Median of two sorted arrays

```
project_name = 'Arrays_Median'
```

```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit(project=project_name)
```

```
[jovian] Detected Colab notebook...
[jovian] Please enter your API key ( from https://jovian.ai/ ):
API KEY: ··········
[jovian] Uploading colab notebook to Jovian...
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/viquiriglos/arrays-median
```

```
'https://jovian.ai/viquiriglos/arrays-median'
```

Problem Statement

Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.

Source: https://leetcode.com/problems/median-of-two-sorted-arrays/

Example1:

Input: nums1 = [1,3], nums2 = [2]

Output: 2.00000

Explanation: merged array = [1,2,3] and median is 2.

Example 2:

Input: nums1 = [1,2], nums2 = [3,4]

Output: 2.50000

Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

Example 3:

Input: nums1 = [0,0], nums2 = [0,0]

Output: 0.00000

Example 4: Input: nums1 = [], nums2 = [1] Output: 1.00000

Example 5: Input: nums1 = [2], nums2 = [] Output: 2.00000

Constraints

1. nums1.length == m
2. nums2.length == n
3. 0 <= m <= 1000
4. 0 <= n <= 1000
5. 1 <= m + n <= 2000
6. -106 <= nums1[i], nums2[i] <= 106

The Method

Here's the systematic strategy we'll apply for solving problems:

1. State the problem clearly. Identify the input & output formats.
2. Come up with some example inputs & outputs. Try to cover all edge cases.
3. Come up with a correct solution for the problem. State it in plain English.
4. Implement the solution and test it using example inputs. Fix bugs, if any.
5. Analyze the algorithm's complexity and identify inefficiencies, if any.
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

This approach is explained in detail in Lesson 1 of the course. Let's apply this approach step-by-step.

Solution

1. State the problem clearly. Identify the input & output formats.

The input is made of two sorted lists of numbers: nums1 and nums2. With this two arrays we need first to build a sorted array which include both input lists. Then, the output should be a number that lies in the middle of this merged-sorted array. This number does not necessarily belongs to any of the input arrays, it could be a floating point number. It's worth to notice that the result is neither necesarily the average value of the array. The median is stablished by the element in the middle of the merged-sorted array when the number of elements is odd or by the average of the two middle elements if the number of elements in the merged-sorted array is even.

Problem

> We have two sorted arrays as the input. We should merge them into a single sorted array. Then we should find the element in the middle of this merged array or the average of the two middle elements (in case the number of elements in the merged array is even).

Input

1. **nums1** (first sorted array)
2. **nums2** (second sorted array)

Output

1. **median** (a single number)

Based on the above, we can now create a signature of our function:

```
# Create a function signature here. The body of the function can contain a single state
def find_median(nums1, nums2):
    pass
```

Save and upload your work before continuing.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/viquiriglos/arrays-median
```

```
'https://jovian.ai/viquiriglos/arrays-median'
```

2. Come up with some example inputs & outputs. Try to cover all edge cases.

Our function should be able to handle any set of valid inputs we pass into it. Here's a list of some possible variations we might encounter:

1. The number of elements in the merged list is odd.
2. The number of elements in the merged list is even.
3. One of the lists is empty.
4. Both lists are empty.
5. One of the lists has only one element and the other is empty.

We'll express our test cases as dictionaries, to test them easily. Each dictionary will contain 2 keys: `input` (a dictionary itself containing one key for each argument to the function and `output` (the expected result from the function).

```
test1 = {
    'input': {
        'nums1':[1,3,7],
        'nums2':[2,5]
    },
    'output': 3
}
#merged=[1,2,3,5,7]
```

```
test2 = {
    'input': {
        'nums1':[1,3,7],
        'nums2':[2,5,6]
    },
    'output': 4
}
#merged=[1,2,3,5,6,7]
#the average of the elements in the middle is 4, i.e. (3+5)/2=4
```

```
test3 = {
    'input': {
        'nums1':[],
        'nums2':[2,4]
    },
    'output': 3
}
#merged=[2,4]
```

```
test4 = {
    'input': {
        'nums1':[],
        'nums2':[]
    },
    'output': []
}
#merged=[]
```

```
test5 = {
    'input': {
        'nums1':[4],
        'nums2':[]
    },
    'output': 4
}
#merged=[4]
```

Create one test case for each of the scenarios listed above. We'll store our test cases in an array called `tests` .

```
tests = [test1, test2, test3, test4, test5]
```

3. Come up with a correct solution for the problem. State it in plain English.

Our first goal should always be to come up with a *correct* solution to the problem, which may not necessarily be the most *efficient* solution. Come with a correct solution and explain it in simple words below:

1. Add the two arrays to form one single array
2. Sort this single array (we should take into account that there might be repeated elements).
3. Find the median
4. We could try bubble sort as first approach.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/viquiriglos/arrays-median
```

```
'https://jovian.ai/viquiriglos/arrays-median'
```

4. Implement the solution and test it using example inputs. Fix bugs, if any.

```
def sort_array_bubble(array):
    n=len(array)

    # 4. Repeat the process n-1 times
    for _ in range(n - 1):

        # 1. Iterate over the array (except last element)
        for i in range(n - 1):

            # 2. Compare the i-th number with the following one
            if array[i] > array[i+1]:

                # 3. Swap the two elements in case they are not in increasing order
                array[i], array[i+1] = array[i+1], array[i]

    return array
```

```
def median_bubble(nums1, nums2):

    m=len(nums1)
    n=len(nums2)
    mid=(n+m)//2

    if m==0 and n==0:
      return []

    merged=nums1+nums2

    sorted=sort_array_bubble(merged)

    if (n+m)%2!=0:
        return sorted[mid]
    else:
        average=(sorted[mid]+sorted[mid-1])/2
        return average
```

We can test the function by passing the input to it directly or by using the `evaluate_test_case` function from `jovian`.

```
from jovian.pythondsa import evaluate_test_case
```

Evaluate your function against all the test cases together using the `evaluate_test_cases` (plural) function from `jovian`.

```
from jovian.pythondsa import evaluate_test_cases
```

```
evaluate_test_cases(median_bubble, tests)
```

```

TEST CASE #0

Input:
{'nums1': [1, 3, 7], 'nums2': [2, 5]}

Expected Output:
3


Actual Output:
3

Execution Time:
0.015 ms

Test Result:
PASSED


TEST CASE #1

Input:
{'nums1': [1, 3, 7], 'nums2': [2, 5, 6]}

Expected Output:
4
```

```
Actual Output:
4.0

Execution Time:
0.02 ms

Test Result:
PASSED


TEST CASE #2

Input:
{'nums1': [], 'nums2': [2, 4]}

Expected Output:
3


Actual Output:
3.0

Execution Time:
0.024 ms

Test Result:
PASSED


TEST CASE #3

Input:
{'nums1': [], 'nums2': []}

Expected Output:
[]


Actual Output:
[]

Execution Time:
0.003 ms
```

```
Test Result:
PASSED


TEST CASE #4

Input:
{'nums1': [4], 'nums2': []}

Expected Output:
4


Actual Output:
4

Execution Time:
0.005 ms

Test Result:
PASSED


SUMMARY

TOTAL: 5, PASSED: 5, FAILED: 0
```

```
[(3, True, 0.015),
 (4.0, True, 0.02),
 (3.0, True, 0.024),
 ([], True, 0.003),
 (4, True, 0.005)]
```

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/viquiriglos/arrays-median
```

```
'https://jovian.ai/viquiriglos/arrays-median'
```

5. Analyze the algorithm's complexity and identify inefficiencies, if any.

The sort_array_bubble() function used inside median_bubble gives a complexity of (n+m)^2 to the median_bubble algorithm. We must reduce the complexity of the sorting algorithm to make a faster search of the median.

```
complexity='O(N^2)'
```

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/viquiriglos/arrays-median
```

```
'https://jovian.ai/viquiriglos/arrays-median'
```

6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

To improve the efficiency of the algorithm we could sort the array while we are merging both inputs. This will be fine, given the fact that both input arrays are already sorted in increasing order.

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "aakashns/python-problem-solving-template" on
https://jovian.ai/
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/aakashns/python-problem-solving-
template
```

```
'https://jovian.ai/aakashns/python-problem-solving-template'
```

7. Come up with a correct solution for the problem. State it in plain English.

Come with the optimized correct solution and explain it in simple words below:

1. Let's check that at least one of the arrays is not empty
2. As both input arrays are in increasing order, we will go through both arrays comparing one element in the first array with an element in the second array.
3. We will store the smallest element in a new array called merged.
4. We will increase in 1 the index of the array to which this stored element belonged and we will perform a new comparison.
5. When we rich the end of one of the arrays, we will add the remaining part of the other array, the tail.
6. In this way, we will obtain a sorted array which merges both input arrays.
7. To understand this procedure it's suggested to watch the video in the following link: https://www.youtube.com/watch?v=GW0USDwhBgo&t=28s

8. Implement the solution and test it using example inputs. Fix bugs, if any.

```python
def median_optimized(nums1, nums2):

    m=len(nums1)
    n=len(nums2)
    mid=(n+m)//2

    if m==0 and n==0:
      return []

  # List to store the results
    merged = []

    # Indices for iteration
    i, j = 0, 0

    # Loop over the two arrays
    while i < m and j < n:

        # Include the smaller element in the result and move to next element
        if nums1[i] <= nums2[j]:
            merged.append(nums1[i])
            i += 1
        else:
            merged.append(nums2[j])
            j += 1

    # Get the remaining parts
    nums1_tail = nums1[i:]
    nums2_tail = nums2[j:]

    # The final merged and sorted array will be:
    merged_sorted = merged + nums1_tail + nums2_tail

    if (n+m)%2!=0:
        return merged_sorted[mid]
    else:
        return (merged_sorted[mid]+merged_sorted[mid-1])/2
```

```python
evaluate_test_cases(median_optimized, tests)
```

```
TEST CASE #0


Input:
{'nums1': [1, 3, 7], 'nums2': [2, 5]}


Expected Output:
```

```
3


Actual Output:
3

Execution Time:
0.011 ms

Test Result:
PASSED


TEST CASE #1

Input:
{'nums1': [1, 3, 7], 'nums2': [2, 5, 6]}

Expected Output:
4


Actual Output:
4.0

Execution Time:
0.01 ms

Test Result:
PASSED


TEST CASE #2

Input:
{'nums1': [], 'nums2': [2, 4]}

Expected Output:
3


Actual Output:
3.0
```

```
Execution Time:
0.005 ms

Test Result:
PASSED


TEST CASE #3

Input:
{'nums1': [], 'nums2': []}

Expected Output:
[]


Actual Output:
[]

Execution Time:
0.003 ms

Test Result:
PASSED


TEST CASE #4

Input:
{'nums1': [4], 'nums2': []}

Expected Output:
4


Actual Output:
4

Execution Time:
0.005 ms

Test Result:
```

```
PASSED



SUMMARY

TOTAL: 5, PASSED: 5, FAILED: 0
```

```
[(3, True, 0.011),
 (4.0, True, 0.01),
 (3.0, True, 0.005),
 ([], True, 0.003),
 (4, True, 0.005)]
```

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Please enter your API key ( from https://jovian.ai/ ):
API KEY: ··········
[jovian] Uploading colab notebook to Jovian...
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/viquiriglos/arrays-median
```

```
'https://jovian.ai/viquiriglos/arrays-median'
```

9. Analyze the algorithm's complexity and identify inefficiencies, if any.

At this point the complexity is order min(n,m) which is O(N) in big O notation. This is a great improvement but, according to leetcode, we still could reduce the run time complexity till O(log (m+n)). Given the form of the complexity, we could assume a strategy of the type "divide-n conquer" should be used to achieve that result.

One thing that we should take into account before start coding, is that the arrays might not be "overlaped". This means that, for instance, if the last element in one array is smaller than the first element in the other array (nums1[-1] < nums2[0] or nums2[-1] < nums1[0]) the merged array will be just the concatenations of both arrays (merged = nums1+nums2 or nums2+nums1).

In the case that they would be partially or completely overlaped, we should go spliting and merging both arrays to find the "intersection" of both arrays. Only this intersection should be sorted.

We can create a function for spliting the arrays (called "split_array") and another function to merge the sub-arrays from both input arrays, while sorting them ("merge"). Later on, we'll build another function that performs these tasks recursively ("sorted_divide_n") and then put all together to obtain the result by means of the function "median_optimized2". The latter will handle all the edge cases previosly to perform any recursive task.

```python
def split_array(array):

    n = len(array)

    # Terminating condition (list of 0 or 1 elements)
    if n <= 1:
        return array
    # Get the midpoint
    mid = n // 2

    # Split the array into two halves
    a0=array[:mid]
    a1=array[mid:]

    return a0, a1
```

```
def merge(nums1, nums2):

    #Check if is there overlaping between both arrays:
    if nums1[-1]<= nums2[0]:
        merged=nums1+nums2

    elif nums2[-1]<= nums1[0]:
        merged=nums2+nums1

    else:
        # List to store the results
        merged = []

        # Indices for iteration
        i, j = 0, 0

        # Loop over the two lists
        while i < len(nums1) and j < len(nums2):

            # Include the smaller element in the result and move to next element
            if nums1[i] <= nums2[j]:
                merged.append(nums1[i])
                i += 1
            else:
                merged.append(nums2[j])
                j += 1

        # Get the remaining parts
        nums1_tail = nums1[i:]
        nums2_tail = nums2[j:]
        merged= merged + nums1_tail + nums2_tail

    # Return the final merged array
    return merged
```

```
def sorted_divide_n(nums1, nums2):
    #We assign the length of each array to a variable
    m=len(nums1)
    n=len(nums2)

    # Terminating condition (list of 0 or 1 elements)
    if m <= 1 or n <= 1:
        sorted = merge(nums1, nums2)
        return sorted

    #We split each input array using the function split_array() defined below
    A0, A1 = split_array(nums1)
    B0, B1 = split_array(nums2)

    #We perform this procedure recursively
    A=sorted_divide_n(A0,A1)
    B=sorted_divide_n(B0,B1)

    # We then combine the results
    sorted_left =  merge(A0, B0)
    sorted_right = merge(A1, B1)

    sorted = merge(sorted_left, sorted_right)

    return sorted
```

```
def median_optimized2(nums1, nums2):

    #We assign the length of each array to a variable
    m=len(nums1)
    n=len(nums2)

    #We stablish a condition when both arrays are empty
    if m==0 and n==0:
        return []

    #We stablish a condition when one of the arrays is empty
    if m==0 and n!=0:
        sorted=nums2
    elif n==0 and m!=0:
        sorted=nums1

    #Now, if none of the arrays are empty we merge them
    else:
        sorted = sorted_divide_n(nums1, nums2)

   #Find the median
    mid=(n+m)//2

    if (n+m)%2!=0:
        return sorted[mid]
    else:
        return (sorted[mid]+sorted[mid-1])/2
```

Let's try out this solution:

```
evaluate_test_cases(median_optimized2, tests)
```

```
TEST CASE #0

Input:
{'nums1': [1, 3, 7], 'nums2': [2, 5]}

Expected Output:
3


Actual Output:
3

Execution Time:
0.025 ms

Test Result:
PASSED


TEST CASE #1

Input:
{'nums1': [1, 3, 7], 'nums2': [2, 5, 6]}

Expected Output:
4


Actual Output:
4.0

Execution Time:
0.023 ms

Test Result:
PASSED


TEST CASE #2

Input:
{'nums1': [], 'nums2': [2, 4]}
```

```
Expected Output:
3


Actual Output:
3.0

Execution Time:
0.004 ms

Test Result:
PASSED


TEST CASE #3

Input:
{'nums1': [], 'nums2': []}

Expected Output:
[]


Actual Output:
[]

Execution Time:
0.002 ms

Test Result:
PASSED


TEST CASE #4

Input:
{'nums1': [4], 'nums2': []}

Expected Output:
4
```

```
Actual Output:
4

Execution Time:
0.004 ms

Test Result:
PASSED



SUMMARY

TOTAL: 5, PASSED: 5, FAILED: 0
```

```
[(3, True, 0.025),
 (4.0, True, 0.023),
 (3.0, True, 0.004),
 ([], True, 0.002),
 (4, True, 0.004)]
```

Let's try some more cases:

```
test6 = {
    'input': {
        'nums1':[1,3,5,7],
        'nums2':[0,2,8,10,11]
    },
    'output': 5
}
#merged=[0, 1, 2, 3, 5, 7, 8, 10, 11]
```

```
test7 = {
    'input': {
        'nums1':list(range(1,10,2)),
        'nums2':list(range(0,10,2))
    },
    'output': 4.5
}
#merged=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
new_tests=[test6, test7]
```

```
evaluate_test_cases(median_optimized2, new_tests)
```

```
TEST CASE #0
```

```
Input:
{'nums1': [1, 3, 5, 7], 'nums2': [0, 2, 8, 10, 11]}

Expected Output:
5


Actual Output:
5

Execution Time:
0.037 ms

Test Result:
PASSED


TEST CASE #1

Input:
{'nums1': [1, 3, 5, 7, 9], 'nums2': [0, 2, 4, 6, 8]}

Expected Output:
4.5


Actual Output:
4.5

Execution Time:
0.08 ms

Test Result:
PASSED


SUMMARY

TOTAL: 2, PASSED: 2, FAILED: 0

[(5, True, 0.037), (4.5, True, 0.08)]
```

As we have to divide by 2 the length of each input array in each step, we reach a complexity log(n+m).

If you found the problem on an external platform, you can make a submission to test your solution.

Share your approach and start a discussion on the Jovian forum: https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/78

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/viquiriglos/arrays-median
```

```
'https://jovian.ai/viquiriglos/arrays-median'
```