AOA Assignment 3

Sunny Shah - 112044068

October 22, 2018

Question 1

Page 246 problem #3

Solution

We can solve this problem using divide and conquer approach by maintaining record of a card that appeared most number of times in each sub problems and then combining this 2 subproblems to get the max card.

Algorithm

```
# The below function returns the Card that occured max number
# of times between start and end
findMaxCard( cards, start, end ) :

    if start == end: # only 1 card
        return cards[start] # since, it is the only max card

    # In case of 2 cards, it doesn't matter what we return,
    # as our merge step will take care of this

    if end-start+1 == 2: # 2 cards
        return cards[start]

    # Else we divide the problem into 2 halves and find the max
    # card in each of them and then use this information to
    # find the max card in the complete problem

    leftMaxCard = findMaxCard( cards, start , (start+end)/2 )
    rightMaxCard = findMaxCard( cards, (start+end)/2+1, end)

    # If the max card in both the left halve and right halve is
    # same then, we can safely take it as max card in the
    # complete problem. This step will require O(1) operation

    if leftMaxCard == rightMaxCard:
        return leftMaxCard

    # But if leftMaxCard and rightMaxCard are not same then
    # we will have to decide that who among leftMaxCard and
    # rightMaxCard is majority between start and end
    # To do this will we compare each card from start to end
```

```
    # with leftMaxCard and count its occurrence
    # Similarly we find the count for the rightMaxCard.

    # The counting operation will take O(n), which is our merge
    # complexity

    leftMaxCardCount = 0;
    rightMaxCardCount = 0;
    for i -> start to end:
        if leftMaxCard is equivalent to cards[i]:
            leftMaxCardCount++
        if rightMaxCard is equivalent to cards[i]:
            rightMaxCardCount++

    if leftMaxCardCount > rightMaxCardCount:
        return leftMaxCard
    else return rightMaxCard

Lets say we have a cards array, which contains the list of cards

cards[1....n] = [c1,c2,.... cn]

# find the max card in the cards list by calling our above method

majorityCard = findMaxCard(cards,1,n)

# However, we will have to check again if its count is actually greater than n/2

count = 0
for i -> 1 to n:
    if cards[i] is equivalent to cards:
        count++

if count>n/2:
    print "the max card is ", majorityCard
else:
    print "the card cannot be found"
```

Complexity

The recurrence for the algorithm can be return as:

$$T(n) = 2 * T(\frac{n}{2}) + n$$

The merge step takes only n operations Thus, the recurrence is similar to a merge sort algorithm and using the master's theoram, we can say that :

$$T(n) = O(n \log n)$$

Page 246 problem #5

Solution

Let's break the problem in finding the visible lines in first half of set of lines and same in second half of the set of lines. And then smartly combine them to find number of lines visible.

To efficiently divide the set of lines into subsets, let us sort them according to their slopes i.e. in the increasing order of their slopes.

In case two lines $y = a_i x + b_i$ and $y = a_j x + b_j$, with same slope $a_i = a_j$, delete the line with lower b value, as it will never be visible.

Algorithm

```
findVisibleLines( lines, start, end ):

    # if no of lines is 1, then it is always visible
    if start == end:
        return (lines,{})

    # Find intersection points of each adjacent line
    intersection_points = {}
    for i -> 0 to n-2:
        Add intersection(lines[i], lines[i+1]) to intersection_points

    # 2 intersecting lines are always visible
    if start+1 == end:
        return (lines, {intersection_points}

    if end - start + 1 == 3:  # 3lines
        # since, we have sorted based on slope, first line
        # will always be visible on the left part of all the
        # lines and third line will always be visible
        # on the right side of all the lines.

        # However, second line will only be visible in the
        # middle part of the region if its x-coordinate of
        # intersection with first line is less than that
        # with 3rd line. Thus, it will be visible between
        # its intersection with 1st and 3rd line

    if intersection(2nd line, 1st line).x < intersection(2nd line, 3rd line).x:
        return (lines, intersection_points)
    else
        return ((line 1, line 3), {intersection of 1 and 3})

    # if no. of lines > 3, then we sub-divide the problem
    # into 2 halves and then merge them as follows:

    (part1Lines, part1IntersectionPts) = findVisibleLines(lines, start, (start+end)/2)
    (part2Lines, part2IntersectionPts) = findVisibleLines(lines, (start+end)/2+1, end)

    # To merge part1Lines and part2Lines, we know that again
```

```
        # the lines[start] and lines[end] will always be visible from left
        # and right part of the region

        # All the lines from part1Lines will be visible, till the point
        # where the lines in part2Lines starts dominating part1Lines

        # To find such a point, we have to find to find the intersection point
        # from part1IntersectionPts and part2IntersectionPts,
        # such that max y-coordinate of intersection of lines
        # in part2Lines is greater than max y-coordinate for
        # lines in part1Lines for some x-coordinate of point in
        # part1IntersectionPts and part2IntersectionPts

        # Also, we will find the line with least index having
        # the max of y-coordinate, as we can safely assume that
        # all lines with index greater than it will be visible
        # as their slope is greater

        # join such that all_intersection_points contain
        # points in increasing order of x-coordinates
        all_intersection_points = Join( part1IntersectionPts, part2IntersectionPts )

        for each point in all_intersection_point:
            if max_y_coordinate of lines in part2Lines > same in part1Lines:
            breakpoint = point

        # Consider all points in part1IntersectionPts and find
        # the last line index(Li) where the intersection point
        # x-coordinate with the next line is less than the
        # breakpoint x-coordinate

        # Similarly, find least line index(Lj) in part2IntersectionPts,
        # where the intersection point x-coordinate with next
        # line is greater than or equal to the breakpoint x-coordinate

        # Now Find the intersection of line Li and Lj, call it
        # partitionPoint

        # lines1, intersection_points_1:
        # Our task is now reduced to finding all lines in Part1Lines,
        # where the intersection point x-coordinate with next line is less
        # than the partitionPoint x-coordinate

        # lines2, intersection_points_2:
        # all lines in part2, where the intersection point
        # x-coordinate with next line is greater
        # than the partitionPoint x-coordinate

        return ( ( lines1 U lines2 ) ,  {, intersection_points_2, intersection_points_2} )
```

Complexity

The recurrence for the algorithm can be return as:

$$T(n) = 2 * T(\frac{n}{2}) + n$$

The merge step takes only n operations Thus, the recurrence is similar to a merge sort algorithm and using the master's theoram, we can say that :

$$T(n) = O(\ n \log n\ )$$

Page 246 problem #6

Solution

The problem can be solved in single top to bottom pass. Here is algorithm:

1) Start from root and perform the step 2

2) Check if it is a leaf node. If yes, then it is the local minima, since we choose any node which has lower value than the current node as seen in the step 4

3) If it is not the leaf node, then compare its value with its child. If both the childs are bigger, then we have found the local minima

4) If the current is not the local minima, then check from any of its smallest child. Thus, the child need to only look at his child to check if it is a local minima.

Algorithm

```
findLocalMinima( Node ): # returns the local minima node

    if Node.right == null and Node.left == null:
        # it is the leaf element. Thus, it is local minima
        # as it is smaller than its parent (its only neighbour)
        return Node

    if Node.val < Node.right.Val and Node.val < Node.left.val:
        # It is the local minima, as we only traversed this Node
        # from a higher valued parent.
        return Node

    if Node.val > Node.left.val:
        return findLocalMinima(Node.left)
    else
        return findLocalMinima(Node.right)

# Find the local minima for the tree rooted at Node root
findLocalMinima( root )
```

Complexity

The algorithm traverses nodes once in any path from root to leaf and thus, it passes through only log(n) nodes in the tree, where log(n) = d depth of the complete binary tree.

Therefore, the overall complexity of the algorithm is O(log(n))

2 a) $A(n) = 4A\left(\lfloor n/2 \rfloor + 5\right) + n^2$

if we assume $\left(\lfloor \frac{n}{2} \rfloor + 5\right) \approx n/2$, then as per master's theorem,

we have,

$k\ f(n) = a\ f(n/b) = 4\ f(n/2) = 4n^2/4 = (1)\ n^2$

$\therefore k = 1$

$\therefore A(n) = O(n^2 \log_b n) = O(n^2 \log_2 n)$

To prove above, we should prove that

for all c, $A(n) \leq cn^2 \log_2 n$

$$A(n) = 4T\left(\lfloor \frac{n}{2} \rfloor + 5\right) + n^2$$

$$\leq 4\ c(n/2)\left(\lfloor \frac{n}{2} \rfloor + 5\right)^2 \log_2\left(\lfloor \frac{n}{2} \rfloor + 5\right) + n^2 \quad (\text{as } A(n) \leq cn^2 \log n)$$

$$\leq 4c\left(\frac{n}{2} + 5\right)^2 \log_2\left(\frac{n}{2} + 5\right) + n^2$$

$$\leq c\left(n^2 + 20n + 100\right) \log\left(\frac{n}{2} + 5\right) + n^2$$

$$\leq cn^2 \log\left(\frac{n}{2} + 5\right) + 20cn \log\left(\frac{n}{2} + 5\right) + 100c \log\left(\frac{n+10}{2}\right) + n^2$$

$$\leq cn^2 \log(n+10) - cn^2 + 20nc \log(n+10) - nc(20) + 100c \log_2(n+10) - 100c + n^2$$

If n tends to ∞, $(n+10) \approx n$

$\therefore A(n) \leq cn^2 \log n - cn^2 + 2nc \log n - 20nc + 100c \log n - 100c + n^2$

To prove above, we just need to prove that,

$$cn^2\log n - cn^2 + 20nc\log n - 20nc + 100c\log n - 100c + n^2 \leq cn^2\log n$$

Dividing above by n^2;

$$c\log n - c + \frac{20c\log n}{n} - \frac{200c}{n} + \frac{100c\log n}{n^2} - \frac{100c}{n^2} + 1 \leq c\log n$$

if $n \to \infty$, we get

$$c\log n - c + 1 \leq c\log n$$

$\therefore$ L.H.S. will be smaller or equal to R.H.S. for $c \geq 0$

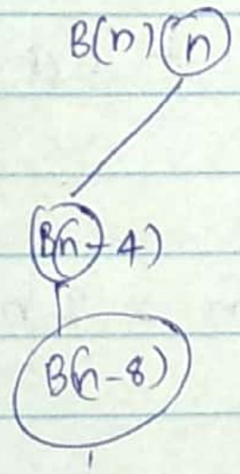
Thus $A(n) = O(n^2 \log_2 n)$

b)

$$B(n) = B(n-4) + 1/n + 5/(n^2+6) + 7n^2/(n^3+8)$$

$$\therefore B(n) = B(n-4) + \frac{1}{n} + \frac{5}{n^2+6} + \frac{7n^2}{n^3+8}$$

$$= B(n-8) + \frac{1}{n-4} + \frac{5}{(n-4)^2+6} + \frac{7(n-4)^2}{(n-4)^3+8} + \frac{1}{n} + \frac{5}{(n^2+6)} + \frac{7n^2}{(n^3+8)}$$



n/4 times.

∴ The recurrence will run n/4 i.e. n times.

$$\therefore B(n) = \sum_{i=1}^{n/4} \frac{1}{n-i4} + \frac{5}{(n-4i)^2+6} + \frac{7(n-4i)^2}{(n-4i)^3+8}$$

$$\sum_{i=1}^{n/4} \left(\frac{1}{n-4i}\right) = \left(\frac{1}{4} + \frac{1}{8} + \cdots \frac{1}{n}\right) = \frac{1}{4}\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots \frac{4}{n}\right)$$

$$\leq \log(n)$$

$$\sum_{i=1}^{n/4} \left(\frac{5}{(n-4i)^2+6}\right) = \left(\frac{5}{6} + \frac{5}{7} + \cdots \frac{5}{10} + \cdots \frac{5}{n^2+6}\right) \approx \frac{1}{1^2} + \frac{1}{2^2} + \cdots \frac{1}{n^2}$$

$$\approx \text{constant}$$

$$\sum_{i=1}^{n/4} \frac{7(n-4i)^2}{(n-4i)^3+8} = \frac{7}{(n-4i)+8/(n-4i)^2} \approx \left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots\right) \approx \log(n)$$

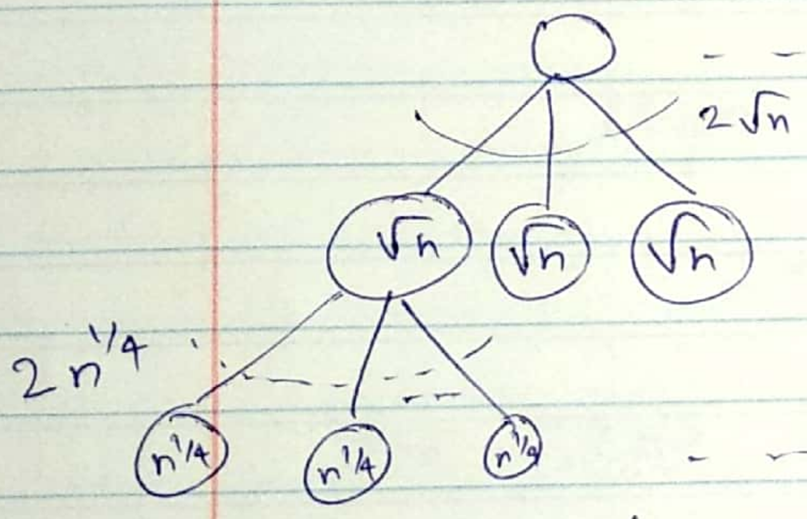
↑ $8/(n-4i)^2$ will be 0 for large n.

$\therefore B(n) \leq \log(n) + \text{constant} + \log(n)$

$\leq 2\log(n)$

$\therefore B(n) = O(\log n)$

c) $C(n) = n + 2\sqrt{n}\, C(\sqrt{n})$



Total Cost (n)
n
$2\sqrt{n} * \sqrt{n} = 2n$
$2\sqrt{n} * 2n^{1/4} * n^{1/4}$
$= 2^2 n$
$2^i n$

where is such that $n^{1/2^i} = \text{constant}$

$\therefore \log(n^{1/2^i}) = \log(\text{constant})$

$\frac{1}{2^i} \log(n) = \log(c)$

$\therefore \log n = \log c(2^i)$

$\log\log(n) = \log(\log c\, 2^i) = \log\log c + \log 2^i$

If $c = 2$

$\log\log(n) = \log\log 2 + \log 2^i = 0 + i(1)$

$\therefore i = \log(\log(n))$

$$\therefore \quad C(n) = \sum_{i=0}^{\log\log n} 2^i n$$

$$= n 2^0 + n 2^1 + \cdots + n 2^{\log\log n}$$

$$= n \left(1 + 2^1 + 2^2 + \cdots + 2^{\log\log n} \right)$$

$$\leq n \, 2^{\log\log n} \qquad \left(1 + 2^1 \cdots - 2^{\log\log n}\right) \leq 2^{\log\log n}$$

$$\leq n \log n$$

$$\therefore C(n) = O(n \log n)$$

Question 3

(a)Show that 5 multiplications are sufficient to compute the square of a 2x2 matrix

Answer:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} (\mathrm{a}^2 + \mathrm{bc}) & (b(a+d)) \\ (c(a+d)) & (bc+\mathrm{d}^2) \end{bmatrix}$$

Thus, squarring the 2x2 matrix requires following 5 multiplications:

a^2, d^2, bc, b(a+ d), c(a + d)

(b)

Answer:

In case of squaring a nxn matrix, we can write it as:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} * \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} (\mathrm{A}^2 + \mathrm{BC}) & (B(A+D)) \\ (C(A+D)) & (BC+\mathrm{D}^2) \end{bmatrix}$$

- In the above, equation we have written (AB + BD) = B(A+D). However, matrix multiplications are not commutative i.e. AB $\neq$ BA. Thus, we cannot break (AB+BD) = B(A+D), when it comes to matrix multiplication.
- Also, we cannot completely find the square of n x n matrix by just using idea of square of 2x2 matrix as discussed above, since, in the equation above, to complete square of a n x n matrix, we need square of 2 (n/2) matrix A,D, but we also need to find 4 matrix multiplication of 2 different (n/2) matrix
  (i.e. BC, BA, BD, C(A+D))

(c)

i)

Using the simple analogy, $(A+B)^2 = A^2 + (AB + BA) + B^2$
we can arrage this as follows:

$$(AB + BA) = (A+B)^2 - A^2 - B^2$$

Therefore, computing (AB + BA) can be done by computing just 3 matrix squares: $(A+B)^2$, A^2, B^2 and then we are just left with matrix addition and subtraction operations, which can be done in n^2

Thus, the overall complexity of computing (AB+BA) will be $3S(n) + n^2$

ii)

AB =

$$\begin{bmatrix} X & 0 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & Y \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & XY \\ 0 & 0 \end{bmatrix}$$

And BA =

$$\begin{bmatrix} 0 & Y \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} X & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Therefore, AB + BA =

$$\begin{bmatrix} 0 & XY \\ 0 & 0 \end{bmatrix}$$

iii)

- From (ii), we know that XY (each nxn matrix) can be computed using A and B (each 2nx2n matrix).
- Thus, we can calculate XY using (AB + BA) as shown in (ii)
- Using (i), we can show that (AB+BA) can be computed in $3S(2n)+(2n^2) \equiv 3S(2n)+4(n^2) \equiv 3S(2n)+O(n^2)$
- Thus, using (i) and (ii), we can show that XY can be computed in $3S(2n) + O(n^2)$