Interview Cake

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. a list of `unsorted_scores`
2. the `highest_possible_score` in the game

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

For example:

```
unsorted_scores = [37, 89, 41, 65, 91, 53]
HIGHEST_POSSIBLE_SCORE = 100

sort_scores(unsorted_scores, HIGHEST_POSSIBLE_SCORE)
# returns [37, 41, 53, 65, 89, 91]
```

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

And we'll treat `highest_possible_score` 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 list.

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

Breakdown

$O(n \lg n)$ is the time to beat. Even if our list of scores were *already sorted* we'd have to do a full walk through the list 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↓

Counting is a common pattern in time-saving algorithms. It can often get you $O(n)$ runtime, but at the expense of adding $O(n)$ space.

The idea is to define a dictionary or list (call it e.g. `counts`) where the keys/indices represent the items from the input set and the values represent the number of times the item appears. In one pass through the input you can fully populate `counts`:

Python

```python
counts = {}
for item in the_list:
    if item in counts:
        counts[item] += 1
    else:
        counts[item] = 1
```

Once you know how many times each item appears, it's trivial to:

- generate a sorted list
- find the item that appears the most times

- etc

. We can build a list `score_counts` where the indices represent scores and the values represent how many times the score appears. Once we have that, can we generate a sorted list of scores?

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

Solution

We use counting sort↴ .

Python ▾

```
def sort_scores(unsorted_scores, highest_possible_score):

    # list of 0s at indices 0..highest_possible_score
    score_counts = [0] * (highest_possible_score+1)

    # populate score_counts
    for score in unsorted_scores:
        score_counts[score] += 1

    # populate the final sorted list
    sorted_scores = []

    # for each item in score_counts
    for score, count in enumerate(score_counts):

        # for the number of times the item occurs
        for time in range(count):

            # add it to the sorted list
            sorted_scores.append(score)

    return sorted_scores
```

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 list. 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 list, 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 `sorted_scores`. How many numbers end up in `sorted_scores` in the end? Exactly how many were in our input list! n!

If we didn't treat `highest_possible_score` 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 list. Could we instead sort the list in place? Does this change the time complexity? The space complexity?