Wiggle Sort II

Given an unsorted array `nums`, reorder it such that `nums[0] nums[2] .`

Example:
(1) Given `nums = [1, 5, 1, 1, 6, 4]`, one possible answer is `[1, 4, 1, 5, 1, 6]`.
(2) Given `nums = [1, 3, 2, 2, 3, 1]`, one possible answer is `[2, 3, 1, 3, 1, 2]`.

Note:
You may assume all input has valid answer.

Follow Up:
Can you do it in O(n) time and/or in-place with O(1) extra space?

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

This post is mainly about what I call "virtual indexing" technique (I'm sure I'm not the first who came up with this, but I couldn't find anything about it, so I made up a name as well. If you know better, let me know).

Solution

```
void wiggleSort(vector<int>& nums) {
    int n = nums.size();

    // Find a median.
    auto midptr = nums.begin() + n / 2;
    nth_element(nums.begin(), midptr, nums.end());
    int mid = *midptr;

    // Index-rewiring.
    #define A(i) nums[(1+2*(i)) % (n|1)]

    // 3-way-partition-to-wiggly in O(n) time with O(1) space.
    int i = 0, j = 0, k = n - 1;
    while (j <= k) {
        if (A(j) > mid)
            swap(A(i++), A(j++));
        else if (A(j) < mid)
            swap(A(j), A(k--));
        else
            j++;
    }
}
```

Explanation

First I find a median using `nth_element`. That only guarantees O(n) **average** time complexity and I don't know about space complexity. I might write this myself using O(n) time and O(1) space, but that's not what I want to show here.

This post is about what comes **after** that. We can use three-way partitioning to arrange the numbers so that those *larger than* the median come first, then those *equal to* the median come next, and then those *smaller than* the median come last.

Ordinarily, you'd then use one more phase to bring the numbers to their final positions to reach the overall wiggle-property. But I don't know a nice O(1) space way for this. Instead, I embed this right into the partitioning algorithm. That algorithm simply works with indexes 0 to n-1 as usual, but sneaky as I am, I rewire those indexes where I want the numbers to actually end up. The partitioning-algorithm doesn't even know that I'm doing that, it just works like normal (it just uses `A(x)` instead of `nums[x]`).

Let's say `nums` is `[10,11,...,19]`. Then after nth_element and ordinary partitioning, we might have this (15 is my median):

```
index:      0  1  2  3   4   5  6  7  8  9
number:    18 17 19 16  15  11 14 10 13 12
```

I rewire it so that the first spot has index 5, the second spot has index 0, etc, so that I might get this instead:

```
index:      5  0  6  1  7  2  8  3  9  4
number:    11 18 14 17 10 19 13 16 12 15
```

And 11 18 14 17 10 19 13 16 12 15 is perfectly wiggly. And the whole partitioning-to-wiggly-arrangement (everything after finding the median) only takes O(n) time and O(1) space.

If the above description is unclear, maybe this explicit listing helps:

Accessing `A(0)` actually accesses `nums[1]`.
Accessing `A(1)` actually accesses `nums[3]`.
Accessing `A(2)` actually accesses `nums[5]`.
Accessing `A(3)` actually accesses `nums[7]`.
Accessing `A(4)` actually accesses `nums[9]`.
Accessing `A(5)` actually accesses `nums[0]`.
Accessing `A(6)` actually accesses `nums[2]`.
Accessing `A(7)` actually accesses `nums[4]`.
Accessing `A(8)` actually accesses `nums[6]`.
Accessing `A(9)` actually accesses `nums[8]`.

Props to apolloydy's solution, I knew the partitioning algorithm already but I didn't know the name. And apolloydy's idea to partition to reverse order happened to make the index rewiring simpler.

written by StefanPochmann original link here

Solution 2

Solution

Roughly speaking I put the smaller half of the numbers on the even indexes and the larger half on the odd indexes.

```
def wiggleSort(self, nums):
    nums.sort()
    half = len(nums[::2])
    nums[::2], nums[1::2] = nums[:half][::-1], nums[half:][::-1]
```

Alternative, maybe nicer, maybe not:

```
def wiggleSort(self, nums):
    nums.sort()
    half = len(nums[::2]) - 1
    nums[::2], nums[1::2] = nums[half::-1], nums[:half:-1]
```

Explanation / Proof

I put the smaller half of the numbers on the even indexes and the larger half on the odd indexes, both from right to left:

```
Example nums = [1,2,...,7]      Example nums = [1,2,...,8]

Small half:  4 . 3 . 2 . 1      Small half:  4 . 3 . 2 . 1 .
Large half:  . 7 . 6 . 5 .      Large half:  . 8 . 7 . 6 . 5
--------------------------      --------------------------
Together:    4 7 3 6 2 5 1      Together:    4 8 3 7 2 6 1 5
```

I want:

- Odd-index numbers are larger than their neighbors.

Since I put the larger numbers on the odd indexes, clearly I already have:

- Odd-index numbers are larger than **or equal to** their neighbors.

Could they be "equal to"? That would require some number M to appear both in the smaller and the larger half. It would be the largest in the smaller half and the smallest in the larger half. Examples again, where S means some number smaller than M and L means some number larger than M.

```
Small half:  M . S . S . S      Small half:  M . S . S . S .
Large half:  . L . L . M .      Large half:  . L . L . L . M
--------------------------      --------------------------
Together:    M L S L S M S      Together:    M L S L S L S M
```

You can see the two M are quite far apart. Of course M could appear more than just twice, for example:

```
Small half:  M . M . S . S      Small half:  M . S . S . S .
Large half:  . L . L . M .      Large half:  . L . M . M . M
--------------------------      --------------------------
Together:    M L M L S M S      Together:    M L S M S M S M
```

You can see that with seven numbers, three M are no problem. And with eight numbers, four M are no problem. Should be easy to see that in general, with n numbers, floor(n/2) times M is no problem. Now, if there were more M than that, then my method would fail. But... it would also be impossible:

- If n is even, then having more than n/2 times the same number clearly is unsolvable, because you'd have to put two of them next to each other, no matter how you arrange them.
- If n is odd, then the only way to successfully arrange a number appearing more than floor(n/2) times is if it appears exactly floor(n/2)+1 times and you put them on all the even indexes. And to have the wiggle-property, all the other numbers would have to be larger. But then we wouldn't have an M in both the smaller and the larger half.

So if the input has a valid answer at all, then my code will find one.

written by StefanPochmann original link here

Solution 3

```
void wiggleSort(vector<int>& nums) {
    vector<int> sorted(nums);
    sort(sorted.begin(), sorted.end());
    for (int i=nums.size()-1, j=0, k=i/2+1; i>=0; i--)
        nums[i] = sorted[i&1 ? k++ : j++];
}
```

Sort and then write the smaller half of the numbers on the even indexes and the larger half of the numbers on the odd indexes, both from the back. Example:

```
Small half:    4 . 3 . 2 . 1 . 0 .
Large half:    . 9 . 8 . 7 . 6 . 5
----------------------------------
Together:      4 9 3 8 2 7 1 6 0 5
```

So write `nums` from the back, interweaving `sorted[0..4]` (indexed by `j`) and `sorted[5..9]` (indexed by `k`).

For more explanation/proof, see my equivalent Python solution.

written by StefanPochmann original link here