# Given a sorted array of positive integers, design an algorithm and implement it using a program to find three indices i, j, k such that arr[i] + arr[j] = arr[k].

**ALGORITHM**

Start

1. Read input n from the user.
2. Create an array of n integers named arr. 3.Read n integers into the arr array.
3. Initialize found by false
4. Traverse the arrayby k from n-1 down to 0:
   1. Initialize variables i=0,j=k-1
   2. While i< j:
5. If arr[i] + arr[j] equals arr[k]

print i+1, j+1, and k+1 and set found to true.

1. If arr[i] + arr[j] is less than arr[k]

increment i.

1. If arr[i] + arr[j] is greater than arr[k] decrement j.
2. If found is still false, print "no such indices found". Stop

# SOURCE CODE

#include <iostream> using namespace std; int fun()

{

int n,k,i; cin>>n;

int arr[n]; for(i=0;i<n;i++)

{

cin>>arr[i];

}

bool found =false; for(int k=n-1;k>=0;k--)

{

int i=0,j=k-1; while(i<j)

{

if(arr[i]+arr[j]==arr[k])

{

found=true; cout<<i+1<<","<<j+1<<","<<k+1<<endl; i++;

j--;

}

else if(arr[i]+arr[j]<arr[k])

{

i++;

}

else{

}

}

}

j--;

if(!found)

cout<<"no such indices found"<<endl;

}

int main()

{

int x; cin>>x;

for(int i=0;i<x;i++)

{

fun();

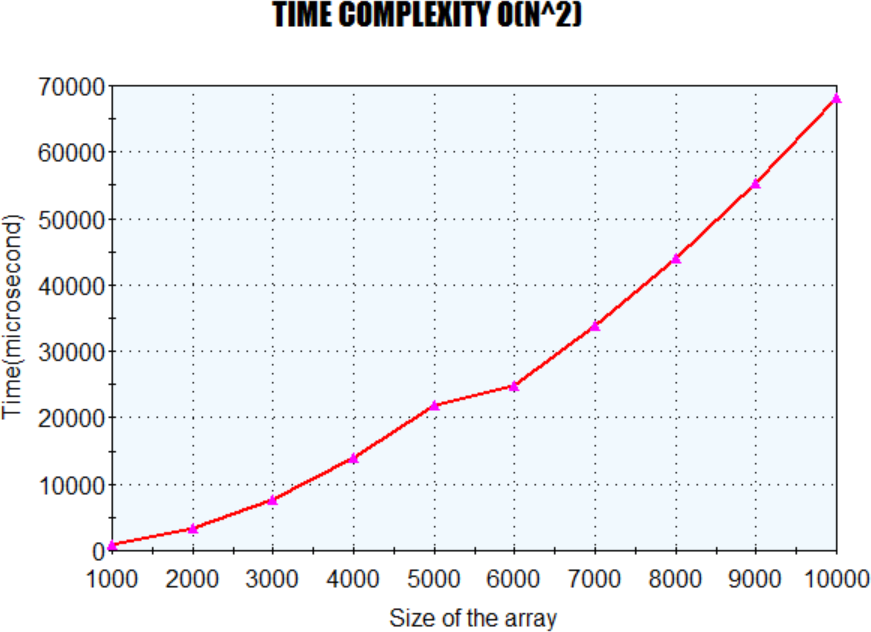
}

}

# OUTPUT

****

**RUN TIME COMPLEXITY GRAPH**



# 2. Given an unsorted array of integers, design an algorithm and implement it using a program to sort an array of elements by dividing the array into two subarrays and combining these subarrays after sorting each one of them. Your program should also find number of comparisons and inversions during sorting the array.

**ALGORITHM**

1. Start
2. Define a function mergeSort(arr) that takes an array of integers arr as an argument.
3. If the length of arr is less than or equal to 1, return arr.
4. Divide the array into two halves by finding the middle index.
5. Recursively call mergeSort on the left half of the array.
6. Recursively call mergeSort on the right half of the array.
7. Merge the two sorted halves of the array into a single sorted array. To do this, initialize two pointers i and j to the beginning of the left and right halves, respectively.
8. Create an empty array result. Iterate over both halves of the array, comparing the elements at i and j and adding the smaller element to result.
9. Increment the pointer of the element that was added to result.
10. If one pointer reaches the end of its half of the array, add the remaining elements of the other half to result.
11. Return the sorted array result.
12. Stop.

# SOURCE CODE

#include <iostream>

int merge(int arr[],int l,int m,int r,int \*count)

{

int n1=m-l+1; int n2=r-m;

int arr1[n1],arr2[n2]; for(int i=0;i<n1;i++) arr1[i]=arr[l+i]; for(int j=0;j<n2;j++)

arr2[j]=arr[m+1+j];

int i=0,j=0,k=l; while(i<n1 && j<n2)

{

if(arr1[i]<=arr2[j])

{

(\*count)++; arr[k]=arr1[i]; i++;

}

else{

(\*count)++; arr[k]=arr2[j]; j++;

}

k++;

}

while(i<n1)

{

arr[k]=arr1[i]; i++;

k++;

}

while(j<n2)

{

arr[k]=arr2[j]; j++;

k++;

}

}

void mergesort(int arr[],int l,int r,int \*count)

{

if(l<r)

{

int m=l+(r-l)/2; mergesort(arr,l,m,count); mergesort(arr,m+1,r,count); merge(arr,l,m,r,count);

}

}

using namespace std; int fun()

{

int n,k,i; cin>>n; int arr[n];

for(i=0;i<n;i++)

{

cin>>arr[i];

}

int count=0; mergesort(arr,0,n-1,&count); for(int i=0;i<n;i++)

cout<<arr[i]<<" "; cout<<endl;

cout << "Comparisions : " << count << endl;

}

int main()

{

int x; cin>>x;

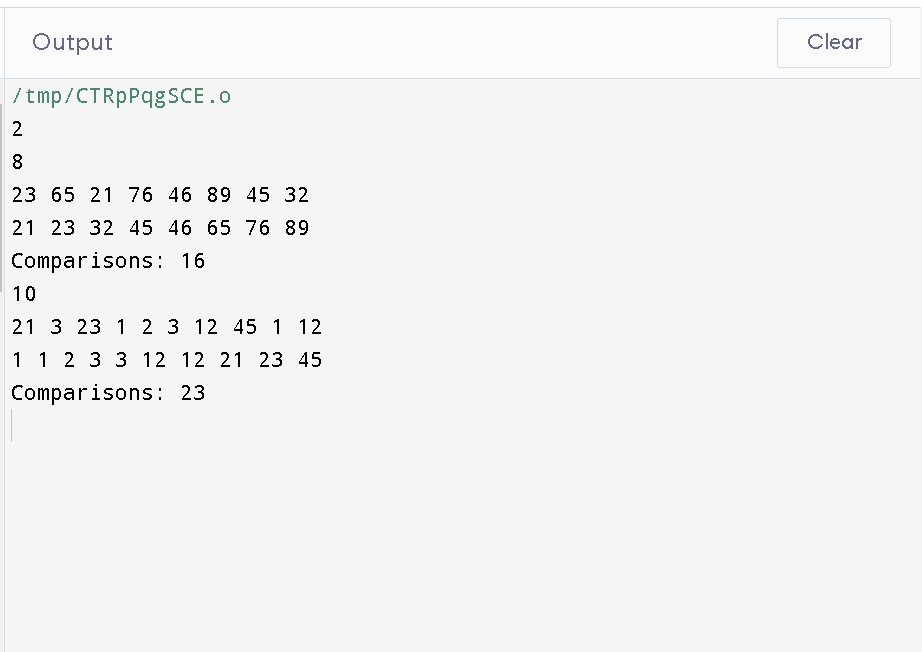
for(int i=0;i<x;i++)

{

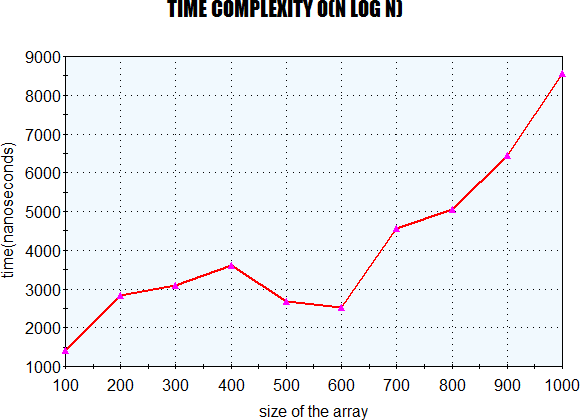
fun();

}

# OUTPUT

****

**RUN TIME COMPLEXITY GRAPH**



# 3 . Given an unsorted array of integers, design an algorithm and implement it using a program to sort an array of elements by partitioning the array into two subarrays based on a pivot element such that one of the sub array holds values smaller than the pivot element while another subarray holds values greater than the pivot element. Pivot element should be selected randomly from the array. Your program should also find number of comparisons and swaps required for sorting the array.

**ALGORITHM**

1. Start
2. Define a function quickSort(arr, low, high) that takes three arguments: the array of integers arr, and the low and high indices to sort within.
3. If low is greater than or equal to high, return.
4. Choose a random element in the subarray between low and high as the pivot.
5. Partition the subarray around the pivot by iterating over the array and moving elements that are less than or equal to the pivot to its left, and elements that are greater than the pivot to its right. Keep track of the index of the pivot in the array.
6. Recursively call quickSort on the left subarray (elements to the left of the pivot) with the new high index being pivot-1.
7. Recursively call quickSort on the right subarray (elements to the right of the pivot) with the new low index being pivot+1.
8. Stop.

# SOURCE CODE

#include <iostream> #include <cmath> using namespace std;

int partition(int arr[], int left, int right, int \*comparison, int \*swaps)

{

int random = left + rand() % (right - left + 1); swap(arr[random], arr[right]);

int pivot = arr[right]; int i = left - 1, j = left; while (j < right)

{

(\*comparison)++; if (arr[j] <=pivot)

{

i++;

swap(arr[i], arr[j]); (\*swaps)++;

}

j++;

}

i++;

swap(arr[i], arr[j]); (\*swaps)++; return i;

}

void quickSort(int arr[], int start, int end, int \*comparison, int \*swaps)

{

if (start < end)

{

int index = partition(arr, start, end, &(\*comparison), &(\*swaps)); quickSort(arr, start, index - 1, &(\*comparison), &(\*swaps)); quickSort(arr, index + 1, end, &(\*comparison), &(\*swaps));

}

}

int main()

{

int n, comparison = 0, swap = 0; cout << "enter no elements:"; cin >> n;

int arr[n];

for (int i = 0; i < n; i++)

{

cin >> arr[i];

}

quickSort(arr, 0, n - 1, &comparison, &swap); for (int i = 0; i < n; i++)

{

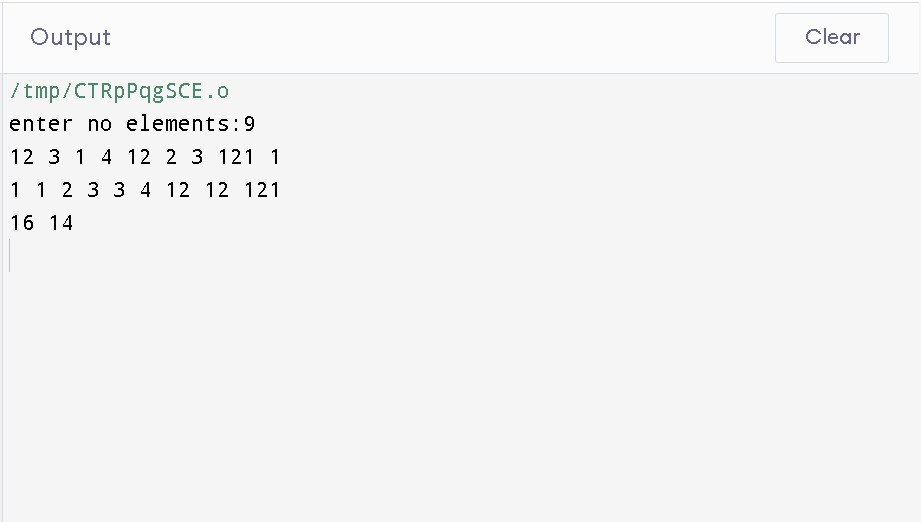
cout << arr[i] << " ";

}

cout << endl;

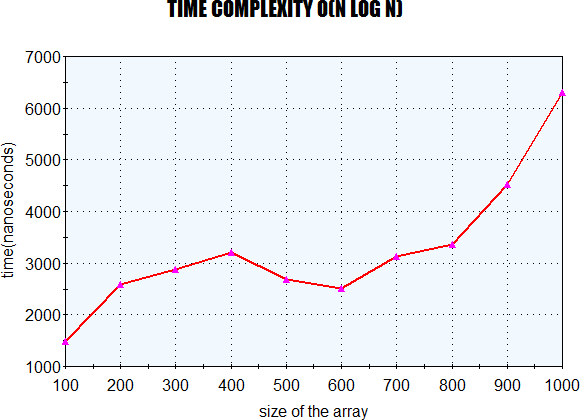
cout << comparison << " " << swap << endl; return 0;

}



# OUTPUT

**RUN TIME COMPLEXITY GRAPH**



# 4. Given an unsorted array of integers, design an algorithm and implement it using a program to find Kth smallest or largest element in the array. (Worst case Time Complexity = O(n))

**ALGORITHM**

1. Start
2. Define a function kthSmallest(arr, n, k) that takes three arguments: the unsorted array of integers arr, its size n, and the value of k.
3. Initialize two variables, left and right, to 0 and n-1 respectively.
4. While low is less than or equal to right, do the following:
5. Choose a pivot element, pivot, by selecting a random element between left and right.
6. Partition the array around the pivot such that all elements less than or equal to the pivot are on its left and all elements greater than the pivot are on its right. Let the index of the pivot after partitioning be pivotIndex.
7. If pivotIndex is equal to k-1, return the element at index pivotIndex as the kth smallest element.
8. If pivotIndex is less than k-1, update left to pivotIndex+1 and repeat step 4.
9. If pivotIndex is greater than k-1, update high to pivotIndex-1 and repeat step 4.
10. If the loop in step 4 completes without finding the kth smallest element, return -1 to indicate that such an element does not exist in the array.
11. Stop.

# SOURCE CODE

#include <iostream> using namespace std;

int partition(int arr[],int left,int right)

{

int x=arr[right],i=left; for(int j=left;j<right;j++)

{

if(arr[j]<=x)

{

swap(arr[i],arr[j]); i++;

}

}

swap(arr[i],arr[right]); return i;

}

int kthsmallest(int arr[],int left,int right,int k)

{

int pivot=partition(arr,left,right); if(pivot==k)

return arr[pivot]; else if(pivot<k)

return kthsmallest(arr,pivot+1,right,k); else

return kthsmallest(arr,left,pivot-1,k);

}

int fun()

{

int n,k,i; cin>>n; int arr[n];

for(i=0;i<n;i++)

{

cin>>arr[i];

}

cin>>k; cout<<kthsmallest(arr,0,n-1,k-1);

// kth smallest element in the array.

}

int main()

{

int x; cin>>x;

for(int i=0;i<x;i++)

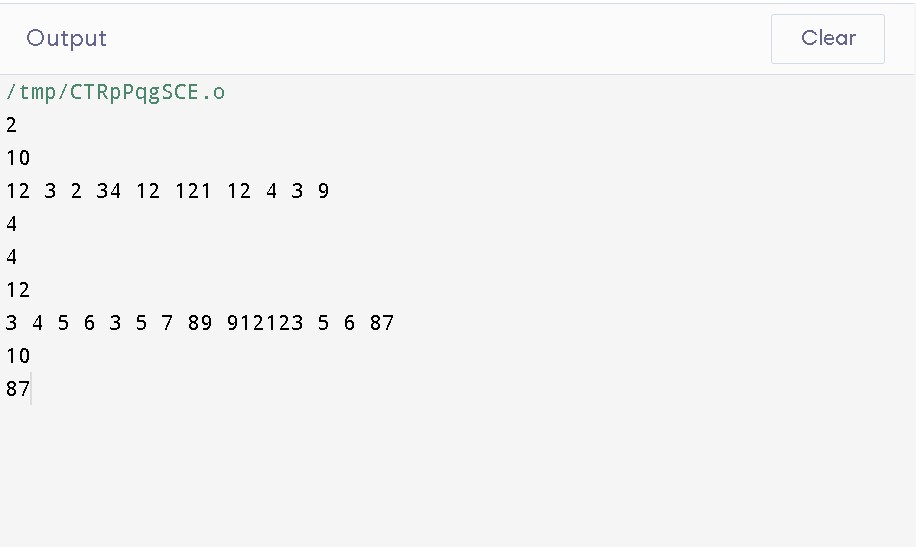
{

fun();

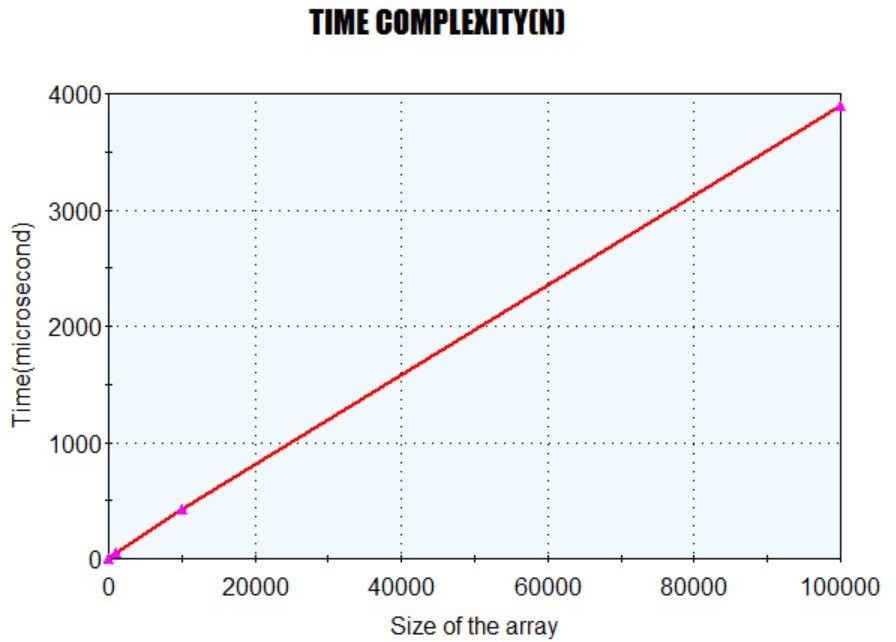
}

}

# OUTPUT



**RUN TIME COMPLEXITY GRAPH**



# 5 . Given an unsorted array of alphabets containing duplicate elements. Design an algorithm and implement it using a program to find which alphabet has maximum number of occurrences and print it. (Time Complexity = O(n)).

**ALGORITHM**

1. Start
2. Initialize an array of characters to count the frequency of the alphabets.
3. For loop from 0 to n-1
4. Increment the count at the arr[i]th location of the character array.
5. Initialize the variable c with -1 and ch.
6. For loop from 0 to n-1
7. If the frequency of the character is more then 1 and more them c, then update ch with ith character and update c with the frequency of this ith character of the array.
8. After the loop, if c is still -1, display "No Duplicates Present”
9. Else display ch and c.
10. Stop.

# SOURCE CODE

#include <iostream> using namespace std; void fun()

{

int n,k,i; cin>>n; char arr[n];

for(i=0;i<n;i++)

{

cin>>arr[i];

}

int count[126]={0}; for(int i=0;i<n;i++)

{

count[arr[i]]++;

}

int c=-1; char ch;

for(int i=0;i<n;i++)

{

if(count[arr[i]]>1 &&count[arr[i]]>c)

{

ch=arr[i]; c=count[arr[i]];

}

}

if(c==-1)

{

cout<<"No Duplicates Present"<<endl;

}

else

{

cout<<ch<<"-"<<c<<endl;

}

}

int main()

{

int x; cin>>x;

for(int i=0;i<x;i++)

{

fun();

}

# OUTPUT



**RUN TIME COMPLEXITY GRAPH**

