# Sorting Algorithms

↳ arrangement

$O(n^2)$ worst case

## Basic Sorting Algorithms

1. Bubble Sort
2. Selection Sort
3. Insertion Sort

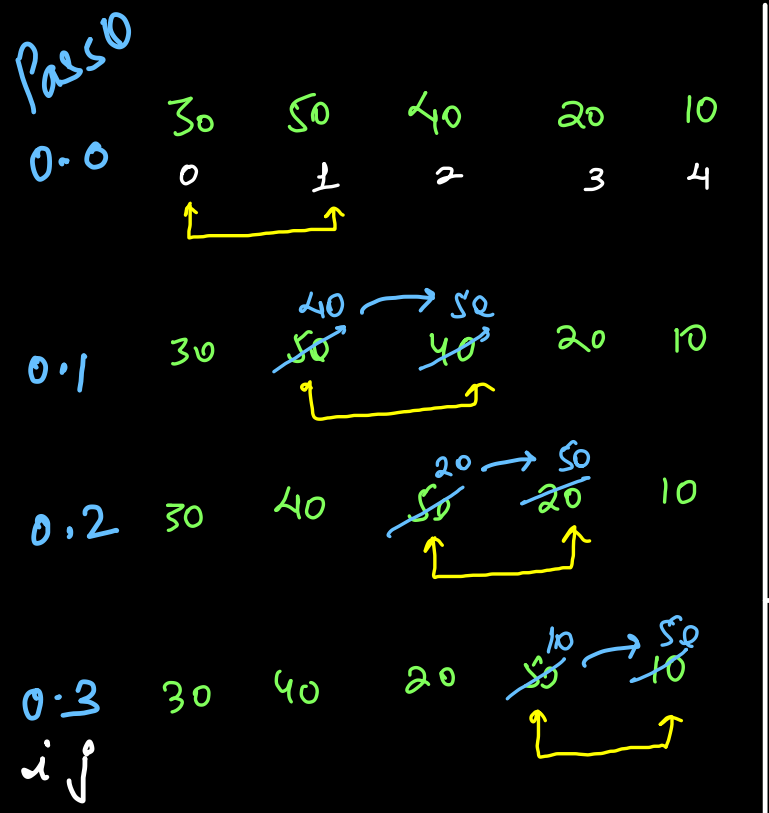
- Stable (Yes/No)
- Inplace (Yes/No)
- Best Case / Avg Case / Worst case time

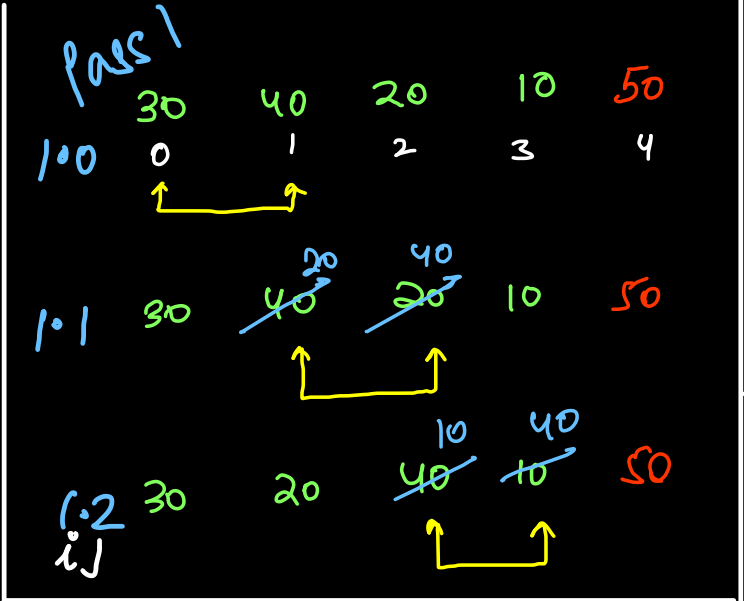
# Increasing order / ascending (min → max)

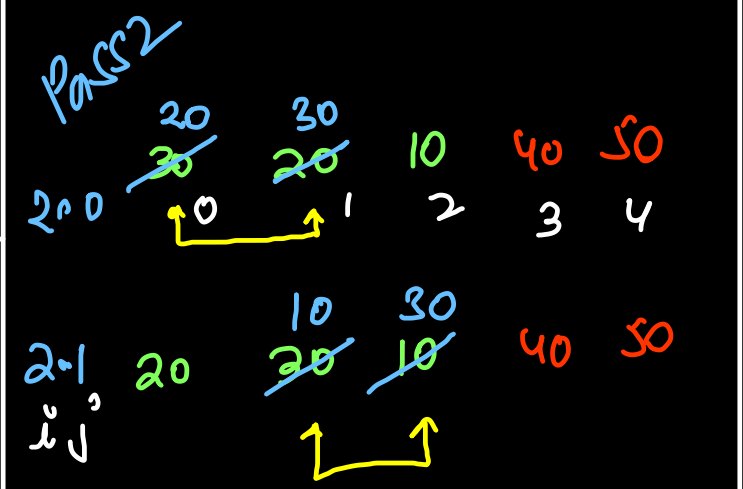
# Decreasing order / decreasing (max → min)

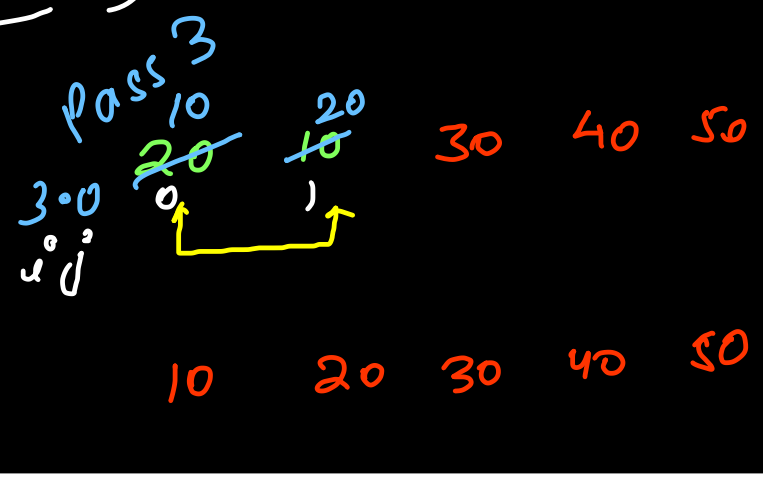
# custom sorting (lambda expression)
↳ comparator

# Bubble Sort (Increasing order)




# total time

$$n-1 + (n-2) + (n-3) + \dots 1$$

$$= \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

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

public static int compare(int[] arr, int l, int r){
    if(arr[l] < arr[r])  return -1; // swap not required
    else if(arr[l] > arr[r]) return +1; // swap required
    else return 0; // equal -> swap not required
}

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

→ multiple passes (multiple for loops)

→ comparison (adjacent indices)

→ swap

→ after each pass (heaviest element will settle at last position)

↳ bubble

→ nested loops

↳ outer loop → pass (i)

↳ inner loop → comparison (j) (& swap)

# time complexity

Best case → O(n) (increasing order)

Avg case → $O(n^2)$ (random)

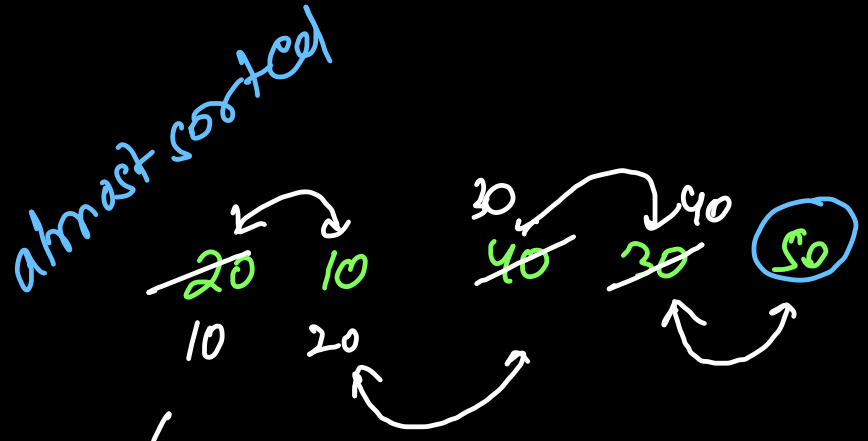
Worst case → $O(n^2)$ (decreasing order)

# Space complexity

(input, output) X

O(1) extra space

↳ in-place (auxiliary data structure)




↳ after each pass

→ Is array sorted?

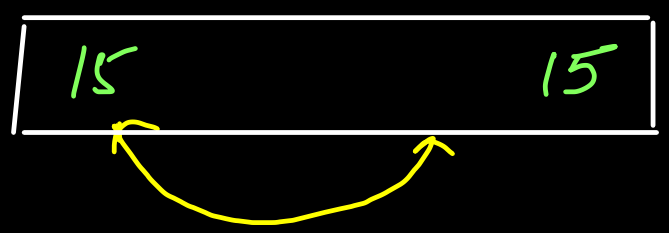
```java
public static void swap(int[] arr, int l, int r){
    int temp = arr[l];
    arr[l] = arr[r];
    arr[r] = temp;
}

public static int compare(int[] arr, int l, int r){
    if(arr[l] < arr[r])  return -1; // swap not required
    else if(arr[l] > arr[r]) return +1; // swap required
    else return 0; // equal -> swap not required
}

public static void bubbleSort(int arr[], int n)
{
    for(int i = 0; i < n - 1; i++){ // pass
        int countSwaps = 0; *
        for(int j = 0; j < n - 1 - i; j++){ // comparison
            if(compare(arr, j, j + 1) > 0){
                swap(arr, j, j + 1);
                countSwaps++; *
            }
        }
        if(countSwaps == 0) break; * (best-case optimize)
    }
}
```

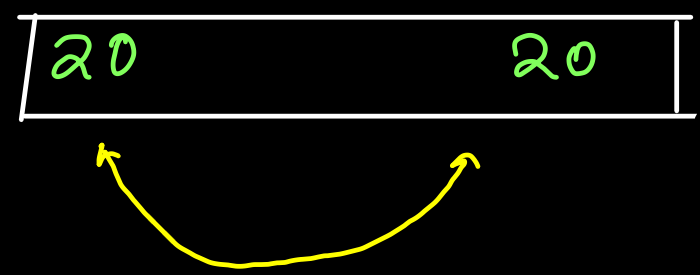
# Stability ~ If two elements are considered equal,
⇒ preserve the order of input

| RNo 1 | RNo 2 | RNo 3 | RNo 4 |
|---|---|---|---|
| Geeta | Ramesh | Sita | Suresh |
| 20 | 15 | 20 | 15 |

↓ sort (increasing order of marks)

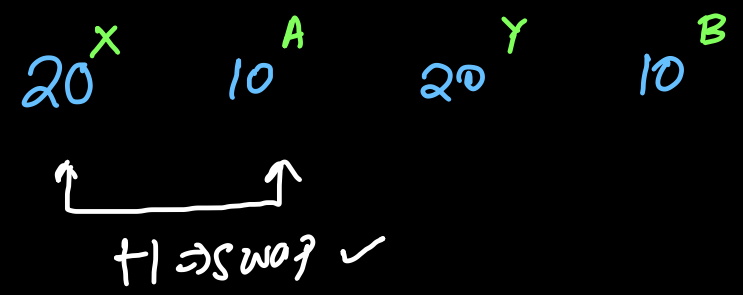
| RNo 2 | RNo 4 | RNo 1 | RNo 3 |
|---|---|---|---|
| Ramesh | Suresh | Geeta | Sita |
| 15 | 15 | 20 | 20 |

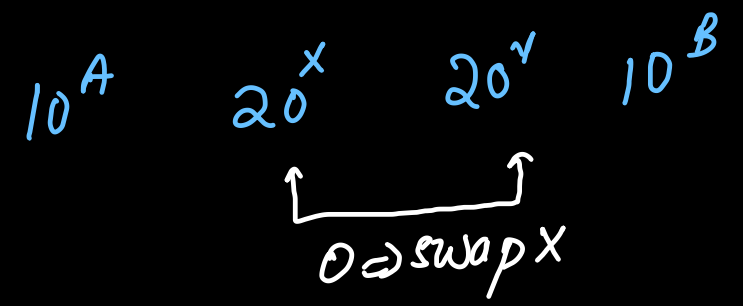


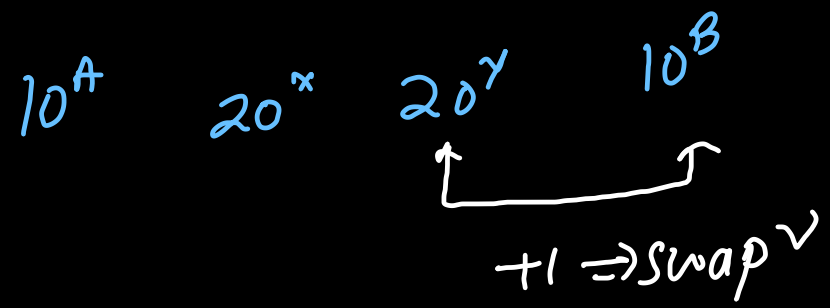


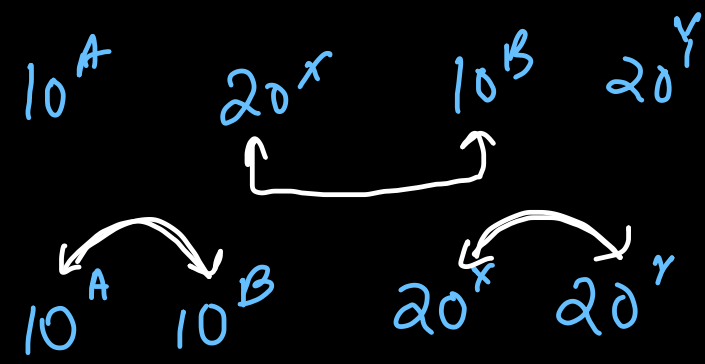
```java
public static void swap(int[] arr, int l, int r){
    int temp = arr[l];
    arr[l] = arr[r];
    arr[r] = temp;
}

public static int compare(int[] arr, int l, int r){
    if(arr[l] < arr[r])  return -1; // swap not required
    else if(arr[l] > arr[r]) return +1; // swap required
    else return 0; // equal -> swap not required
}

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

"bubble sort is Stable"



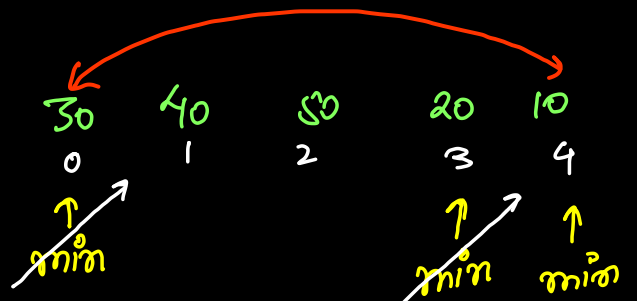



# Selection Sort (increasing order)

**Pass 0** $i = 0$




$j = 1$ : $40 > 30$

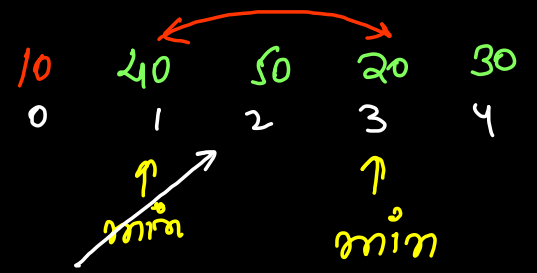
$j = 2$ : $50 > 30$

$j = 3$ : $20 < 30$

$j = 4$ : $10 < 20$

swap(arr, i, min)

**Pass 1** $i = 1$



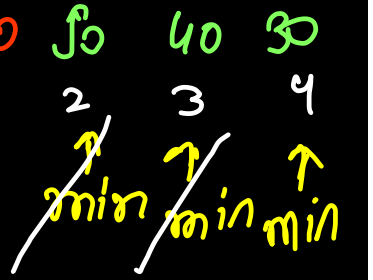

$j = 2$ : $50 > 40$

$j = 3$ : $20 < 40$

$j = 4$ : $30 > 20$

swap(arr, i, min)

**Pass 2** $i = 2$

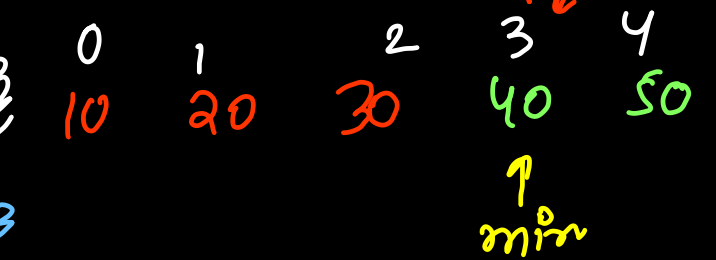



$j = 3$ : $40 < 50$

$j = 4$ : $30 < 40$

swap(arr, i, min)

**Pass 3** $i = 3$




$j = 4$ : $50 > 40$

swap(arr, i, min)

10 20 30 40 50

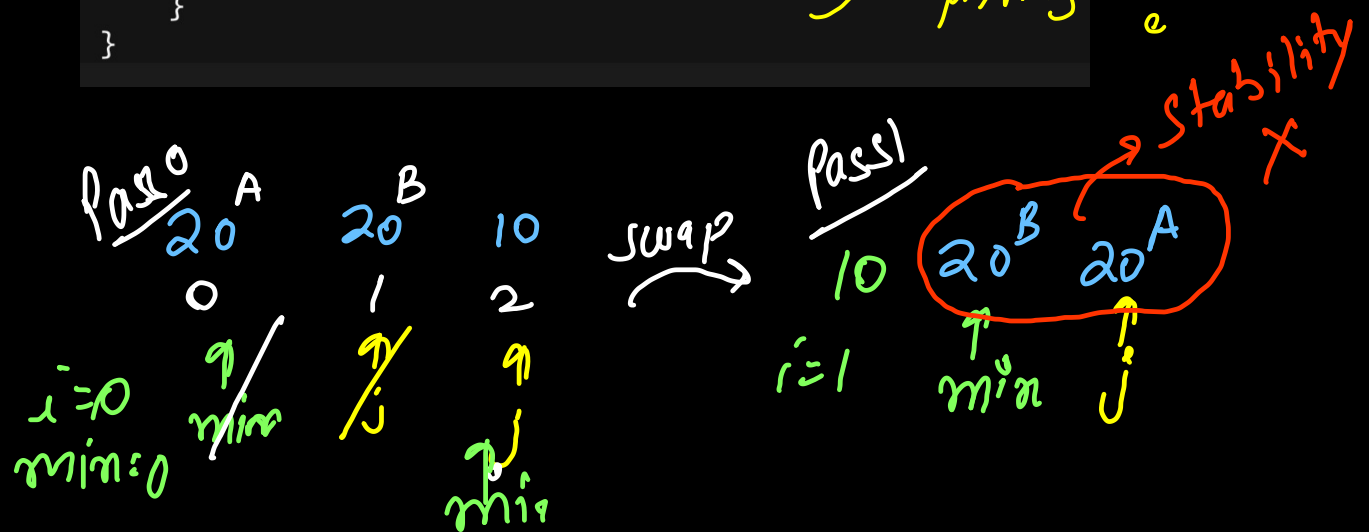
10 20 30 40 50

logic → find min in unsorted array & make it go to the correct position

→ only swap once after every pass
(better in terms of runtime)

```java
void swap(int[] arr, int l, int r){
    int temp = arr[l];
    arr[l] = arr[r];
    arr[r] = temp;
}

int compare(int[] arr, int l, int r){
    if(arr[l] < arr[r])  return -1; // swap not required
    else if(arr[l] > arr[r]) return +1; // swap required
    else return 0; // equal -> swap not required
}

void selectionSort(int arr[], int n)
{
    for(int i = 0; i < n - 1; i++){ // pass
        int min = i;
        for(int j = i + 1; j < n; j++){ // comparison
            if(compare(arr, min, j) > 0){
                min = j;
            }
        }
        if(i != min) swap(arr, i, min);
    }
}
```

not best case optimized!

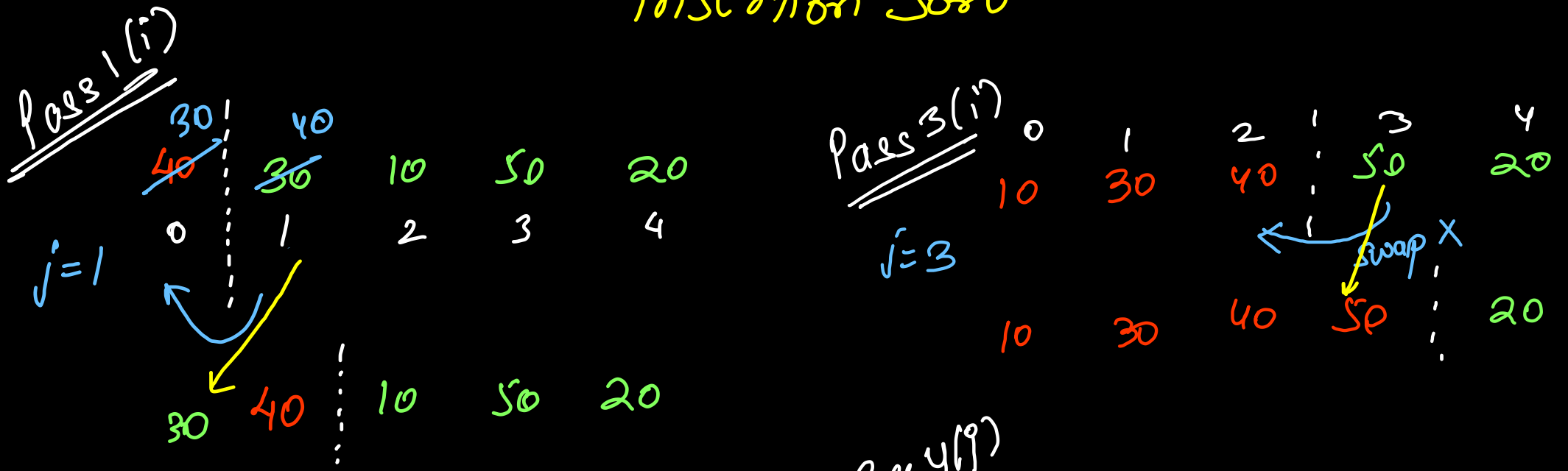



stability → No

inplace → O(1) space, Yes

time
- best case → O(n) if isArraySorted
- Avg case → O(n²) random
- worst case → O(n²) decreasing

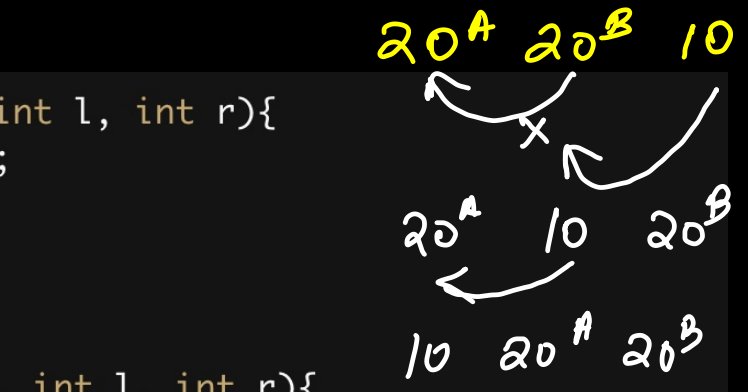
# Insertion Sort



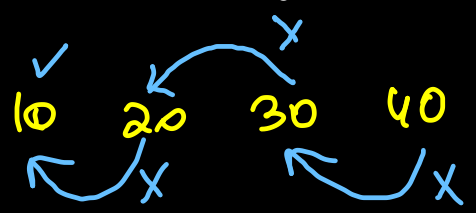
logic → 2 partitions ← Sorted region
Unsorted region

one by one pick element from unsorted region and insert it to sorted position in sorted region

```java
void swap(int[] arr, int l, int r){
    int temp = arr[l];
    arr[l] = arr[r];
    arr[r] = temp;
}

int compare(int[] arr, int l, int r){
    if(arr[l] < arr[r])  return -1; // swap not required
    else if(arr[l] > arr[r]) return +1; // swap required
    else return 0; // equal -> swap not required
}

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




★ best case optimization




# adjacent swaps ↓ locality of reference

Stable Yes

space
- inplace → O(1) space

Yes

time
- best case → O(n) increasing
- avg case → O(n²) random
- worst case → O(n²) decreasing

# Inbuilt Sorting

1) time = O(nlogn)
avg/worst case

# Arrays.sort(array);

2) space = O(1)
inplace sorting

int
Integers ⟶ ]
double
Double ⟶ ] → Increasing order

char
Character ⟶ Ascii of 128 characters ⟶ unique integral code

'a' < 'z' ; 'A' < 'Z' ; 'a' > 'A' '0' < '9'
97 122 65 90 97 65 48 57

(derived)
String ⟶ Lexicographical / Dictionary order

```java
class Solution {
    public int[] sortArray(int[] nums) {
        Arrays.sort(nums);
        return nums;
    }
}
```

LC 912) Sort Array

```java
public static void main(String[] args){
    List<Integer> arr = new ArrayList<>();
    arr.add(40);
    arr.add(20);
    arr.add(10);
    arr.add(50);
    arr.add(30);
    Collections.sort(arr);
    System.out.println(arr);
```

```
Finished in 68 ms
[10, 20, 30, 40, 50]
```

**Java 6's Arrays.sort() – Quicksort**

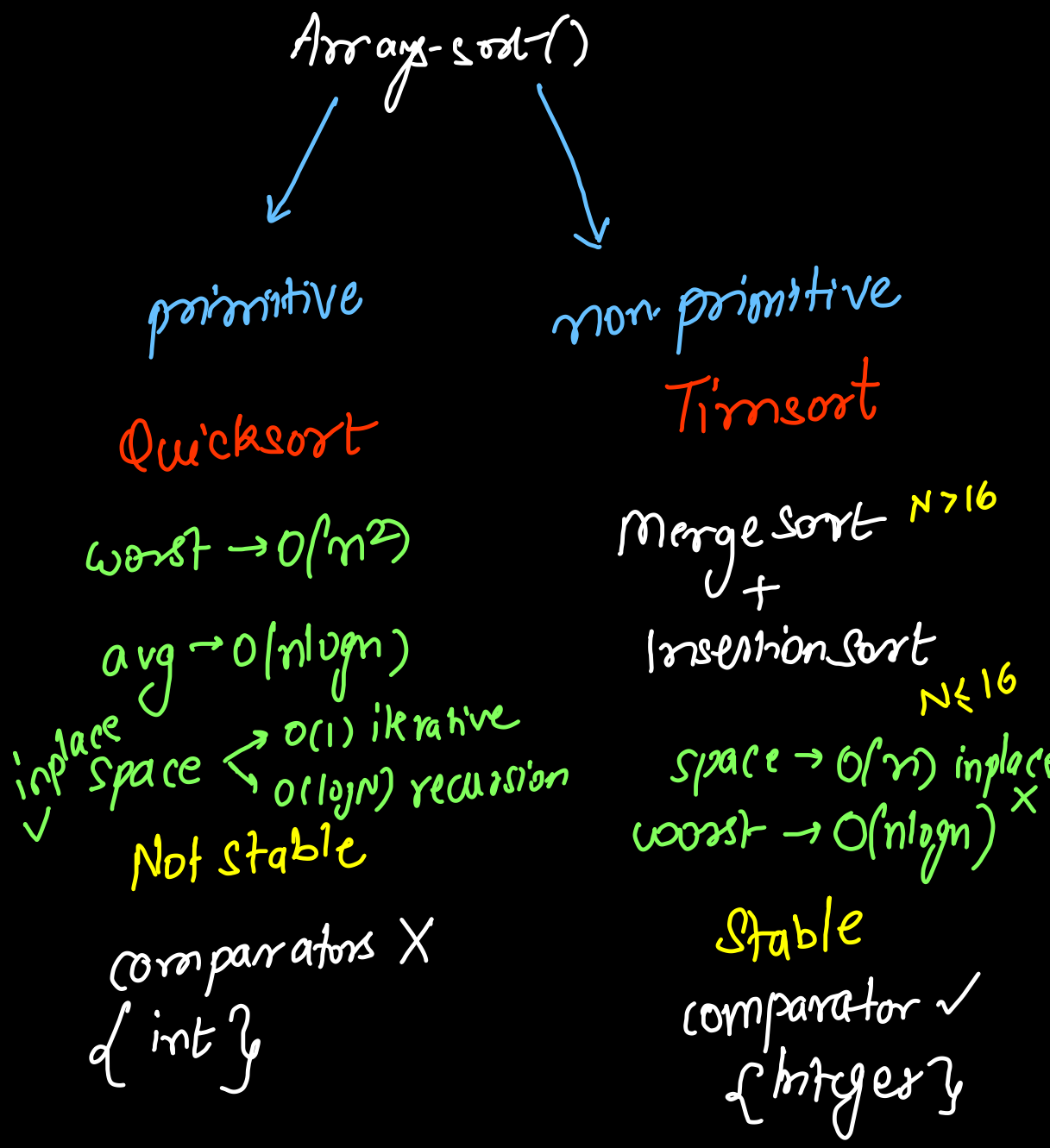
- Best Case Time Complexity- O(NlogN)
- Average Case Time Complexity- O(NlogN)
- Worse Case Time Complexity- $O(N^2)$
- Auxiliary Space- O(logN)
- Stable- Depends
- Adaptive- No

**Java 7's Arrays.sort() – Timsort (Hybrid of Mergesort and Insertion Sort)**

- Best Case Time Complexity- O(N)
- Average Case Time Complexity- O(NlogN)
- Worse Case Time Complexity- O(NlogN)
- Auxiliary Space- O(N)
- Stable- Yes
- Adaptive- Yes

**Java's Collections.sort() – Mergesort**

- Best Case Time Complexity- O(NlogN)
- Average Case Time Complexity- O(NlogN)
- Worse Case Time Complexity- O(NlogN)
- Auxiliary Space- O(N)
- Stable- Yes
- Adaptive- Yes



# Merging + Merge Sort

## Merge 2 Sorted Arrays (codestudio)

$n = 5$ arr1:

| 10 | 30 | 50 | 60 | 70 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

$m = 7$ arr2:

| 20 | 40 | 60 | 80 | 90 | 100 | 110 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

$n + m = 12$ res:

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

# Brute force ( Naïve Soln)

Time => $O((n_1+n_2)\log(n_1+n_2))$

Space => O(1) inplace

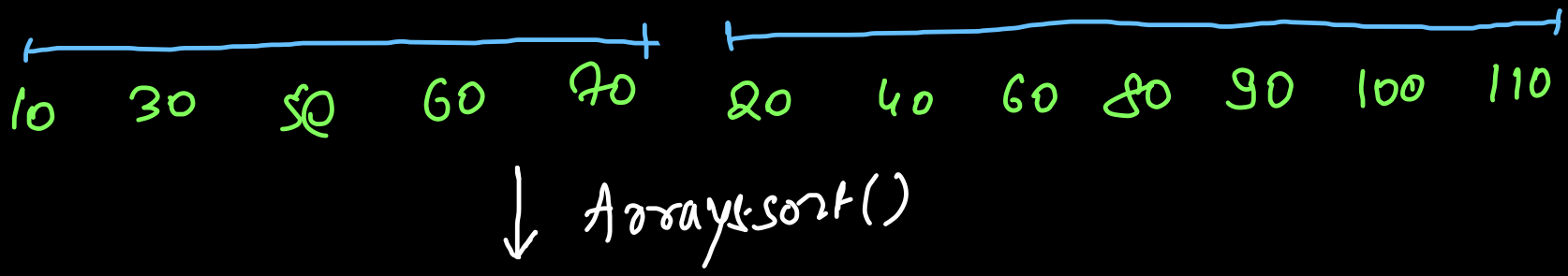
$n_1 = 5$ arr1:

| 10 | 30 | 50 | 60 | 70 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

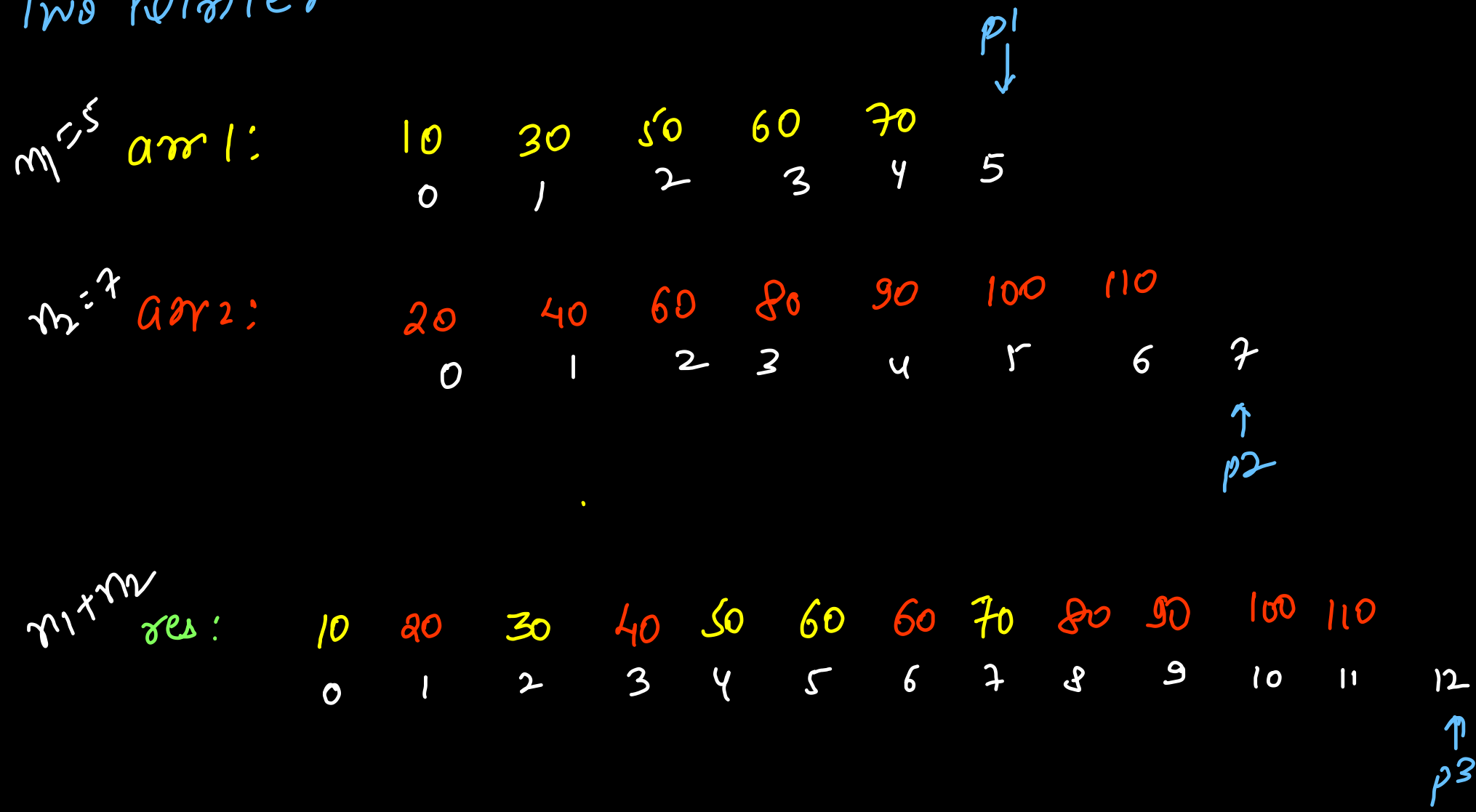
$n_2 = 7$ arr2:

| 20 | 40 | 60 | 80 | 90 | 100 | 110 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

$N = 12$ $(n_1+n_2)$ res:



# Two Pointer


p1
m=5 arr1: 10 30 50 60 70
0 1 2 3 4 5
n=7 arr2: 20 40 60 80 90 100 110
0 1 2 3 4 5 6 7
p2
m+n res: 10 20 30 40 50 60 60 70 80 90 100 110
0 1 2 3 4 5 6 7 8 9 10 11 12
p3

```java
public static int[] ninjaAndSortedArrays(int arr1[], int arr2[], int n1, int n2) {
    int[] arr3 = new int[n1 + n2];
    int p1 = 0, p2 = 0, p3 = 0;

    while(p1 < n1 && p2 < n2){
        if(arr1[p1] <= arr2[p2]){
            arr3[p3] = arr1[p1];
            p3++; p1++;
        } else {
            arr3[p3] = arr2[p2];
            p3++; p2++;
        }
    }

    while(p1 < n1){
        arr3[p3] = arr1[p1];
        p3++; p1++;
    }

    while(p2 < n2){
        arr3[p3] = arr2[p2];
        p3++; p2++;
    }

    return arr3;
}
```

compare & insert

only one of these two loops will run

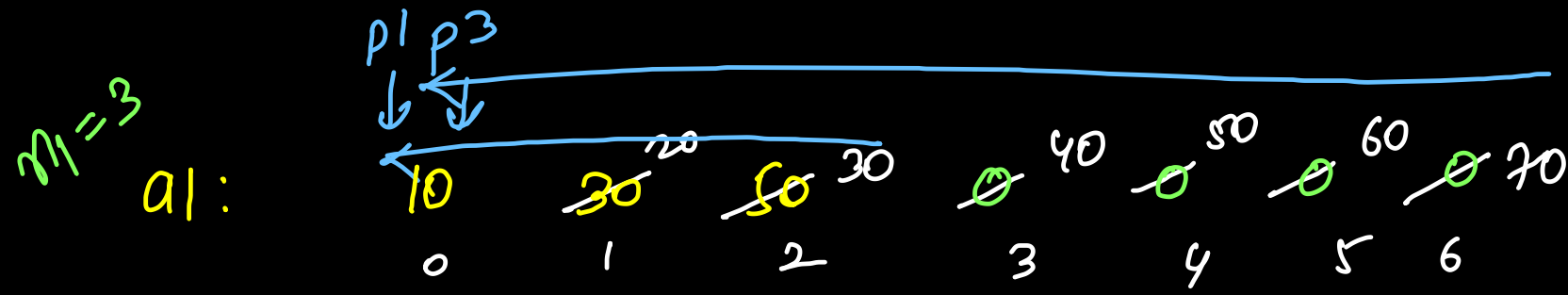
time = $O(n_1+n_2)$

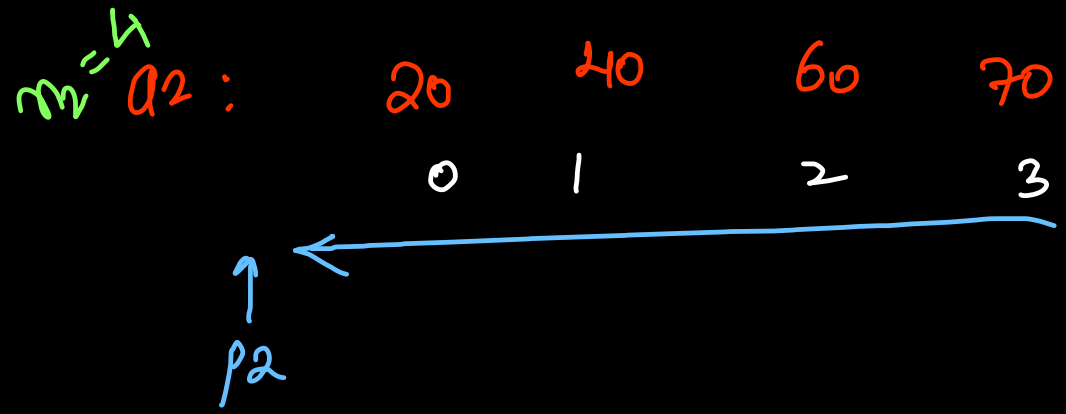
Space = $O(1)$ inplace ✓ (extra)

input X (a1, a2)

output X (a3)

# # Variation → Leetcode (88)




```java
public void merge(int[] arr1, int n1, int[] arr2, int n2) {
    int p1 = n1 - 1, p2 = n2 - 1, p3 = n1 + n2 - 1;

    while(p1 >= 0 && p2 >= 0){
        if(arr1[p1] > arr2[p2]){
            arr1[p3] = arr1[p1];
            p3--; p1--;
        } else {
            arr1[p3] = arr2[p2];
            p3--; p2--;
        }
    }

    while(p2 >= 0){
        arr1[p3] = arr2[p2];
        p3--; p2--;
    }
}
```

time = $O(m_1 + m_2)$

Space = $O(1)$
inplace

# variation (GFG)

$n_1 = 4$

a1 :

| (10) | (20) | (30) | (40) |
|---|---|---|---|
| 10 | 30 | 70 | 80 |
| 0 | 1 | 2 | 3 |

$n_2 = 6$

a2 :

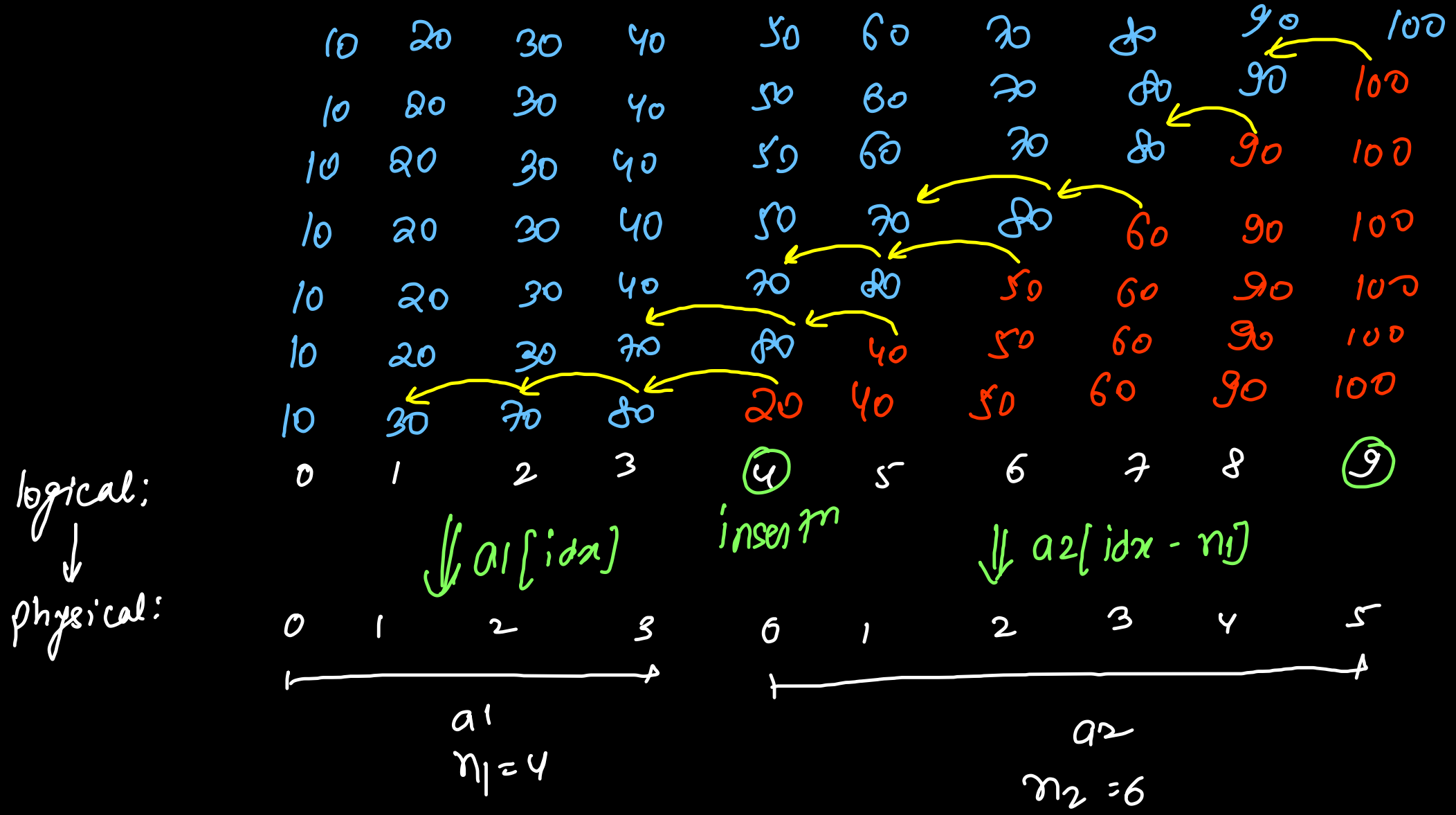
| 20 | 40 | 50 | 60 | 90 | 100 |
|---|---|---|---|---|---|
| ~~4~~ 0 | ~~5~~ 1 | ~~6~~ 2 | ~~7~~ 3 | ~~8~~ 4 | ~~9~~ 5 |
| (50) | (60) | (70) | (80) | (90) | (100) |

Hint


time $\Rightarrow O(N^2)$

where $N = N_1 + N_2$

space $\Rightarrow O(1)$
inplace


10 20 30 40 50 60 70 80 90 100
10 20 30 40 50 60 70 80 90 100
10 20 30 40 50 60 70 80 90 100
10 20 30 40 50 70 80 60 90 100
10 20 30 40 70 80 50 60 90 100
10 20 30 70 80 40 50 60 90 100
10 30 70 80 20 40 50 60 90 100
0 1 2 3 4 5 6 7 8 9
logical:
physical:
a1[idx]
insert
a2[idx - n1]
0 1 2 3
0 1 2 3 4 5
a1
n1 = 4
a2
n2 = 6

```java
void swap(long[] arr1, long[] arr2, int l, int r){
    int n1 = arr1.length, n2 = arr2.length;
    if(l < n1 && r < n1){
        long temp = arr1[l];
        arr1[l] = arr1[r];
        arr1[r] = temp;
    }
    else if(l < n1 && r >= n1){
        long temp = arr1[l];
        arr1[l] = arr2[r - n1];
        arr2[r - n1] = temp;
    }
    else {
        long temp = arr2[l - n1];
        arr2[l - n1] = arr2[r - n1];
        arr2[r - n1] = temp;
    }
}
```

$l \to a_1, r \to a_1$

$l \to a_1, r \to a_2$

$l \to a_2, r \to a_2$

```java
static int compare(long[] arr1, long[] arr2, int l, int r){
    int n1 = arr1.length;
    long lval = (l < n1) ? arr1[l] : arr2[l - n1];
    long rval = (r < n1) ? arr1[r] : arr2[r - n1];

    if(lval < rval)  return -1; // swap not required
    else if(lval > rval) return +1; // swap required
    else return 0; // equal -> swap not required
}

public static void merge(long arr1[], long arr2[], int n1, int n2)
{
    // Insertion sort
    for(int i = n1; i < n1 + n2; i++){ // pass
        for(int j = i; j > 0; j--){ // comp
            if(compare(arr1, arr2, j - 1, j) > 0){
                swap(arr1, arr2, j - 1, j);
            }
            else break;
        }
    }
}
```

TLE ⇒ Insertion Sort ⇒ $O(N^2)$ time

$O(1)$ space (in place)

⇓

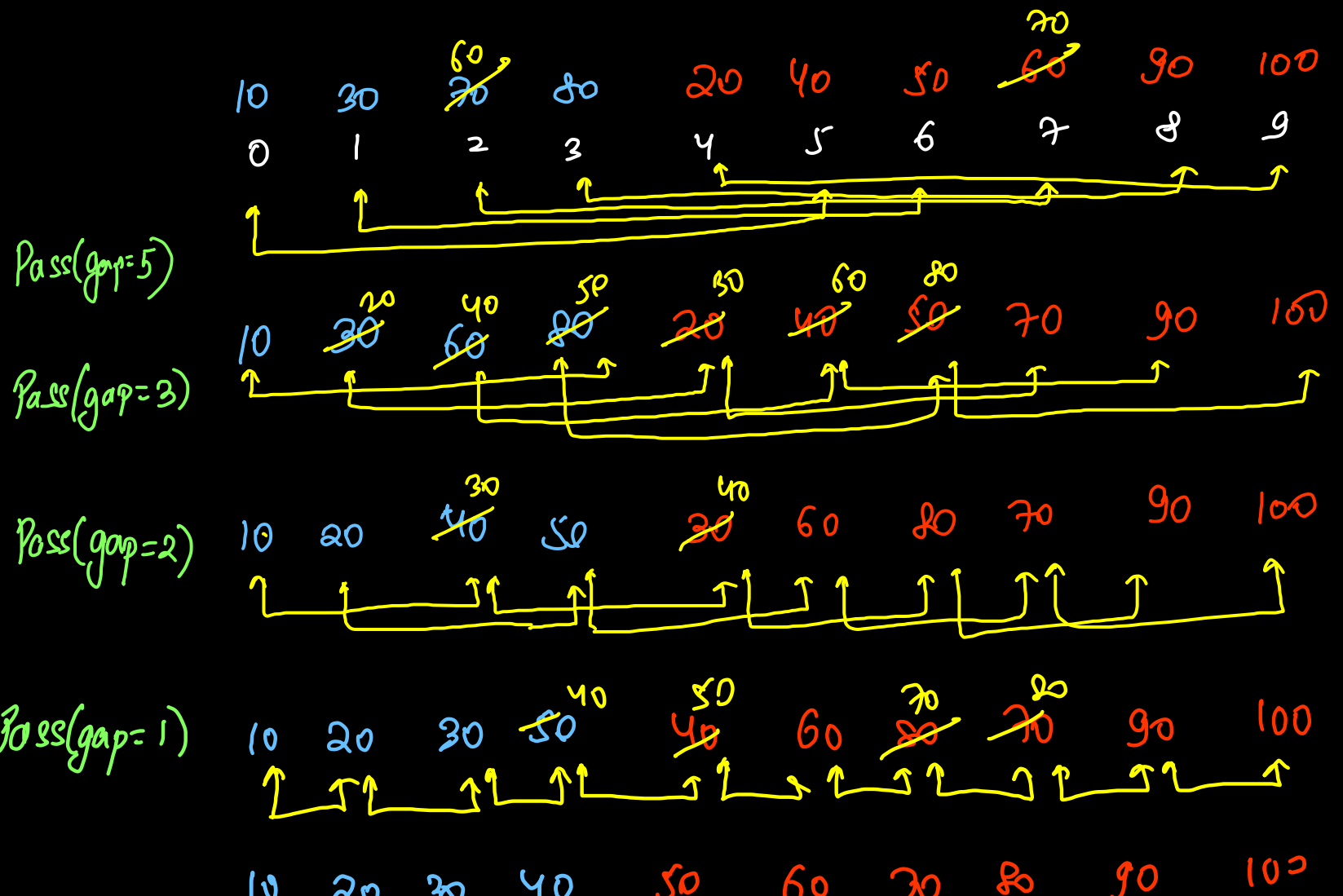
☆ shell sort optimization

# #Shell sort

Time = O(nlogn)

Space = O(1)

$n_1 + n_2 = 10$




no of passes = logn

each pass = n

Time: nlogn

Space: O(1)

```java
public static void merge(long arr1[], long arr2[], int n1, int n2)
{
    int i = n1 + n2; // gap or pass
    while(i >= 1){
        for(int j = 0; j + i < n1 + n2; j++){ // comparison
            if(compare(arr1, arr2, j, j + i) > 0){
                swap(arr1, arr2, j, j + i);
            }
        }
        if(i == 1) break;
        i = i / 2 + i % 2;
    }
}
```

(to avoid infinite loop)

ceil division

or $i = (i+1)/2$

$O(n \log n)$ time

best/avg/worst

inplace ✓ stable ✗

Swap & compare are same as previous code!

logic → instead of comparing adjacent,

compare elements at gap $n/2, n/4, n/8, \ldots 1$

# Lc 977) Squares of Sorted Array

non-decreasing → increasing or either same

| 49 | 25 | 9 | 4 | 0 | 1 | 9 | 16 | 36 | 64 | 81 |
|---|---|---|---|---|---|---|---|---|---|---|
| -7, | -5, | -3, | -2, | 0, | +1, | +3, | +4, | +6, | +8, | +9 |

left → ← right

[ 0, 1, 4, 9, 9, 16, 25, 36, 49, 64, 81 ]

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

    for(int idx = nums.length - 1; idx >= 0; idx--){
        if(nums[left] * nums[left] > nums[right] * nums[right]){
            res[idx] = nums[left] * nums[left];
            left++;
        } else {
            res[idx] = nums[right] * nums[right];
            right--;
        }
    }

    return res;
}
```

→ output

Time ⇒ O(n)

Space ⇒ O(1)

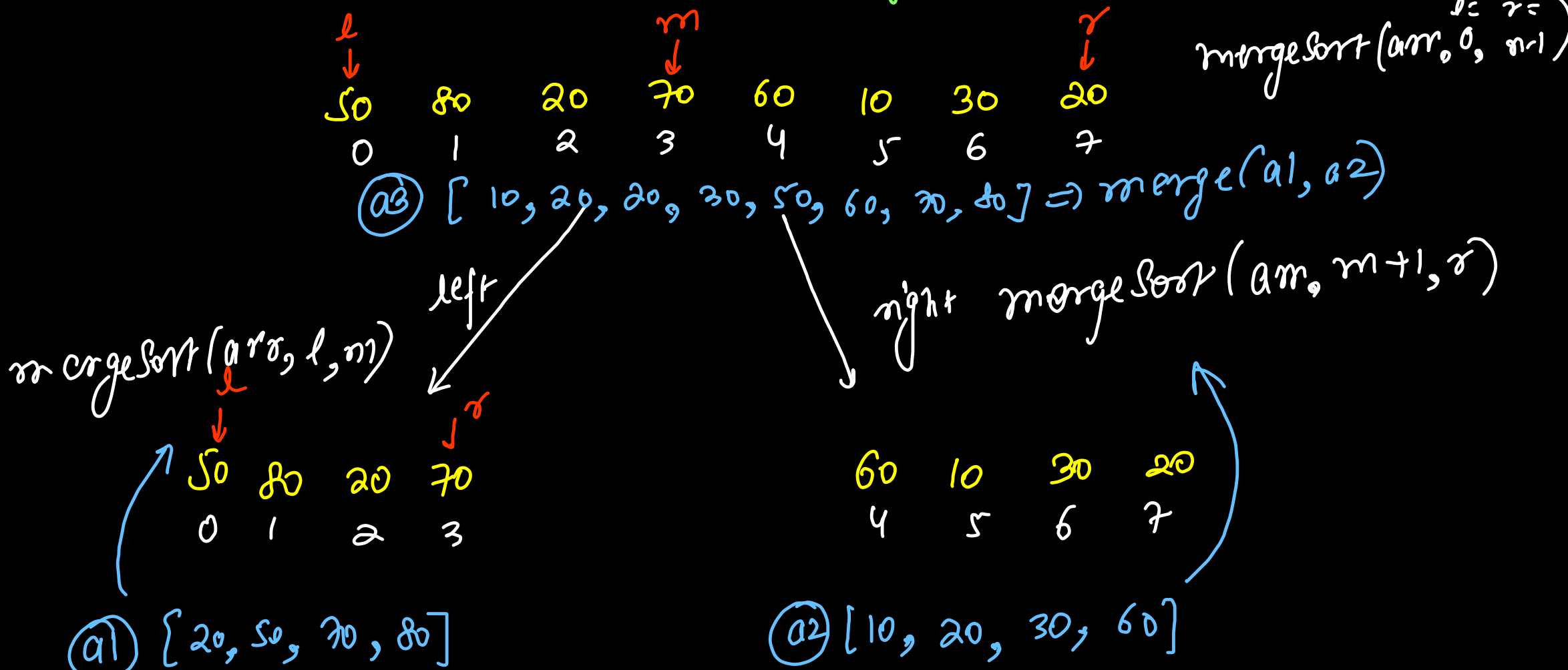
LC 977

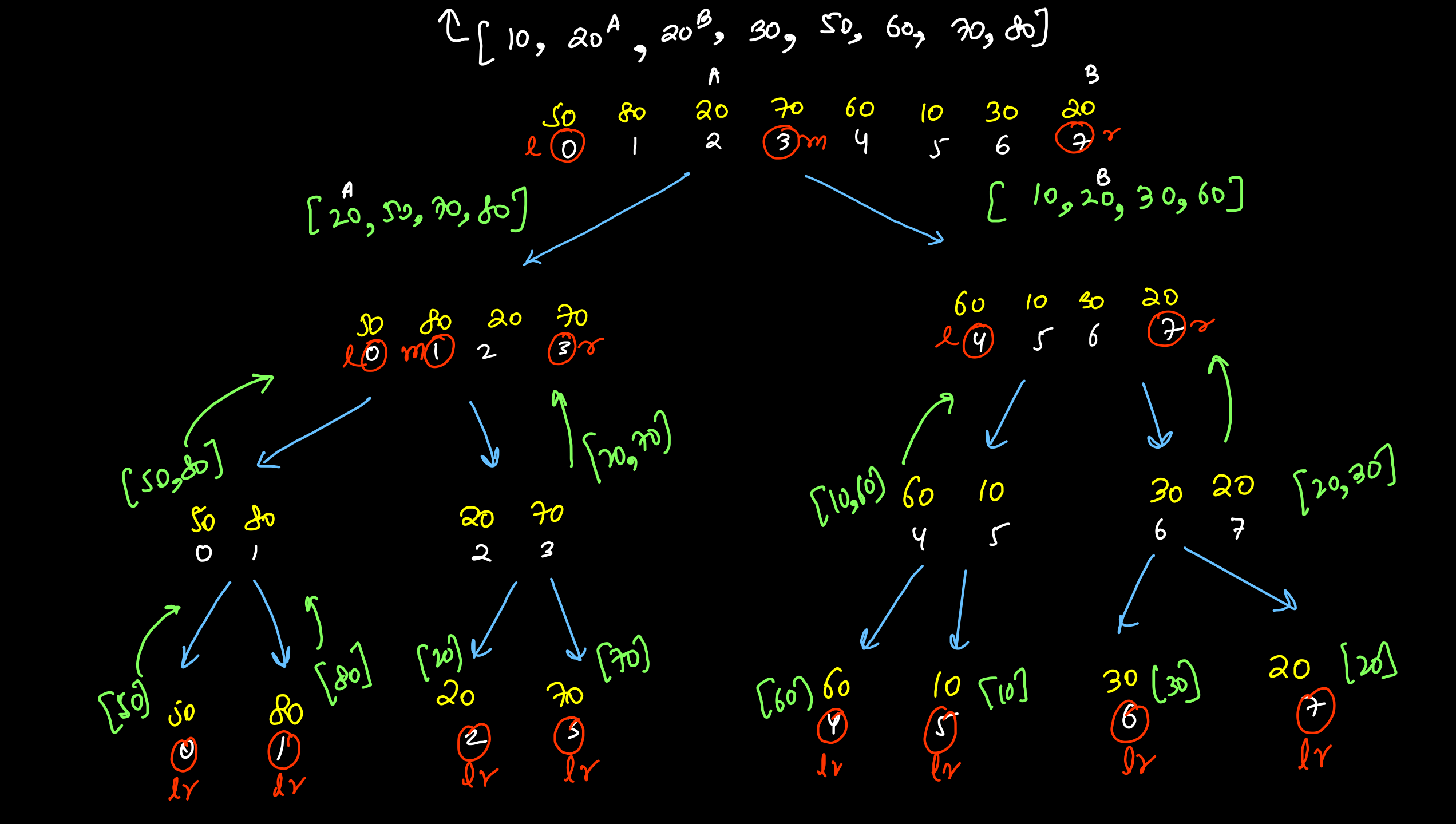
logic → More negative ⇒ Square More Bigger!

More positive ⇒

# Merge Sort (With Return type)

→ Divide and Conquer Algorithm




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

```java
int[] merge(int arr1[], int arr2[]) {
    int n1 = arr1.length, n2 = arr2.length;
    int[] arr3 = new int[n1 + n2];
    int p1 = 0, p2 = 0, p3 = 0;

    while(p1 < n1 && p2 < n2){
        if(arr1[p1] <= arr2[p2]){
            arr3[p3] = arr1[p1];
            p3++; p1++;
        } else {
            arr3[p3] = arr2[p2];
            p3++; p2++;
        }
    }

    while(p1 < n1){
        arr3[p3] = arr1[p1];
        p3++; p1++;
    }

    while(p2 < n2){
        arr3[p3] = arr2[p2];
        p3++; p2++;
    }

    return arr3;
}
```

work = n (merge)
calls = 2
depth = log n

recursion

```java
public int[] mergeSort(int[] nums, int left, int right){
    if(left == right){
        // one element -> already sorted
        return new int[]{ nums[left] };
    }

    int mid = left + (right - left) / 2;
    int[] larr = mergeSort(nums, left, mid);
    int[] rarr = mergeSort(nums, mid + 1, right);
    return merge(larr, rarr);
}

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

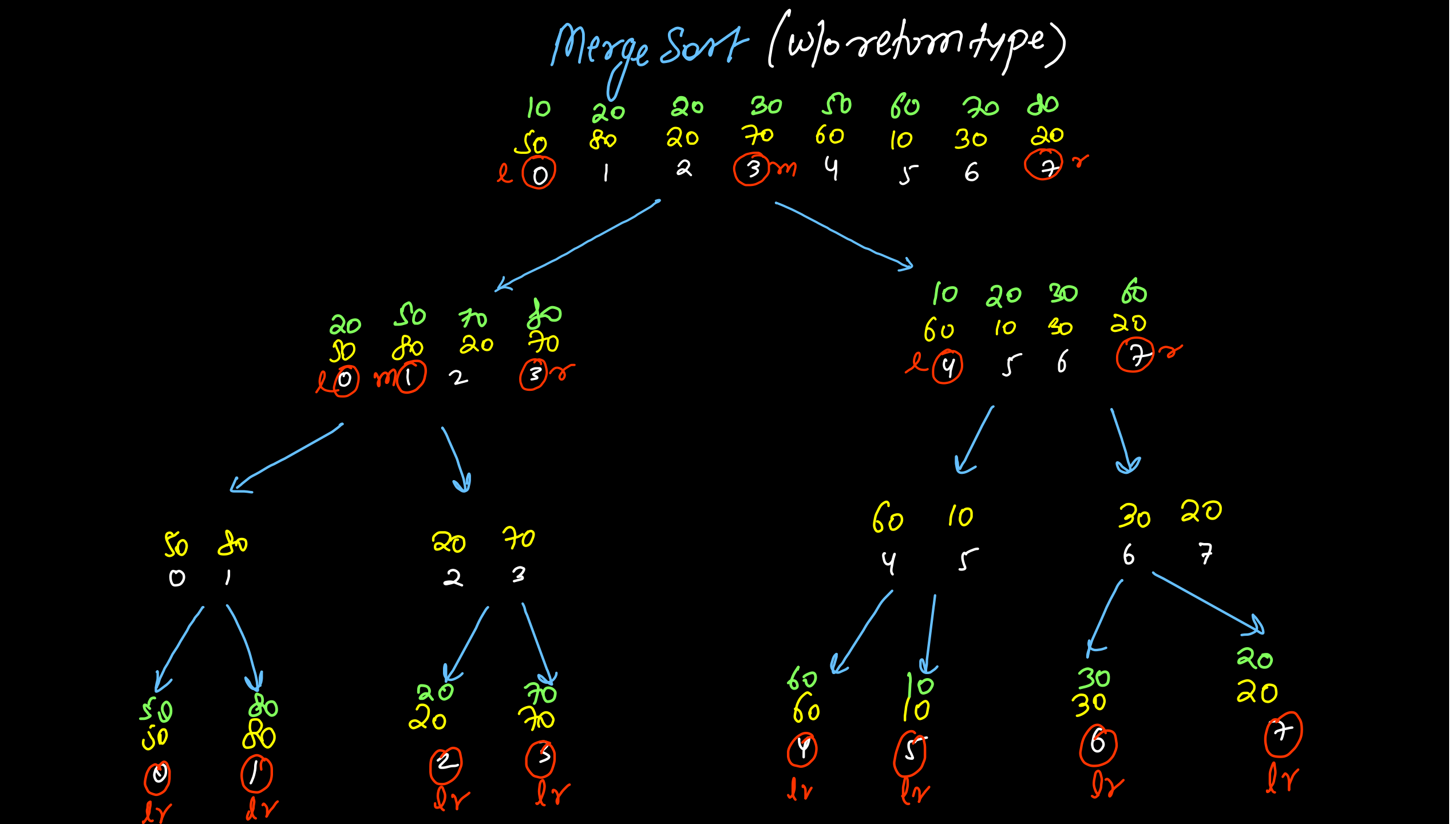
larr, rarr → extra space

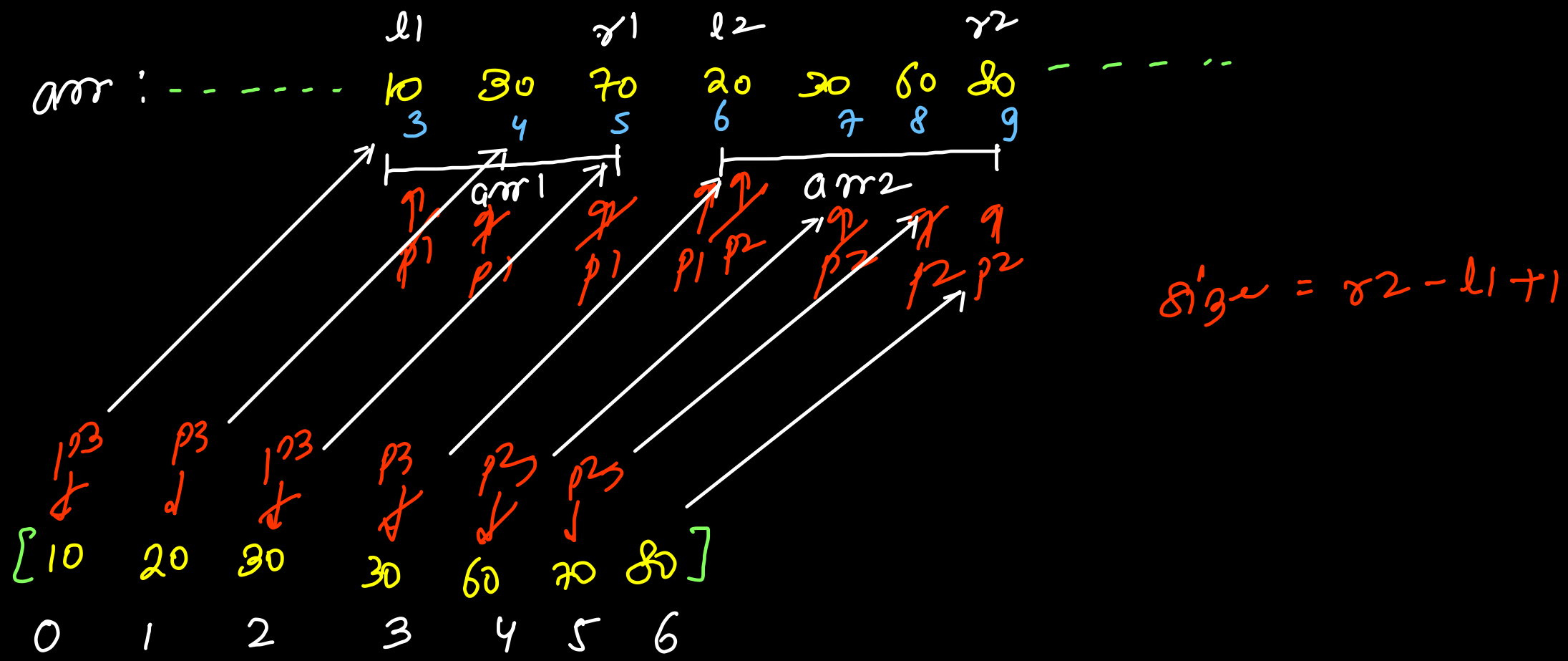
work

$$\text{Time} = O(2^{\log_2 n} + n \log n)$$
$$= O(n + n \log n) = O(n \log n)$$

Space =
- O(n) extra space due to merge
- O(log n) recursion call stack

# Merge Sort (w/o return type)




l1
r1
l2
r2
arr : - - - - - - -
10 30 70 20 30 60 80 - - - - -
3 4 5 6 7 8 9
arr1
arr2
p1 p1 p1 p1 p2 p2 p2
p3 p3 p3 p3 p2 p2 p2
[10 20 30 30 60 70 80]
0 1 2 3 4 5 6
Size = r2 - l1 + 1

```java
void merge(int[] nums, int l1, int r1, int l2, int r2) {
    int[] res = new int[r2 - l1 + 1];
    int p1 = l1, p2 = l2, p3 = 0;

    while(p1 <= r1 && p2 <= r2){
        if(nums[p1] <= nums[p2]){
            res[p3] = nums[p1];
            p3++; p1++;
        } else {
            res[p3] = nums[p2];
            p3++; p2++;
        }
    }

    while(p1 <= r1){
        res[p3] = nums[p1];
        p3++; p1++;
    }

    while(p2 <= r2){
        res[p3] = nums[p2];
        p3++; p2++;
    }

    for(int idx = l1; idx <= r2; idx++)
        nums[idx] = res[idx - l1];
}
```

extra

copy

```java
public void mergeSort(int[] nums, int left, int right){
    if(left >= right) return;
    int mid = left + (right - left) / 2;
    mergeSort(nums, left, mid);
    mergeSort(nums, mid + 1, right);
    merge(nums, left, mid, mid + 1, right);
}

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

Time → O(nlogn)

Space → O(n) extra space

Space → O(logn) recursion

# Partitioning Algorithms

Partition Array {≤ pivot → left, > pivot → right}

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

pivot = 50

Brute force :→ Output Array

time ⇒ O(n)

space ⇒ O(n)

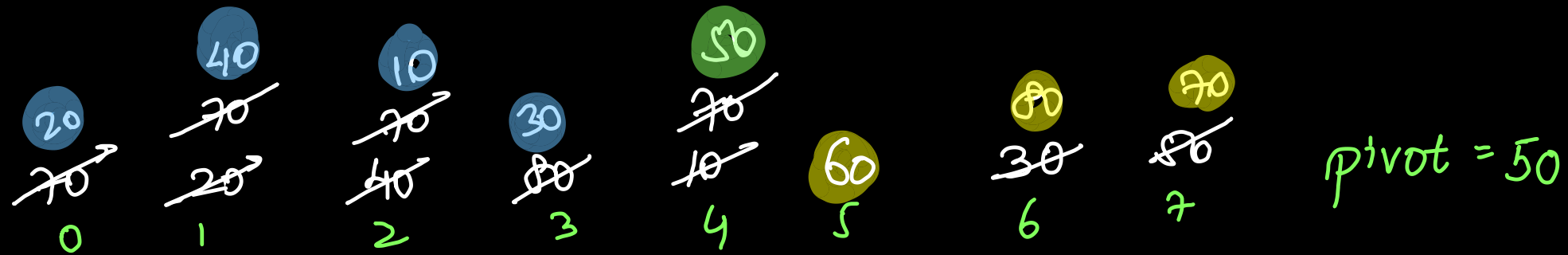
~~in place~~

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| res : | 20 | 40 | 10 | 30 | 50 | 60 | 80 | 70 |
| (output) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

l (≤ pivot)

r (> pivot)

# optimized Approach → Lomuto Partitioning




time ⇒ O(n) linear

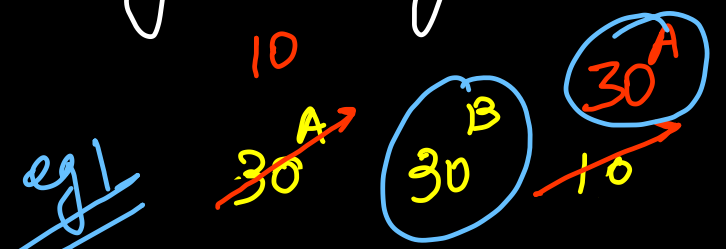
space ⇒ O(1) constant

Inplace

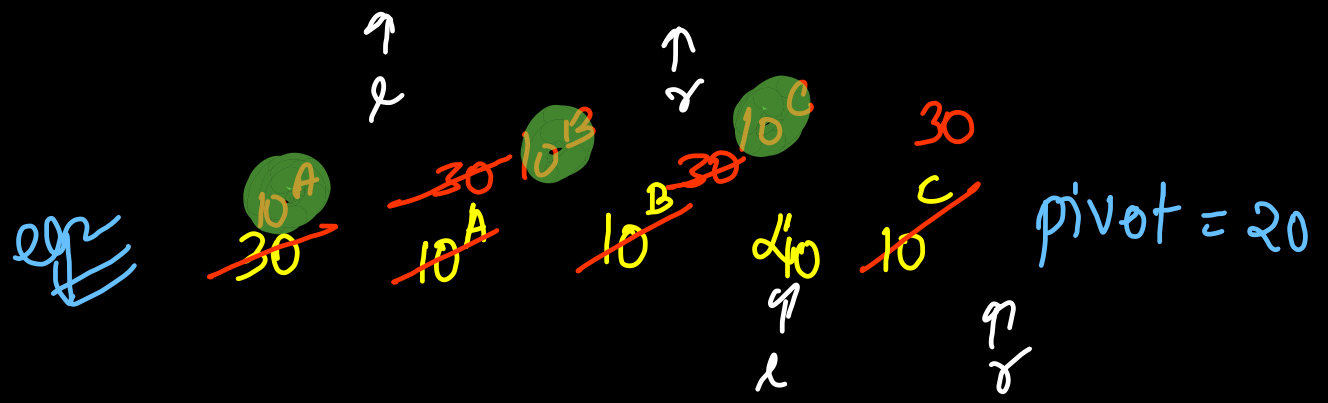
StableX

```java
public static void partition(int[] arr, int pivot){
    int left = 0, right = 0;

    while(right < arr.length){
        if(arr[right] <= pivot){
            // left (<= pivot)
            swap(arr, left, right);
            left++; right++;
        } else {
            // right (> pivot)
            right++;
        }
    }
}
```

# # why stability is not ensured ?




pivot = 20




input

$30^A < 30^B$

output

$30^B < 30^A$

```
public static void partition(int[] arr, int pivot){
    int left = 0, right = 0;

    while(right < arr.length){
        if(arr[right] <= pivot){
            // left (<= pivot)
            swap(arr, left, right);
            left++; right++;
        } else {
            // right (> pivot)
            right++;
        }
    }
}
```

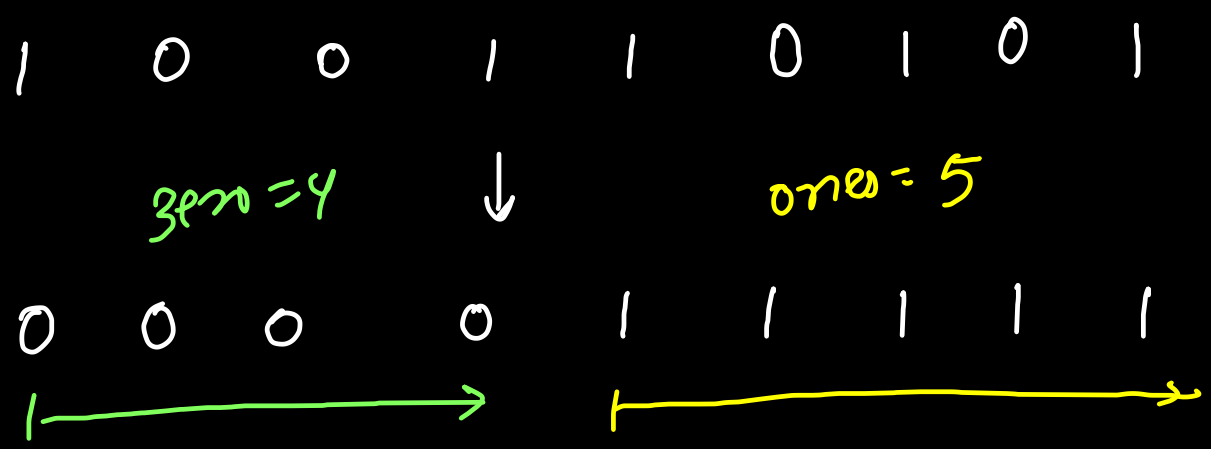
Stable X

↳ greater region

↳ stable ✓ smaller region

# #Variations

## 1) Sort 01 / Sort Binary Array




- Alternate approach :->

Pass1 count 0s, count 1s

Pass2 store 0s & 1s

Time => O(2n)

Space => O(1)

- Optimized Soln

Lomuto Partitioning

Time => O(n) Single pass

space => O(1)

```java
static void swap(int[] arr, int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
static void binSort(int arr[], int N)
{
    int left = 0, right = 0;

    while(right < arr.length){
        if(arr[right] == 0){
            // left (<= pivot)
            swap(arr, left, right);
            left++; right++;
        } else {
            // right (> pivot)
            right++;
        }
    }
}
```

2) Segregate -ve & +ves (w/o relative order)

3) Segregate odd & even (w/o relative order)

4) Segregate male/female, X/Y, etc

# Sort 012 (LC 75) ⭐

## Dutch National Flag Algorithm (DNF Sort)

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

↓

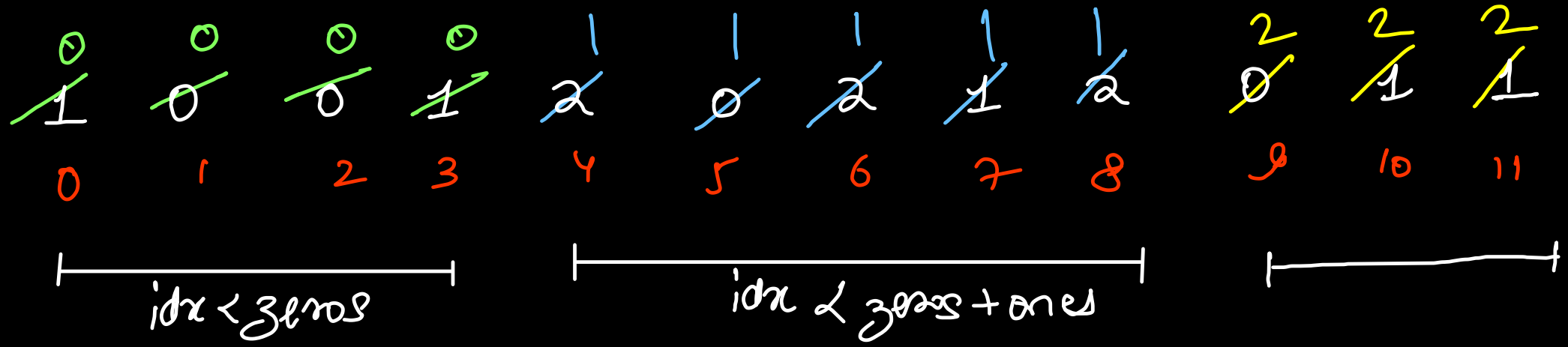
0 0 0 0 1 1 1 1 1 2 2 2

① Sorting (MergeSort)
time: $n \log n$
space: $n$

② extra array
time: $O(n)$
space: $O(n)$

③ counting (2 pass)
time: $O(2n)$
space: $O(1)$

④




zeros = 4

ones = 5

twos = 3

Two loops Algo

```java
public void sortColors(int[] nums) {
    int zeros = 0, ones = 0, twos = 0;

    // First Pass
    for(int val : nums){
        if(val == 0) zeros++;
        else if(val == 1) ones++;
        else twos++;
    }

    // Second Pass
    for(int idx = 0; idx < nums.length; idx++){
        if(idx < zeros) nums[idx] = 0;
        else if(idx < zeros + ones) nums[idx] = 1;
        else nums[idx] = 2;
    }
}
```

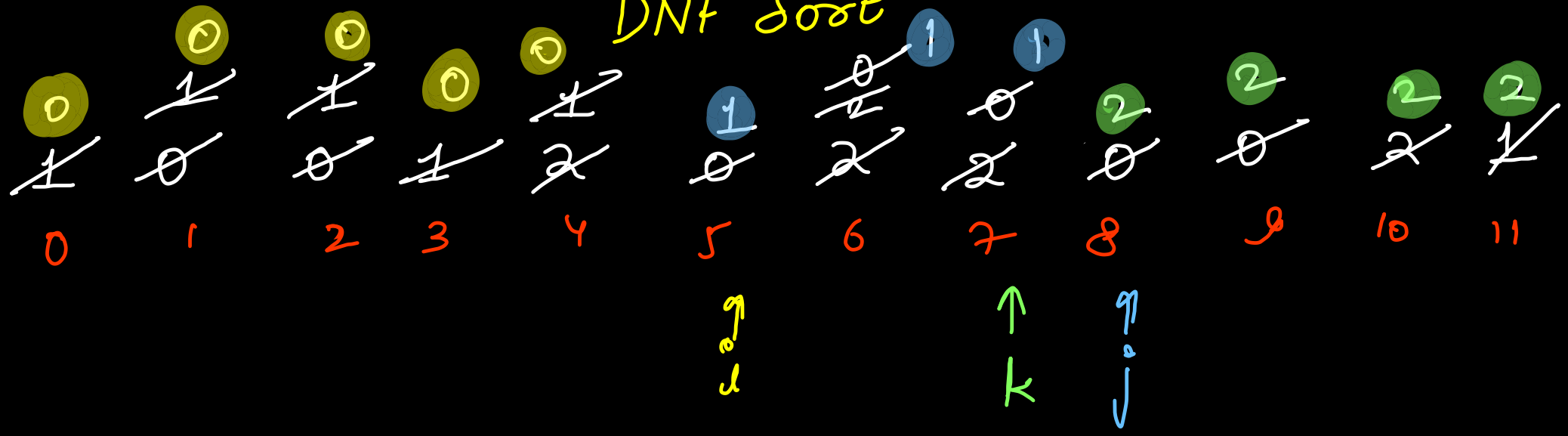
time = O(2n)

Space : O(1)

Inplace ✓

Stable ✓

# DNF Sort




$[0, i) \Rightarrow$ (0s)

$[i, j) \Rightarrow$ (1s)

$[j, k] \Rightarrow$ unexplored

$(k, n) \Rightarrow$ (2s)

```
while (j <= k) {
    arr[j] == 0      swap(arr, i, j)   i++ j++
    arr[j] == 1      j++
    arr[j] == 2      swap(arr, j, k)   k--
}
```

```java
public void swap(int[] nums, int i, int j){
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
public void sortColors(int[] nums) {
    int i = 0, j = 0, k = nums.length - 1;

    while(j <= k){
        if(nums[j] == 0){
            // left (0s)
            swap(nums, i, j);
            i++; j++;
        } else if(nums[j] == 1){
            // middle (1s)
            j++;
        } else {
            // right (2s)
            swap(nums, j, k);
            k--;
        }
    }
}
```

single pass

Time = O(n)

Space = O(1)

Stability = No

Inplace = Yes

# Variations

① pivot ⇒ partitioning (three ways)

| < pivot | = pivot | > pivot |
|---|---|---|
| left | mid | right |

70 20 50 30 80 50 10 90 50

pivot = 50

↓ output

20 30 10 50 50 50 80 70 90

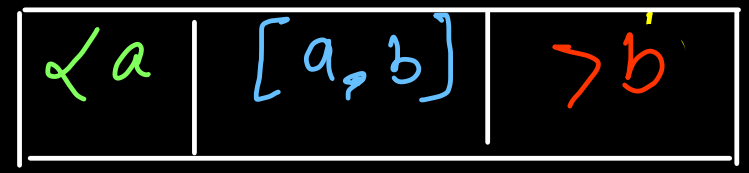
stability ✓

stability ✗

eg #2161 → LC → com extra array

# ② Dual pivot partitioning

pivot = a, b




eg 10 30 80 20 5 3 40 90 50 a = 20 b = 50

↓

10 5 3 20 40 50 40 90 80

three way partitioning → GfG

```java
public void threeWayPartition(int nums[], int a, int b)
{
    int i = 0, j = 0, k = nums.length - 1;

    while(j <= k){
        if(nums[j] < a){
            // left ( < a )
            swap(nums, i, j);
            i++; j++;
        } else if(nums[j] <= b){
            // middle [a, b]
            j++;
        } else {
            // right ( > b )
            swap(nums, j, k);
            k--;
        }
    }
}
```

③ Segregate −ves, 0s, +ves (Parity)

④ Segregate male/female/others, X/Y/Z

Colors → Red/Green/Blue

# Quick Sort ⇒ Divide & Conquer Algorithm

preorder work → Partitioning Algo { pivot → Last element in range }

→ after partitioning pivot value will get placed at sorted position.

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

work: partition (pivot = 50)

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

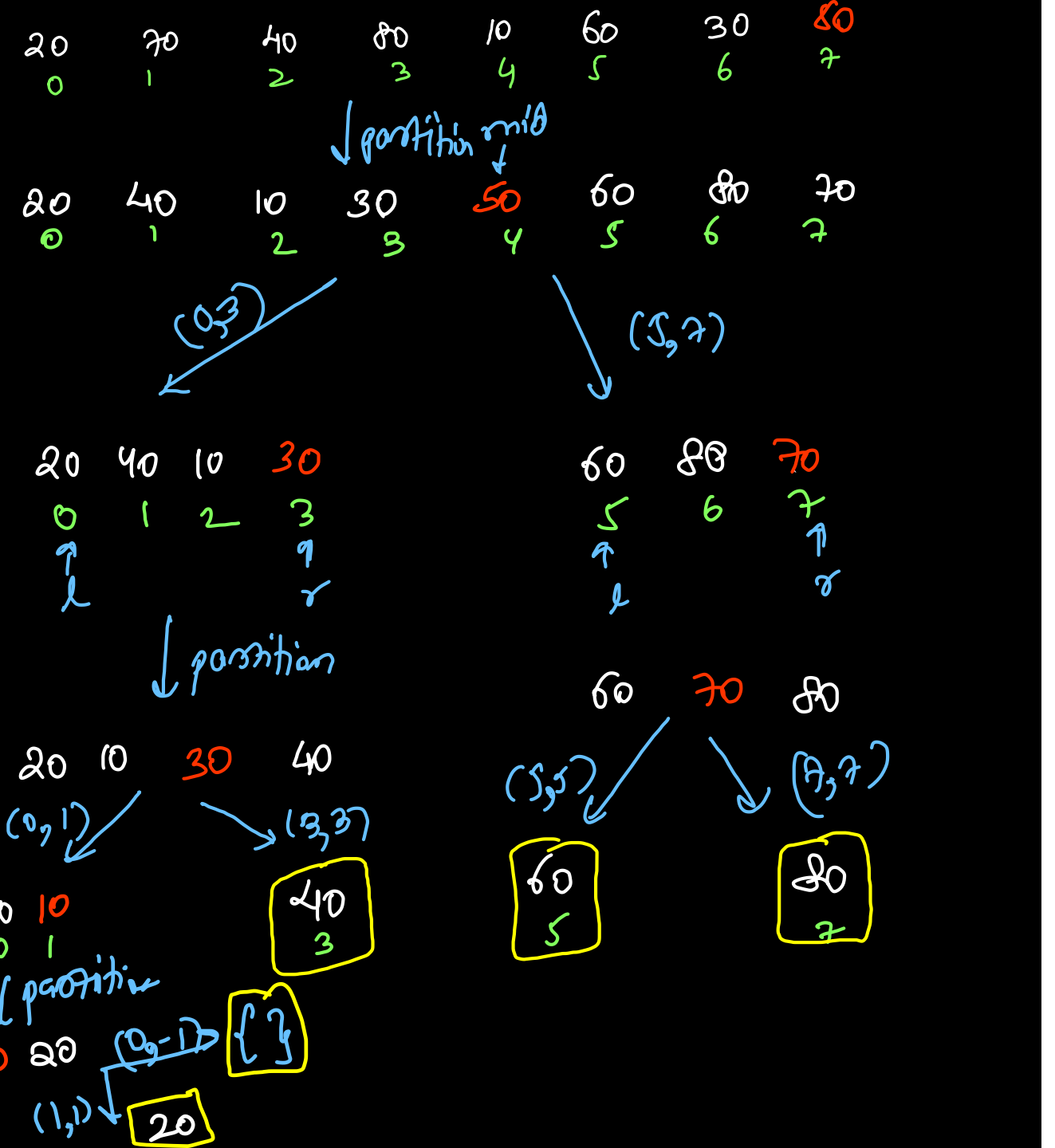
left → 20 40 10 30 (left, mid-1)

mid

right → 60 80 70 (mid+1, right)

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

    while(j <= right){
        if(nums[j] <= pivot){
            swap(nums, i, j);
            i++; j++;
        } else {
            j++;
        }
    }

    return (i - 1);
}
```

```java
public void quickSort(int[] nums, int left, int right){
    if(left >= right) return; // base case (0 or 1 size)
    int mid = partition(nums, left, right); // preorder work
    quickSort(nums, left, mid - 1); // left recursive call (faith)
    quickSort(nums, mid + 1, right); // right recursive call (faith)
}

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



Why pivot is last element

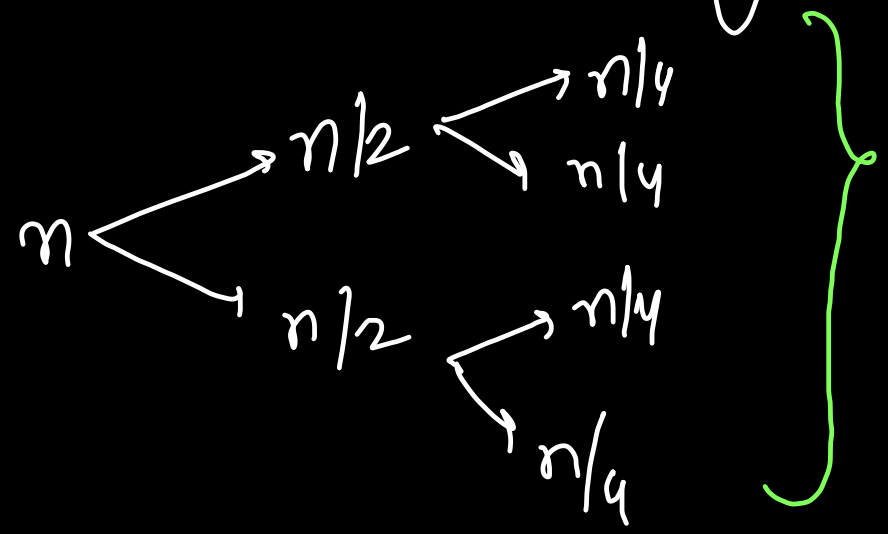
TLE? → time?

space?

optimization → Dual pivot
→ randomized
→ three way

# Average Time Complexity

Assumption ⇒ Partition is mostly around mid index




$T(n) = 2T(n/2) + O(n)$

depth = $O(\log_2 N)$

calls = 2

work = $O(n)$ partitioning

$calls^{depth} + work \times depth = 2^{\log_2 N} + n \times \log_2 n$

$= N + N\log N \Rightarrow \underline{O(N\log N)}$

# Space Complexity

Partitioning → O(1) space {Inplace ✓}

Quick sort → recursive call stack space

⇒ depth / height

⇒ $O(\log_2 N)$ avg case

# TLE ? ⇒ Worst case time complexity

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

↓ partition

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

right → X

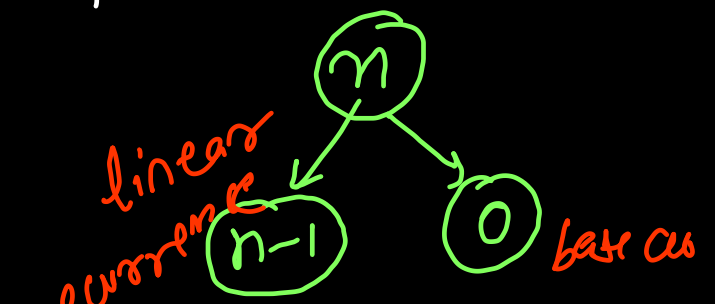
↓ left (N-1) ele

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

~~right~~ X

↓ left (N-2) ele

pivot → min^m / max^m




$T(n) = T(n-1) + O(n)$

calls = 1

depth : N

work = N

time : $1^N + N*N$

= $O(N^2)$ worst

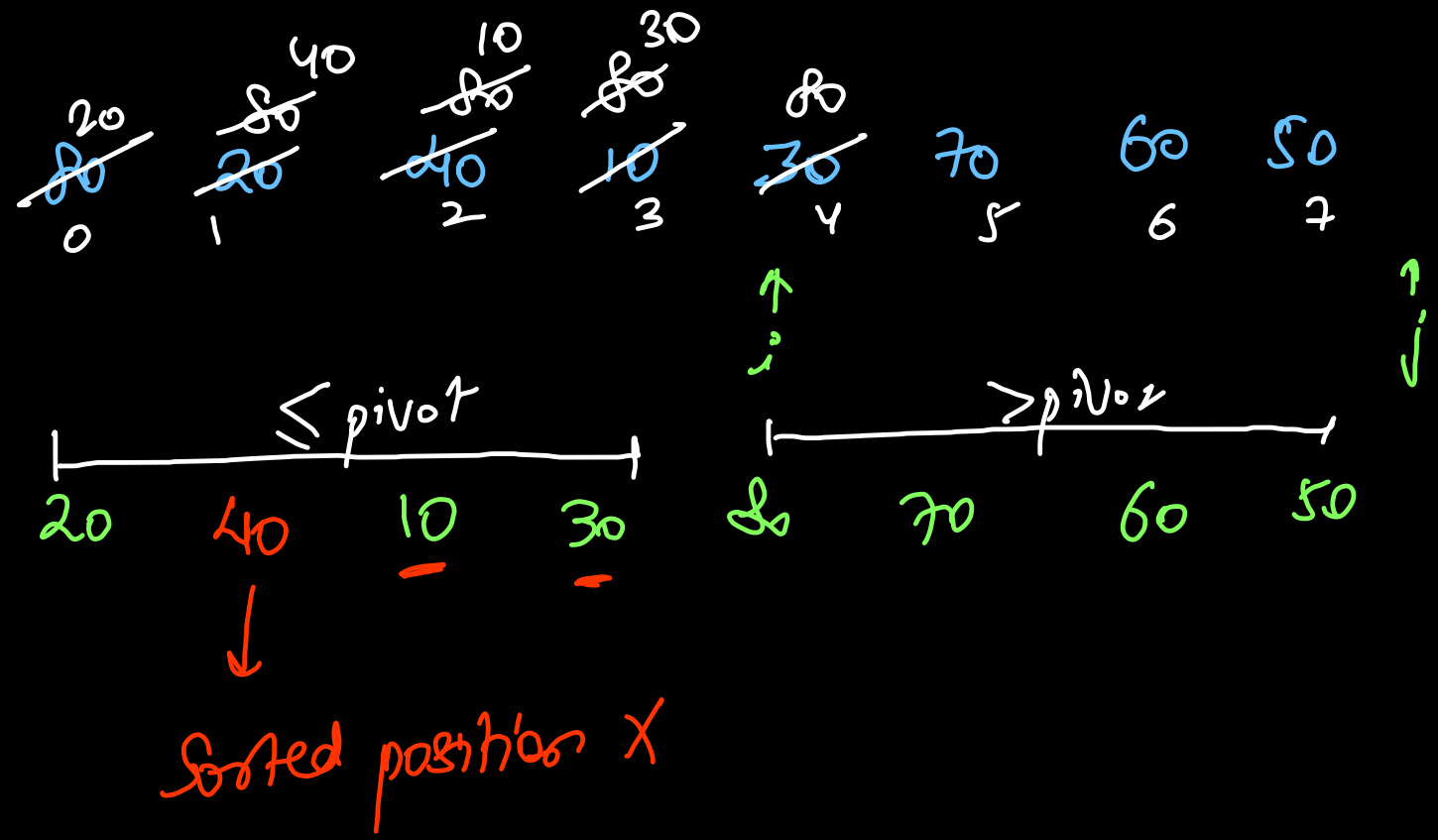
# Optimization

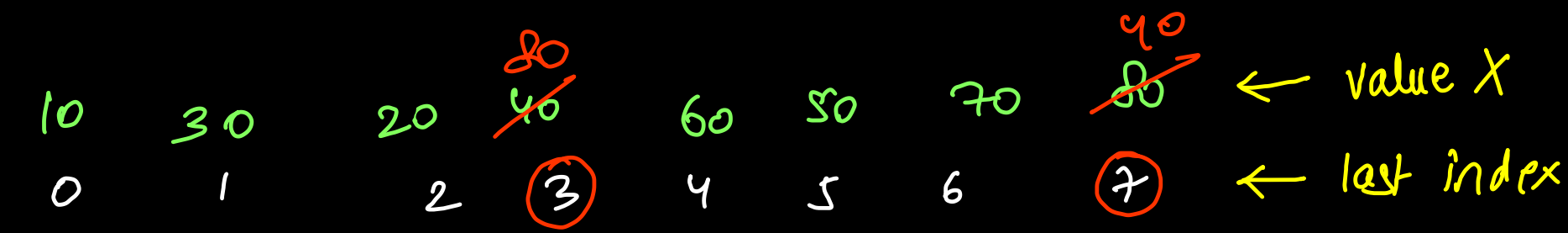
(randomized quicksort)

last index should be pivot ?

pivot = 40

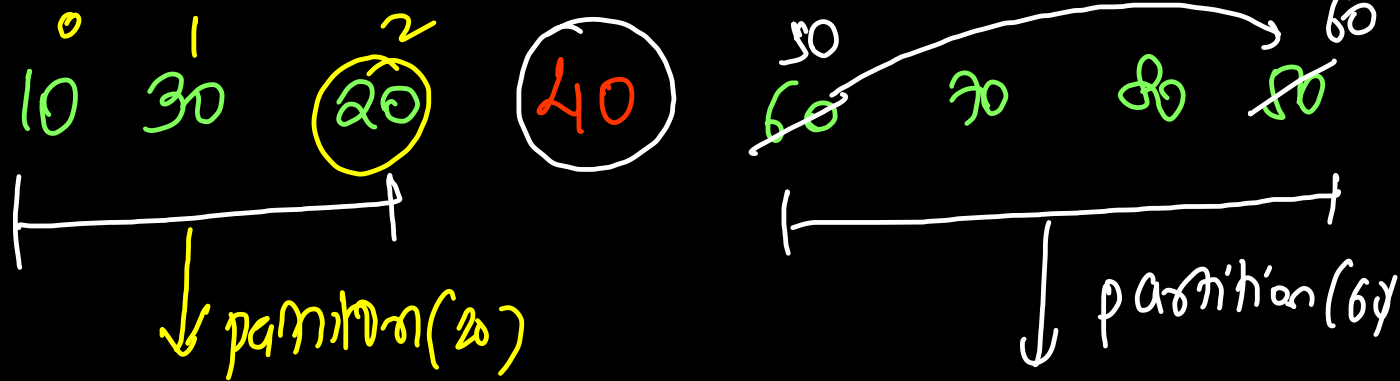
(what will go wrong if last index is not pivot)






random index = 3

partition(pivot = 40)





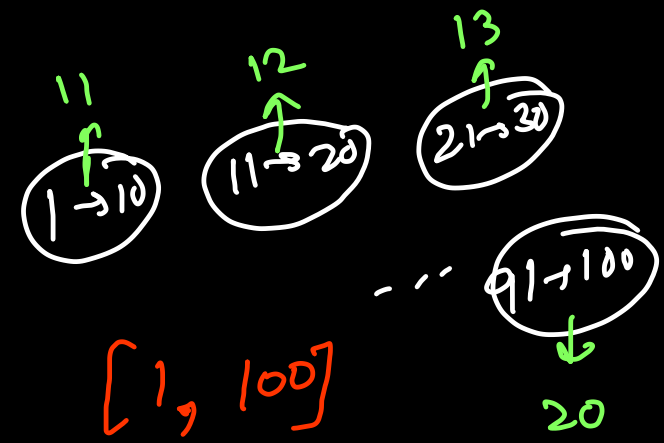


10 20 30

50 60 70 80

# randomized quick sort

- worst case will still remain $O(N^2)$
- chances of worst case will drastically reduce




[1, 100]

↓

[11, 20]

(java → inbuilt)

randomize()

double [0, 1]

↓ * (right - left + 1) + left

int [left, right]

eg [3, 7]

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

public int partition(int[] nums, int left, int right){
    int pivot = nums[right];
    int i = left, j = left;

    while(j <= right){
        if(nums[j] <= pivot){
            swap(nums, i, j);
            i++; j++;
        } else {
            j++;
        }
    }

    return (i - 1);
}
```

```java
public void randomize(int[] nums, int left, int right){
    double d = Math.random(); // random value between [0, 1]
    int idx = (int)(d * (right - left + 1) + left);
    swap(nums, idx, right);
}

public void quickSort(int[] nums, int left, int right){
    if(left >= right) return; // base case (0 or 1 size)
    randomize(nums, left, right); // randomization

    int mid = partition(nums, left, right); // preorder work
    quickSort(nums, left, mid - 1); // left recursive call (faith)
    quickSort(nums, mid + 1, right); // right recursive call (faith)
}

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

Lophiomi Sam

# Counting Sort

① → Count Sort
② → Radix Sort
③ → Bucket Sort

} frequency based

applicat$^n$: Frequency Sort

# quick sort / merge sort

↳ Time : O(nlogn) avg

→ Time ⇒ O(n+d) linear

Data Structure : frequency array
or
hashmap : O(d)

n = 11

eg1)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 18 | 16 | 13 | 15 | 16 | 18 | 20 | 17 | 16 | 15 |
| 13 | 13 | 15 | 15 | 16 | 16 | 16 | 17 | 18 | 18 | 20 |

| 2 | 0 | 2 | 3 | 1 | 2 | 0 | 1 |
|---|---|---|---|---|---|---|---|
| (13) 0 | (14) 1 | (15) 2 | (16) 3 | (17) 4 | (18) 5 | (19) 6 | (20) 7 |

range = d = 20 - 13 + 1 = 8

Properties

① repeating values

② smaller range of values

1st pass) Store frequencies → O(n)

+

2nd pass) Frequency array → O(d)

Time = O(n+d)

Space = O(d)

eg2) { 1 (0), 1000 (1), $10^9$ (2), $10^7$ (3), $-10^6$ (4) }

range : $10^9 - (-10^6) + 1 = \underline{10^9 + 10^6 + 1}$

↳ Counting sort will fail

↓

freq array ✗

java / c++ → max array ~ $10^6 - 10^7$ size

Limitation

① scattered values

② bigger range of values

eg: { -5, 4, 0, 2, -3, -4, 2, -5, 0, -1 }

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| ~~2~~ | 1 | 1 | 0 | 1 | ~~2~~ | 0 | ~~2~~ | 0 | 1 |
| (-5) 0 | (-4) 1 | (-3) 2 | (-2) 3 | (-1) 4 | (0) 5 | (1) 6 | (2) 7 | (3) 8 | (4) 9 |

range = d = 4 - (-5) + 1 = 10

counting sort can handle negatives also

# Sort array (912 LC)

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

    // Pass 1: input array = O(N)
    for(int ele : nums){
        freq[ele - min]++;
    }

    // Pass 2: frequency array = O(d)
    int idx = 0;
    for(int ele = min; ele <= max; ele++){
        while(freq[ele - min] > 0){
            nums[idx] = ele;
            idx++;
            freq[ele - min]--;
        }
    }

    return nums;
}
```

Variation 1

n = input array size, d = range

Time ⇒ O(n+d)

Space ⇒ O(d) inplace X

Stability ⇒ No

# Stable counting sort

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| arr | $13^A$ | $18^A$ | $16^A$ | $13^B$ | $15^A$ | $16^B$ | $18^B$ | 20 | 17 | $16^C$ | $15^B$ |
| output | $13^A$ | $13^B$ | $15^A$ | $15^B$ | $16^A$ | $16^B$ | $16^C$ | 17 | $18^A$ | $18^B$ | 20 |

| | (13) 0 | (14) 1 | (15) 2 | (16) 3 | (17) 4 | (18) 5 | (19) 6 | (20) 7 |
|---|---|---|---|---|---|---|---|---|
| freq | 2 | 0 | 2 | 3 | 1 | 2 | 0 | 1 |
| prefix | 2 | 2 | 4 | 7 | 8 | 10 | 10 | 11 |

1st pass) Create frequency array

2nd pass) Convert freq array → prefix array

3rd pass) reverse order of input

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

    // Pass 1: input array = O(N)
    for(int ele : nums){
        freq[ele - min]++;
    }

    // Pass 2: frequency array -> prefix array = O(d)
    for(int idx = 1; idx < freq.length; idx++){
        freq[idx] += freq[idx - 1];
    }

    // Pass 3: output sorted array creation = O(n)
    int[] res = new int[nums.length];
    for(int i = nums.length - 1; i >= 0; i--){
        int ele = nums[i];
        freq[ele - min]--;
        int last = freq[ele - min];
        res[last] = ele;
    }

    return res;
}
```

Time = O(n+d)

Space = O(n+d)

res ↑ freq

Stability ✓

inplace X

# Radix Sort

Counting sort + bigger range

321
982
799
638
710
099
900
853
637
542

(Stable count)

unit's place →

O(n+10)

710
900
321
982
542
853
637
638
799
099

ten's place →

O(n+10)

900
710
321
637
638
542
853
982
799
099

(last sort)

highest-priority ↑

hundred's place →

O(n+10)

099
321
542
637
638
710
799
853
900
982

$O(n \log_{10}^{max})$

$\approx O(n)$

Sorted ✓

no of passes = no of digits = constant = $\log_{10}^{max}$

Why not in 100 → 10 → 1 order?

321
982
799
638
710
099
900
853
637
542

lowest priority ↑

hundred →

099
321
542
638
637
799
710
853
982
900

tens →

900
710
321
638
637
542
853
982
099
799

highest priority ↑

ones →

900
710
321
542
982
853
637
638
099
799

$$digit = (ele \% (place * 10)) / place$$

ele = 54321

| | place | digit | |
|---|---|---|---|
| unit (1) → | | (1) | ele % 10 = 1 → 1/1 = 1 |
| tens (10) → | | (2) | ele % 100 = 21/10 = 2 |
| hundred (100) → | | (3) | ele % 1000 = 321/100 = 3 |
| thousand (1000) → | | (4) | ele % 10000 = 4321/1000 = 4 |

Stable count sort on a digit

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

    // Pass 1: input array = O(N)
    for(int ele : nums){
        int digit = (ele % (10 * place)) / place;
        freq[digit]++;
    }

    // Pass 2: frequency array -> prefix array = O(d)
    for(int idx = 1; idx < freq.length; idx++){
        freq[idx] += freq[idx - 1];
    }

    // Pass 3: output sorted array creation = O(n)
    int[] res = new int[nums.length];
    for(int i = nums.length - 1; i >= 0; i--){
        int ele = nums[i];
        int digit = (ele % (10 * place)) / place;
        freq[digit]--;
        int last = freq[digit];
        res[last] = ele;
    }

    return res;
}
```

```java
public int maximumGap(int[] nums) {
    // radix sort → count sort digit by digit
    for(int place = 1; place <= (int)1e8; place *= 10){
        nums = countSort(nums, place);
    }

    int gap = 0;
    for(int idx = 1; idx < nums.length; idx++){
        if(nums[idx] - nums[idx - 1] > gap){
            gap = nums[idx] - nums[idx - 1];
        }
    }
    return gap;
}
```

max^m gap b/w adjacent

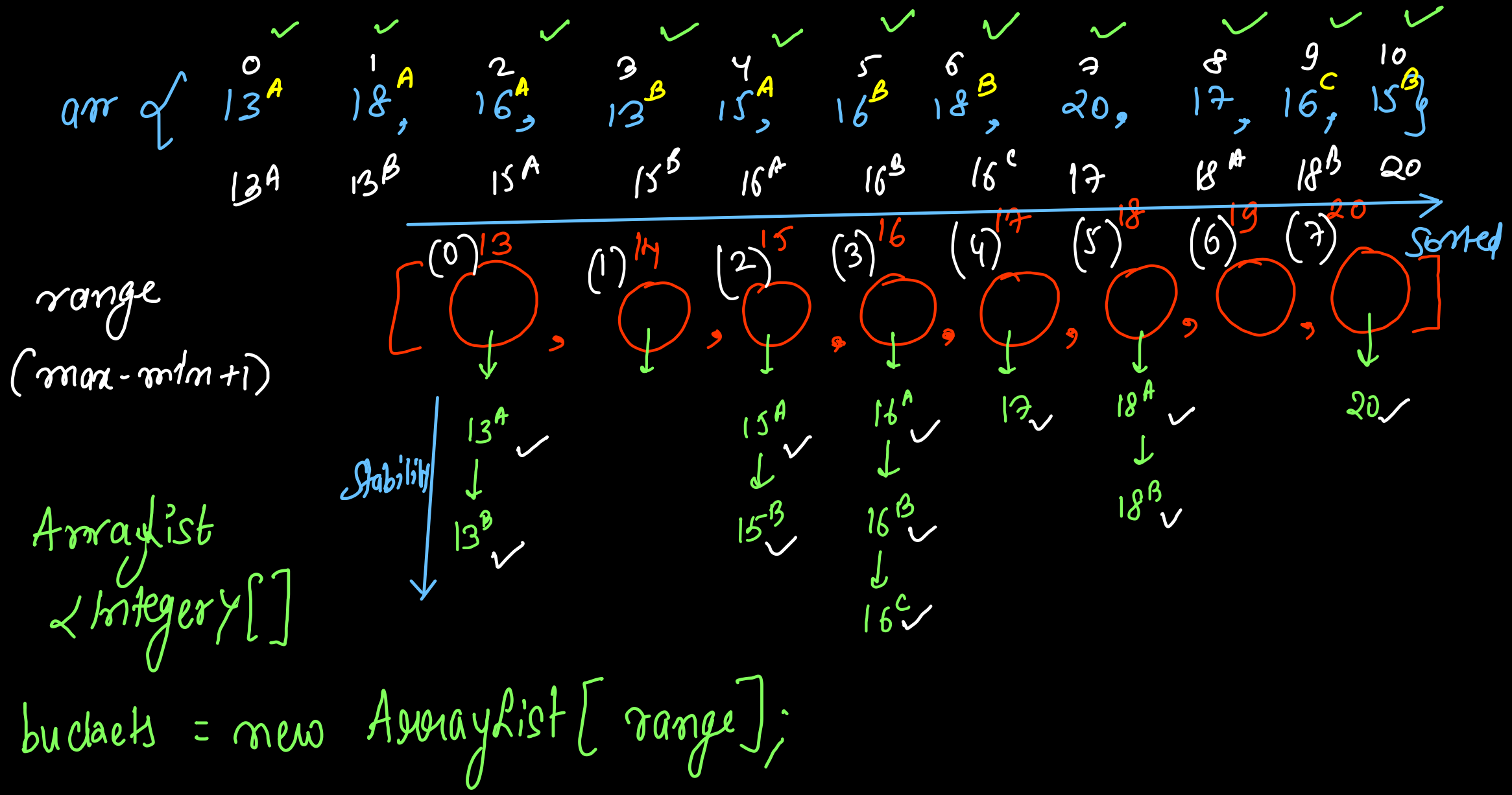
# LC 164) Maxm gap

time : O(n)

space : O(n)

linear

# Bucket Sort


arr of
0 1 2 3 4 5 6 7 8 9 10
13A 18A 16A 13B 15A 16B 18B 20 17 16C 15B
13A 13B 15A 15B 16A 16B 16C 17 18A 18B 20
Sorted
range
(max - min + 1)
(0)13 (1)14 (2)15 (3)16 (4)17 (5)18 (6)19 (7)20
Stability
13A → 13B
15A → 15B
16A → 16B → 16C
17
18A → 18B
20
ArrayList
<Integer>[]
buckets = new ArrayList[range];

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

    ArrayList<Integer>[] buckets = new ArrayList[max - min + 1];
    // all buckets are initialized to null

    for(int idx = 0; idx < buckets.length; idx++){
        buckets[idx] = new ArrayList<>();
        // all buckets are initalized with empty arraylist
    }

    // Pass 1: Insert Items in Bucket
    for(int ele : nums){
        buckets[ele - min].add(ele);
    }

    // Pass 2: Sorted Array from Bucket Array
    int idx = 0;
    for(int bucket = min; bucket <= max; bucket++){
        for(int ele : buckets[bucket - min]){
            nums[idx++] = ele;
        }
    }

    return nums;
}
```

O(d)

O(n)

O(d+n)

d

n

Stability ✓

time = O(n+d)

space = O(n+d)

# L(e 912) Sort Array

# Frequency Sort

LC 1636) Frequency Sort on Array

LC 451) Frequency Sort on String

LC 347) Top K frequent elements

LC 1636)

Priority 1) Increasing order of frequency

Priority 2) (If same freq) decreasing order of values

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

qoo input

| 10 | 10 | 10 | 60 | 70 | 70 |
|---|---|---|---|---|---|
| 8 | 9 | 10 | 11 | 12 | 13 |

frequency

key: ele ⇒ value: freq

| | |
|---|---|
| 10 → 3 | 50 → 3 |
| 20 → 2 | 60 → 1 |
| 30 → 1 | 70 → 2 |
| 40 → 2 | |

Output

| 60 | 30 | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | | | | | freq=1 |
| 70 | 70 | 40 | 40 | 20 | 20 | |
| 2 | 3 | 4 | 5 | 6 | 7 | freq=2 |
| 50 | 50 | 50 | 10 | 10 | 10 | |
| 8 | 9 | 10 | 11 | 11 | 12 | freq=3 |

integer → integer

key: ele ⇒ value. freq

| | |
|---|---|
| 10 → 3 | 50 → 3 |
| 20 → 2 | 60 → 1 |
| 30 → 1 | 70 → 2 |
| 40 → 2 | |

frequency array

inverse (bucket)

integer ⇒ list/Integer?

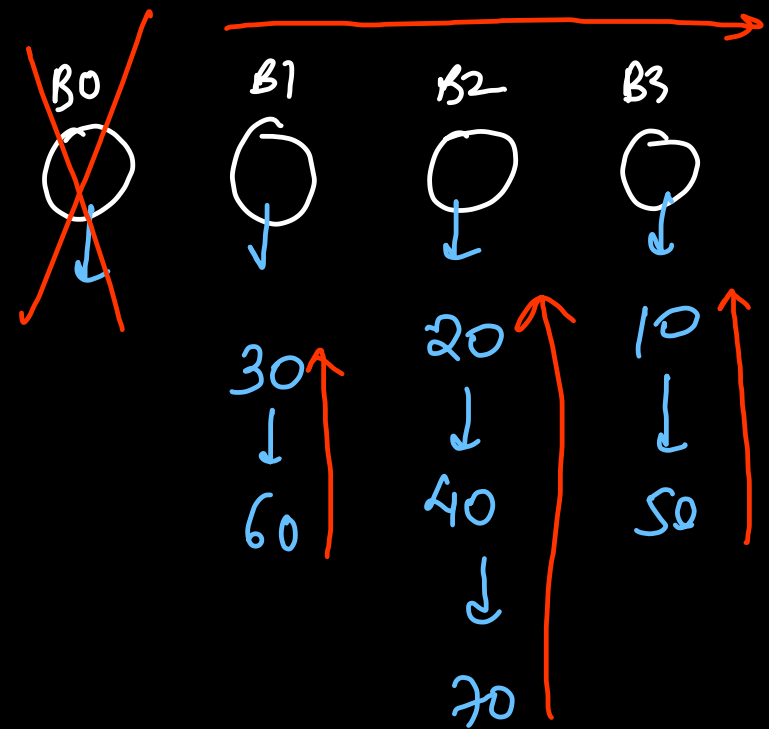
key: freq ⇒ value : ele(s)

P1

1 → { 30, 60 }

2 → { 20, 40, 70 }

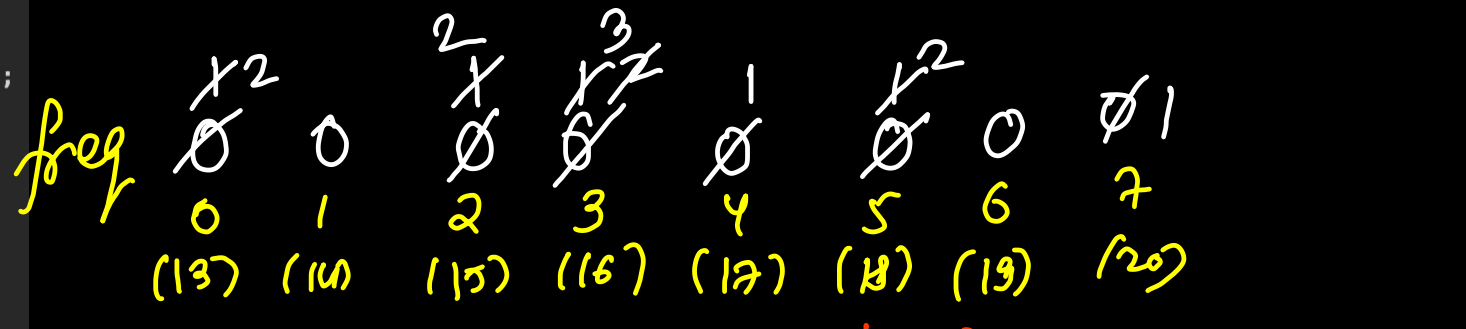
3 → { 10, 50 }

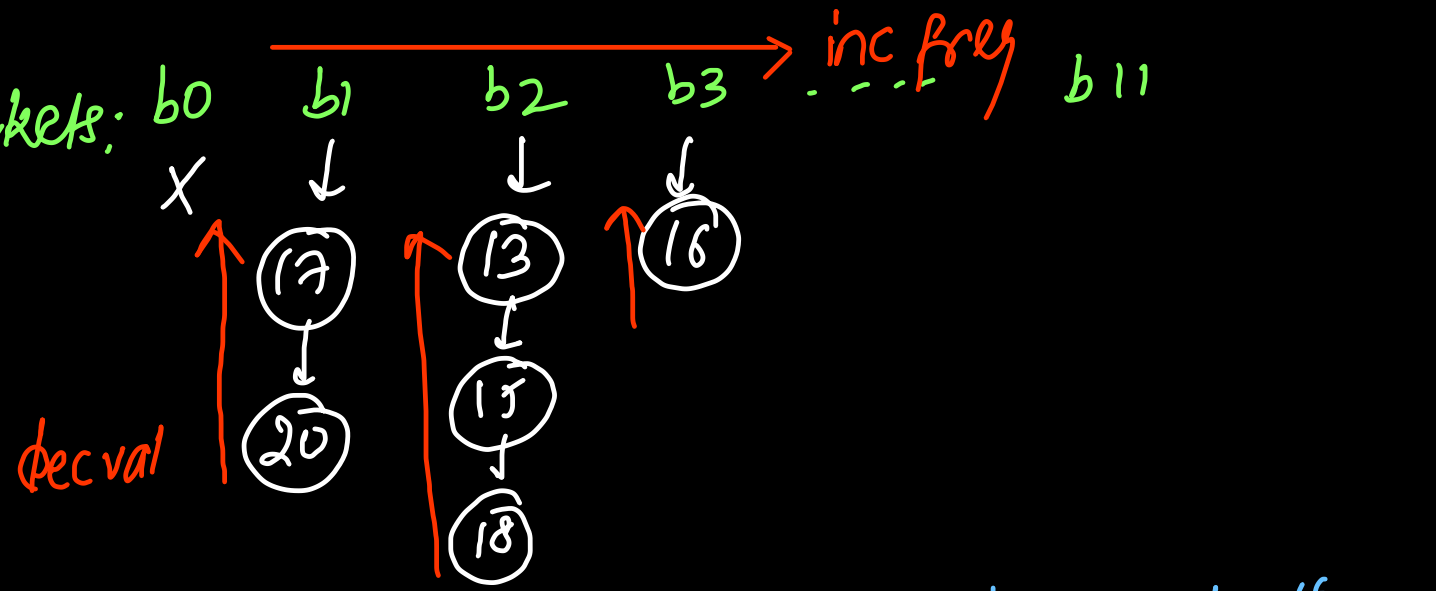


```java
public int[] frequencySort(int[] nums) {
    int min = -100, max = 100;
    int[] freq = new int[max - min + 1];

    // Pass 1: Create Frequency Array
    for(int ele : nums){
        freq[ele - min]++;
    }

    // Pass 2: Create Bucket Array
    // number of buckets = max freq of any ele + 1 = n + 1
    ArrayList<Integer>[] buckets = new ArrayList[nums.length + 1];

    for(int bucket = 0; bucket < buckets.length; bucket++){
        buckets[bucket] = new ArrayList<>();
    }

    for(int ele = min; ele <= max; ele++){
        int count = freq[ele - min];
        if(count == 0) continue;
        buckets[count].add(ele);
    }

    // Pass 3: Create Sorted Array
    int idx = 0;
    for(int bucket = 1; bucket < buckets.length; bucket++){
        Collections.reverse(buckets[bucket]);
        for(int ele : buckets[bucket]){
            for(int f = 0; f < bucket; f++){
                nums[idx++] = ele;
            }
        }
    }

    return nums;
}
```

Time → O(n+d)  Space → O(n+d)

arr of [13^A, 18^A, 16^A, 13^B, 15^A, 16^B, 18^B, 20, 17, 16^C, 15^B]

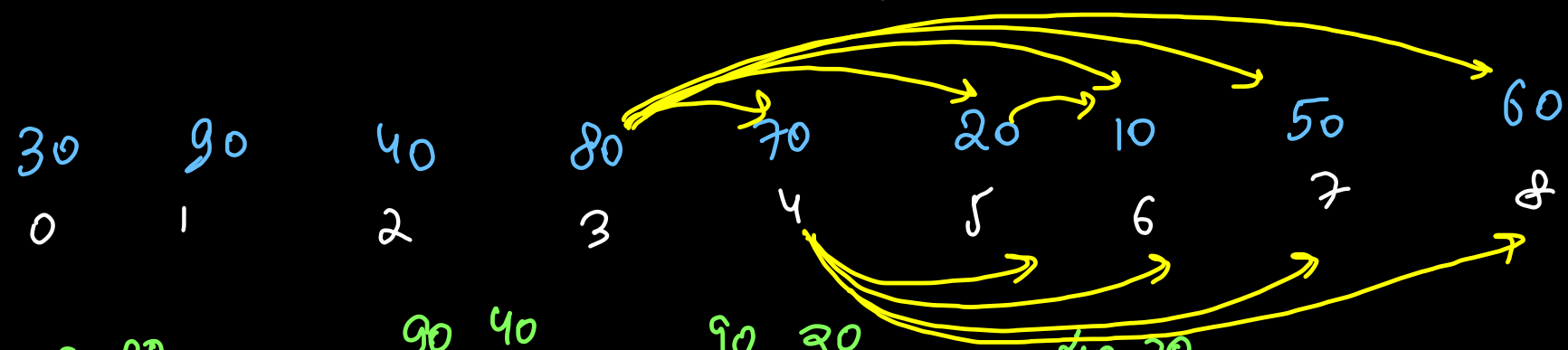


# Count Inversions (Merge sort)

GFG

Inversion Count : no of pairs (i, j) ∀ a[i] > a[j], i < j




30 20 (0, 5)

30 10 (0, 6)

90, 40 (1, 2)

90, 80 (1, 3)

90, 30 (1, 4)

90, 50 (1, 7)

90, 20 (1, 5)

90, 10 (1, 6)

90, 50 (1, 8)

40, 20 (2, 5)

40, 10 (2, 6)

output : 21

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

Brute force

outer loop → i

inner loop → j

Time => $O(n^2)$

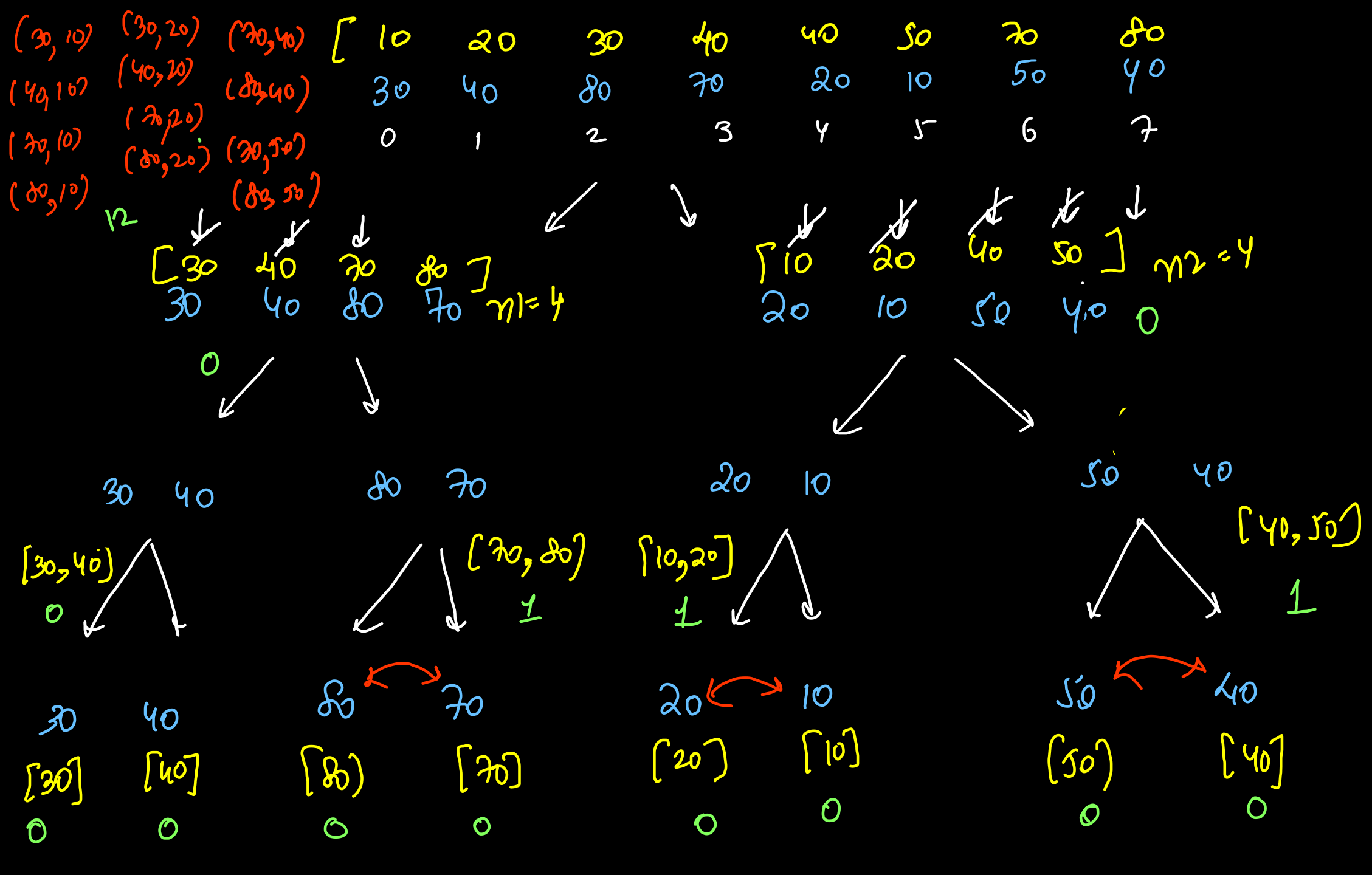
Space => $O(1)$

$n = 10^5 \rightarrow n^2 = 10^{10}$

TLE

Optimized Approach

Merge Sort


(30,10) (30,20) (30,40)
(40,10) (40,20) (80,40)
(70,20)
(70,10) (80,20) (70,50)
(80,10) (80,50)
[ 10 20 30 40 40 50 70 80
30 40 80 70 20 10 50 40
0 1 2 3 4 5 6 7
12
[30 40 70 80 ] n1 = 4
30 40 80 70
0
[10 20 40 50 ] n2 = 4
20 10 50 40
0
30 40
[30,40]
0
80 70
[70,80]
1
20 10
[10,20]
1
50 40
[40,50]
1
30 40
[30] [40]
0 0
80 70
[80] [70]
0 0
20 10
[20] [10]
0 0
50 40
[50] [40]
0 0

```java
static long count = 0;

static long[] merge(long arr1[], long arr2[]) {
    int n1 = arr1.length, n2 = arr2.length;
    long[] arr3 = new long[n1 + n2];
    int p1 = 0, p2 = 0, p3 = 0;

    while(p1 < n1 && p2 < n2){
        if(arr1[p1] <= arr2[p2]){
            arr3[p3] = arr1[p1];
            p3++; p1++;
        } else {
            count += (n1 - p1);
            arr3[p3] = arr2[p2];
            p3++; p2++;
        }
    }

    while(p1 < n1){
        arr3[p3] = arr1[p1];
        p3++; p1++;
    }

    while(p2 < n2){
        arr3[p3] = arr2[p2];
        p3++; p2++;
    }

    return arr3;
}
```

global variable (accessible everywhere)

*

```java
static long[] mergeSort(long[] nums, int left, int right){
    if(left == right)
        return new long[]{ nums[left] };

    int mid = left + (right - left) / 2;
    long[] larr = mergeSort(nums, left, mid);
    long[] rarr = mergeSort(nums, mid + 1, right);
    return merge(larr, rarr);
}

static long inversionCount(long nums[], long N) {
    count = 0; // for each test case
    mergeSort(nums, 0, nums.length - 1);
    return count;
}
```

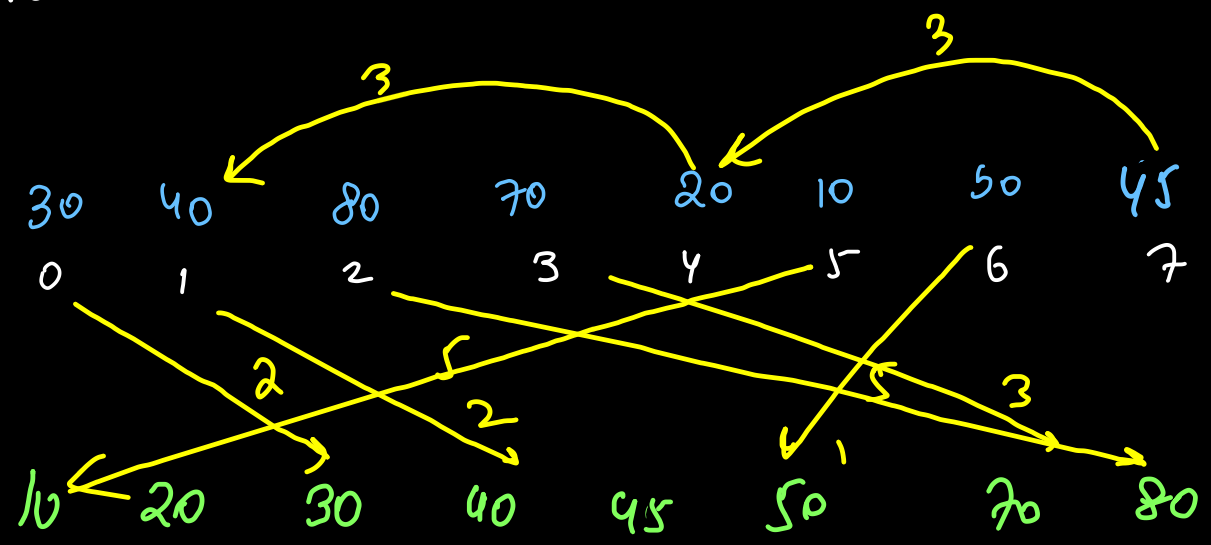
T1: count = 15 ✓

T2: count = ~~10~~ 10+15 ✓

time = O(nlogn), space = O(n)

# Min swaps to Sort (adjacent Swaps)

= inversion count




{4,1,2,3}

(3)

(4,1) (4,2) (4,3)

Swaps = 24/2

= (12)

# Reverse Pairs (LC 493)

$(i, j)$ $i < j$

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

| 19 | 25 | 2 | 9 | 6 | 40 | 12 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

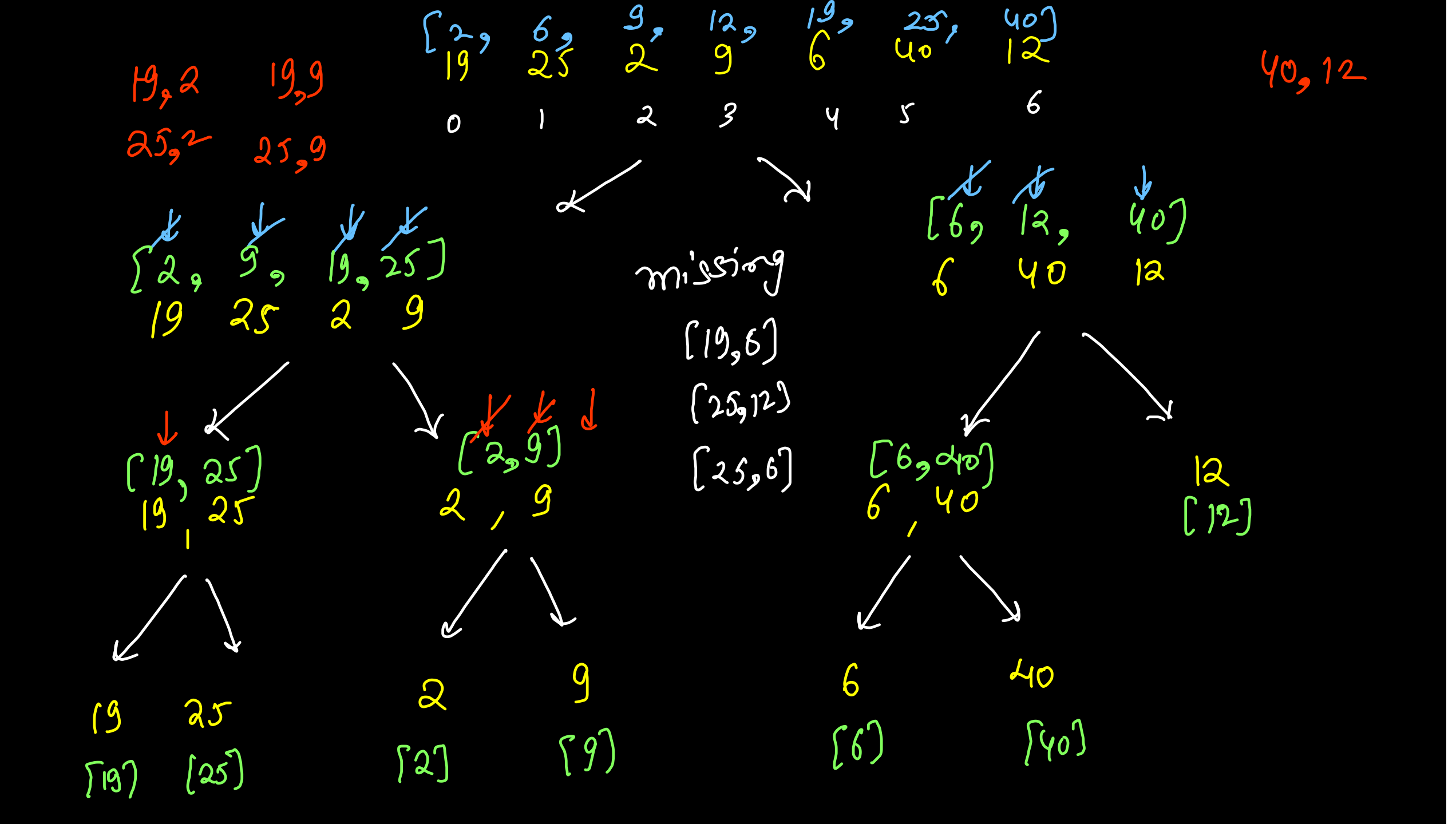
count = 8

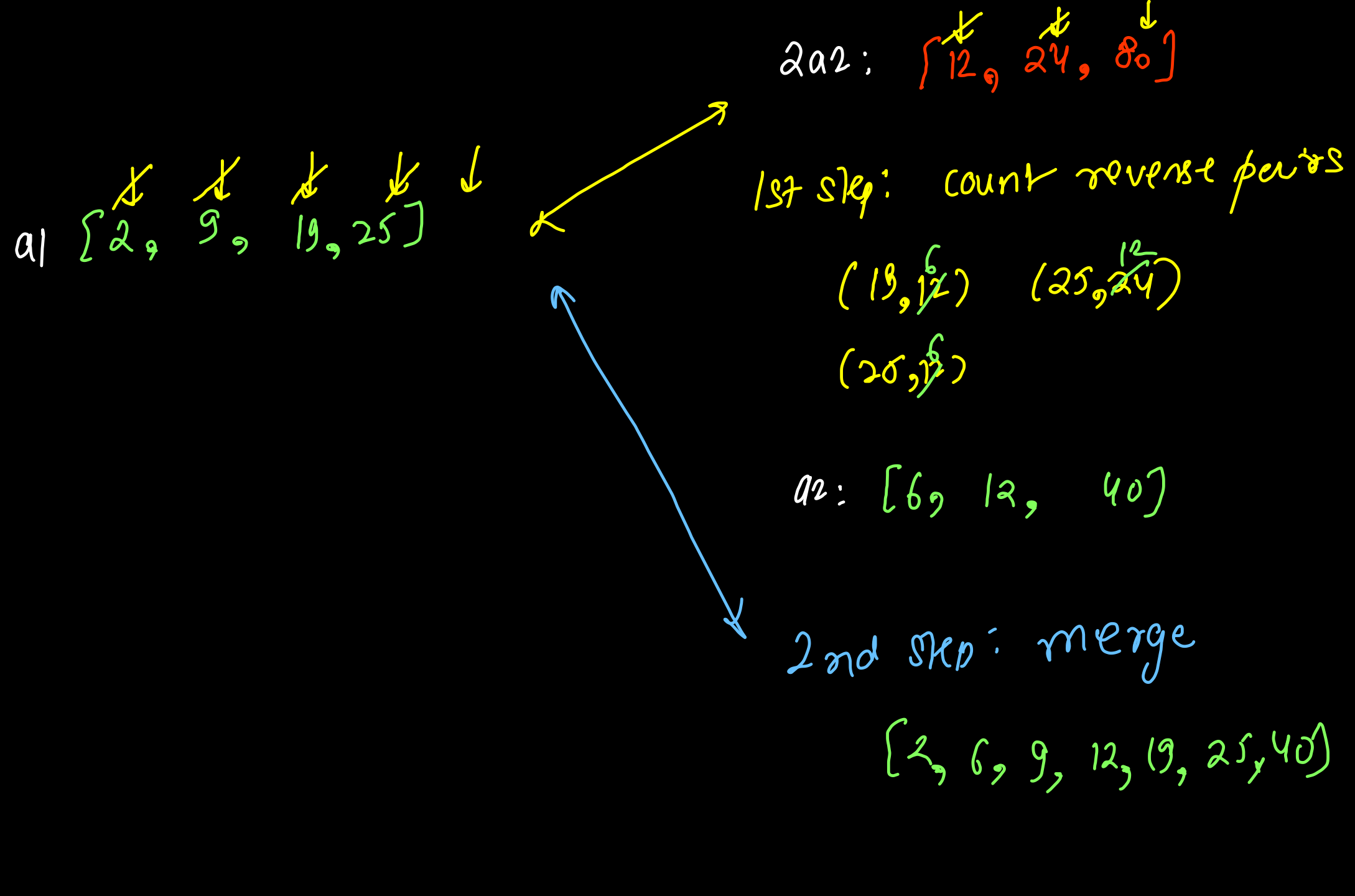
(19, 2) (25, 2)

(19, 9) (25, 9)

(19, 6) (25, 6)

(40, 12) (25, 12)


19, 2
19, 9
25, 2
25, 9
[2, 6, 9, 12, 19, 25, 40]
19 25 2 9 6 40 12
0 1 2 3 4 5 6
40, 12
[2, 9, 19, 25]
19 25 2 9
missing
[19, 6]
[25, 12]
[25, 6]
[6, 12, 40]
6 40 12
[19, 25]
19 25
[2, 9]
2 9
[6, 40]
6 40
12
[12]
19 25
[19] [25]
2 9
[2] [9]
6 40
[6] [40]


a2: [12, 24, 80]
a1 [2, 9, 19, 25]
1st step: count reverse pairs
(19, 12) (25, 24)
(20, 12)
a2: [6, 12, 40]
2nd step: merge
[2, 6, 9, 12, 19, 25, 40]

```java
int count = 0;

void countPairs(int[] arr1, int[] arr2){
    int n1 = arr1.length, n2 = arr2.length;
    int p1 = 0, p2 = 0;

    while(p1 < n1 && p2 < n2){
        if(arr1[p1] <= 2l * arr2[p2]){
            p1++;
        } else {
            count += (n1 - p1);
            p2++;
        }
    }
}

int[] merge(int arr1[], int arr2[]) {
    int n1 = arr1.length, n2 = arr2.length;
    int[] arr3 = new int[n1 + n2];
    int p1 = 0, p2 = 0, p3 = 0;

    countPairs(arr1, arr2);

    while(p1 < n1 && p2 < n2){
        if(arr1[p1] <= arr2[p2]){
            arr3[p3] = arr1[p1];
            p3++; p1++;
        } else {
            arr3[p3] = arr2[p2];
            p3++; p2++;
        }
    }
```

count $a1, a2$

merge $a1, a2$

```java
    while(p1 < n1){
        arr3[p3] = arr1[p1];
        p3++; p1++;
    }

    while(p2 < n2){
        arr3[p3] = arr2[p2];
        p3++; p2++;
    }

    return arr3;
}

int[] mergeSort(int[] nums, int left, int right){
    if(left == right)
        return new int[]{ nums[left] };

    int mid = left + (right - left) / 2;
    int[] larr = mergeSort(nums, left, mid);
    int[] rarr = mergeSort(nums, mid + 1, right);
    return merge(larr, rarr);
}

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

Time $4 n \log n$

Space $4n$

# Count of substrings in a Binary String that contains more 1s than 0s

Google Coding assessment question