

# Searching & Sorting

## Lecture ① & ②

- Bubble Sort
  - Selection Sort
  - Insertion Sort
  - merge 2 sorted Arrays
  - Merge Sort
  - Target sum Pair
- Bubble Sort, Selection Sort, Insertion Sort are grouped under  $\underline{\mathcal{O}(n^2)}$ .

## Lecture ③

- Partition Array
- Quick Sort
- Sort 01
- Sort 012
- Quick Select

## Lecture ④ & ⑤

- Count Sort
- Radix Sort
- Sort Dates
- Binary Search
- Broken Economy
- First & last Index
- Rotated Sorted Array

# Data Structure Operations Cheat Sheet

Data Structure Name	Average Case Time Complexity				Worst Case Time Complexity				Space Complexity
	Accessing $n^{th}$ element	Search	Insertion	Deletion	Accessing $n^{th}$ element	Search	Insertion	Deletion	
Arrays	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stacks	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queues	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Binary Trees	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Trees	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Balanced Binary Search Trees	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$
Hash Tables	N/A	$O(1)$	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Note: For best case operations, the time complexities are  $O(1)$ .

Narasimha

## Sorting Algorithms Cheat Sheet

Sorting Algorithm Name	Time Complexity			Space Complexity Worst Case	Is Stable?	Sorting Class Type	Remarks
	Best Case	Average Case	Worst Case				
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Comparison	Not a preferred sorting algorithm.
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Comparison	In the best case (already sorted), every insert requires constant time.
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Comparison	Even a perfectly sorted array requires scanning the entire array.
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	Yes	Comparison	On arrays, it requires $O(n)$ space; and on linked lists, it requires constant space.
Heap Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$	No	Comparison	By using input array as storage for the heap, it is possible to achieve constant space.
Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(logn)$	No	Comparison	Randomly picking a pivot value can help avoid worst case scenarios such as a perfectly sorted array.
Tree Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(n)$	Yes	Comparison	Performing inorder traversal on the balanced binary search tree.
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Yes	Linear	Where $k$ is the range of the non-negative key values.
Bucket Sort	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n)$	Yes	Linear	Bucket sort is stable, if the underlying sorting algorithm is stable.
Radix Sort	$O(dn)$	$O(dn)$	$O(dn)$	$O(d + n)$	Yes	Linear	Radix sort is stable, if the underlying sorting algorithm is stable.

Duplicates

→ Inplace Without Extra Space

## Bubble Sort

4.0 5 9 8 2 1

3.0 5 8 2 1 9

Best Case 2 5 8 9  
already sorted  $\rightarrow O(N)$

4.1 5 9 8 2 1

3.1 5 8 2 1 9

2.1 2 5 8 9

1.0 2 1 5 8 9

4.2 5 8 2 1

3.2 5 2 1 9

2.1 2 5 8 9  
2 1 5 8 9

4.3 5 8 2 1

5 2 1 8 9

1 2 5 8 9

5 8 2 1 9

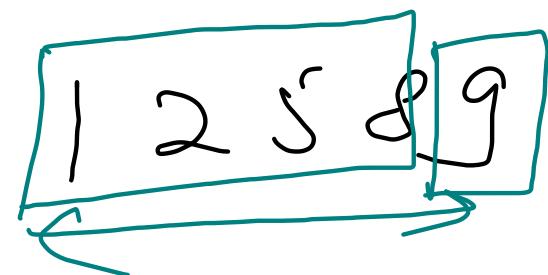
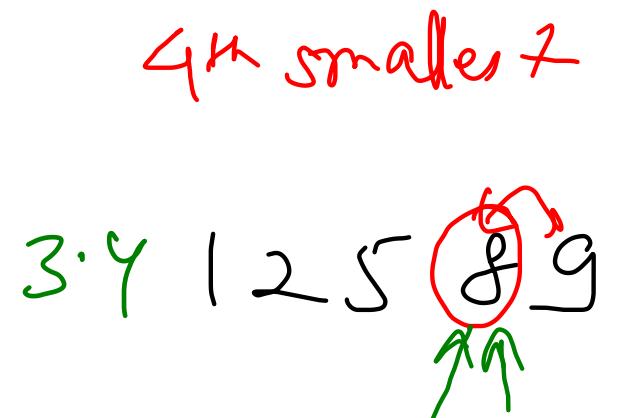
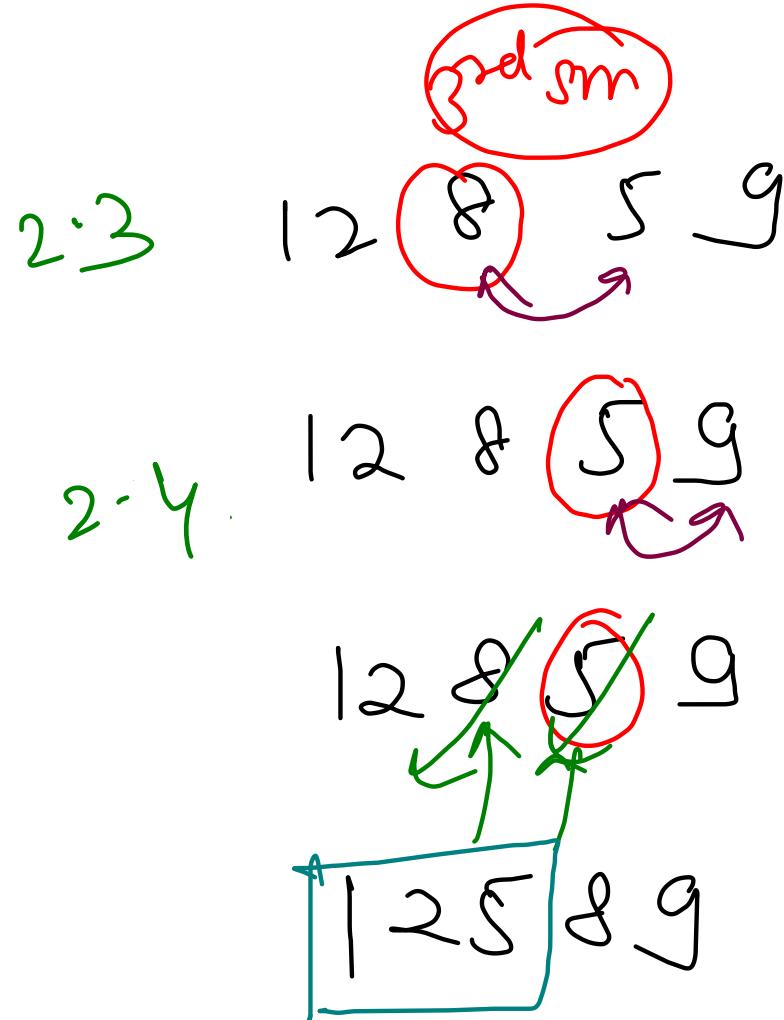
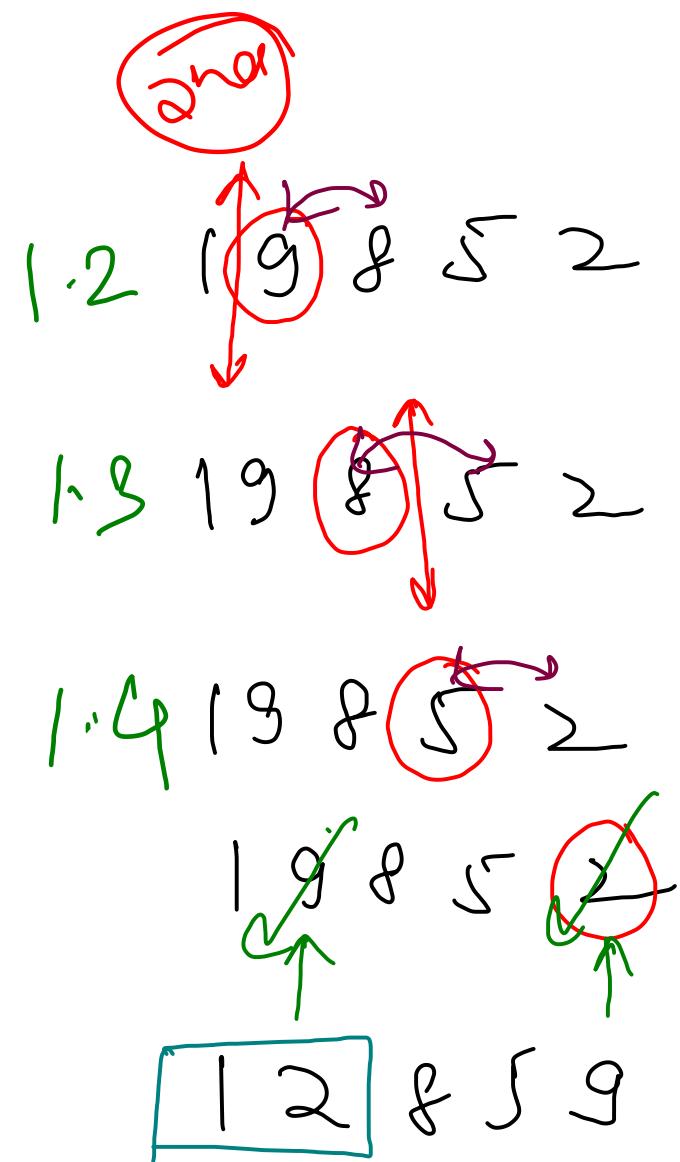
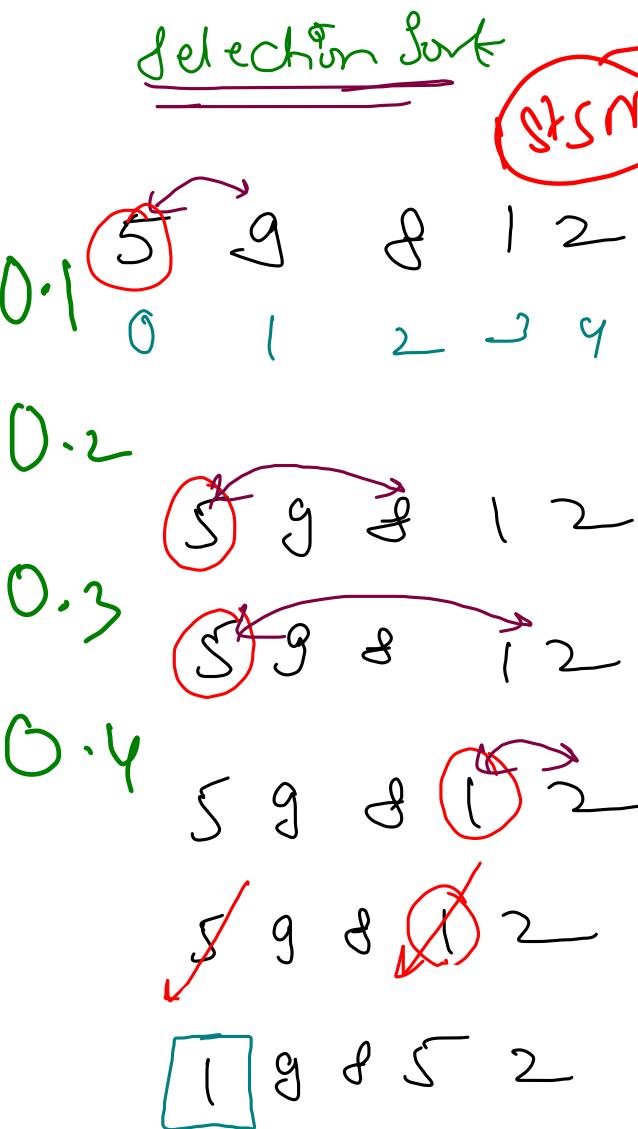
$$(n-1) + (n-2) + (n-3) + \dots + 1$$

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

$$= \frac{n^2}{2} - \frac{n}{2} \quad O(N^2)$$

→ adjacent swaps  
→ heavy ele

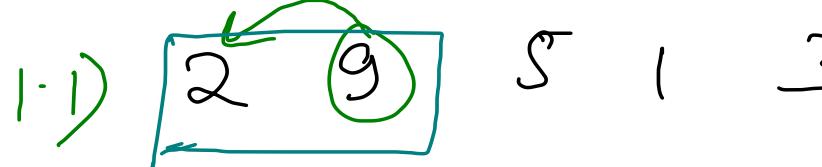
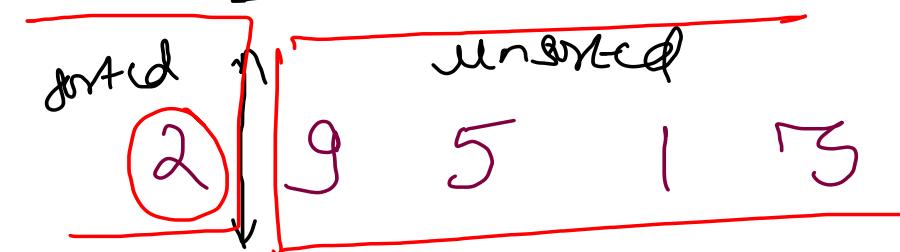
```
public static void bubbleSort(int[] arr) {
    for(int i=arr.length-1; i>0; i--){
        for(int j=0; j<i; j++){
            if(isSmaller(arr, j + 1, j)){
                swap(arr, j + 1, j);
            }
        }
        if(count == 0) return;
    }
}
```



$O(n^2)$

```
public static void selectionSort(int[] arr) {
    for(int i=0; i<arr.length-1; i++){
        int minIdx = i;
        for(int j=i+1; j<arr.length; j++){
            if(isSmaller(arr, j, minIdx)){
                minIdx = j;
            }
        }
        swap(arr, i, minIdx);
    }
}
```

## Insertion Sort



3.3) 2 5 9 (1) 3

3.2) 2 5 (1) 9 3

3.1) 2 (1) 5 9 3

1 2 5 9 3

2.2) 2 9 (5) 1 3

2.1) 2 (5) 9 1 3

4.4) 1 2 5 9 (3)

4.3) 1 2 5 (3) 9

4.2) 1 2 (3) 5 9

1 2 3 5 9

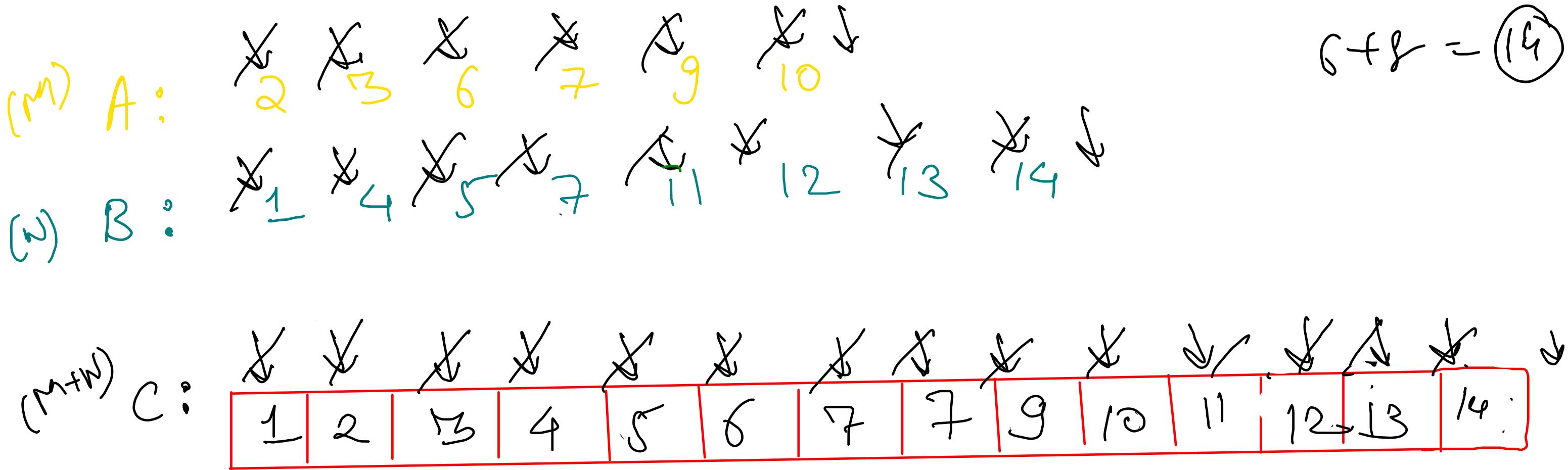
```
for(int i=1; i<arr.length; i++){
    for(int j=i; j>0; j--){
        if(isGreater(arr, j - 1, j)){
            swap(arr, j - 1, j);
        } else {
            break;
        }
    }
}
```

$O(n^2)$

{ best case  
1 2 3 4 5  
 $O(n)$  }

- merge 2 sorted Arrays
- Merge Sort → Recursion
- Target sum Pair
- Partition Array
- Quick Sort → Recursion
- Sort 01

## Merge 2 Sorted Arrays = {Two Pointer Technique}



```
// write your code here
int[] c = new int[a.length + b.length];

int i = 0, j = 0, k = 0;
while(i < a.length && j < b.length){
    if(a[i] <= b[j]){
        c[k] = a[i];
        i++; k++;
    } else {
        c[k] = b[j];
        j++; k++;
    }
}

while(i < a.length){
    c[k] = a[i];
    i++; k++;
}

while(j < b.length){
    c[k] = b[j];
    j++; k++;
}

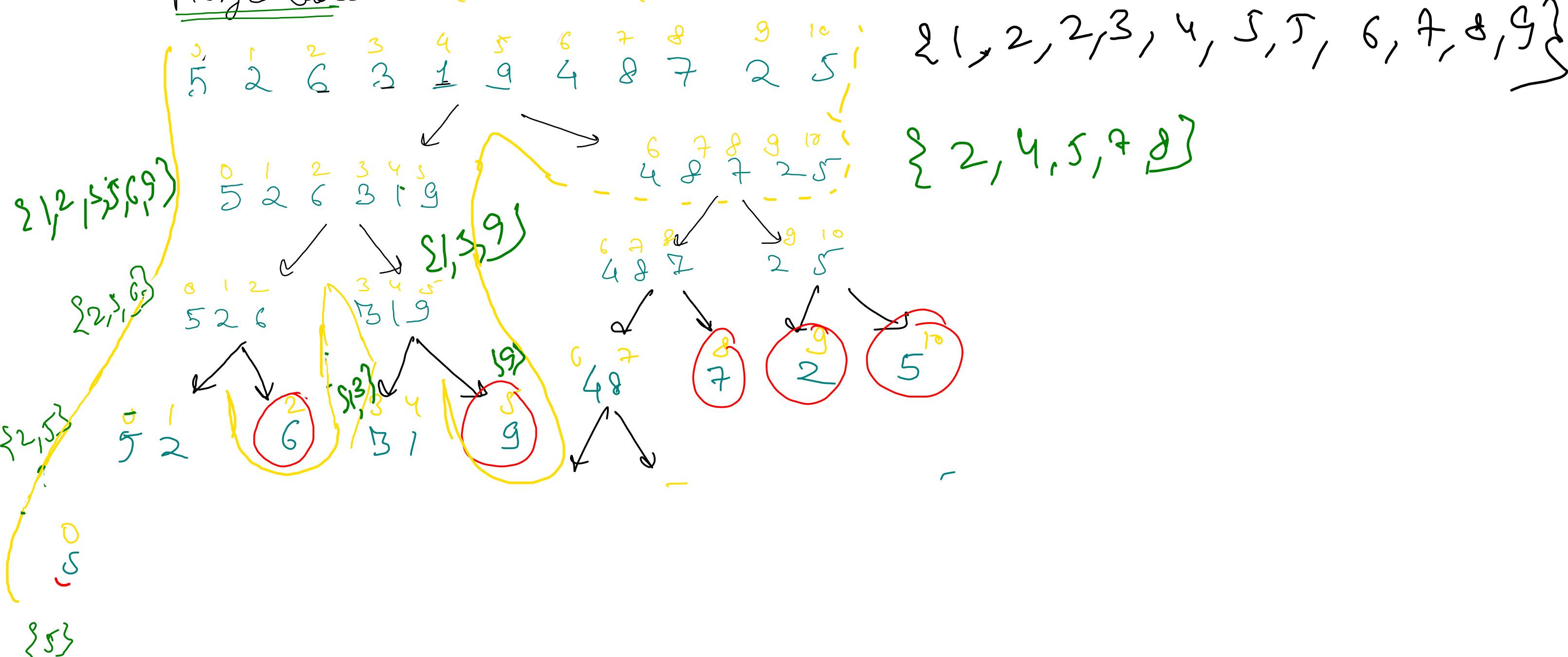
return c;
```

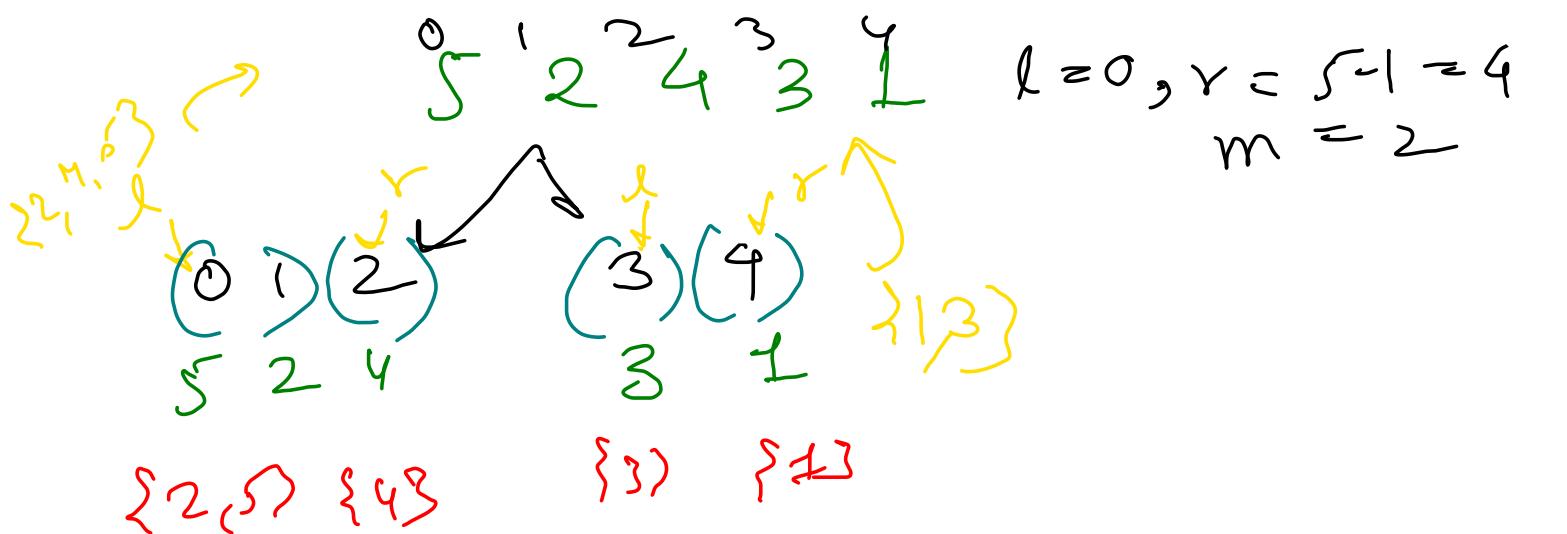
$O(n+m)$  ↗ worst case  
↗ avg case  
↗ best case

## Merge Sort

Divide & Conquer → Merge (postorder)

Faith





```

public static int[] mergeSort(int[] arr, int left, int right) {
    if(left == right){
        int[] base = new int[1];
        base[0] = arr[left];
        return base;
    }

    int mid = (left + right) / 2;

    int[] leftSorted = mergeSort(arr, left, mid);
    int[] rightSorted = mergeSort(arr, mid + 1, right);

    int[] sorted = mergeTwoSortedArrays(leftSorted, rightSorted);
    return sorted;
}

```

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Divide      Conquer  
 Fast      Merge  
 $\frac{n}{2}$        $n \log_2 n$

$\frac{n}{2} + (k+n) \log_2 n$

$O(n \log_2 n)$

Best      Avg      Worst case

$$\{ T(n) = 2T(n/2) + O(n) \} * 2^0$$

$$\{ T(n/2) = 2T(n/4) + O(n/2) \} * 2^1$$

$$\{ T(n/4) = 2T(n/8) + O(n/4) \} * 2^2$$

$$\{ T(n/8) = 2T(n/16) + O(n/8) \} * 2^3$$



$$\{ T(1) = 2T(0) + O(1) \} * 2^k$$

$$\begin{aligned}
 T(n) &= \\
 &\cancel{2^0 * n} + \cancel{2^1 * n/2} \\
 &\cancel{+ 2^2 * \frac{n}{4}} + \cancel{2^3 * \frac{n}{8}} \\
 &\dots \cancel{2^x * \frac{N}{2^x}} \\
 &+ 2^n * T(0)
 \end{aligned}$$

$$= n * x + 2^x k$$

$$\begin{aligned}
 &= n \log_2 n + 2^{\log_2 n} k \\
 &= \cancel{O(n \log_2 n)} + \cancel{n}
 \end{aligned}$$

Space  
comp.

① Extra space / Auxiliary space

$\Theta \rightarrow O(N)$   
Lat merging }

②

Input/Output  $\Theta \rightarrow O(N)$

③

Recursion call stack  $\Theta \rightarrow O(\log N)$

<u>Target</u>	<u>Sum</u>	<u>Pair</u>	<u>(Array is sorted)</u>	$\nearrow$ two pointer
0	1	2	3	4

10    12    13    14    18

↓      ↑  
left    right

target = 25

(Nested loop)

① Brute force

Time:  $\rightarrow O(n^2)$

Space:  $\rightarrow O(1)$

$$10 + 18 = 28 > 25$$

$$\rightarrow \text{nums(left)} + \text{nums(right)} > \text{target}$$

$$\Rightarrow \text{right} = -$$

$$10 + 14 = 24 < 25$$

$$\rightarrow \text{nums(left)} + \text{nums(right)} < \text{target}$$

$$\Rightarrow \text{left}++$$

$$12 + 14 = 26 > 25$$

$$12 + 13 = 25$$

$$\text{nums(left)} + \text{nums(right)} = \text{target}$$

$$\Rightarrow \text{return that pair.}$$

~~LCX~~



target = 25

$l == r \rightarrow \text{pair not found}$

$$10 + 18 = 28 > 25$$

$$10 + 14 = 24 < 25$$

$$13 + 14 = 27 > 25$$

```

int left = 0, right = nums.length - 1;
while(left < right){
    int sum = nums[left] + nums[right];
    if(sum == target){
        return new int[]{left + 1, right + 1};
    }
    if(sum < target){
        left++;
    } else {
        right--;
    }
}
return null;
    
```

$$T(n) = T(n-1) + O(1) \Rightarrow O(N)$$

~~Yank~~  
Array & Sort(    nums )

Tim Sort

Merge Sort  
 $N > 16$

+  
Inversion Sort  
 $N < 16$

Small Array  
Insertion Sort  
better

Partition An Array

X

Quick Sort

X

Quick Select

X

Sort 01

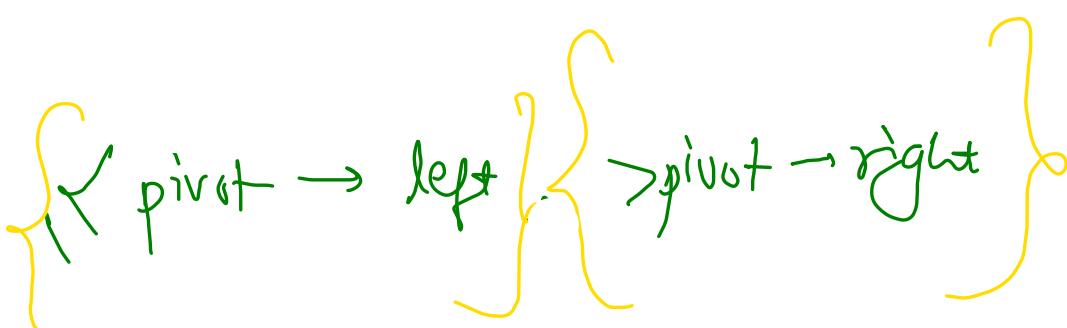
X

Sort 012

Searching & Sorting - lecture ③

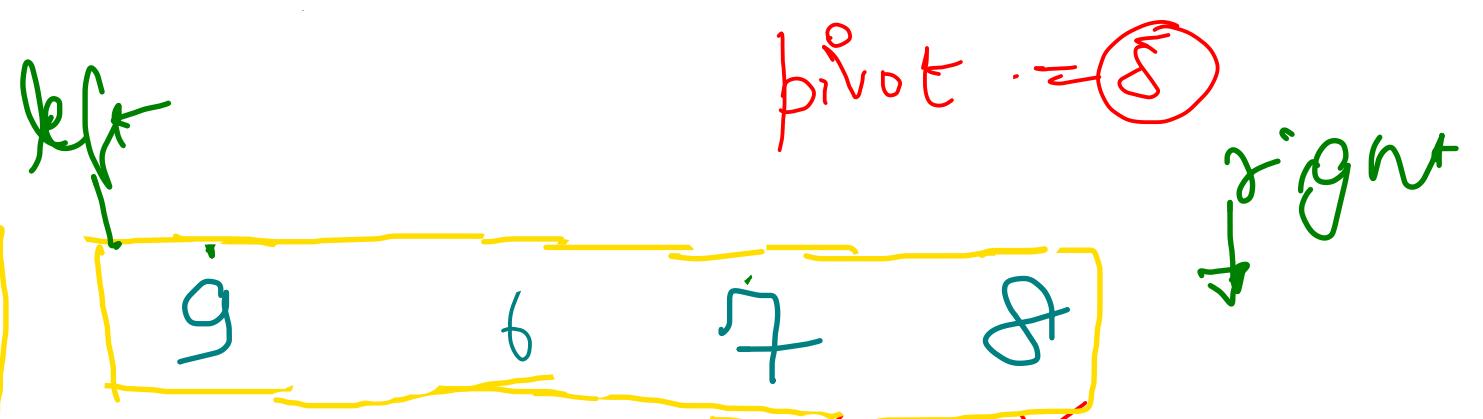
## Partition An Array

$(l-r-n)$   
 completed  
 $(0-l-1)$   
 < pivot  
 $\geq$  pivot  
 $(l-r-1)$



left → first ele of greater region

right → first ele of unexplored region



$\text{pivot} = 8$   
 $\text{left} = 0, \text{right} = 0$   
 while ( $\text{right} < \text{arr.length}$ )  
 if ( $\text{arr}[\text{right}] > \text{pivot}$ )  
      $\text{right}++$   
 else {  
     swap( $\text{arr}[l], \text{arr}[r]$ );  
      $\text{left}++; \text{right}++$   
 }

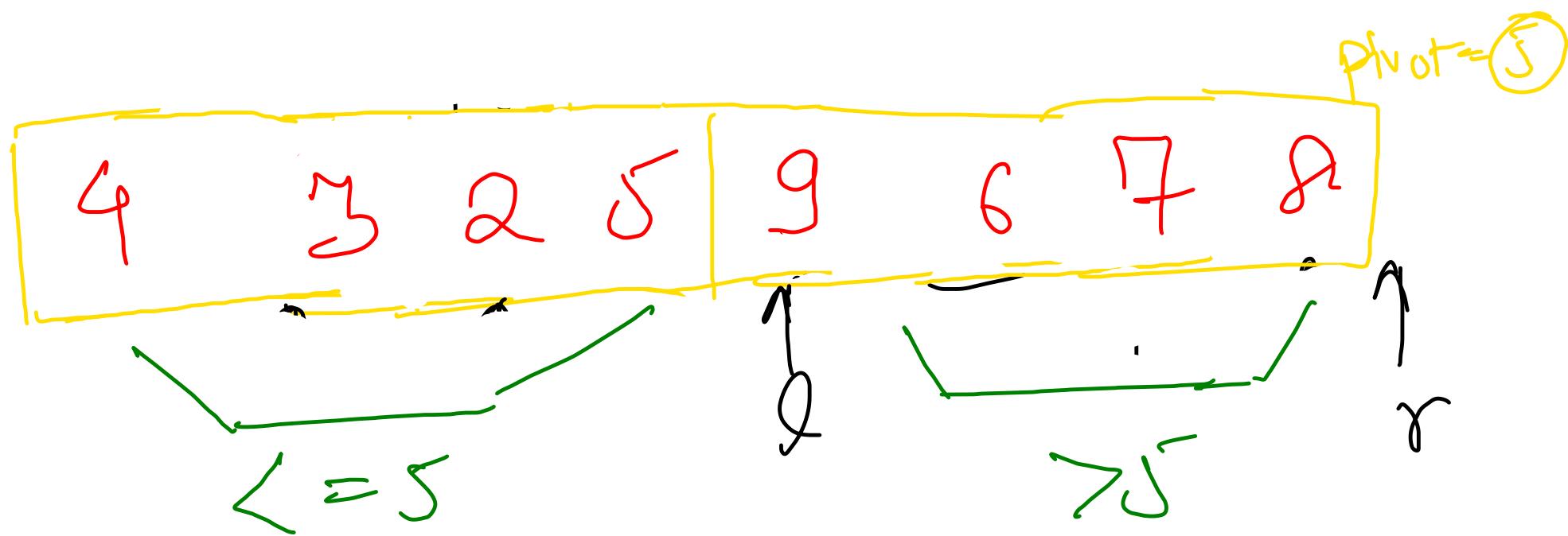
3

```

int left = 0, right = 0;

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

```



Sort 01

{Binary Array}



3 region

- ① Unemployed ( $r - n$ )
- ② 0's ( $0 - l - 1$ )
- ③ 1's ( $l - r - 1$ )

① Two traversals

1st traversal :

Count of zeros

Count of ones

2nd traversal :

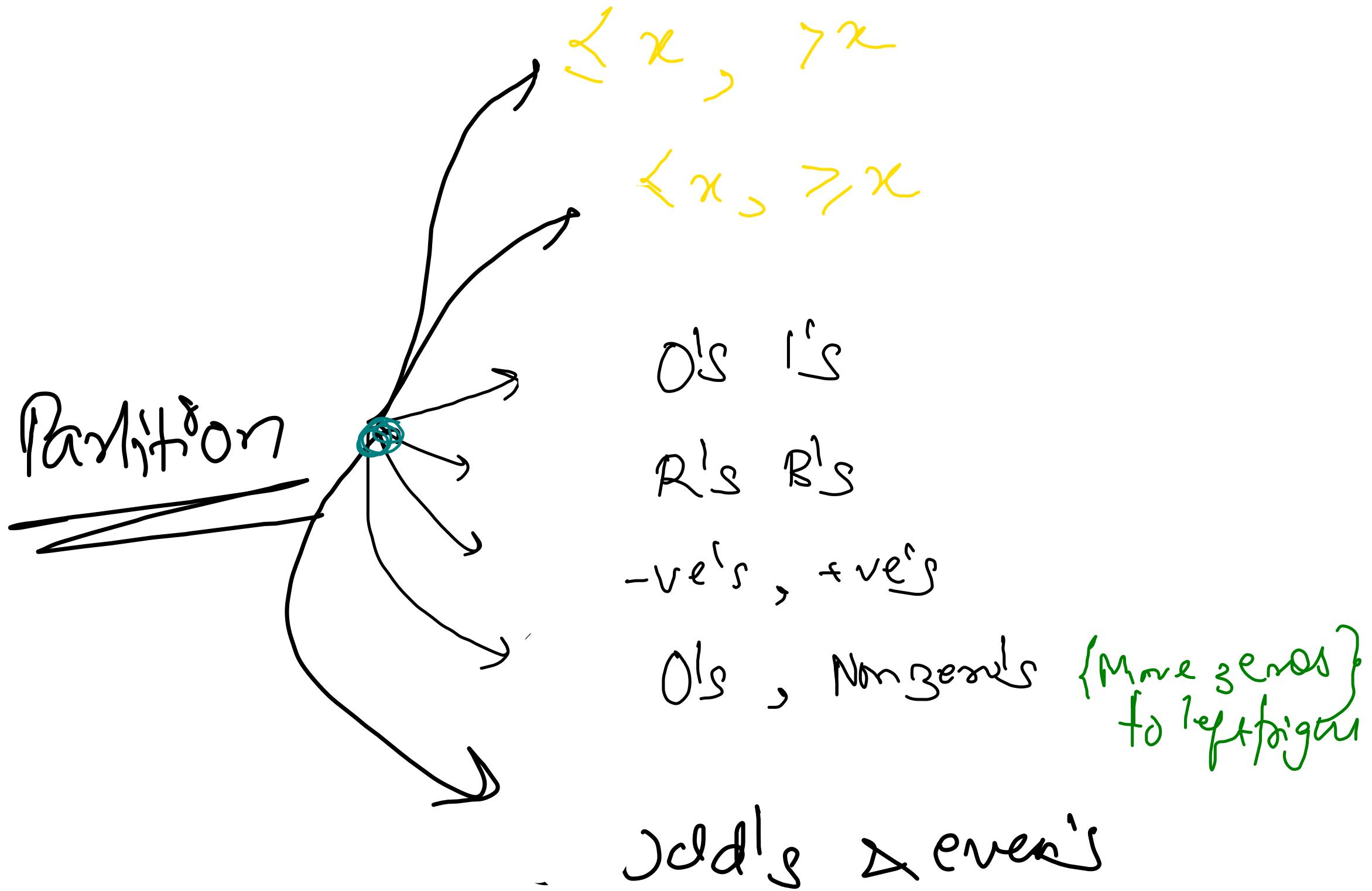
Sort based on count

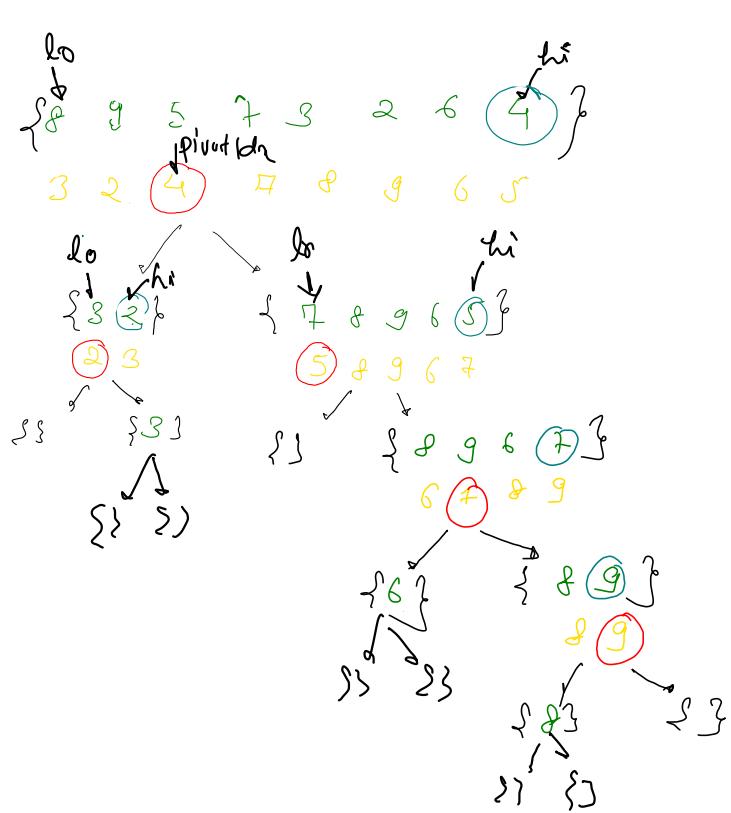
② Single traversal

```
int left = 0, right = 0;
```

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

## Variations





Quick Sort → Partition

Expectation

```
public static void quickSort(int[] arr, int lo, int hi) {  
    if(lo > hi){  
        // no element -> base case -> already return  
        return;  
    }  
  
    int pivotIdx = partition(arr, arr[hi], lo, hi); → Measuring Comp (Conquer)  
    // partition around last element -> last ele is at current index  
  
    quickSort(arr, lo, pivotIdx - 1); ↓ Faith/Decrease  
    quickSort(arr, pivotIdx + 1, hi);  
}
```

Decrease & Conquer

Space  
Input/output  $\rightarrow O(N)$

```
public static int partition(int[] arr, int pivot, int lo, int hi) {  
    System.out.println("pivot -> " + pivot);  
    int i = lo, j = lo;  
    while (i <= hi) {  
        if (arr[i] <= pivot) {  
            swap(arr, i, j);  
            i++;  
            j++;  
        } else {  
            i++;  
        }  
    }  
    System.out.println("pivot index -> " + (j - 1));  
    return (j - 1);  
}
```

Extra space  $\rightarrow O(1)$   
Recursion call stack  $\rightarrow O(n)$   
worst comp (depth of Recursion tree)  
Avg case  $O(h)$   
 $= O(\log n)$

# Partition → In Bcoader

# Partition : Subarray  $[l-r]$

↳ pivot : why it should  
be arr[right]?

→ Partitioning on pivot  
will result in  
pivot element  
to be on its  
current index.  
(Sorted index)

→ Best case }  
Avg case }  
\* | Worst case } ↗ loc  
                    ↘ pec

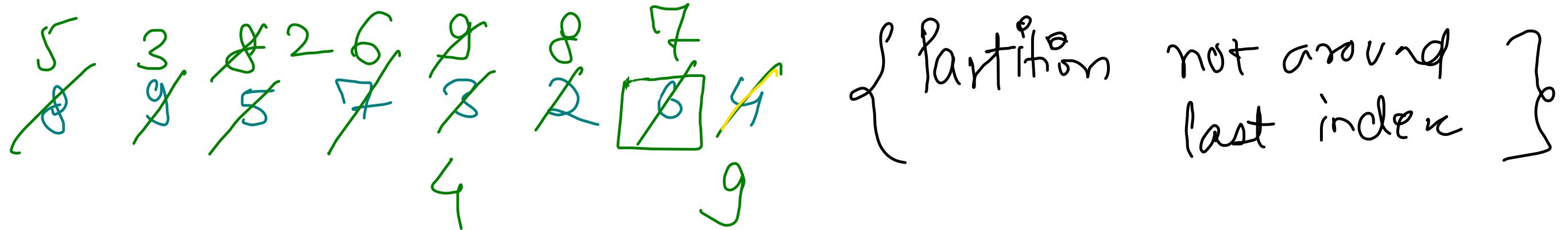
→ Stability

→ Merge Sort vs Quick Sort

→ Extra space }

→ Randomized QS -

→ Improving Quick Sort



(i) 6 is at correct index

(ii)  $<_6$  left,  $>_6$  right

(iii) left, right

# Searching & Sorting - Lecture ④

→ Quick Sort

→ Quick Select

→ Sort 012

→ Count Sort

→ Radix Sort

→ Sort Dates

Any case /  
best case

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

partition

$\Theta(n \log n)$   
same as  
merge sort

Worst case

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

partition

$O(n^2)$   
corner cases  
Already sorted  
(mc/dec)

# Randomized Quicksort  
 ↗ (left to right) range, pick any random index, last swap,  
 partition  
 ↘ Probability of hitting the worst case

2 1 5 5 7 6 8 4  $\Rightarrow$  Nearly / Almost sorted

$(2 \ 3 \ 1) \ 4 \ (5 \ 7 \ 8 \ 6)$   
 ↓  
 $(5) \ 0 \ (8 \ 7)$

Arg  $\Theta(n \log n)$   $\xrightarrow{\text{Sorted}} =$   
 Worstcase  $\Theta(n^2)$

## Merge Sort

Best }  
Avg }  $O(n \log n)$  Time  
Worst }

Extraspaces :  $\rightarrow O(n)$

Inplace  $\times$  { Merge 2 sorted  
Array }

Stability

MS is preferred for  
non sequential (LL)

## Quick Sort

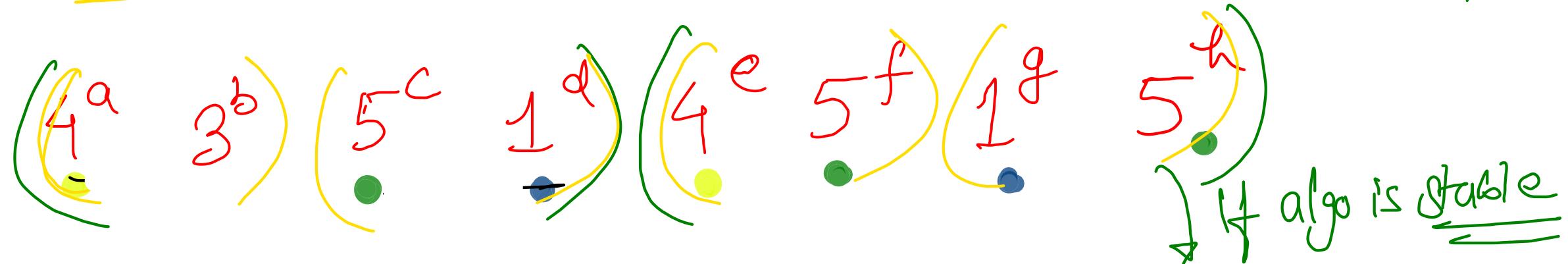
Best, Avg  $\Rightarrow \Theta(n \log n)$   
Worst case  $\Rightarrow O(n^2)$

Extra space :  $\rightarrow O(1)$

Stable

QS is preferred for  
Arrays

Stability  $\Rightarrow$  Input  $\rightarrow$  Duplicate Elements  $\Rightarrow$  Same order of Output as input



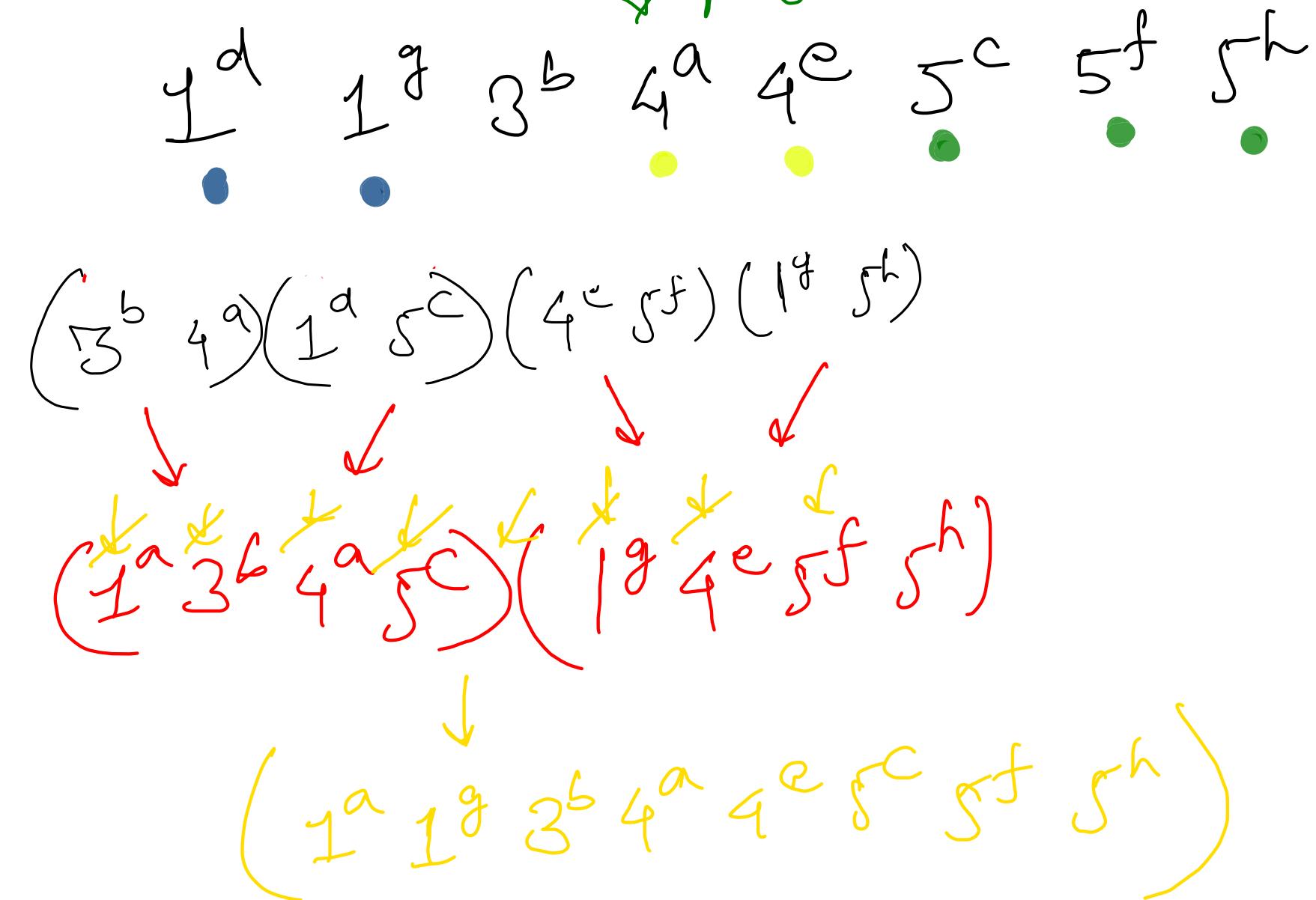
```
//write your code here
int[] c = new int[a.length + b.length];

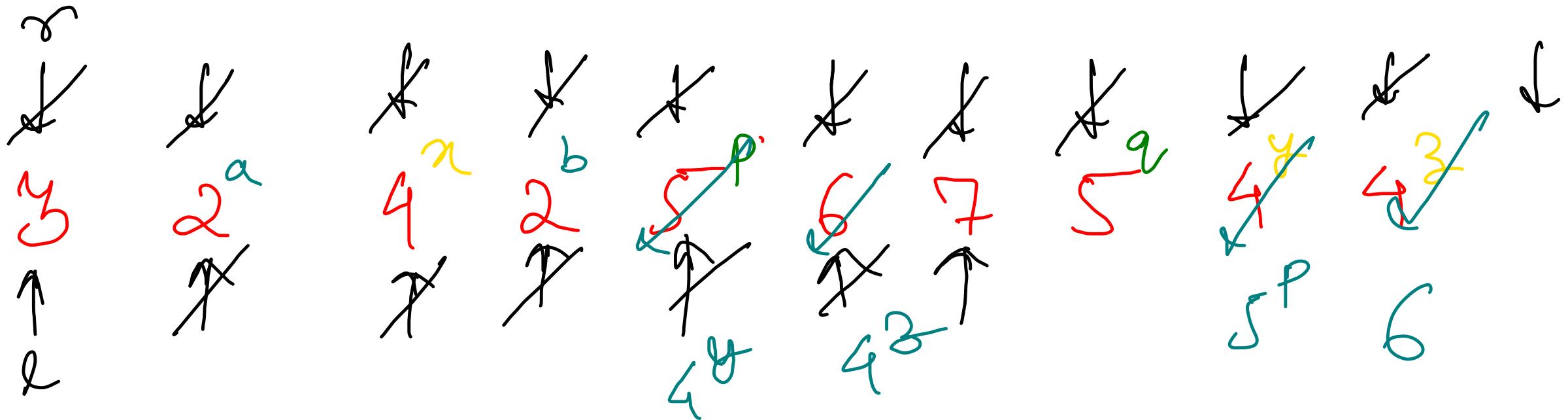
int i = 0, j = 0, k = 0;
while(i < a.length && j < b.length){
    if(a[i] <= b[j]){
        c[k] = a[i];
        i++; k++;
    } else {
        c[k] = b[j];
        j++; k++;
    }
}

while(i < a.length){
    c[k] = a[i];
    i++; k++;
}

while(j < b.length){
    c[k] = b[j];
    j++; k++;
}

return c;
```





```

public static int partition(int[] arr, int pivot, int lo, int hi) {
    System.out.println("pivot -> " + pivot);
    int i = lo, j = lo;
    while (i <= hi) {
        if (arr[i] <= pivot) {
            swap(arr, i, j);
            i++;
            j++;
        } else {
            i++;
        }
    }
    System.out.println("pivot index -> " + (j - 1));
    return (j - 1);
}

```

$O(N)$  time

$3 \ 2^a \ 4^a \ 2^b \ 4^y \ 4^z \ 7 \boxed{5^q \ 5^p \ 6}$

"Quicksort is not stable  
due to stability"

Quick select {  $k^{\text{th}}$  smallest } → Binary Search / Divide & Conquer

0 1 2 3 4 5 6 7  
8 3 5 7 6 1 4 2

$$k = 4$$

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

```
public static int quickSelect(int[] arr, int lo, int hi, int k) {
    if(lo == hi) return arr[lo];

    int pivotIdx = partition(arr, arr[hi], lo, hi);
    if(pivotIdx == k) return arr[pivotIdx];
    if(pivotIdx < k)
        return quickSelect(arr, pivotIdx + 1, hi, k);
    return quickSelect(arr, lo, pivotIdx - 1, k);
}
```

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

$$T(n/2) = T(n/4) + O(n/2)$$

$$T(n/4) = T(n/8) + O(n/4)$$

$$T(n/8) = T(n/16) + O(n/8)$$

$$T(1) = T(0) + O(1)$$

$O(n^r)$

$$T(n) = k + n \left\{ 1 + \left\{ \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \dots + \frac{1}{2^n} \right\} \right\}$$

$$a=1, r=1/2;$$

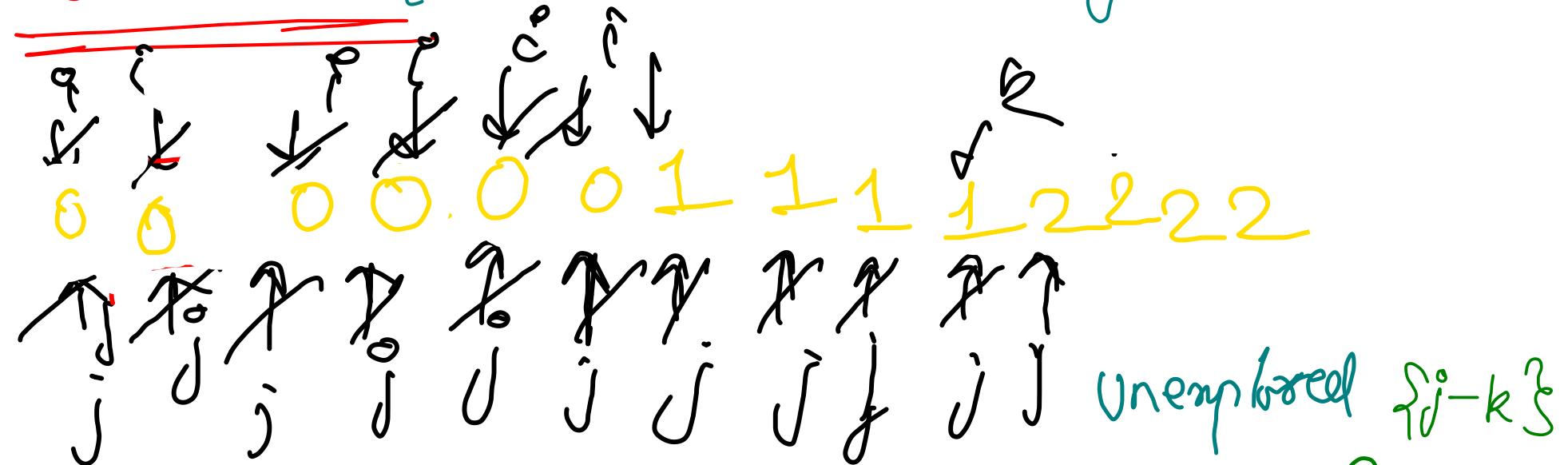
$$x=O_{2^{-N}}$$

$$\frac{a(1-r^n)}{(1-r)} = 1 * \left( 1 - \left( \frac{1}{2} \right)^{O_{2^{-N}}} \right)$$

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

$$= 2 - \frac{2}{n}$$

sort 02 { Dutch National Flag Algorithm } } DNF Sort }



Unemployment  $\{j-k\}$

0's  $\{0-j-1\}$

1's  $\{i-(j-1)\}$

2's  $\{k+1-n\}$

$$T(n) = T(n-1) + \Theta(1)$$

$\boxed{\Theta(n)}$

② Single Traversal

```
int i = 0, j = 0, k = arr.length-1;
while(j <= k){
    if(arr[j] == 0){
        swap(arr, j, i);
        i++; j++;
    } else if(arr[j] == 1){
        j++;
    } else {
        swap(arr, j, k);
        k--;
    }
}
```

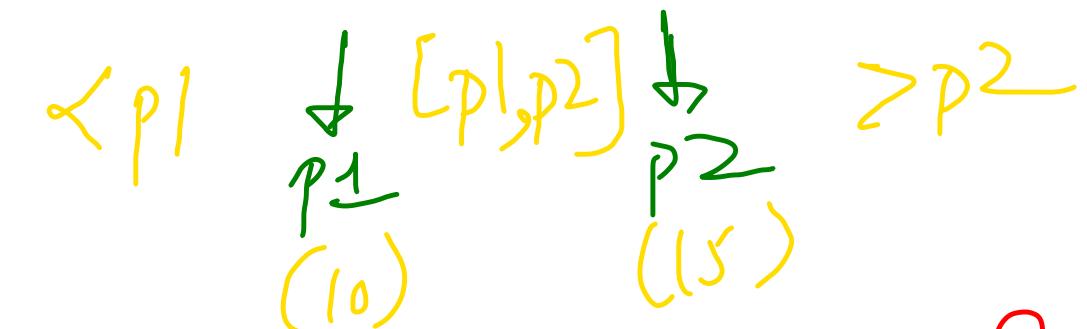
## Variations

① Sort OS, ls, 2g

② Sort R, G, B

③ Sort  $\%3 = 0, \%3 = 1, \%3 = 2$

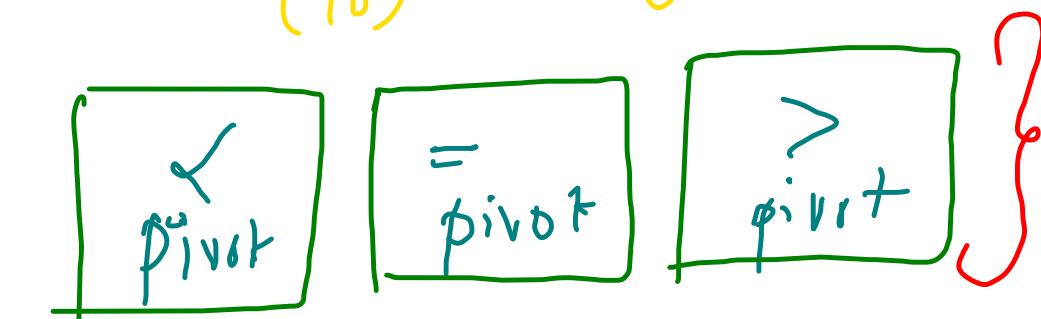
④ Dual Pivot Partitioning



GFG  
⑤ Three way Partitioning

Quick Sort

↳ Duplicate Elements



Quick Sort  
on Duplicate Ele  
(eg1)

5 4 2 3 4 2 3 4 1 5 2 1 3

(2 2 1 2 1) (3 3 3) (5 4 4 4 5)  
 $\leq 3$        $= 3$        $\geq 3$   
Already  
Sort-

(eg2)

10 10 10 10 10 10 10 10 10

{10} {10 ... 10} {10}

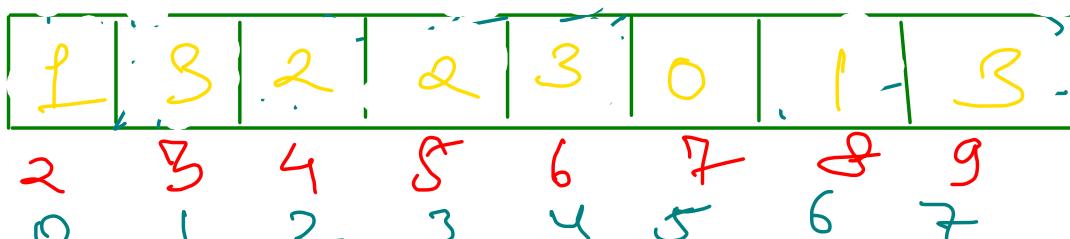
O(N)  
All elements are equal

Count Sort → Range of Elements in Array is fixed & small  
→ extra space (frequency Array)

$$\frac{9}{0}, \frac{6}{1}, \frac{3}{2}, \frac{5}{3}, \frac{3}{4}, \frac{4}{5}, \frac{3}{6}, \frac{9}{7}, \frac{6}{8}, \frac{4}{9}, \frac{6}{10}, \frac{5}{11}, \frac{8}{12}, \frac{2}{13}, \frac{9}{14}$$

① First traversal  $O(n)$   
→ find minimum ele & find max ele  
(2) (9)

② Second traversal  
→ Frequency Assay



③ 3<sup>rd</sup> traversal  
2 3 3 3 4 4 5 5 6 6 6 8 9 9 9

$$(k + 3k + 2k + 2k + 3k + 0k) + 1k \leq 3k = k + n \Rightarrow \boxed{O(n)}$$

```
int[] freq = new int[max - min + 1];
for(int i=0; i<arr.length; i++){
    int idx = arr[i] - min;
    freq[idx]++;
}

int count = 0;
for(int i=0; i<freq.length; i++){
    for(int j=0; j<freq[i]; j++){
        int val = i + min;
        arr[count] = val;
        count++;
    }
}
```

main()  $\uparrow$   $\uparrow$  Fill Seq  
 Array  
 $O(N) + O(N)$   
 $+ O(n)$   
 $\downarrow$   
 Sort Array  
 using  
 Seq  
 Array

Searching & sorting → lecture 5

Count Sort → Stable

Radix Sort

Just Dates

Broken Economy

First & last Index

Pivot for Rotated Sorted Array

Binary Search

# Count Sort - with stability

9' 6' 3' 5' 3'' 4' 3''' 9' 6'' 4' 6'' 5'' 8 2 9''

- ① First traversal  $O(n)$   
→ find minimum ele & find max ele  
(2) (9)

- ② Second traversal  
→ Frequency Array

1	3	2	2	3	0	1	3
2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

- ③ Prefix sum Array {3rd traversal}

1	4	6	8	11	11	12	15
2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

- ④ 4th traversal {Right to left}

9' 6' 3' 5' 3'' 4' 4'' 5' 5'' 6' 6'' 6'' 8 2 9'' right to left

2 3' 3'' 3''' 4' 4'' 5' 5'' 6' 6'' 6'' 8 9' 9'' 9'''

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

$O(n^2)$  DP  $O(n)$

$$PS(2) = freq(2)$$

$$PS(3) = freq(2) + freq(3) = PS(2) + f(3)$$

$$PS(4) = f(2) + f(3) + f(4) = PS(3) + f(4)$$

$$PS(5) = f(2) + f(3) + f(4) + f(5) = PS(4) + f(5)$$

$$PS(9) = f(2) + f(3) + \dots + f(9) = PS(8) + f(9)$$

$$PS(i) = PS(i-1) + f(i)$$

$PS(i)$  = last index of  $i$  in sorted array (1-based)

4.1  $res[val - min] = val;$

4.2  $res[PS[val - min]] = val;$

### 1. Linked List, Stack & Queue Level 1 + 2 (Single Choice)

85/85 (100%) answered

Yes (73/85) 86%

No (12/85) 14%

### 1. Backlogs (Single Choice) \*

86/86 (100%) answered

No Backlog (30/86) 35%

From Fundamentals (Getting Started & Patterns) (4/86) 5%

1D Arrays, 2D Arrays & Strings (8/86) 9%

Recursion & Backtracking (27/86) 31%

Only Few Lectures of S&S (17/86) 20%

level ①  
GFG + Lectures  
{200} {100}



```

public static void countSort(int[] arr, int min, int max) {
    int[] freq = new int[max - min + 1];
    for(int i=0; i<arr.length; i++){
        int idx = arr[i] - min;
        freq[idx]++;
    }

    int[] prefSum = new int[max - min + 1]; → O(n)
    prefSum[0] = freq[0];
    for(int i=1; i<freq.length; i++)
        prefSum[i] = prefSum[i - 1] + freq[i]; → O(n)

    int[] res = new int[arr.length]; → loss of info
    for(int i=arr.length-1; i>=0; i--){
        prefSum[arr[i] - min]--;
        int idx = prefSum[arr[i] - min];
        res[idx] = arr[i];
    }

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

```

To array contiguous allocation  
 $O(S^2 n) = O(n)$  time  
 man. min + 1  
 ↗ freq ↗ prefSum  
 Extra space →  $O(\text{Range})$   
 Input / Output Space →  $O(N)$   
 ↗ Arr ↗ Res

## Radix Sort

✓ 213  
 ✓ 97  
 ✓ 718 (1)  
 ✓ 123 →  
 ✓ 37 stable  
 ✓ 443  
 ✓ 982  
 ✓ 64  
 ✓ 375  
 ✓ 683

HT.O  
good good fast

(Count Sort)

✓ 982

✓ 213

✓ 123

✓ 443

✓ 683

✓ 64

✓ 375

✓ 97

✓ 37

✓ 718

(10) tens  
stable

{ No restriction of Range }

✓ 213

✓ 718

✓ 123 (100)

✓ 037 hundred

✓ 443 →

✓ 064 stable

✓ 375

✓ 982

✓ 683

✓ 097

(arr[ ] / end) % 10

037

064

097

123

213

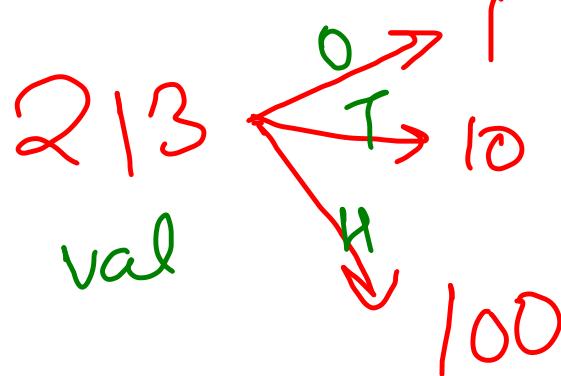
375

443

683

718

982



$$\frac{213}{10} = 21 \text{ R } 3$$

$$\frac{213}{10} = 21 \text{ R } 3$$

$$\frac{213}{100} = 2 \text{ R } 13$$

```
public static void radixSort(int[] arr) {
    int max = 0;
    for(int val: arr) max = Math.max(max, val);

    int digits = (int)Math.log10(max) + 1;
    int maxPlaceValue = (int)Math.pow(10, digits - 1);

    for(int i=1; i<=maxPlaceValue; i*=10){
        countSort(arr, i);
    }
}
```

$$O(N * (\log_{10} \max))$$

(val/exp) % 10

```
public static void countSort(int[] arr, int exp) {
    System.out.print("After sorting on " + exp + " place -> ");

    int[] freq = new int[10];
    for(int i=0; i<arr.length; i++){
        int idx = (arr[i] / exp) % 10;
        freq[idx]++;
    }

    int[] prefSum = new int[10];
    prefSum[0] = freq[0];
    for(int i=1; i<freq.length; i++)
        prefSum[i] = prefSum[i - 1] + freq[i];

    int[] res = new int[arr.length];
    for(int i=arr.length-1; i>=0; i--){
        prefSum[(arr[i] / exp) % 10]--;
        int idx = prefSum[(arr[i] / exp) % 10];
        res[idx] = arr[i];
    }

    for(int i=0; i<arr.length; i++)
        arr[i] = res[i];

    print(arr);
}
```

Number of digits

$$1 \rightarrow 1$$

$$10 \rightarrow 2$$

$$100 \rightarrow 3$$

$$1000 \rightarrow 4$$

$$10000 \rightarrow 5$$

$$10^5 \rightarrow 6$$

$$10^6 \rightarrow 7$$

$$V \rightarrow \log_{10} V + 1$$

Radix Sort

$$\left\{ \text{Count Sort} * \text{No of digits} \right\}$$

$\downarrow$                        $\downarrow$   
 $N$                        $\log_{10} \text{man}$

$$= O(N * \log_{10} \text{man})$$

$$\text{Integer man} \rightarrow 2^{31} - 1 \approx 10^9$$

$$= O(N * 10) = O(N)$$

# Sort Dates of Homework - HW }

✓ 12041996  
✓ 20101996  
✓ 05061997  
✓ 12041989  
✓ 11081987

DD MM YYYY

1<sup>st</sup> 2<sup>nd</sup> 3<sup>rd</sup>

→ 05 06 1997      12 04 1996      11 08 1987  
→ 21 08 1987      12 04 1989      05 06 1997  
→ 12 04 1996      20 10 1996      05 06 1987  
→ 12 04 1989      20 10 1996      12 04 1996  
→ 11 08 1987      20 10 1996      05 06 1987

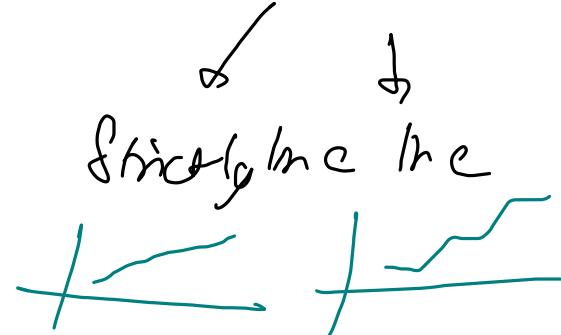
Month → Year

## # Binary Search

→ Array is sorted

↓  
Monotonic function

Monotonic Inc



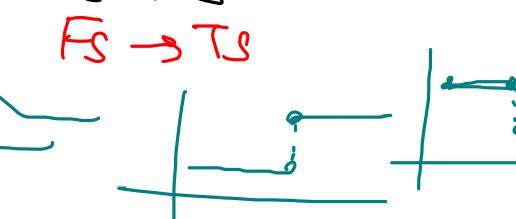
Monotonic Dec

Strictly Dec Dec



Binary f

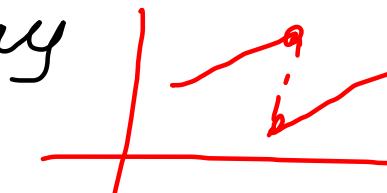
IS - OS  
OS → IS  
FS → TS



BS on Answer

## A Variations

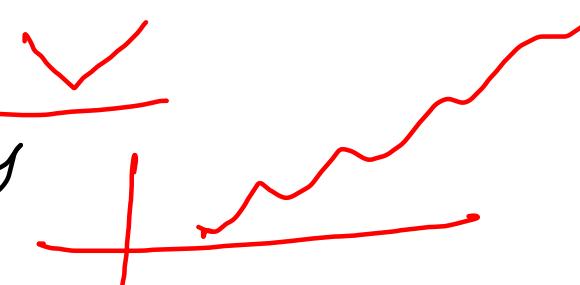
→ Rotated Sorted Array



→ Bitonic Array



→ Infinite / Unbounded sorted Arrays



→ Nearly Sorted Array { 2 1 3 5 4 }

{ 2 3 4 5 }

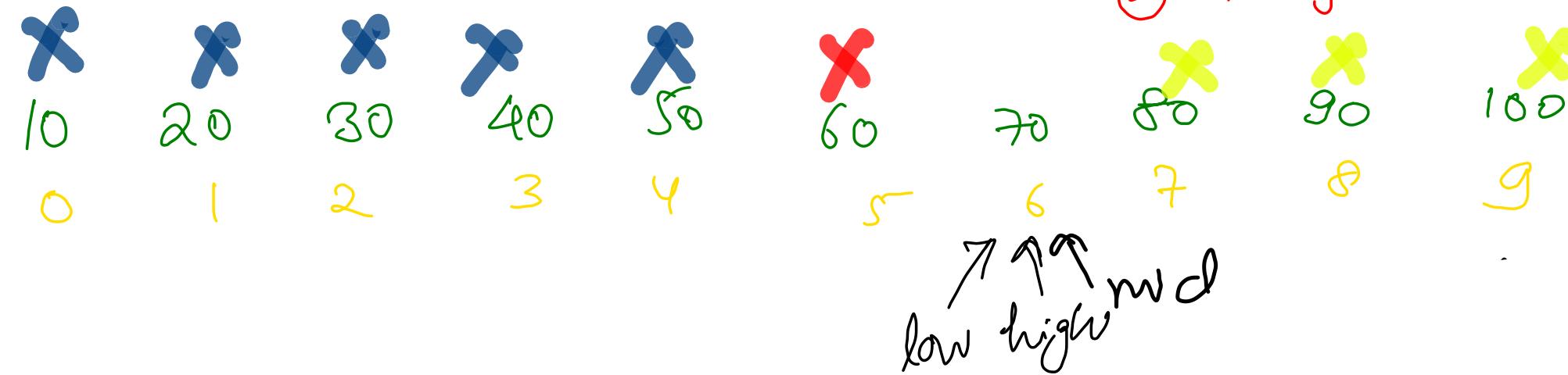
linear search  
{order → unsorted}  
 $O(N)$  worst case

4 3 5 1 6 2

target = 2

## Binary Search {Divide & conquer}

sorted



① low = 0, high = 9  
mid = 9  
 $\textcircled{50} < \textcircled{70}$   
reject left range  $\Rightarrow$  low = mid + 1

② low = 5, high = 9  
mid = 7  
 $\textcircled{80} > \textcircled{70}$   
reject right range  
 $\Rightarrow$  high = mid - 1

③ low = 5, high = 6  
mid = 5  
 $\textcircled{60} < \textcircled{70}$   
left reject  
 $\Rightarrow$  low = mid + 1

④ low = 6, high = 6  
mid = 6  
 $\textcircled{70} = \textcircled{70}$   
return mid

worst case?  $\rightarrow$  if half of search space is rejected

$$T(n) = T(n/2) + O(1) \Rightarrow O(\log_2 N)$$

- ① target =  $\textcircled{70} \rightarrow 6$
- ② target =  $35 \rightarrow -1$

best case  
 $O(n)$

Avg/worst case  
 $O(\log n)$



$$l=0, r=9$$

$$m=4$$

$$\textcircled{50} > \textcircled{35}$$

$$l=0, r=3$$

$$m=1$$

$$\textcircled{20} < \textcircled{35}$$

$$l=2, r=3$$

$$m=2$$

$$\textcircled{30} < \textcircled{35}$$

```

public int search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;

    while(left <= right){
        int mid = (left + right) / 2;

        if(nums[mid] == target){
            return mid; // search successful
        } else if(nums[mid] < target){
            left = mid + 1;
        } else {
            // nums[mid] > target
            right = mid - 1;
        }
    }

    return -1; // unsuccessful element
}

```

