

(Improved Version of quick sort)

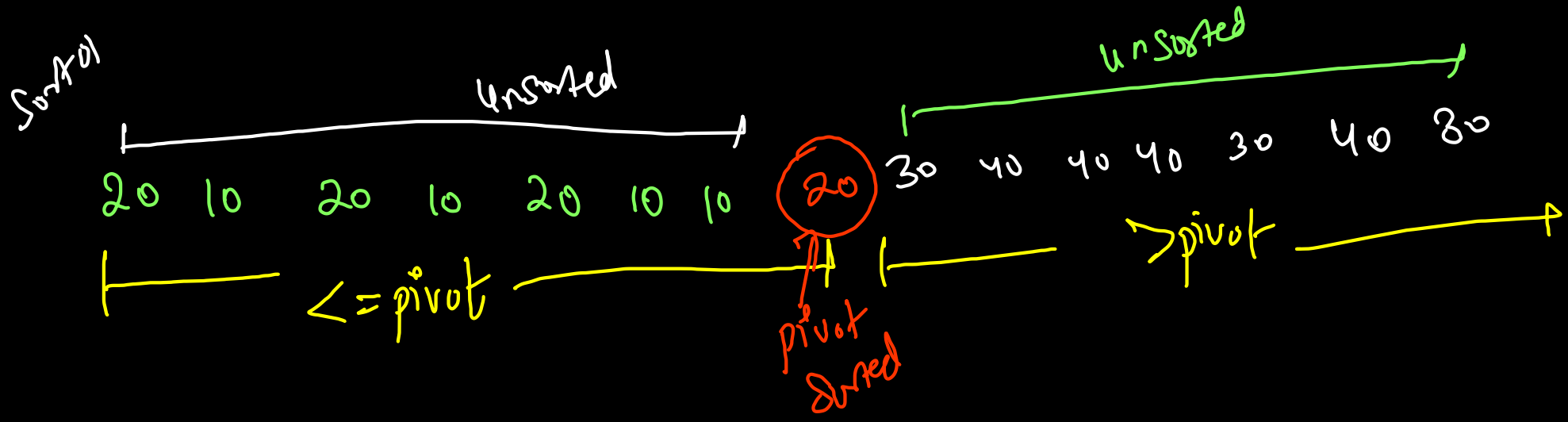
# Three Way Partitioning

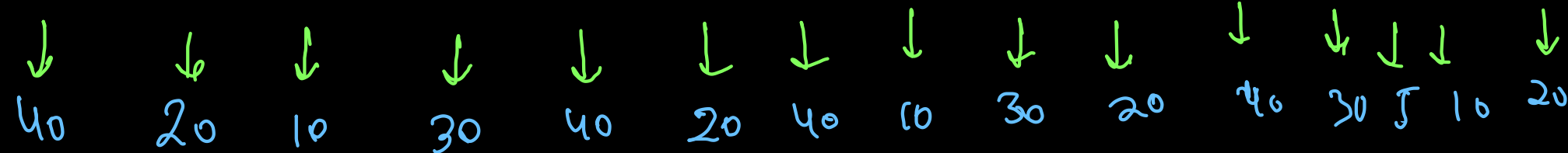
↓

Three way QuickSort

Corner case → Duplicate elements!
occurence ↑ !

40 20 10 30 40 20 40 10 30 20 40 30 10 10 20






Sort 012

< pivot

10 10 5 10

Sort left

= pivot

20 20 20 20

sorted

fastocc

c

lastocc

> pivot

40 30 40 40 30 40 30

Sort right

Sort (start, fo-1)

Sort (lastocc+1, end)

```java
public void swap(int[] nums, int p1, int p2){
    int temp = nums[p1];
    nums[p1] = nums[p2];
    nums[p2] = temp;
}

public void randomize(int[] nums, int start, int end){
    int randomIdx = (int)(Math.random() * (end-start+1) + start);
    swap(nums, randomIdx, end);
}
```

```java
public int[] partition(int[] arr, int start, int end){
    int left = start, mid = start, right = end;

    randomize(arr, start, end);
    int pivot = arr[end];

    while (mid <= right) {
        if (arr[mid] < pivot) {
            swap(arr, left, mid);
            left++;
            mid++;
        } else if (arr[mid] == pivot) {
            mid++;
        } else {
            swap(arr, right, mid);
            right--;
        }
    }

    return new int[]{left, right};
    // first & last occurence of pivot
}
```

```java
public void quickSort(int[] nums, int start, int end){
    if(start >= end) return;
    // either 0 elements or 1 array: already sorted

    int[] pivot = partition(nums, start, end);
    int firstOcc = pivot[0], lastOcc = pivot[1];

    quickSort(nums, start, firstOcc - 1);
    quickSort(nums, lastOcc + 1, end);
}

public int[] sortArray(int[] nums) {
    quickSort(nums, 0, nums.length - 1);
    return nums;
}
```

already sorted corner case

Avg → O(log n)
worst case
→ O(n)

Partition
↓
O(n)

duplicate corner case

Time → O(n log n) average
O(n²) worst

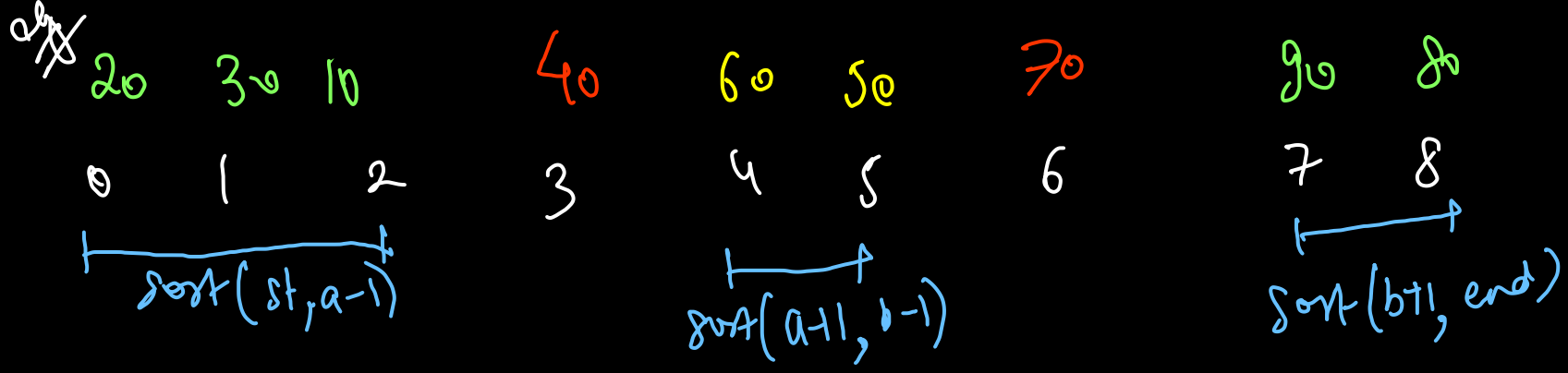
chances of worst case are very less!

# Dual-Pivot Partitioning based quick Sort

20 30 20 50 90 10 40 60 70 50 30

< a | a | [a,b] | b | > b

< 40 | [40,70] | > 70



Inbuilt Sorting in Java()

Tim Sort

Arrays.sort(arr) → by default → increasing order

↳ Combination of Insertion sort & merge sort

N < 16 :→ Insertion Sort

↳ Divide & Conquer →

+ Recursion → Overhead ↑

N > 16 :→ Merge Sort

| Case | Complexity |
|---|---|
| Best Case | O(n) |
| Average Case | O(n*log(n)) |
| Worst Case | O(n*log(n)) |
| Space | O(n) |
| Stable | YES |
| In-Place Sorting | NO, as it requires extra space |

# Merge Sort vs Quick Sort

## (1) Similarities

(i) divide & conquer

(ii) best/average time → O(nlogn)

(iii) recursive call stack space → O(logn)

## (2) differences

| Merge Sort | Quick Sort |
|---|---|
| (i) worst case → O(nlogn) | worst case → O(n²) quadratic |
| (ii) space → O(n) extra space, not inplace | space → O(1) inplace |
| (iii) stable algorithm → relative order maintained | relative order → not maintained ⟷ not stable |
| (iv) Merging → Postorder | Partitioning → Preorder |
| (v) usually preferred for linked list & locality of reference | usually preferred for arrays & strings. |

divide & conquer

best/average time → O(nlogn)

recursive call stack space → O(logn) avg.

# Counting Based Sorting Algorithms

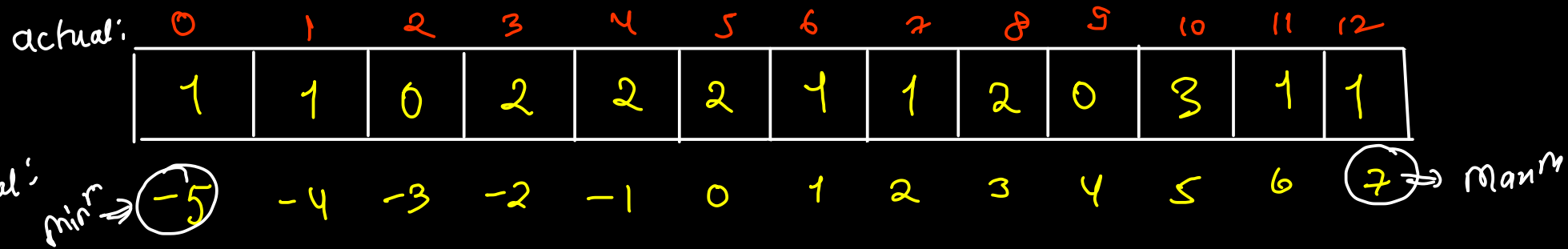
1) Count or Counting Sort → Range of Elements should be less
   - $Max^m - min^m \leq$ Array Size
   - $\approx 10^6 - 10^7$
2) Radix Sort
3) Bucket Sort

# Count Sort or Counting Sort

arr:

| 5 | -2 | 5 | -4 | 6 | 3 | 1 | 0 | -1 | -2 | -5 | +7 | -1 | 3 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

(1) Frequency Array { O(Range) = O(max - min + 1) }




actual idx = logical idx − min

logical idx = actual idx + min

```
int max = 50000, min = -50000; // according to constraint
int[] freq = new int[max - min + 1];

// 1. Fill the Frequency Array
for(int val: nums) freq[val - min]++;
```

```java
public int[] sortArray(int[] nums) {
    int max = 50000, min = -50000; // according to constraint
    int[] freq = new int[max - min + 1];

    // 1. Fill the Frequency Array
    for(int val: nums) freq[val - min]++;
```

→ O(n)

Time ⇒ O(n+k)

Space ⇒ O(n+k)

idx idx idx idx idx idx idx

res : -5 -4 -2 -2 -1 -1 2 3 3 5 5 5 6 7

(Sorted)

```java
int[] res = new int[nums.length];
int idx = 0;

for(int actualIdx = 0; actualIdx < freq.length; actualIdx++){
    int val = actualIdx + min;
    for(int count = 0; count < freq[actualIdx]; count++){
        res[idx++] = val;
    }
}

return res;
```

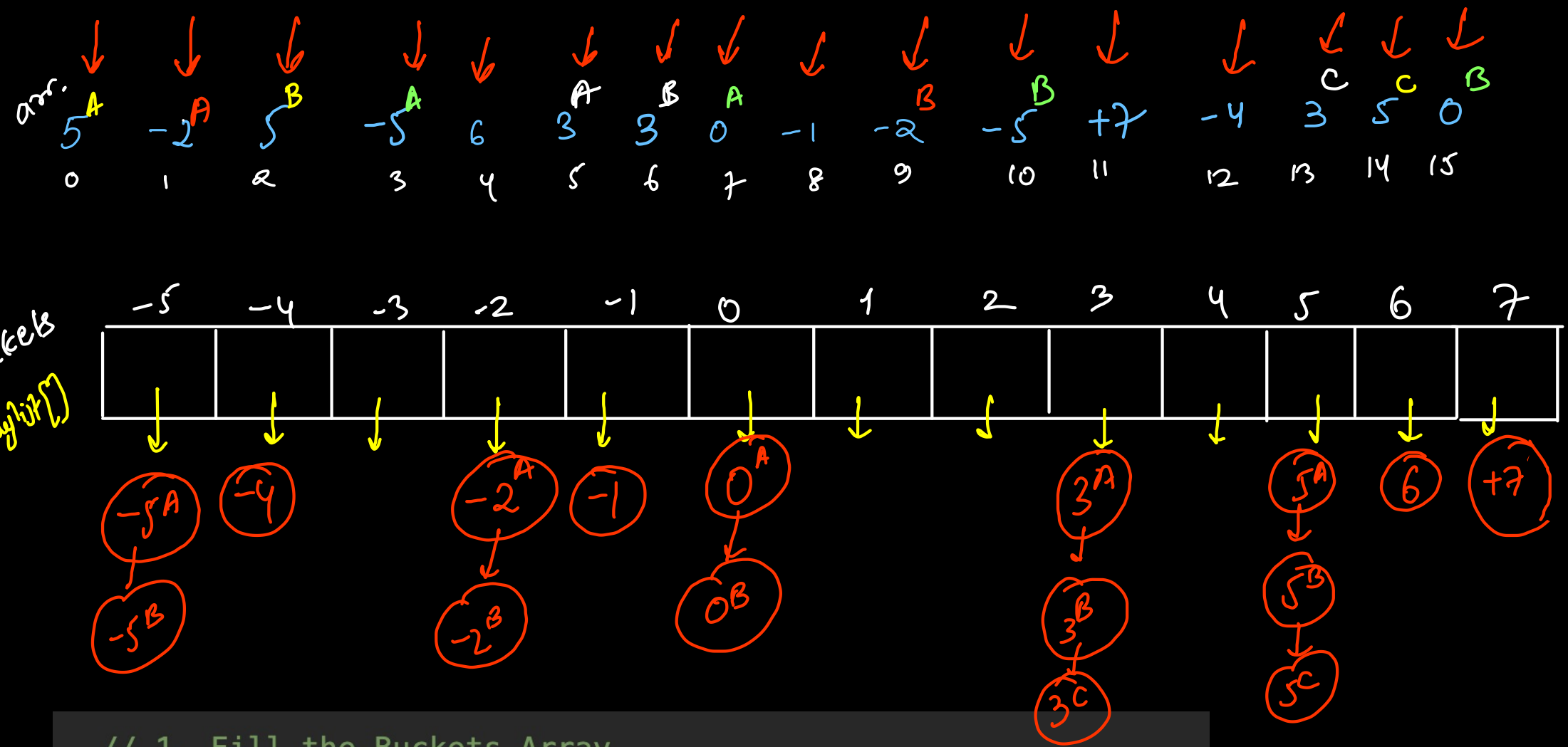
→ O(range) + O(n) = O(k+n)

2 independent loops

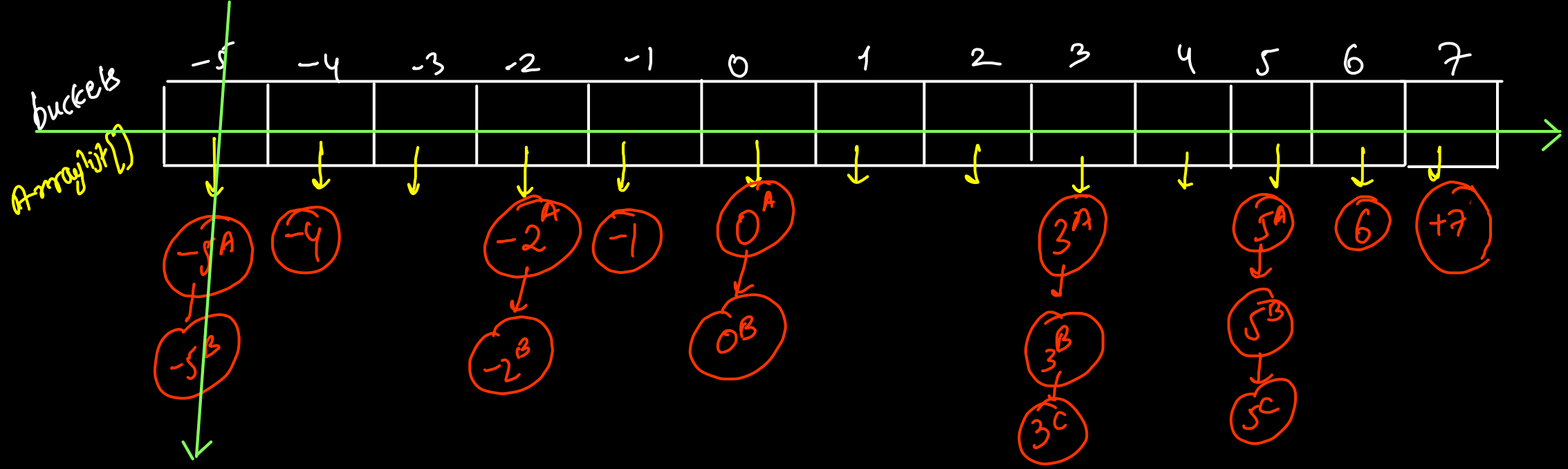
actualIdx actualIdx a idx a idx a idx

0→-5 1→-4 2→-3 3→-2 4→-1 5→0 6→1 7→2 8→3 9→4 10→5 11→6 12→7

| 1 | 1 | 0 | 2 | 2 | 2 | 1 | 1 | 2 | 0 | 3 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|




```java
// 1. Fill the Buckets Array
ArrayList<Integer>[] buckets = new ArrayList[max - min + 1];
for(int idx = 0; idx < buckets.length; idx++){
    buckets[idx] = new ArrayList<>();
}
```




$-5^A$ $-5^B$ $-4$ $-2^A$ $-2^B$ $-1$ $0^A$ $0^B$ $3^A$ $3^B$ $3^C$ $5^A$ $5^B$ $5^C$ $6$ $7$

Sorted ?
Stable ?

```java
public int[] sortArray(int[] nums) {
    int max = 50000, min = -50000; // according to constraint

    // 1. Fill the Buckets Array
    ArrayList<Integer>[] buckets = new ArrayList[max - min + 1];
    for(int i = 0; i < buckets.length; i++){
        buckets[i] = new ArrayList<>();
    }

    for(int val: nums) buckets[val - min].add(val);


    // 2. Fill the Result Sorted Array
    int[] res = new int[nums.length];
    int idx = 0;

    for(int actualIdx = 0; actualIdx < buckets.length; actualIdx++){
        for(Integer val: buckets[actualIdx]){
            res[idx++] = val;
        }
    }

    return res;
}
```

O(max - min + 1) = O(range) = O(k)

O(n)

→ loop on all buckets (left + right)

O(range)

independent

→ loop on the current bin (top to bottom)

O(n)

O(range + n) = O(k + n)

Time ⇒ O(n + k)

Space ⇒ O(n + k)

# Applications (Frequency Sort)

① Sort Array by Frequency — LC 1636

→ Different freq → Inc order of frequency

→ Same freq → decreasing order of value

| 4 | 5 | -3 | 4 | -2 | 4 | 5 | 2 | 1 | 2 | 0 | 1 | 4 | 5 | -2 | 4 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

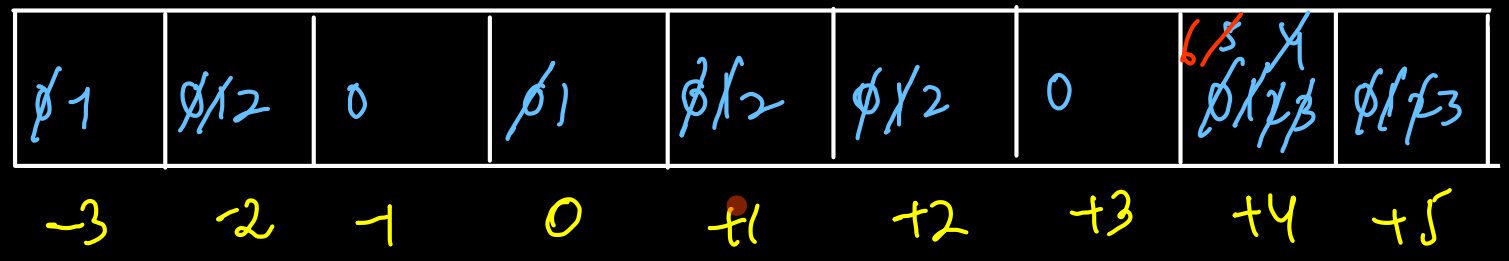
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ϕ/2 | ϕ/2 | 0 | ϕ1 | ϕ/2 | ϕ/2 | 0 | ϕ/3 | ϕ/3 |
| -3 | -2 | -1 | 0 | +1 | +2 | +3 | +4 | +5 |

① val → frequency

freq

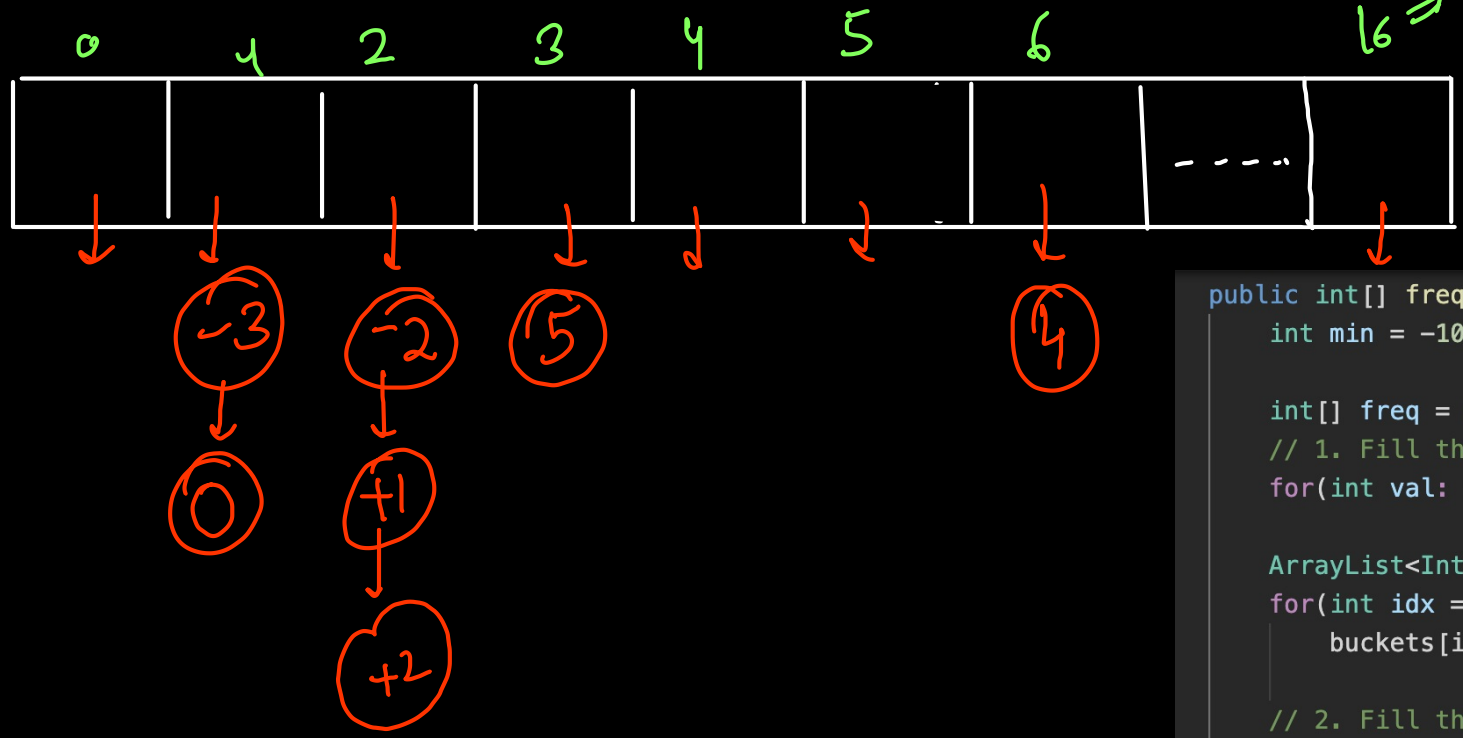
0 +2 +2 +1 +1 -2 -2 5 5 5 4 4 4 4 4 4

freq = 1 | freq = 2 | freq = 3 | freq = 6




val → frequency

frequency Array




Buckets Array

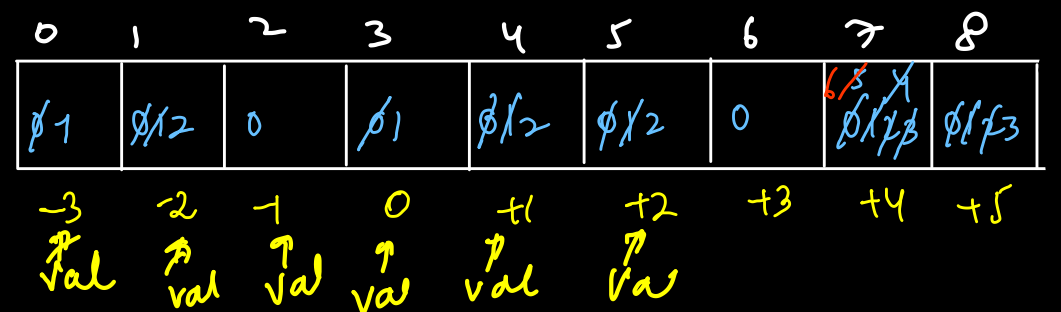
freq → val

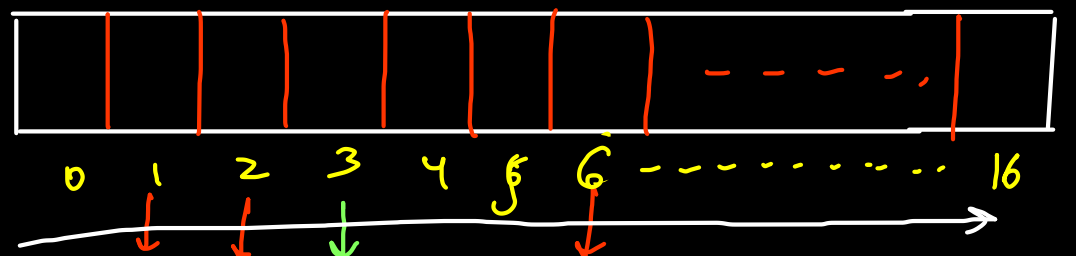
(n+1) buckets

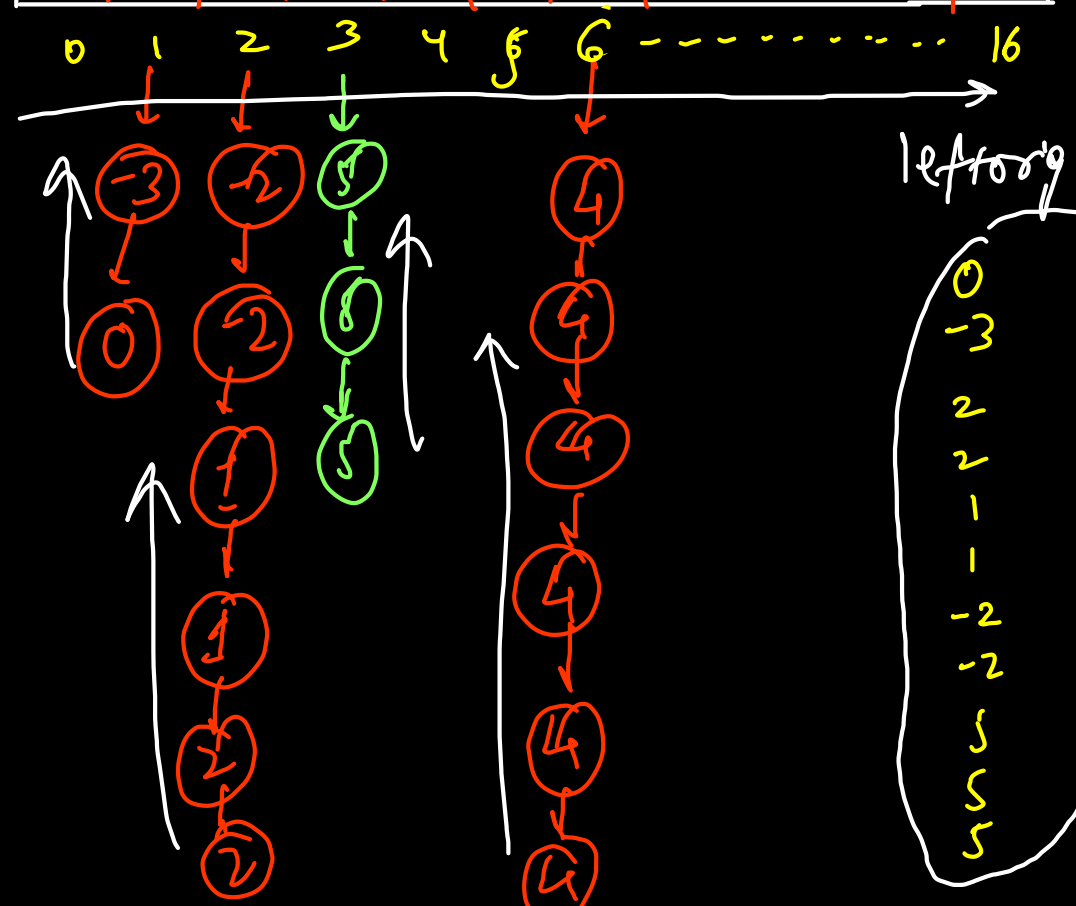
ArrayList[]

```java
public int[] frequencySort(int[] nums) {
    int min = -100, max = 100;

    int[] freq = new int[max - min + 1];
    // 1. Fill the Frequency Array
    for(int val: nums) freq[val - min]++;

    ArrayList<Integer>[] buckets = new ArrayList[nums.length + 1];
    for(int idx = 0; idx < buckets.length; idx++)
        buckets[idx] = new ArrayList<>();

    // 2. Fill the Buckets Array
    for(int val = 0; val < freq.length; val++){
        int count = freq[val];
        buckets[count].add(val);
    }
```

```java
public int[] frequencySort(int[] nums) {
    int min = -100, max = 100;

    int[] freq = new int[max - min + 1];
    // 1. Fill the Frequency Array
    for(int val: nums) freq[val - min]++;

    ArrayList<Integer>[] buckets = new ArrayList[nums.length + 1];
    for(int idx = 0; idx < buckets.length; idx++)
        buckets[idx] = new ArrayList<>();

    // 2. Fill the Buckets Array
    for(int idx = 0; idx < freq.length; idx++){
        int val = idx + min;
        for(int c = 0; c < freq[idx]; c++)
            buckets[freq[idx]].add(val);
    }

    // 3. Fill the Sorted Array
    int idx = 0;
    // Increasing Order of Frequency: Left to Right
    for(int count = 1; count < buckets.length; count++){
        Collections.reverse(buckets[count]);
        // For Same Frequency: Decreasing order: Bottom to Top
        for(Integer val: buckets[count]){
            nums[idx++] = val;
        }
    }

    return nums;
}
```

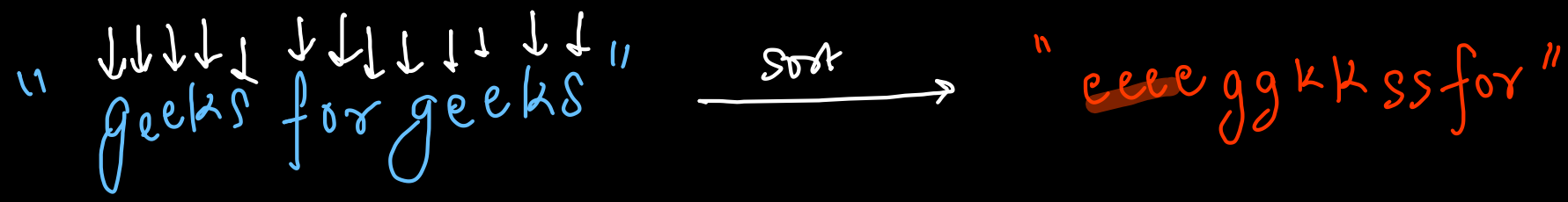


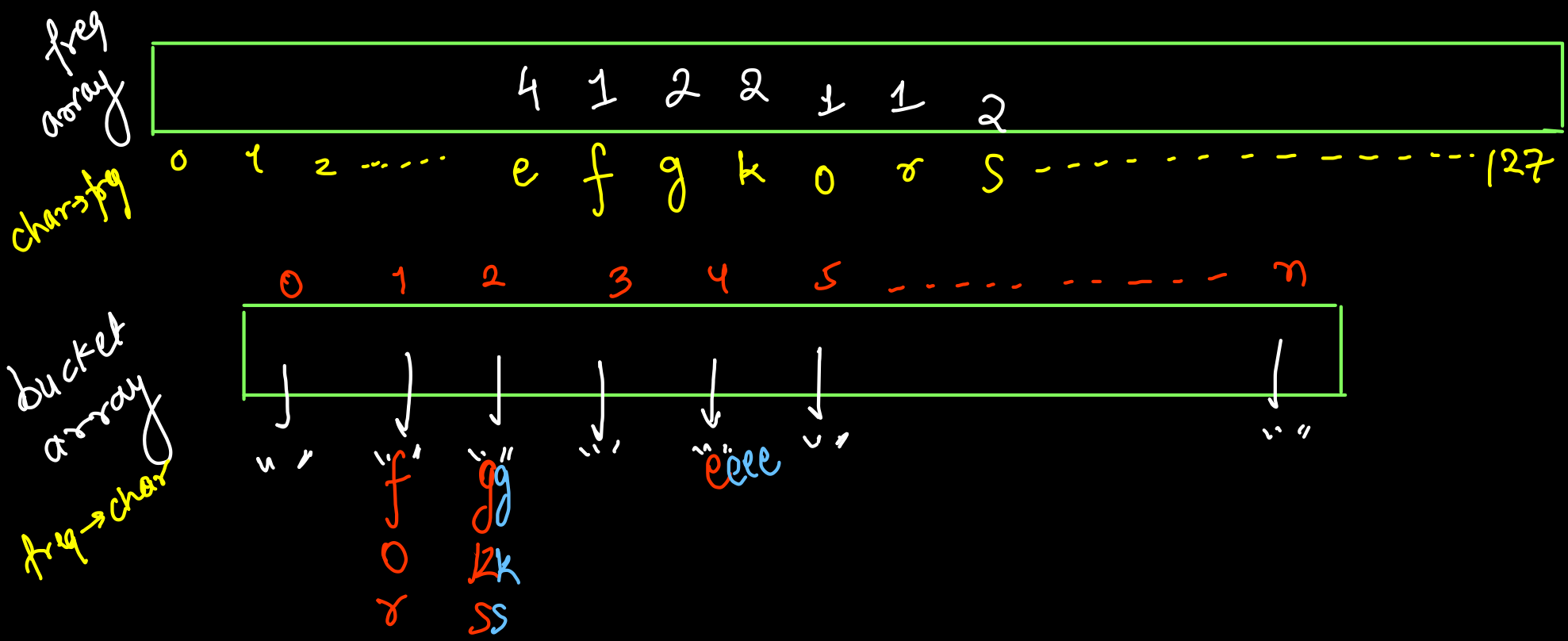





# Sort characters by Frequency

LC 451



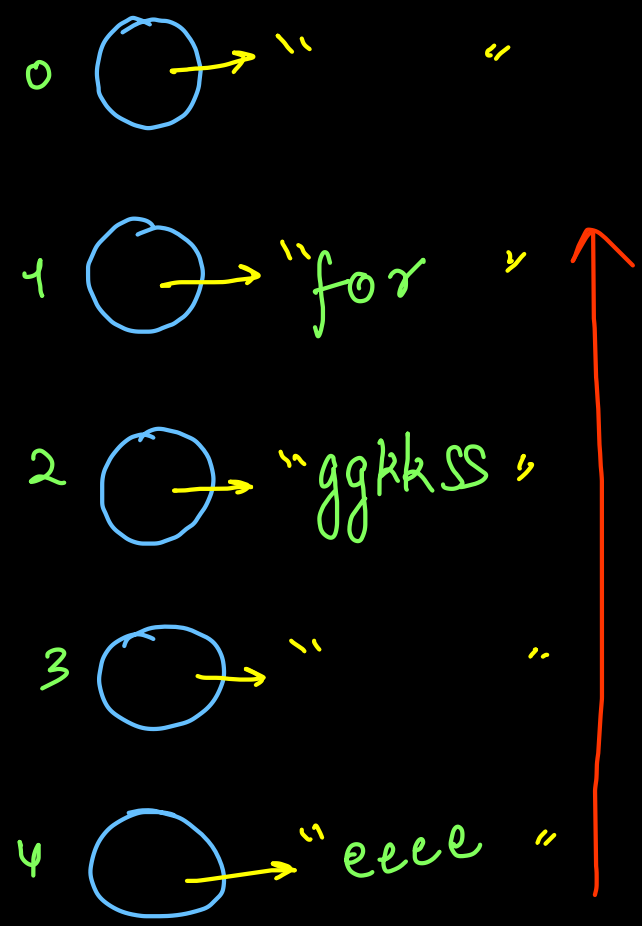



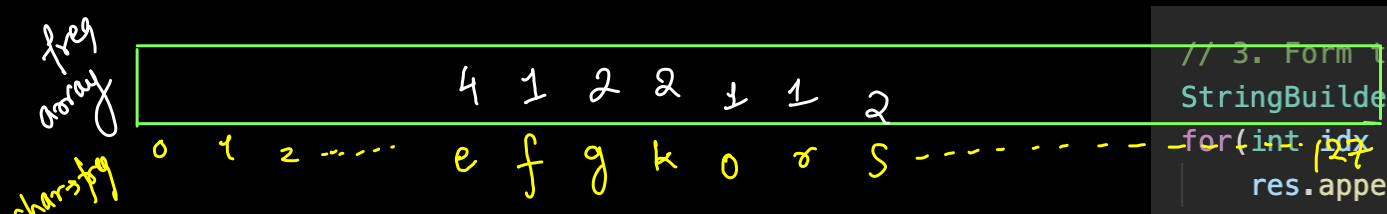
```java
public String frequencySort(String s) {
    // 1. Fill the Frequency Array
    int[] freq = new int[128];
    for(int idx = 0; idx < s.length(); idx++){
        char ch = s.charAt(idx);
        freq[ch]++;
    }

    // 2. Fill the Buckets Array
    StringBuilder[] buckets = new StringBuilder[s.length() + 1];
    for(int idx = 0; idx < buckets.length; idx++){
        buckets[idx] = new StringBuilder();
    }

    for(int idx = 0; idx < 128; idx++){
        char ch = (char)idx;
        for(int c = 0; c < freq[idx]; c++){
            buckets[freq[idx]].append(ch);
        }
    }
```

Time = O(n+k)

Space = O(n+k)

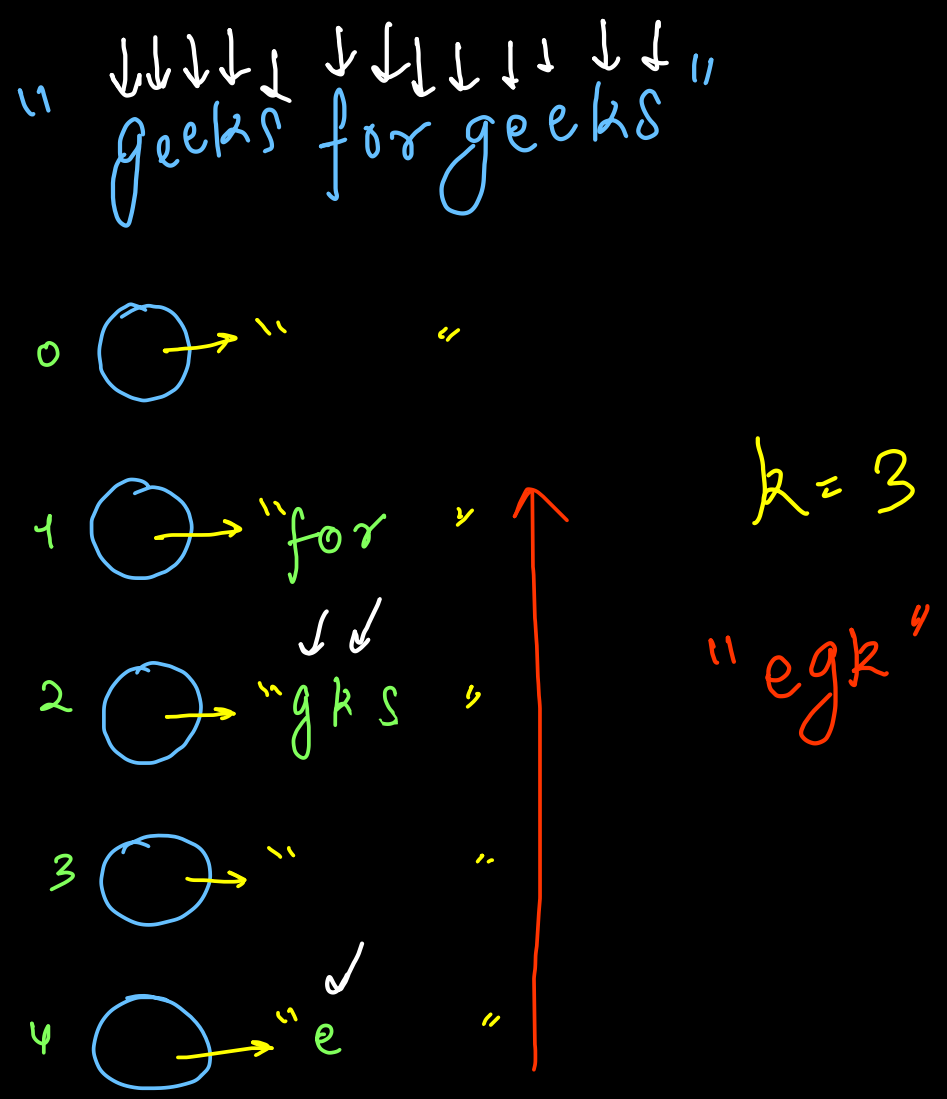
R, n } independent




```java
    // 3. Form the Resultant String (Decreasing order of Frequency)
    StringBuilder res = new StringBuilder();
    for(int idx = buckets.length - 1; idx >=0; idx--){
        res.append(buckets[idx]);
    }

    return res.toString();
```

# Top K Frequent Elements



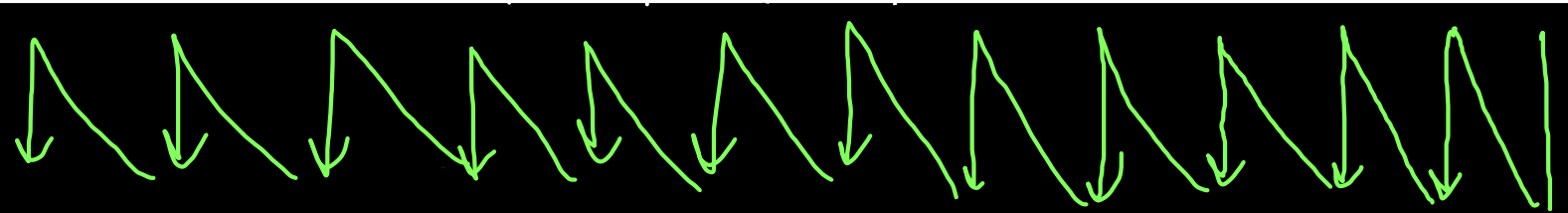

```java
public int[] topKFrequent(int[] nums, int k) {
    int min = -10000, max = 10000;

    int[] freq = new int[max - min + 1];
    // 1. Fill the Frequency Array
    for(int val: nums) freq[val - min]++;

    ArrayList<Integer>[] buckets = new ArrayList[nums.length + 1];
    for(int idx = 0; idx < buckets.length; idx++)
        buckets[idx] = new ArrayList<>();

    // 2. Fill the Buckets Array
    for(int idx = 0; idx < freq.length; idx++){
        int val = idx + min;
        buckets[freq[idx]].add(val);
    }

    int[] res = new int[k];
    int idx = 0;
    // 3. Form the resultant Array
    for(int b = buckets.length - 1; b > 0; b--){
        for(Integer val: buckets[b]){
            res[idx++] = val;
            if(idx == k) return res;
        }
    }
    return res;
}
```

# Count Sort

| arr | 5 | -2 | 5 | -4 | 6 | 3 | 1 | 0 | -1 | -2 | -5 | 7 | -1 | 3 | 5 | 0 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

freq:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 2 | 2 | 2 | 1 | 1 | 2 | 0 | 3 | 1 | 1 |



suff(i) = suff(i+1) + freq(i)

suffix sum freq:

| 17 | 16 | 15 | 15 | 13 | 11 | 9 | 8 | 7 | 5 | 5 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

```java
public int[] sortArray(int[] nums) {
    int max = 50000, min = -50000; // according to constraint
    int[] freq = new int[max - min + 1];

    // 1. Fill the Frequency Array
    for(int val: nums) freq[val - min]++;

    // 2. Convert it to the Suffix Array
    for(int i = freq.length - 2; i >= 0; i--)
        freq[i] += freq[i + 1];
```

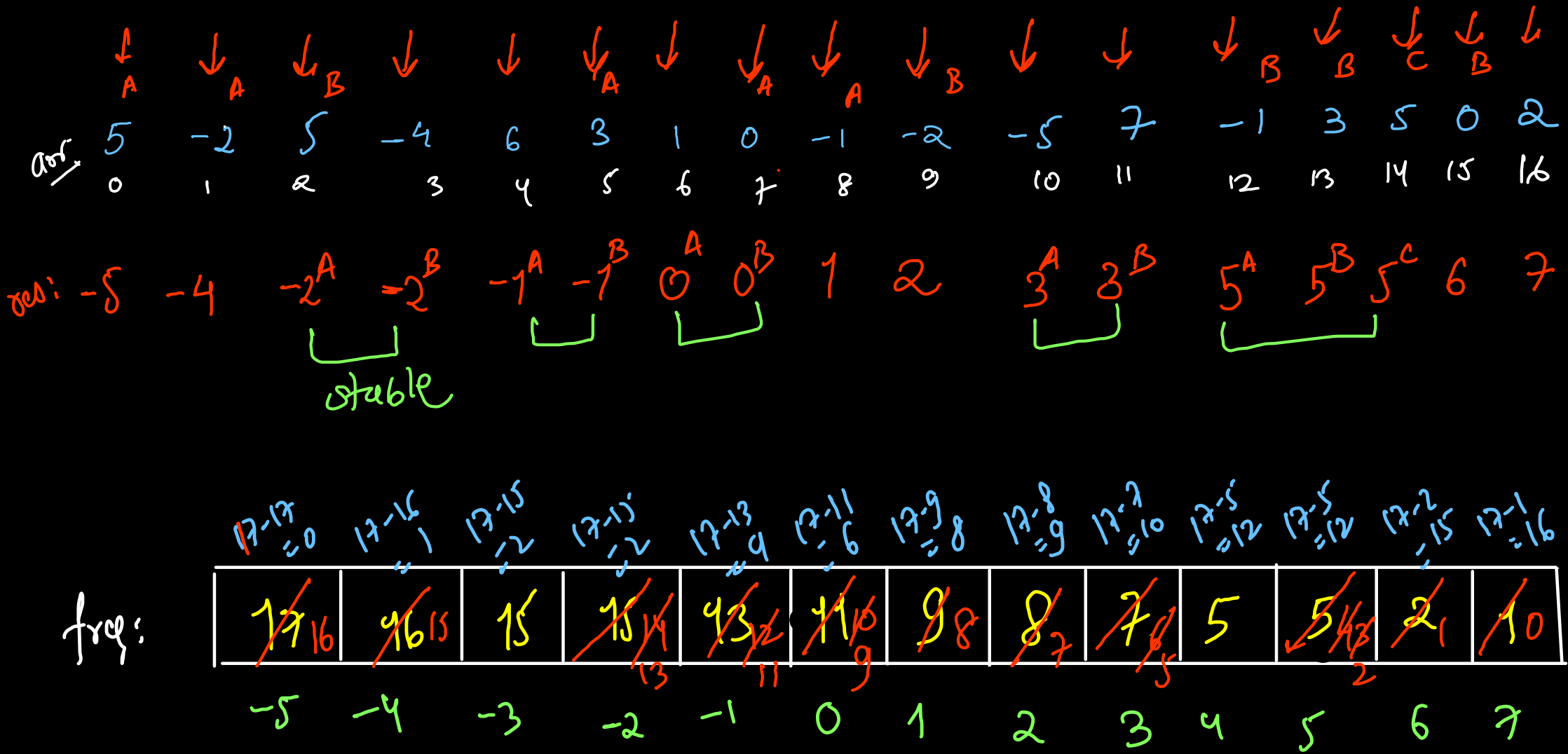
arr: 5 -2 5 -4 6 3 1 0 -1 -2 -5 7 -1 3 5 0 2

idx: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

val: -5 -4 -2 -2 -1 -1 0 0 1 2 3 3 5 5 5 6 6

freq:

| 17-17 =0 | 17-16 =1 | 17-15 =2 | 17-15 =2 | 17-13 =4 | 17-11 =6 | 17-9 =8 | 17-8 =9 | 17-7 =10 | 17-5 =12 | 17-5 =12 | 17-2 =15 | 17-1 =16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 17 | 16 | 15 | 15 | 13 | 11 | 9 | 8 | 7 | 5 | 5 | 2 | 1 |
| -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |


arr: 5 -2 5 -4 6 3 1 0 -1 -2 -5 7 -1 3 5 0 2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
sol: -5 -4 -2A -2B -1A -1B 0A 0B 1 2 3A 3B 5A 5B 5C 6 7
stable
freq:
-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7

```java
public int[] sortArray(int[] nums) {
    int max = 50000, min = -50000; // according to constraint
    int[] freq = new int[max - min + 1];

    // 1. Fill the Frequency Array
    for(int val: nums) freq[val - min]++;

    // 2. Convert it to the Suffix Array
    for(int i = freq.length - 2; i >= 0; i--)
        freq[i] += freq[i + 1];

    // 3. Sort - stable
    int[] res = new int[nums.length];

    for(int val: nums){
        int idx = nums.length - freq[val - min];
        freq[val - min]--; // next occurence to next index
        res[idx] = val;
    }

    return res;
}
```

Time ⇒ O(n+k)

Space ⇒ O(n+k)

Stable sorting Algo !!

Radix Sort → Count sort on bigger range

↳ 0 - Integer.MAX_VALUE range

↳ Intuition ⇒ apply count sort

↳ units place → highest place

rightmost → leftmost

freq array → 10 size

Time ⇒ O(n+10)

↳ countSort

971 549 635 932 590 099 633 547 917

⇓ Sort on unit's place using count sort

590 971 932 633 635 547 917 549 099

⇓ Sort on ten's place using count sort

917 932 633 635 547 549 971 590 099

⇓ Sort on hundred's place

099 547 549 590 633 635 917 932 971

Radix Sort

↳ $O(n \log_{10} n)$

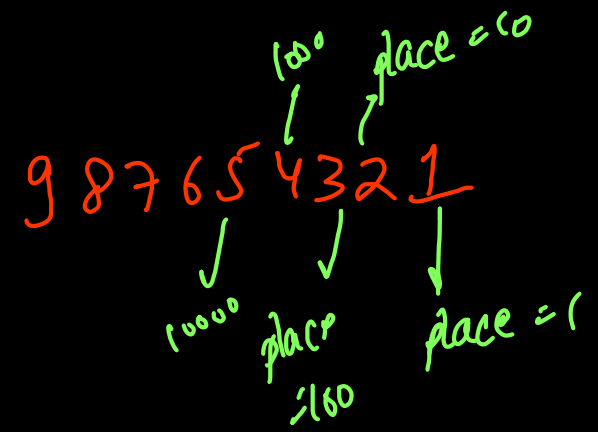
Less time Complexity

maxdigits = 10 (max)

$O(10n)$

↳ linear

```java
public int[] countSort(int[] nums, int place){
    int[] freq = new int[10];

    // 1. Fill the Frequency Array
    for(int val: nums)
    {
        int digit = ?;
        freq[digit]++;
    }

    // 2. Convert it to the Suffix Array
    for(int i = freq.length - 2; i >= 0; i--)
        freq[i] += freq[i + 1];

    // 3. Sort
    int[] res = new int[nums.length];

    for(int val: nums){
        int digit = ?;
        int idx = nums.length - freq[digit];
        freq[digit]--; // next occurence to next index
        res[idx] = val;
    }

    return res;
}
```




val % place

987654321 % 100

= 21

(val / place) % 10

98765 4321 / 100

= (98765 43) % 10 = 3

→ Count sort on basis of a digit.

```java
public int[] countSort(int[] nums, int place){
    int[] freq = new int[10];

    // 1. Fill the Frequency Array
    for(int val: nums)
    {
        int digit = (val / place) % 10;
        freq[digit]++;
    }

    // 2. Convert it to the Suffix Array
    for(int i = freq.length - 2; i >= 0; i--)
        freq[i] += freq[i + 1];

    // 3. Sort
    int[] res = new int[nums.length];

    for(int val: nums){
        int digit = (val / place) % 10;
        int idx = nums.length - freq[digit];
        freq[digit]--; // next occurence to next index
        res[idx] = val;
    }

    return res;
}
```

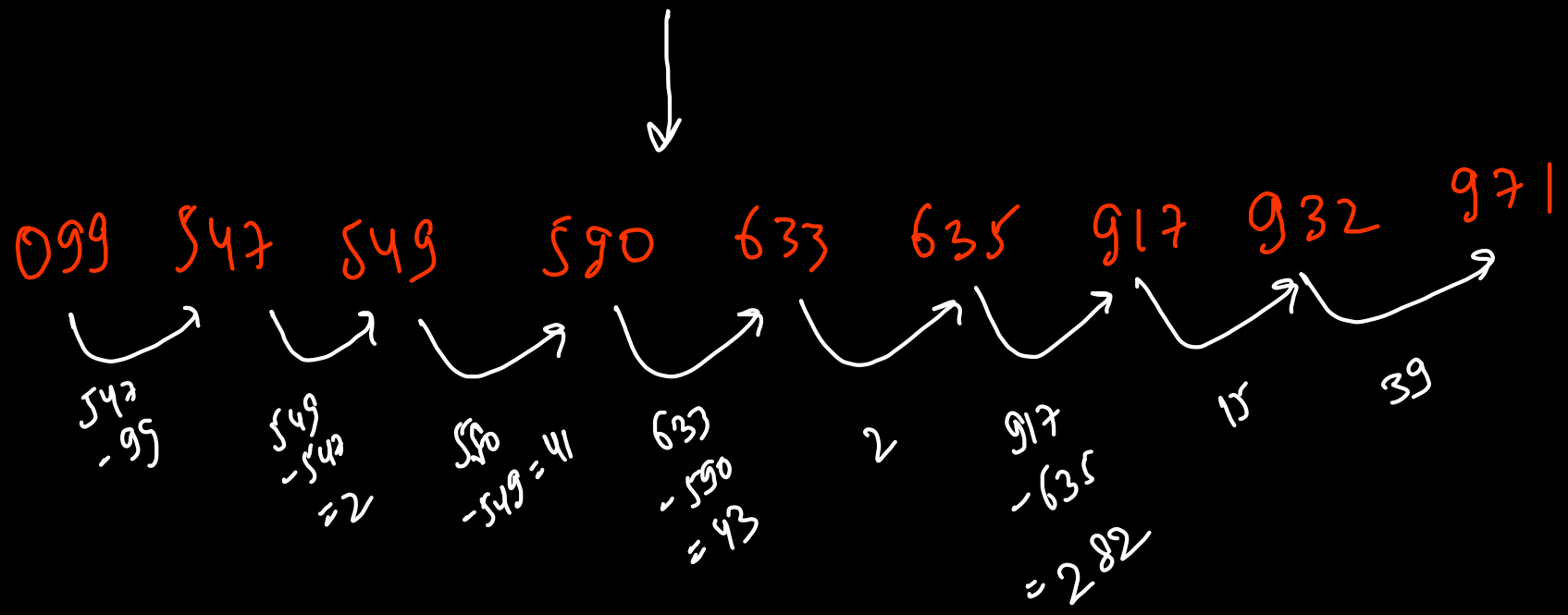
Maximum Gap {#C164}

Hard question

$0 \le nums[i] \le 10^9$ ⇒ unsorted

positive

971 549 635 932 590 099 633 547 917



```java
public int maximumGap(int[] nums) {
    if(nums.length <= 1) return 0;

    Arrays.sort(nums);

    int gap = Integer.MIN_VALUE;
    for(int idx = 0; idx < nums.length - 1; idx++){
        gap = Math.max(gap, nums[idx + 1] - nums[idx]);
    }

    return gap;
}
```

Approach - (1)

nlogn {merge sort} + n

O(nlogn) time

O(n) space
↳ merge Sort

```java
public int[] countSort(int[] nums, int place){
    int[] freq = new int[10];

    // 1. Fill the Frequency Array
    for(int val: nums)
    {
        int digit = (val / place) % 10;
        freq[digit]++;
    }

    // 2. Convert it to the Suffix Array
    for(int i = freq.length - 2; i >= 0; i--)
        freq[i] += freq[i + 1];

    // 3. Sort
    int[] res = new int[nums.length];

    for(int val: nums){
        int digit = (val / place) % 10;
        int idx = nums.length - freq[digit];
        freq[digit]--; // next occurence to next index
        res[idx] = val;
    }

    return res;
}
```

```java
public int maximumGap(int[] nums) {
    if(nums.length <= 1) return 0;

    // Radix Sort = O(10 * N) = Linear Sorting
    for(int place = 1; place <= 1000000000; place = place * 10){
        nums = countSort(nums, place);
    }

    int gap = Integer.MIN_VALUE;
    for(int idx = 0; idx < nums.length - 1; idx++){
        gap = Math.max(gap, nums[idx + 1] - nums[idx]);
    }

    return gap;
}
```

Radix Sort

- Time ⇒ O(n) linear
- Space ⇒ O(n) Space