Searching & Sorting

Lecture (1) & (2)

- → Bubble Sort
- → Selection Sort
- → Insertion Sort
- → Merge 2 Sorted Arrays
- → Merge Sort
- → Target Sum Pair

$O(N^2)$

Lecture (3)

- → Partition Array
- → Quick Sort
- → Sort 01
- → Sort 012
- → QuickSelect

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	Worst Case
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

duplicates

Sorting Algorithm Name	Time Complexity			Space Complexity	Is Stable?	Sorting Class Type	Remarks
	Best Case	Average Case	Worst 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.

Inplace ⇒ without Extra space

Bubble Sort

Best case → already sorted → O(N)

→ adjacent swaps

→ heavy etc

```
  0 1 2 3 4
4.0 5 9 8 2 1
4.1 5 9 8 2 1
4.2 5 8 9 2 1
4.3 5 8 2 9 1
    5 8 2 1 9

3.0 5 8 2 1 9
3.1 5 8 2 1 9
3.2 5 2 8 1 9
    5 2 1 8 9

2.0 5 2 1 8 9
2.1 2 5 1 8 9
    2 1 5 8 9

1.0 2 1 5 8 9
    1 2 5 8 9
```

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

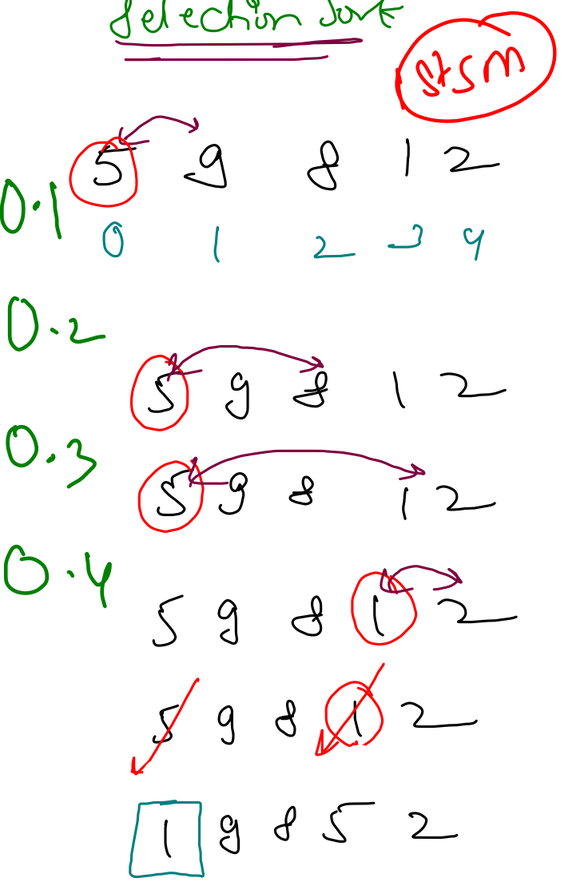
$$= \frac{(n-1)*(n)}{2}$$

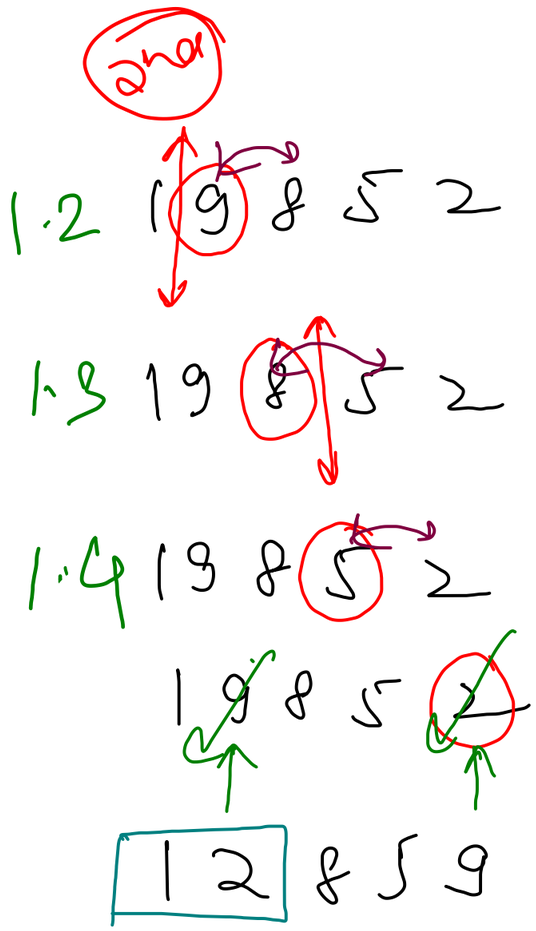
$$= \frac{n^2}{2} - \frac{n}{2}$$

$O(N^2)$

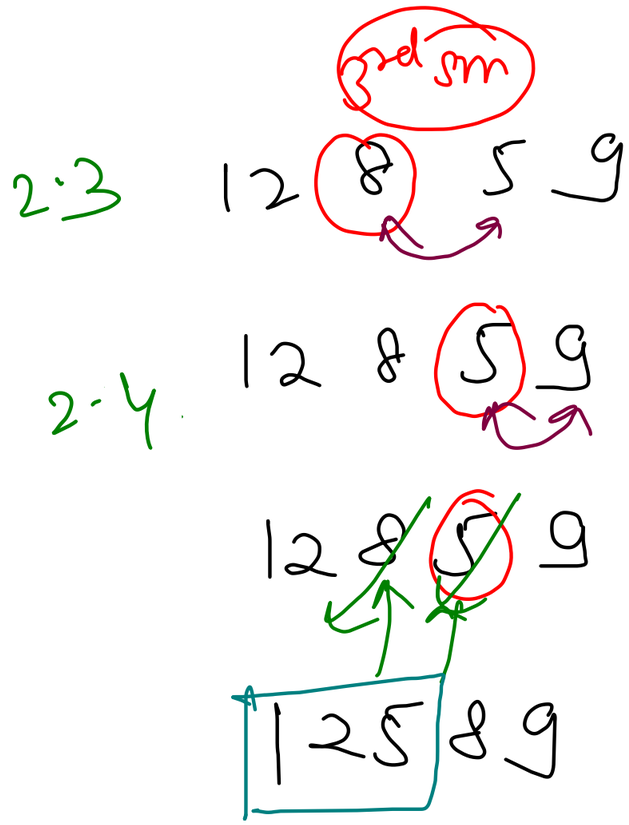
```java
public static void bubbleSort(int[] arr) {
    for(int i=arr.length-1; i>0; i--){
        for(int j=0; j<i; j++){
            if(isSmaller(arr, j + 1, j)){
                swap(arr, j + 1, j);
            }
        }
    }
}
```

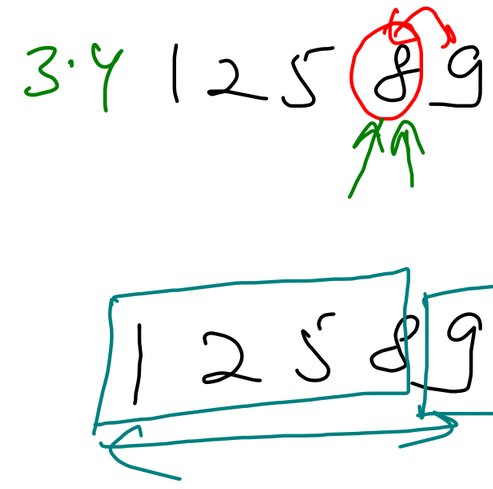
Selection Sort





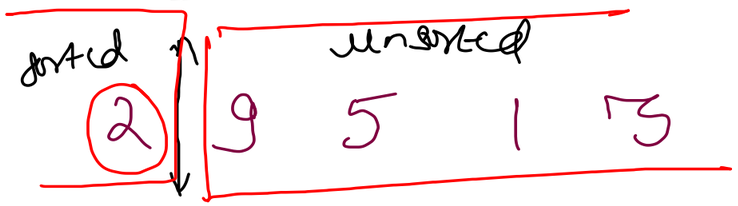
O(n2)






```java
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



1.1) 2 9 5 1 3

2.2) 2 9 5 1 3

2.1) 2 5 9 1 3

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

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(N²)

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}

(M) A: 2 3 6 7 9 10

(N) B: 1 4 5 7 11 12 13 14

6+8 = 14

(M+N) C:

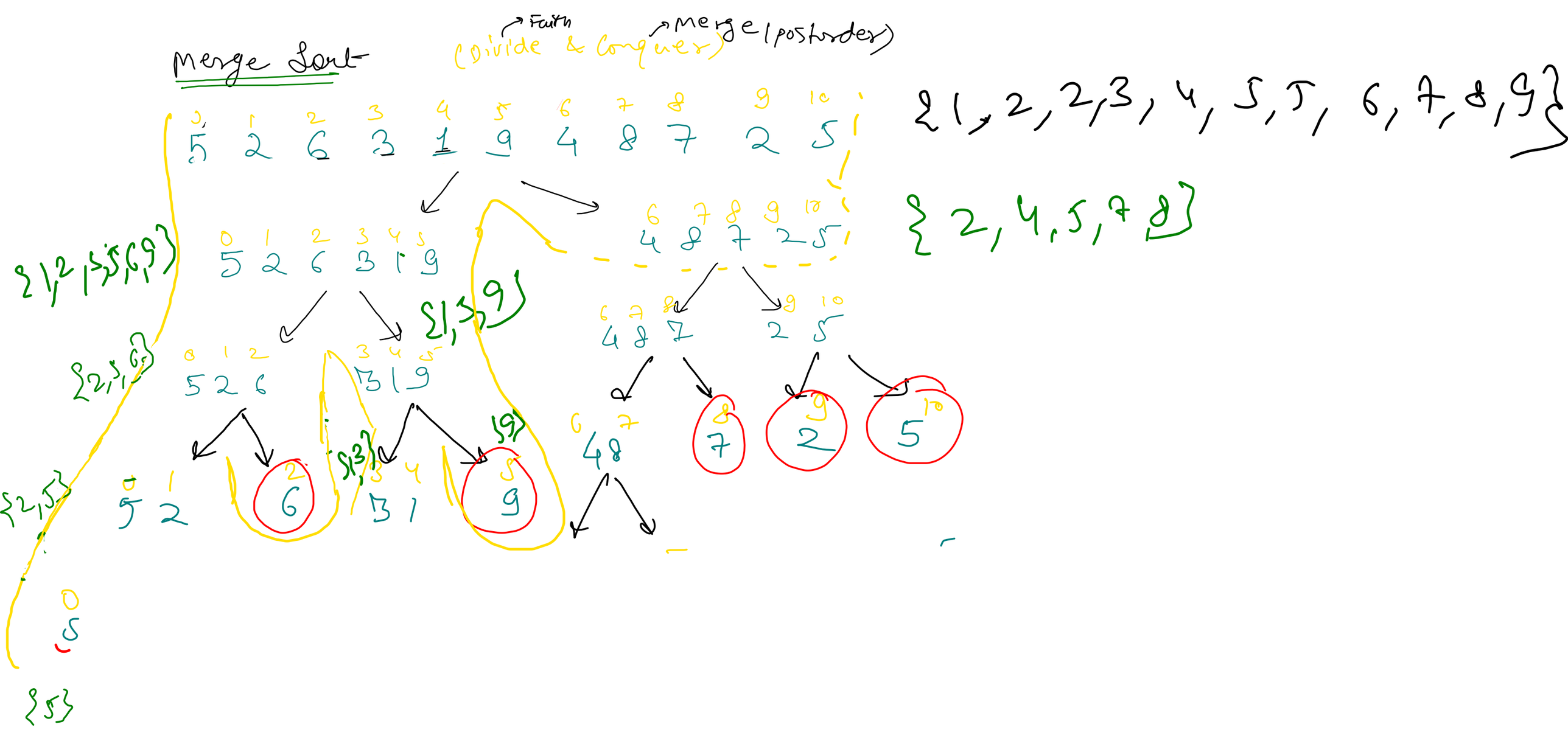
1	2	3	4	5	6	7	7	9	10	11	12	13	14

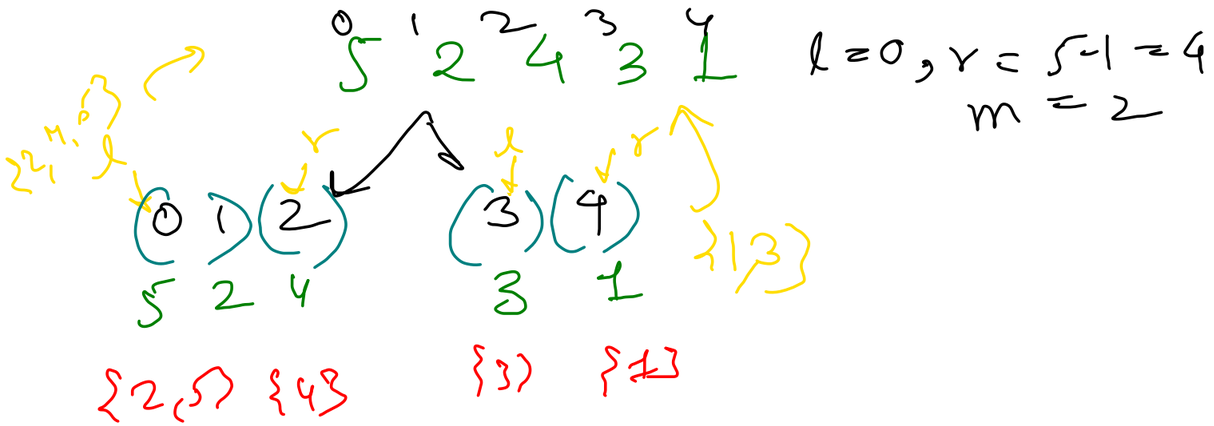
```java
//write your code here
int[] c = new int[a.length + b.length];

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) → Faith → merge (postorder)






```java
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;

}
```

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

Fast Divide Merge (Conquer)

(calls) height + (pre + post) * height

mid merge

$2^{\log_2 n} + \{k + n\} * \log_2 n$

$n + n \log_2 n$

$O(n \log_2 n)$

Best Avg Worst case

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

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

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

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

$\vdots$

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

$T(n) =$

$2^0 * n + 2^1 * n/2$

$+ 2^2 * \frac{n}{4} + 2^3 * \frac{n}{8}$

$+ \cdots 2^x * \frac{N}{2^x}$

$+ 2^x * T(0)$

$= n * x + 2^x k$

$= n \log_2 n + 2^{\log_2 n} k$

$= O(n \log_2 n) + n$

Space Comp.

① Extra space / Auxiliary space :→ O(N) {at merging}

② Input/ output :→ O(N)

③ Recursion call stack :→ O($\log_2 N$)

Target Sum Pair (Array is Sorted)

↳ two pointer

target = 25

0	1	2	3	4
10	12	13	14	18

left, right

10 + 18 = 28 > 25

→ nums[left] + nums[right] > target

⇒ right --

10 + 14 = 24 < 25

→ nums(left) + n[right] < target

⇒ left++

12 + 14 = 26 > 25

12 + 13 = 25 = 25

→ n[l] + n[r] == target

⇒ return that pair.

(Nested loop)

① Brute force

Time :→ O(N²)

Space :→ O(1)

167

0	1	2	3
10	13	14	18

l r

target = 25

l == r → 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;
```

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

Java

Arrays.sort(nums)

↓

Tim Sort

↙ ↘

Merge Sort

N > 16

+

Insertion Sort

N < 16

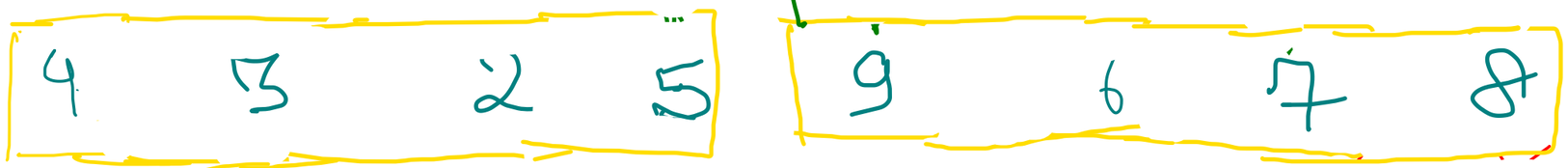
Small Array

{ Insertion Sort }

better

Searching & Sorting - lecture (3)

Partition An Array



(0-n)
unexplored
(0-l-1)
≤ pivot
> pivot
(l-r-1)

{ < pivot → left } { > pivot → right }

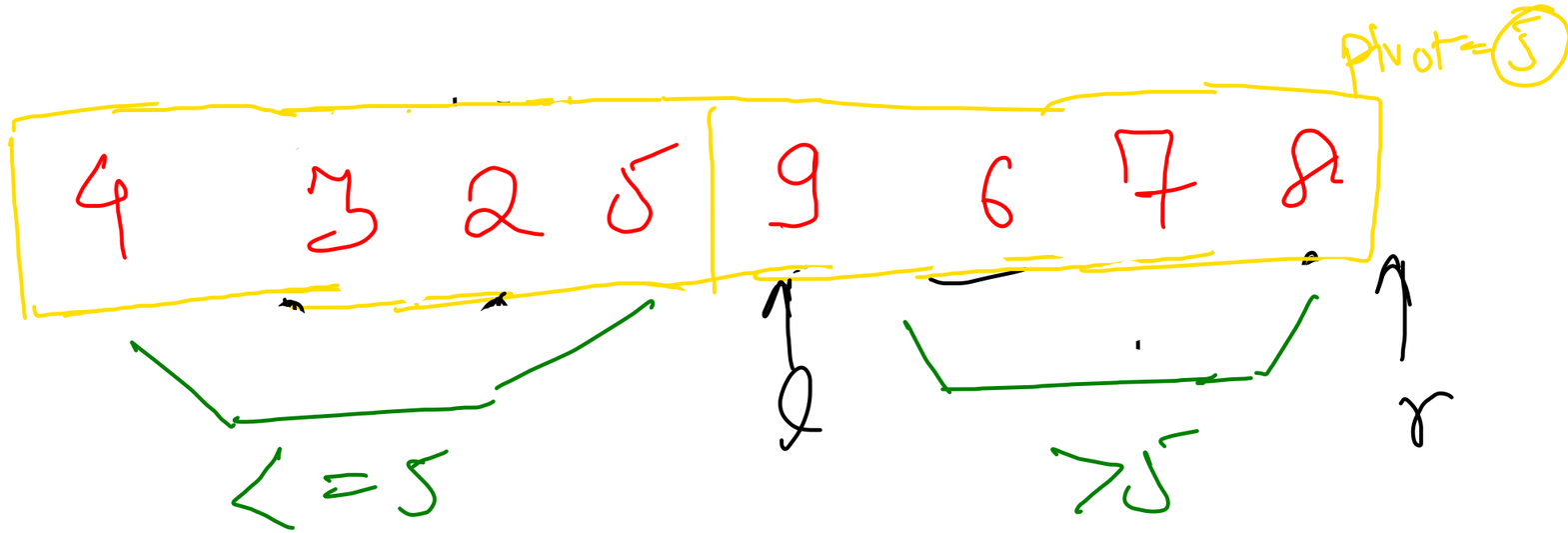
left → first ele of greater region

right → first ele of unexplored region

left = 0, right = 0;

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

```
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) unexplored (r - n)
(2) 0's (0 - l-1)
(3) 1's (l - r-1)

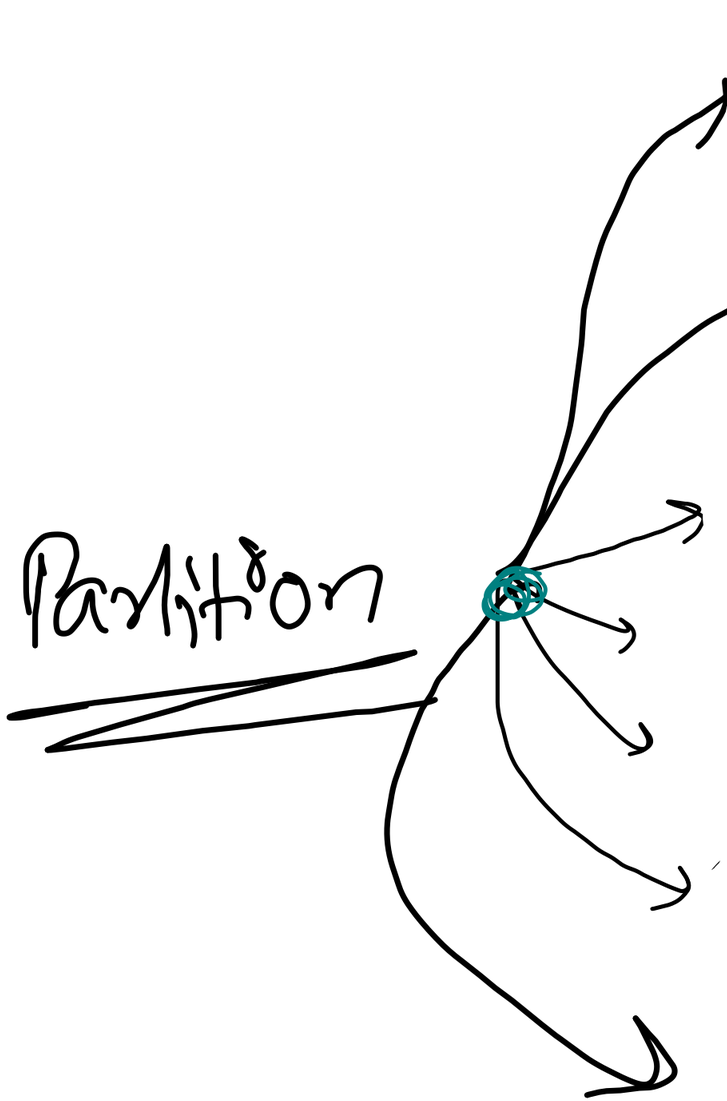
(1) Two traversals

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



$< x$, $> x$

$\leq x$, $\geq x$

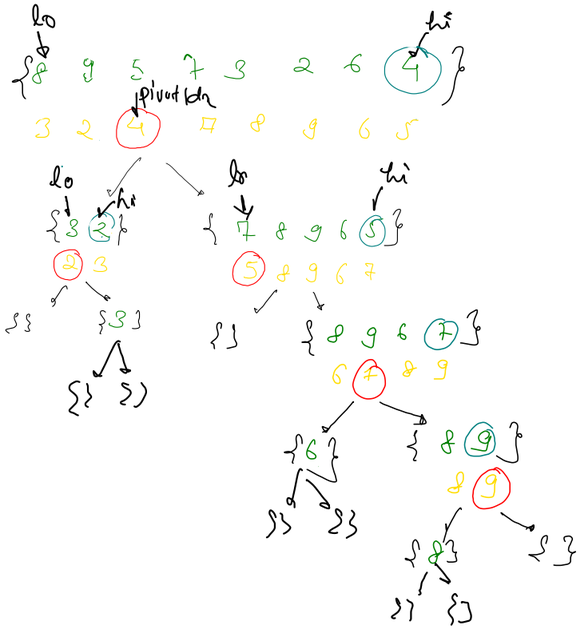
0's 1's

R's B's

-ve's , +ve's

0's , Non zeros {Move zeros to left/right

Odd's & even's



Quick Sort → Partition

Expectation

```java
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);
    // partition around last element -> last ele is at current index

    quickSort(arr, lo, pivotIdx - 1);
    quickSort(arr, pivotIdx + 1, hi);
}
```

→ Meeting Exp (Conquer)

} Faith/Decrease

```java
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);
}
```

Decrease & Conquer

Space

Input/output → O(N)

Extra space → O(1)

Recursion call stack → worst case O(n) (depth of Recursion tree)

Avg case O(h) = O(log n)

Partition → In Recorder

Partition : Subarray [l-r]

pivot : why it should be arr[right]?

→ Partitioning on pivot will result in pivot element to be on it's 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

5 3 8 2 6 9 8 7

{ Partition not around last index }

4 9

(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

Avg case/ best case

$T(n) = 2T(n/2) + O(n)$ → $\Theta(n \log n)$ → same as merge sort

Partition

Worst case

$T(n) = T(n-1) + O(n)$ → $O(n^2)$

Partition

* corner cases
  - Already sorted (inc/dec)

Randomized Quicksort

(left to right) raye, pick any random index, last swap, partition

Probability of hitting the worst case

2 1 3 5 7 6 8 4 ⇒ Nearly / Almost Sorted

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

↓

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

↓

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

↓

1 2 3 4 5 6 (8 7)

↓

1 2 3 4 5 6 7 8

Sorted

Avg $\Theta(n \log n)$

Worstcase $O(n^2)$

Merge Sort

Best
Avg
Worst
} O(nlogn) Time

Extra space : → O(n)

Inplace ✗ { Merge 2 Sorted Array }

Stability ✓

MS is preferred for non sequential (LL)

Quick Sort

Best, Avg ⇒ Θ(nlogn)

Worst case ⇒ O(n²)

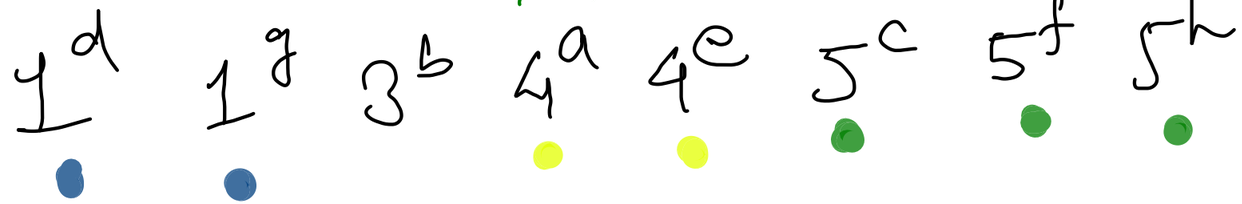
Extra space : → O(1) ← Inplace ✓

Stable ✗

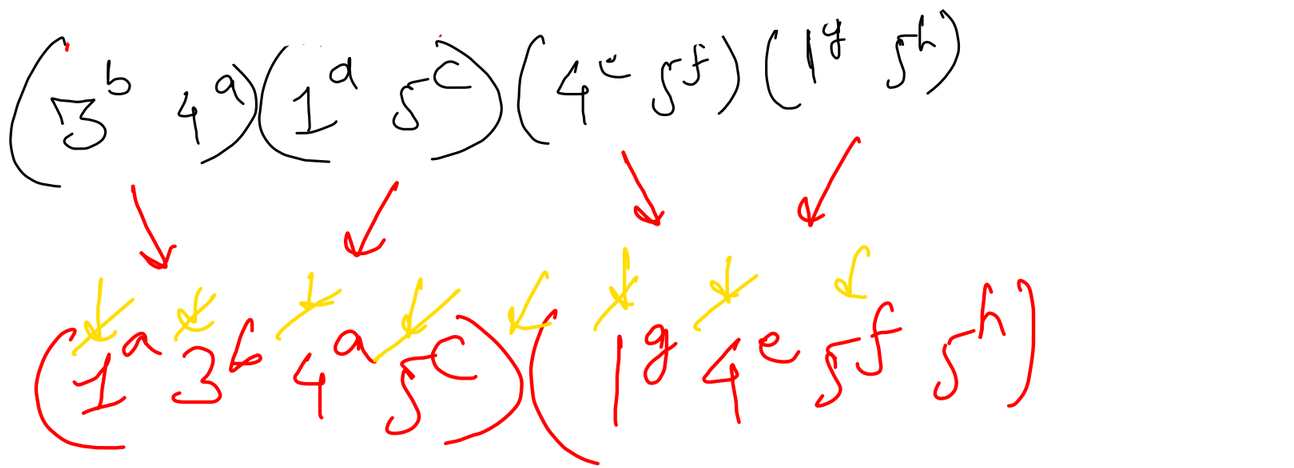
QS is preferred for Arrays

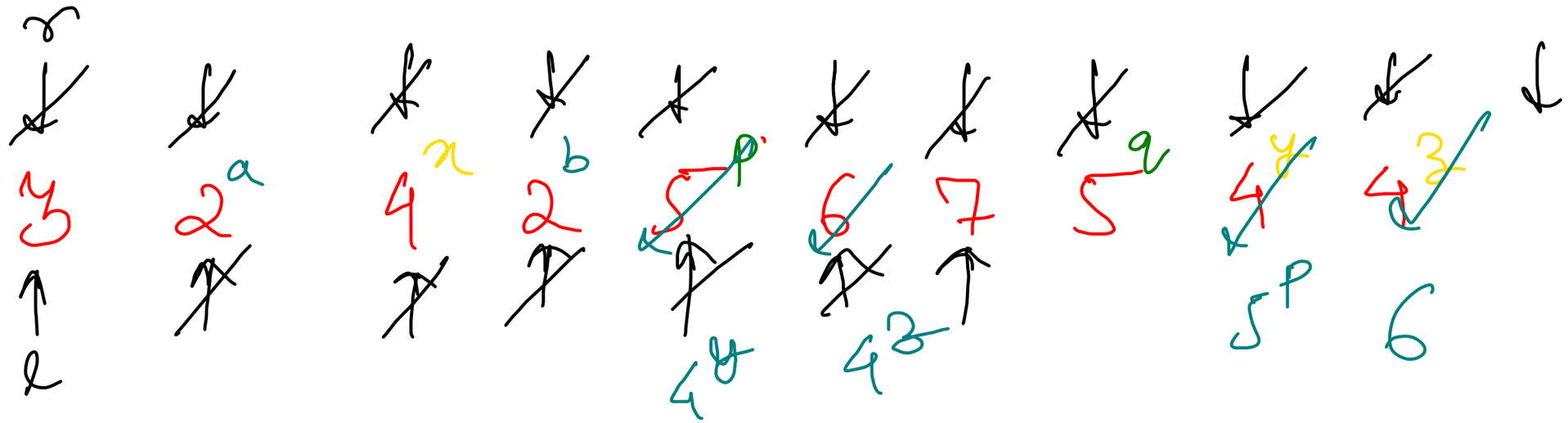
Stability ⟹ Input → Duplicate Elements ⟹ Same order of Output as input

If algo is stable



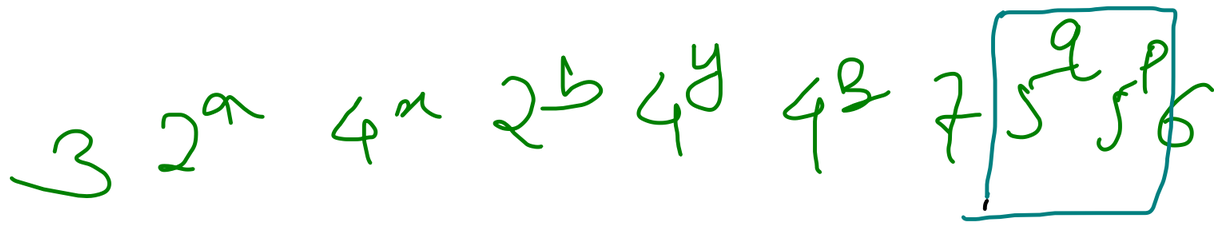

```
int[] c = new int[a.length + b.length];

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;
```





```java
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



"Quick Sort is not stable due to stability"

Quick Select { Kth smallest } → Binary Search / Divide & Conquer

→ Partition

pivot

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

```java
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);
}
```

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

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

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

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

$\vdots$

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

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

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

$x = \log_2 N$

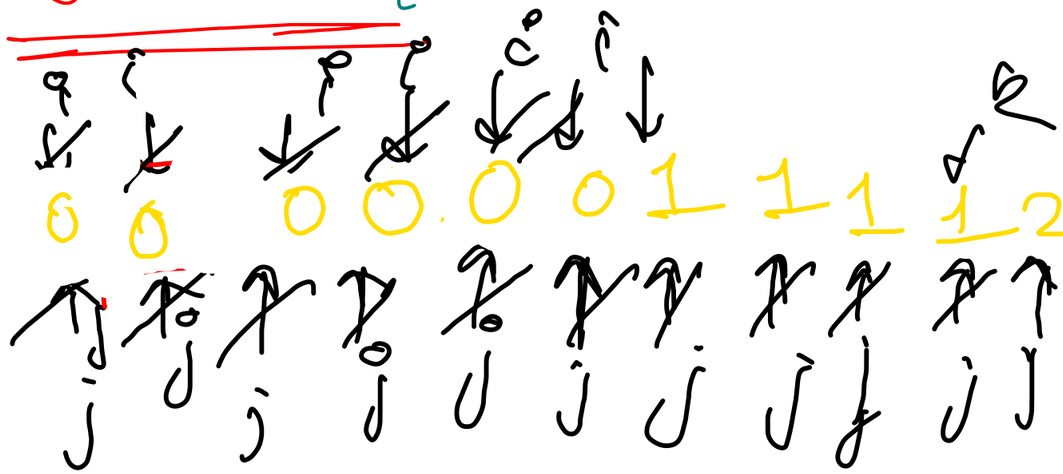
$O(N)$

$$\frac{a(1-r^n)}{(1-r)} = \frac{1 * \left(1 - \left(\frac{1}{2}\right)^{\log_2 N}\right)}{1 - \frac{1}{2}}$$

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

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

Sort 012 { Dutch National Flag Algorithm } { DNF Sort }



Unexplored {j-k}

0's {0-i-1}

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

2's { k+1 - n }

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

$O(n)$

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 0s, 1s, 2s
2. Sort R, G, B
3. Sort %3==0, %3==1, %3==2
4. Dual Pivot Partitioning

5. Three way Partitioning

GFG

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)

<3 =3 >3

Already

Sort-

(eg2) 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)

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

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

② Second traverseed $O(n)$
→ Frequency Assay

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 3rd traversal

"2 3 3 3 4 4 5 5 6 6 8 9 9 9"

$1k + 3k + 2k + 2k + 3k + 0k + 1k + 3k = k * n \Rightarrow O(n)$

$\{l, r\}$

$[l, r] \rightarrow r - l + 1$

$(l, r] \text{ or } [l, r) \rightarrow r - l$

$(l, r) \rightarrow r - l - 1$

Val - min = idx

```java
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++;
    }
}
```

$O(n)$ $O(n)$

max/min Fill Freq Array

$O(N) + O(N) + O(N)$

↓

Sort Array using Freq Array

Searching & Sorting → lecture (5)

Count Sort → Stable

Radix Sort

Sort Dates

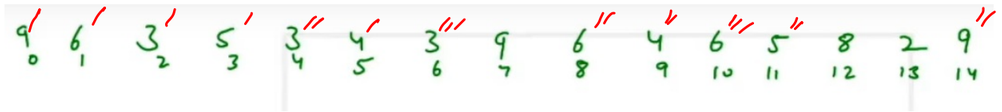
Broken Economy

First & Last Index

Pivot in Rotated Sorted Array

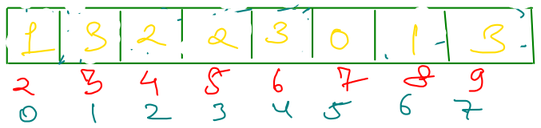
→ Binary Search

Count Sort - with stability

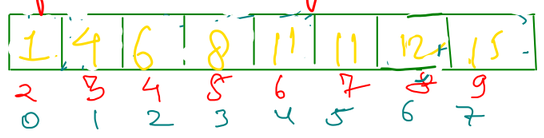


① First traversal O(n)
→ find minimum ele & find max ele
(2) (9)

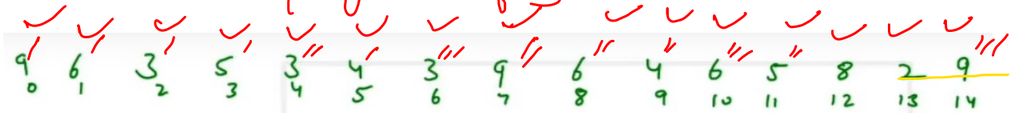
② Second traverseel
→ Frequency Array



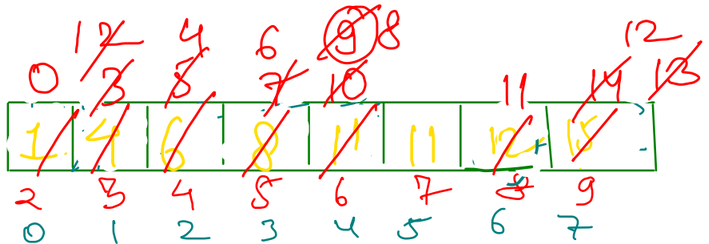
③ Prefix sum Array {3rd traversal}

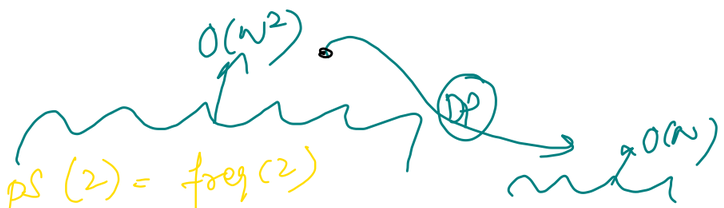


④ 4th traversal {Right to left}



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





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) + ... + f(9) = ps(8) + f(9)

ps(i) = ps(i-1) + f(i)

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

right to left

4.1 ps[val - min]--;

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 + Leetcode

{200} {100}

```java
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];
    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] - min]--;
        int idx = prefSum[arr[i] - min];
        res[idx] = arr[i];
    }

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

1
2
3
4
5

Stability

avoid loss of info

Array → contiguous allocation

$O(5n) = O(n)$ time

Extra Space → freq, prefSum → max-min+1 → O(Range)

Input / Output Space → O(N)

Arr

Res

Radix Sort (: Count Sort) { No restriction of Range }

(arr[i]/exp)%10

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

H T O
3rd 2nd 1st

(1) ones → Stable

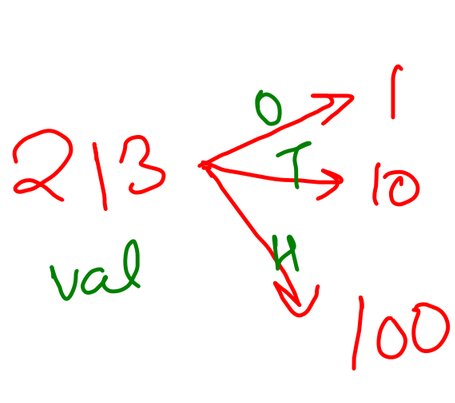
9 8 2
2 1 3
1 2 3
4 4 3
6 8 3
6 4
3 7 5
9 7
3 7
7 1 8

(10) tens → Stable

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

(100) hundred → Stable

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



val/exp

213/1 = 213 % 10 = 3 (val/exp) % 10

213/10 = 21 % 10 = 1

213/100 = 2 % 10 = 2

```
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 * log10 max)

```
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 → 1

10 → 2

100 → 3

1000 → 4

10000 → 5

10^5 → 6

10^6 → 7

V → $\log_{10} V$ + 1

Radix Sort

{ Count Sort * No of digits }

↓ N

↓ $\log_{10}^{max}$

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

Integer max → $2^{31} - 1 \approx 10^9$

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

Sort Dates {Homework – HW}

12041996
20101996
05061997
12041989
11081987

DD MM YYYY
1st 2nd 3rd

pass →

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

Months →

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

Year →

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

Binary Search

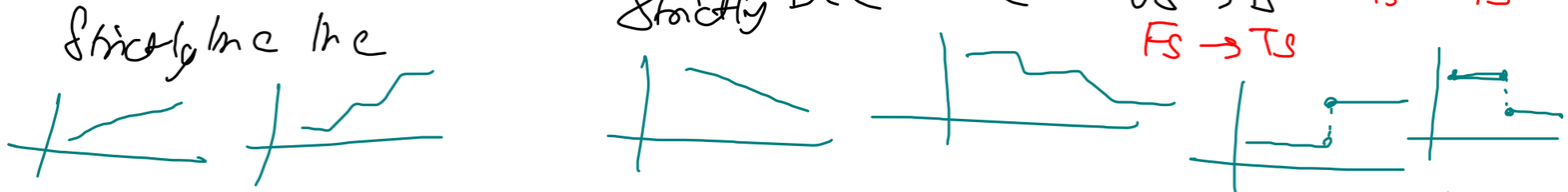
linear search
{order → unsorted}
O(N) worst case

4 3 5 1 6 2
target = 2

→ Array is Sorted

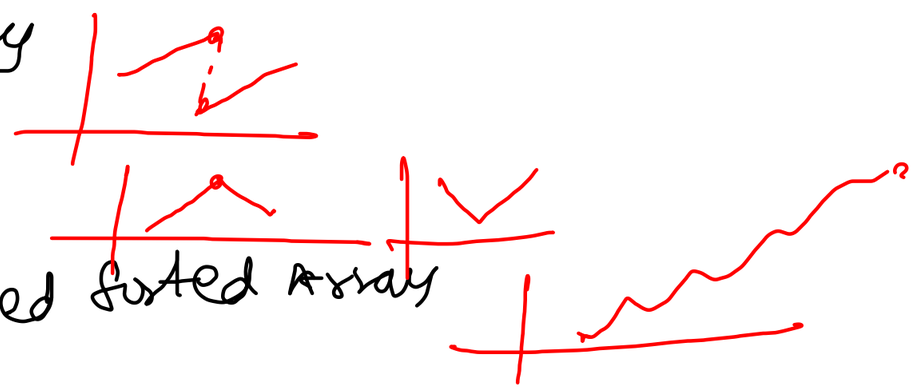
→ Monotonic function

- Monotonic Inc
  - Strictly Inc
  - Inc
- Monotonic Dec
  - Strictly Dec
  - Dec
- Binary
  - 0s → 1s
    Fs → Ts
  - 1s — 0s
    Ts — Fs



variations

- Rotated Sorted Array
- Bitonic Array
- Infinite/ Unbounded Sorted Array
- Nearly Sorted Array { 2 1 3 5 4 }
  (2 3 4 5)



Binary Search {Divide & Conquer}

Sorted

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

low high mid

① target = 70 → 6

② target = 35 → -1

best case
O(1)

Avg/worst case
O(log N)

① low = 0, high = 9
mid = 4
(50) < (70)
reject left range ⇒ low = mid + 1

② low = 5 high = 9
mid = 7
(80) > (70)
reject right range
⇒ high = mid - 1

③ low = 5 high = 6
mid = 5
(60) < (70)
left reject
⇒ low = mid + 1

④ low = 6, high = 6
mid = 6
70 = 70
return mid

worst case → if half of search space is rejected

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

l=0, r=9

m=4

55 > 35

l=0, r=3

m=1

20 < 35

l=2, r=3

m=2

30 < 35

l=3 r=3

m=3

40 > 35

```java
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; // unsuccesful element
}
```