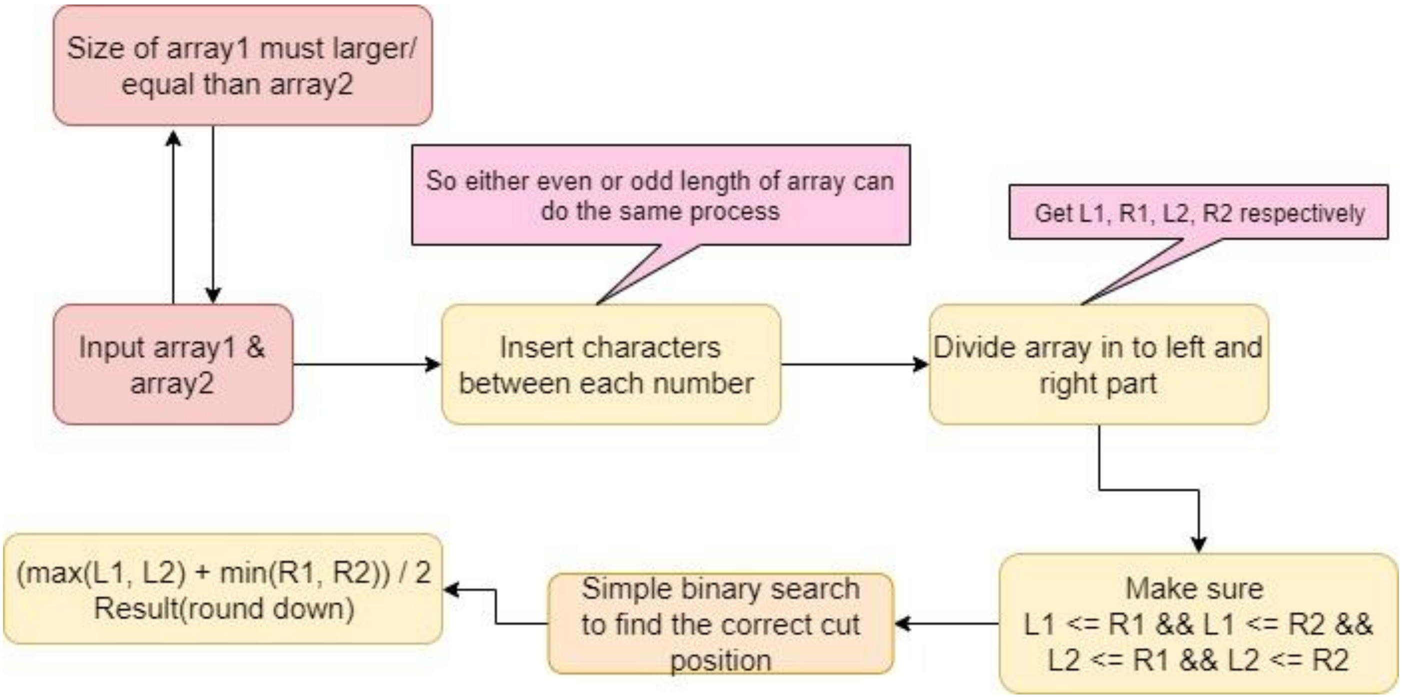
Program assignment 2

Implementation:

Although the program requirement is the same length for two arrays, I wrote a more general solution, which can cap with different length of arrays.

Here is my **procedure flowchart**:

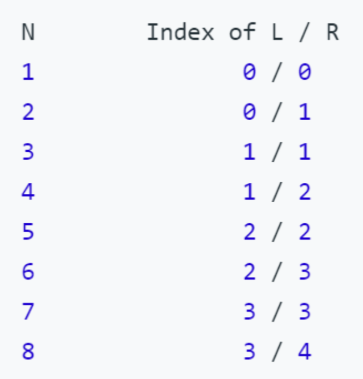

Size of array1 must larger/ equal than array2
So either even or odd length of array can do the same process
Get L1, R1, L2, R2 respectively
Input array1 & array2
Insert characters between each number
Divide array in to left and right part
(max(L1, L2) + min(R1, R2)) / 2 Result(round down)
Simple binary search to find the correct cut position
Make sure L1 <= R1 && L1 <= R2 && L2 <= R1 && L2 <= R2

Pseudocode:

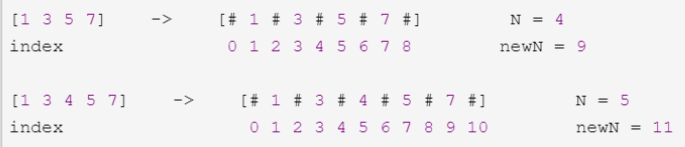
```
FindMedian(array1 , array2)
    N1 = size of array1, N2 =size of array2
    if (N1 < N2) return FindMedian(array2, array1);   // Make sure array2 is the shorter one.
    int left = 0, right = N2 * 2
    while (left <= right)
        int mid2 = (left + right) / 2;      // Try Cut 2
        int mid1 = N1 + N2 - mid2;    // Calculate Cut 1 accordingly

        double L1 = (mid1 == 0) ? INT_MIN : array1 [(mid1-1)/2] // Get L1, R1, L2, R2 respectively
        double L2 = (mid2 == 0) ? INT_MIN : array2 [(mid2-1)/2]
        double R1 = (mid1 == N1 * 2) ? INT_MAX : array1 [(mid1)/2]
        double R2 = (mid2 == N2 * 2) ? INT_MAX : array2 [(mid2)/2]

        if (L1 > R2) lo = mid2 + 1;           // A1's lower half is too big; need to move C1 left (C2 right)
        else if (L2 > R1) hi = mid2 - 1;      // A2's lower half too big; need to move C2 left
        else return (max(L1,L2) + min(R1, R2)) / 2;  // Otherwise, that's the right cut
```

Approach explanation:

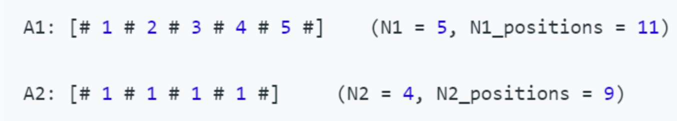
It’s quite a difficult thought, I spent a lot of time to understand it. Since a naïve way to implement it is to consider odd-length and even -length arrays as two cases, and treat them separately. But actually, it can be treated as the same case if we cut the sorted array to two halves of *equal lengths*, then median is the average of max in the left half and min in the right half. For instance,
[3 5 / 7 8] for even length, we have L=5 and R=7, respectively.
[2 4 6 / 6 8 9] for odd length, we have L=6 and R=6(we split 6 into two halves, both L and R parts contain 6)
We can observe the index of L and R with different length as shown on right:
So the index of L = (N-1)/2 , R = N/2 ,
the median can be obtained by (L+R)/2 =(A[(N-1)/2] + A[N/2]   )/2
To simplify the conditions of different lengths of arrays, I used a trick, which is to add a character between each number.



```
N          Index of L / R
1               0 / 0
2               0 / 1
3               1 / 1
4               1 / 2
5               2 / 2
6               2 / 3
7               3 / 3
8               3 / 4
```



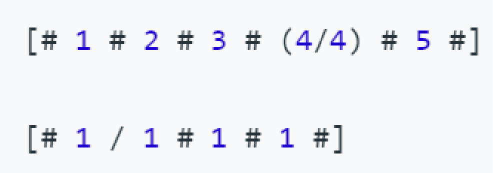
```
[1 3 5 7]    ->    [# 1 # 3 # 5 # 7 #]        N = 4
index               0 1 2 3 4 5 6 7 8        newN = 9

[1 3 4 5 7]    ->    [# 1 # 3 # 4 # 5 # 7 #]        N = 5
index                  0 1 2 3 4 5 6 7 8 9 10        newN = 11
```



```
A1: [# 1 # 2 # 3 # 4 # 5 #]    (N1 = 5, N1_positions = 11)

A2: [# 1 # 1 # 1 # 1 #]     (N2 = 4, N2_positions = 9)
```

By doing so, no matter if the length of the array is odd or even, it will become even length afterward(2*N+1), and that the split point is fixed. index(L) = (CutPosition-1)/2, index(R) = (CutPosition)/2.
Here are some observations:

1. There’re 2N1 + 2N2 + 2 positions in total. Therefore, there must be exactly N1 + N2 positions on each side of the cut and 2 positions directly on the cut.
2. When cutting at position C2 = k in A2, then the cut position in A1 must be C1 = N1 + N2 – k.
E.x. If C2 = 2, we must have C1 = 4+5-2=7
3. When the cuts are done, we will have:
L1 = A1[(C1-1)/2]; R1 = A1[C1/2];
L2 = A2[(C2-1)/2]; R2 = A2[C2/2];



```
[# 1 # 2 # 3 # (4/4) # 5 #]

[# 1 / 1 # 1 # 1 #]
```

To make sure L1 <= R1 && **L1 <= R2 && L2 <= R1** && L2 <= R2, we use simple binary search to find out the result.

If L1 > R2, it means that too many large numbers on the left side of A1, we have to move C1 to the left.
If L2 > R1, it means too many large numbers on the left side of A2, move C2 to the left.
Otherwise, the cut point is correct.
After finding the right cut position, the median can be calculated as (max(L1, L2) + min(R1, R2)) /2

Corner cases:

When the cut falls on 0th(first) or 2N th(last) position, which exceeds the boundary of arrays. So I imagine that both A1 and A2 have two extra elements, INT_MIN at A[-1] and INT_MAX at A[N]. Thus, if any L falls out of the left boundary of the array, the L = INT_MIN, vice versa.

Time complexity:

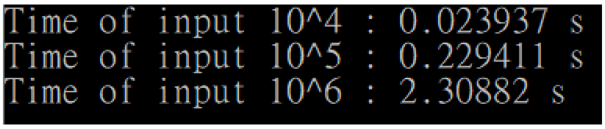
Since C1, C2 are mutually determined, we can just move one of them first, then calculate the other accordingly. However, it's much quicker to move C2 (the one on the shorter array) first. Also, moving only on the shorter array gives a runtime complexity of O(lg(min(N1, N2))) (Binary search time complexity). And since the length of two arrays are the same in this problem(N1=N2), so time complexity is O(lgn).

Screenshot result:

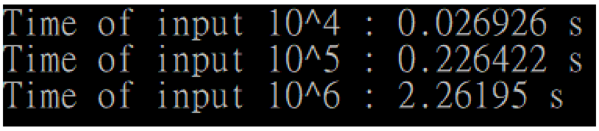
O(nlgn) version: <*Codeblocks windows environment*>

Pseudocode:

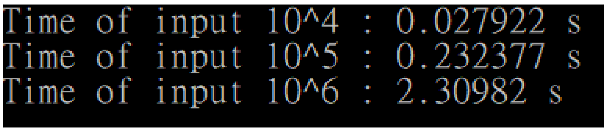
```
FindMedian(array1 , array2)
    append array2 element to array1    // O(n) for n elements
    sort(array1.begin(),array1.end())    // use merge sort O(nlogn)
    if(array1.size()%2==1)
        result= array1 [((array1.size())-1)/2]
    else
        result=( array1 [(array1.size())/2]+ array1 [(array1.size())/2-1])/2.0
```



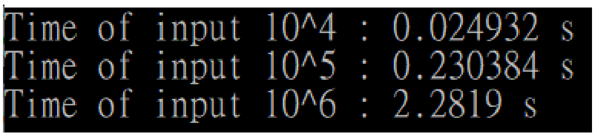
```
Time of input 10^4 : 0.023937 s
Time of input 10^5 : 0.229411 s
Time of input 10^6 : 2.30882 s
```



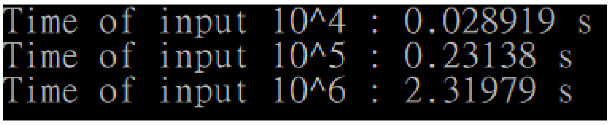
```
Time of input 10^4 : 0.026926 s
Time of input 10^5 : 0.226422 s
Time of input 10^6 : 2.26195 s
```



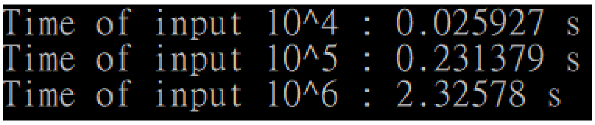
```
Time of input 10^4 : 0.027922 s
Time of input 10^5 : 0.232377 s
Time of input 10^6 : 2.30982 s
```



```
Time of input 10^4 : 0.024932 s
Time of input 10^5 : 0.230384 s
Time of input 10^6 : 2.2819 s
```



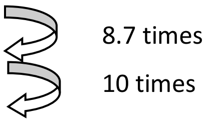
```
Time of input 10^4 : 0.028919 s
Time of input 10^5 : 0.23138 s
Time of input 10^6 : 2.31979 s
```



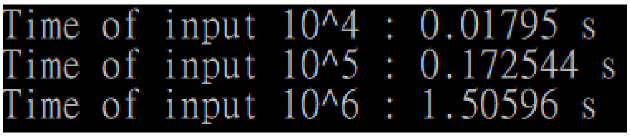
```
Time of input 10^4 : 0.025927 s
Time of input 10^5 : 0.231379 s
Time of input 10^6 : 2.32578 s
```

I ran for 6 times to calculate the average of results to get better accuracy.

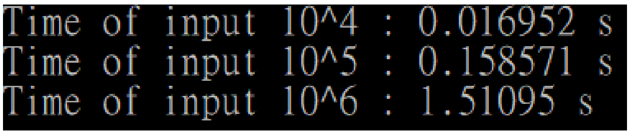
Time of input 10^4 = 0.02652717
Time of input 10^5 = 0.2302255
Time of input 10^6 = 2.30134333



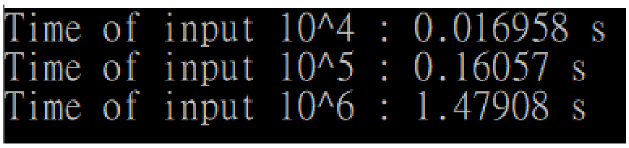
O(lgn) version: <*Codeblocks windows environment*>



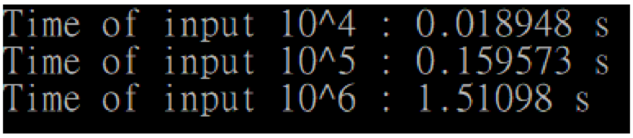
```
Time of input 10^4 : 0.01795 s
Time of input 10^5 : 0.172544 s
Time of input 10^6 : 1.50596 s
```



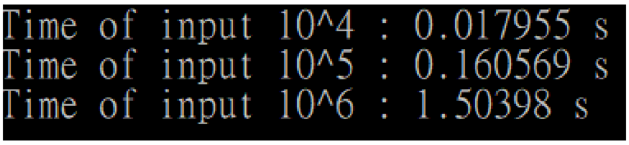
```
Time of input 10^4 : 0.016952 s
Time of input 10^5 : 0.158571 s
Time of input 10^6 : 1.51095 s
```



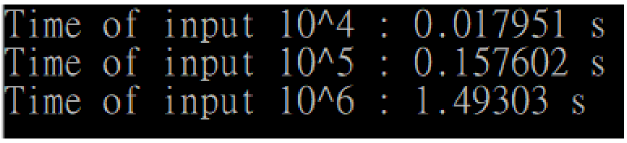
```
Time of input 10^4 : 0.016958 s
Time of input 10^5 : 0.16057 s
Time of input 10^6 : 1.47908 s
```



```
Time of input 10^4 : 0.018948 s
Time of input 10^5 : 0.159573 s
Time of input 10^6 : 1.51098 s
```



```
Time of input 10^4 : 0.017955 s
Time of input 10^5 : 0.160569 s
Time of input 10^6 : 1.50398 s
```



```
Time of input 10^4 : 0.017951 s
Time of input 10^5 : 0.157602 s
Time of input 10^6 : 1.49303 s
```

Time of input 10^4 = 0.01778567
9.08 times
Time of input 10^5 = 0.1615715
9.29 times
Time of input 10^6 = 1.50066333

Analysis:

Since I use Codeblocks to test my code, so it's normal that the run time is much longer comparing to Linux environment. I've already eliminated the error by compling code for 6 times and get the average of the runtime.

However, the runtime of two versions of code indeed have obvious differences, either in different input size or different code. Here's my calculation:
For O(nlgn) version:
When input size increasing from 10^4 to 10^5 time increases 8.7 times in runtime, while input size increasing from 10^5 to 10^6 time increases 10 times. So the increasing rate is 10 / 8.7 = 1.15.
For O(lgn) version:
When input size increasing from 10^4 to 10^5 time increase 9.08 times in runtime, while input size increasing from 10^5 to 10^6 time increases 9.29 times. So the increasing rate is 9.29 / 9.08 = 1.023.
I expected that two version of increasing rate will be 10 times in difference (since nlgn / lgn = n, n=10),
So firstly, I calculated the difference of increasing rate, which is 0.15 / 0.023 =6.52 times actually.

O(nlgn)	O(lgn)
0.02652717 1.5 times	0.01778567
0.2302255 1.43 times	0.1615715
2.30134333 1.533 times	1.50066333

Then, I multiply 6.52 by 1.5 times equals 9.78, nearly 10 times.
Therefore, my assumption is correct, as two versions of code have 10 times runtime differences.