Wiggle Sort: Algorithmic Comparison

Task Description

Wiggle Sort

Given an integer array `nums`, reorder it such that:

$$nums[0] < nums[1] > nums[2] < nums[3] \ldots$$

Example 1:

- Input: `[1,5,1,1,6,4]`
- Output: `[1,6,1,5,1,4]`

Example 2:

- Input: `[1,3,2,2,3,1]`
- Output: `[2,3,1,3,1,2]`

Algorithm 1: One-Pass Greedy Swap (Pseudocode)

Idea: Traverse the array once, swapping adjacent elements to enforce the wiggle property.

Pseudocode:

```
for i = 0 to n-2:
    if (i is even and nums[i] >= nums[i+1]) or
       (i is odd and nums[i] <= nums[i+1]):
        swap(nums[i], nums[i+1])
```

Algorithm 1: One-Pass Greedy Swap (C Code)

Main Code (C):

```c
void wiggleSort1(int a[], int size){
  for(int i=0;i<size -1;i++){
    if ((i%2==1 && a[i]<=a[i+1])||
        (i%2==0 && a[i]>=a[i+1])){
      swap(&a[i],&a[i+1]);
    }
  }
}
```

Complexity: Time: $O(n)$, Space: $O(1)$ (in-place)

Algorithm 2: Sort and Interleave (Pseudocode)

Idea:

- Sort the array.
- Split into two halves and reverse each.
- Interleave largest of smaller half and largest of larger half.

Pseudocode:

```
sort(nums)
split nums into small (first half) and large (second half)
reverse small
reverse large
for i from 0 to n-1:
    if i is even:
        nums[i] = small[next]
    else:
        nums[i] = large[next]
```

Algorithm 2: Sort and Interleave (C Code)

Main Code (C):

```c
void wiggleSort(int* nums, int numsSize) {
    mergeSort(nums, 0, numsSize - 1);

    int half = (numsSize + 1) / 2;
    int* small = ...; // first half reversed
    int* large = ...; // second half reversed

    int i = 0, s = 0, l = 0;
    while (i < numsSize) {
        if (i % 2 == 0) nums[i++] = small[s++];
        else nums[i++] = large[l++];
    }
}
```

Complexity: Time: $O(n \log n)$, Space: $O(n)$

Complexity Comparison

Algorithm	Time Complexity	Space Complexity
Greedy One-Pass	$O(n)$	$O(1)$
Sort and Interleave	$O(n \log n)$	$O(n)$

Summary:

- The greedy approach is most efficient for both time and space.
- The sort-and-interleave algorithm is more robust for some edge cases or output forms.
- Both guarantee a valid *wiggle sort*.

Main Functions and Highlights

Main Functions:

- `swap()` — Helper to swap two elements.
- `wiggleSort1()` — Greedy one-pass method.
- `mergeSort()`, `merge()` — Used in the sort-and-interleave approach.
- `reverse()` — Reverse array halves.
- `wiggleSort()` — Sort and interleave method.

Conclusion

- Both algorithms correctly solve the Wiggle Sort problem.
- **Greedy One-Pass** (`wiggleSort1`):
  - Best for performance and memory.
- **Sort and Interleave** (`wiggleSort`):
  - Useful when a sorted or specific ordering is needed as a basis.