🍰 **Interview Cake**

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

Each round, players receive a score between 0 and 100, which you use to rank them from highest to lowest. 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. a vector of `unsortedScores`
2. the `highestPossibleScore` in the game

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

For example:

C++ ▾

```cpp
const vector<int> unsortedScores {37, 89, 41, 65, 91, 53};
const int HIGHEST_POSSIBLE_SCORE = 100;

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

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 vector.

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

Breakdown

$O(n \lg n)$ is the time to beat. Even if our vector of scores were *already sorted* we'd have to do a full walk through the vector 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 a vector `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 vector 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 vector `sortedScores` as many times as `count` of appearances.

Solution

We use counting sort.

C++ ▼

```cpp
vector<int> sortScores(const vector<int>& unorderedScores, const int highestPossibleScore)
{
    // vector of 0s at indices 0..highestPossibleScore
    vector<size_t> scoreCounts(highestPossibleScore + 1);

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

    // populate the final sorted array
    vector<int> sortedScores(unorderedScores.size());
    size_t currentSortedIndex = 0;

    // for each item in scoreCounts
    for (int score = highestPossibleScore; score >= 0; --score) {
        size_t count = scoreCounts[score];

        // for the number of times the item occurs
        for (size_t 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 vector. 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 vector, 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 vector! 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 vector. Could we instead sort the vector in place? Does this change the time complexity? The space complexity?

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.