CSE 202 – Homework 0

Wenjun Zhang
A53218995
10/04/2017

Problem 1: Maximum contiguous rectangle

Algorithm description:

The algorithm is based on the following steps:

1. Divide the sequence equally into two subsequences, find the maximum area contiguous subsequence in the two subsequences.
2. Find the maximum area contiguous subsequence which involves elements from both the two subsequences we divided.
3. To find the maximum area contiguous subsequence including elements from the two divided subsequences, we need to consider all the possibilities with length from 2 to n.
4. To find the maximum area contiguous subsequence of the divided subsequence, go back to step 1 and run the algorithm recursively until the sequence's length is 1.

Proof of correctness:

The algorithm is based on divide-and-conquer approach. By dividing the sequence into two subsequences, the problem becomes smaller and the additional operation is to consider the areas that link the two subsequences. By running the algorithm recursively we can obtain the maximum area contiguous subsequence in each subsequence. And by comparing the 3 potential maximum area contiguous subsequence, the maximum area contiguous subsequence of the whole sequence is obtained.

Time Analysis:

The algorithm divides the subsequence into two equally parts, and need to find the maximum area contiguous subsequence including elements from the two divided subsequences, which will take $O(n^2)$. So the time complexity is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$$

which revolves in $O(n^2)$.

Have tried some other approaches, but haven't figure out a linear-time algorithm.

Problem 2: Balanced Parentheses

Algorithm description:

The algorithm is to scan the string from begin to end and use a stack to store the indices of the unmatched left brackets, with the following operations:

1. If currently it's a left bracket, the index is pushed into the stack.
2. If the stack is not empty and currently it's a right bracket, the previous unmatched left bracket is matched and we pop the top of the stack.
3. If currently it's a right bracket but the stack is empty, we change this right bracket into left bracket and push the index into the stack.
4. When we reach the end of the string, if the stack is empty then the string has been balanced. Otherwise, we record the size of the stack N. If N is odd we return -1 and if N is even, we change the top N/2 left brackets in the stack into right brackets.

Proof of correctness:

The stack stores the indices of unmatched brackets. Since for a brackets pair, the left bracket always appears first, we only need to store the indices of the unmatched left brackets. If a right bracket appears, it will match with the most recent unmatched left bracket, so the top index in the stack can be pop out. But if a right bracket appears while the stack is empty, this means there are currently no unmatched brackets and to make the parentheses balanced we have to change this right bracket into left bracket. Since now the bracket becomes a left one and it's unmatched, the index of it should be push into the stack.

When we reach the end of the string, if the stack is empty, it means that all the brackets has been matched. If the size of the stack is odd, there is no way to make the parentheses balanced. IF the size N is even but not zero, by changing the top N/2 left brackets in the stack into right brackets, the unmatched brackets can be matched.

Time Analysis:

The algorithm is to scan the string once, so the time complexity is O(n).

Problem 3: Olay

Algorithm description:

The algorithm is to find the median of the y coordinates. The basic idea is to use Quick Select algorithm to partition the array with pivot and find the K[th] largest element. Assuming the number of coordinates is n. If n is odd, K is equal to $\frac{n-1}{2}+1$. If K is even, we find the $(\frac{n}{2})^{th}$ and $(\frac{n}{2}+1)^{th}$ largest element, and the median is the average of two elements. To find the K largest element, the steps are as followed:

1. Select a y coordinate as pivot
2. Put y coordinates larger than pivot to the pivot's left of the pivot and y coordinates smaller than pivot to the pivot's right.
3. If the index of the pivot is equal to K, return the pivot as the median. If index of the pivot is less than K, keep checking left part to pivot and go back to step 1. If index of the pivot is greater than K, keep checking right part to pivot and go back to step 1.

Proof of correctness:

It's clear that $|y_i - M| + |y_j - M| \leq |y_i - y_j|$, if $y_i \leq M \leq y_j$, assuming $y_j \geq y_i$. For convenience, we define the function of the sum of the distance:

$$S = \sum_{i=1}^{n} |y_i - M|$$

Assuming y is sort with the order from small to large. There are two conditions:

1. n is even: for any pair $\{y_i, y_{n-i}\}$, $|y_i - M| + |y_{n-i} - M| = |y_{n-i} - y_i|$ for $1 \leq i \leq \frac{n}{2}$ and $y_{\frac{n}{2}} \leq M \leq y_{\frac{n}{2}+1}$. Hence, the median $(y_{\frac{n}{2}} - y_{\frac{n}{2}+1})/2$ is an optimal result.
2. n is odd: for any pair $\{y_i, y_{n-i}\}$, $|y_i - M| + |y_{n-i} - M| = |y_{n-i} - y_i|$ for $1 \leq i \leq \frac{n-1}{2}$ and $M = y_{\frac{n+1}{2}}$. And for $|y_{\frac{n+1}{2}} - M| = 0$ when $M = y_{\frac{n+1}{2}}$. Hence, the median $y_{\frac{n+1}{2}}$ is the optimal result.

The Quick Select algorithm is to separate the elements smaller than the pivot to the left and elements larger than the pivot to the right, and return the index of the pivot. So its sufficient to find the K[th] largest element.

Time Analysis:

The average time complexity is $T(n) = T(n/2) + O(n) = O(n)$, the worst time complexity is $T(n) = T(n-1) + O(n) = O(n^2)$.

Problem 4: Maximum difference in a matrix

Algorithm description:

The algorithm is to use dynamic programming. The problem requests that M[c, d] – M[a, b] should satisfy c > a and d > b. So dp[i][j] stores the value of the largest element M[c, d] in matrix such that c > i and d > j. A the end we can return the largest result of dp[i][j] – M[i][j] where $0 \leq i \leq n$ and $0 \leq j \leq n$. The details are as followed:

1. Set the last column and last row of dp to be INT_MIN.
2. The DP formula is:

$$dp[i][j] = \max(dp[i][j+1], dp[i+1][j], M[i+1][i+1])$$

and the direction is from the bottom right to the top left.

3. Set the result res to be INT_MIN at the beginning, and then

$$res = \min(res, dp[i][j] - M[i][j]) \qquad for\ 0 \leq i \leq n, 0 \leq j \leq n$$

Proof of correctness:

The last column and last row of dp is set to be INT_MIN, so the result of $dp[i][j] - M[i][j]$ will be very small, which will not influence the maximum difference. The DP formula is:

$$dp[i][j] = \max(dp[i][j+1], dp[i+1][j], M[i+1][i+1])$$

where dp[i][j+1] involves all the candidates of the following columns and dp[i+1][j] involves all the candidates of the following rows. By combining M[i+1][j+1], dp[i][j] should have considered all the possible largest candidates that satisfies c > i and d > j. So this dynamic programming should compute the value of the largest element M[c, d] in matrix such that c > i and d > j, and by traversing the matrix again we can obtain the maximum difference.

Time Analysis:

The algorithm traverses the matrix twice, so the time complexity is O(n).

Problem 5: Perfect matching in tree

Algorithm description:

The algorithm determines if the root is null, we return true. If the root is not null while both its left and right node are null, we return false.

For a root, if it only has one node, it can be separated into 2 subtrees, where the roots are the left and right nodes of the left node or right node of the root.

If the root has both left and right nodes, the tree can be separated into 3 subtrees, where the roots are the right node of the root, left node of the left node, the right node of the left nod; or the left node of the root, left node of the right node, the right node of the right node.

For each subtree root, we run the algorithm recursively.

Code:

```
bool PM(TreeNode* root){
    if(!root)
        return true;
    if(!root->left && !root->right)
        return false;
    else if(!root->left)
        return PM(root->right->left) && PM(root->right->right);
    else if(!root->right)
        return PM(root->left->left) && PM(root->left->right);
    else
        return (PM(root->left->left) && PM(root->left->right) && PM(root->right))
|| (PM(root->right->left) && PM(root->right->right) && PM(root->left));
}
```

Proof of correctness:

If the root is null, it means no following untouched nodes, so it will return true. If the root is not null, while both of it left and right nodes is null, it means this root cannot be touched with perfect matching, so it will return false. For a root we can select either edges that connect it with its node, then we can separate the tree into 2 or 3 subtrees, then by determining if they are perfect matching, we can determine if the whole tree is

perfect matching.

Time Analysis:

The time complexity is

$$T(n) = 2(T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{4}\right) + O(1)$$

which is indeed a linear time O(n) algorithm.