CS3510-A: Design and Analysis of Algorithms, Spring 2021

Homework-5

February 16, 2021

DUE DATE: Tuesday, February 23, 1l:59pm

Note-1: Your homework solutions should be electronically formatted as a single PDF document that you will upload on Gradescope. If you have to include some handwritten parts, please make sure that they are very clearly written and that you include them as high resolution images.
Note-2: Please think twice before you copy a solution from another student or resource (book, web site, etc). It is not worth the risk and embarrassment.
Note-3: You need to **explain/justify** your answers. Do not expect full credit if you just state the correct answer.
Note-4: You will get 2 extra points if you submit electronically typed solutions instead of hand-written.

Problem-1 (30 points)

A binary tree is called "complete" if (1) every internal node has two children, and (2) every leaf node has the same depth (distance from the root).

Describe a divide-and-conquer algorithm that computes the largest complete subtree T* of a given binary tree T. The algorithm should return both the root and the depth of T*. Note that the leaves of T* may not be leaves in the T (i.e., T* may be "internal" in T). Also analyze the run time of your algorithm.

Solution

```python
def find_depth(node):
    '''
    Gets the root node for the largest possible complete tree.
    Returns the depth of the tree and the root node.

    node: the current node in the traversal
    '''

    # if current node has less than two children
    # (can't be the root of a perfect tree larger than 0 edge depth)
    if node.left == None or node.right == None:
        return 0, node
    else:
        # Create two subproblems with two halves of tree
        # alpha = 2; beta = 2
        l = find_depth(node.left)[0]
        r = find_depth(node.right)[0]

        # Combine/select subproblem solutions and return
        if l == r:
            return l + 1, node
        elif l > r:
            return l, node.left
        elif l < r:
            return r, node.right
```

Explanation

The algorithm starts the recursive solution at the root node for the whole tree. If a node has < 2 child nodes then it is a base case and $(0, curr)$ is returned as the solution for the subproblem. If the current node has two children then two subproblems are created, one for each child node. This results in two subproblems with size $n/2$. Once the solutions two the two subproblems are found they are "combined". If the two solutions return an identical depth, then they can become two symmetrical halves with the current node as the new root (solution is $(l + 1, curr)$). If one subproblem's solution is larger than the other's solution, then that $(depth, node)$ solution gets passed up to the parent call until a larger tree can be made. This combination of solutions can be done in $O(1)$.

Runtime

General runtime for this algo:
$T(n) = 2T(n/2) + O(1)$

For the master theorem, the values are:
$\alpha = 2$: because each call of the function results in two recursive subcalls.
$\beta = 2$: because the size of the tree for each subcall call will be at most of size $n/2$.
$\gamma = 0$: because the work of verifying the solution is $O(1)$, meaning for $O(n^\gamma)$, $\gamma = 0$.

In this case: $\gamma < \log_\beta \alpha$ $(0 < \log_2 2)$
Therefore this algorithm has: $O(n^{\log_2 2)}$ or $O(n)$.

Problem-2 (35 points)

Suppose that we have two sorted lists a and b. The size of a is n entries and the size of b is m entries. Design an algorithm to find the k'th smallest element in the union of a and b in $O(log(n+m))$ time. Please also analyze the run time of your algorithm.

Solution

```
def kth(a, b, k, i=0, j=0):
    # return using the rest of opposing array once one
    # pointers reaches the end of its corresponding array
    if (i == len(a)):
        return b[j + k - 1]
    if (j == len(b)):
        return a[i + k - 1]

    # Return null if invalid inputs
    if (k == 0 or k > (len(a) - i) + (len(b) - j)):
        return -1

    # if k is 1 then return the kth value
    # (should be the smaller value at the two pointers)
    if (k == 1):
        if(a[i] < b[j]):
            return a[i]
        else:
            return b[j]

    # Select an array to to return from,
    # or increment its pointer and recurse
    if(k//2 - 1 >= len(a) - i):
        # if end of a is smaller than current point at b,
        # then return correct value from b
        if (a[-1] < b[j + k//2 - 1]):
            return b[j + (k - (len(a) - i) - 1)]
        else:
            return kth(a, b, k - k//2, i, j + k//2)
    if (k//2 - 1 >= len(b) - j):
        if (b[-1] < a[i + k//2 - 1]):
            return a[i + (k - (len(b) - j) - 1)]
        else:
            return kth(a, b, k - k//2, i + k//2, j)

    else:
        # if a at pointer is smaller, then for a
        # halve k, increment the pointer, and recruse
        if (a[k//2 + i - 1] < b[k//2 + j - 1]):
            return kth(a, b, k - k//2, i + k//2, j)

        # for b halve k, increment the pointer, and recruse
        else:
            return kth(a, b, k - k//2, i, j + k//2)
```

Explanation

By using a pointer for each array and recursively incrementing pointers by half of k and recursively halving k, the algorithm essentially performs a binary search over both arrays at once. The algorithm has several checks to the behavior of having two pointers. If one pointer reaches the end of its array then the corresponding value at the pointer in the other array will be returned. If no value can be returned, then the pointer for either array will be incremented by $k/2$ and k will decrease

by $k/2$. This recursion results in at most one subcall for any call to the function and moving the pointers eliminates at most half the total values in all arrays. "Combining" the solution from a subcall is done by simply retuning the result to the parent call, therefore combining results can be done in $O(1)$.

Runtime

General runtime for this algo:
$T(n) = T(n/2) + O(1)$

For the master theorem, the values are:
$\alpha = 1$: because each call of the function results in only one recursive call.
$\beta = 2$: because the total number of values considered by the algo decreases by $n/2$ for each subcall
$\gamma = 0$: because the "combination" of results can be done in $O(1)$, meaning for $O(n^\gamma)$, $\gamma = 0$

In this case: $\gamma = \log_\beta \alpha$ $(0 = \log_2 1)$
Therefore this algorithm has: $O(\log n)$ but in this case n is the total number of values in both arrays so the real runtime is $O(\log(n + m))$.

Problem-3 (35 points)

You are given n non-vertical lines in the plane, labeled $L_1, ..., L_n$, with the i'th line specified by the equation $y = a_i * x + b_i$. We will make the assumption that no three of these lines meet at a single point and these lines extend infinitely.

We say that line L_i is **uppermost at a given x-coordinate** x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at x_0 : $a_i * x_0 + b_i > a_j * x_0 + b_j$ for all $j \neq i$.

We say that line L_i is **visible** if there is some x-coordinate at which L_i is uppermost. Intuitively, some portion of L_i can be "seen" if you look down from $y = +\infty$.

Design an algorithm that takes n lines as input and in $O(n \log n)$ time returns the set of lines that are visible. Fig. 1 gives an example.

Please also analyze the run time of your algorithm.

Note: Make sure that your solution uses the divide-and-conquer approach.

Hint: You can first sort all lines by their slopes, and then split the problem.

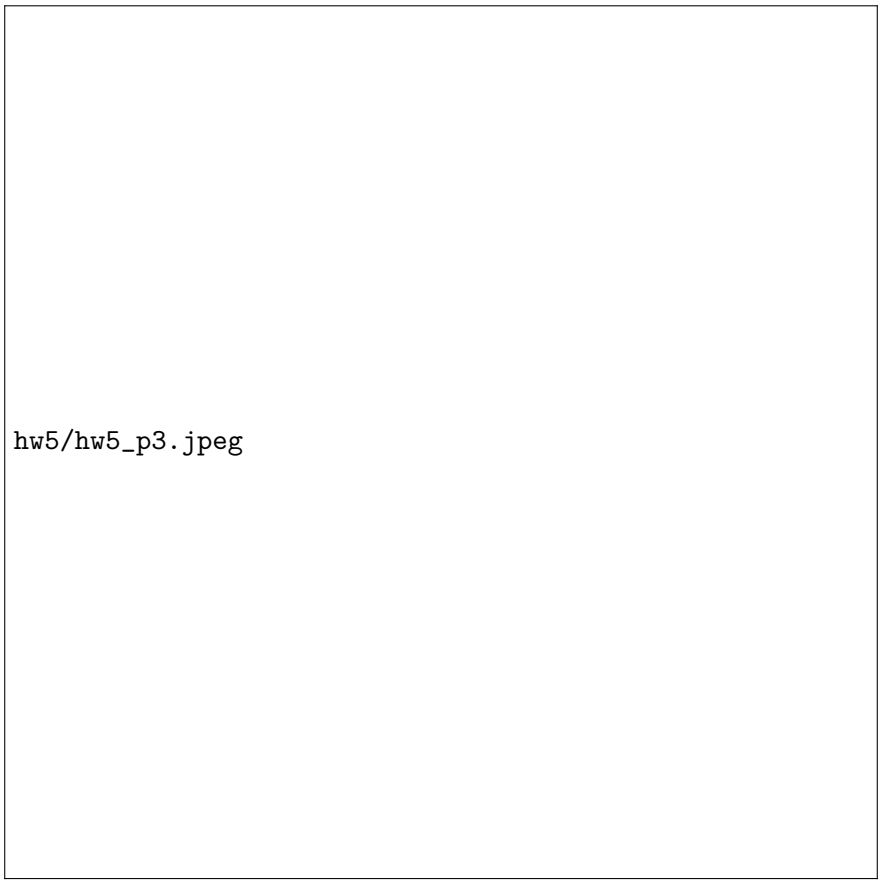

Figure 1: All the lines except for 2 are visible.

Solution

```python
def solution(a, b):
    '''
    Find solution of two lines.
    Uses m and b values (from format y=mx+b)
    '''
    slopes = a.m - b.m
    offsets = b.b - a.b
    x = offsets / slopes
    y = a.f(x)
    return (x, y)

def b_visible(a, b, c):
    '''
    Checks that the line with middle slope (b) is visible
    '''
    x, y = solution(a, c)
    return b.f(x) > y

def visible(lines):
    '''
    Takes in a list of lines.
    Lines have values m and b (to fit format y=mx+b)
    Value on a line at x can by found by line.f(x) (to fit format f(x)=mx+b)
    '''
    lines.sort(key=lambda x: x.m)
    return visible_rec(lines)

def visible_rec(lines):
    '''
    Recursive function to find solutions
    '''
    # Base cases
    if len(lines) <= 3:
        if len(lines) < 3:
            return lines
        else:
            a, b, c = lines
            if b_visible(a, b, c):
                return lines
            else:
                return [a, c]

    middle = len(lines)//2

    # Recurse down to find solutions to sub problems
    # alpha = 2; beta = 2
    left = visible_rec(lines[:middle])
    right = visible_rec(lines[middle:])

    combined = left.copy()
    for line in right:
        combined.append(line)

        # Check if new line "hides" previous lines O(n)
        last_removed = True
        while last_removed:
            if len(combined) >= 3:
                a, b, c = combined[-3:]
                if not b_visible(a, b, c):
                    combined.remove(b)
                else:
                    last_removed = False
            else:
                last_removed = False

    return combined
```

Explanation

The algorithm sorts the list of lines by slope. The algorithm then recursively splits the list of lines into two lists, split at the middle of the list (two groups of size n/2). Once the recursive function receives a list with length ≤ 3 if length ≤ 2, then the algo returns the list. If length $= 3$ then the function checks that median line (by slope) is visible ($f(x)$ for the median line is greater than the $f(x)$ for the other lines where x is the intersection of the other two lines). Finding the intersection of two lines and checking the visibility of the median line in a list of three sorted lines can both be

done in $O(1)$. Once two subproblems have been solved the algorithm combines two solutions by iteratively adding solutions from the "right" (higher slope) side to solutions from the "left" (lower slope) side, while checking if newly added lines will "hide" existing lines in the larger solution. If an existing line in the combined solution is hidden by the newly added line then the hidden line will be removed from the solution. This combination of solutions for subproblems can be done in $O(n)$.

Runtime

General runtime for this algo:
$T(n) = 2T(n/2) + O(n)$

For the master theorem, the values are:
$\alpha = 2$: because each call that is not a base case will result in the creation of two subproblems.
$\beta = 2$: because the input size for each subproblem will be $n/2$.
$\gamma = 1$: because combining suproblem solutions can be done in $O(n)$, meaning for $O(n^\gamma), \gamma = 1$

In this case: $\gamma = \log_\beta \alpha$ $(1 = \log_2 2)$
Therefore this algorithm has: $O(n \log n)$