CSE 202: Design and Analysis of Algorithms **(Due: 10/19/19)**

Homework #1

Instructor: Ramamohan Paturi *Name:* Shihan Ran, *Netid:* A53313589

Problem 1: Maximum weight subtree

Problem Description

The maximum weight subtree is as follows. You are given a tree T together with (not necessarily positive) weights $w(i)$ for each node $i \in T$. A subtree of T is any connected subgraph of T, (so a subtree is not necessarily the entire subtree rooted at a node). You wish to find a subtree of T that maximizes $\sum_{i \in S} w(i)$. Design an efficient algorithm for solving this problem. Note that there is a linear (in the number of nodes of the tree) time algorithm for this problem. You can assume that for each node in the tree T, you are given a list of its children as well as the parent pointer (except for the root node).

Solution

(High-level description)

Assume for each node of this tree, it at most has m children nodes. Our solution is to do a tree traversal. At each node, we iterate all the children nodes and find their subtree value. Then the value of subtree rooted at the current node is equal to the sum of current node value and the subtree values of all m children nodes. After getting the current subtree weight sum, we compare it with the stored maximum value and take the largest.

(Pseudo Code)

Algorithm 1: Maximum weight subtree

Input: root node *root*
Output: $maxWeightSum$

```
if root is not valid then
    return 0;
initialize maxWeightSum as the min value of the integer, say, -inf;
findMaxWeightSum(root, maxWeightSum);
return maxWeightSum;
```

Algorithm 2: Find max weight sum

Input: root node *root*, $maxWeightSum$
Output: $maxWeightSum$

```
if root is not valid then
    return 0;
weightSum = root.value;
for child in root.children do
    weightSum = weightSum + findMaxWeightSum(child, maxWeightSum)
maxWeightSum = max(maxWeightSum, weightSum) return weightSum;
```

(Correctness)

(Time complexity)

Since the tree traversal only visits each node once, hence the time complexity is $O(n)$, and n is the number of nodes.

We can also interpret the complexity as the following recursion tree:

$T(n) = m * T(n/m) + c$.

And the second step is $T(n/m) = m * T(n/(m^2)) + c$, thus $T(n) = m^2 * T(n/(m^2)) + 2c + c$ ...

And the last step is $T(n) = n * T(1) + c(1 + m^1 + ...m^h)$, with h is the hight of the recursion tree $h = \log_m(n)$.
So the time complexity is $O(n * c + c * (m^(h + 1) - 1)/(m - 1)) = O(n)$

Problem 2: Sorted matrix search

Problem Description

Given an $m \times n$ matrix in which each row and column is sorted in ascending order, design an algorithm to find an element.

Solution

(High-level description)

The basic brute force solution is iterate through each row and column, compare the value of the current element with the target value. However, this takes O(mn) time complexity.

What we should do is taking advantage of the condition that each row and column is sorted in ascending order, which means we can start iterating from the corner elements and follow a zigzag path to find the target value.

(Pseudo Code)

Algorithm 3: Sorted matrix search

```
Input: matrix as a two-dimensional array, target
Output: res
initialize res as an empty list;
if len(matrix) == 0 then
    // invalid input, return an empty list;
    return res;
// initialize indexes, starting from the bottom left corner;
colIndex, rowIndex = 1, len(matrix);
// find the rough row index;
while rowIndex >= 0 and target < matrix[rowIndex][colIndex] do
    rowIndex = rowIndex - 1;
// as long as column index is valid;
while colIndex <= len(matrix[0]) do
    if target == matrix[rowIndex][colIndex] then
        // found the target element;
        add (rowIndex, colIndex) to the res;
        return res;
    if target > matrix[rowIndex][colIndex] then
        // since the column is sorted in ascending order;
        colIndex = colIndex + 1;
    // if target < matrix[rowIndex][colIndex];
    else if rowIndex > 0 then
        rowIndex = rowIndex - 1;
    // row index is invalid;
    else
        return res;
    // column index is invalid;
    return res;
```

(Correctness)

(Time complexity)

In the worst case, we need to iterate through one row and one column, ending up in the upper right corner element, which means the time complexity is O(m+n).

Problem 3: Largest set of indices within a given distance

Problem Description

You are given a sequence of numbers $a_1, \ldots, a_n$ in an array. You are also given a number k. Design an efficient algorithm to determine the size of the largest subset $L \subseteq \{1, 2, \cdots, n\}$ of indices such that for all $i, j \in L$ the difference between a_i and a_j is less than or equal to k. There is an $O(n \log n)$ algorithm for this problem. For example, consider the sequence of numbers $a_1 = 7, a_2 = 3, a_3 = 10, a_4 = 7, a_5 = 8, a_6 = 7, a_7 = 1, a_8 = 15, a_9 = 8$ and let $k = 3$. $L = \{1, 3, 4, 5, 6, 9\}$ is the largest such set of indices. Its size is 6.

Solution

(High-level description)

Since we only need to determine the size of the largest subset L instead of determining the subset L itself, indexes are not so important then. What we really care about is the differences between elements and the max differences should be less than or equal to k.

Thus, we can sort this array first, and then iterate through the sorted array, find out the maximum length of continuous elements with a maximum difference less than or equal to k.

One tricky thing is how to find the maximum length efficiently. Our solution is using two pointers, the left one stays at index i and the right one points to index j. We have $i < j, A[i] <= A[j]$.

- While $A[j] - A[i] <= k$, we shift our right pointer.
- Compare the current length $j - i$ with the longest size we've seen before, and take the maximum of these two values.
- Shift the left pointer to $i+1$, since our array is sorted, then $A[i+1] >= A[i]$, which means $A[j]-A[i+1] <= k$ still stands. However this time the length is $j - (i + 1)$, and it must be less than the maximum value we just got. Hence, we can shift our right pointer to find the newest index j'.
- Go to Step 1.

After iterate through the whole array, we will get the maximum length *size*.

(Pseudo Code)

Algorithm 4: Largest set of indices within a given distance

Input: A as an array, k
Output: *size*

sort the input array A;
initialize size as the min value of the integer, say, $-\inf$;
$leftPointer = 1, rightPointer = 1$;
while $rightPointer <= len(A)$ **do**
  while $A[rightPointer] - A[leftPointer] <= k$ **do**
    $rightPointer = rightPointer + 1$;
  $size = \max(size, rightPointer - leftPointer)$;
  $leftPointer = leftPointer + 1$;
return $size$;

(Correctness)

(Time complexity)

The sorting algorithm can take $O(n \log n)$ time complexity. As for the remaining part, for both left pointer and right pointer, we only need to iterate through the array once, hence it's $O(n)$ time complexity.

Overall, it's $O(n \log n)$.

Problem 4: Pond sizes

Problem Description

A Toeplitz matrix is an $\boldsymbol{n} \times \boldsymbol{n}$ matrix $A = (a_{ij})$ such that $a_{ij} = a_{i-1,j-1}$ for $i = 2, 3, ..., n$ and $j = 2, 3, ..., n$.

- Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?
- Describe how to represent a Toeplitz matrix so that two $\boldsymbol{n} \times \boldsymbol{n}$ Toeplitz matrices can be added in $O(n)$ time.
- Give an $O(nlgn)$-time algorithm for multiplying an $\boldsymbol{n} \times \boldsymbol{n}$ Toeplitz matrix by a vector of length n. Use your representation from part (b).
- Give an efficient algorithm for multiplying two $O(nlgn)$ Toeplitz matrices. Analyze its running time.

Solution

(High-level description)

(Pseudo Code)

Algorithm 5: identify Row Context

Input: r_i, $Backgrd(T_i)$=$T_1, T_2, \ldots, T_n$ and similarity threshold θ_r
Output: $con(r_i)$

$con(r_i) = \Phi$;
for $j = 1; j \leq n; j \neq i$ **do**
  float $maxSim = 0$;
  $r^{maxSim} = null$;
  while *not end of* T_j **do**
    compute Jaro$(r_i, r_m)(r_m \in T_j)$;
    if $(Jaro(r_i, r_m) \geq \theta_r) \wedge (Jaro(r_i, r_m) \geq r^{maxSim})$ **then**
      replace r^{maxSim} with r_m;
  $con(r_i) = con(r_i) \cup r^{maxSim}$;
return $con(r_i)$;

(Time complexity)

(Data Structure)