Name: Muhammad Hamza Khan

Roll: 21L-5654

Date: Day:

Homework #2

Question 1

We can sort Red, blue and white colours in a sorted array at most O(n) time /worst case. The algo is simple. We take variables red and white and initialize with start index of array. and blue with last index. ~~No~~

Now ~~insertion~~ sorting will take place in such way that red will stored first and white ~~inder~~ at next and then blue. If white and red are compared at same index, they will be incremented until they get at required position through swaping. Blue will be stored ~~at~~ starting from end index.

```
ColourSort(A, size){
    red = 0, white = 0, blue = size-1
while(white < blue)
{   if(A[white] == "red")
    { swap(A[red], A[white])
        white++
        red++
    }
else if(A[blue/white] == "blue)
{   swap(A[blue], A[white])
    blue--; // ~~e~~end to start
}
else { white++ }
```

Date: Day:

```
return A;
}
```

Question 2

```
mergesort( X[], Y[], sorted[], M, N)
    for(i=1 — N)
        left = 1
        right = M
        while ( right > left)
            { mid = (left + right)/2;
        }
        if X[mid] < Y[i]
for(i — M+N)
    if(j<M && k<N)
        if ( X[j] < Y[k]{
            sorted[i] = x[j];
                j++;
            }
        else { sorted[i] = Y[k]
                k++
            }
        i++;
    }
else if (j == m)
    { for ( k — m+n)
            sorted[i] = Y[k]
        }           i++;
else{for( i — m+n){
        sorted[i] = X[j]
            j++
    }
```

Date: Day: ①

Question 3

Pseudocode for finding rotations in array:

```
int findPivot(A, left, right){

    if right < left  return -1
    if right == left  return left

    mid = (left + right)/2

    if  mid > left && A[mid] < A[mid-1]
            return mid -1

    if  A[left] ≤ A[mid] {
            return findPivot(A, mid+1, right);
    return findPivot(A, left, mid-1);
}
```

~~Dry Run:~~

```
int countRotations(A, size)
    {
        Pivot = findPivot(A, 0, size-1);
        if (pivot)
            return ((pivot + 1) % size);
        else
            return 0;
    }
```

Date: Day:

In this code, there are two functions counting rotations and find pivot. The counting rotations takes the array and size and passes them to find pivot function.

Find Pivot function takes parameters as starting index (left), end index (right) and array. Then it compares if end < start no pivot, size of array is one, return left index..

Calculation of mid and then comparing value of mid, if mid of array is less than A[mid -1] return index of mid -1, and if A[left] ≤ A[mid] recursion through right else recursion through left.

~~[As specific case, if left, mid and right are same then recursion on both sides]~~

Dry Run:

arr = {7, 9, 11, 12, 15}

left = 0, right = 4, mid = 2

A[left] < A[mid]

7 < 11

↙ recursion of right

find Pivot(A, 3, 4) 12 (3) | 15 (4)

left = 3, right = 4, mid = 3

↙ recursion

find pivot (A, 4, 4) 15 (4)

15 (4) return 4;

Answer: 4

Date: Day:

Question 4

Majority Element

~(a)~

The majority element problem can be solved in O(nlgn) time by spliting array A into two half size arrays A1 and A2. Now, we will calculate count of an element ~~(in both arrays and)~~ target in left and right array depending on its position. Finally, we compare the count if it is greater than half size of array A. If count is greater, we return the element i.e majority element else, there will be no majority element.

~~ner~~ ~~Sort arrays into hal~~

```
Sort (array, left, right) {
    if (left == right)  return array[left];
    mid = (left + right)/2;
    left_maj = Sort (array, left, mid);          //A1
    right_maj = Sort (array, mid+1, right);      //A2

    if (merge_count (array, left, right, left_maj)
            > (right - left + 1)/2)
        return left_maj;

    if (merge_count (array, left, right, righ_maj)
            > (right - left + 1)/2)
        return right_maj;

    return -1;
}
```

Date: Day:

```
merge_count (array, left, right, element)
{
    count = 0
    for (i = left ---- i < right)
        if (array[i] == element)
            count++;
    return count;
}
```

Sort ( ) — O(logn)
merge_count ( ) — O(n)] → O(nlgn)

~ (b) ~

Yes, a linear time algo can be implemented for majority element problem by using divide and conquer approach.

```
findMajorityElement (A, left, right)
    if (left == right) return A[left]
  // divide the array into two parts
    left_maj = findMajorityElement (A, left, mid);
    right_maj = findMajorityElement (A, mid+1, right);

  // if left_maj == right_maj, Keep lef_maj
    if (left_maj == right_maj)
        return left_maj;
  // else we discard both
  // now we count occurances in linear time
    for (i = ~~right~~ left ---- right)
        if (~~A~~ A[i] == left_maj)
            lefcount++;
```

Date: Day:

```
else if (A[i] == right_maj)
        right_count ++;

// now we return the highest majority count element
if (left_count > right_count)
        return (left_count maj) left_maj
else if (
        return (right_count) right_maj
}
// we cant compare elements as > or <, so we
// only compare their counts occurances.
```

From above algo, we will be returning majority element. in $O(n)$ time because

$$O(\lg n) + O(n) \approx O(n).$$

Date: Day:

Question 5

(a)

Table for number of each comparison while executing HeapSort, MergeSort and QuickSort
Number is 10^6

HeapSort	MergeSort	QuickSort	i
39095792	18674356	12431188	1
78192242	3734600	25554792	2
117291076	56023011	38388803	3
156390156	74697064	50804786	4
195485506	93371162	62457604	5

averages

HeapSort = $\frac{586454772}{5} = 117290954.4$

MergeSort = $\frac{246500193}{5} = 49300038.6$

QuickSort = $\frac{189637173}{5} = 37927434.6$

(b)

Time taken:

i	HeapSort	MergeSort	QuickSort
1	1829.25	968.848	286.052
2	700.615	1033.63	576.756
3	663.19	651.097	292.361
4	657.26	705.765	793.649
5	936.909	702.309	425.627

$$\text{HeapSort} = \frac{4587.224}{5} = 917.4448 \text{ miliseconds}$$

$$\text{MergeSort} = \frac{4061.649}{5} = 812.3298 \text{ miliseconds}$$

$$\text{QuickSort} = \frac{2374.445}{5} = 474.889 \text{ miliseconds}$$

However, Both Heapsort, Mergesort ~~& Quick sort and Quick sort (best)~~ have $O(n \lg n)$ time complexity, & Quicksort has $O(n^2)$.

But, In this analysis, we can say that

Heapsort time > Mergesort > Quicksort.