Nordeus QA challenge: JobFair 2021

Introduction

This is response to Nordeus 2021 JobFair QA challenge made by Branko Cvetkovic, student at University of Belgrade – Faculty of Mathematics. Here I shall address four tasks given, in their order as written in the official challenge document. Necessary tasks' code can be found here, as well as on my Github page.

Task 1: Discovering code purpose

At the beginning, we are met with an array of integers and its size. Afterwards, a new, same-sized, integer array is created. Following that, the code tries to determine the array's maximum number and allocate enough space for that many integers and set it all to zeros. This is very characteristic for the so-called "counting sort", a sorting algorithm that has linear time and space complexity, but works only for integer-like types. After this, the algorithm attempts to count all the elements of the initial array, place them in order in pre-allocated array, and finally copy those back in the initial one. This is exactly what counting sort does, so the conclusion can be made that this **function is written to perform counting sort, a.k.a. sort the initial array**.

Task 2: Finding as many bugs as possible

1) First of all, code provided is made to work only with integer arrays that contain non-negative elements, as noted in the comment. Since this property is never checked, it is a bug and must be fixed to include all arrays. Therefore, we will also need a minimum of the array. Also, the function itself should not have size as parameter, instead we will declare new size variable and delete it as parameter. Of course, we must be cautious here – if array doesn't even exist, we cannot take its length, and NullPointerException will be raised. Because of that, we must first check whether the given array is actually null.

```
void doSomething(int array[]) { // Performs counting search
    if (array == null) {
        return;
    }
    int size = array.length;
    int[] output = new int[size];
```

2) Next, the line below also contains a bug and can throw ArrayIndexOutOfBoundsException if the original array is empty.

```
int max = array[0];
```

Therefore, it must be changed to the one below. Also, in this whole "maximum finding" section, minimum should also be found (code for variable min will be provided at the end of the section).

```
if (size == 0) {
    return;
}
```

3) Next one is pretty obvious.

```
max = array[i++];
```

Since counter variable is already being incremented in for loop, here it will just cause every other element to be skipped. So new incrementation must be removed.

```
max = array[i];
```

4) We now need additional variable to store length of counting array, since we are now including negative integers. This produces some code changes, but isn't hard to change. The nasty bug here is not incrementing variable k in while loop, which can lead to infinite loop.

```
int countLength = max - min + 1;
int[] count = new int[countLength];

// Initialize count array with all zeros.
int k = 0;
while (k < countLength) {
    count[k] = 0;
    k ++;
}
```

5) Next piece of core regarding counting elements is also bugged.

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

Here we can see classic bug: > instead of <. Also, element addressing needs to be changed to fit our convention.

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

6) Storing cumulative counts also contains bugs.

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

If we aren't careful enough, we can go inside an infinite loop. Namely, inside the loop body, counter is being decremented in each iteration and incremented after each iteration, which

produces the bug. Also, loop boundaries must be changed from max to new one, and <= must become <, otherwise ArrayIndexOutOfBoundsException will be thrown. Therefore, to produce desired behavior, the code must be changes, as shown below.

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

7) Next bug is in creating output array.

```
    output[count[array[i]]] = array[i];
    count[array[i]]--;
```

Output index must be decremented here, otherwise ArrayIndexOutOfBoundsException will be thrown. Also, some convention-style changes must be made.

```
    output[count[array[i] - min] - 1] = array[i];
    count[array[i] - min] --;
```

This concludes the bugs found. Provided below is the list of all the bugs fixed (10 in total), as well as the "clean" code with bugs removed.

- Deleted size parameter from function signature.
- Included case when the array is null.
- Fixed finding maximum – case of empty list.
- Fixed double incrementation while finding maximum.
- Added counter incrementation in the loop that sets all integers in variable count to 0.
- Fixed for loop when counting element occurrences – swapped > with <.
- Fixed for loop when creating cumulative count – swapped <= with <.
- Eliminated infinite loop that is created by decreasing and increasing counter in alternate fashion.
- Fixed ArrayIndexOutOfBoundsException that rises when creating output array.
- Redesigned code so that it can sort all integer arrays.

```
import java.util.Arrays;

public class JobFair {

    void doSomething(int array[]) { // Performs counting search
        if (array == null) {
            return;
        }
        int size = array.length;
        int[] output = new int[size];

        // Find the largest and the smallest element of the array
        if (size == 0) {
            return;
        }
        int max = array[0];
        int min = array[0];
        for (int i = 1; i < size; i ++) {
```

```
            if (array[i] > max) {
                max = array[i];
            }
            if (array[i] < min) {
                min = array[i];
            }
        }
        int countLength = max - min + 1;
        int[] count = new int[countLength];

        // Initialize count array with all zeros
        int k = 0;
        while (k < countLength) {
            count[k] = 0;
            k ++;
        }

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

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

        // 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] - min] - 1] = array[i];
            count[array[i] - min] --;
        }

        // Copy elements into original array
        for (int i = 0; i < size; i ++) {
            array[i] = output[i];
        }
    }
}
```

Task 3: Discussing the algorithm and optimizing it

Counting sort algorithm itself is of linear time complexity, which is rare among the sorting algorithms. Bad side of it is that it also has linear space complexity, a.k.a. requires additional space to store some data that is (asymptotically) equivalent to the length of the beginning array. Function name doSomething could be changed to, for example, countingSort, but we won't do it since it requires refactoring the code it might have been used in. Now, one may ask why did we change the very signature of the method; but it was necessary in order to fix serious bug – mismatching actual array size with the passed one.

In previous section it was mentioned that redesigning code so that it can handle integer arrays with negative values was a bug fix – in some it was because of possibility that some values

could have been negative by mistake, but the bug fixing could also include checking whether the array itself contained negatives (of course, this is only a fix if we have need to sort exclusively arrays with all positive elements).

The algorithm itself, as noted before, is very efficient when the range of the elements does not vary a lot. For example, array {2, 5, -3, 7, 5, 0, 1, 11, -9, 8, 5, 1} can be sorted very efficiently, but array {0, -10000000, 5, -12, 10000000, 13} cannot. Main reason for this is both time and space complexity being **O(n + m)**, where **n** is the length of the array being sorted, and **m** is the difference between the smallest and the largest element of it (needed for counting array). Therefore, this sort should be used carefully and only in controlled situations where it is known that **m** does not go beyond **n** too much (**n** and **m** are from the previous notation).

Speaking of the optimization, we can immediately see that the code section below can be deleted, since the using of operator new with integer arrays in Java automatically sets all of its elements to 0.

```
    // Initialize count array with all zeros.
    int k = 0;
    while (k < countLength) {
        count[k] = 0;
        k ++;
    }
```

Also, importing java.utils.Arrays at the beginning is not necessary since no methods are used from there.

Speaking of finding the minimum and maximum of the array, some optimization can be done. Specifically, it's not needed to access the memory four times, but only once. Of course, there is chance that this will be optimized by the compiler.

```
    for (int i = 1; i < size; i ++) {
        if (array[i] > max) {
            max = array[i];
        }
        if (array[i] < min) {
            min = array[i];
        }
    }
```

So, instead of the upper code, the one below is better.

```
    int currentElement;
    for (int i = 1; i < size; i ++) {
        currentElement = array[i];
        if (currentElement > max) {
            max = currentElement;
        }
        if (currentElement < min) {
            min = currentElement;
        }
    }
```

Same goes for storing a cumulative count of the array. Below are given two code snippets: upper one is slower and the bottom one is optimized. Please note that in the bottom version, taking count[0] is valid and will not throw exceptions, since count is of size max – min + 1, which must be at least 1 because max cannot be lower than min.

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

```
    // Store the cumulative count of each array
    int lastElement = count[0];
    for (int i = 1; i < countLength; i ++) {
        lastElement = (count[i] += lastElement);
    }
```

And, of course, next piece of code can also be optimized (upper code is old one, bottom one is optimized one).

```
    // 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] - min] - 1] = array[i];
        count[array[i] - min] --;
    }
```

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

Please note that arrayI – min will most surely be optimized by compiler.

Full optimized code is given below.

```
public class JobFair {

    void doSomething(int array[]) { // Performs counting search
        if (array == null) {
            return;
        }
        int size = array.length;
        int[] output = new int[size];

        // Find the largest and the smallest element of the array
        if (size == 0) {
            return;
        }
        int max = array[0];
        int min = array[0];
        int currentElement;
```

```
        for (int i = 1; i < size; i ++) {
            currentElement = array[i];
            if (currentElement > max) {
                max = currentElement;
            }
            if (currentElement < min) {
                min = currentElement;
            }
        }
        int countLength = max - min + 1;
        int[] count = new int[countLength];

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

        // Store the cumulative count of each array
        int lastElement = count[0];
        for (int i = 1; i < countLength; i ++) {
            lastElement = (count[i] += lastElement);
        }

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

        // Copy elements into original array
        for (int i = 0; i < size; i ++) {
            array[i] = output[i];
        }
    }
}
```

Task 4: Suggesting test cases

Test cases that should be used in order to confirm that the code works are: the ones that include both positive and negative numbers, ones that contain large differences between largest and smallest numbers, empty array, non-existing array (null), ones that include several same elements, and ones that include large number of elements.

The test cases that were used to test the algorithm are given in a table below. Please note that the compiler used to test the given result is the online compiler, so the results may not be the same for two any attempts.

Test case	Result	Execution time	Memory consumed
{5, 17, 22, 49, -9, 56, 1, 6, 2, 7, -6, 3, 0}	Successfully sorted	0.1 s	32028 KB
{}	No output	0.09 s	31700 KB
null	No output	0.09 s	30856 KB
{1, 8, -10000000, 15, 1, 1, 10000000, 1000000}	Successfully sorted	0.14 s	111428 KB
Case that contains 2000 numbers between -1000000000 and 1000000000	Error thrown: OutOfMemoryError	/	/