Algorithm task

Dominator

---

1) First approach in c (for loop approach)

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n; scanf("%d",&n);
    int arr[n];
    for(int i=0;i<n;i++){
        scanf("%d",&arr[i]);
    }
    int count1 = 0 , count2 = 0,dom = arr[0] ;

    for(int i=0;i<n;i++){
            int e = arr[i];
            count1=0;
        for(int j=0;j<n;j++){
            if(arr[j]==e) count1++;
        }
    if(count2<count1){
            count2 = count1;
            dom = arr[i];
        }
    }
//    printf("%d\n",count2);
//    printf("%d\n",dom);
if(count2 <= n/2) printf("-1");
else{
    for(int i=0;i<n;i++){
        if(arr[i]==dom) printf("%d ",i);
    }
}
    return 0;
}
```

Note : this approach is the worst in terms of time complexity

As it has a time complexity of O (n^2)

$T(n) \approx n^2 + 3n + 4 + c$

Pseudocode :

```
For( i=0 to n-1 ){
Elem ← arr[i];      //variables/array scanned from user :
Count_in ←0;                  // n, arr[];
    For( j=0 to n-1){         //variables initialized :
    If( arr[j] = Elem)        //count_in ←0,count_fin←0
        Count_in +=1;         //Dominator ←arr[0]
    }
  If( count_fin<count_in ){
  Count_fin ←count_in;
  Dominator ←arr[i]
  }
}
```

If(count_fin <= (n/2))

Print(-1);

Else{

For(i=0 to n-1){

If(arr[i] = Dominator)

Print(i);

}

}

---

2) Second approach in c(frequency arrays)

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int x; scanf("%d",&x);
    int arr[x];
    int freq[100000]={0};
    for(int i=0;i<x;i++){
        scanf("%d",&arr[i]);
        freq[arr[i]]++;
    }
    int maxe = arr[0] , maxx = 0;

    for(int i=0;i<x;i++){
    if(maxx<freq[arr[i]]){
        maxx = freq[arr[i]];
        maxe = arr[i];
        }
    }
    if(maxx<=(x/2))
        printf("-1");
```

```
    else{
    for(int i=0;i<x;i++){
    if(arr[i]==maxe){
        printf("%d ",i);
    }
        }
    }
    return 0;
}
```

Note : this approach is less stable when dealing with numbers more than 100000 and negative numbers but has less time complexity than the previous one O(n)

T(n) ≈ 3n + 3 + c

Pseudocode :

//after scanning number of elements and array elements from the user , we initialized an array called freq[] and initialized its elements with zeros.

//this algorithm uses elements of the original array as an index of the freq[] array

//we initialzed elements : maxE←arr[0] and

max ←0;

```
For( i=0 to n-1 ){
Freq[arr[i]]++;
}
For( i=0 to n-1 ){
If( max<freq[ arr[i] ] ){
max ←freq[arr[i]];
maxe ←arr[i];
}
}
If( max <=(n/2))
        Printf( -1 );
Else{
For( i=0 to n-1 ){
If(arr[i] = maxe){
print( i );
}
```

}

}

---

3) Third approach in c++ (most optimal)

```
#include <iostream>
#include <bits/stdc++.h>

using namespace std;

int main()
{
    unordered_map<int,int> mp;
    int x; cin>>x;
    int arr[x];
    for(int i=0;i<x;i++){
        cin>>arr[i];
        mp[arr[i]]++;
    }
    int mx= mp[arr[0]];
    int maxe = arr[0];
     for(int i=0;i<x;i++){
        if(mx<mp[arr[i]]){
            mx = mp[arr[i]];
            maxe = arr[i];
        }
    }
//     cout<<mx<<"\n"<<maxe<<"\n";
    if(mx <=(x/2)){
        cout<<-1;
    }else{
        for(int i=0;i<x;i++){
            if(arr[i]==maxe) cout<<i<<" ";
        }
    }

    return 0;
}
```

This approach we used unordered map and this code has O(n) also

And a T(n) ≈ 3n + 3 + c

But this code is able to handle larger input values with better time complexity .

Pseudocode :

// initialize an unordered map from the STL in c++

//unordered_map<int,int> mp;

//we will use the same method as the frequency array but inserting and counting will be more efficient as we can handle negative and positive numbers

[-2147483648 , 2147483647]

For(i=0 to n-1)

{

mp[arr[i]]++;

}

```
For(i=0 to n-1){
If(max < mp[arr[i]]){
max = mp[arr[i]];
maxe = arr[i];
}
}
If(max<=(n/2)){
Print( -1 );
}else{
For(i=0 to n-1){
If(arr[i] = maxe)
        Print (i);
}
}
```

4)fourth and final approach (recursive)

```
#include <bits/stdc++.h>

using namespace std;

int find_candidate(int A[], int size)
{
    int candidate = A[0];
    int count = 1;

    for (int i = 1; i < size; i++)
    {
        if (A[i] == candidate)
        {
            count++;
        }
        else
        {
            count--;
            if (count == 0)
            {
                candidate = A[i];
                count = 1;
            }
        }
    }

    return candidate;
}

int count_occurrences(int A[], int size, int candidate)
{
    int count = 0;
    for (int i = 0; i < size; i++)
    {
        if (A[i] == candidate)
        {
            count++;
        }
    }
    return count;
```

```c
}

int find_dominator_index(int A[], int start, int end)
{
    if (start == end)
    {
        return start;
    }

    int mid = start + (end - start) / 2;

    int left_dominator = find_dominator_index(A, start, mid);
    int right_dominator = find_dominator_index(A, mid + 1, end);

    if (left_dominator == right_dominator)
    {
        return left_dominator;
    }

    int left_candidate = A[left_dominator];
    int right_candidate = A[right_dominator];

    int left_count = count_occurrences(A, end - start + 1, left_candidate);
    int right_count = count_occurrences(A, end - start + 1, right_candidate);

    if (left_count > (mid - start + 1) / 2)
    {
        return left_dominator;
    }
    else if (right_count > (end - mid) / 2)
    {
        return right_dominator;
    }
    else
    {
        return -1;
    }
}

int main()
{
    int A[] = {3, 4, 3, 2, 3, -1, 3, 3};
    int size = sizeof(A) / sizeof(A[0]);
    int result = find_dominator_index(A, 0, size - 1);
    if (result != -1)
```

```
    {
        cout << "Dominator index: " << result << endl;
    }
    else
    {
        cout << "No dominator found." << endl;
    }
    return 0;
}
```

In this approach we used a recursive divide and conquer algorithm (binary search like) to find the index of the dominator

This algorithm has a time complexity of O(nlogn)

Pseudocode :

Find_candidate(int Arr[],size){

Candidate ←arr[0]

Count ←1

For(i=1 to size){

If(arr[i]=candidate) count +=1;

Else{

Count-=1;

```
If(count =0)

{

Candidate = Arr[i];

Count = 1;

}

}

}

Return candidate;

}

int count_occurrences(A,size, candidate) {

   count ←0


   for (i = 0 to size – 1) {

      if A[i] = candidate {

         count ← count + 1

      }

   }
```

```
    return count
}
int find_dominator_index(A, start, end) {
    if start == end {
        return start
    }

    mid <-- start + (end - start) / 2

    left_dominator <-- find_dominator_index(A, start, mid)
    right_dominator <-- find_dominator_index(A, mid + 1,
end)

    if left_dominator == right_dominator {
        return left_dominator
    }
```

```
    left_candidate <-- A[left_dominator]
    right_candidate <-- A[right_dominator]

    left_count <-- count_occurrences(A, left_candidate)
    right_count <-- count_occurrences(A, right_candidate)

    if left_count > (mid - start + 1) / 2 {
        return left_dominator
    } else if right_count > (end - mid) / 2 {
        return right_dominator
    } else {
        return -1
} }
//main function with array declaration and function
calling
```

Points of comparison	Algorithm 1 in c	Algorithm 2 in c	Algorithm 3 in c++	Algorithm 4 in C++
Time complexity	O (n^2)	O (n)	O (n)	O (nlogn)
Accuracy	Pretty accurate except for the time complexity part	Not accurate with edge cases	Most accurate one	Accurate enough but only returns 1 index
Recurrence relation	**$T(n) \approx n^2 + 3n + 4 + c$**	**$T(n) \approx 3n + 3 + c$**	**$T(n) \approx 3n + 3 + c$**	**$T(n) \approx 2(t/2)+n+c$**