TIME & SPACE Complexity

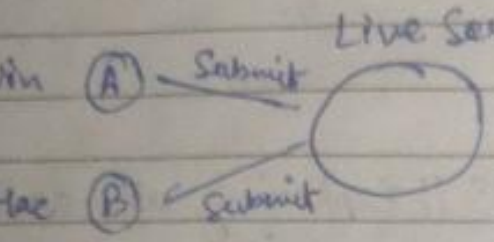
Time Complexity

Machine dependant

Runtime ↑

{100 ms, 250 ms, 1000 ms, 1 ms}

ARCHITECTURAL NEUTRALITY



Runtime depends on this
(∵ Runtime shows same on all machines & runs on a server.)

⇒ If Live Server is slow → Runtime is slow

⇒ we look into Asymptotic Analysis

Algorithm ∝ input size (n)

ignore all other factors like server, machine, etc.

↑

Makes Asymptotic Analysis Machine Independant

T(N) ⇒ function (I/p size)

↑

Growth of runtime w.r.t i/p size

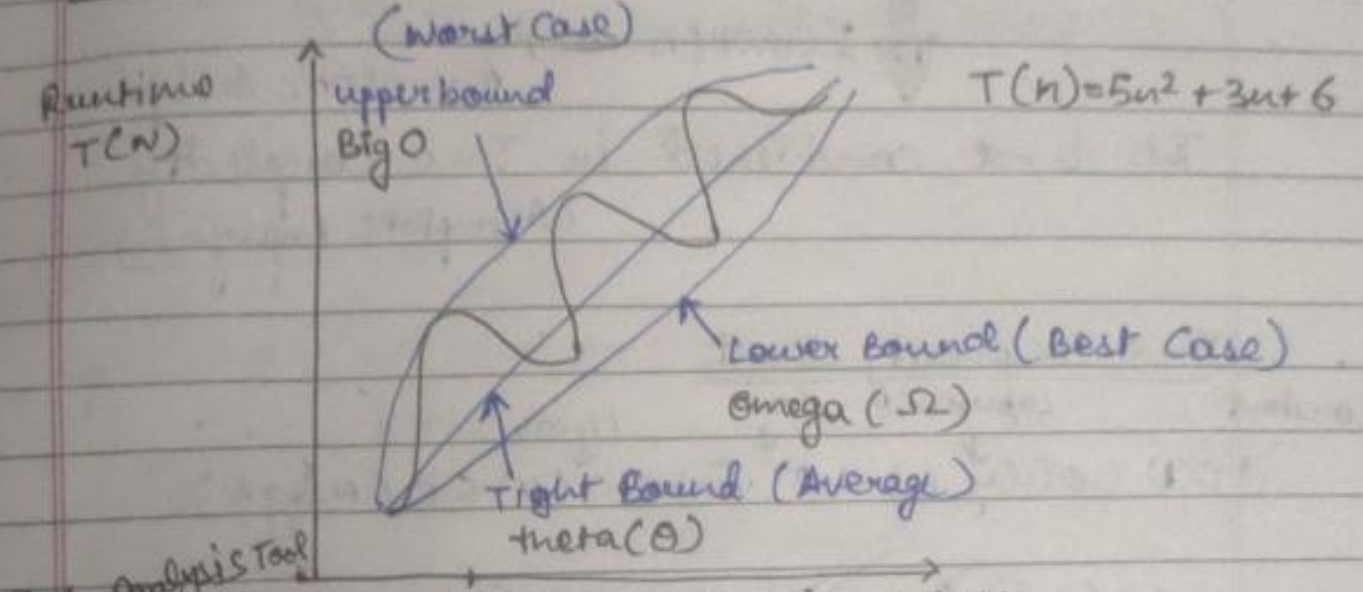
2 machines take diff. times to run an algo.

(A)	(B)	Growth
N = 100 elements	N = 100 elems	A ⇒ N = T
Time = 100 ms	Time = 200 ms	
		B = N = 2T
N = 1000 elems	N = 1000 elems	
Time = 1000 ms	N = 2000 ms	∴ N ∝ T

⇒ Growth of runtime in both A & B is linear which is same.

∴ The algo is linearly dependent on I/p size

GRAPH



Worst case

Closest Bigger value

$T(n) = 15n^2 \Rightarrow O(n^2)$

$15n^2 \geq 5n^2 + 3n - 16$ } always true

$n > 0$

When we take $10n^2$ then for $n=1$

$10n^2 < 5n^2 + 3n - 16$

This is Upper Bound Too

Best Case

Closest Smaller value

$T(n) = 4n^2 \Rightarrow \Omega(n^2)$

$4n^2 \leq 5n^2 + 3n - 16$ } always true

$n > 0$

$n^2 + 3n - 16 \geq 0$

Average

Avg. of Best & Worst

$T(n) = 8n^2 \Rightarrow \Theta(n^2)$

always depends on n^2

$N \log 2 = 3 \log N$

$\frac{N}{\log N} = \frac{3}{\log 2}$

Compilation & Runtime

Compile^n time { Linking + Loading }

↙ ↘

Pgkages Header Files

↓

IGNORED in Runtime & Asymptotic Analysis

This is not considered in Time Complexity (Asymptotic Analysis)

Time Complexity Chart

Constant Logarithmic Root Linear

$O(1) < O(\log_2 N) < O(\sqrt{N}) < O(N) < O(N \log N)$

$< O(N^2) < O(N^3) < \cdots < O(2^N) < O(3^N) < O(N!)$

↑ Quadratic ↑ Cubic or $O(N^N)$

exponential

constant < Logarithmic < Polynomial < Exponential

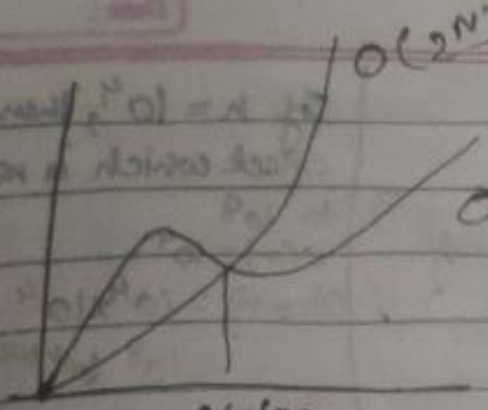
$O(N^3) < O(2^N)$

$n=2 \quad O(2^3) = O(8) \not< O(2^2) = O(4)$

$n=3 \quad O(3^3) = O(81) \not< O(2^3) = O(8)$

$n=4 \quad O(4^3) = O(64) \not< O(2^4) = O(16)$

This doesn't hold but still it is True Because don't consider small i/p in Time Complexity)



small value not considered

Google → Algo → millions of users (NT)

Time Complexity

```
for(i=1; i<n; i*=2){
    ... k ms
}
```

Let N=32

i	
1	k
2	k
4	k
8	k
16	k
32	k

GP = $k(1+2+4+\cdots n)$

$a=1 \quad r=2 \quad ar^{x-1} = n$

Put a & r to get

$x = \log_2 n + 1$

$O(\log_2 n)$

Competetive Programming

leetcode codeforces codechef

↓

1s ⇒ 10^8 operations (fixed)

If $n = 10^4$, then check which is nearest to 10^8

$O(n) = 10^4$

$O(n*n) = 10^4 \times 10^4 = 10^8$ ✓ works

$O(n^3) = 10^{12}$ ✗ won't work $\because 10^{12} > 10^8$

If $N = 10^8 \Rightarrow O(N) \rightarrow 10^8$ operations

↑

Only this algo will work for $N = 10^8$

* Algo. Chart

I/p Size (N)	Time Complexity (worst Case)	Hint of Algorithm
≤ 10	$O(n!)$ or $O(2^n)$	Backtracking {E.g. permute, subsets}
≤ 18	$O(2^n \cdot n)$	Travelling Salesman
≤ 22	$O(2^n \cdot n^2)$	DP with Bitmasking
$\leq 10^2$	$O(n^4)$	Quadruplets / 4 Nested loops; (4D DP)
$\leq 4 * 10^2$	$O(n^3)$	3 nested loops, Floyd Warshall; (3D DP)
$\leq 2 * 10^3$	$O(n^2 \log N)$	Nested loops & Binary Search
$\leq 10^4$	$O(n^2)$	2 Nested loops (Bubble, Insert, Select^n) (2D DP)
$\leq 10^5 - 10^6$	$O(N \log N)$	sorting (QS, MS), BS on Answer, Greedy, etc.
$\leq 10^8$	$O(N)$	Array & String, (1D DP)
$\leq 10^{18}$	$O(\log_2 N)$ or $O(1)$	Mathematical formula (Bit Manipulation)

* Recursion Generic Formula

worst case Time Complexity (TC)

$$TC = (\text{No of calls})^{\text{height}} + (\text{preorder} + \text{postorder}) * \text{height}$$

Total calls from 1 node

Display Arr

Base Order : 5

5 : 4

4 : 3

3 : 2

2 : 1

1 : idx = 0

5 (N) calls ↑ height

1 call from 1 node

Nothing in post order

$TC = (1)^N + \{k + 0\} * N$

Recursion (Printing)

$= 1 + kN$

$TC = O(N)$

Increasing Print

Base Case 0

1

2

3

4

5

Height (N) ↑

$TC = (1)^N + (0+k)N$

$= 1 + kN$

$TC = O(N)$

Print Decreasing

$TC = O(N)$

Print Inc - Dec

$TC = (1)^N + (k+k)N$

$= 1 + kN$

$TC = O(N)$

Power Linear

E.g. $2^6 = 2 \times 2^5$

$= 2 \times 2 \times 2^4$

$= 2 \times 2 \times 2 \times 2^3$

$= \frac{1}{2} \times 2 \times 2 \times 2 \times 2 \times 2$

Height = N

Calls = 1

$TC = (1)^N + (0k + k)N$

$= 1 + 2kN$

$\boxed{TC = O(N)}$

Imp

Power Logarithmic

P(2,6)

↓ 1

P(2,3) 1

↓ 2

P(2,1) 3

↓

P(2,0) Base case

Calls = 1 PreOrder = k PostOrder = k

Height = h

$TC = (1)^h + (k+k)h$

$= 1 + 2kh = 1 + 2\log_2 N$

Since, recursion is dividing by 2 everytime

$\therefore$ Height = $\log_2 N$

$\therefore \boxed{TC = O(\log_2 N)}$

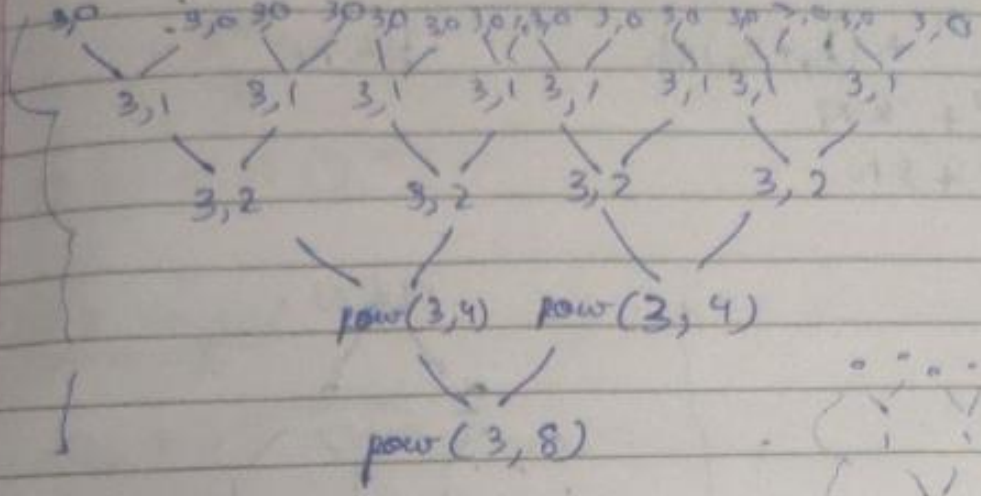
Call stack height

tree depth

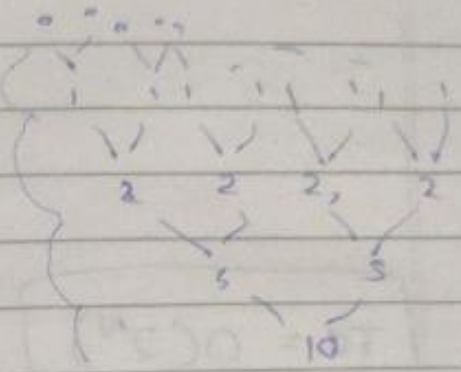
Page No.
Date :

Powers

```
xpn = pow(x, n/2) * pow(x, n/2);
```



Height = 4



Calls = 2

PreOrder = k

PostOrder = k

$TC = (2)^{\log_2 N} + 2k \log_2 N$

$= N + k \log_2 N$

$b^{\log_b a} = a$

$TC = O(N)$

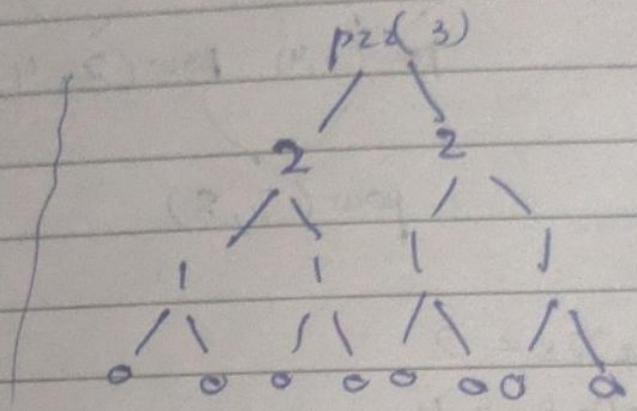
Print Zig Zag

Rec = Call ^{height} + (Pre + Post) * height

$= 2^h + (k_1 + k_2) \times h$ $\quad h = N$

$= 2^N + kN$

$= 2^N + kN$



$TC = O(2^N)$

& Last

First ~~Case~~ Index of Arr

Best Case $O(1)$

Worst Case $O(N)$

Avg Case $O(N/2) \Rightarrow O(N)$

Print Subsequences

TC = $2^N + (K)N$

$= 2^N + KN$

TC = $O(2^N)$

Print Keypad

Calls = 4^N ← worst case

$TC = (4)^N + \{K\}N$

$= 4^N$

$TC = O(4^N)$

Print Permutation (Worst Case)

Calls = N^N ← depend on Str length

Height = N

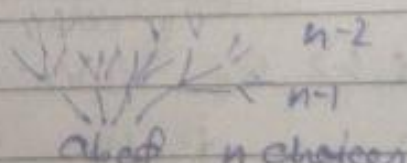
~~$TC = (N)^N$~~ ← Taking substring O(N)

PreOrder = N^N

PostOrder = 0

$TC = N^N + N^2$

$= O(N^N)$ ← $O(N!)$



Print Stair paths

Calls ^{h} + (Pre + Post) h

worst case

$3^N + (k+0)N$

$TC = O(3^N)$

Print Encodings

$2^N + (k+1)N$

$TC = O(2^N)$

Target Sum Subset

$TC = O(2^N)$

Print Maze Paths

$2^{n+m} + (k+0)(m+n)$

$TC = O(2^{n+m})$

Maze Path with Jumps

horiz. choices are no. of cols

vertical choices = rows

diag: = min(row, col)

h_1, h_2, h_3 v_1, v_2 $d_1 d_2$

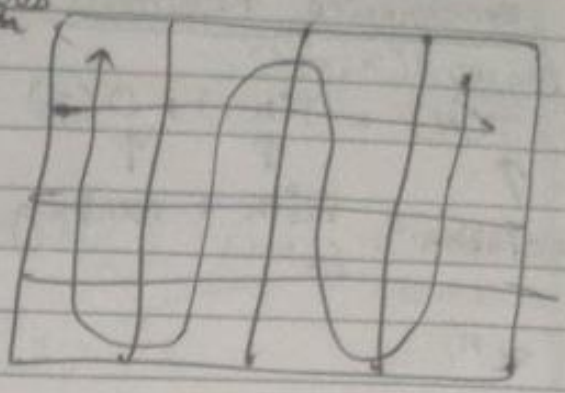
$\Rightarrow n + m + \min(n, m)$

$O((n+m)^{n+m})$

Page No.
Date :

Flood Fill

In worst case all path visited



Height = n * m

$TC = O(4^{n*m})$

N-Queen

Calls = No. of cols (m)

Height = No. of rows (n)

Pre = Check safe queen = $O(N)$ row

$TC = m^n + (n) n$

$\boxed{TC = O(m^n)}$

Recurrence Relations

→ Power (x, n-1)

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

Expectation = Faith + Meeting Expectation.
(x^{n-1})

(x^n)

→ Power $(x, n/2)$

$T(n) = T(n/2) + O(1)$

$(x^n) = x^{n/2} + \text{constant}$

Exp Faith M.E

→ Power $(x, n/2) \cdot (x * x)$

$T(n) = 2T(n/2) + O(1)$

x^n $x^{n/2} \cdot x^{n/2}$ const

Exp Faith ME

* Deriving

⇒ Power Linear

Let, $T(n) = T(n-1) + O(1) \Rightarrow k$

$T(n-1) = T(n-2) + O(1) \Rightarrow k$

$T(n-2) = T(n-3) + O(1) \Rightarrow k$

⋮

$T(1) + O(1) \Rightarrow k$

$T(1) = T(0) + O(1) \Rightarrow k$

Adding All we get

$T(n) = T(0) + n * k$

↑
Base Case (const)

$T(n) = k_2 + n * k_1$

$\therefore$ $\boxed{T(n) = O(n)}$

⇒ Power Logarithmic ($P(x, n/2)$) — BINARY SEARCH (Divide & Conquer)

$T(n) = T(n/2) + O(1) \rightarrow k$

~~$T(n-1) = T(n-\frac{1}{2}) + O(1) \rightarrow k$~~ $T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + k$

~~$T(n-2) = T\left(\frac{n-2}{2}\right) + k$~~ $T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + k$

~~$T(n-3) = T\left(\frac{n-3}{2}\right) + k$~~ $T\left(\frac{n}{8}\right) = T\left(\frac{n}{16}\right) + k$

⋮

~~$T(1) = T\left(\frac{1}{2}\right) + k = T(0) + k$~~ $T(1) = T(0) + k$

Adding ~~Ea~~ All — No of terms = $\log_2 n$

$T(n) = T(0) + x\,k$

$T(n) = T(0) + \log_2 n\, k$

$\boxed{T(n) = O(\log_2 n)}$

Power $P(x, n/2) * P(x, n/2)$

$T(n) = 2T\left(\frac{n}{2}\right) + O(1) \rightarrow k$

$2x \{ T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + k \}$

$4x \{ T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + k \}$

$8x \{ T\left(\frac{n}{8}\right) = 2T\left(\frac{n}{16}\right) + k \}$

$16x \{ \vdots \}$

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

~~$T(n) = 2$~~

$2^0 T(n) = 2^1 T\left(\frac{n}{2}\right) + k$

$2^1 T\left(\frac{n}{2}\right) = 2^2 T\left(\frac{n}{4}\right) + 2^1 k$

$2^2 T\left(\frac{n}{4}\right) = 2^3 T\left(\frac{n}{8}\right) + 2^2 k$

$2^3 T\left(\frac{n}{8}\right) = 2^4 T\left(\frac{n}{16}\right) + 2^3 k$

$\vdots$

$2^x T(1) = 2^{x+1} T(0) + 2^x k$ (No. of terms $= \log_2 n$)

Add All

$T(n) = 2^{x+1} * \log T(0) + k[2^0 + 2^1 + 2^2 \ldots 2^x]$

$= 2^{x+1} T(0) + k\left[\frac{2^x - 1}{2 - 1}\right] = 2^{x+1} T(0) + k 2^x$

$$T(n) = 2^{x+1} T(0) + k * 2^x$$

$$= 2^{\log_2 N + 1} + k' + k\, 2^{\log_2 N}$$

$$= Nk' + kN$$

$$\boxed{T(N) = O(N)}$$

HW Derive Time Complexity for

4. $T(n) = T(n-1) + O(n)$
5. $T(n) = T(n-1) + O(\log_2 n)$
6. $T(n) = 2T(n-1) + O(1)$
7. $T(n) = 2T(n/2) + O(n)$
8. $T(n) = T(\sqrt{n}) + O(1)$ ← Root Function

Iteration ← Cheap → Recursion ← DP memorization

DP tabulation ↑ ↓ Costly (time)

Space Complexity → Recursion call stack space

Generally asked ★

Auxillary Space ↑ Extra ↓ Data Structure

Input Space & Output Space

space to take i/p

space to print o/p

Recursion call stack space ↑ when recursion is used

Overhead ↓ Pushing & popping on other is costly or Switching b/w 2 opers (Context Switch) costly

Constant Extra Space

int ↓ int[26] ↓ double

↓ Primitive datatypes

~~⇒ Inverse Array (Recursion)~~

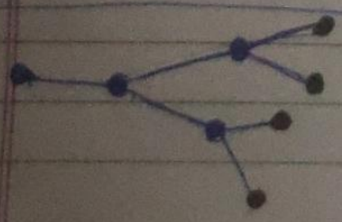
If return void ⇒ ~~NO~~ O(1) o/p space

★ Time vs Space Tradeoff

This is more important

Algorithm		
	space ↑ time ↓	Generally this is seen
	space ↓ time ↑	

★ Recursion call stack Space = Recursion Depth / Recursion tree Height



At a time max 3 calls there

SORTING

(In-place, No extra space)

BUBBLE SORT $O(N^2)$

1 5 9 8 2 1 Compare arr[0], arr[1]

1.1 5 9 8 2 1 1 2

1.2 5 8 9 2 1 (swap) 2 3

1.3 5 8 2 9 1 3 4

1.4 5 8 2 1 9 4 5

1st iteration complete & Heaviest (9) elem is settled at last (sorted)

2. 5 8 2 1 9 0 1

2.1 5 8 2 1 9 1 2

2.2 5 2 8 1 9 2 3

2.3 5 2 1 8 9 3 4

2nd iteration complete, second heaviest (8) elem is sorted

3. 5 2 1 8 9 0 1

3.1 2 5 1 8 9 1 2

3.2 2 1 5 8 9 2 3

3rd iteration completed, 3rd test (5) is sorted

Page No.
Date:

SORTING

(In-place, No extra space)

BUBBLE SORT $O(N^2)$

1	5 9 8 2 1	Compare arr[0], arr[1]
1.1	5 9 8 2 1	1 2
1.2	5 8 9 2 1 (swap)	2 3
1.3	5 8 2 9 1	3 4
1.4	5 8 2 1 9	4 5

1st iteration complete & Heaviest (9) elem is settled at last (sorted)

2.	5 8 2 1 | 9	0 1
2.1	5 8 2 1 | 9	1 2
2.2	5 2 8 1 | 9	2 3
2.3	5 2 1 8 | 9	3 4

2nd iteration complete, second heaviest (8) elem is sorted

3.	5 2 1 | 8 9	0 1
3.1	2 5 1 | 8 9	1 2
3.2	2 1 5 | 8 9	2 3

3rd iteration completed, 3rd heavy (5) is sorted

4. 2 1 5 8 9 Compare arr[0], arr[1]

4.1 1 2 5 8 9

4th iteration completed & all elements are sorted (as if (n-1) elements are sorted then it means all (n) elements are sorted.

Sorted arr: 1 2 5 8 9

Time Complexity: $(n-1) + (n-2) + (n-3) \dots 1$

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

$$O(n^2)$$

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

NOTE: Best Case: Array already sorted

After 1 for loop keep a check if new arr is same as before, then array is already sorted.

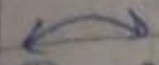
* Selection Sort (In-place, without Extra Space)

select

1st iteration → smallest → stored at arr[0] i=0

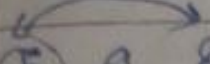
2nd " 2nd " " i=1

3rd " 3rd " " i=2



(5) 9 8 1 2

1st min = 5



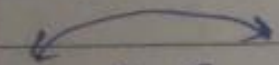
1.1 (5) 9 8 1 2

min = 5

1.2 (5) 9 8 1 2

min = 1

1.3 5 9 8 (1) 2

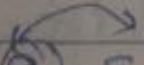


1.4 5 9 8 1 2

Swap 1 & 5 ⇒ Swapping at last

1 5 (9) 8 5 2

min = 9

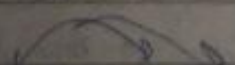


2.1 1 9 (8) 5 2

min = 8

2.2 1 9 8 (5) 2

min = 5



2.3 1 9 8 5 2

min = 2

Swap 9 & 2

1 2 8 5 9

1st 2 elems sorted

1 2 8 5 9

3.1 1 2 8 5 9
min = 8

3.2 1 2 8 5 9
min = 5

3.3 1 2 8 5 9
min = 5

swap 8 & 5

1 2 5 8 9
min = 8

4.1 1 2 5 8 9
min = 8

swap 8 & 8

1 2 5 8 9

If 4 elems are sorted then all sorted

Little optimized due to less swapping

↑ costly for object

```
for (i=0; i<arr.length; i++) {
  int minIdx = i;
  for (int j=i+1; j<arr.length; j++) {
    if (isSmaller(arr, j, minIdx) == true) {
      minIdx = j;
    }
  }
  swap(arr, i, minIdx);
}
```

* Insertion Sort (In-place, No extra space)

sorted | unsorted

2 | 9 5 1 3

Take 9 & insert in sorted part

1.1 [2 9] : 5 1 3

2.1 2 9 5 : 1 3 (swap 5 & 9)

2.2 2 5 9 : 1 3

3.1 2 5 9 1 : 3

3.2 2 5 1 9 : 3

3.3 2 1 5 9 : 3

3.4 1 2 5 9 : 3

~~Array~~ ~~Sorted~~

4.1 1 2 5 9 3

4.2 1 2 5 3 9

4.3 1 2 3 5 9

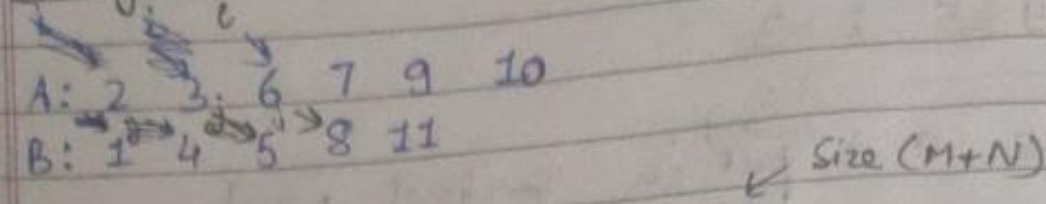
[1 2 3 5 9]

Code

```
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;
    }
  }
}
```

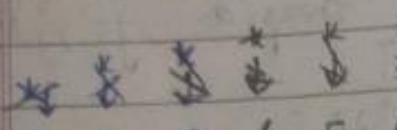
2ptr approach

Merge 2 Sorted Arrays O(N)



(Duplicates should be included)

→ Take 2 ptr in each arr, include the smaller elem in result arr & ↑se ptr.



Code

```
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;
```

* MERGE SORT (Recursion)

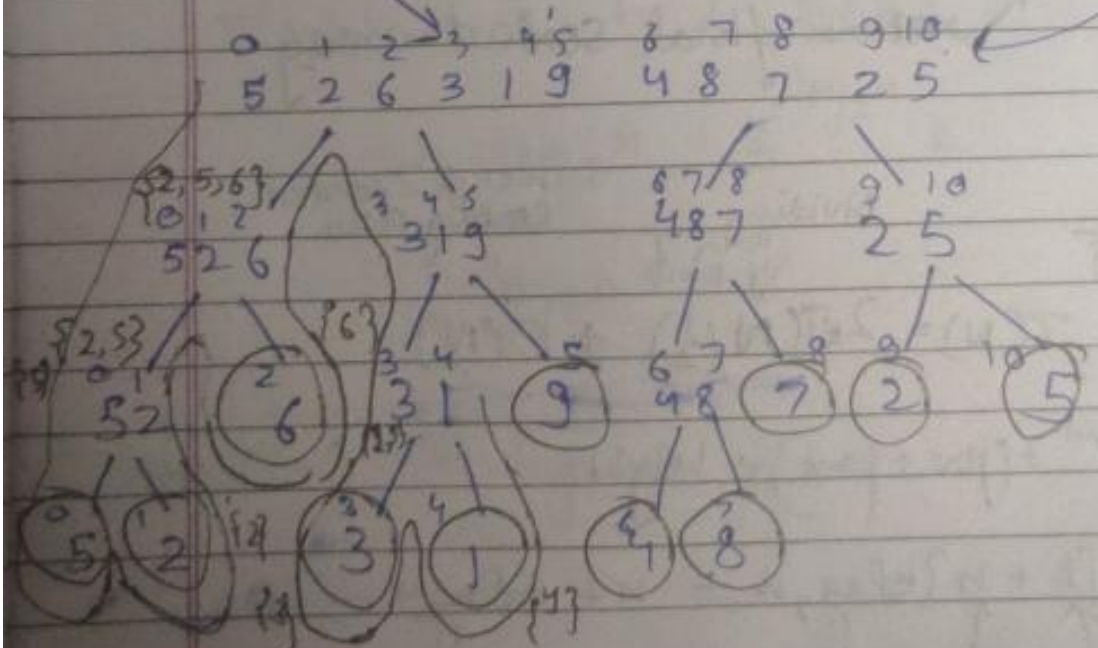
	0	1	2	3	4	5	6	7	8	9	10
arr:	5	2	6	3	1	9	4	8	7	2	5

→ Have Faith in each part that they are sorted by our function

→ Apply 2 ptr. approach in previous ques. to merge the 2 halves.

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

1 2 2 3 4 5 5 6 7 8 9



1 element is sorted

```
public static int[] mergeSort(int[] arr, int left,
                              int right) {
// base case only 1 element remaining
if (left == right){
    int[] base = new int[1];
    base[0] = arr[left]; // or arr[right]
    return base;
}

int mid = (left + right)/2;   // finding mid to divide arr

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

int[] sorted = mergeTwoSortedArrays(leftSorted, rightSorted);
                // merge these 2 sorted arrs

return sorted;   // return final sorted array
}
```

Complexity

Dividing arr into 2 part — merge 2 sorted arrs

Time $T(N) = 2 * T(N/2) + O(N)$

$(\text{Calls})^{\text{height}} + (\text{pre} + \text{post}) * \text{height}$

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

k — const. (base case + mid calc); n — merge 2 sorted arrs

$O(n \log_2 n)$

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

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

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

$\{T(1) = 2T(0) + O(1)\}^{2^x}$ → $\frac{n}{2^n}$

$T(n) = 2^0 * n + 2^1 \frac{n}{2} + 2^2 \frac{n}{2^2} + 2^3 \frac{n}{2^3}$

$\ldots\ldots 2^x * \frac{n}{2^x} + 2^x * T(0)$

no. of eq^n

$T(n) = n.x + 2^x \Sigma(k)$

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

$= n \log_2 n + nk$

$T(n) = n \log_2 n$

Date :

Space (Merge Sort)

① Extra space / Auxillary space

$O(N)$ ← at merging

② Input / Output

i/p in if" parameter (arr)

$O(N)$ ← o/p returning sorted arr

③ Recursion call stack

$O(\log_2 N)$

Target Sum Pair

target = 25

10 ~~28~~ 12 13 14 18

Find 2 nos. which has sum = target

Brute Force

Time → $O(N^2)$

Space → $O(1)$

```
for(int i=0; i<arr.length; i++){
    for(int j=i+1; j<arr.length; j++){
        if(arr[i]+arr[j]==target){
            return new int[]{i, j};
        }
    }
}
return -1;
```

← no pair found.

2 ptr. technique (array sorted)

10 12 13 14 18

left right right right

sum > target

10+18 = 28 > 25 ← we have to reduce sum ~~target~~

∴ right--

sum < target

10+14 = 24 < 25 ← to see sum ⇒ left++

12+14 = 26 > 25

12+13 = 25 = 25 ← return pair

LeetCode (2 Sum II)

```
public int[] twoSum(int[] nums, int target){
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;
}
```

NOTE

Arrays.sort(nums)

↓

uses Tim Sort

↙ ↘

merge sort + insertion sort

(for N>16) (for N<16)

Small arr → insertion sort work better

Page No.
Date :

Time

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

↓

solved

$O(N)$

Space

→ Extra space

Constant $O(1)$

3 2 4 target = 6

↑ ↑ ↑

s e e

Sum = 5

Page No.
Date :

Partition An Array

pivot
~~target~~ = 5

7 9 4 8 3 6 2 5

⇒ Partition around pivot following the b.cond^n below

All elements ≤ pivot in any order

All elements ≥ pivot in any order

LEFT

RIGHT

left → first elem of > region
right → " " " unexplored region

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

l r

7 9 4 8 3 6 2 5

Create these 3 regions using 2 pointers

left(l), right(r)

(r-n) unexplored

[0-(l-1)] ≤ pivot

[l-(r-1)] → pivot

left=0, right=0;

while (right < arr.length)

{ if (arr[right] > pivot){ // element wants to go to > pivot region

right ++; ← sending ptr to > pivot region

} else {

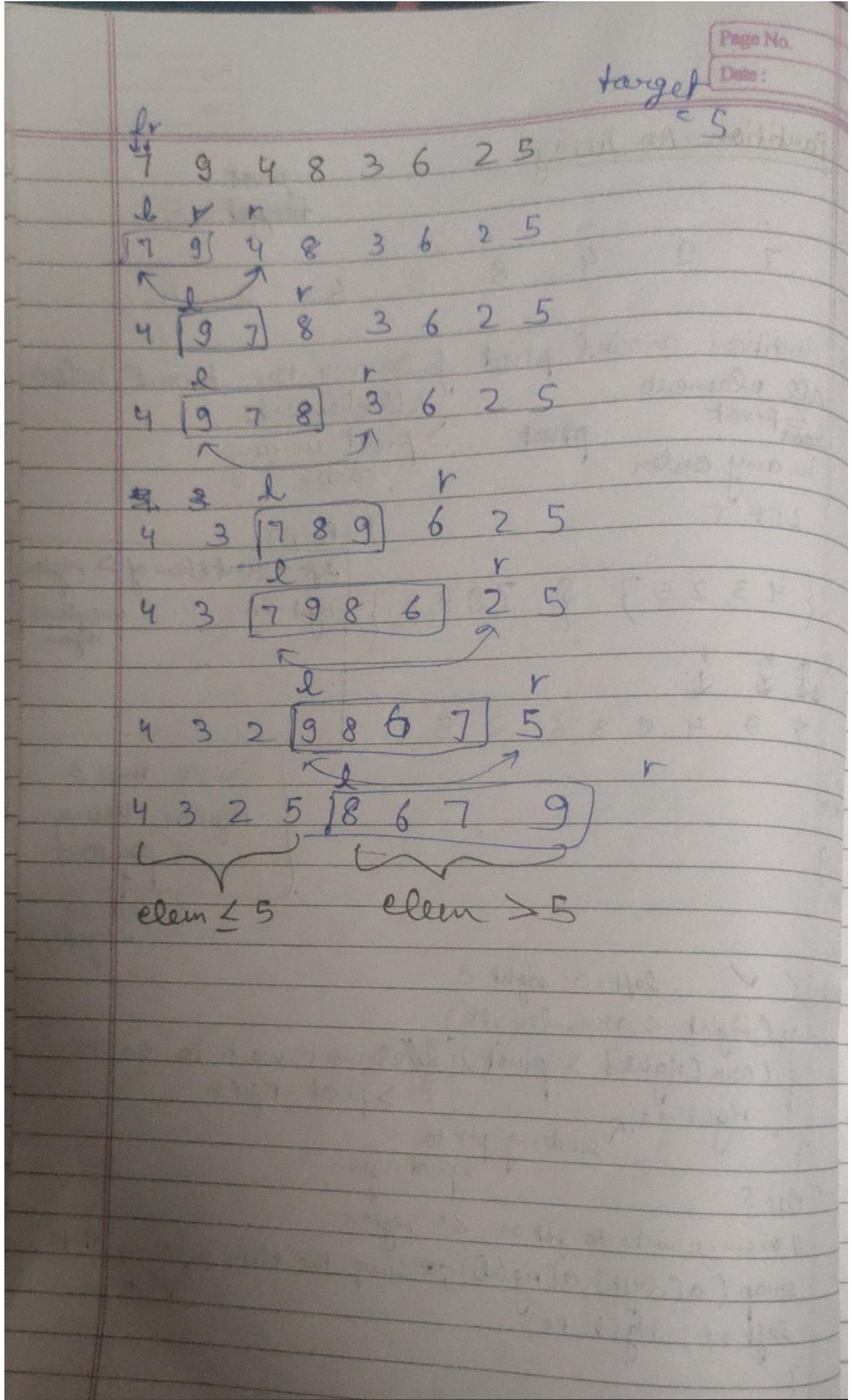
// elem. wants to go to ≤ region

swap (a[left], a[right]); ← swap the elem & bring it to left

left ++; right ++;

}

}


target = 5
l r
7 9 4 8 3 6 2 5
l r r
7 9 4 8 3 6 2 5
l r
4 9 7 8 3 6 2 5
l r
4 9 7 8 3 6 2 5
l r
4 3 7 8 9 6 2 5
l r
4 3 7 9 8 6 2 5
l r
4 3 2 9 8 6 7 5
l r
4 3 2 5 8 6 7 9
elem ≤ 5
elem > 5

Page No.
Date :

★

Sort 01 { Binary Array }

1 0 1 1 0 0 1 0 → 0 0 0 0 1 1 1 1

① 2 Traversals O(N) but 2 traversal
1st traversal ⇒ Count no. of 0's & 1's
2nd traversal ⇒ sort based on count

② ~~Brute~~ Use Partition of array & take pivot as 0

```
int left = 0; int right = 0;
while( right < arr.length){
  if( arr[right] == 1) {
        right++;
  }else{
     swap ( arr[right], arr[left]);
     left++;
     right++;
  }
}
```

3 regions

① Unexplored (r - n)
② 0's [0 - (l - 1)]
③ 1's [l - (r - 1)]

Variations

① Partition → (1) $\leq x$, $> x$

→ (2) $< x$, $\geq x$

② Sort 0 1 → (1) 0's 1's

→ (2) Red's, Blue's

→ (3) -ve's, +ves

→ (4) 0's, nonzeros { Move 0's to left or right }

→ (5) odd's & even's

All same Code.

8 8 8 7 9 8 7 9

5 3 2 6 4

QUICKSORT

① Why? Pivot on last element

② Stable → Duplicate elements

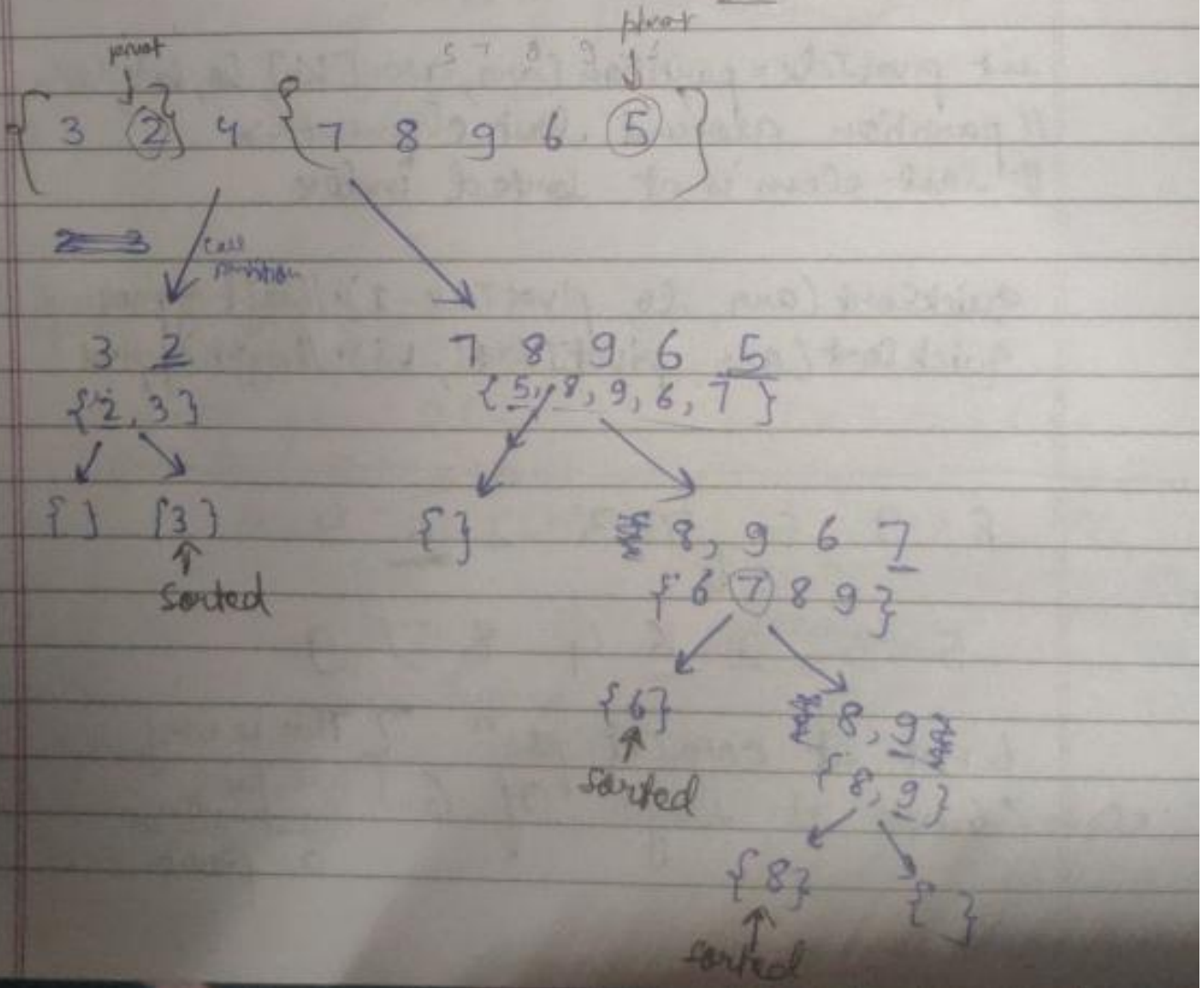
ava: 7 9 4 8 3 6 2 5 ← pivot

using partition algo we get:

[4 3 2] 5 [9 6 7 8]

↑ correct position

E.g. 8 9 5 7 3 2 6 4



* Partition : Subarray [l-r]

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

↳ Partitioning on pivot will result in pivot element to be on its exact (sorted) index.

Partitioning is done in pre-order.

code:

```
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 elem is at sorted index

    quickSort(arr, lo, pivotIdx-1); // left of pivot
    quickSort(arr, pivotIdx+1, hi); // right of pivot
}
```

8 9 5 7 3 2 6 4

↓

5 3 2 6 4 8 7 9

6 not at correct pos^n } This is why
elem < 6 not at left of 6 } we take last elem as pivot

Code for partition

```
int partition (int [] arr, int pivot, int lo, int hi){
int i = lo, j = hi;
while (i <= hi){
  if (arr[i] <= pivot){
    swap (arr, i, j);
    i++;
    j++;
  }
else {
    i++;
  }
}
return (j-1);
}
```

← index of pivot element

Space

input / output : O(n)

extra space : O(1)

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

1 2 3 4 5 6 7 8 → N height of recursion tree

1 2 3 4 5 6 7

1 2 3 4 5 6

Avg O(h) → height of tree

Date:

Quicksort

Time

Avg. / Best case ← 2 equal parts

$T(n) = 2T(?) + O(n)$ ← Partition Algo

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

$O(n \log n)$

Array already sorted & n-1 parts

worst case ← $T(n) = T(?) + O(n)$ ← Partition Algo

Worst Case

1 2 3 4 5 6

After partition

[1 2 3 4 5] 6 ( ) ← Base case ignore

↑ n-1 parts

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

$T(n-1) = T(n-2) + O(n)$

⋮

$T(n) = T(0) + x\,O(n)$

$O(n^2)$ → Corner cases

* Already sorted (↑ing / ↓ing)

Ω ← Best Case
O ← Worst Case
θ ← Avg. Case

Reduces probability of hitting worst case by picking random element

Randomize Quicksort (Reduce worst case time complexity)

2 1 3 5 4 6 8 7

Nearly Sorted (Quicksort Worst case)

→ Let suppose to reduce time complexity, we want to partition around 4 but we are allowed to partition around last element only i.e. 7.

→ In that case we can firstly swap 4 & 7 then perform partition around 4.

2 1 3 5 7 6 8 4

NOTE: If array is nearly sorted then we
→ swap a middle & last element
→ perform partitioning

This fix our chances of having 2 equal parts ($n/2$) leading to $\Omega(n \log n)$ time complexity which is better than $O(n^2)$ for worst case

NOTE: we need not pick middle element, we can pick any random element & swap with last & the perform partitioning which is why its called Randomized Quicksort.

MERGE SORT V/S QUICK SORT

MERGE SORT	QUICK SORT
Best, Avg, Worst } $O(n \log n)$ Time	Best, Avg $\Rightarrow O(n \log n)$ Worst case $\Rightarrow O(n^2)$
Extra Space : $O(n)$ ↑ Merge 2 sorted array	Inplace Extra space : $O(1)$
Stability ✓	Stability ✗
Preferred for non-sequential data (Linked List)	Preferred for Array

Duplicate elements → repeats

* Stability → If element ~~comes~~ in an array then the order in i/p = order in o/p
i.e. 4^a before 4^e always

4^a 3^b 5^c 1^d 4^e 5^f 1^g 5^h

4^e ↑ called Stable

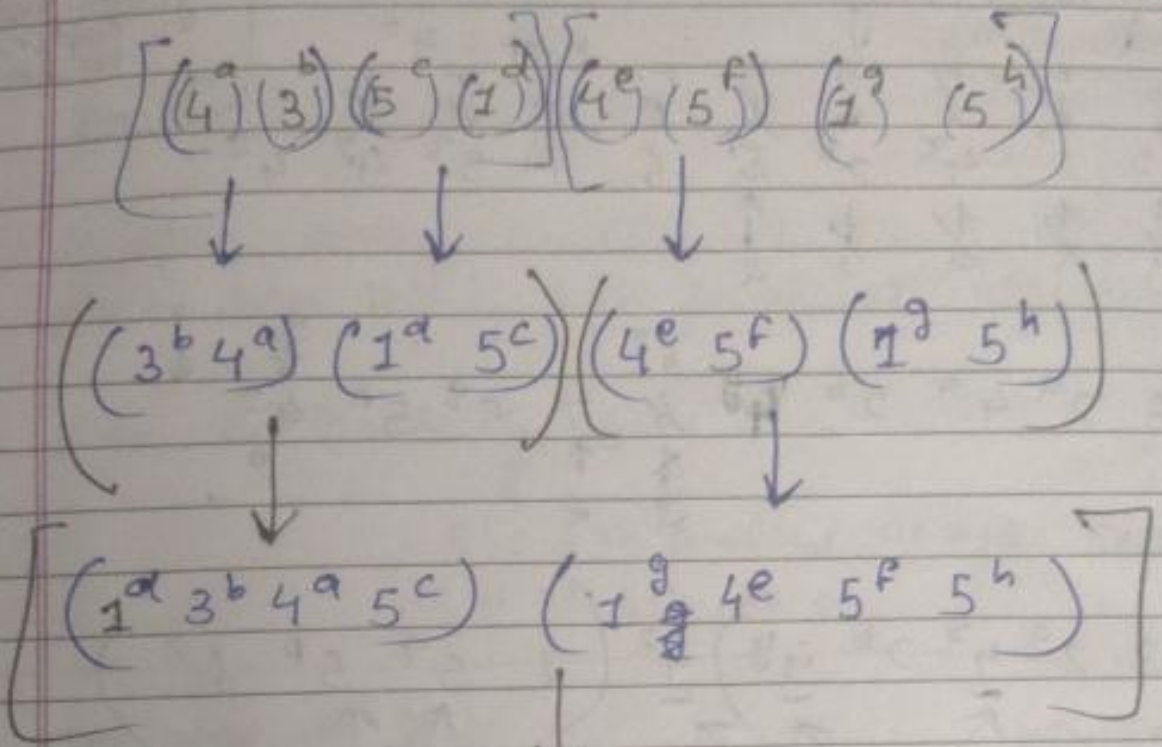
For stable, After sorting

1^d 1^g 3^b 4^a 4^e 5^c 5^f 5^h

Merge Sort

Reason for stable

due to used of Merge 2 Sorted Arrays



∵ The order of duplicate elements is maintained

∴ Stable

Stability is maintained because of this =

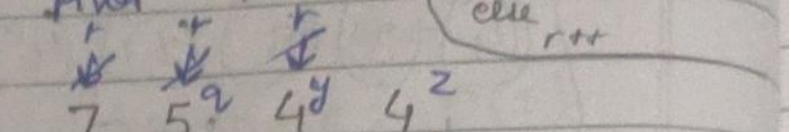
algo

```
if(a[i] <= b[j]){
    c[k] = a[i];
    i++; k++;
}
else{
    c[k] = b[j];
    j++; k++;
}
```

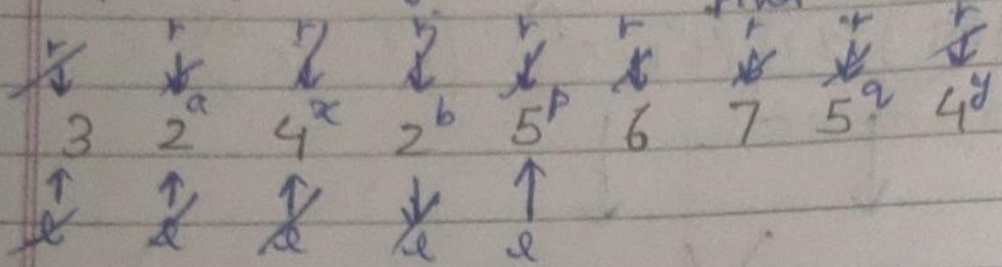
Page No.
Date :

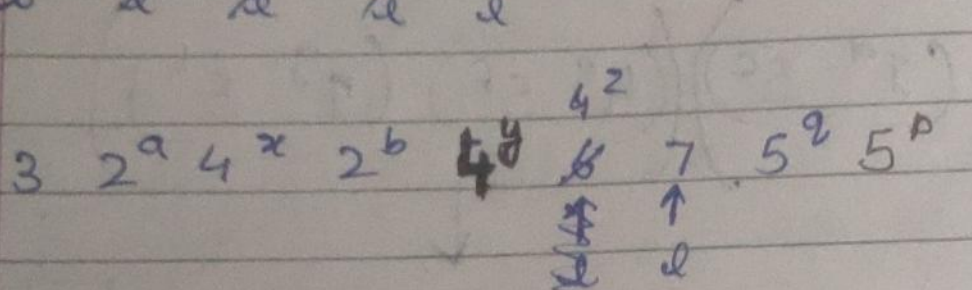
Quicksort

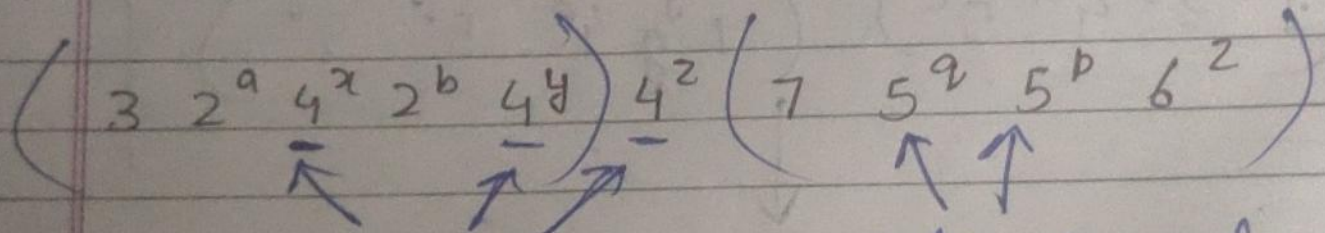
Not stable due to use of partition algo of an array



pivot = 4^z







order intact

order changed
i.e. 5^q is before 5^p
which is not same as
i/p array.

This change in order of 5^p & 5^q caused quicksort to be unstable.

Quick Select

Binary Search / Div & Conquer

→ Quicksort / Partition

Kth smallest element from an unsorted array

8 4 5 7 3 2 6 4 1

6th Smallest

→ If after sorting an element comes at Kth place then it is kth smallest element

1 2 3 4 4 5 6 7 8

↑

5 is 6th smallest as after sorting it is at 6th place.

Take pivot at last

→ Ask that after partition is it at kth position ~~its first position~~

E.g.: 8 9 5 7 3 1 2 6 4 (3) are you 6th smallest

At 3rd ✓ ≠ 6th ∴ not 6th smallest

1 2 3 (8 9 5 7 6 (9)) Pivot

Now, 6th smallest lies here as 3rd pos "4th

(1 2 3) 4 (7 6 9 5 8) Pivot

4th idx ≠ 6

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

∴ 8th > 6 ∴

Answer lies here

Do this to find 6th min

```
public static int quickselect(arr, lo, hi, k){
    if(lo == hi) return arr[lo]; ← only 1 element
after partition find index of pivot elem
    int pivotIdx = partition(arr, arr[hi], lo, hi);
pivot is kth smallest
    if(pivotIdx == k) return arr[pivotIdx];
pivot is not
    kth smallest is at rt. side of pivot
    if(pivotIdx < k) return quickselect(arr, pivot+1,
                                        hi, k);
kth smallest is at left side of pivot
    return quickselect(arr, lo, pivot-1, k);
}
```

Time — Divide in terms of partition / partition algo

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

without sorting

$\therefore O(n)$

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

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

① 2 traversal

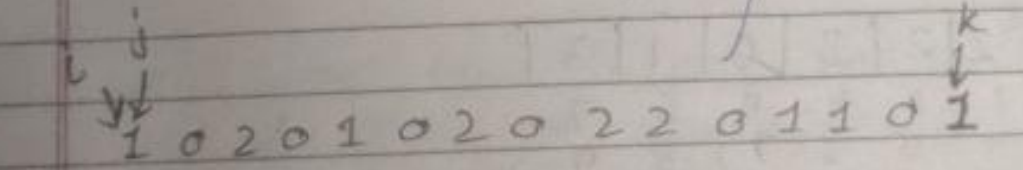
→ Count no. of 0's, 1's & 2's
→ Put a/c to count of each value

② 1 traversal

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

$O(n)$

4 regions (3 ptr)
→ unexplored (j - k)
→ 0's [0 - (i-1)]
→ 1's (i - (j-1))
→ 2's (from back) [(k+1) - n]
↖ to reduce no. of swap




```
arr.length - 1;
i=0, j=0, k= ~~n-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--;
  }
```

Count Sort

↳ use of Extra Space

Apply this when
→ Range of elements in Array is fixed & SMALL

→ 1st traversal
• Find minimum element & find max. elem.
(2) (9)

→ 2nd traversal
• 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

★ Range (2 - 9) ⇒ (9 - 2 + 1) = 8 unique type of elements at max

★ Create 8 boxes as 8 unique elements possible

1	3	2	2	3	0	1	3
idx = 0	1	2	3	4	5	6	7
value = 2	3	4	5	6	7	8	9

Map :- 0 → 2
1 → 3 ⇒ val - min elem = idx
7 → 9

★ In this freq. array store freq of all nos present in i/p array.

Page No.
Date :

⇒ If you don't want to ensure STABILITY of i/p arr then
loop through freq. arr & display each val. no. of times the freq.

Ans: 2 3 3 3 4 4 5 5 6 8 9 9 9

Code

```
count-sort ( arr ) {

    int min = Integer. MAX-VALUE;
    int max = Integer. MIN-VALUE;

    for(int i=0; i< arr. length; i++) {
        min = Math. min(min, arr[i]);
        max = Math. max(max, arr[i]);
    }

    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 ++;
        }
    }
}
```

Total Time

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

↑ min max find ↑ freq arr creation ↑ sort arr using freq arr

Freq

1	3	2	2

Times each num keep moving

k + 3k + 2k + 2k

= 8k = k * n

⇒ O(n)