JobFair 2021

Nordeus - QA puzzle: Find all the bugs in the given code

Table of Contents

Original Code

```java
import java.util.Arrays;

class JobFair {
  void doSomething(int array[], int size) { // Array contains non-negative integers
    int[] output = new int[size];

    // Find the largest element of the array
    int max = array[0];
    for (int i = 1; i < size; i++) {
      if (array[i] > max)
        max = array[i++];
    }
    int[] count = new int[max];

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

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

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

    // Find the index of each element of the original array in count array, and
33     // place the elements in output array
34     for (int i = size - 1; i >= 0; i--) {
35       output[count[array[i]]] = array[i];
36       count[array[i]]--;
37     }
38
39     // Copy elements into original array
40     for (int i = 0; i < size; i++) {
41       array[i] = output[i];
42     }
43   }
   }
```

What it does

Right now it doesn't do anything since it ends up in an infinite loop, but what it's trying to accomplish is to sort a given `array` in ascending order by counting how many times does any number and its predecessors appear and putting those values in `count`. For example if `count` looks like { 0, 0, 2, 4, 5 } it will give us an `output` of { 2, 2, 3, 3, 4 }.

Finding Bugs

Let's fix all the bugs and then we will try to optimize the code. Since this is a short and simple code snippet, I firstly glimpsed over it manually and noticed a couple of things:

- Line 11 `max = array[i++];` needs to be changed to `max = array[i];`, so we don't increment `i` twice every time we find a new greatest element in an array (and skipping some elements by doing this), since it already increments at the end of each iteration.
- Line 18 `count[k] = 0;` needs to be changed to `count[k++] = 0;`, so we don't end up in an infinite loop because `k` would never reach `max` otherwise.
- Lines 13 `int[] count = new int[max];` and 17 `while (k < max) {` need to be changed to `int[] count = new int[max+1];` and `while (k <= max) {` because `count` is supposed to contain all numbers from `0` to `max`.
- Line 22 `for (int i = 0; i > size; i++) {` has to be changed to `for (int i = 0; i < size; i++) {` or otherwise we would never even enter the `for` loop unless `size` is negative (in which case we would be stuck there).
- Line 28 `count[i] += count[--i];` has to be changed to `count[i] += count[i-1];`, so that, once again, we don't end up being stuck in `for` loop forever because in every iteration we would decrement and increment `i` once, resulting in it always having a value of `1`.

After that I tried to run a short example to help me figure out what's the purpose of lines 33-36. I discovered another bug:

- Line 34 `output[count[array[i]]] = array[i];` needs to be changed to `output[count[array[i]]-1] = array[i];`, so we aren't accessing an element outside of `output`'s bounds in first iteration of `for` loop.

Revamped Code 1

Now if we use `doSomething` as it is intended (we pass it a non-empty `array` with only non-negative integers and `size` that corresponds to number of elements in `array`) it will work, although it's still prone to errors if someone mistakenly doesn't pass the right arguments to it. We are also importing `java.util.Arrays` for no reason.

```
1 import java.util.Arrays;
2
3 class JobFair {
 4   void doSomething(int array[], int size) { // Array contains non-negative
 5 integers
 6     int[] output = new int[size];
 7 
 8     // Find the largest element of the array
 9     int max = array[0];
10     for (int i = 1; i < size; i++) {
11       if (array[i] > max)
12         max = array[i];
13     }
14     int[] count = new int[max+1];
15 
16     // Initialize count array with all zeros.
17     int k = 0;
18     while (k <= max) {
19       count[k++] = 0;
20     }
21 
22     // Store the count of each element
23     for (int i = 0; i < size; i++) {
24       count[array[i]]++;
25     }
26 
27     // Store the cumulative count of each array
28     for (int i = 1; i <= max; i++) {
29       count[i] += count[i-1];
30     }
31 
32     // Find the index of each element of the original array in count array, and
33     // place the elements in output array
34     for (int i = size - 1; i >= 0; i--) {
35         output[count[array[i]]-1] = array[i];
36         count[array[i]]--;
37     }
38 
39     // Copy elements into original array
40     for (int i = 0; i < size; i++) {
41       array[i] = output[i];
42     }
43   }
   }
```

About the algorithm

This is a linear time sorting algorithm with a time complexity of O(5n+2k) which equates to O(n+k), where k is `max+1`, which means the lesser the greatest value in an input array is, the quicker the algorithm. It also has space complexity of O(n+k+1)=O(n+k). We could address the latter in some cases where range of the values isn't that big, but the values themselves are, by also saving the smallest element of input array (for example what if input array has elements ranging from 155K to 160K, we wouldn't want to make a count array that starts from 0).

Revamped Code 2

Let's fix the issues from 2 previous chapters and optimize the code a bit:

- Removed line 1 `import java.util.Arrays;`.
- **Assuming we can change the declaration of the function** we can remove `size` as an argument and calculate it inside of the function and even change to name of the function to something more descriptive

of what it does, for example `countingSort`. We should also check if `array` is empty by checking calculated `size`. If we can't change declaration of the function then we will skip this step.

- We can also find `min` in the same `for` loop on line 10 where we are calculating `max`.
- With both `min` and `max` found, we can calculate `int range = max-min+1;`, and create count with the size of range instead of `max+1`.
- We don't need to initialize count array with all zeros because in Java that is done by default when creating an integer array, so we are going to remove lines 16-19.
- Line 23 `count[array[i]]++;` needs to be changed to `count[array[i]-min]++;`, because we want to be counting numbers from `min` value instead of zero.
- Line 27 `for (int i = 1; i <= max; i++) {` needs to be changed to `for (int i = 1; i < range; i++) {`, because of previous changes.
- Lines 34 `output[count[array[i]]-1] = array[i];` and 35 `count[array[i]]--;` need to be changed to `output[count[array[i]-min]-1] = array[i];` and `count[array[i]-min]--;`, because of previous changes.
- Changed some comments so they are a bit clearer.

After fixing all of this, **with the assumption that we can change the declaration of the function**, the newly revamped code looks like this:

```
 1 class JobFair {
 2   void countingSort(int array[]) {
 3     int size=array.length;
 4     if(size==0)
 5       return;
 6 
 7     // Find the largest and the smallest element in the array
 8     int max = array[0];
 9     int min = max;
10     for (int i = 1; i < size; i++) {
11       if (array[i] > max)
12         max = array[i];
13       else if (array[i]<min)
14         min=array[i];
15     }
16 
17     int range = max-min+1;
18     int[] count = new int[range];
19     int[] output = new int[size];
20 
21     // Count how many times does each element appear in the array
22     for (int i = 0; i < size; i++) {
23       count[array[i]-min]++;
24     }
25 
26     // Add the number of appearances of all its predecessors to each count
27     for (int i = 1; i <range; i++) {
28       count[i] += count[i-1];
29     }
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]-min]-1] = array[i];
35         count[array[i]-min]--;
36     }
37 
38     // Copy elements into original array
39     for (int i = 0; i < size; i++) {
40       array[i] = output[i];
```

```
    }
  }
}
```

Revamped Code 3

We can surround all the code in a try catch box, just to be sure that program won't crash in case some other unwanted errors appear. Another thing we can do is check the amount of free heap we are working with and exit from the function if we would be going over it (this needs more work, there are some cases in which garbage collector would free enough memory for the function to run fine, but we would still be exiting function).

```
class JobFair {
  void countingSort(int array[]) {
    try{
      int size=array.length;
      if(size==0)
        return;

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

      int range = max-min+1;

      long freeHeapSize = Runtime.getRuntime().freeMemory(); //check memory size
      if(Long.valueOf(range+2*size)*Integer.SIZE/8>=freeHeapSize){
        System.out.print("Error - Heap size too small");
        return;
      }

      int[] count = new int[range];
      int[] output = new int[size];

      // Count how many times does each element appear in the array
      for (int i = 0; i < size; i++) {
        count[array[i]-min]++;
      }

      // Add the number of appearances of all its predecessors to each count
      for (int i = 1; i <range; 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];
```

```java
      }
    }
    catch(Exception ex){
      System.out.print("Error - "+ex.toString());
    }
  }
}
```

Examples

Let’s check some examples and see how do Revamped Code 1 (C1) and Revamped Code 3 (C2) fare against each other. (I added a try catch block to C1)

```
n=100000
k=100000
Negative numbers allowed: false
Input array: { 6417, 94749, 48238, 35600, 74992, 62787, 65956, 92391, 33633, 36068, 50397,
57331, 8933, 36258, 71654, 68793, 68244, 87560, 30609, 69197, 12831,  ... } 100000 elements
-------------------------
C2:
Array: { 1, 1, 3, 3, 5, 5, 6, 6, 7, 7, 8, 9, 10, 10, 11, 12, 12, 12, 12, 13, 14,  ... }
100000 elements
Time: 25ms
-------------------------
C1:
Array: { 1, 1, 3, 3, 5, 5, 6, 6, 7, 7, 8, 9, 10, 10, 11, 12, 12, 12, 12, 13, 14,  ... }
100000 elements
Time: 30ms
```

Example 1.

```
n=100000000
k=10
Negative numbers allowed: false
Input array: { 7, 5, 8, 9, 7, 7, 3, 0, 6, 1, 4, 1, 0, 0, 0, 2, 6, 4, 7, 6, 5,  ... }
100000000 elements
-------------------------
C2:
Array: { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  ... } 100000000
elements
Time: 522ms
-------------------------
C1:
Array: { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  ... } 100000000
elements
Time: 289ms
```

Example 2.

```
n=10
k=100000000
Negative numbers allowed: false
Input array: { 78448153, 13695568, 32666180, 57168674, 64894520, 44525700, 7575831,
62317025, 48995088, 40000899 }
-------------------------
C2:
Array: { 7575831, 13695568, 32666180, 40000899, 44525700, 48995088, 57168674, 62317025,
64894520, 78448153 }
Time: 203ms
-------------------------
C1:
Array: { 7575831, 13695568, 32666180, 40000899, 44525700, 48995088, 57168674, 62317025,
64894520, 78448153 }
Time: 221ms
```

Example 3.

```
n=100000
k=100000
Negative numbers allowed: true
Input array: { -5896, -31562, 7668, -24898, -18447, -25974, -800, -38326, -33542, 25122,
34678, -34137, -22226, 42616, -48841, -22639, 31499, -24636, -37144, 33463, -29354,  ... }
100000 elements
-------------------------
C2:
Array: { -49999, -49998, -49998, -49997, -49997, -49996, -49993, -49992, -49988, -49988, -
49987, -49986, -49985, -49984, -49984, -49984, -49982, -49976, -49976, -49975, -49974,  ...
} 100000 elements
Time: 20ms
-------------------------
C1:Error - java.lang.ArrayIndexOutOfBoundsException: Index -5896 out of bounds for length
49995
Array: { -5896, -31562, 7668, -24898, -18447, -25974, -800, -38326, -33542, 25122, 34678, -
34137, -22226, 42616, -48841, -22639, 31499, -24636, -37144, 33463, -29354,  ... } 100000
elements
Time: 6ms
```

Example 4.

```
n=0
k=1000000
Negative numbers allowed: true
Input array: { }
-------------------------
C2:
Array: { }
Time: 0ms
-------------------------
C1:Error - java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
Array: { }
Time: 0ms
```

Example 5.

```
n=1000000
k=1073741823
Negative numbers allowed: false
Input array: { 385108444, 499895334, 534816287, 983378987, 865140139, 1027664661, 985372237,
396321994, 885088722, 62158850, 691860483, 787123159, 890156468, 593324245, 433756366,
426131179, 340617031, 710614274, 687725088, 147391436, 120765834,  ... } 1000000 elements
-------------------------
C2:Error - Heap size too small
Array: { 385108444, 499895334, 534816287, 983378987, 865140139, 1027664661, 985372237,
396321994, 885088722, 62158850, 691860483, 787123159, 890156468, 593324245, 433756366,
426131179, 340617031, 710614274, 687725088, 147391436, 120765834,  ... } 1000000 elements
Time: 6ms
-------------------------
C1:Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at JobFair.doSomething(JobFair.java:67)
        at App.main(App.java:40)
```

Example 6.

We would need to use at least three examples to check the functionality of C2, one with an empty array, one where k and n are very large (to check how does the program handle lack of heap space), and one with mediocre size of k and n (anything which is allowed by our memory size). Also notice the big difference in runtime between C2 and C1 when k is small and n is big in the second example – we traded some performance for the ability to sort an array that also contains negative numbers; but as long as k isn't too small C2 should overall be quicker because of the optimizations.