You created a game that is more popular than Angry Birds.

You rank players in the game from highest to lowest score. So far you're using an algorithm that sorts in $O(n \lg n)$ time, but players are complaining that their rankings aren't updated fast enough. You need a faster sorting algorithm.

Write a function that takes:

1. an array of `unsortedScores`
2. the `highestPossibleScore` in the game

and returns a sorted array of scores in less than $O(n \lg n)$ time.

For example:

Java ▾

```java
int[] unsortedScores = {37, 89, 41, 65, 91, 53};
final int HIGHEST_POSSIBLE_SCORE = 100;

int[] sortedScores = sortScores(unsortedScores, HIGHEST_POSSIBLE_SCORE);
// sortedScores: [37, 41, 53, 65, 89, 91]
```

We're defining n as the number of `unsortedScores` because we're expecting the number of players to keep climbing.

And we'll treat `highestPossibleScore` as a constant instead of factoring it into our big O time and space costs, because the highest possible score isn't going to change. Even if we *do* redesign the game a little, the scores will stay around the same order of magnitude.

Gotchas

Multiple players can have the same score! If 10 people got a score of 90, the number 90 should appear 10 times in our output array.

We can do this in $O(n)$ time and space.

Breakdown

$O(n \lg n)$ is the time to beat. Even if our array of scores were *already sorted* we'd have to do a full walk through the array to confirm that it was in fact fully sorted. So we have to spend *at least* $O(n)$ time on our sorting function. If we're going to do better than $O(n \lg n)$, we're probably going to do exactly $O(n)$.

What are some common ways to get $O(n)$ runtime?

One common way to get $O(n)$ runtime is to use a greedy algorithm . But in this case we're not looking to just grab a specific value from our input set (e.g. the "largest" or the "greatest difference")—we're looking to reorder the whole set. That doesn't lend itself as well to a greedy approach.

Another common way to get $O(n)$ runtime is to use counting . We can build an array `scoreCounts` where the indices represent scores and the values represent how many times the score appears. Once we have that, can we generate a sorted array of scores?

What if we did an in-order walk through `scoreCounts`. Each index represents a `score` and its value represents the `count` of appearances. So we can simply add the score to a new array `sortedScores` as many times as `count` of appearances.

Solution

We use counting sort

Counting sort is a very time-efficient (and somewhat space-inefficient) algorithm for sorting that avoids comparisons and exploits the $O(1)$ time insertions and lookups in an array.

The idea is simple: if you're sorting integers and you know they all fall in the range $1..100$, you can generate a sorted array this way:

- Allocate an array `numCounts` where the indices represent numbers from our input array and the values represent how many times the index number appears. Start each value at 0.
- In one pass of the input array, update `numCounts` as you go, so that at the end the values in `numCounts` are correct.
- Allocate an array `sortedArray` where we'll store our sorted numbers.

- In one in-order pass of `numCounts` put each number, the correct number of times, into `sortedArray`.

Java ▾

```java
public int[] countingSort(int[] theArray, int maxValue) {

    // array of 0's at indices 0...maxValue
    int numCounts[] = new int[maxValue + 1];

    // populate numCounts
    for (int num : theArray) {
        numCounts[num] += 1;
    }

    // populate the final sorted array
    int[] sortedArray = new int[theArray.length];
    int currentSortedIndex = 0;

    // for each num in numCounts
    for (int num = 0; num < numCounts.length; num++) {
        int count = numCounts[num];

        // for the number of times the item occurs
        for (int x = 0; x < count; x++) {

            // add it to the sorted array
            sortedArray[currentSortedIndex] = num;
            currentSortedIndex++;
        }
    }

    return sortedArray;
}
```

Counting sort takes $O(n)$ time and $O(n)$ additional space (for the new array that we end up returning).

> **Wait, aren't we nesting two loops towards the bottom? So shouldn't it be $O(n^2)$ time?** Notice *what those loops iterate over*. The *outer* loop runs once for each *unique* number in the array. The *inner* loop runs once for each *time that number occurred*.
>
> So in essence we're just looping through the n numbers from our input array, except we're splitting it into two steps: (1) each unique number, and (2) each time that number appeared.
> Here's another way to think about it: in each iteration of our two nested loops, we append one item to `sortedArray`. How many numbers end up in `sortedArray` in the end? Exactly how many were in our input array! n!

There are some rare cases where even though our input items aren't integers bound by constants, we can write a function that *maps* our items to integers from 0 to some constant such that different items will always map to different integers. This allows us to use counting sort.

.

Java ▾

```java
public int[] sortScores(int[] unorderedScores, int highestPossibleScore) {

    // array of 0s at indices 0..highestPossibleScore
    int[] scoreCounts = new int[highestPossibleScore + 1];

    // populate scoreCounts
    for (int score : unorderedScores) {
        scoreCounts[score]++;
    }

    // populate the final sorted array
    int[] sortedScores = new int[unorderedScores.length];
    int currentSortedIndex = 0;

    // for each item in scoreCounts
    for (int score = 0; score <= highestPossibleScore; score++) {
        int count = scoreCounts[score];

        // for the number of times the item occurs
        for (int occurrence = 0; occurrence < count; occurrence++) {

            // add it to the sorted array
            sortedScores[currentSortedIndex] = score;
            currentSortedIndex++;
        }
    }

    return sortedScores;
}
```

Complexity

$O(n)$ time and $O(n)$ space, where n is the number of scores.

> **Wait, aren't we nesting two loops towards the bottom? So shouldn't it be $O(n^2)$ time?** Notice *what those loops iterate over*. The *outer* loop runs once for each *unique* number in the array. The *inner* loop runs once for each *time that number occurred*.
>
> So in essence we're just looping through the n numbers from our input array, except we're splitting it into two steps: (1) each unique number, and (2) each time that number appeared.
>
> Here's another way to think about it: in each iteration of our two nested loops, we append one item to `sortedScores`. How many numbers end up in `sortedScores` in the end? Exactly how many were in our input array! n!

If we didn't treat `highestPossibleScore` as a constant, we could call it k and say we have $O(n + k)$ time and $O(n + k)$ space.

Bonus

Note that by optimizing for time we ended up incurring some space cost! What if we were optimizing for space?

We chose to generate and return a separate, sorted array. Could we instead sort the array in place? Does this change the time complexity? The space complexity?