1.

# algorithm in python

# assuming the size of array A is greater than 0

def local\_min(A: [int]) -> int:

# base case: if the size of array is 1, return the only element

# if the size is 2, return the smaller element

if len(A) == 1:

return A[0]

if len(A) == 2:

return min(A)

# choose the middle element in the array as the pivot

pivot = len(A) // 2

# if the pivot is already a local minimum, return it

if A[pivot-1] >= A[pivot] <= A[pivot+1]:

return A[pivot]

# else, do recursive call on the partition of array with the smaller

# element, not including the pivot. Since that partition is guaranteed

# to have a local minimum

if A[pivot-1] < A[pivot+1]:

# since end index is not included in list slicing

return local\_min(A[:pivot])

else:

return local\_min(A[pivot+1:])

In the algorithm above, every step other than the recursive call takes constant time; and in the recursive call, the size of input is approximately half of original input size.

Therefore, the recurrence of this algorithm is , and according to the Master Theorem, is . Additionally, we know that the pivot may already be a local minimum and we may terminate the search early, so the overall asymptotic runtime is .

We know the base cases are correct; after checking the base cases, the array will have at least 3 elements so the comparisons with the pivot must be possible. In the partition step, using the partition with the smaller element guarantees that a local minimum will exist in that partition, so we do not need to check the other partition.

2.

# algorithm in python

def find\_median(vector1, vector2, n):

# base case: if n is small (we chose 2 here), simply use a brute force algorithm

if n <= 2:

union = sorted(vector1 + vector2)

# size of union vector will be either 2 or 4, depending on the value of n

size = len(union)

# the median of the two (small) vectors is the average of

# the middle two elements

return (union[size/2] + union[size/2-1]) / 2

# find the median of each vector. Since we know that the two vectors are sorted,

# the median is simply the middle element

median1 = len(vector1) // 2

median2 = len(vector2) // 2

if n % 2 == 0:

# if n is even, need to shift the index in one vector by one so that,

# when we divide the vectors, their size, n, is still equal

median2 -= 1

if vector1[median1] < vector2[median2]:

# in this case, the median will be in the following paritions:

# vector1: from index of median1 to the end of vector

# vector2: from beginning to index of median2

# divide the vector into these partitions and do recursion

partition1 = vector1[median1:]

partition2 = vector2[:median2+1]

else:

# in this case, we parition the vectors similarly, but in the opposite

# directions

partition1 = vector1[:median1+1]

partition2 = vector2[median2:]

# recursion step

return find\_median(partition1, partition2, len(partition1))

Since we are only querying the middle element of the vector in each recursion step; and deciding which direction we need to partition the vector. We are effectively doing binary search in both vectors. Therefore the total number of queries are O(log n).