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)

```c
#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 and final approach in c++ (most optimal)

```cpp
#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);
}
}
```

```c
#include <stdio.h>

void selectionSort(int A[], int N) {
    for (int i = 0; i < N - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < N; j++) {
            if (A[j] < A[minIndex]) {
                minIndex = j;
            }
        }
        int temp = A[i];
        A[i] = A[minIndex];
        A[minIndex] = temp;
    }
}


int findDominator(int A[], int N) {
    selectionSort(A, N);
    int count = 1;
    int dominator = A[0];
    int maxCount = 1;

    for (int i = 1; i < N; i++) {
        if (A[i] == A[i - 1]) {
            count++;
        } else {
            count = 1;
        }
        if (count > maxCount) {
            maxCount = count;
            dominator = A[i];
        }
    }

    if (maxCount > N / 2) {
        return dominator;
    } else {
        return -1;
    }
}
```

```c
int main() {
    int N;
    scanf("%d",&N);
    int A[N];
    for(int i=0;i<N;i++)
        scanf("%d",&A[i]);

    int dominator = findDominator(A, N);
    if (dominator != -1) {
        printf("The dominator is %d\n", dominator);
//        for(int i=0;i<N;i++)
//            if(A[i]==dominator) printf("%d ",i);
    } else {
        printf("Array does not have a dominator\n");
        printf("-1\n");
    }
    return 0;
}
```

Pseudocode :

```
selectionSort(A, N):
    for i= 0 to N - 2:
        minIndex = i
        for j = i + 1 to N - 1:
            if A[j] < A[minIndex]:
                minIndex = j;
        swap A[i] with A[minIndex];
```

```
function findDominator(A, N):
    selectionSort(A, N);
    count ←1;
    dominator ← [0];
    maxCount← 1;

    for i from 1 to N - 1:
        if A[i] = A[i - 1]:
            count++;
        else:
            count ← 1;
        if count > maxCount:
            maxCount ← count;
            dominator ← A[i];

    if maxCount > N / 2:
        return dominator;
```

```
    else:
        return -1;
main():
    dominator← findDominator(A, N)
    if dominator != -1:
        print"The dominator is " + dominator
            else:
        print "Array does not have a dominator"
        print "-1"
```

$T(n) \approx n^2 + 2n + 5 + c$

In this approach we used a sorting algorithm (selection sort) then we iterated through the array to check for number of occurrences for each element .

This algorithm has a time complexity of $O(n^2)$

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 (n^2)
Accuracy	Pretty accurate except for the time complexity part	Not accurate with edge cases	Most accurate one	Time complexity Isn't the best
Recurrence relation	**$T(n) \approx n^2 + 3n + 4 + c$**	**$T(n) \approx 3n + 3 + c$**	**$T(n) \approx 3n + 3 + c$**	**$T(n) \approx n^2 + 2n + 5 + c$**