CSE 431/531 HW3 Spring 2022 4/4/2022

Feng Wei 50428109

Problem 1.

(a) $T(n) = 4T(n/3) + O(n)$ $\quad T(n) = O(n^{\log_3 4})$

(b) $T(n) = 3T(n/3) + O(n^2)$ $\quad T(n) = O(n^2)$

(c) $T(n) = 4T(n/2) + O(n^2\sqrt{n})$ $\quad T(n) = O(n^2\sqrt{n})$

(d) $T(n) = 8T(n/2) + O(n^3)$ $\quad T(n) = O(n^3 \log n)$

$$T(n) = \begin{cases} O(n^{\log_b a}) & c < \log_b a \\ O(n^c \lg n) & c = \log_b a \\ O(n^c) & c > \log_b a \end{cases}$$

Problem 2.

I modify the count-Inversion between B and C from Slides 14/3.

In the orignal algorithm we count the inversion ① and merge B and C ② and the same time.

For this problem, we wanna to count the strong inversion. (A[i] > 2A[j])

So, I will do count ① and merge ② seperatly.

```
sort-and-count(A, n)
1:  if n = 1 then
2:      return (A, 0)
3:  else
4:      (B, m1) ← sort-and-count(A[1 ... ⌊n/2⌋], ⌊n/2⌋)
5:      (C, m2) ← sort-and-count(A[⌊n/2⌋+1 ... n], ⌈n/2⌉)
6:      (A, m3) ← merge-and-count(B, C, ⌊n/2⌋, ⌈n/2⌉)
7:      Return (A, m1 + m2 + m3)
```

merge-and-count (B, C, m_1, m_2)

1: count ← 0 ;

2: A ← Array of size m_1+m_2 ; i ← 1 ; j ← 1 ;

3: while $i \le m_1$ or $j \le m_2$ do → time complexity O(n)

4: if $j > m_2$ or ($i \le m_1$ and $B[i] \le 2 \times C[j]$) then

count strong inversion

5: i ← i+1

6: count ← count + (j-1)

7: else

8: j ← j+1

9: i ← 1 ; j ← 1 ;

10: while $i \le m_1$ or $j \le m_2$ do → time complexity O(n)

if $j > m_2$ or ($i \le m_1$ and $B[i] < C[j]$) then → sorted

11: A[i+j-1] ← B[i] ; i ← i+1

12:

13: else

A[i+j-1] ← C[j] ; j ← j+1

14: return (A, count)

Recurrence for running time:

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

running time $= O(n \lg n)$

Correctness is obvious, since I slightly modified the Algorithm from ppt.

problem 3.

There may be multiple local minimum in A.

We are only required to find A local minimum.

Since $A[0] = A[n+1] = \infty$, so for corner cases A[1] and A[n],

we only need to compare one neighbor.

For $n=1$, the local minimum is simplely A[1].

In the below Algorithm, we assume $n > 1$.

Also, since A is an array of distinct numbers. So $A[i] \neq A[j]$ for $i \neq j$

1: localminmum (A, n)

2: $i \leftarrow \lfloor n/2 \rfloor$;

3: $B \leftarrow A[1,2 \cdots i]$; $C \leftarrow A[i+1, \cdots n]$;

4: if ($A[i] < A[i-1]$ and $A[i] < A[i+1]$) then

5: return $A[i]$

6: else if ($A[i] > A[i-1]$) then

7: return localminmum (B, i)

8: else

9: return localminmum (C, i)

Correctness :

if $n=1$ local minmum : $A[1]$ is obvious.

For $n>1$. $i = \lfloor n/2 \rfloor$

① if $A[i] < A[i-1]$ and $A[i] < A[i+1]$

$A[i]$ is a local minmum.

② if $A[i] > A[i-1]$, then we can find a local minmum in the left sub Array $B \leftarrow A[1, \cdots i]$

proof: For $B = A[1, \cdots i]$ and $A[i] > A[i-1] \rightarrow B[i] > B[i-1]$

if there is no local minmum, then we have

$B[i] > B[i-1] \rightarrow B[i-1] > B[i-2]$

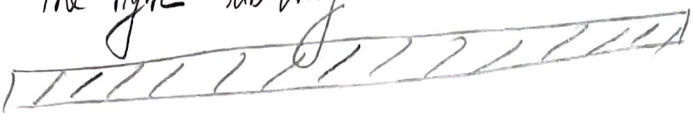
$\rightarrow B[i-2] > B[i-3]$ ..... that means B is an sorted Array with increasing order.

But we have: $B[1] < B[2]$ and $B[1] < B[0] = \infty$ So $B[1]$ is a local m.

that means We can find a Local minmum in $B \leftarrow A[1, \cdots i]$, if $A[i] > A[i-1]$

② if $A[i] > A[i+1]$ then we can find a local minimum in

the right sub array $C \leftarrow A[i+1, \cdots n]$



Prove:

For $C = A[i+1, \cdots n]$ and $A[i] > A[i+1] \rightarrow C[i] > C[i+1]$

corner

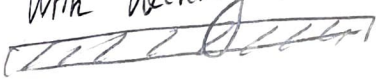
if there is no local minimum in C, then we have

$C[i] > C[i+1] \rightarrow C[i+1] > C[i+2]$

$\vdots$

$\rightarrow C[n-1] > C[n]$ that means C is an sorted array

with decreasing order.



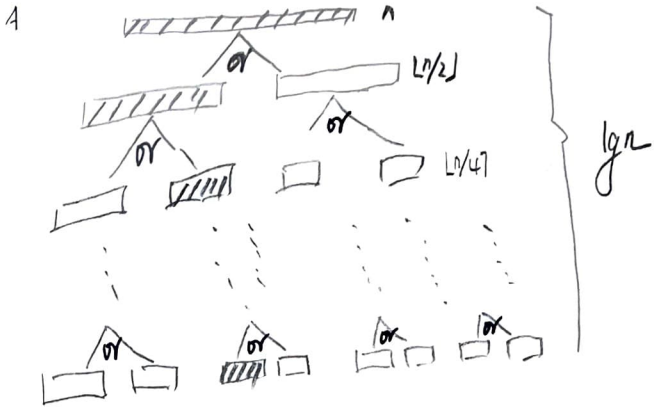
But we have $C[n] < C[n+1] = \infty$ and $C[n] < C[n-1]$

so $C[n]$ is a local minimum

That means we can find a local minimum in $C \leftarrow A[i+1, \cdots n]$

if $A[i] > A[i+1]$.

time complexity.



For each step we divide the array into two sub array.

But we only choose one sub array to continue

For each level, we need O(1) running time for comparision

There are O(lgn) levels for the worse case

Total running time O(lgn)

Problem 4: $2^n \times 2^n$ with 1×1 missing

1: Cover-with-L-shape ($2^n \times 2^n$)

2: if $n = 1$ then

3: It is a "L" shape. Done.

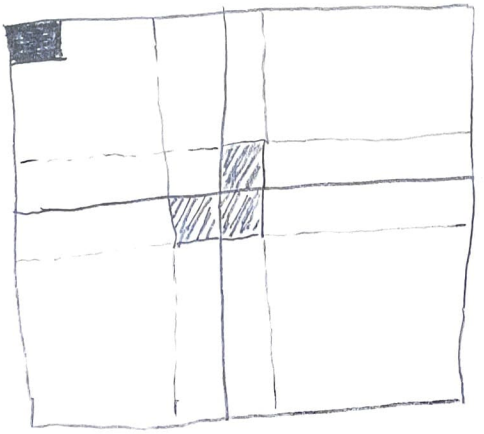
4: else

5: Cover the center of $2^n \times 2^n$ with a "L" shape. And done

cover the subsquare with the missing 1×1

6: So, we have 4 $2^{n-1} \times 2^{n-1}$ each with a missing 1×1

7: Do cover-with-L-shape ($2^{n-1} \times 2^{n-1}$) 4 times



time complexity:

$k = 2^n$ $\quad$ $2^{n-1} = k/2$

$T(k) = 4(k/2) + O(1)$

$T(k) = O(k^2)$

As a result, the time complexity is $O(2^{2n}) = O(4^n)$

Alternatively, the whole area is $2^n \times 2^n - 1 = 2^{2n} - 1$, "L" is 3

To cover the whole area we need $(2^{2n}-1)/3$ numbers of "L" shape.

For each L shape we need $O(1)$ time

So, the total time is $(2^{2n}-1)/3$

$\rightarrow \quad \Theta(2^{2n}) = O(4^n)$