Algorithms project

Wiggle Sort

Team no. 162
Task no. 7

Team Members & IDs :

- ✧ Reem Tarek - 20210348
- ✧ Rana Ashraf - 20210334
- ✧ Roaa Waleed - 20210345
- ✧ Ziad Tamer - 20210361
- ✧ Omar Tarek - 20210606
- ✧ Ahmed Mohamed - 20210095

✧ What is Wiggle Sort?

- The Wiggle Sort algorithm takes an array of integers and modifies it in-place to reorder its elements in the wiggle pattern. It does this by iterating over the array and swapping adjacent elements if they do not satisfy the wiggle pattern.

✧ Project 7 instructions:

Given an integer array nums,
reorder it such that :

nums[0] < nums[1] > nums[2] < nums[3]....

You may assume the input array always has a valid answer.

Example 1:
Input: nums = [1,5,1,1,6,4]
Output: [1,6,1,5,1,4]
Explanation: [1,4,1,5,1,6] is also accepted

Example 2:
Input: nums = [1,3,2,2,3,1]
Output: [2,3,1,3,1,2] .

Constraints:

- 1 <= nums.length <= 5 * 104
- 0 <= nums[i] <= 5000
- It is guaranteed that there will be an answer for the given input nums.

Example 1

Example 1 **Pseudo-code:**

1. **FOR** each index i in the array from 0 to the second to last index:

2. **IF** i is even **AND** the current number is greater than the next number
   OR i is odd **AND** the current number is less than the next number:

3. **SWAP** the current number with the next number to create the wiggle pattern.

4. Print the sorted array.

Example 1 **Analysis:**

- The algorithm iterates over the array once and swaps adjacent elements as necessary to create the wiggle pattern.

Therefore, the time complexity of the algorithm is O(n), where n is the size of the input array.

Example 1 Implementation:
- Non-recursive implementation

```
#include <stdio.h>

void wiggleSort(int* nums, int numsSize) {
    for (int i = 0; i < numsSize - 1; i++) {
        // If the current index is even and the current number is greater than the next number
        // OR if the current index is odd and the current number is less than the next number
        if ((i % 2 == 0 && nums[i] > nums[i + 1]) || (i % 2 == 1 && nums[i] < nums[i + 1])) {
            // Swap adjacent elements to create the wiggle pattern
            int temp = nums[i];
            nums[i] = nums[i + 1];
            nums[i + 1] = temp;
        }
    }
}

void printArray(int* nums, int numsSize) {
    printf("[");
    for (int i = 0; i < numsSize; i++) {
        printf("%d", nums[i]);
        if (i != numsSize - 1) {
            printf(",");
        }
    }
    printf("]\n");
}

int main() {
    int nums[] = {1, 5, 1, 1, 6, 4};
    int numsSize = 6;

    printf("Input array: ");
    printArray(nums, numsSize);

    // Sort the array into the wiggle pattern
    wiggleSort(nums, numsSize);

    printf("Output array: ");
    printArray(nums, numsSize);

    return 0;
}
```

Output

```
/tmp/ZEAlOqTYK1.o
Input array: [1,5,1,1,6,4]
Output array: [1,5,1,6,1,4]
```

Example 2

Example 2 **Pseudo-code:**

1. Check if the input array size is less than or equal to 1.
   If yes, return.

2. Divide the input array into two halves (left and right).

3. Recursively call the wiggle sort function on the left half and the right half.

4. Merge the two sorted halves into a wiggle sorted array as follows:

   a. If the current element of the left half is greater than the next element, swap them.
   b. If the current element of the right half is less than the next element, swap them.

5. Return the wiggle sorted array.

Example 2 **Analysis:**

-The time complexity of this algorithm is **O(n log n)**, where n is the size of the input array.
-This is because the algorithm uses a recursive divide-and-conquer approach, with each level of recursion splitting the array in half and performing two linear passes over each half until the base case is reached.
-Then, the two halves are merged and sorted using two for loops that have O(n) time complexity each.

Therefore, the overall time complexity of the algorithm is O(n log n).

Example 2 Implementation:
- Recursive implementation

```
#include <stdio.h>

void wiggleSort(int* nums, int numsSize) {
    if (numsSize <= 1) {
        return;
    }
    // Divide the array into two halves
    int mid = numsSize / 2;

    // Recursively sort the first half
    wiggleSort(nums, mid);

    // Recursively sort the second half
    wiggleSort(nums + mid, numsSize - mid);

    // Merge the two sorted halves into a wiggle
    for (int i = 0; i < mid - 1; i++) {
        if (nums[i] > nums[i+1]) {
            int temp = nums[i];
            nums[i] = nums[i+1];
            nums[i+1] = temp;
        }
    }

    for (int i = mid; i < numsSize - 1; i++) {
        if (nums[i] < nums[i+1]) {
            int temp = nums[i];
            nums[i] = nums[i+1];
            nums[i+1] = temp;
        }
    }
}

void printArray(int* nums, int numsSize) {
    printf("[");
    for (int i = 0; i < numsSize; i++) {
        printf("%d", nums[i]);
        if (i < numsSize - 1) {
            printf(", ");
        }
    }
    printf("]\n");
}
```

```
int main() {
    int nums[] = {1, 3, 2, 2, 3, 1};
    int numsSize = sizeof(nums) / sizeof(nums[0]);
    printf("Input array: ");
    printArray(nums, numsSize);
    wiggleSort(nums, numsSize);
    printf("Output: ");
    printArray(nums, numsSize);
    return 0;
}
}
```

Input: nums = [1,3,2,2,3,1]

Output: [2,3,1,3,1,2] .

Comparison Between Example 1 and Example 2:

Example 1:

The first algorithm is a non-recursive implementation of wiggle sort.
The **wiggleSort** function iterates through the input array nums and checks if each pair of adjacent elements meets the wiggle condition.
If the condition is not met, the function swaps the elements to create the wiggle pattern.
The **printArray** function simply prints the input and output arrays.
The time complexity of this algorithm is **O(n)**, where n is the length of the input array nums. This is because the algorithm only iterates through the input array once.

Example 2:

The second algorithm is a recursive implementation of wiggle sort, It uses a divide-and-conquer approach.
The **wiggleSort** function recursively divides the input array nums into two halves until each subarray has a length of 1 or 0.
The function then performs the wiggle sort on each subarray and merges them back together to produce the final Output array.
The **printArray** function simply prints the input and output arrays.

The time complexity of this algorithm is **O(n log n)**, where n is the length of the input array nums.
This is because the algorithm recursively divides the input array in half until each subarray has a length of 1 or 0, which takes **O(log n)** time.
After the recursion, the algorithm performs the wiggle sort on each subarray, which takes **O(n)** time.
The total time complexity is therefore **O(n log n).**

In comparison, the non-recursive implementation of wiggle sort **(Example 1)** has a time complexity of **O(n)**, which is faster than the recursive implementation **(Example 2)**. However, the recursive implementation is better in if the input array is very large.