NAME : PRAGYA AGARWAL
STUDENT ID: 1001861779
ASSIGNMENT ID : 01

ASSIGNMENT-01

Ques 1 You have an array containing integers from 1 to 10 (not in order) but one number is missing (there are 9 numbers in the array).

a) write a pseudo code to find the missing number.

a) To find the one missing number from array containing integers 1 to 10 (not in order) we will use template / technique of XOR.

XOR has certain properties :

(1) When both numbers are distinct , $a \wedge b = 1$

(2) When both numbers are same , $a \wedge a = 0$

(3) When a number is XOR-ed with 0 , $a \wedge 0 = a$

PSEUDO CODE:

```
input : Array A of integers , size of
        array given  n = 9
int XOR1 = A[0] ;   /* Create variable to save
                       XOR of all the elements in
                       given array . Initialized
                       to first element of an
                       array. */
```

```
int XOR2 = 1;      /* Create variable to save XOR of
                      all the elements from 1 to 10.
                      Initialized to 1. */

for (int K = 1; K < 9; K++)   /* Run for loop from
                                 index 1 to 9 with K
                                 as counter */

    XOR1 = XOR1 ^ A[K]        /* Updating XOR1 for
                                 each XOR operation
                                 performed with elements
                                 of given array */

for (int j=2; j <= 10; j++)   /* Run for loop to
                                 traverse all the array
                                 elements i.e. 10 elements
                                 Starting from 2 to 10
                                 as XOR2 = 1 */

    XOR2 = XOR2 ^ j;          /* Updating XOR2 for each
                                 XOR operation performed
                                 between elements 1 to 10
                                 */

return (XOR1 ^ XOR2);         /* Return missing number */
```

Example:

Input array A[10] = {1, 2, 4, 5, 6, 7, 8, 9, 10}

Size of given array n = 9

Actual no. of elements in array n+1 = 10 (1 to 10)

(1^2^4^5^6^7^8^9^10) ^ (1^2^3^4^5^6^7^8^9^10)

Given array elements ↓ Actual elements ↓

0 ^ 0 ^ 3 ^ 0 ^ 0 ^ 0 ^ 0 ^ 0 ^ 0 ^ 0

Missing number = 3

b) What is the worst case run time complexity of your suggested solution.

b) The time complexity for suggested solution in part (a) = O(n)

- only single traversal of the array is required
  - → for (int K=1 ; K<9 ; K++)
    XOR1 = XOR1 ^ A[K]
  - → for (int j=2; j<=10 ; j++)
    XOR2 = XOR2 ^ j

⇒ T(n) = O(n) + O(n) = O(n)

The space complexity for suggested solution in part (a) = O(1) as no extra space is required by the program to run/execute.

⇒ space complexity = O(1)

Ques 2 You are given an array of integers:

a) write a pseudo code to find all pairs of numbers whose sum is equal to a particular number.

a) To find all the pairs of numbers whose sum is equal to a particular number we will use the two-pointer approach.
To use the two pointer approach, we first

sort the array in ascending order.

input : Array A of integers, target sum, no. of elements n

PSEUDO CODE

```
int pairSumFind (int n, int A[], int target) {
    sort (A, A+n);          /* sorting the array */
    int i = 0;              /* Initializing the two pointer
    int j = n-1;               variables with the starting
                               and ending indices of the
                               array respectively */
    int countPair = 0;      /* Initializing the variable
                               that keeps count of no. of
                               the pairs with target sum */
    while (i<j) {           /* while loop that runs while
                               i<j */
        if ( A[i] + A[j] == target) {
            countPair ++;   /* Incrementing the counter
                               when a pair with
                               sum equal to target sum
                               is found */
            cout <<A[i] << "and" << A[j] << endl;
            /* Printing the pair with target sum */
            j--;            /* incrementing the left pointer
            i++;               and decrementing the right
        }                      pointer to find the next
                               pair of elements with
                               target sum */
        else if ( A[i] + A[j] > target) {  /* if the sum
                                              of elements is
            j--;                              greater than target sum then
        }                                     decrement the right pointer
        else {                                otherwise increment the left
            i++;                              pointer as the array is sorted
        }                                     in ascending order & right &
                                              left pointer point at comparison
```

```
}
return countPair;
}
```

to each other point at greater and smaller values in array and doing so will help in reducing & increasing the sum, that may result in the target sum finding */

/* countPair is returned with no. of pairs with target sum */

b) what is the worst case run time complexity of your suggested solution

b) The time complexity for worst case of the suggested solution in part (a) ⇒

Time complexity of sorting the array + Time complexity of two-pointer approach

sorting → in-built sort function ⇒ $T(n) = O(n \log n)$

Two pointer approach only comprises of traversing once through the array ⇒ $O(n)$

$$T(n) = O(n \log n) + O(n)$$

$$\boxed{T(n) = O(n \log n)} \quad \text{as } (n \log n > n)$$

The space complexity for the suggested solution in part (a) $= O(1)$ as no extra space is required by the program to run and execute successfully.

$$\boxed{\text{space complexity} = O(1)}$$

Ques 3 You are given an array of integers :

a) write a pseudocode to remove duplicates from your array

a) To remove duplicates from your array we are assuming that our array is unsorted. we will use Hash Table to find duplicate elements in the array.

PSEUDO CODE

```
input : Array A , number of elements n

create Hashtable M      /* create Hash Table to
                           keep storing elements*/

int answer[] ;  /* create an answer [] array
                   to store unique elements */
int K = 0;
for (K=0 ; K<n ; K++)   /* for loop to traverse
                           through all elements of
                           array A[] */
 {
     if (A[K] not in M) {   /* check before
                               inserting a new
                               element in hash
                               table if it is
                               present or not */

          answer .insert(A[K]   /* If current
                                   element is
                                   duplicate then
                                   ignore, else store
                                   it in answer []
                                   */
          M. insert (A[K])      /* Insert in hashtable
                                   if unique */

     }
 }

return answer           /* return answer
                           array answer [] */
```

b) What is the worst case run time complexity of your suggested solution

b) Time complexity to remove duplicates from array can be calculated as :

→ for (K=0 ; K<n; K++) /* Array Traversal */

→ if (A[K] not in M) /* HashTable search for element */

→ answer. insert (A[K]) /* Elements inserted into array answer[] and M. insert (A[K]) HashTable */

Time complexity = Array Traversal + HashTable search for element + elements inserted into array answer[] and HashTable

= O(n) + O(n) + O(n)

Time complexity T(n) = O(n)

Space complexity = HashTable storage + Answer array storage

= O(n) + O(n)

Space complexity = O(n)

Ques 4 You are given two sorted arrays :

a) write a pseudo code to find the median of the two sorted arrays (combined)

a) To find the median of the combined array made up of the two sorted arrays we will first merge them into a single sorted array as is done in the merge function of mergesort and then the median can be found out by accessing the $\left(\frac{m+n+1}{2}\right)^{th}$ element in the resultant array when the combined array has odd no. of elements, and by accessing the $\left(\frac{m+n}{2}\right)^{th}$ and $\left(\frac{m+n}{2}+1\right)^{th}$ elements & taking their average when the no. of elements in the resultant array is even, where m & n are no. of elements in the two arrays respectively.

PSEUDO CODE: (input : two sorted arrays)

```
int calculateMedian (int A1[], int A2[], int m,
                     int n) {
    int i = 0;        /* To keep track of till which
                        index A1 elements have
                        been merged in the
                        resultant array */
    int j = 0;
    int result[m+n];  /* To keep track of till which
                        index A2 array elements
                        have been merged in the
                        resultant array */
    int k = 0;

/* Declaring a new array which will
   store the resultant of the two merged
   sorted arrays, and a pointer to keep
```

track of till what index elements have been added in the resultant array.

```
while (i < m && j < n) {          /* while loop
    if (A1[i] < A2[j]) {          to make a
        result[k] = A1[i];        new array
        k++;                      from the two
        i++;                      sorted
    }                             array which
    else {                        is also
        result[k] = A2[j];        sorted */
        k++;
        j++;
    }

while (i < m) {                   /* while loop
    result[k] = A1[i];            to add the
    k++;                          left out elements
    i++;                          of the A1
}                                 array to
                                  the resultant
                                  array */

while (j < n) {
    result[k] = A2[j];            /* while loop to
    k++;                          add the left out/
    j++;                          remaining elements
}                                 of the A2 array
                                  to the resultant
                                  array */

if (m+n % 2 != 0) {
    return result[(m+n)/2];
}
else {
    return (result[(m+n)/2 - 1] + result[(m+n)/2]) / 2;
}
}
```

b) what is the worst case run time complexity of your suggested solution

b) The worst case run time complexity of the solution suggested in part (a) is $O(m+n)$ and if $n \gg m$ then $O(n)$, as the solution only consists of using three while loops to merge the two sorted arrays in a sorted form.

Ques 5 use one of the sorting algorithms to sort the following array input =

{ 4, 1, 2, 7, 10, 1, 2, 4, 4, 7, 1, 2, 1, 10, 1, 2, 4, 1, 2, 7
10, 1, 2 } ;

Show step by step what happens for the input (no pseudocode is required)

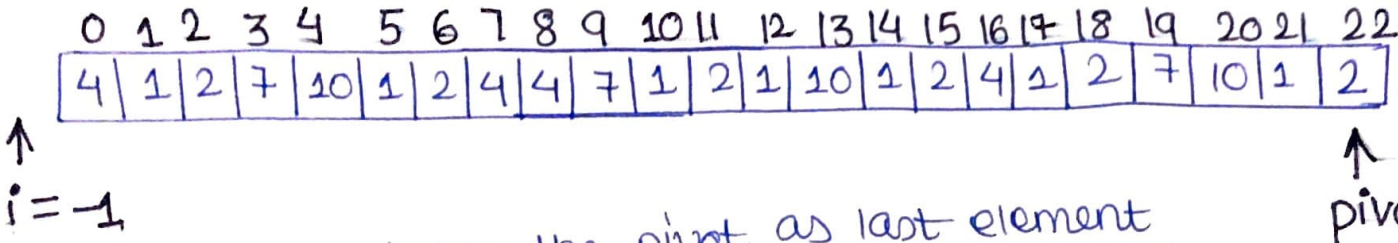
Sorting Algorithm used : Quicksort (it is an in-place sorting algorithm meaning it does not require extra space)

Quicksort algorithm involves yirstly choosing the pivot element about which partitioning is to be done. we will be choosing the last element in the array as the pivot element. It then involves rearranging the element in the array to place the pivot element at correct position such that on the left, elements are smaller than pivot, and on the right, elements are greater than pivot.

⇒ 1) Select the pivot element (last element of the array)

2) Rearrange the array to place at correct position and partition it such that
left subarray : elements ≤ pivot
right subarray : elements > pivot

3) Recursively sort the left and right subarrays upto the point where only single element is present in the sub-array.
we then obtain a sorted array.



we have chosen the pivot as last element now, we re-arrange the array, and to do so traverse from $j = 0$ to $pivot - 1 = 21$

→ i here is a pointer variable to keep track of the index till where elements smaller than pivot are present.

→ As we traverse through the array, we compare each element to pivot while $j \leq pivot-1$

- if $A[j] \leq$ pivot element
  - ⇒ i++ /* increment the i pointer and to reflect one more smaller than pivot element index in the array */
  - ⇒ swap (arr[i], arr[j]) and then swap to keep all smaller elements together at one side
- if $A[j] >$ pivot element
  - continue;

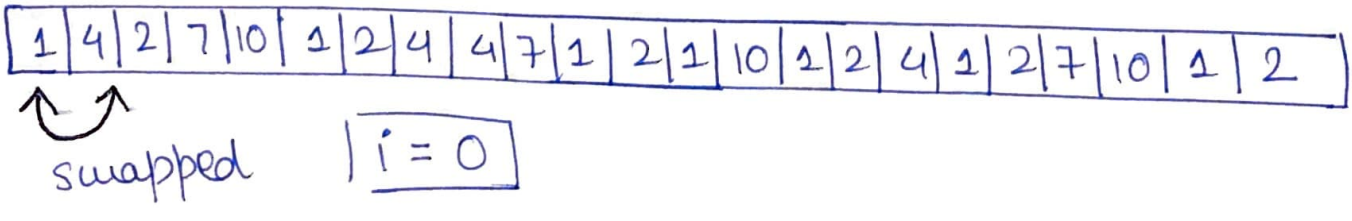
Then at last we swap the A[i+1] and pivot element to keep pivot element at correct position.

* $j = 0 \rightarrow 21$:

⇒ $j = 0$; $A[j] = 4$; $4 > 2$; continue

⇒ $j = 1$; $A[j] = 1$; $1 < 2$; i++ ; swap (A[i], A[j])

$i = -1 \Rightarrow i+1 \Rightarrow i = 0$



⇒ j=2 ; A[j] = 2 ; 2 ≤ 2 ; i++ ; swap (A[i], A[j])

i=0 → i++ ⇒ i=1

1	2	4	7	10	1	2	4	4	7	1	2	1	10	1	2	4	1	2	7	10	1	2

swapped

i = 1

⇒ j=3 ; A[j] = 7 ; 7 > 2 ; continue

⇒ j=4 ; A[j] = 10 ; 10 > 2 ; continue

⇒ j=5 ; A[j] = 1 ; 1 < 2 ; i++ ; swap (A[i] ; A[j])

i = 1 → i++ ⇒ i = 2

1	2	1	7	10	4	2	4	4	7	1	2	1	10	1	2	4	1	2	7	10	1	2

swapped

i = 2

And similarly continuing for other elements till j = 21 and then swapping in the pivot element at it's correct position in the array we obtain :

1	2	1	2	1	2	1	1	2	1	2	1	2	10	4	4	4	7	10	7	10	4	7

left subarray | partition (pivot) | right subarray

1	2	1	2	1	2	1	1	2	1	2	1	2	10	4	4	4	7	10	7	10	4	7

↳ rearrange and sort recursively | pivot | ↳ rearrange and sort recursively | pivot

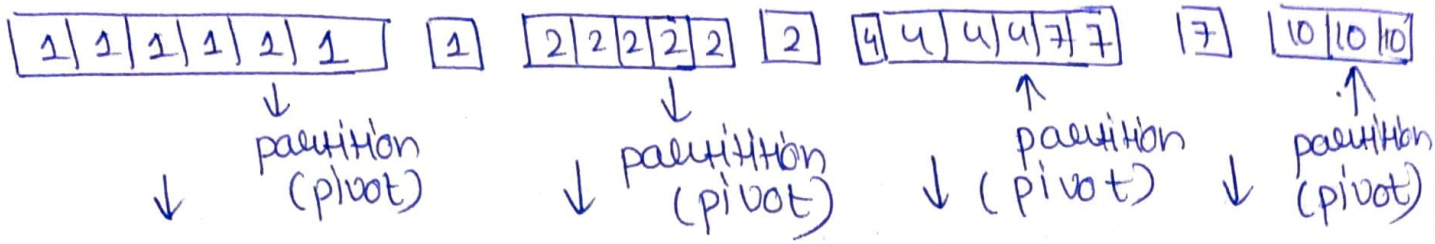
1	1	1	1	1	1	1	2	2	2	2	2	2	4	4	4	7	7	4	7	10	10	10

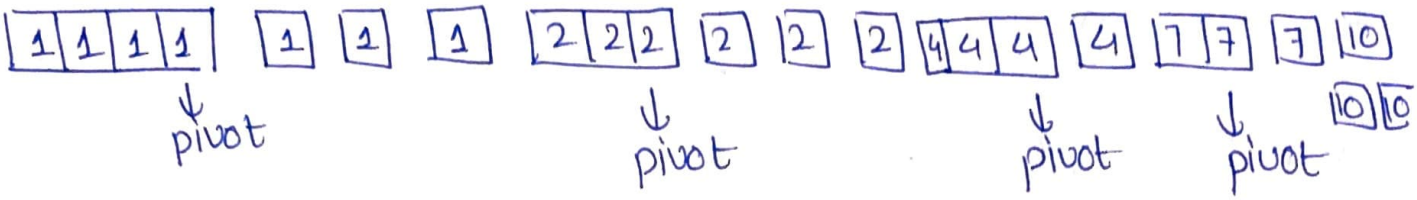
partition (pivot) | partition (pivot)

1	1	1	1	1	1	1	2	2	2	2	2	2	4	4	4	7	7	4	7	10	10	10

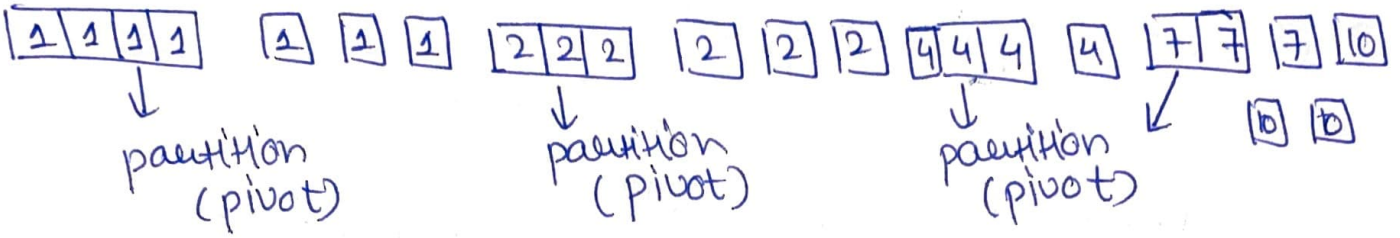
pivot | pivot | pivot | pivot

rearrange and sort recursively

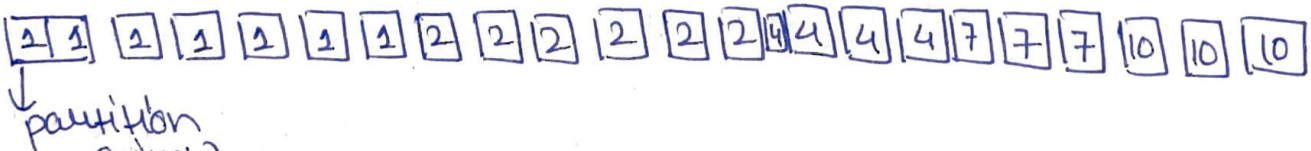


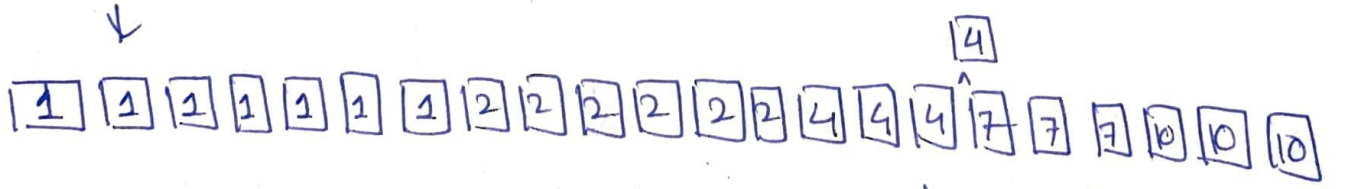


rearrange and sort recursively



rearrange and sort recursively





As the quicksort sorting algorithm is an in-place sorting algorithm, the final sorted array is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	1	1	1	1	1	1	2	2	2	2	2	2	4	4	4	4	7	7	7	10	10	10

Question 6 Answer the following questions :

a) when does the worst case of quick sort happen and what is the worst case run time complexity in terms of big O ?

a) The worst case time complexity of quicksort takes place when the pivot element is chosen is the last element of the array and the array is sorted in ascending order when we want it to be descending order and vice versa, and also when all the elements of the array are same. This all same elements and sorting results in partitioning of the array into sub-arrays for recursive calls with un-even or disproportional lengths with 1 element to the right and N-1 to the left each time, which results in many recursive calls. (already sorted in reverse order)

Quicksort is efficient when the pivot element chosen results in proportional division of the array into left & right subarray each call.

worst case run time complexity of quicksort is $O(n^2)$.

b) when does the best case of bubble sort happen and what is the best case run time complexity in terms of big O ?

b) The best case ~~complexity~~ of bubble sort will take place when the array is already sorted in the order that we desire and

therefore no swapping is to be performed and it just has to loop once through all the elements in the array to identify / recognise that the elements are in the correct desired sorted order.

The time complexity in the best case is $O(n)$.

c) what is the runtime complexity of Insertion sort in all 3 cases? Explain the situation which results in best, average and worst case complexity.

c) • The best case run time complexity for ~~quicksort~~ insertion sort takes place when the array provided is already sorted, in the order that we want. Then it just has to execute the outer for loop once to ~~recognise~~ recognise that the elements are already placed in the ~~correct order~~. It requires just looping through once in the array. Thus time complexity is a linear function

Best Case Time Complexity (insertion sort) : O(n)

• The average case run time complexity for insertion sort takes place when an unsorted array is provided, where for every element from n elements, comparison is made with the other elements out of n elements. Thus making the no. of comparisons in the array in order of n^2, and then inserting the elements at correct position.

Average case Time complexity (insertion sort) : $O(n^2)$

- The worst case run time complexity for insertion sort takes place when the array provided is sorted in the reverse order to what we want. In this case, every call for comparison & insert comprises of comparing with every element to it's left side, and this happens for all the elements of the array, which results of running two nested loops, and the time complexity comes out to be $O(n2)$ in worst case.