CS515: Algorithms and Data Structures
Homework 1

Marjan Adeli (932969351), Rouzbeh Behnia (932987697), and
Abtin Khodadadi (932239415)

October 11, 2016

Problem 1

In order to address this problem, our algorithm uses the first element of preorder node sequence to find the root of tree, and locate the root in the inorder node sequence. Then, in inorder sequence, everything on the left side of the root element is the inorder node sequence of the left subtree and everything on the right side of of the root element is the inorder node sequence of the right tree. We use the length of the left and right subtrees of inorder to locate them in our preorder sequence and extract the preorder subsequence nodes of left and right subtrees . We then repeat the same method to the left and right subtrees until we recived to subtrees by length one, which are leaves all the nodes in the tree.

$const\_tree(preorder[i..j], inorder[i..j])$

$root = preorder[i]$

$if\ (i \neq j)$

$i_R \leftarrow find\_index(inorder[i..j], root)$

$root.LeftChild \leftarrow$

$cons\_tree(preorder[(i+1)\,..i_R], inorder[i..\,(i_R - 1)])$

$root.RightChild \leftarrow$

$cons\_tree(preorder[(i_R+1)\,..j], inorder[(i_R + 1)\,..n])$

$return\ \ root$

Given we have two arrays $(reorder[1...n], inorder[1...n])$ which contain the preorder and inorder sequence of our binary tree, our function *const_tree*

$(preorder[1...n], inorder[1...n])$ finds the index $root = pre[1]$, and uses the function $i_r \leftarrow find\_index(inorder[1...n], root)$ to locate the location of the root in the inorder array. We then apply the same function for the left and right subtrees "$cons\_tree(preorder[(i+1)..i_R], inorder[i..(i_R - 1)])$" and "$cons\_tree$ $(preorder[(i_R+1)..j], inorder[(i_R+1)..n])$" to extract their roots. This process continues until we find all the roots of the tree.

Problem 2

Two line Segments i and j intersect if and only if one of the following conditions hold.

- $p_i < p_j$ and $q_i > q_j$ OR
- $p_i > p_j$ and $q_i < q_j$

It is obvious that if i intersects j with the first condition, then j would also intersect i with the second condition. This counts for redundancy. Therefore, for computing the number of intersections we only consider one of the above conditions. Here, we consider the first one.

We should create an array $I[1..n]$ such that the order of corresponding q of l-th p is $I[l]$. In other words, if $I[l] = k$ and p_i is the point with l-th smallest x in p set then q_i has k-th smallest x in q set.

Obs 1:

Suppose

- $1 \leq l_1 < l_2 \leq n$ and
- p_i has the l_1-th smallest x and
- p_j has the l_2-th smallest x

If $I[l_1] > I[l_2]$then $p_i < p_j$ and $q_i > q_j$: Line Segments i and j intersect.

So the number of intersections is equal to the number of such pairs of (l_1, l_2) Or the number of inversions in array I.

For Building Array I,

1. $Q[1..n] \leftarrow$ sort q set based on x-coordinate.
2. Make $Q'[1..n]$ whereas $Q'[i]$ is equal to the order of q_i in Q
3. $P[1..n] \leftarrow$ sort p set based on x-coordination.
4. Then $\forall 1 \leq l \leq n$ $P[l] \leftarrow Q'[P[l]]$, the index of according q in Q. Now $P[l]$ shows the order of according q of l-th p.

For Counting the number of inversions in Array I, we can change the merge part of merge sort algorithm slightly.The Idea is as follow.

- The number of inversions in a sorted array is zero.

- If we have two equal sized left and right subarray sorted, and we want to merge them into a bigger merged array (as in merge part of merge sort), we move the element from left and right of subarrays to merge arrays by keeping two pointer to left subarray and right subarray (pointers are pointing to the smallest remaining elements in left and right subarray), comparing the numbers which are pointed at, moving the smallest number to the merged array, and advancing it's corresponding pointer in related subarray.
- When an element from right array moves to the merged array, this means that it is smaller than all the remaining elements in the right subarrays, but is located before them. Therefore, we have the inversions as the number of remaining elements in left subarray. Note that the the remaining elements in right subarray are greather than the selected elements, so they don't lead to inverted pairs.
- When an element from left subarray moves to the merged array, this means that that element is smaller than all the remaining elements in the left and right subarrays, but it is located before them in the array, so this elements are not considered inverted.
- So if in merge part of MergeSort algorithm we start counting the number of the remaining elements in the left part whenever we move an element from right part to the merged array(by $O(1)$: by subtracting the index of pointed element from the index of last elements in the left subarray + 1), at the end of MergeSort algorithm we have the number of inversions in array.

Running Time of Algorithm:

1. The running time of making P and Q arrays are $O(n \lg n)$.
2. The running time of making $Q^{'}$ array (by using Q array) is $O(n)$.
3. The running time of making I array (by using P and $Q^{'}$ arrays) is $O(n)$.
4. The running time of finding the number of inversions in Array I by changing the merged part of MergeSort algorithm, as mentioned above, is $O(n \lg n)$.

Hence, the running time of the entire algorithm is $O(n \lg n)$.

Problem 3

$new\_max \leftarrow 0;$
  $potential\_max \leftarrow 0;$
  for $(j = 1 \text{ to } n)$
    if $new\_max + A[i] \geq 0$

$new\_max \leftarrow new\_max + A[i]$
else $new\_max \leftarrow 0$
$potential\_max \leftarrow max\{potential\_max, new\_max\}$
return $potential\_max$

Loop Invarient: at the end of lth iteration of the loop we have to invariant:

- Potential_Max = $\max_{1 \leq i \leq j \leq l} \overset{[}{k}= i]j \sum A[k]$
- New_max: $\begin{cases} the\ maximum\ of\ subsequences\ which\ end\ with\ index\ l & if\ the\ value\ is\ not\ negati \\ 0 & otherwise \end{cases}$

if we prove these invarients then it is proved that at the end of nth iteration we have $\max_{1 \leq i \leq j \leq n} \overset{[}{k}= i]j \sum A[k]$.

Prof of loop invarients:

Base:

If $l = 1$we have two option: either $A[1]$is negative or not. if it is negative the else statement would be run, so the $new\_max = potential\_max$= 0; Otherwise the if statement would be run and $new\_max \leftarrow A[1]$ and then in the end $potential\_max \leftarrow new\_max$.

Hypothesis:

we consider those Invarint hold at the end of $K - 1$iteration .

Proof: those those Invarint hold at the end of Kth iteration :

At the first of kth iteration we have two situations:

1. $potential\_max = new\_max$:in this situation we know that the maximum subsequence ends at $A[K-1]$. . It is obvious if $A[K] > 0$ it must be in the maximum suquence. According to our algorithm in such a situation the new_max will be updated by adding $A[K]$ and since it's value increases our potentioal_max will be updated as well. Otherwise, if $A[K] <= 0$, potential_max wouldn't be updated.
2. $potential\_max <> new\_max$: in this situation we know that the maximum subsequence does not end at $A[K-1]$. If $new\_max + A[K] > potentil\_max$ it means that there is the maximum sequence at the end of $A[1..K]$ considering $A[K]$. According our algorithm, $potential\_max$ will be updated to the new_max value. Otherwise the value of our $potential\_max$ will not change.

Running Time of the Algorithm:

Since in the proposed algorithm, we are visiting each number in the sequence only once, and there is just one iteration on the numbers with constant time, the overal running time of our algorithm is $O(n)$.

Problem 4

For computing the maximum length palindromic substrring of a given string $A[1...n]$, we can see if the first and last characters of string are equal or not. If yes, then we could consider these two characters as the first and last characters of Max Length palindrome and then find the Max Palindromic substring of the remaining string and insert it between the first and last characters $(A[1].MaxPalindromicSubstring(A[2..n-1].A[n]))$. But if the first and last characters of A are not equal, it's impossible for both of them to play role in palindrome (they may be present or not), so we should solve the problem of finding $MaxPalindromicSubstring(A[2..n])$ and $MaxPalindromicSubstring(A[1..n-1]$ and select the one with maximum length:

$$MaxPS[i,j] = \left\{ \begin{array}{c} \left\{ MaxPS\,[i+1, j-1] + 2 \quad if(A[i] = A[j]) \right. \\ \left\{ Max \left\{ \begin{array}{c} MaxPS\,[i+1, j] \\ \\ MaxPS\,[i, j-1] \end{array} \right\} \quad if(A[i] <> A[j]) \right. \end{array} \right\}$$

However, if we make the recursion tree of this recursive function, we can see some repetitive recursion which makes the running time of the function slow. So, it is better to use a matrix $P[n,n]$for memorizing the results of recursions. $P[i,j]$ equals to the length of maximum palindromic substring of $A[i..j]$.

- these values, $P[i,j]$s are only definable for $1 \leq i \leq j \leq n$
- It is obvious that for each substring with the length 1 that substring is itself a palindrome of length 1.

So we initialize this matrix

$$\left\{ \begin{array}{ll} P[i,j] \leftarrow 1 & i = j \\ P[i,j] \leftarrow 0 & o.w. \end{array} \right\}$$

$Palindrome(i...j)$

$if(P[i,j] <> 0)$

$return\ P[i,j];$

$if(A[i] = A[j])$

return $palindrome\ (i+1, j-1) + 2$

$else$

return $max\,\{palindrome(i+1, j), palindrome(i, j-1)\}$

In fact, in this recursive algorithm, the array A fills diagonally. The non-recursive version of this algorithm is:

$NRPalindrome(A[1...n])$

$for(i \leftarrow 1to, n)$

$for(j \leftarrow i, to, n)$

$if(i = j)$

$P[i,j] \leftarrow 1$

$else$

$P[i,j] \leftarrow 0$

$for(l \leftarrow 1to, n-1)$

$for(i \leftarrow 1, to, n-l)$

$j \leftarrow i + l$
$if(A[i] = A[j])$
$P[i,j] \leftarrow P[i-1, j-1] + 2$
$else$
$P[i,j] \leftarrow max\,\{P[i-1,j], P[i,j-1]\}$

$return\ P[1,n]$

Proof:

MaxPalindromicSubstring(A[1..n] return the maximum length of a palindromic substring of the given strin with length n

Base case: $A[1]$.Every single character is palindromic and our algorithm returns 1.

Hypothesis: $\forall k < n$ and MaxPalindromicSubstring(A) returns the length of the maximum palindrome of the given k-length input string.

Claim: MaxPalindromicSubstring(A) returns the length of the maximum palindrome of the given n-length input string.

Condition 1: If the first and last characters of the given string are equal, then these two characters should be considered as the first and last character of palidromic substring. So if we could find the answer of remaining substring (length $k-2$) correctly we can simply add it by 2. This value is calculated by $MaxPalindromicSubstring(A[2..k-1])$, according to the hypothesis.

Condtion 2: If the first and last characters of A are not equal, it's impossible for both of them to play role in palindrome, so the maximum palindrome either consider $A[2..n]$ or $A[1..n-1]$ and according to the hypothesis, the algorithm can correctly return these two sub-problems and the maximum of the answers should be considered as the length of maximum polindromic subsequence.

Complexity of Algorithm:

As in non-recursive algorithm, we have to spend $O(n^2)$ to initial an compute P. We also need an $O(n^2)$ space array p to memorize the intermediate computations. Of course in computing each matrix diagonal we need only to memorize the value of previous diagonal. So, we can reduce the required Space to $O(n)$.