Daily Coding Problem #144

Problem

This problem was asked by Google.

Given an array of numbers and an index `i`, return the index of the nearest larger number of the number at index `i`, where distance is measured in array indices.

For example, given `[4, 1, 3, 5, 6]` and index `0`, you should return `3`.

If two distances to larger numbers are the equal, then return any one of them. If the array at `i` doesn't have a nearest larger integer, then return null.

Follow-up: If you can preprocess the array, can you do this in constant time?

Solution

We can do this the brute force method by simply iterating over the range of the array, and trying each offset in both directions until we find a number larger than `arr[i]`:

```
def nearest(arr, i):
    for j in range(1, len(arr)):
        low = i - j
        high = i + j
        if 0 <= low and arr[low] > arr[i]:
            return low
        if high < len(arr) and arr[high] > arr[i]:
            return high
```

This takes O(1) space but O(n) time, since we may need to iterate over the whole array.

To speed up the querying, we can preprocess the array to build a lookup array. We'll store the closest index for each corresponding index in the array. To build it fast, we can simply look at pairwise elements along the array to see which one's bigger, and set the index of the lookup array appropriately.

However, there may be cases where we end up with nulls, such as with our example input:

`[4, 1, 3, 5, 6]` would become `[null, 2, 3, 4, null]` -- the first null shouldn't be there!

We'll fallback to our original method to fill in the nulls:

```
def _nearest(arr, i):
    for j in range(1, len(arr)):
        low = i - j
        high = i + j
        if 0 <= low and arr[low] > arr[i]:
            return low
        if high < len(arr) and arr[high] > arr[i]:
            return high

def preprocess(arr):
    cache = [None for _ in range(len(arr))]

    for j in range(len(arr) - 1):
        if arr[j] > arr[j + 1]:
            cache[j + 1] = j
        elif arr[j + 1] > arr[j]:
            cache[j] = j + 1

    for i, val in enumerate(cache):
        if val is None:
            cache[i] = _nearest(arr, i)

    return cache
```

Now we can call `preprocess(arr)` on our array and query the index at `i`.

If the elements in the original array are distinct, then this will take linear time. Otherwise, if there are duplicates, then it could take O(n^2) time in the worst case.