```
#include <stdio.h>

//INSERTION SORT
void insertionSort(int a[], int size){

	int i,j, k;

	for (i = 1; i < size; i++){

		k = a[i];

		j = i - 1;

		while(j >= 0 && a[j] > k){

			a[j+1] = a[j];

			j--;
		}

		a[j+1] = k;
	}
}

//QUICK SORT


// function to swap elements
void swap(int *a, int *b) {
```

```
  int t = *a;
  *a = *b;
  *b = t;
}

// function to find the partition position
int partition(int array[], int low, int high) {

  // select the rightmost element as pivot
  int pivot = array[high], j;

  // pointer for greater element
  int i = (low - 1);

  // traverse each element of the array
  // compare them with the pivot
  for (j = low; j < high; j++) {
    if (array[j] <= pivot) {

      // if element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;

      // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
  }

  // swap the pivot element with the greater element at i
```

```
  swap(&array[i + 1], &array[high]);

  // return the partition point
  return (i + 1);
}

void quickSort(int array[], int low, int high) {
  if (low < high) {

    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot
    int pi = partition(array, low, high);

    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);

    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}




//MergeSort

// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {
```

```
// Create L ‹ A[p..q] and M ‹ A[q+1..r]
int n1 = q - p + 1;
int n2 = r - q;
int i, j, k;

int L[n1], M[n2];

for ( i = 0; i < n1; i++)
  L[i] = arr[p + i];
for (j = 0; j < n2; j++)
  M[j] = arr[q + 1 + j];

// Maintain current index of sub-arrays and main array
i = 0;
j = 0;
k = p;

// Until we reach either end of either L or M, pick larger among
// elements L and M and place them in the correct position at A[p..r]
while (i < n1 && j < n2) {
  if (L[i] <= M[j]) {
    arr[k] = L[i];
    i++;
  } else {
    arr[k] = M[j];
    j++;
  }
  k++;
}
```

```
  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}

// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r) {
  if (l < r) {

    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
```

```
}



//Binary Search ITERATIVE


int binarySearchITERATIVE(int array[], int x, int low, int high) {
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (array[mid] == x)
      return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
      high = mid - 1;
  }

  return -1;
}

//Binary Search Recursive
int binarySearchRec(int array[], int x, int low, int high) {
  if (high >= low) {
    int mid = low + (high - low) / 2;
```

```
    // If found at mid, then return it
    if (array[mid] == x)
      return mid;

    // Search the left half
    if (array[mid] > x)
      return binarySearchRec(array, x, low, mid - 1);

    // Search the right half
    return binarySearchRec(array, x, mid + 1, high);
  }

  return -1;
}
//Array Yazdir
void ArrayYazdir(int a[], int size){

        int i;

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

                printf("%d",a[i]);

                if(i != size-1)
                        printf(",");
        }
}

int main(){
```

```
int n, i, key;

printf("Dizi boyutu n:");
scanf("%d", &n);
int arr[n];
int arrQ[n];
int arrM[n];
int arrSoru4[n];

for(i = 0; i < n; i++){
        arr[i] = rand()%101;
        arrQ[i] = arr[i];
        arrM[i] = arr[i];
        arrSoru4[i] = arr[i];
}


ArrayYazdir(arr, n);
printf("\n");
insertionSort(arr, n);
printf("\nInsertion:");
ArrayYazdir(arr,n);

printf("\n");
quickSort(arrQ, 0, n-1);
printf("\nQuick:");
ArrayYazdir(arrQ,n);
```

```
printf("\n");
mergeSort(arrM, 0, n-1);
printf("\nMerge:");
ArrayYazdir(arrM,n);

printf("\naranacak sayi:\n");
scanf("%d", &key);

//ARRAY SIRALI OLMALI O YUZDEN SIRALI
//ARRAYLERDEN BIRINI KULLANALIM

int res = binarySearchITERATIVE(arr,key,0,n-1);

if (res == -1)
printf("Not found");
else
printf("Element is found at index %d\n", res);


res = binarySearchRec(arr,key,0,n-1);

if (res == -1)
printf("Not found");
else
printf("Element is found at index %d", res);


printf("\n\n");
```

```
//Soru 3 siralama yapilan arrayi baska bir arraya tesrsten al.

int ters[n];
for(i = n-1; i >= 0; i--){
        ters[n - (i+1)] = arr[i];
}

ArrayYazdir(ters, n);

printf("\n\n");

//Soru 4

//En buyuk eleman
int max = 0;
int min = 100;
int tekrar[100] = {0};

for(i = 0; i < n; i++){
        int x = arrSoru4[i];
        tekrar[x]++;
        if(max < x)
                max = x;
        if (min > x)
                min = x;

}
```

```
	printf("\nmax = %d", max);
	printf("\nmin = %d", min);

	int enTekrar = 0;
	int enTekrarEleman = 0;
	for(i = 0; i < 100; i++){

		if(tekrar[i] > enTekrar){
			enTekrarEleman = i;
			enTekrar = tekrar[i];
		}

	}
	printf("\nen Tekrar = %d", enTekrarEleman);

	return 0;
}
```