SORTING ALGORITHMS

NON-comparison based sorting

Comparison based sorting

What does comparison based sorting mean?

if nums[i] > nums[j]
swap(i,j)

We keep comparing items (strings, characters, doubles ...)
~ keep making decisions according to these comparisons

Result: we have to make at least $\log_2 n!$ comparisons to sort an array

Stirling formula yields: $\Omega(N \log N)$
So this is a lower bound, we are not able to do any better !!!

Non-comparison based sorting

Can we do better? **YES**, the solution is not to use comparisons

There are simpler algorithms that can sort a list using partial information about the keys

For example: radix sort, bucket sort

SORTING ALGORITHMS

COUNTING SORT

Counting sort

- It operates by counting the number of objects that have each distinct key value
- Integer sorting algorithm: we assume the values to be integers
- And using arithmetic on those counts to determine the positions of each key value in the output sequence
- It is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items
- It can be used as a subroutine in radix sort
- Because counting sort uses key values as indexes into an array → it is not a comparison based sorting algorithm, so linearithmic running time can be reduced

Counting sort

- Running time: **O(N+k)**
- **N** → number of items we want to sort
- **k** → difference between the maximum and minimum key values, basically the number of possible keys
- Conclusion : it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items

1	4	1	7	1	7	10	3

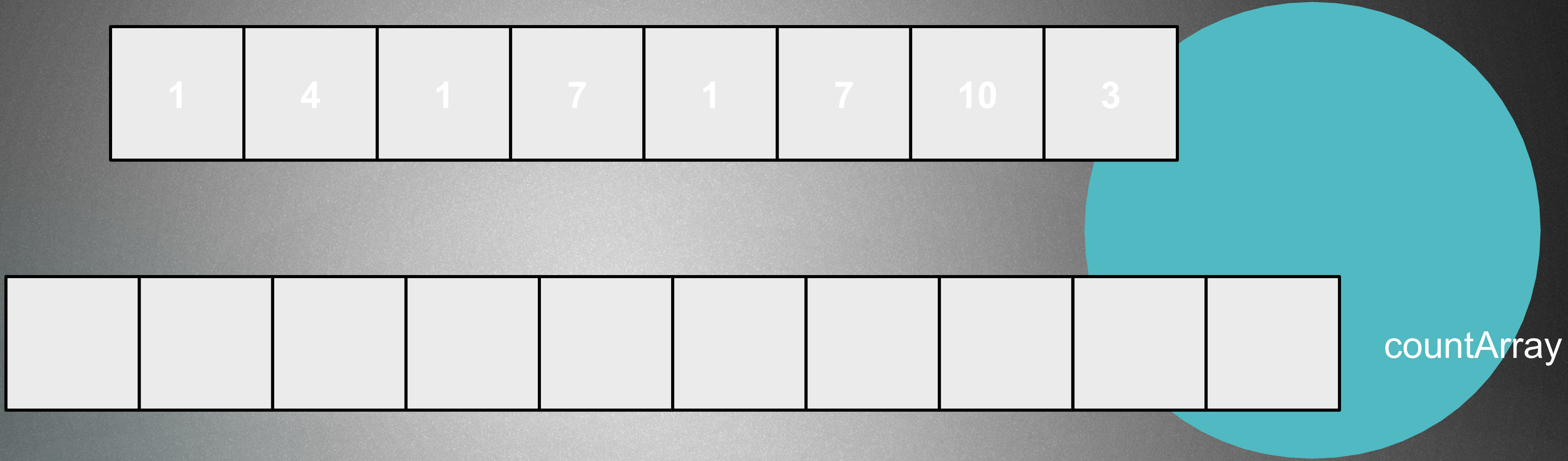
Initial array we want to sort, we have **N** items

1	4	1	7	1	7	10	3

Initial array we want to sort

Allocate memory → for an array size **k**, we want to track and count that how many occurances are there in the original array for the given key

1.) Iterate through the original array **O(N)**
2.) the value in the array will be the index of the temporary array: we increment the counter there
3.) traverse the array of counters (array size **k**) and print out the values **O(k)**
4.) it is going to yield the numerical ordering



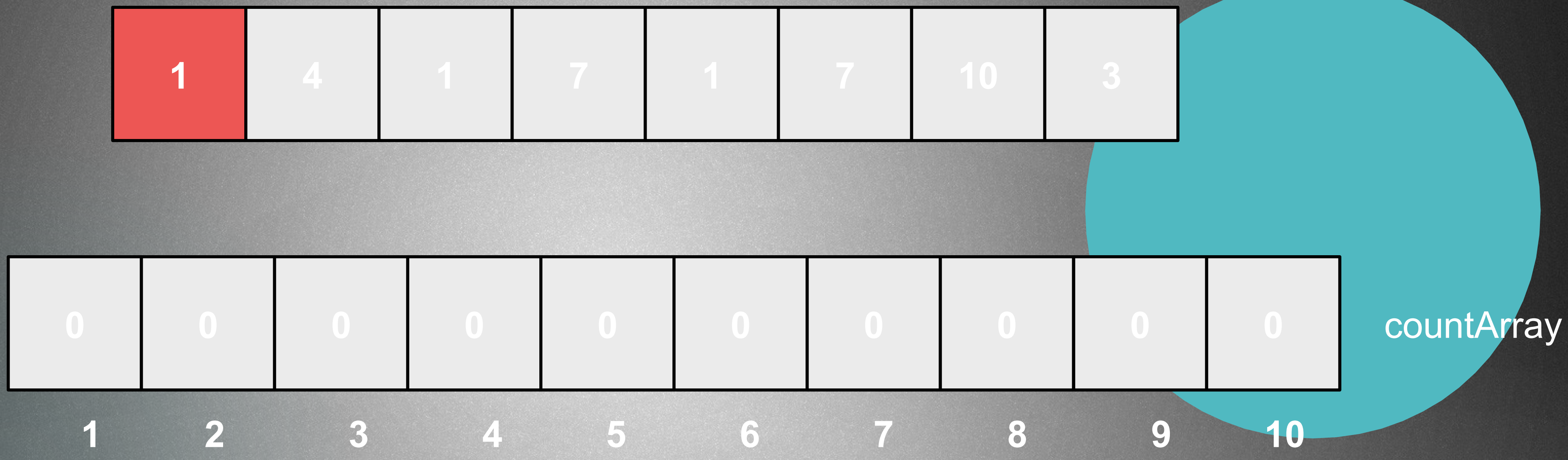
We allocate an array with size **k** for the counters: what is **k** exactly?

We will have 10 (**max – min +1**) slots in our array

1	4	1	7	1	7	10	3

countArray

1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0


1 4 1 7 1 7 10 3
0 0 0 0 0 0 0 0 0 0
countArray
1 2 3 4 5 6 7 8 9 10

1	4	1	7	1	7	10	3

countArray

1	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10

1	4	1	7	1	7	10	3

countArray

1	0	0	1	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10

1	4	1	7	1	7	10	3

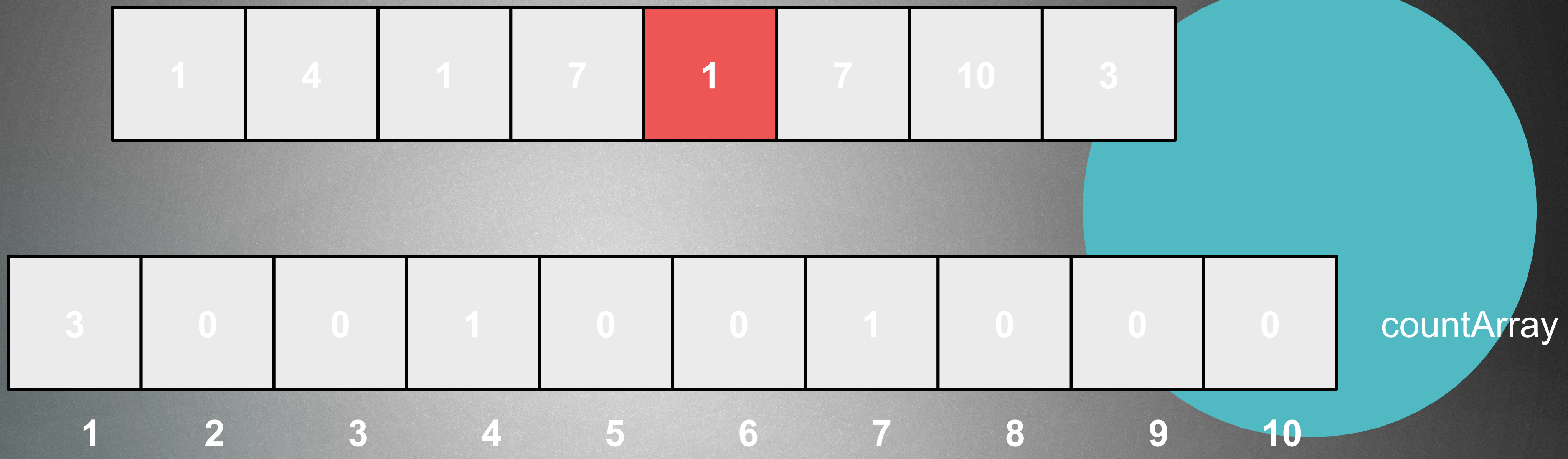
countArray

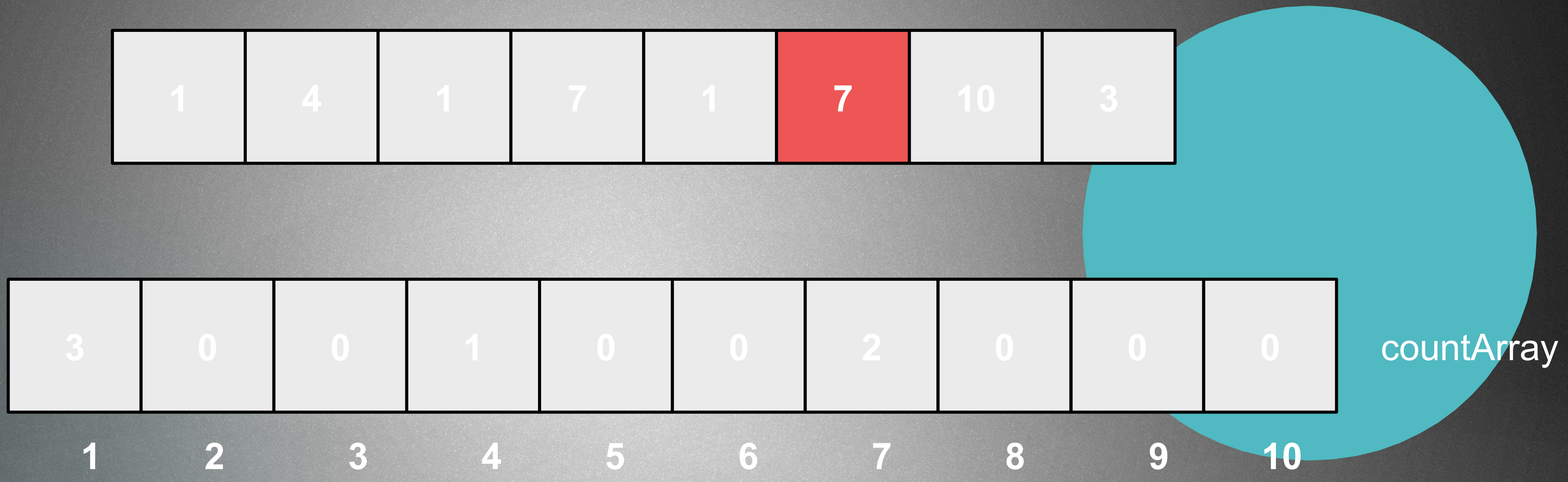
2	0	0	1	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10

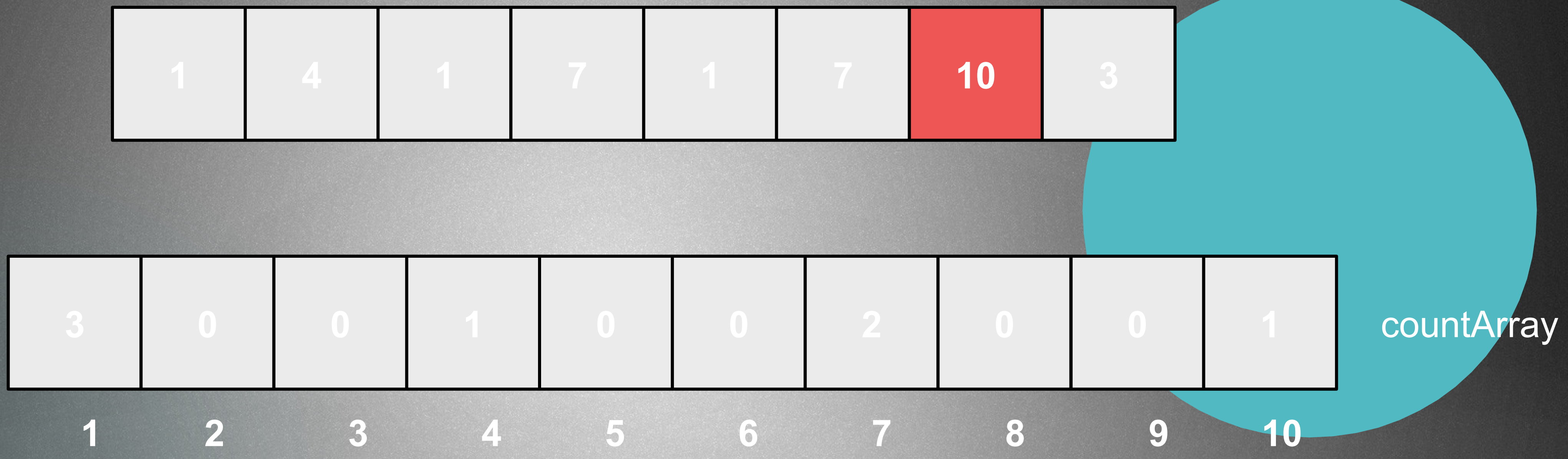
1	4	1	7	1	7	10	3

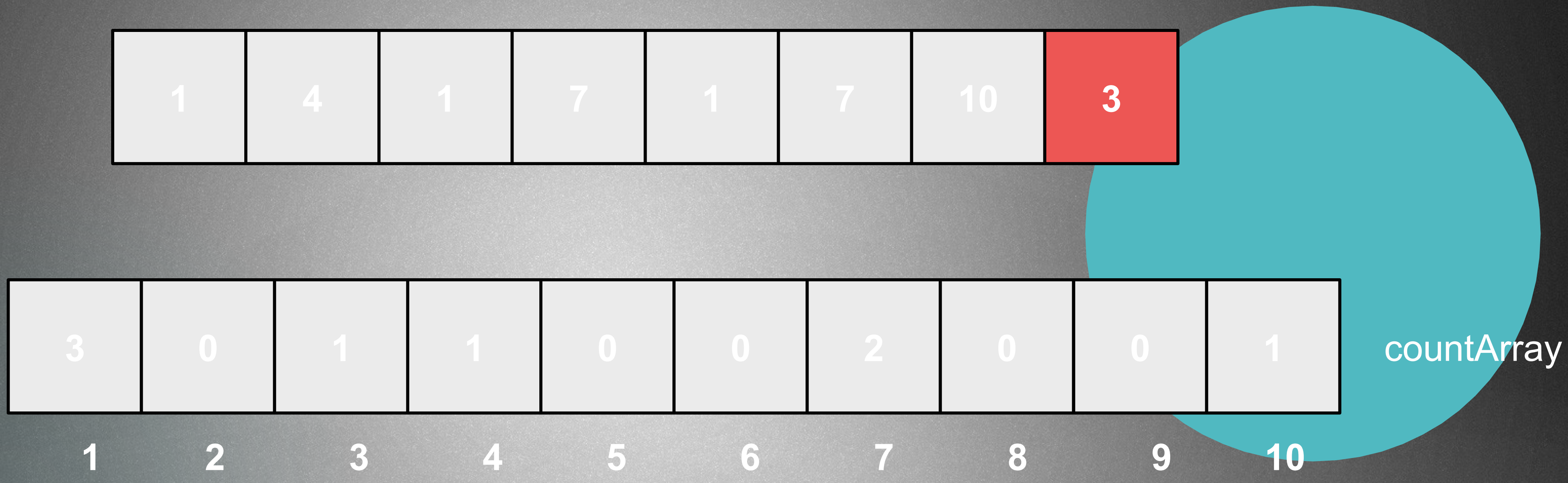
countArray

countArray	2	0	0	1	0	0	1	0	0	0
	1	2	3	4	5	6	7	8	9	10


1 4 1 7 1 7 10 3
3 0 0 1 0 0 1 0 0 0
countArray
1 2 3 4 5 6 7 8 9 10


1 4 1 7 1 7 10 3
3 0 0 1 0 0 2 0 0 0
countArray
1 2 3 4 5 6 7 8 9 10


1 4 1 7 1 7 10 3
3 0 0 1 0 0 2 0 0 1
countArray
1 2 3 4 5 6 7 8 9 10

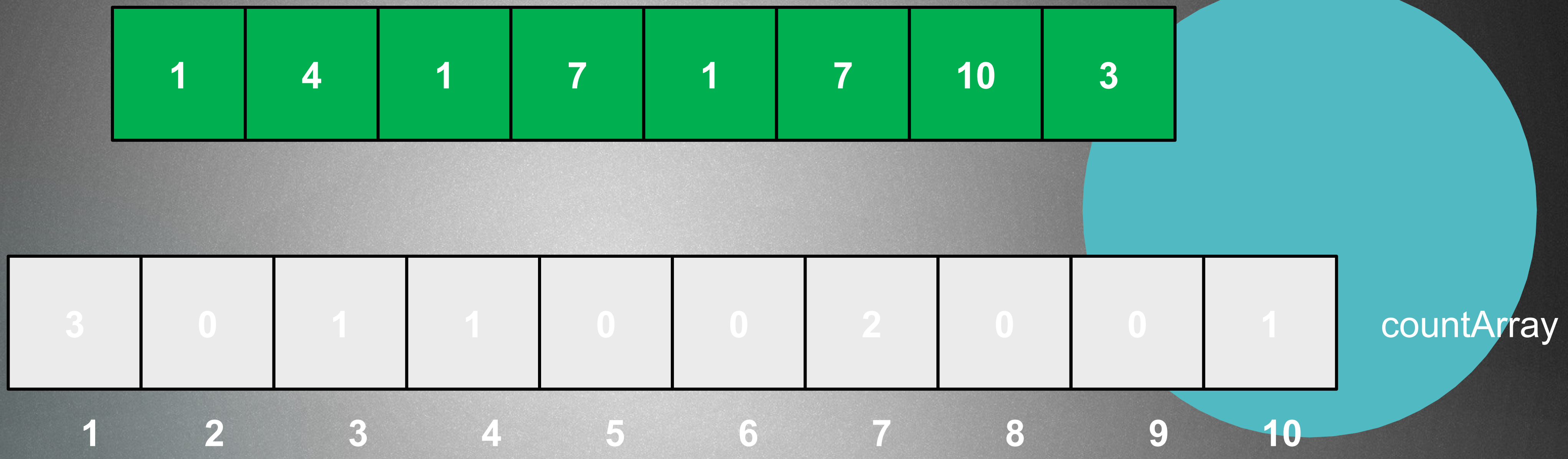

1 4 1 7 1 7 10 3
3 0 1 1 0 0 2 0 0 1
countArray
1 2 3 4 5 6 7 8 9 10

We have considered all the items in the array in **O(N)** time

1	4	1	7	1	7	10	3

3	0	1	1	0	0	2	0	0	1	countArray
1	2	3	4	5	6	7	8	9	10	

We have considered all the items in the array in **O(N)** time



We consider every item in our counter array! The content of the array determines how many times the given index is present in the original array

We have considered all the items in the array in **O(N)** time

1	4	1	7	1	7	10	3

countArray

3	0	1	1	0	0	2	0	0	1
1	2	3	4	5	6	7	8	9	10

Content value: 3
Index: 1

Meaning: we have 3 items with key 1 in out original array

We have considered all the items in the array in **O(N)** time

1	4	1	7	1	7	10	3

countArray

3	0	1	1	0	0	2	0	0	1
1	2	3	4	5	6	7	8	9	10

Numerical ordering:

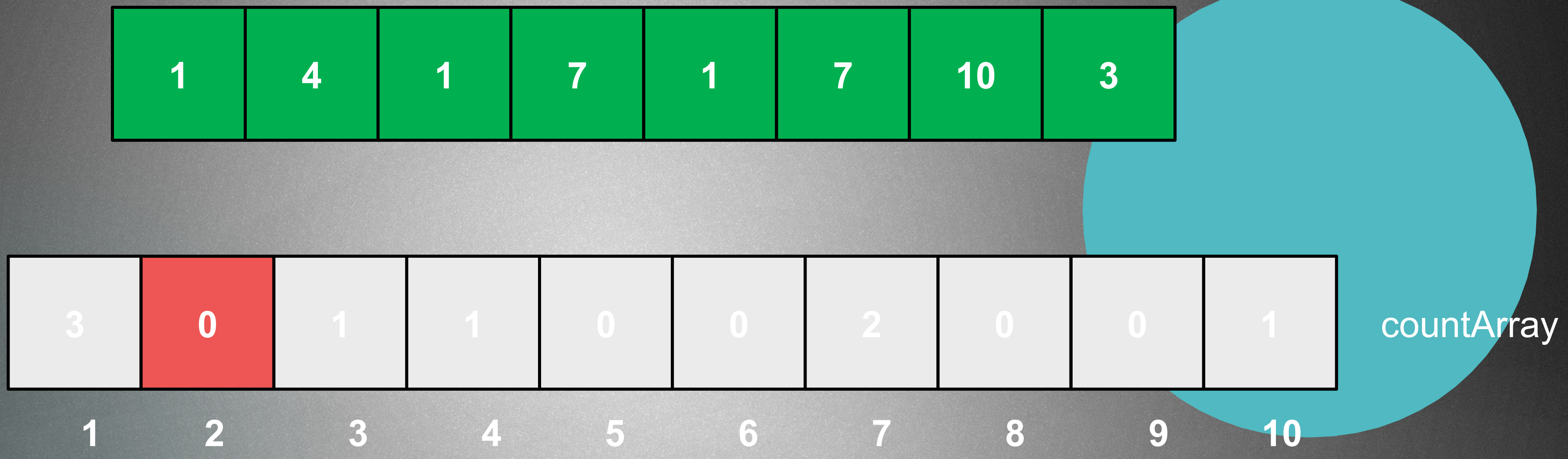
We have considered all the items in the array in **O(N)** time

1	4	1	7	1	7	10	3

3	0	1	1	0	0	2	0	0	1	countArray
1	2	3	4	5	6	7	8	9	10	

<u>Numerical ordering:</u> 1, 1, 1

We have considered all the items in the array in **O(N)** time



Numerical ordering: 1, 1, 1

We have considered all the items in the array in **O(N)** time

1	4	1	7	1	7	10	3

countArray

3	0	1	1	0	0	2	0	0	1
1	2	3	4	5	6	7	8	9	10

Numerical ordering: 1, 1, 1, 3

We have considered all the items in the array in **O(N)** time

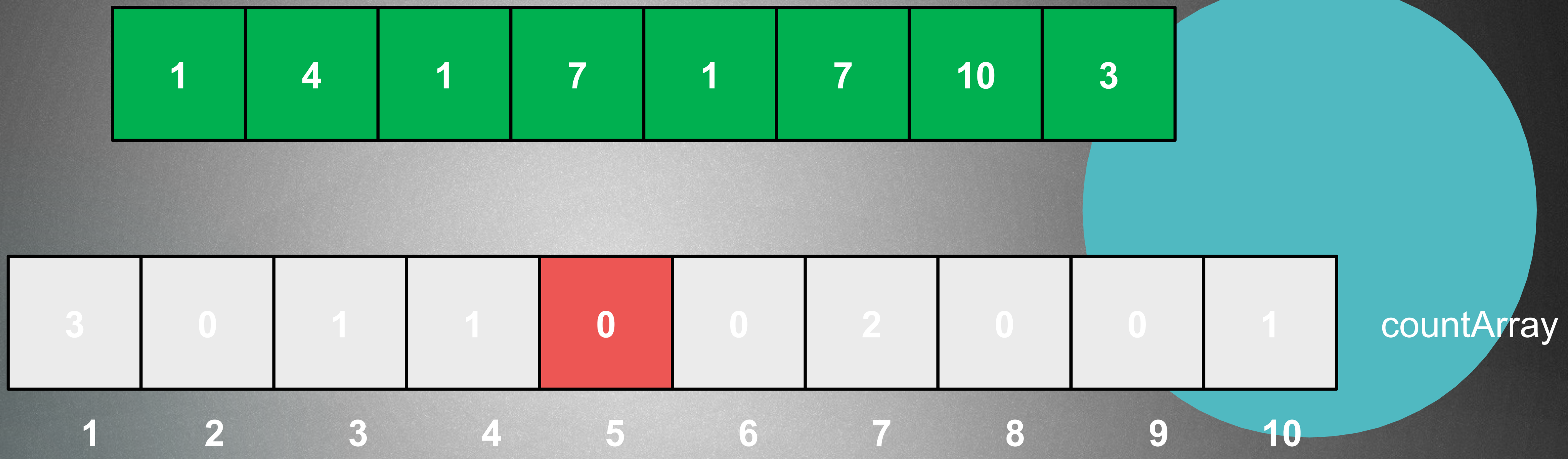
1	4	1	7	1	7	10	3

countArray

3	0	1	1	0	0	2	0	0	1
1	2	3	4	5	6	7	8	9	10

<u>Numerical ordering:</u> 1, 1, 1, 3, 4

We have considered all the items in the array in **O(N)** time



__Numerical ordering:__ 1, 1, 1, 3, 4

We have considered all the items in the array in **O(N)** time

1	4	1	7	1	7	10	3

countArray

3	0	1	1	0	0	2	0	0	1
1	2	3	4	5	6	7	8	9	10

Numerical ordering: 1, 1, 1, 3, 4

We have considered all the items in the array in **O(N)** time

1	4	1	7	1	7	10	3

countArray

1	2	3	4	5	6	7	8	9	10
3	0	1	1	0	0	2	0	0	1

Numerical ordering: 1, 1, 1, 3, 4, 7, 7

We have considered all the items in the array in **O(N)** time

1	4	1	7	1	7	10	3

3	0	1	1	0	0	2	0	0	1	countArray
1	2	3	4	5	6	7	8	9	10	

Numerical ordering: 1, 1, 1, 3, 4, 7, 7

We have considered all the items in the array in **O(N)** time

1	4	1	7	1	7	10	3

countArray

3	0	1	1	0	0	2	0	0	1
1	2	3	4	5	6	7	8	9	10

Numerical ordering: 1, 1, 1, 3, 4, 7, 7

We have considered all the items in the array in **O(N)** time

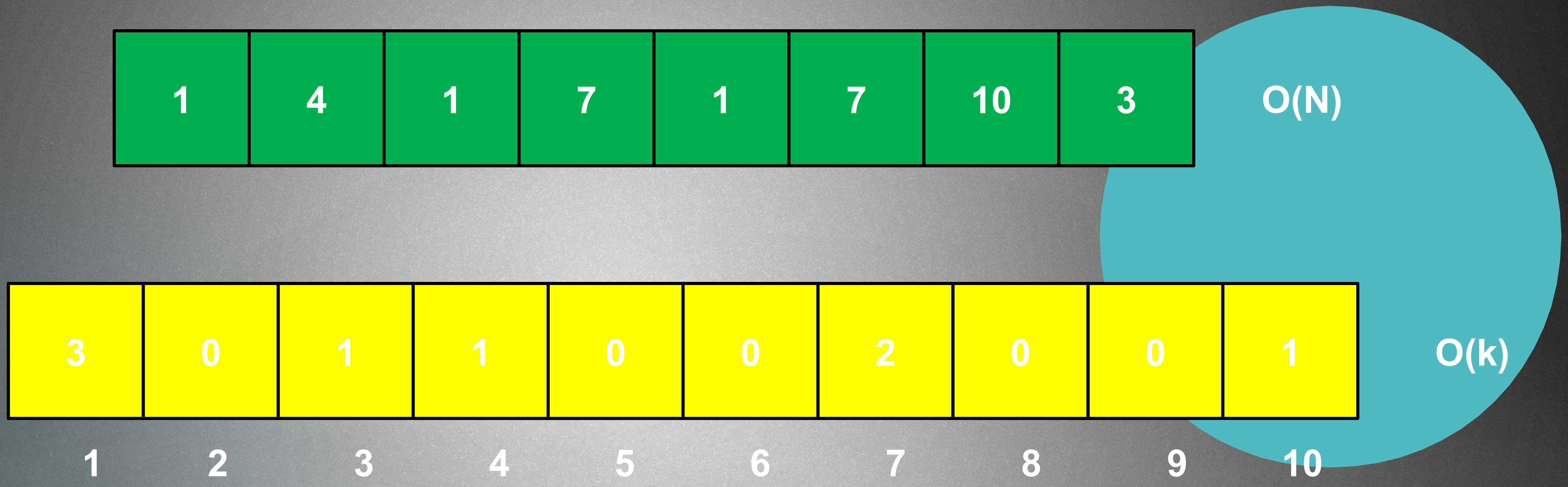
1	4	1	7	1	7	10	3

countArray

3	0	1	1	0	0	2	0	0	1
1	2	3	4	5	6	7	8	9	10

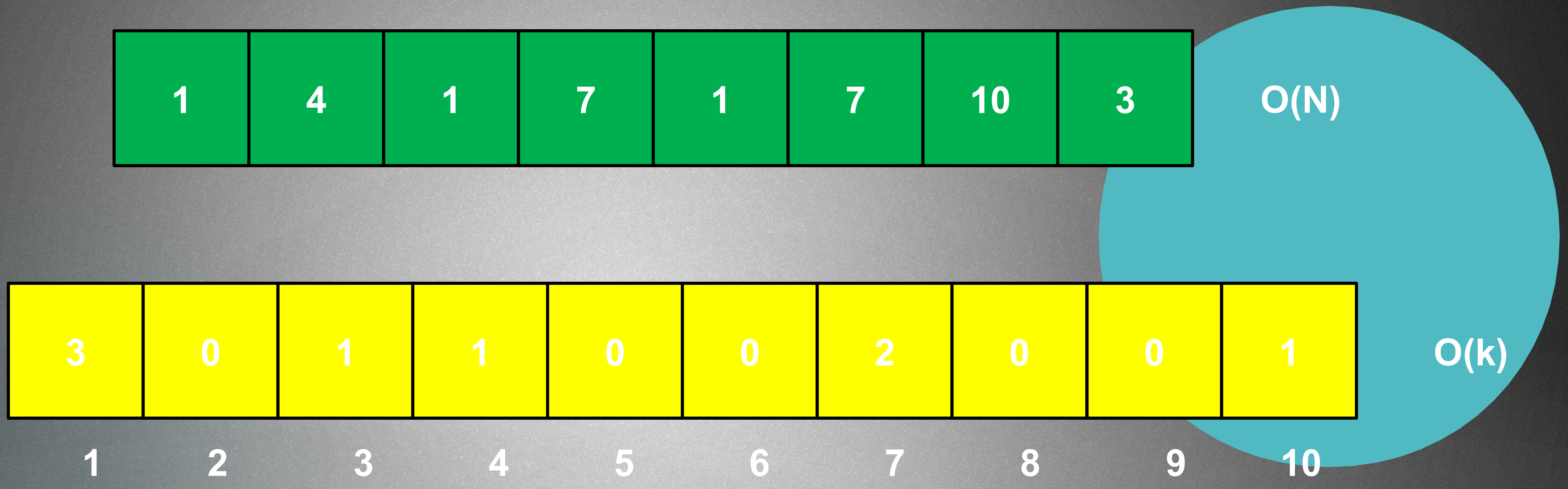
Numerical ordering: 1, 1, 1, 3, 4, 7, 7, 10

We have considered all the items in the array in **O(N)** time



Numerical ordering: 1, 1, 1, 3, 4, 7, 7, 10

We have considered all the items in the array in **O(N)** time



<u>**Numerical ordering:**</u> **1, 1, 1, 3, 4, 7, 7, 10**

Overall running time: **O(k) + O(N) = O(N+k)**

Problem: **k** can be very very large, and the counting sort algorithm will be slow

Pseudocode:

```
countingSort(array, max, min)

    countArray = new array with size [max-min+1]

    for i in array
        increment countArray[i-min]
    end

    z = 0

    for i in array
        while countArray[i-min] > 0
            array[z] = i
            z = z + 1
            countArray[i-min] = countArray[i-min] - 1
        end
    end
end
```

Pseudocode:

```
countingSort(array, max, min)

    countArray = new array with size [max-min+1]

    for i in array
        increment countArray[i-min]
    end

    z = 0

    for i in array
        while countArray[i-min] > 0
            array[z] = i
            z = z + 1
            countArray[i-min] = countArray[i-min] - 1
        end
    end
end
```

We have to specify the array, the minimum key and the maximum key

Pseudocode:

```
countingSort(array, max, min)

    countArray = new array with size [max-min+1]

    for i in array
        increment countArray[i-min]
    end

    z = 0

    for i in array
        while countArray[i-min] > 0
            array[z] = i
            z = z + 1
            countArray[i-min] = countArray[i-min] - 1
        end
    end
end
```

Counting sort is not in place, we do need some additional memory

max-min+1 going to determine the size of the array

Thats why counting sort can get slow if this countArray is huge

Pseudocode:

```
countingSort(array, max, min)

    countArray = new array with size [max-min+1]

    for i in array
        increment countArray[i-min]
    end

    z = 0

    for i in array
        while countArray[i-min] > 0
            array[z] = i
            z = z + 1
            countArray[i-min] = countArray[i-min] - 1
        end
    end
end
```

We consider every key in the original array in **O(N)** and increment the counterArray according to the right position

Pseudocode:

```
countingSort(array, max, min)

    countArray = new array with size [max-min+1]

    for i in array
        increment countArray[i-min]
    end

    z = 0

    for i in array
        while countArray[i-min] > 0
            array[z] = i
            z = z + 1
            countArray[i-min] = countArray[i-min] - 1
        end
    end
end
```

This operation has **O(k)** time complexity

Basically while the counterArray is not **0** at a given position → we have to consider that index as many times as the value in **counterArray[index]**

It yields the numerical ordering

SORTING ALGORITHMS

RADIX SORT