Sorting - 1

Sorting => Arranging elements in a particular order

Why sorting?

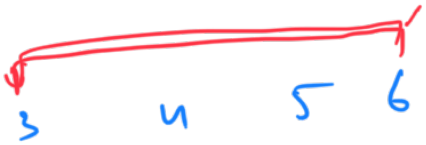
1) Easy to search in ordered items (Dictionary) -> they are stored in sorted.
   ↓ grandchildren

2) Binary Search (sorted/ ordered arrays)

3)

A = [3 5 1 6 4 2]

(N)

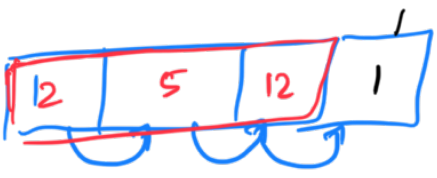


A = 1 2

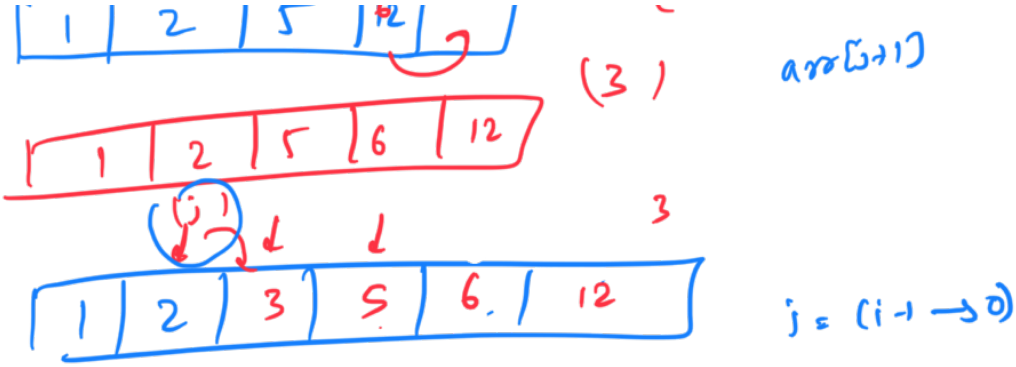
Sorting Techniques:

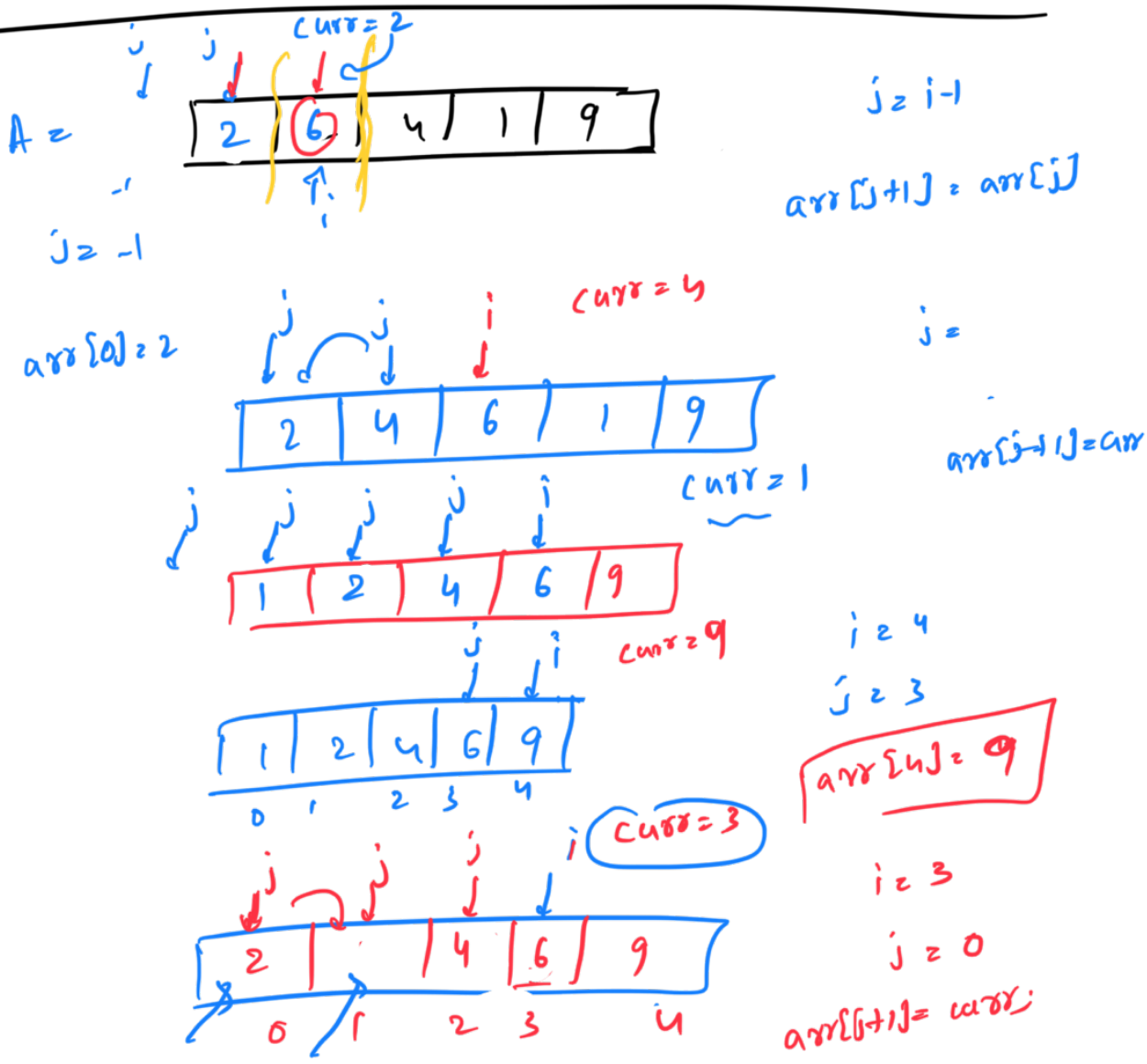
Problem: Arrange students in increasing order of heights

Approach: (Insertion Sort) (Playing cards)



6






```
void insertionSort(arr, N) {
    for(j=1; i<N; i++) {
        curr = arr[i];
        j = i-1;
        while(j>=0 && arr[j] > curr) {
            arr[j+1] = arr[j];
```

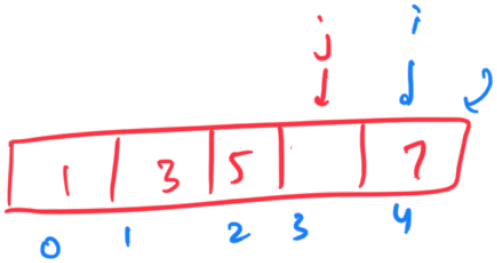
j--;

arr[j+1] = curr;

curr = 4

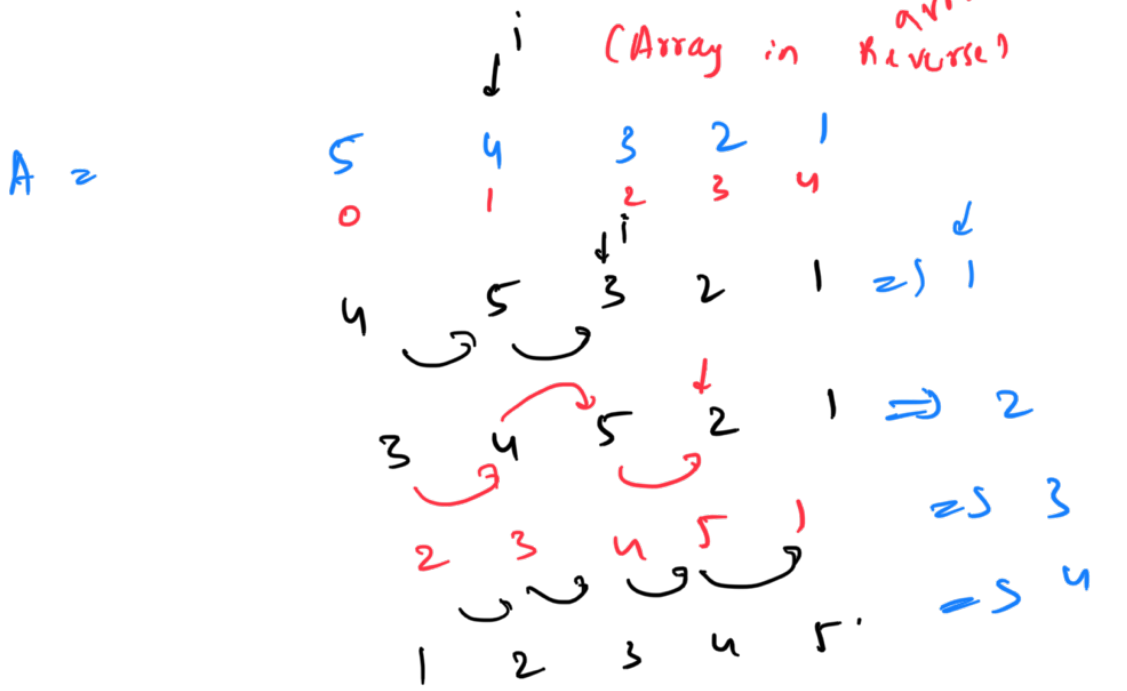
i = 4

j = 3



arr[i]

arr[4].



$1 + 2 + 3 + 4 \cdots (n-1) = O(n^2)$

(1) + (1) + (1) + (1)

O(n) operations

T.C: O(n)

A: 1 3 2 4 5

1 2 3 4 5

operations: (1) + (1+1) + 1 + 1 + 1

$O(n)$

If few elements are out of place, the T.C would be $O(n)$

Best case: $O(n)$ ✓ [Sorted Array]

Worst case: $O(n^2)$ ✓ [Reverse Sorted Array]

S.C: $O(1)$

Quick sort & Merge sort

Selection Sort

Approach 2:

A = 8 4 3 1 2 5

A = 1 4 3 8 2 5

1 2 3 8 4 5

1 2 3 8 4 5

1 2 3 4 8 5

1 2 3 4 5 8

Right / left

Correct position

(i to n-1)

Best Case : O(n)

A = 1 2 3 4 5 6

Best Case : $O(n^2)$

Worst Case : $O(n^2)$

$n + n-1 + n-2 + \dots 1$

$O(n^2)$

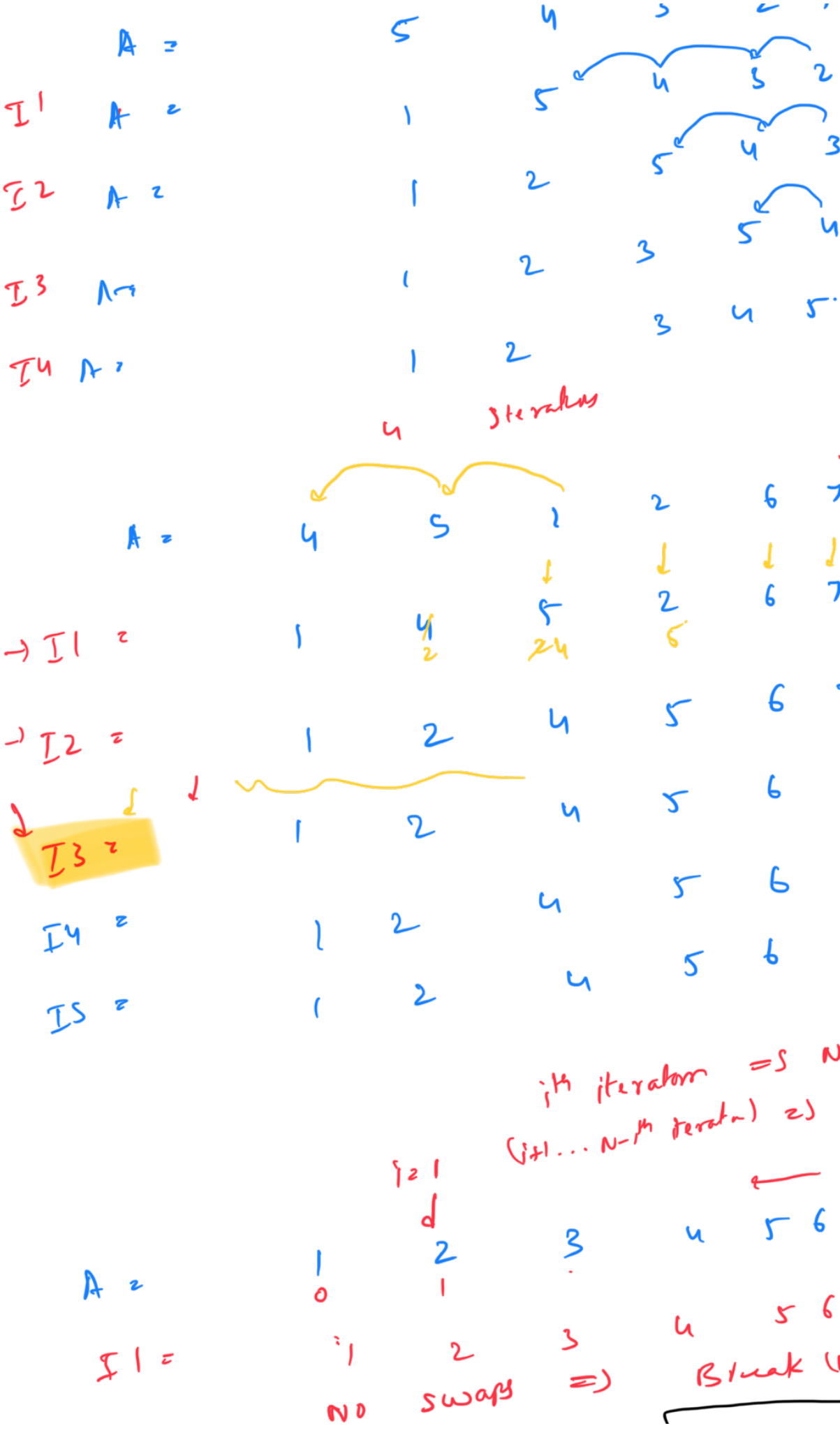
Approach 3 : Bubble Sort

→ the smallest number bubbles up to the correct position

→ we have 4 elements in its right place

→ (N-1) iteration

N = 5


A =
I1 A =
I2 A =
I3 A =
I4 A =
5 1 2 3 4
4 5 2
5 4 3
5 4
3 4 5
1 2
iterations
N = 6
iterations : 5
A =
4 5 1 2 6 7
I1 =
I2 =
I3 =
I4 =
I5 =
1 2 4 5 6 7
i = 1
A =
I1 =
1 2 3 4 5 6
NO swaps ⇒ Break (N-1)th elem
i = N-2
i+1 ... N-1
ith iteration = 0 swaps
(i+1 ... N-1th iteration) ⇒ N swaps
O(n)

0 1 2 3 ... N-1

indices.

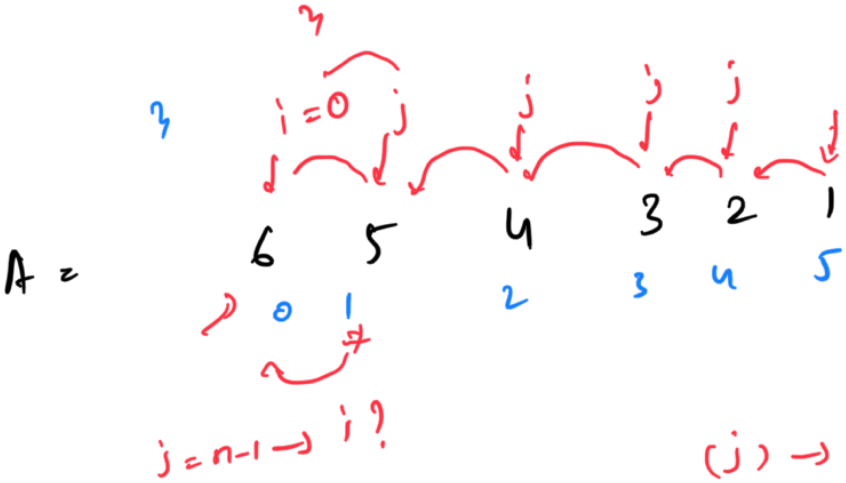
Iterations => n-1

```
void bubbleSort(arr, N) {                    → N-1 iterations
    for (i = 0; i <= N-2; i++) {
        bool swap = false;
        for (j = n-1; j >= i+1; j--) {
            if (A[j] < A[j-1]) {
                swap(A[j], A[j-1]);
                swap = true;
            }
        }
        if (swap == false) break;
    }
```

N = 6

i = 0

j = 5



Worst Case T.C: $O(n^2)$ (Reverse Sorted)

Best Case T.C: $O(n)$ (Sorted Array)

Count Sort:

If the numbers of the array are in a limited range, we can use count sort.

A: 0 0 0 1 1 1 2 2 2

Approach 1: Use any of the above sorting

$O(n^2)$ $O(1)$ to $O(n)$

Approach 2:

count_zeros = 3
count_ones = 4
count_twos = 3

3 variables. $O(3) = O(1)$

Array → 0 0 0 1 1 1 1 2 2 2

T.C: $O(n)$

S.C: $O(1)$

sorting a string

"lower case"

S = "c b b a a d c c b"

[a - z]

(Hashmap)

a	b	c	d	e	...	x	y	z
2	3	3	1	0		0	0	0

freq[26] =

→ S = a a + b b b + c c c + d

S = a a b b b c c c d

S-C: $O(N)$? ⇒ $O(26)$

freq array of size 26.

⇒ 26 distinct characters.

T-C: $O(n)$ ⇒ $O(n + 26)$ ⇒ $O(n)$

$O(n + R)$

for (i = 0; i < 26; i++) {

for (

print freq[i] no. of

}

Question: Array elements are in the range $0 - 10^6$

$10^9, 10^9$

Distinct chars = 10^6

$10^8 \Rightarrow$ 106

arr[10^6] ⇒

freq[10^6]

$(10^7 - 10^9)$

No of distinct element

$10^9 - 10^7$

$= 10^7 (100 - 1)$

$= 99 \times 10^7$ ✗

10^7 distinct char

[L, R]

$[R - L] \leq 10^6 \Rightarrow$ then we can use count-sort

Max size of array we can creat.

A =

Question: Merge 2 sorted arrays into one single sorted array

A = | 1 | 3 | 4 | N=3

B = | 2 | 5 | 6 | 8 | M=4

Ans = | 1 | 2 | 3 | 4 | 5 | 6 | 8 |

Approach 1:

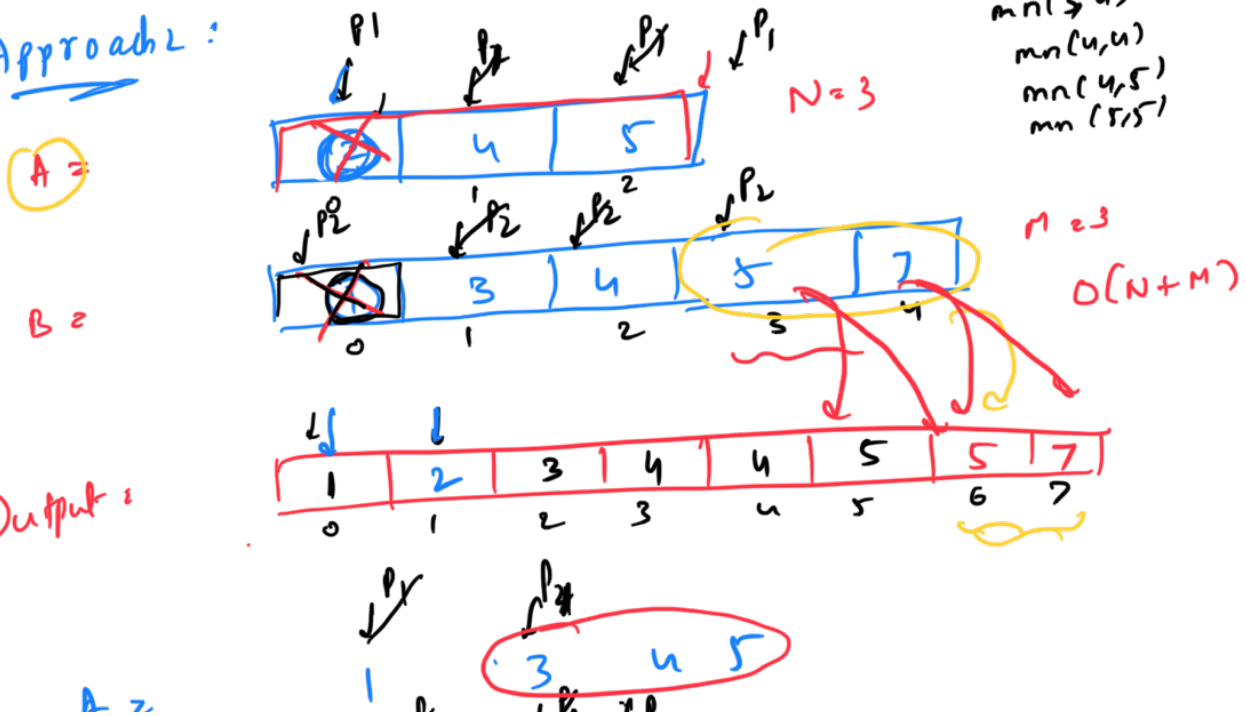
Take a new array and copy the elements

B = | 1 | 3 | 4 | 2 | 5 | 6 | 8 |

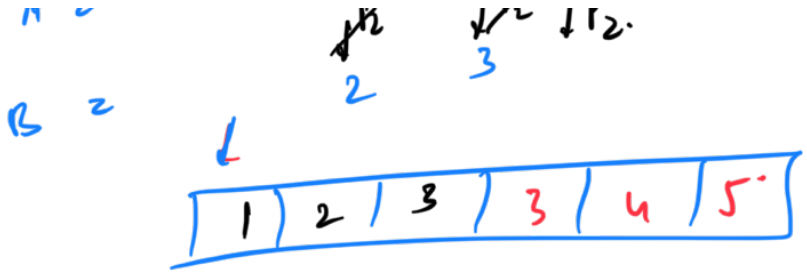
Use any sorting algorithm

T.C: $O(n) + O(n^2) = O(n^2)$

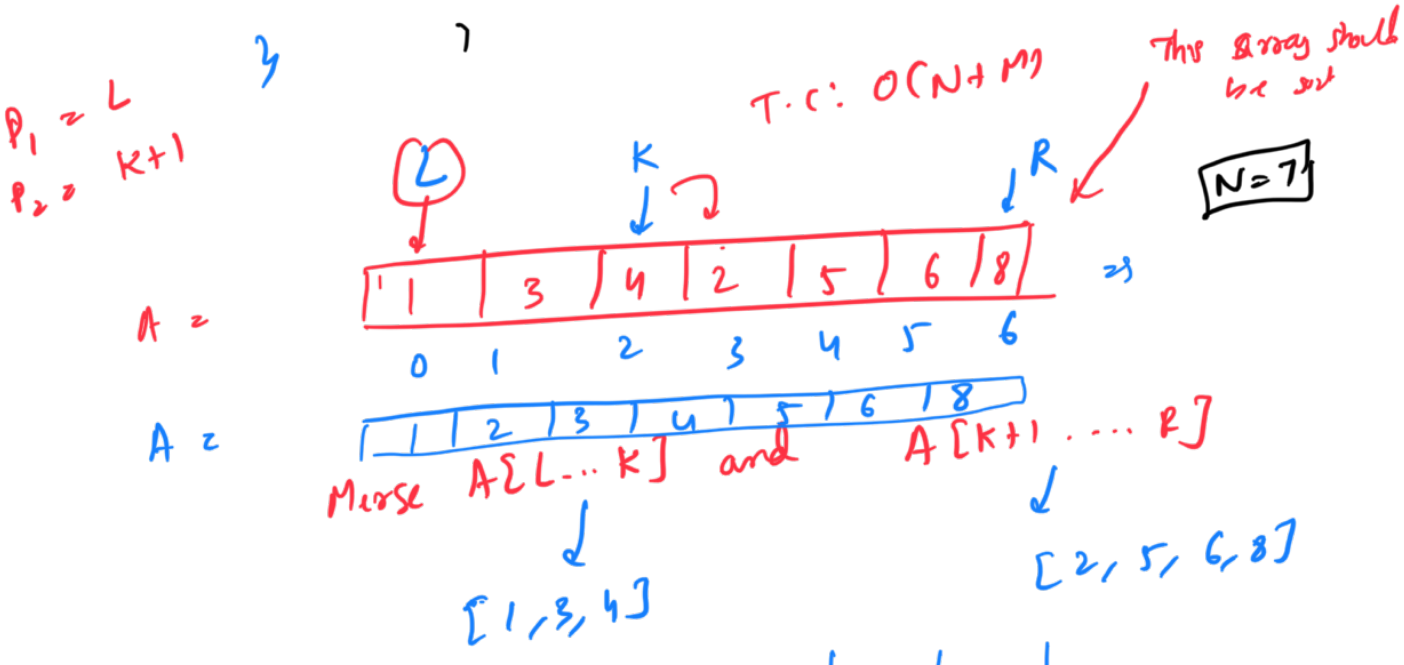
Approach 2:



min(2, 1) = 1
min(2, 3) = 2
min(3, 4)
min(4, 4)
min(4, 5)
min(5, 5)

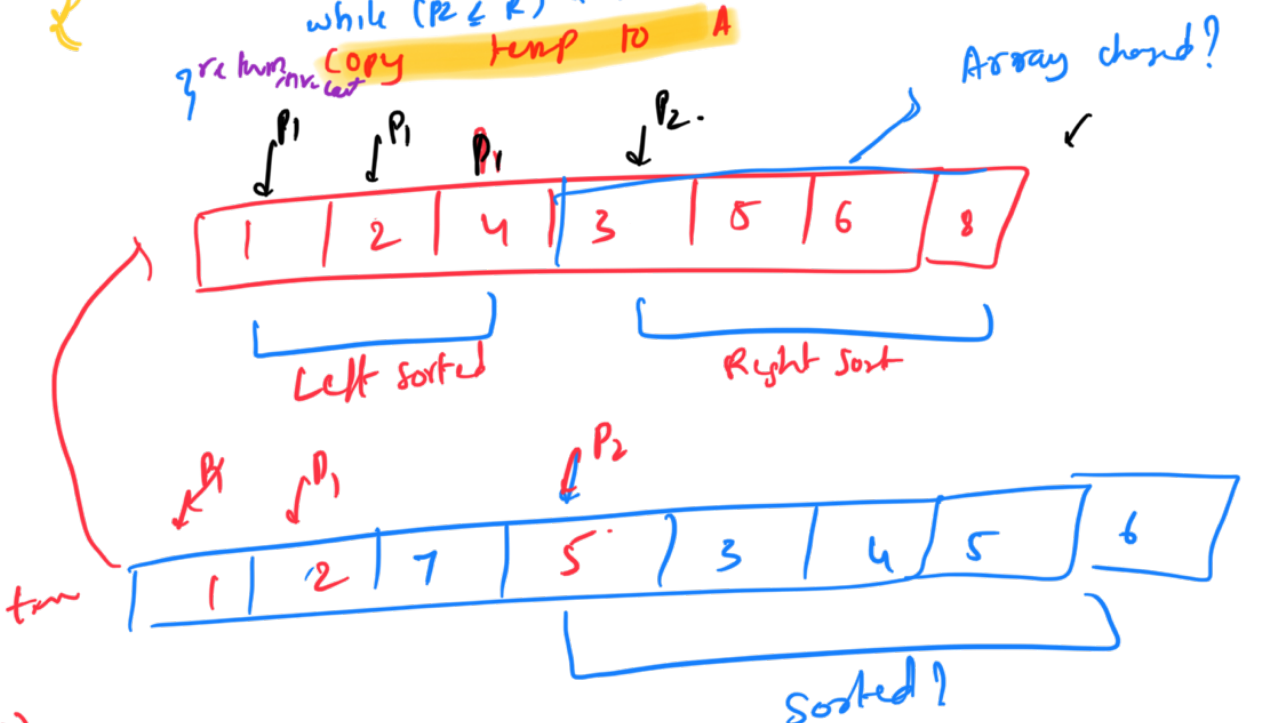



```
void merge( A, n, B, m ) {
int p1 = 0, p2 = 0, i = 0;
while ( p1 < N && p2 < M ) {
  if( A[p1] < B[p2] ) {
      Ans[i] = A[p1];
      p1++; i++;
  }
  else {
      Ans[i] = B[p2];
      p2++; i++;
  }
}
while (p1 < n) {
    Ans[i] = A[p1];
    i++; p1++;
}
while (p2 < m) {
    Ans[i] = B[p2];
    i++; p2++;
}
}
```




```
( a, L, K, R ) {
```

```
void merge(
    p1 = L;   p2 = k+1;   i = L;
    temp[];   inv_count = 0;
    while( p1 <= k && p2 <= R ) {
        if( a[p1] <= a[p2] )
            temp[ i++ ] = a[p1++];
        else {
            temp[i] = a[p2];
            i++;  p2++;
            inv_count += (k - p1 + 1);
        }
    }
    while (p1 <= k) . . . .
    while (p2 <= R) . . . .
    return inv_count;
    Copy temp to A
}
```



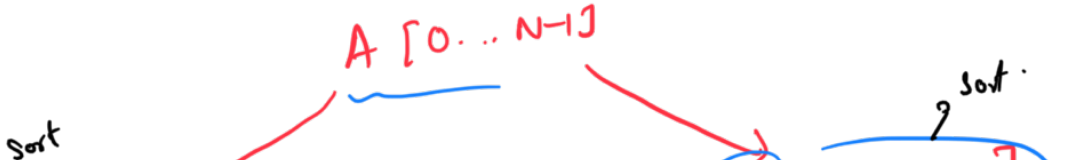
min(1,2)
min(5,2) = 2

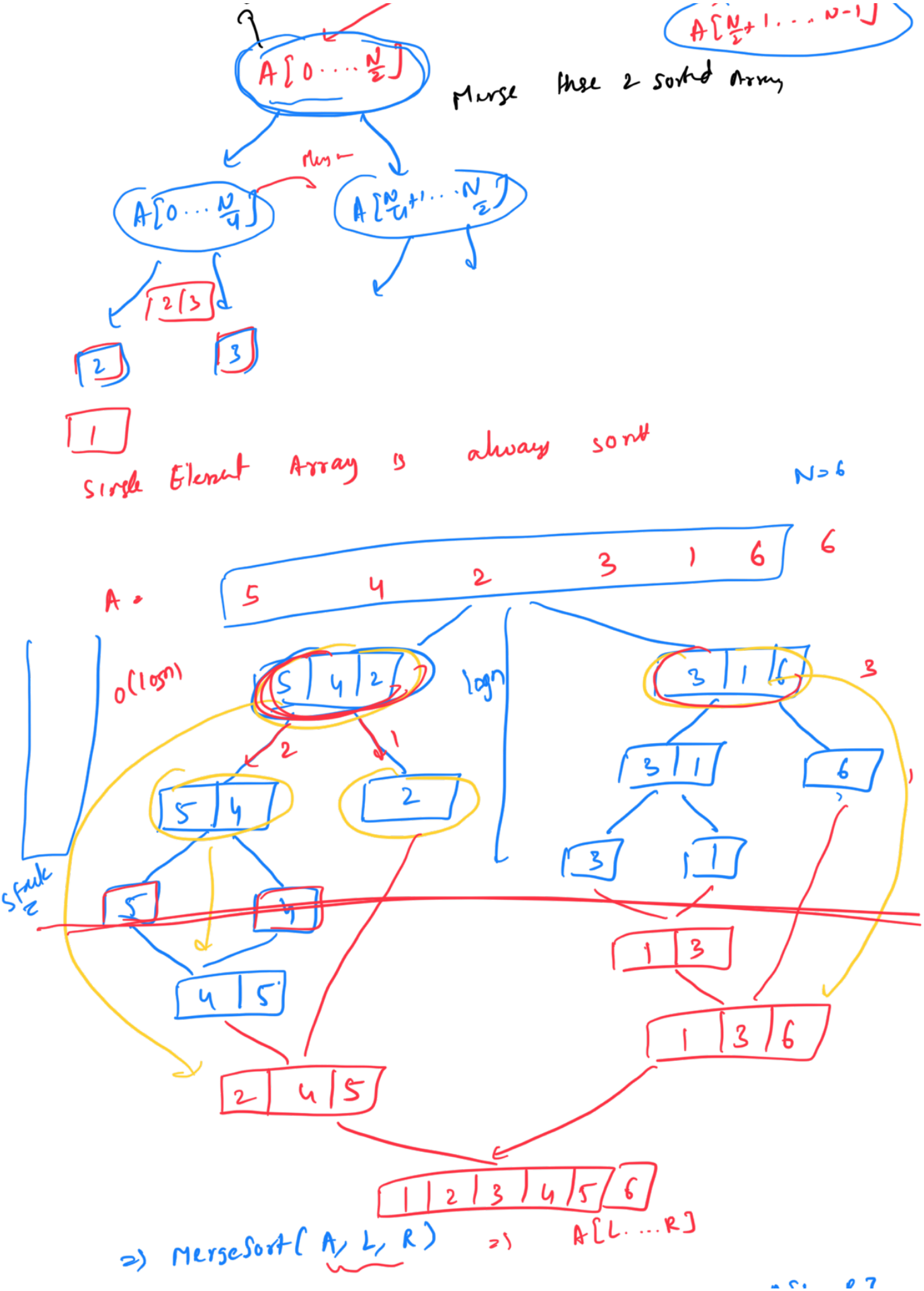
T.C: O(n)

S.C: O(n) → temp array

Merge Sort

Divide and Conquer Approach

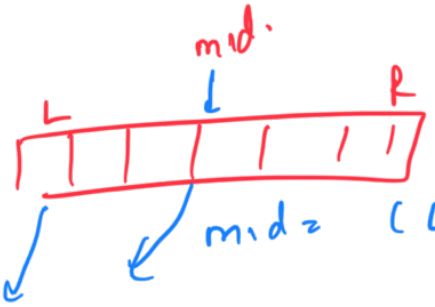



A[0 .... N/2]
A[N/2+1 ... N-1]
Merge these 2 sorted Array
Recursion
A[0 ... N/4]
A[N/4+1 ... N/2]
1 2 3
2
3
1
Single Element Array is always sort
N=6
A:
5 4 2 3 1 6
6
O(log n)
log n
5 4 2
3 1 6
3
2
3 1
6
5 4
2
3
1
5
4
Stack
4 5
1 3
2 4 5
1 3 6
1 2 3 4 5 6
=> MergeSort(A, L, R)
=> A[L .... R]

Assumption:
mergeSort(A, L, R) sorts the array A[L...R]

Main Logic:
```
mid = (L+R)/2;
mergeSort(A, L, mid);
mergeSort(A, mid+1, R);
merge(A, L, mid, R);
```

Base Case: if(L == R) return;



mid = (L+R)/2;

Left = (L, mid)
Right = (mid+R)

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

$O(n \log n)$

(Master's theorem)

$T(n) = a \cdot T(\frac{n}{b}) + O(n^d)$

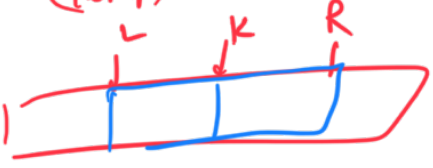
a = 2
b = 2
d = 1

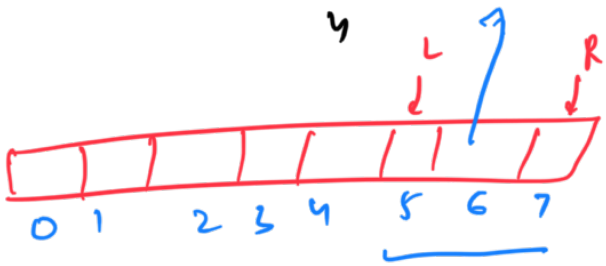
```
mergeSort(A, L, R) {
    if(L == R) return;
    mid = (L+R)/2;
    mergeSort(A, L, mid);      // T(n/2)
    mergeSort(A, mid+1, R);    // T(n/2)
    merge(A, L, mid, R);       // O(n)
}
```

K


```
main() {
    mergeSort(A, 0, n-1);
}
```

(temp, Compare)





$\frac{L+R}{2}$

R/2 = 7/2 = 3 — mid?

$T(N) =$

T.C : $O(n \log n)$

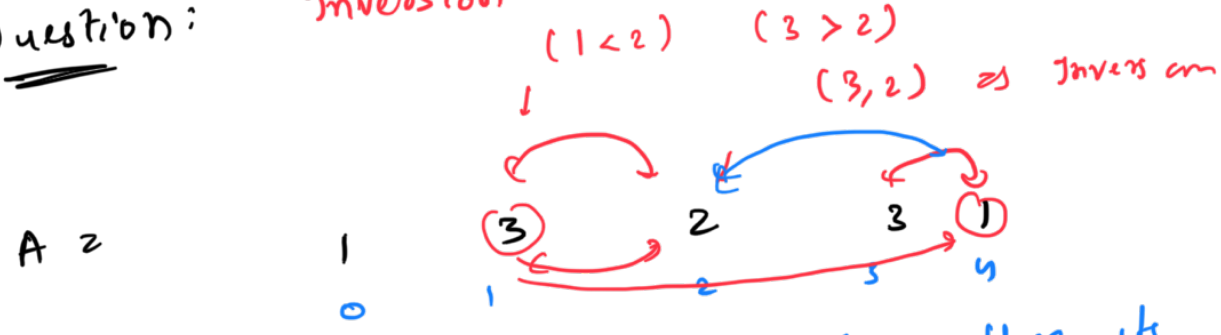
S.C : $O(n)$ + $O(\log n)$

$O(n)$ → temp

$O(\log n)$ → stack space.

S.C : $O(n)$

Question: Inversion Count in an array



if (i < j) and A[i] > A[j], then it is an inversion pair.

Ans = (3,2) (3,1) (2,1) (3,1) 4 Pairs

Ex 1 A =

0	1	2	3
5	4	3	2

(5,4) (5,3) (5,2) → 3
(4,3) (4,2) → 2
(3,2) → 1

3 + 2 + 1 = 6

Ex 2 A =

4	5	1	2
↓	↓	↓	↓
1	2	7	3

(4,1) (4,2) (4,3)
(5,1) (5,2) (5,3) => 7

3 + 3 + 1 = 7

(1, 3)

Brute Force:

For every element, count no. of element less than it to the right.

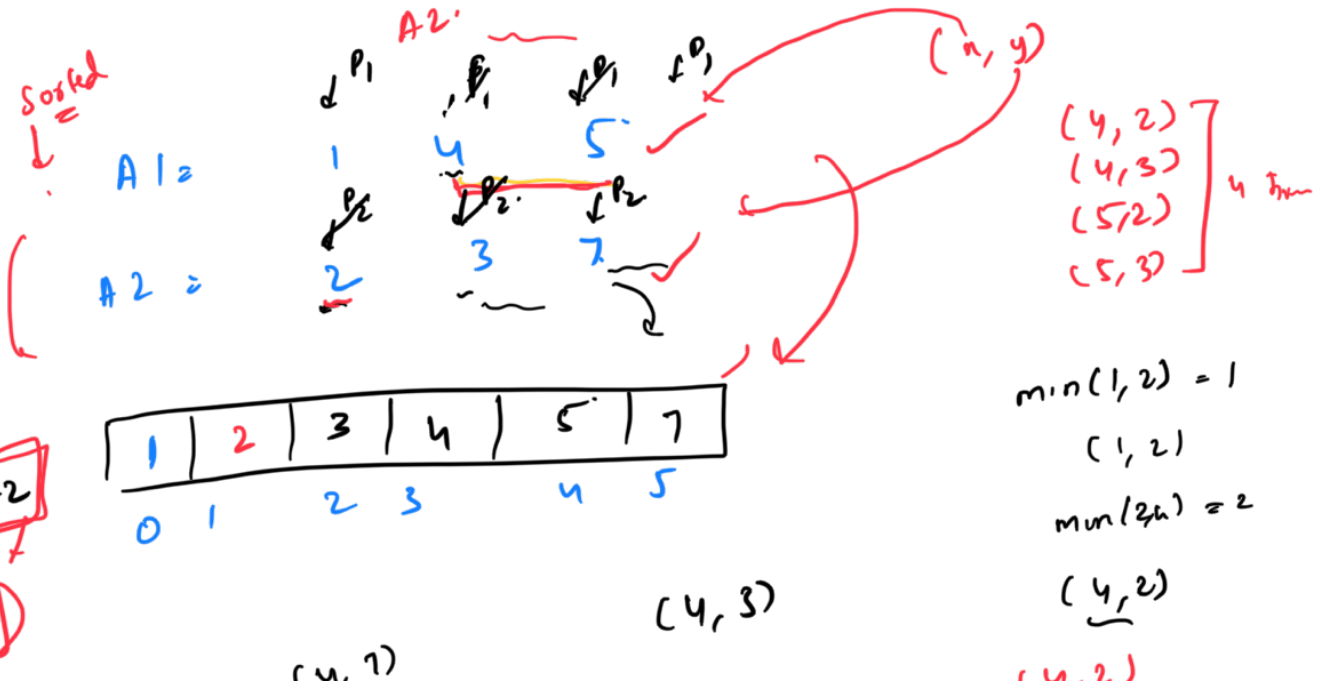
```
for(i = 0; i < n; i++) {
    for(j = i+1; j < n; j++) {
        if(A[j] < A[i])
            cnt++;
    }
}
```

T.C: $O(n^2)$

S.C: $O(1)$

Approach 2:

Question: Given 2 sorted Arrays count the no of inversion pairs where 1st element is from A1 and 2nd element is from A2.



min(1, 2) = 1

(1, 2)

min(2, 4) = 2

(4, 2)

(4, 2)

(4, 3)

(4, 7)

(5, 7)

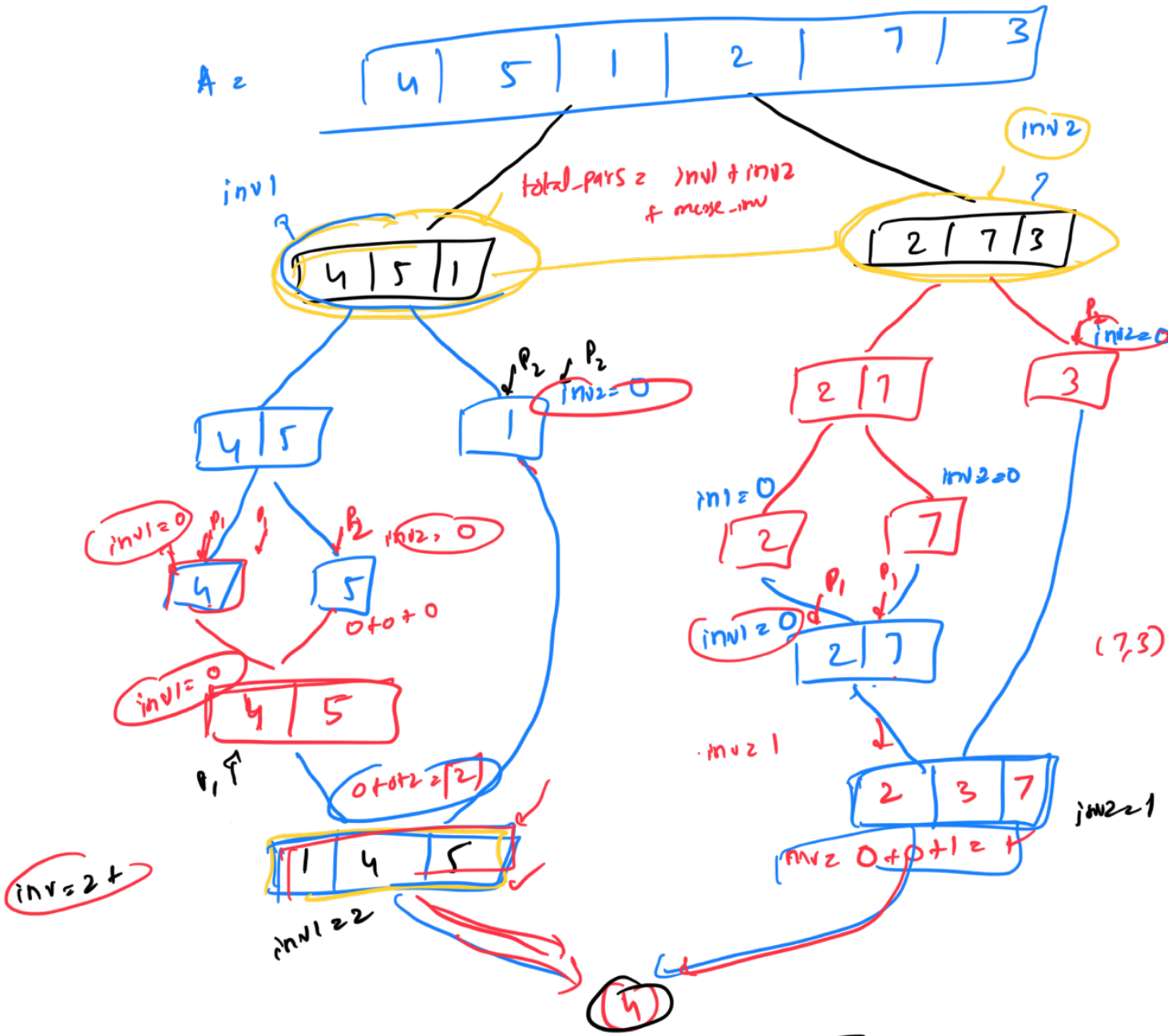
T.C: $O(n+m)$

2 unsorted Array

A1 = 4 5 1 4

A2 = 2 7 3

If we sort the 2 arrays, we can find inversion pairs in O(n+m)




```
int inv(A, L, R) {
    if (L == R) return 0;
```

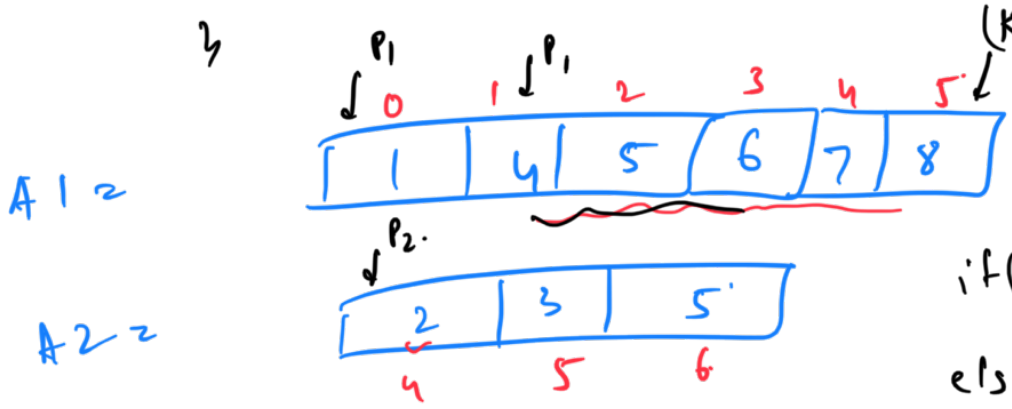
ans = 0

ans += inv(A, L, mid);

ans += inv(A, mid+1, R);

ans += merge(A, L, mid, R);

}



K - P1 = 5 - 1 = 4

(K - P1 + 1)

(K)

(1, 2)

if(A1[P1] < A2[P2])

else {

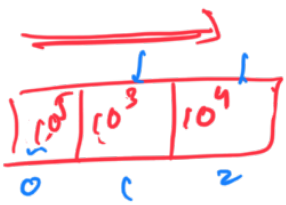
}

(4, 2)

T.C : O(n log n)

S.C : O(n) + O(log n)

(temp) (stack)



$0 - 10^6$


```
int freq[10^6];
for(i=0; i<n; i++){
    freq[A[i]]++;
}
```

] => O(n)

$O(10^6) = O(1)$

Case1 A = 3 4 5 6 7 N = 5 n iterations

case2 A = 3 5 4 6 7

= 3 4 5 6 7

O(n+n) = O(n)