Problem #1

a.

b. The Tower of Hanoi problem is solved using recursion. The recurrence relation or the formula to calculate the minimum number of moves is T(n) = 2*T(n-1) + 1.

An explanation of why T(4) = 15 is possible:

1.Move n-1 (3) disks to an auxiliary peg: T(3) = 7 moves.
2.Move the largest disk to the destination peg: 1 move.
3. Move n-1 (3) disks from auxiliary to destination peg: T(3) = 7 moves.

Total moves: 7 + 1 + 7 = 15.

It's impossible to solve in less than 15 moves due to the problem's recursive nature. The solution for n disks depends on the solution for n-1 disks. You need to move n-1 disks twice and the nth disk once, totaling 2*T(n-1) + 1 moves, which is always greater than T(n-1). Hence, solving in less than T(n) moves is impossible.

We can also prove this by proof of contradiction by assuming that the minimum moves CAN be less than 15, and then use the recurrence formula to show that the least is actually 15 which contradicts with our assumption, hence the given statement must be true.

Cite: https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/

c. The following recurrence relation is derived for the Tower of Honai program, T(n) = 2*T(n-1) + 1, as shown in part b. Again, as explained in part b, the recurrence relation T(n) = 2*T(n-1) + 1 holds for the Tower of Hanoi problem due to the nature of the problem itself.

The Tower of Hanoi problem involves moving n disks from one peg to another, with the constraint that a larger disk cannot be placed on top of a smaller disk and You need to move n-1 disks twice and the nth disk once, totaling 2*T(n-1) + 1 moves. This is why the recurrence relation T(n) = 2*T(n-1) + 1 holds for the Tower of Hanoi problem.

The recurrence relation for the Tower of Hanoi problem is T(n) = 2*T(n-1) + 1 with the base case T(1) = 1.
To solve this recurrence relation, we can use back substitution.

Let's start with T(n):
T(n) = 2*T(n-1) + 1

Substitute T(n-1) from the recurrence relation: T(n) = 2*(2*T(n-2) + 1) + 1 = 4*T(n-2) + 2 + 1
Substitute T(n-2) from the recurrence relation: T(n) = 2*(2*(2*T(n-3) + 1) + 1) + 1 = 8*T(n-3)
+ 4 + 2 + 1

Continuing this process, we can see a pattern emerging. Assume we have reached some K steps:

T(n) = 2^k*T(n-k) + 2^(k-1) + 2^(k-2) + ... + 2^0

This continues until k = n-1, at which point T(n-k) = T(1) = 1. So, we get:

T(n) = 2^(n-1)*T(1) + 2^(n-2) + 2^(n-3) + ... + 2^0
T(n) = 2^(n-1) + 2^(n-2) + 2^(n-3) + ... + 2^0

This is a geometric series with n terms, the sum of which is given by the formula:
a*(r^n - 1)/(r – 1). In this case, a = 1, r = 2, and n = n. So, we get:

T(n) = 1*(2^n - 1)/(2 - 1)
T(n) = 2^n – 1

So, the solution to the recurrence relation T(n) = 2*T(n-1) + 1 for the Tower of Hanoi problem is T(n) = 2^n - 1, which is true for all integers n ≥ 1.

d. First substitute n=log(m) into the recurrence relation T(n)=2T(n−1)+1.

Let's denote S(m)=T(log(m)), Then we can rewrite the original recurrence as S(m)=2S(m/2)+1. This is a standard form of recurrence that can be solved by the Master Theorem. In our case, a=2 b=2, and f(m)=1. So, c = log a with base b = log 2 with base 2 = 1. And f of m = O(m^0) = O(1). we are in the first case of the Master Theorem, and so S(m)=theta(m^ log a with base b)=theta(m).

However, this is the complexity in terms of m, not n. Since we made the substitution n=log(m) or equivalently m=2n ,we need to express our result in terms of n. Thus, S(m)=theta(m)=theta(2n). So, T(n)=theta(2n) when n=log(m).
To show that T(n) = theta(2^n) from part c, we need to show that there exist positive constants c1, c2, and n0 such that for all n >= n0, **c1 * 2^n <= T(n) <= c2 * 2^n**.
From T(n) = 2^n - 1, we can see that for n >= 1, 2^n - 1 <= 2^n and 2^n - 1 >= 1/2 * 2^n.
Therefore, we can choose c1 = 1/2, c2 = 1, and n0 = 1 to satisfy the conditions of theta notation.

Problem #2

a.

- First, we need to choose a pivot element. There are different ways to do this, but for simplicity, we will always choose the last element of the subarray as the pivot. So, for the input array A, the pivot is 4.
- Next, we need to partition the array around the pivot, such that all elements smaller than or equal to the pivot are on the left side, and all elements larger than the pivot are on the right side. We will do this in the following steps in order:
    - Initialize partition index i to -1, which tracks the rightmost boundary of elements less than the pivot
    - Iterate through the array from the first element to the last element right before the pivot
    - When an element less than or equal to the pivot is encountered, increment i and swap the current element with the element at i.

So, we'll apply these steps to A:
A = (3, 1, 5, 7, 6, 2, 4), i = -1 (before the start of the array)
Compare each element to the pivot, which is 4 in this case, and perform swaps where needed:

- A[0] is 3 (<= 4), so increment i to 0 and swap A[0] (there is no change here, just to explain)
- A[1] is 1 (<= 4), so increment i to 1 and swap A[1] with A[1] (no change)
- A[2] is 5 (> 4), so no change
- A[3] is 7 (> 4), so no change
- A[4] is 6 (> 4), so no change
- A[5] is 2 (<= 4), so increment i to 2 and swap A[2] with A[5]. This gives us:
    - A = [3, 1, 2, 7, 6, 5, 4]
- After all elements are processed, swap pivot with element at i+1. Since I ended at position 2 after the last swap, we swap the pivot (4) with element at position 3, which gives us:
    - A = [3, 1, 2, 4, 6, 5, 7]

Now, we apply Quicksort to the two subarrays resulting from the partition, [3, 1, 2] (left) and [6, 5, 7] (right). For the left subarray, we choose the last element as the pivot, which is 2, and partition accordingly:

- i = -1 (subarray start)
- A[0] is 3 (> 2), so no change
- A[1] is 1 (<= 2), so increment i to 0 and swap A[0] with A[1]. This gives us:
    - A = [1, 3, 2, 4, 6, 5, 7]
- After all elements are processed, swap the pivot with element at i+1, which gives us:
    - A = [1, 2, 3, 4, 5, 6, 7]

For the right subarray, we choose the last element as the pivot, which is 7. Normally we would partition here, but all elements are less than the pivot, and are already in the correct positions, so no swaps are necessary.

Finally, we have sorted all the subarrays, and we have the output array (1,2,3,4,5,6,7).

b. In the Quicksort algorithm, if we always choose the last element as the pivot (under Lomuto partition scheme), the partition process will require $n-1$ comparisons for an array of size n. This is because each of the other $n - 1$elements need to be compared with the pivot to determine its position. With the array A=[1,2,3,4,5,6,7], let's now break down the number of comparisons:

1. For the first partition, the pivot is 7. We compare it with the other 6 elements. So, we have 6 comparisons.
2. In the next step, the array is split into two sub-arrays: [1,2,3,4,5,6] and [7]. The second sub-array has only one element (the pivot itself), so no comparisons are needed there. In the first sub-array, the pivot is now 6. We compare this one with the other 5 elements, adding 5 more comparisons.
3. We continue this process with the sub-array [1,2,3,4,5], adding 4 comparisons, and so on down to a single element.

If we total the sum of the first six natural numbers, we get the final number of comparisons which is 6+5+4+3+2+1=21. This arises since each time the array is partitioned, the size of the subarray that still needs to be partitioned decreases by one, and we compare each element in the subarray to the pivot only one time.

c. The Quicksort algorithm achieves the minimum number of comparisons when the partitioning is balanced, which occurs when the pivot element is the median. However, we are required for the pivot to always be the last element of the array instead, so we must structure the array such that the last element is the median after each partition.

Consider A = [3, 1, 2, 4, 6, 5, 7]. If we always choose the last element as the pivot, the partitioning steps would be as follows:

1. Choose 4 as the pivot. The array is split into:

   [3, 1, 2] and [6, 5, 7]. This requires 6 comparisons as we are comparing each of the other elements to the pivot.

2. Choose 2 as the pivot for the first sub-array [3, 1, 2] and 6 as the pivot for the second sub-array [6, 5, 7]. This results in [1] and [3] from the first sub-array, and [5] and [7] from the second sub-array. This requires 2 comparisons for the first sub-array, and 2 comparisons for the second sub-array. Each sub-array requires n-1 comparisons, so the for the first sub-array [3,1,2], which has 3 elements, 2 comparisons are needed. Similarly, for the second sub-array [6, 5, 7], which has 3 elements, 2 comparisons are needed as well.

3. Since all remaining arrays have only one element, no more comparisons are needed.

So, the total number of comparisons is 6 (from initial partitioning) + 2 (from first sub-array) + 2 (from second sub-array) = 10 comparisons. This is the minimum number of comparisons required for quicksort with the last element as the pivot on a 7-element array, because by arranging the array so that the last element is the median at each partitioning step, we ensure that the array is split as evenly as possible at each step, minimizing the number of comparisons needed. There also cannot exist a 7-element array which requires fewer comparisons than this when using the Lomuto partition scheme, since any less balanced partitioning would increase the number of comparisons as having sub-arrays of different sizes would be largely inefficient.

d.

Best case: The minimum number of comparisons in Quicksort occurs when each partition splits the array into two equal halves at each step. This is the best case scenario for Quicksort. If we have an array of size $n = 2^k - 1$, where *k* is a positive integer, the array divides perfectly. The number of comparisons at each level *i* of the recursion is $n - 2^i$, as the array halves at each level. The total number of comparisons *C(n)* across all the levels is the sum of the comparisons over *k* levels of recursion, which can be calculated as:

$$C(n) = \sum_{i=0}^{k-1} (n - 2^i)$$

$$C(n) = k * n - (2^k - 1)$$

By substituting $k = \log_2(n + 1)$, we get:

$$C(n) = n \log_2(n + 1) - n$$

Since this is the base case scenario, we can say that, this represents an $O(n \log n)$ time complexity for the best-case scenario of quicksort. Since, the equation has nlogn and n, therefore, nlogn beats the growth of n asymptotically and we can conclude and show that the time complexity of best case scenario is nlogn.

Worst case: The maximum number of comparisons in Quicksort occurs when the array is split so that one part has all the elements except the pivot. This would result in a scenario with the most unbalanced partitions possible, where one subarray has n-1 elements, then the other has 0 elements with each partition. If we have an array of size n = 2^k - 1, where k is a positive integer, the array can be divided in such a way at each step of the recursion.

In this case, the total number of comparisons *C(n)* is the sum of the comparisons for each of the *n*-1 levels of recursion. Which is given by:

$$C(n) = \sum_{i=1}^{n-1} i$$

$$C(n) = \frac{n * (n-1)}{2}$$

So, the maximum number of comparisons required by Quicksort for an array of size n = 2^k - 1 is n^2/2 - n/2. From this equation, we can conclude that the time complexity of worst case scenario as well. Since, the most dominant term is n^2, therefore, in the worst case, the running time of Quicksort is $O(n^2)$.

problem #3: Kevin Chen

a. Prove $T(n) = \Theta(n^{\log_y(x)})$, when $z < \log_y(x)$, given that $T(n) = xT(n/y) + n^z$, we have the form $T(n) = aT(n/b) + f(n)$ with $a = x$, $b = y$, $f(n) = n^z$.
Now, applying master theorem, we compare $f(n)$ with $n^{\log_b(a)}$, which in this case is $n^{\log_y(x)}$. Since we have the condition $z < \log_y(x)$, it's implied that $f(n)$ grows slower than $n^{\log_y(x)}$, or $f(n) = O(n^{\log_y(x)-\epsilon})$ for $\epsilon > 0$. This aligns with case 1 of the Master Theorem, where $f(n)$ is asymptotically less than $n^{\log_b(a)}$, and recursive $aT(n/b)$ dominates the solution.
According to master theorem, $T(n)$ is dominated by the cost of the recursive calls, leading us to conclude to $T(n) = \Theta(n^{\log_b(a)})$, and substituting our values gives $T(n) = \Theta(n^{\log_y(x)})$, if $z < \log_y(x)$.
This is in agreement with the original answer which completes the proof.

b. If $z = \log_y(x)$, prove $T(n) = \Theta(n^{\log_y(x)} \cdot \log_y n)$
Given that $f(n) = n^z$, we can say that $f(n) = n^{\log_y(x)}$. Master theorem requires us to express this as $n^{\log_b(a)}$. And, $b = y$ since we're dividing by y with each recursive call. And x recursive calls are made, so $a = x$.
Thus, $z = \log_b(a)$.
Now, case 2 of master theorem applies when $f(n) = \Theta(n^{\log_b(a)} \log^k(n))$ for some $k \geq 0$. Then, $T(n) = \Theta(n^{\log_b(a)} \log^{k+1}(n))$. So since $f(n) = \Theta(n^{\log_y(x)})$, it follows that:

$$T(n) = \Theta(n^{\log_y(x)} \log^{0+1}(n))$$
$$T(n) = \Theta(n^{\log_y(x)} \log(n))$$

This matches the problems request, thus completing the proof.

c. Given recurrence relation $T(n) = xT(n/y) + n^z$, we have the following:
$a = x$, number of subproblems at each recursion level
$b = y$, factor which is divided by problem size
$f(n) = n^z$, non-recursive work at each step
Then, case 3 of master theorem states if $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$, and if conditions $af(n/b)$ for some $k < 1$ and sufficiently large n holds true, then $T(n) = \Theta(f(n))$.
Since $z > \log_y(x)$, n^z grows faster than $n^{\log_y(x)}$, so $f(n) = n^z$ satisfies case 3 conditions.
To satisfy conditions of case 3, we know that $x \cdot f(\frac{n}{y}) = x \cdot (\frac{n}{y})^z \leq k \cdot n^z$ for some $k < 1$ and sufficiently large n. Since $z > \log_y(x)$, we substitute z with $\log_y(x)$ and get:

$$x \cdot y^{-z} = x \cdot y^{-\log_y(x)-\epsilon} = x \cdot \frac{1}{x} \cdot y^{-\epsilon} = y^{-\epsilon} < 1.$$

So, the regularity condition is satisfied since $y^{-\epsilon} < 1$ for $\epsilon > 0$. Thus all regularity conditions are satisfied and we can conclude $T(n) = \Theta(f(n)) = \Theta(n^z)$ by master theorem case 3. for $z > \log_y(x)$, $T(n) = \Theta(n^z)$.

d. Given recurrence relation for the Strassen's algorithm is:$T(n) = 7T\left(\frac{n}{2}\right) + n^2$.

We can apply the Master Theorem to solve this recurrence relation, which applies to relations of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a \geq 1, b > 1$, and $f(n)$ is an asymptotically positive function. In this case, a = 7, b = 2, and $f(n) = n^2$, indicating that z, which is the exponent of n in the non-recursive term f(n), equals 2. So the critical exponent to compare with z is $\log_b(a) = \log_2(7)$.

Now, comparing exponent z of f(n) with $\log_b(a)$, we can see that z = 2 is indeed less than $\log_2(7)$. According to Master Theorem, if $z < \log_b(a)$, the running time T(n) is dominated by the relation's recursive portion. So we can conclude that $T(n) = \theta(n^{\log_b(a)})$.

Next, recall that the running time of Strassen's algorithm is $T(n) = \theta(n^{\log_2(7)})$. Since $\log_2(7)$ comes out to approximately 2.81, which is less than three, we can now conclude that Strassen's algorithm has a lower asymptotic time complexity than the standard matrix multiplication algorithm, which has a runtime of $\theta(n^3)$. In other words, Strassen's algorithm is faster than the standard matrix multiplication algorithm for large n.

Problem #4

a.The described algorithm isn't linear, since the problem implies an additional sorting step. Merging two sorted arrays, A and B, each containing n elements, into a single sorted array C includes this merging process: Comparing elements from both arrays, then inserting the lesser one into the new C array. This would take O(n) time since each element from both arrays are all considered only one time, which results in 2n comparisons and insertions.
After the merge, the C array is sorted. However, the problem details that after the merge, the combined array should be sorted to produce a new array C with 2n elements. The median of the combine dataset is the average of the nth and (n+1)th elements, since the arrays have 2n distinct elements total. If we were to access those two elements from array C, which again is already sorted, it would be O(1) or a constant time operation as there are no further comparisons or movements made.
So, while the merge operation itself is linear, and finding the median from a sorted array takes constant time, the actual overall algorithm's time complexity is dominated by the sort step, which would make the algorithm linearithmic, or $O(n \log n)$.

I have written a pseudocode demonstration for merging two sorted arrays below, but note that if after merging, sorting is still required (as the problem states), this would make the time complexity to linearithmic:

```
void merge(int A[], int B[], int n, int C[]) {
   int i = 0, j = 0, k = 0;
   while (i < n && j < n) {
      if (A[i] < B[j]) {
         C[k++] = A[i++];
      } else {
         C[k++] = B[j++];
      }
   }
   while (i < n) {
      C[k++] = A[i++];
   }
   while (j < n) {
      C[k++] = B[j++];
   }
//Array C is sorted here
}

double findMedian(int C[], int n) {
//Since array C is sorted, the median is the average of two middle elements
      return (C[(n/2) - 1] + C[n/2]) / 2.0;
}
```

b. In a sorted array, elements are arranged in such a way that any which element at index i will be less than or equal to any element at an index greater than i. In this case, since A[5] < B[5], and the arrays have an odd number of elements, we know that the median of the combined array will be between elements of A[5:9] and B[1:5]. This is because A[5] is the middle element of array A, meaning there must be an equal number of elements less than A[5] in array A and greater than A[5] in array B. Similarly, B[5] is the middle element of B,

meaning that an equal number of elements in B are greater than B[5] and less than B[5] in A. Therefore, the combined array's median is the average of A[5] and B[5], and this is confirmed by the problem statement where the median is 55, the average of the two middle numbers from the combined sub-arrays A[5:9] and B[1:5].

c. If A[(n+1)/2] > B[(n+1)/2], then the median of A is greater than the median of B. Since both arrays are sorted, this means that the elements in B going up to B[(n+1)/2] are all less than or equal to the elements in A from A[(n+1)/2] onwards. But, this doesn't mean that the median of that combined array automatically directs to A[(n+1)/2] or B[(n+1)/2]. Instead, the media would be between inclusive A[1] and A[(n+1)/2]. This is due to the combined array's median being at index n or n+1, and since there are n elements in A less than or equal to A[(n+1)/2] and at least (n+1)/2 elements in B less than A[(n+1)/2], then the median must be in the lower half of A or the upper of B.
Next, if A[(n+1)/2] < B[(n+1)/2], the elements in A up to A[(n+1)/2] and the elements in B from B[(n+1)/2] and beyond would contain the median. The median in this case is also greater than or equal to A[(n+1)/2], and less than or equal to B[(n+1)/2], since there are n elements in B greater than or equal to B[(n+1)/2] and at least (n+1)/2 elements in A greater than B[(n+1)/2], which puts the median in the upper half A or lower half of B, the reverse of the previous example.

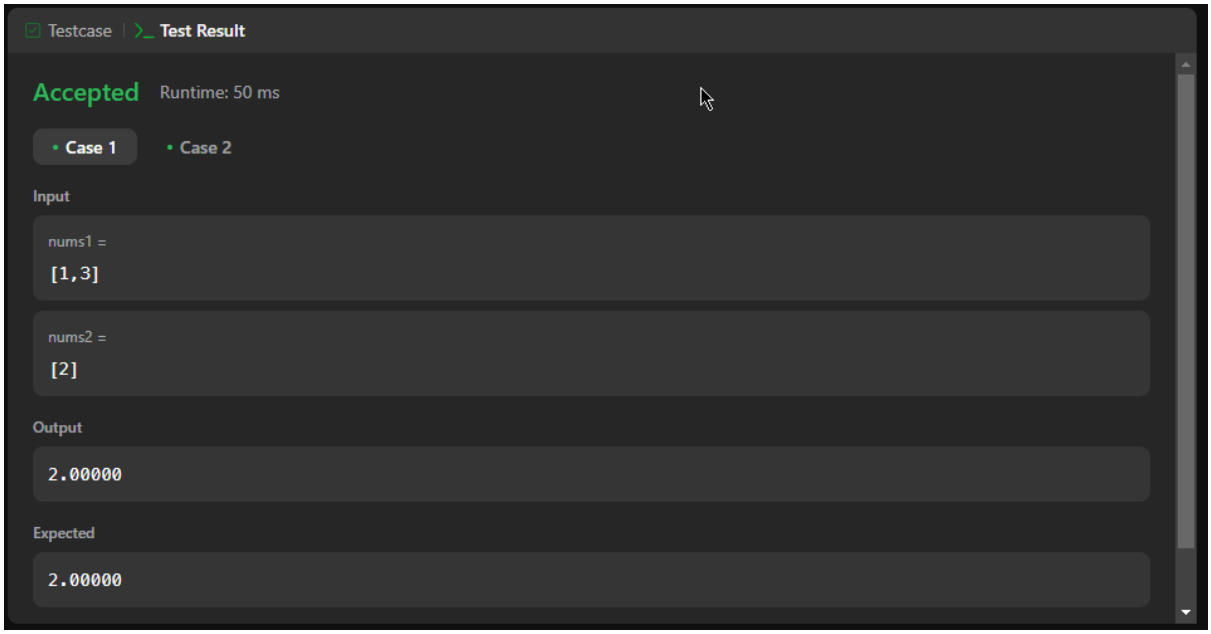
d. The algorithm to find the median of two sorted arrays A and B when n = 2^k + 1 for some k >= 1 can be designed using a divide-and-conquer approach. We can make it compare the medians of the arrays A and B and eliminate the half of each array that cannot contain the median of the combined array. Algorithm therefore is:
1. Calculate the mid index as (n + 1)/2.
2. Compare the elements at both arrays' mid index A[mid] and B[mid].
3. If A[mid] > B[mid], then the median must lie in the first half of A up to the mid index, or in the second half of B from the mid index onwards. So, recursively find the median in A[1 : mid] and B[mid : n].
4. If A[mid] < B[mid], then the median must lie in the second half of A from the mid index onwards, or the first half of B up to the mid index. So, recursively find the median in A[mid : n] and B[1 : mid].
5. The recursion continues until we reach the base case, where n=1. Here, the median is the smaller one between the elements A[1] and B[1].
6. The median of the combined array is the median of the sub-arrays determined by the above steps.
(Assuming arrays are 1-indexed).
This algorithm is guaranteed to return the correct output because at each step, it eliminates the half of each array that cannot contain the median of the combined array. This is based on the properties of sorted arrays and the definition of the median. The running time of the algorithm is O(logn) because at each step, the algorithm reduces the problem size by half. This results in a logarithmic number of steps, which is much faster than linear or linearithmic time.

E. Leetcode screenshots:

Code

Python3 Auto

```python
class Solution:
    def findMedianSortedArrays(self, nums1, nums2) -> float:
        if len(nums1) > len(nums2):
            nums1, nums2 = nums2, nums1
        m, n = len(nums1), len(nums2)

        imin, imax = 0, m
        while imin <= imax:
            i = (imin + imax) // 2
            j = (m + n + 1) // 2 - i

            if i < m and nums1[i] < nums2[j-1]:
                imin = i + 1
            elif i > 0 and nums1[i-1] > nums2[j]:
                imax = i - 1
            else:
                if i == 0:
                    max_of_left = nums2[j-1]
                elif j == 0:
                    max_of_left = nums1[i-1]
                else:
                    max_of_left = max(nums1[i-1], nums2[j-1])

                if (m + n) % 2 == 1:
                    return max_of_left

                if i == m:
                    min_of_right = nums2[j]
                elif j == n:
                    min_of_right = nums1[i]
                else:
                    min_of_right = min(nums1[i], nums2[j])

                return (max_of_left + min_of_right) / 2.0

```



Problem Solving Approach: The problem is to find the median of two sorted arrays, but without merging them. The median divides a set into two parts of equal-length, and one part will always be larger than the other. My algorithm finds the correct position to split the array to form the two parts. Here's how it works:

1. Identify smaller array to reduce search space for implemented binary search
2. Perform the binary search on smaller array, which returns a midpoint
3. Use this midpoint to find a type of mirror partition within the larger array
4. Check if the elements around the partitions are in the correct order
5. Using those checks, change the range of search within the smaller array accordingly
6. Calculate the median from the elements around the correct partitions that were found

Time Complexity: The binary search I implemented runs in O(logm), where m is the smaller array's length. Since m is less than or equal to n, this is also O(log(min(m,n))), but since the median is found in no more than log(min(m, n)) + 1 steps, this is also equivalent to O(log(m+n)) which satisfies the problem's requirements.

Space Complexity: The space complexity of this solution is O(1), since we're using a constant amount of dedicated extra space for the variables to store indices and the max and min of the arrays' left and right parts. No extra space is allocated for differing input sizes either.