Given an array of of size n and a number k, find all elements that appear more than n/k times

Given an array of size n, find all elements in array that appear more than n/k times. For example, if the input arrays is {3, 1, 2, 2, 1, 2, 3, 3} and k is 4, then the output should be [2, 3]. Note that size of array is 8 (or n = 8), so we need to find all elements that appear more than 2 (or 8/4) times. There are two elements that appear more than two times, 2 and 3.

A **simple method** is to pick all elements one by one. For every picked element, count its occurrences by traversing the array, if count becomes more than n/k, then print the element. Time Complexity of this method would be O(n^2).

A better solution is to **use sorting**. First, sort all elements using a O(nLogn) algorithm. Once the array is sorted, we can find all required elements in a linear scan of array. So overall time complexity of this method is O(nLogn) + O(n) which is O(nLogn).

Following is an interesting **O(nk) solution:**
We can solve the above problem in O(nk) time using O(k-1) extra space. Note that there can never be more than k-1 elements in output (Why?). There are mainly three steps in this algorithm.

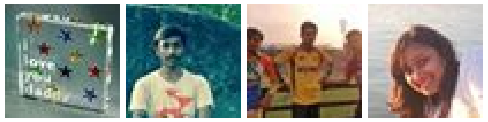
1) Create a temporary array of size (k-1) to store elements and their counts (The output elements are going to be among these k-1 elements). Following is structure of temporary array elements.

```
struct eleCount {
    int element;
    int count;
};
```

Google™ Custom Search

GeeksforGeeks

53,519 people like GeeksforGeeks.



Interview Experiences
Advanced Data Structures
Dynamic Programming
Greedy Algorithms
Backtracking
Pattern Searching
Divide & Conquer
Mathematical Algorithms
Recursion

```
struct eleCount temp[];
```

This step takes O(k) time.

2) Traverse through the input array and update temp[] (add/remove an element or increase/decrease count) for every traversed element. The array temp[] stores potential (k-1) candidates at every step. This step takes O(nk) time.

3) Iterate through final (k-1) potential candidates (stored in temp[]). or every element, check if it actually has count more than n/k. This step takes O(nk) time.

The main step is step 2, how to maintain (k-1) potential candidates at every point? The steps used in step 2 are like famous game: Tetris. We treat each number as a piece in Tetris, which falls down in our temporary array temp[]. Our task is to try to keep the same number stacked on the same column (count in temporary array is incremented).

```
Consider k = 4, n = 9
Given array: 3 1 2 2 2 1 4 3 3

i = 0
         3 _ _
temp[] has one element, 3 with count 1

i = 1
         3 1 _
temp[] has two elements, 3 and 1 with
counts 1 and 1 respectively

i = 2
         3 1 2
temp[] has three elements, 3, 1 and 2 with
counts as 1, 1 and 1 respectively.

i = 3
         - - 2
         3 1 2
temp[] has three elements, 3, 1 and 2 with
```

Geometric Algorithms

Popular Posts

All permutations of a given string
Memory Layout of C Programs
Understanding “extern” keyword in C
Median of two sorted arrays
Tree traversal without recursion and without stack!
Structure Member Alignment, Padding and Data Packing
Intersection point of two Linked Lists
Lowest Common Ancestor in a BST.
Check if a binary tree is BST or not
Sorted Linked List to Balanced BST

```
counts as 1, 1 and 2 respectively.

i = 4
         - - 2
         - - 2
         3 1 2
temp[] has three elements, 3, 1 and 2 with
counts as 1, 1 and 3 respectively.

i = 5
         - - 2
         - 1 2
         3 1 2
temp[] has three elements, 3, 1 and 2 with
counts as 1, 2 and 3 respectively.
```

Now the question arises, what to do when temp[] is full and we see a new element – we remove the bottom row from stacks of elements, i.e., we decrease count of every element by 1 in temp[]. We ignore the current element.

```
i = 6
         - - 2
         - 1 2
temp[] has two elements, 1 and 2 with
counts as 1 and 2 respectively.

i = 7
           - 2
         3 1 2
temp[] has three elements, 3, 1 and 2 with
counts as 1, 1 and 2 respectively.

i = 8
         3 - 2
         3 1 2
```

```
temp[] has three elements, 3, 1 and 2 with
counts as 2, 1 and 2 respectively.
```

Finally, we have at most k-1 numbers in temp[]. The elements in temp are {3, 1, 2}. Note that the counts in temp[] are useless now, the counts were needed only in step 2. Now we need to check whether the actual counts of elements in temp[] are more than n/k (9/4) or not. The elements 3 and 2 have counts more than 9/4. So we print 3 and 2.

Note that the algorithm doesn't miss any output element. There can be two possibilities, many occurrences are together or spread across the array. If occurrences are together, then count will be high and won't become 0. If occurrences are spread, then the element would come again in temp[]. Following is C++ implementation of above algorithm.

```
// A C++ program to print elements with count more than n/k
#include<iostream>
using namespace std;

// A structure to store an element and its current count
struct eleCount
{
    int e;  // Element
    int c;  // Count
};

// Prints elements with more than n/k occurrences in arr[] of
// size n. If there are no such elements, then it prints nothing.
void moreThanNdK(int arr[], int n, int k)
{
    // k must be greater than 1 to get some output
    if (k < 2)
       return;

    /* Step 1: Create a temporary array (contains element
       and count) of size k-1. Initialize count of all
       elements as 0 */
    struct eleCount temp[k-1];
    for (int i=0; i<k-1; i++)
        temp[i].c = 0;

    /* Step 2: Process all elements of input array */
    for (int i = 0; i < n; i++)
    {
        int j;
        /* If arr[i] is already present in
           the element count array, then increment its count */
        for (j=0; j<k-1; j++)
        {
            if (temp[j].e == arr[i])
            {
                 temp[j].c += 1;
                 break;
            }
        }

        /* If arr[i] is not present in temp[] */
        if (j == k-1)
        {
            int l;

            /* If there is position available in temp[], then place
              arr[i] in the first available position and set count as
            for (l=0; l<k-1; l++)
            {
                if (temp[l].c == 0)
                {
                    temp[l].e = arr[i];
                    temp[l].c = 1;
                    break;
                }
            }

            /* If all the position in the temp[] are filled, then
               decrease count of every element by 1 */
            if (l == k-1)
                for (l=0; l<k; l++)
                    temp[l].c -= 1;
        }
    }

    /*Step 3: Check actual counts of potential candidates in temp[]*/
    for (int i=0; i<k-1; i++)
    {
        // Calculate actual count of elements
        int ac = 0;  // actual count
        for (int j=0; j<n; j++)
            if (arr[j] == temp[i].e)
                ac++;

        // If actual count is more than n/k, then print it
        if (ac > n/k)
```

```
            cout << "Number:" << temp[i].e
                 << " Count:" << ac << endl;
    }
}

/* Driver program to test above function */
int main()
{
    cout << "First Test\n";
    int arr1[] = {4, 5, 6, 7, 8, 4, 4};
    int size = sizeof(arr1)/sizeof(arr1[0]);
    int k = 3;
    moreThanNdK(arr1, size, k);

    cout << "\nSecond Test\n";
    int arr2[] = {4, 2, 2, 7};
    size = sizeof(arr2)/sizeof(arr2[0]);
    k = 3;
    moreThanNdK(arr2, size, k);

    cout << "\nThird Test\n";
    int arr3[] = {2, 7, 2};
    size = sizeof(arr3)/sizeof(arr3[0]);
    k = 2;
    moreThanNdK(arr3, size, k);

    cout << "\nFourth Test\n";
    int arr4[] = {2, 3, 3, 2};
    size = sizeof(arr4)/sizeof(arr4[0]);
    k = 3;
    moreThanNdK(arr4, size, k);

    return 0;
}
```

Output:

```
First Test
Number:4 Count:3


Second Test
Number:2 Count:2


Third Test
```

```
Number:2 Count:2

Fourth Test
Number:2 Count:2
Number:3 Count:2
```

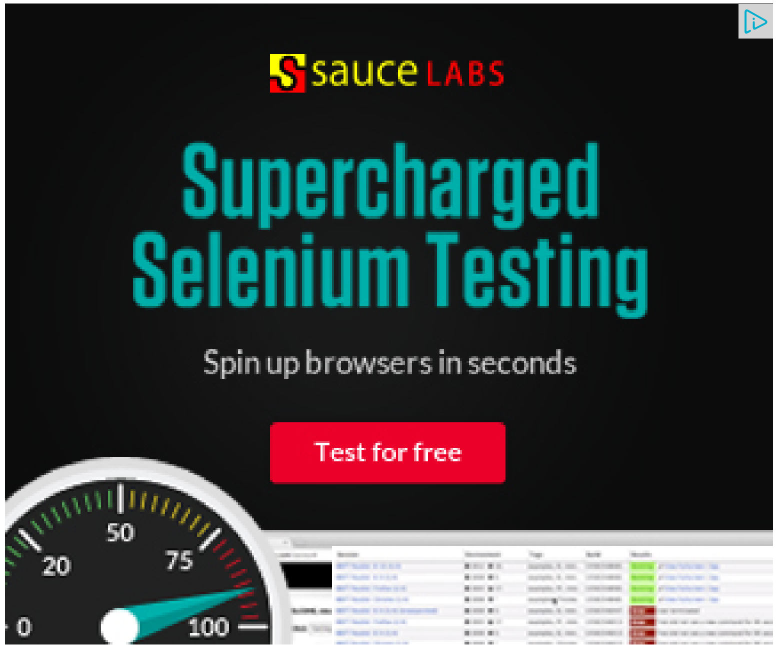
Time Complexity: O(nk)
Auxiliary Space: O(k)

Generally asked variations of this problem are, find all elements that appear n/3 times or n/4 times in O(n) time complexity and O(1) extra space.

Hashing can also be an efficient solution. With a good hash function, we can solve the above problem in O(n) time on average. Extra space required hashing would be higher than O(k). Also, hashing cannot be used to solve above variations with O(1) extra space.

Exercise:
The above problem can be solved in O(nLogk) time with the help of more appropriate data structures than array for auxiliary storage of k-1 elements. Suggest a O(nLogk) approach.

This article is contributed by **Kushagra Jaiswal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above



Related Tpoics:

- Remove minimum elements from either side such that 2*min becomes more than max
- Divide and Conquer | Set 6 (Search in a Row-wise and Column-wise Sorted 2D Array)
- Bucket Sort
- Kth smallest element in a row-wise and column-wise sorted 2D array | Set 1
- Find the number of zeroes
- Find if there is a subarray with 0 sum
- Divide and Conquer | Set 5 (Strassen's Matrix Multiplication)
- Count all possible groups of size 2 or 3 that have sum as multiple of 3

35 Tweet 4 1

Writing code in comment? Please use **ideone.com** and share the link here.