ASSIGNMENT

PROB #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.

The following sources have been used to obtain the recurrence relation of this program:
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.

PROB #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. To do this, we use two pointers, i and j, that start from the beginning and the end of the array, respectively. We swap the elements at i and j if they are on the wrong side of the pivot, and we move the pointers until they cross each other. Then, we swap the pivot with the element at i, and we have the partitioned array. For example:

A = (3, 1, 5, 7, 6, 2, 4), pivot is 4
i = 0, j = 5
A = (3, 1, 5, 7, 6, 2, 4), no swap, move i and j
i = 1, j = 4
A = (3, 1, 5, 7, 6, 2, 4), no swap, move i and j
i = 2, j = 3
A = (3, 1, 5, 7, 6, 2, 4), swap 5 and 2, move i and j
i = 3, j = 2
A = (3, 1, 2, 7, 6, 5, 4), i and j cross, swap 4 and 7
i = 4, j = 2
A = (3, 1, 2, 4, 6, 5, 7), partitioned array

- Now, we have two subarrays, (3, 1, 2) and (6, 5, 7), that need to be sorted. We apply the same steps of choosing a pivot, partitioning the subarray, and creating smaller subarrays recursively, until we have only one or zero elements in each subarray. For example:

A = (3, 1, 2, 4, 6, 5, 7), pivot is 4
Left subarray = (3, 1, 2), pivot is 2
i = 0, j = 1
A = (3, 1, 2, 4, 6, 5, 7), swap 3 and 1, move i and j
i = 1, j = 0
A = (1, 3, 2, 4, 6, 5, 7), i and j cross, swap 2 and 3
i = 2, j = 0
A = (1, 2, 3, 4, 6, 5, 7), partitioned subarray
Left subarray = (1), one element, sorted
Right subarray = (3), one element, sorted

Right subarray = (6, 5, 7), pivot is 7
i = 4, j = 5
A = (1, 2, 3, 4, 6, 5, 7), swap 6 and 5, move i and j
i = 5, j = 4

A = (1, 2, 3, 4, 5, 6, 7), i and j cross, swap 7 and 6
i = 6, j = 4
A = (1, 2, 3, 4, 5, 6, 7), partitioned subarray
Left subarray = (5) one element, sorted
Right subarray = (7) one element, sorted

- 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, the partition process will require n−1comparisons for an array of size n. This is because each element needs to be compared with the pivot to determine its position. With the array A=[1,2,3,4,5,6,7], let us 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, we have two sub-arrays: [1,2,3,4,5,6] and [7]. The second sub-array has only one element, so no comparisons are needed there. In the first sub-array, the pivot is 6. We compare it with the other 5 elements, adding 5 comparisons.
3. We continue this process with the sub-array [1,2,3,4,5], adding 4 comparisons, and so on.

So, the total number of comparisons is 6+5+4+3+2+1=21, which matches your statement. This scenario represents the worst-case complexity of Quicksort.

c. The Quicksort algorithm achieves the minimum number of comparisons when the pivot chosen at each step is the median of the array. This results in the array being split into two equal halves at each step, leading to a balanced partition.

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

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

   [1, 2, 3] and [5, 6, 7]. This requires 6 comparisons (3 for each side).

2. In the next step, choose 2 and 6 as the pivots for the respective sub-arrays. The arrays are split into [1] , [3] , [5] , and [7] . This requires 4 comparisons (2 for each side).
3. Since all remaining arrays have only one element, no more comparisons are needed. So, the total number of comparisons is 6 + 4 = 10.

This is the minimum number of comparisons required for a 7-element array, because choosing the median as the pivot, we ensure that the array is split as evenly as possible at each step, minimizing the number of comparisons needed. Any other method of choosing the pivot would result in more unbalanced partitions and hence require more comparisons. Therefore, there cannot exist a 7-element array requiring fewer comparisons than this array.

d. The minimum number of comparisons in Quicksort occurs when each partition splits the array into two equal halves. 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 can be perfectly divided into two equal halves at each step of the recursion.

In this case, the number of comparisons at each level `i` of the recursion is `n/2^i - 1`. The total number of comparisons `C(n)` is the sum of comparisons at each level, which can be calculated as:

C(n) = summation (i = 0 to k -1) of (n/2^i−1)

= summation (i = 0 to k – 1) 1 / 2^i - k

This sum is a geometric series with k terms, ratio 1 / 2 , and first term 1. The sum of a geometric series is (1 - r^n) / (1 – r). So, the sum of the series is 1 – 1 / 2^k. Substituting k = log2(n+1), we get:

C(n) = n(1 − 1/2log2(n+1)) − log2(n+1)

Simplifying, we get:

C(n)=n−1−log2(n+1).

Since this is the base case scenario, we can say that, the formula we have derived can be used to explain the time complexity of the best case scenario. 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.

Now, moving onto the worst case scenario. The maximum number of comparisons in Quicksort occurs when each partition splits the array into two parts such that one part contains n-1 elements and the other part contains 0 elements. This is the worst-case scenario for Quicksort. 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 number of comparisons at each level i of the recursion is n-i. The total number of comparisons C(n) is the sum of comparisons at each level, which can be calculated as:

C(n)= summation (i = 0 to n – 1) n^2/2 − n/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 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).

PROB #3

Date

Q. to prove: $T(n) = \Theta(n^{\log_y(x)})$, when $z < \log_y(x)$.

Given that,

$T(n) = xT(n/y) + n^z$, $z < \log_y(x)$

$T(n)$ can be rewritten as,

$T(n) = xT(n/y) + O(n^z)$

$\because z < \log_y x \therefore n^z = O(n^{\log_y x})$

or $T(n) = xT(n/y) + O(n^{\log_y x})$

A/c to master theorem, if $T(n) = aT(n/b) + O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b(a)})$.

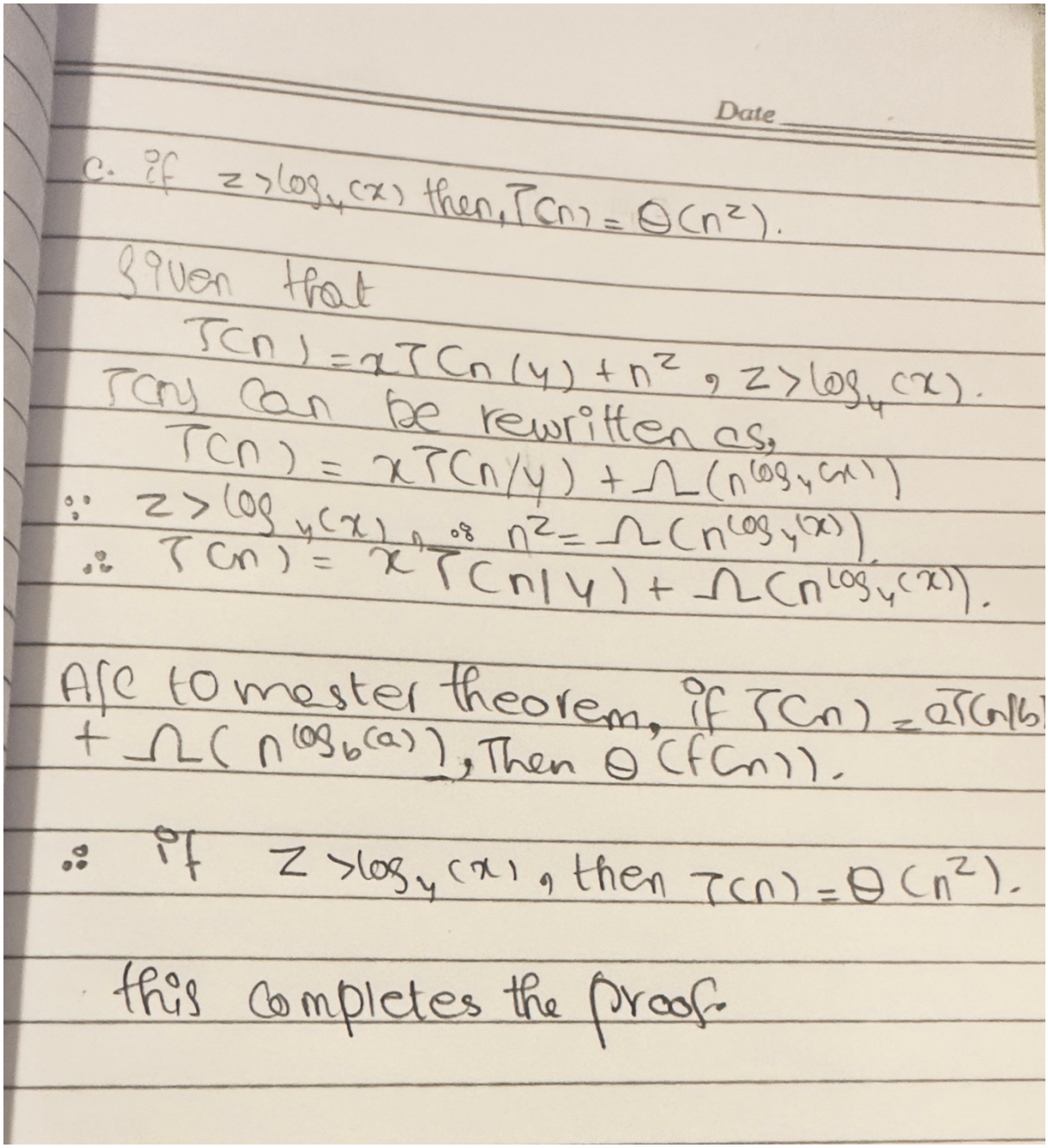
So, if $z < \log_y x$ then, $T(n) = \Theta(n^{\log_y x})$ thus completing the proof.

b. If $z = \log_y(x)$, then prove
$T(n) = \Theta(n^{\log_y(x)} * \log_y n)$.

A/c to master's theorem, if $f(n) = \Theta(n^{\log_b(a)} * \log^k(n))$ for some $k \geq 0$. Then, $T(n) = \Theta(n^{\log_b(a)} * \log^{k+1}(n))$. In this case, $k = 0$, so if $z = \log_y x$ Then, $T(n) = \Theta(n^{\log_y x} * \log(n))$.

Given that,
$T(n) = xT(n/y) + n^z$, $z = \log_y(x)$
$T(n)$ can be rewritten as,
$T(n) = xT(n/y) + \Theta(n^{\log_y(x)})$
Since, $z = \log_y(x)$, we got $n^z = n^{\log_y x} = \Theta(n^{\log_y(x)})$. $\therefore T(n) = xT(n/y) + \Theta(n^{\log_y(x)})$
but a/c to master theorem:
If $T(n) = aT(n/b) + \Theta(n^{\log_b(a)} * \log^k(n))$
then $T(n) = \Theta(n^{\log_b a} * \log^{k+1}(n))$
$\therefore$ If $z = \log_y x$ then, $T(n) = \Theta(n^{\log_y(x)} * \log(n))$.

This completes the proof.



d. Given recurrence relation for the Strassen's algorithm is: T(n) = 7T(n/2) + n^2.

We can apply the Master Theorem to solve this recurrence relation. In this case, a = 7, b = 2, and f(n) = n^2. Therefore, log (a) with base b = log(7) with base 2 and z = 2. Comparing z with log (a) with base b, we see that z < log(a) with base b.

Therefore, we can use the result from part (a) of the Master Theorem proof, which states that if z < log (a) with base b, then T(n) = theta(n^log(a) with base b).

So, the running time of Strassen's algorithm is T(n) = theta(n^log_2(7)). Now, let's compare this with the running time of the standard matrix multiplication algorithm, which is theta(n^3). Since log(7) with base 2 is approximately 2.81, which is less than 3, we can

conclude that Strassen's algorithm is faster than the standard matrix multiplication algorithm for large n.

PROB #4

a.

The described algorithm is a linear algorithm. This is because merging two sorted arrays can be done in O(n) time, where n is the total number of elements in the two arrays. Here's why: When merging two sorted arrays, you start at the beginning of each array and compare the first elements. You take the smaller of the two and add it to the new array, and move the pointer of that array to the next element. You repeat this process until you've gone through all the elements in both arrays. Since you're only going through each array once, the time complexity is O(n), where n is the total number of elements in the two arrays. Finding the median from the merged and sorted array is a constant time operation, O(1), because you're simply accessing the elements at two specific indices (n and n+1) and taking their average. Therefore, the overall time complexity of the algorithm is dominated by the merging process, which is O(n). So, the algorithm is linear, not 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++];
  }
}

double findMedian(int C[], int n) {
  if (n % 2 == 0) {
    return (C[n/2 - 1] + C[n/2]) / 2.0;
  } else {
    return C[n/2];
  }
}
```

b. In a sorted array, all elements before a certain index are less than or equal to the elements after that index. In this case, since A[5] < B[5], we know that all elements in A up to index 5 are less than or equal to B[5]. This is because A and B are sorted arrays. When we combine the sub-arrays A[5:9] and B[1:5], we are essentially taking the elements from A and B that are greater than or equal to A[5] and less than or equal to B[5]. Since these elements are in the middle of the combined sorted array, the median of the combined array will be among these elements. Therefore, the median of the combined sub-arrays A[5:9] and B[1:5] is the correct answer. This is confirmed by the fact that the median of the combined sub-arrays in the example is indeed 55, matching the correct answer.

c. If A[(n + 1)/2] > B[(n + 1)/2], it means that the median of array A is greater than the median of array B. In this case, the elements in the second half of array B and the first half of

array A will form the middle part of the combined array. This is because all elements in the second half of array B are less than or equal to A[(n + 1)/2] and all elements in the first half of array A are greater than or equal to B[(n + 1)/2]. Therefore, the median of the combined sub-arrays A[1 : (n + 1)/2] and B[(n + 1)/2 : n] will be the correct answer.
On the other hand, if A[(n + 1)/2] < B[(n + 1)/2], it means that the median of array A is less than the median of array B. In this case, the elements in the first half of array B and the second half of array A will form the middle part of the combined array. This is because all elements in the first half of array B are greater than or equal to A[(n + 1)/2] and all elements in the second half of array A are less than or equal to B[(n + 1)/2]. Therefore, the median of the combined sub-arrays A[(n + 1)/2 : n] and B[1 : (n + 1)/2] will be the correct answer.

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 A[mid] and B[mid].
3. If A[mid] > B[mid], then the median must lie in the first half of A or the second half of B. 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 or the first half of B. So, recursively find the median in A[mid : n] and B[1 : mid].
5. The base case for the recursion is when n = 1. In this case, return the smaller of A[1] and B[1].
6. The median of the combined array is the median of the sub-arrays determined by the above steps.
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
Problem Solving Approach: The problem is to find the median of two sorted arrays. The approach used here is to merge the two sorted arrays into a third array, and then find the median of this merged array.

1. First, we initialize three pointers i, j, and k to 0. These pointers represent the current index in nums1, nums2, and nums3 respectively.
2. We calculate the total size n of the merged array, which is the sum of the sizes of nums1 and nums2.
3. We allocate memory for nums3 to hold n integers.
4. We then enter a loop where we compare the current elements of nums1 and nums2. The smaller element is added to nums3 and the corresponding pointer is incremented. This is done until we reach the end of either nums1 or nums2.
5. If there are remaining elements in nums1 or nums2, we add them to nums3.
6. Finally, we calculate the median. If n is even, the median is the average of the two middle elements in nums3. If n is odd, the median is the middle element in nums3.
7. We free the memory allocated for nums3 and return the median.

Time Complexity: The time complexity of this function is O(n), where n is the total number of elements in nums1 and nums2. This is because we have to iterate through all the elements in both arrays to merge them into nums3.

Space Complexity: The space complexity of this function is also O(n), where n is the total number of elements in nums1 and nums2. This is because we are creating a new array nums3 to hold all the elements from nums1 and nums2. The space used by the other variables in the function is constant, so it does not affect the overall space complexity.

My code for copy/test purpose:

```
double findMedianSortedArrays(int* nums1, int nums1Size, int* nums2, int
nums2Size) {
    int i = 0, j = 0, k = 0;
    int n = nums1Size + nums2Size;
    int* nums3 = (int*)malloc(n * sizeof(int));

    while (i < nums1Size && j < nums2Size) {
        if (nums1[i] < nums2[j]) {
            nums3[k++] = nums1[i++];
        } else {
            nums3[k++] = nums2[j++];
        }
    }
    while (i < nums1Size) {
        nums3[k++] = nums1[i++];
    }
    while (j < nums2Size) {
        nums3[k++] = nums2[j++];
    }

    double median;
    if (n % 2 == 0) {
        median = (nums3[n/2 - 1] + nums3[n/2]) / 2.0;
    } else {
        median = nums3[n/2];
    }

    free(nums3);
    return median;
}
```

Screenshot of code:

Code

C Auto

```c
double findMedianSortedArrays(int* nums1, int nums1Size, int* nums2, int nums2Size) {
    int i = 0, j = 0, k = 0;
    int n = nums1Size + nums2Size;
    int* nums3 = (int*)malloc(n * sizeof(int));

    while (i < nums1Size && j < nums2Size) {
        if (nums1[i] < nums2[j]) {
            nums3[k++] = nums1[i++];
        } else {
            nums3[k++] = nums2[j++];
        }
    }
    while (i < nums1Size) {
        nums3[k++] = nums1[i++];
    }
    while (j < nums2Size) {
        nums3[k++] = nums2[j++];
    }

    double median;
    if (n % 2 == 0) {
        median = (nums3[n/2 - 1] + nums3[n/2]) / 2.0;
    } else {
        median = nums3[n/2];
    }

    free(nums3);
    return median;
}
```

Testcase | Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

nums1 =

[1,2]

nums2 =

[3,4]

Output

2.50000

Expected

2.50000