JobFair 2021
Nordeus-QA puzzle

Djordje Karisic

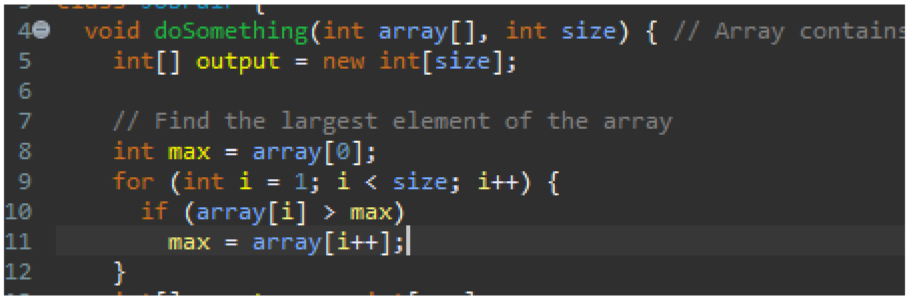
November 2021

1 Discover what the purpose of the code is

The purpose of the given code is, using a non-negative *int* array, to create an additional array containing the cumulative frequency of each element in the original array, and then to return the original array using the cumulative frequency one. In the cumulative frequency array, every index represents a number from the original array, and the address the index points to contains the cumulative frequency of that number. If we find an index that points to *null*, or *0*, that means that, the element it's pointing to does not exist in the original array, and so, going from there we have to find a way to link those two arrays.

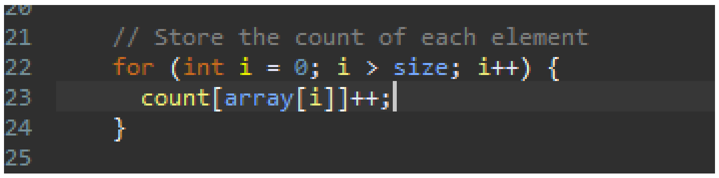
2 Detect as many bugs as you can

First, we will discuss the bugs that I've found in this program, without discussing it's final functionality.



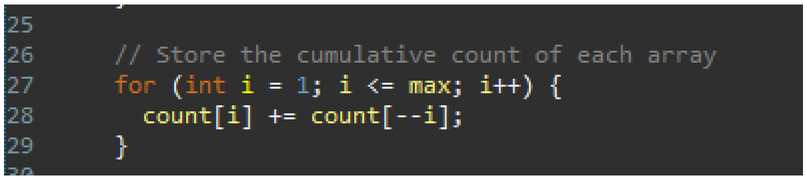
```
4   void doSomething(int array[], int size) { // Array contains
5     int[] output = new int[size];
6
7     // Find the largest element of the array
8     int max = array[0];
9     for (int i = 1; i < size; i++) {
10      if (array[i] > max)
11        max = array[i++];
12    }
```

In the *for* loop in line 9, the loop counter i skips the first member of the array, which is alright because max points to the first member, but in line 11, max points, instead of the element greater than max, to the element after that one. Also, $i++$ makes loop counter i skip an iteration.



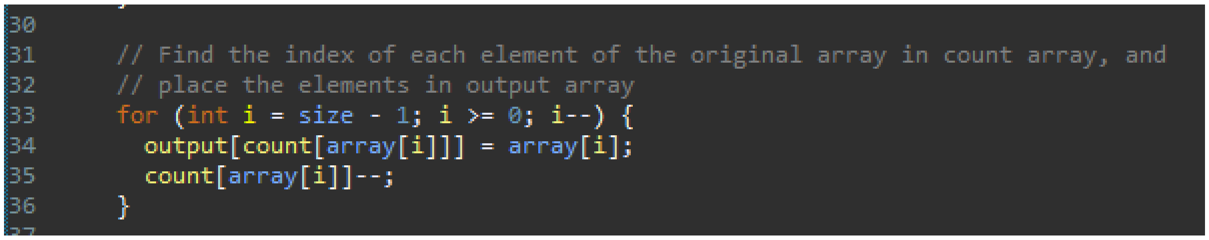
```
21      // Store the count of each element
22      for (int i = 0; i > size; i++) {
23        count[array[i]]++;
24      }
25
```

In the for loop on line 22, the condition for the loop to work and to iterate through the array is not met. *i* is initiated as 0, and thus is not greater than *size*, as size can't be 0. Also, this piece of code is useless, as Java automatically initializes all the members of an int array to be 0 after the call of the default new int[len] constructor.



```
25
26      // Store the cumulative count of each array
27      for (int i = 1; i <= max; i++) {
28        count[i] += count[--i];
29      }
```

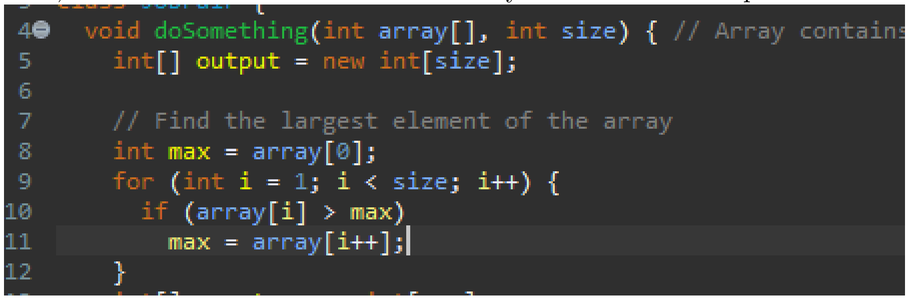
First, *i* is set to 1, which means we skip the first iteration, starting array could countain a 0, which is a non negative integer(0 is a neutral number), that means our *count* array might have it's 0 index pointing to a number. In the line 28 we increase count[i] by its predecessor count[–i], and simultaneously decrease loop counter *i*, setting the loop back to where is started.



```
30
31      // Find the index of each element of the original array in count array, and
32      // place the elements in output array
33      for (int i = size - 1; i >= 0; i--) {
34        output[count[array[i]]] = array[i];
35        count[array[i]]--;
36      }
```

In the line 34, our *count* array could contain an element whose cumulative frequency is larger than the size of output array, and we got an IndexOutOfBounds Exception.

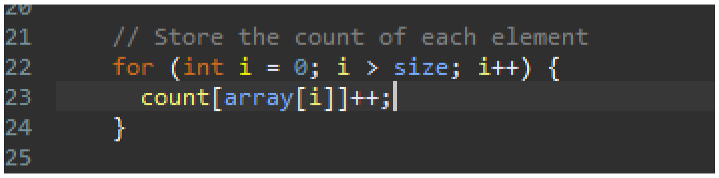
Now, lets discuss the desired functionality of each of these pieces of code.



```
 4    void doSomething(int array[], int size) { // Array contains
 5      int[] output = new int[size];
 6
 7      // Find the largest element of the array
 8      int max = array[0];
 9      for (int i = 1; i < size; i++) {
10        if (array[i] > max)
11          max = array[i++];
12      }
```

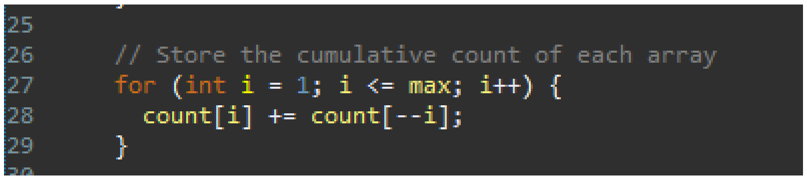
This piece of code should return the highest number in the array, which would

be later used to create an array in which we would store the cumulative frequencies of each element. The max tells us that there are less or equal number of elements in the array.
$len(count) = max(array)$



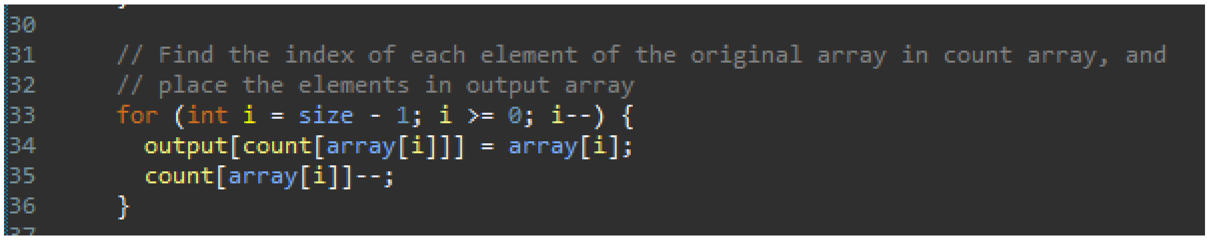
```
    // Store the count of each element
    for (int i = 0; i > size; i++) {
      count[array[i]]++;
    }
```

This piece of code should fill the *count* array with the number of times each element is found within the original array. The value of each $count[i]$ represents how many times does the element i occur in the array.



```
    // Store the cumulative count of each array
    for (int i = 1; i <= max; i++) {
      count[i] += count[--i];
    }
```

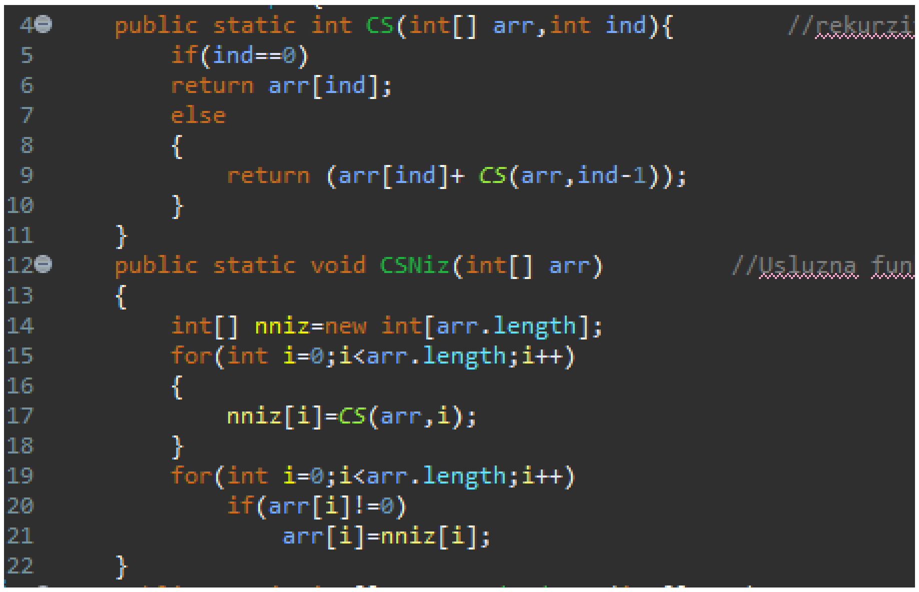
Here we should write a function to turn our relative frequency array into cumulative frequency array. The function written here won't do the job, as discussed previously.



```
    // Find the index of each element of the original array in count array, and
    // place the elements in output array
    for (int i = size - 1; i >= 0; i--) {
      output[count[array[i]]] = array[i];
      count[array[i]]--;
    }
```

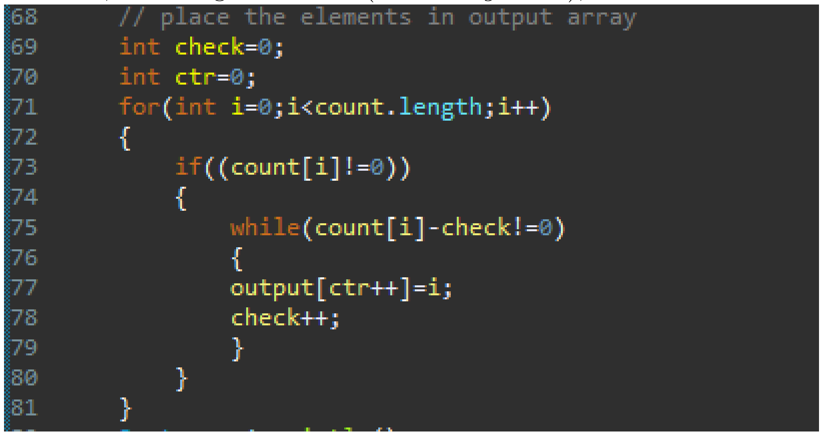
This piece of code should use the connection between the *count* and array arrays and transform the *count* into output array, which should contain the same elements as *array* array. The problem with this piece of code is also discussed previously.

3 Discussing the algorithm



```
    public static int CS(int[] arr,int ind){        //rekurzi
        if(ind==0)
        return arr[ind];
        else
        {
            return (arr[ind]+ CS(arr,ind-1));
        }
    }
    public static void CSNiz(int[] arr)         //Usluzna fun
    {
        int[] nniz=new int[arr.length];
        for(int i=0;i<arr.length;i++)
        {
            nniz[i]=CS(arr,i);
        }
        for(int i=0;i<arr.length;i++)
            if(arr[i]!=0)
                arr[i]=nniz[i];
    }
```

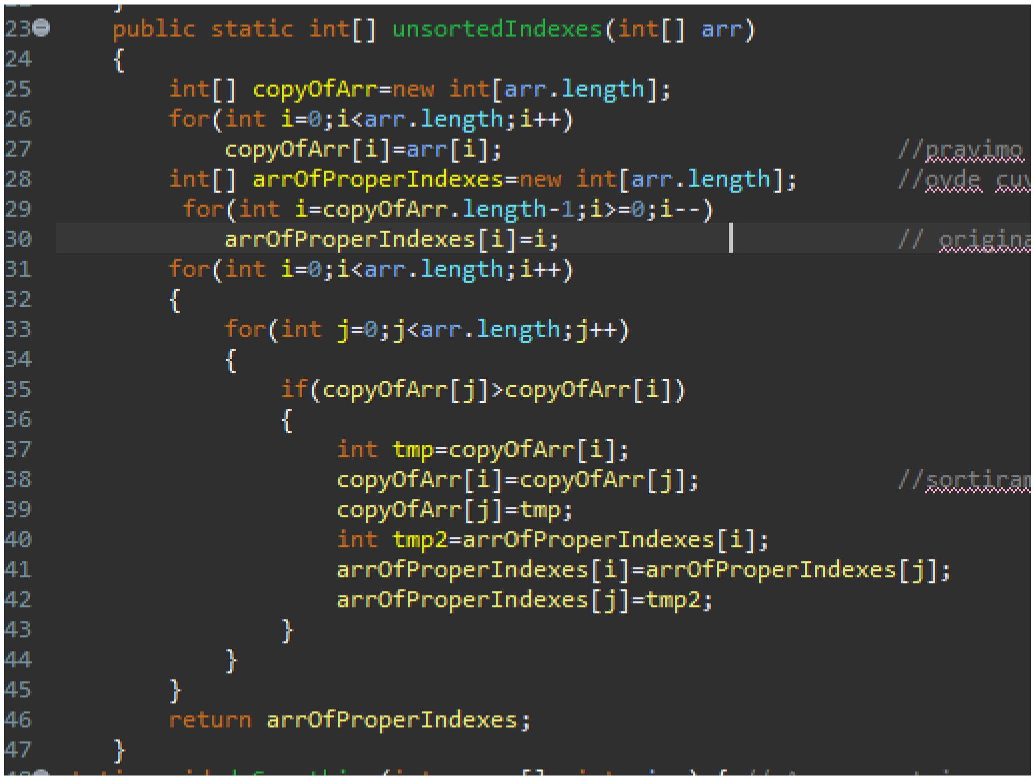
In my opinion if we want to get an array of cumulative frequency of each element, we should use recursive function to get each element, and an iterative one to fill the array with those elements. I think that increases the functionality of this program without complicating the readability, there's the algorithm for transforming *count-¿output*, where I used the function which sorts the copy of the *array* and all the original indexes in the ascending order. In the *count* array we have elements whose values represent how many times each element of *array* shows in *array*, and those *array* elements are represented by *count* indexes. As we know, indexes go from 0 to $(count.length - 1)$, which is an ascending order.



```
    // place the elements in output array
    int check=0;
    int ctr=0;
    for(int i=0;i<count.length;i++)
    {
        if((count[i]!=0))
        {
            while(count[i]-check!=0)
            {
            output[ctr++]=i;
            check++;
            }
        }
    }
```

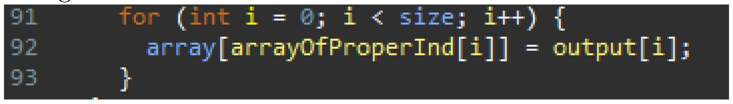
If $count[i]$ is not zero, it means that i is contained in *array*, ($count[i]$-(the cumulative frequency of the previous contained element)) times.

To check how many times an element is contained in *array*, we need to calculate ($count[i]$-(the cumulative frequency of the previous contained element)) To do that, we can use a *check* number, which will point to a current value of cumulative frequency. The result of $(count[i] - check)$ will provide us with how many times we need to add the i element to *output* array.
When the loop is over, our *output* array will have all the elements of the *array* array, but sorted. To return the original array, all we need is to arrange the elements back to their original places.



```
public static int[] unsortedIndexes(int[] arr)
{
    int[] copyOfArr=new int[arr.length];
    for(int i=0;i<arr.length;i++)
        copyOfArr[i]=arr[i];                                //pravimo
    int[] arrOfProperIndexes=new int[arr.length];           //ovde cu
     for(int i=copyOfArr.length-1;i>=0;i--)
        arrOfProperIndexes[i]=i;                            // origina
    for(int i=0;i<arr.length;i++)
    {
        for(int j=0;j<arr.length;j++)
        {
            if(copyOfArr[j]>copyOfArr[i])
            {
                int tmp=copyOfArr[i];
                copyOfArr[i]=copyOfArr[j];                  //sortiran
                copyOfArr[j]=tmp;
                int tmp2=arrOfProperIndexes[i];
                arrOfProperIndexes[i]=arrOfProperIndexes[j];
                arrOfProperIndexes[j]=tmp2;
            }
        }
    }
    return arrOfProperIndexes;
}
```

This function here should sort the copy of the original array and it's indexes. $arrOfProperIndexes$ will contain the original position(index) of every element in the, now sorted, *array*. Copy of the *array* is not necessary, I used it so that I don't change the current *array*, which in this task is not a problem, as we change it at the end.



```
for (int i = 0; i < size; i++) {
  array[arrayOfProperInd[i]] = output[i];
}
```

This piece of code arranges the elements to original order.

4 Suggesting test cases

For this code, we won't need much testing as it is not complex. If we were having some custom Object data type or complicated algorithm we would need to pay much more attention.
Our input array must contain only int data, so to make it foolproof, we should add an exception handler like IllegalArgumentException, and implement it through a try/catch block. In the code I wrote, I used the next int[] arrays to check the functionality:
$\{$1,1,1,1,1,1,1,0,2$\}$
To check how well it handles more elements having the same value:
$\{$9,8,7,6,5,4,3,2,1$\}$
To check the *unsortedIndexes* function
I had *print* function in the *CS* function to test the transformation, and also one for the *count* array, but I deleted them and left the ones in the *main* class.

The code I wrote: JobFairImpr.java
The original code: JobFair.java