ALGO assignment.

Date:

Name: Zain Al Abidin RollNO: 21L-6260

Q#1

```
int sort (int arr, int size){
    red, white = 0;
    blue = size-1;
    while (white <= blue){
        if (arr(white] == "red"){
            Swap (arr(red), arr(white));
            red += 1;
            white += 1;
        } else if ( arr [white] == "white"){
            white += 1;
        }
        else { swap( arr(white), arr(blue));
            blue -= 1;
        }
    }
    return arr;
}
```

O(n) is the worst case

O(1) if the array already sorted

Date:

Q#2

```
int   sorted_arr ( int x[] , int y [])
    int size_x = len [x] ; int size_y = len[y] ; int size_c = size_x-size_y ;
    int c [size_c] = { 0 } ;
    int index = 0 ;
    for (int i=0 ; i< size_y ; i++){
        index =   BST. bisect_left ( x , y[i]) ;
[redacted]

        for (int j= size_x + i ; j< index ; j--){
            x[j] = x [j-1] ;
        }
        x [index] = y [i] ;
    }
    for (int i=0 ; i< size_c ; i++){
        c[i] = x [i] ;
    }
return c;
```

Assumptions Made: We have already written the code for BST and the function bisect_left which helps in a value at the appropriate index to maintain a sorted BST.

Time complexity of outer most loop is O(N) and O (log(M)) for the inner loop therefore the combined time complexity will be

$O(N \log M)$

Date:

Q#3

```
int find_pivot (int arr[], int left , int right){
    if (right < left){ return -1; }
    if (right == left){ return left; }

    int mid = (left + right) / 2 ;

    if (mid < right && arr [mid] > arr [mid+1]) { return mid; }
    if (mid > left && arr [mid] < arr [mid - 1]) { return (mid-1); }.
    if (arr [left] <= arr [mid]) { return find_pivot (arr, mid+1, right);
    }

    return find_pivot (arr, mid+1, right);
}.

int count_rots (int arr [], int n){
    int index = find_pivot (arr, 0 , n-1);
    if (index == -1) { return 0; }.

    return ((index + 1) % n);
}
```

Date:

Q#4

(a)

```
int Sort (arr, left, right){
        if (left == right){
            return array[left];
        }
        int mid = (left + right);
        int left_m = sort (arr, left, mid);
        int right_m = Sort (arr, right, mid);
        if (merge_count (arr, left, right, right_m) > (right - left + 1/2))
        { return right_m; }
        else if (merge_count (arr, left, right, left_m) > (right - left + 1/2)) {
            return right_m;
        }
}
~~merge_count~~
int merge_count (arr, left, right, E) {
    int count = 0;
    for (int i = left; i < right; i++){
        if (arr[i] == E){
            count ++;
        }
    }
    return count;
}
```

In This algorithm we are Splitting an array into half and Calculating the count of Target element in both arrays if the count is greater than half size of the array then we return it as majority element. Time complexity of Sort function is O(lgn) and Count function O(n)

So total time complexity = O(nlgn)

Date:

(6)

```
int MajorityElement (arr, left, right){
        if (left == right) {
            return arr[left]; }
        left_m = MajorityElement (arr, left, mid);
        right_m = MajorityElement (arr, mid+1, right);
        if (left_m = right_m){
            return left_m; }

        for (int i=left ; i<right ; i++){
            if (arr[i] == left_m) {
                leftcount ++ ; }
            else if (arr[i]== right_m){
                rightcount ++ ; }

            if ( leftcount > rightcount) { return left_m; }
        else { ~~return right_m;~~
            return right_m; }
```

Same as (a) part, however a new count function introduced so now time complexity $\Rightarrow$ $O(\lg n) + O(n)$.