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)

We update $maxWeightSum$ each time we use $findMaxWeightSum(root, maxWeightSum)$ function to find a maximum weight sum. Every time we calculate the weight sum, we compare it with the maximum weight sum we've seen so far.

The weight sum at each node is computed as root weight plus all the children's weights. There are two cases,

1. We choose the root node. Then the maximum weight sum should be children's maximum weight sum plus node weight. We'll iterate through each child and add up the final weight sum, then compare the weigh sum with the maximum sum we've seen so far.

2. We don't choose the root node. Then the maximum weight sum is one of the children's maximum weight sum. We'll update the $maxWeightSum$ when we iterate the $findMaxWeightSum$ function to compute children's value.

Also, our algorithm terminates. It returns 0 for leaf's children node value if the node is a leaf node. Thus, it's correct.

(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: indexes of the element

```
if len(matrix) == 0 then
    // invalid input, return an empty list;
    return 0;
// 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;
        return (rowIndex, colIndex);
    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 0;
    // column index is invalid;
    return 0;
```

(Correctness)

The element we want to find is x. If x is in the matrix, let's prove x must be in the submatrix whose bottom left corner is y.

1. This clearly holds in the beginning, since our algorithm starts from the bottom left corner and y is the bottom left corner of the input matrix. It will output y's indexes.

2. If $y > x$, then all elements from y to the right end are also larger than x, so x cannot be contained in that row. Therefore our assumption continues to hold with x replaced with its neighbor below (if any. If y has no below neighbors, we can deduce that x is not found in the matrix).

3. If $y < x$ then all elements below y are also smaller than x, and so x cannot be contained in that column. Therefore the invariant continues to hold with y replaced with its right neighbor (if y has no right neighbors, we can deduce that x is not found in the matrix).

On a $m \times n$ matrix, this process terminates in at most $m + n$ steps, either finding x or determining that it does not appear in the matrix.

(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'.
- $i = i + 1$, $j = j'$. 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)

After sorting, the array should be kept in ascending order. If the maximum difference between $A[i]$ and $A[j]$ is k, then we prove that if there exists any x, and $i \leq x \leq j$, then the maximum difference between $A[x]$ and $A[j]$ is less than or equal to k.

Proof: Since our array is sorted, then $A[x] >= A[i]$ and we have $A[j]-A[i] <= k$, which means $A[j]-A[x] <= k$ still stands.

The length of this subset is $j-x$, and it must be less than the maximum size value we just got $\max(max, j-i) >= j-i >= j-x$. Hence, for x, our algorithm again tries to find the maximum right boundary. By doing this, our algorithm ensures for each $A[i]$, we find the maximum length of the subset for all elements in the subset where the difference between any pair of elements is less than or equal to k.

(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: Toeplitz matrices

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 $j = 2, 3, ..., n$.

(SubProblem1)

Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?

Solution

Define two Toeplitz matrices A and B. According to the definition, we have $a_{ij} = a_{i-1,j-1}$ and $b_{ij} = b_{i-1,j-1}$.

1. The sum of two Toeplitz matrices is also a Toeplitz since

$$(A + B)_{ij} = a_{ij} + b_{ij} = a_{i-1,j-1} + b_{i-1,j-1} = (A + B)_{i-1,j-1}$$

2. The product of two Toeplitz matrices is not always a Toeplitz since

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = \sum_{k=1}^{n} a_{i-1,k-1} b_{k-1,j-1} \neq \sum_{k=1}^{n} a_{i-1,k} b_{k,j-1} = (AB)_{i-1,j-1}$$

(SubProblem2)

Describe how to represent a Toeplitz matrix so that two $\boldsymbol{n} \times \boldsymbol{n}$ Toeplitz matrices can be added in $O(n)$ time.

Solution

Any $\boldsymbol{n} \times \boldsymbol{n}$ Toeplitz matrix has the form:

$$A = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \ddots & & \vdots \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{n-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{bmatrix}$$

We observed that there are only $2n - 1$ diagonals in this matrix and the values on a diagonal are always the same. Hence, we only need a $2n - 1$ length list to store all the diagonal values and to represent a Toeplitz matrix.

In detail, we only need to store the first row and first column values in the list, which are

$$a_0, a_{-1}, \ldots, a_{-(n-1)}, a_1, a_2, \ldots, a_{n-1}$$

We further adapt the indexes to $1, 2, \ldots, 2n - 1$, thus the vector should be $a = a_1, a_2, \ldots, a_{2n-1}$. A could be reconstructed as

$$\mathrm{A} = \begin{bmatrix} \mathrm{a_n} & \mathrm{a_{n-1}} & \cdots & \mathrm{a_1} \\ \mathrm{a_{n+1}} & \mathrm{a_n} & \cdots & \mathrm{a_2} \\ \vdots & \vdots & \ldots & \vdots \\ \mathrm{a_{2n-2}} & \mathrm{a_{2n-3}} & \cdots & \mathrm{a_{n-1}} \\ \mathrm{a_{2n-1}} & \mathrm{a_{2n-2}} & \cdots & \mathrm{a_n} \end{bmatrix}$$

(Time complexity)

Then two Toeplitz matrices can be added in $O(n)$ time and space complexity.

(SubProblem3)

Give an $O(n \log n)$-time algorithm for multiplying an $\boldsymbol{n} \times \boldsymbol{n}$ Toeplitz matrix by a vector of length n. Use your representation from part (b).

Solution

We multiply an $\boldsymbol{n} \times \boldsymbol{n}$ Toeplitz matrix A by a vector of length n, and the output size will be $\boldsymbol{n} \times \mathbf{1}$. As the matrix multiplication rule, we have

$$\mathrm{A} = \begin{bmatrix} \mathrm{a_n} & \mathrm{a_{n-1}} & \cdots & \mathrm{a_1} \\ \mathrm{a_{n+1}} & \mathrm{a_n} & \cdots & \mathrm{a_2} \\ \vdots & \vdots & \cdots & \vdots \\ \mathrm{a_{2n-2}} & \mathrm{a_{2n-3}} & \cdots & \mathrm{a_{n-1}} \\ \mathrm{a_{2n-1}} & \mathrm{a_{2n-2}} & \cdots & \mathrm{a_n} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$$(A \times b)_{ij} = c_i = \sum_{k=1}^{n} a_{n+i-k} b_k, \text{ for } 1 \leq i \leq n$$

The above formulation can be reconstructed as

$$a = \begin{bmatrix} a_1 & a_2 & \cdots & a_{2n-1} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$temp = a \oplus b$$

We compute the convolution of vectors a and b as $temp$, then output the $(A \times b)_{ij} = c_i = temp_{n+i}$.

(Correctness)

We prove the algorithm's correctness by using the definition of convolution and matrix multiplication, which is:

$$temp_i = \sum_{k=1}^{i} a_{i-k} b_k \text{ for } 1 \leq i \leq 2n-1$$

$$c_i = \sum_{k=1}^{n} a_{n+i-k} b_k, \text{ for } 1 \leq i \leq n$$

$$(A \times b)_{ij} = c_i = temp_{n+i}, 1 \leq i \leq n$$

(Time complexity)

Computing the convolution of two vectors of size $2n-1$ requires a time complexity of $O(n \log n)$.

(SubProblem4)

Give an efficient algorithm for multiplying two $\boldsymbol{n} \times \boldsymbol{n}$ Toeplitz matrices. Analyze its running time.

Solution

(High-level description)

We are given two $\boldsymbol{n} \times \boldsymbol{n}$ Toeplitz matrices as followes:

$$\mathrm{A} = \begin{bmatrix} \mathrm{a_n} & \mathrm{a_{n-1}} & \cdots & \mathrm{a_1} \\ \mathrm{a_{n+1}} & \mathrm{a_n} & \cdots & \mathrm{a_2} \\ \vdots & \vdots & \dots & \vdots \\ \mathrm{a_{2n-2}} & \mathrm{a_{2n-3}} & \cdots & \mathrm{a_{n-1}} \\ \mathrm{a_{2n-1}} & \mathrm{a_{2n-2}} & \cdots & \mathrm{a_n} \end{bmatrix}, \mathrm{B} = \begin{bmatrix} \mathrm{b_n} & \mathrm{b_{n-1}} & \cdots & \mathrm{b_1} \\ \mathrm{b_{n+1}} & \mathrm{b_n} & \cdots & \mathrm{b_2} \\ \vdots & \vdots & \dots & \vdots \\ \mathrm{b_{2n-2}} & \mathrm{b_{2n-3}} & \cdots & \mathrm{b_{n-1}} \\ \mathrm{b_{2n-1}} & \mathrm{b_{2n-2}} & \cdots & \mathrm{b_n} \end{bmatrix}$$

What we want to do is to multiply these two Toeplitz matrices.

According to the definition of Toeplitz matrices, we get

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = \sum_{k=1}^{n} a_{n+i-k} b_{n-j+k}$$

For example, for $(1,1)$ and $(2,2)$, we have

$$(AB)_{1,1} = a_n * b_n + a_{n-1} * b_{n+1} + \cdots + a_1 * b_{2n-1} = \sum_{k=1}^{n} a_{n+1-k} b_{n-1+k}$$

$$(AB)_{2,2} = a_{n+1} * b_{n-1} + a_n * b_n + \cdots + a_2 * b_{2n-2} = \sum_{k=1}^{n} a_{n+2-k} b_{n-2+k}$$

Here comes the amazing thing: we noticed that $(AB)_{2,2} = (AB)_{1,1} - a_1 * b_{2n-1} + a_{n+1} * b_{n-1}$. And this rule also applies to other diagonals, which means, to compute for the final output matrix, for each one of the $2n-1$ diagonals, we only need to compute dot products over length n vector a and b. Then other values on this diagonal can be computed in constant time $O(1)$.

So, in practice, we first compute the first row and the first column of AB, by multiplying the corresponding rows and columns of the input matrices:

$$(AB)_{1j} = \sum_{k=1}^{n} a_{n+1-k} b_{n-j+k}$$

$$(AB)_{i1} = \sum_{k=1}^{n} a_{n+i-k} b_{n-1+k}$$

Then, we compute other values on each diagonal using the previously computed values.

(Correctness)

We want to prove that the values on each diagonal can be computed by simple addition and subtraction based on $(AB)_{1,j}$ and $(AB)_{i,1}$ iteratively.

We use the center diagonal as an example, which is $(AB)_{i,i}$.

$$\begin{aligned} (AB)_{i-1,i-1} &= \sum_{k=1}^{n} a_{n+i-1-k} b_{n-i-1+k} \\ (AB)_{i,i} &= \sum_{k=1}^{n} a_{n+i-k} b_{n-i+k} \\ &= (AB)_{i-1,i-1} - a_{i-1} b_{2n-i-1} + a_{n+i-1} b_{n-i+1} \end{aligned}$$

It's proved. After we calculate $(AB)_{1,j}$ and $(AB)_{i,1}$, then we can iteratively calculate all values on the $2n-1$ diagonals.

(Time complexity)

Computing $(AB)_{1j}$ and $(AB)_{j1}$ takes n^2 since we need to iterate through n elements and at each element, the computation of two length-n vectors dot products takes $O(n)$ time complexity.

Then after getting $(AB)_{1,j}$ and $(AB)_{i,1}$, we need to compute all values on the $2n-1$ diagonals, thus takes O(n) time. For each diagonal, we need to iterate through at most n elements, takes O(n). Each time we need an O(1) time computation to do an addition and a subtraction based on the previously computed values, thus, computing each diagonal value takes total $O(n * 1)$ time.

Overall speaking, this algorithm takes $O(n^2)$.