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

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

Problem 2.

I modify the Count-Inversion between B and C from slides 14/13.

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

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

So, I will do count and merge seperatly.

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

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

```
count ← 0;
A ← Array of size m1+m2;   i ← 1;  j ← 1;
while i ≤ m1 or j ≤ m2 do                          → time complexity O(n)
    if j > m2 or (i ≤ m1 and B[i] ≤ 2×C[j]) then
                                                    Count strong inversion
        i ← i+1
        count ← count + (j-1)
    else
        j ← j+1

i ← 1; j ← 1;
while i ≤ m1 or j ≤ m2 do                          → time complexity O(n)
    if j>m2 or (i ≤ m1 and B[i] < C[j]) then      → sort
        A[i+j-1] ← B[i];  i ← i+1
    else
        A[i+j-1] ← C[j];  j ← j+1
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 ppts.

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 simply $A[1]$.

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

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

```
localminmum ( A, n )
    i ← ⌊n/2⌋ ;
    B ← A[1,2 ⋯ i] ;    C ← A[i+1, ⋯ n] ;
    if ( A[i] < A[i-1]  and  A[i] < A[i+1] ) then
            return A[i]
        else if ( A[i] > A[i-1] )  then
            return localminmum ( B, i )
        else
            return localminmum ( C, i )
```

should be (n -i)

Correctness :

if $n=1$ Local mimum : $A[1]$ is obvious.

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

⓪ if $A[i] < A[i-1]$ and $A[i] < A[i+1]$

$A[i]$ is a local minmum.

① if $A[i] > A[i-1]$, then we can find a local minnum

in the left sub Array $B \leftarrow A[1,\cdots i]$

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

if there is no local minnum, 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 min.

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

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

$$\vdots$$

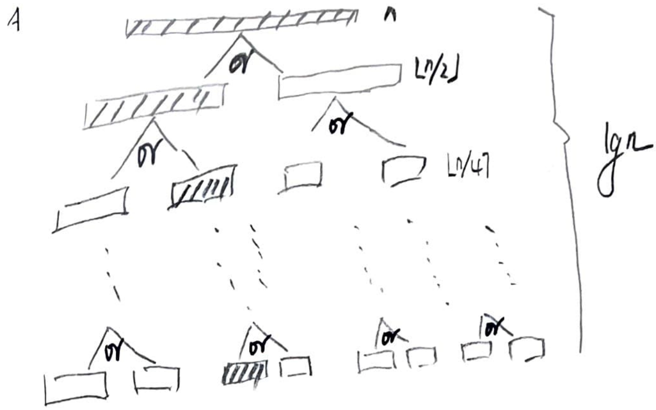
$\rightarrow C[n-1] > C[n]$ that means C is an sorted array with decreasing order.

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 subarray.

But we only choose one subarray to continue

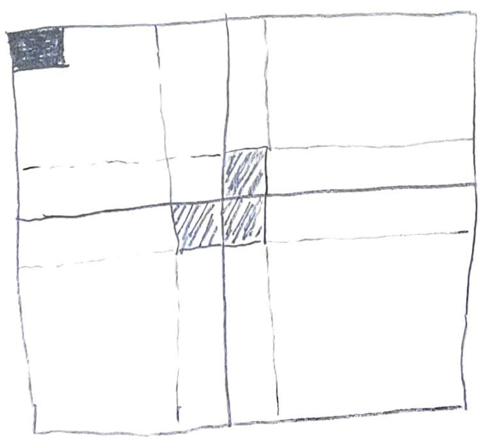
For each level, we need $O(1)$ running time for comparision

There are $O(\lg n)$ levels for the worse case

Total running time $O(\lg n)$

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

```
Cover-with-L-shape (2^n x 2^n)
  if n = 1 then
      It is a "L" shape. Done.
  else
      cover the center of 2^n x 2^n with a "L" shape. And dont
  cover the subsquare with the missing 1x1
      So, we have 4 2^(n-1) x 2^(n-1) each with a missing 1x1
      Do cover-with-L-shape (2^(n-1) x 2^(n-1)) 4 times
```



time complexity:

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

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

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

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 $(2^{2n}-1)/3$

$$\to \Theta(2^{2n}) = \Theta(4^n)$$