CSC301 HW4

Alex Zhang

Feb 2023

Question 1

Based on the question, since n is a power of 2, n will always be even. The following is the pesudocode for the algorithm,

```
function v = MEDIAN(A,B,n)
    if n == 1 then return max(A[0],B[0])
    m1 = (A[n/2-1]+A[n/2]) / 2
    m2 = (B[n/2-1]+B[n/2]) / 2
    if m1 == m2
        then return min(A[n/2],B[n/2])
    if m1 < m2
        return MEDIAN(A[n/2:], B[:n/2], n/2)
    else
        return MEDIAN(A[:n/2], B[n/2:], n/2)
end function
```

Proof of Correctness

The algorithm assumes that the return value of median will be the number on the right side of two medians. In the base case where n = 1, the length of array A and B is 1, which the median of these two arrays are $A[0]$ and $B[0]$. Since the algorithm assumes to get the larger median, it will return max number between these two. The base case works.
For inductive cases, we assume that the algorithm will work when the length is $n/2$ since n is a power of 2, and we need to show that n length will also work.

Case 1: $m1 = m2$
In this case, in array A, there are total $n/2$ numbers that is smaller than $m1$ and there is $n/2$ number bigger than $m1$. This is also same for array B and number $m2$. Therefore, in the union of two arrays, there are total n number smaller and n number larger than $m1$, indicating $m1$ or $m2$ is also the median of the union of two arrays. Since

$$m1 = \frac{(A[n/2-1] + A[n/2])}{2}$$
$$m2 = \frac{(B[n/2-1] + B[n/2])}{2}$$

And they are euqal to each other. We can get that larger number between $A[n/2-1]$ and $B[n/2-1]$ lands at position $n/2-1$ and the smaller number between $A[n/2]$ and $B[n/2]$ lands as position $n/2$. Based on the algorithm, it will return the larger median which is the the smaller number between $A[n/2]$ and $B[n/2]$.

Case 2: $m1 < m2$
In this case, $m1$ and $m2$ will stil follow the case that in each array, there is going to be $n/2$ smaller and larger than $m1$ and $m2$. However, since $m1 < m2$, this means that all number smaller than $m1$ in array A will also be smaller than $m2$. Therefore, there will be more than n number smaller than $m2$ so the true median m_t has the property that

$$m1 \leq m_t \leq m2$$

We can then eliminate the left side of $m1$ in array because they are all too small, and also the right side of $m2$ in array B since they are all too big. The algorithm does the recursion and then following with computing median based on the new array.

Case 3: $m1 > m2$
This case is very similar to the case above. Again, in array A, there are $n/2$ number smaller than $m1$ and also $n/2$ number bigger than $m1$. This also holds true for $m2$. Since $m1 > m2$ this means that the true median will locate between $m1$ and $m2$ Because for the right side of $m1$ in array A, the number are too big becuase there are at least n number before it. We can also eliminate the left side of B since they are too small. The largest number in left side is smaller than $m2$ which means only bigger than $n/2 - 2$ numbers. For the algorithm, it chops off the sides where true median will not exist and does the recursion on new arrays.
For second and third cases, each recursion, n is divided by two until $n = 1$. Since we assume that $n/2$ is true, the recursion will work. ■

Costs Analysis

For each recursion, the time complexity will be $O(1)$. The addition and division are assumed to be constant time since the number in arrays are not n-bits number. The comparison and splicing will also only need constant time and n is a power of 2, $n/2$ costs $O(1)$. Since we assume that the length of two arrays are n-bits numbers, and we are doing the recursion by dividing by two each time until $n = 1$. $T(1) = O(1)$. We can then thrive the equation,

$$T(n) = T(n/2) + O(1)$$

In base case, the comparison also only require constant time, so $T(1) = O(1)$. Assume we have to go through r times of recursive calls until the base case, r can be represented as,

$$\begin{aligned} \frac{n}{2^r} &= 1 \\ n &= 2^r \\ n &= 2^{\log_2 n} \\ r &= \log n \end{aligned}$$

There are $\log n$ recursions and each recursions cost constant time. We conclude that the running time for this algorithm will be $O(\log n)$.

Question 2

Question 3

From the lecture, the time complexity for Strassen's algorithm is $O(n^{\log_2 7}) = O(n^{2.81})$. The classical matrix multiplication of partition into 3×3 grid blocks needs 27 recursive calls. We assume that in each recursion, the cost is $O(n^2)$. We can draw the equation

$$T(n) = 27 \cdot T(n/3) + O(n^2)$$

Through master theorem, we know that the time complexity will be $O(n^{\log_3 27})$. If we want to create a algorithm which is better than Strassen, we have to make sure

$$\begin{aligned} O(n^{\log_3 x}) &< O(n^{2.81}) \\ \log_3 x &< 2.81 \\ x &< 3^{2.81} \\ x &< 21.91 \end{aligned}$$

Where x is the number of recursive calls. After rounding up, we have to find an algorithm for matrix multiplication with 3×3 parition that only has 21 or less recurvie calls in each level. So the largest recursive calls is 21.