Assignment 1

Stefano Guerra

October 11, 2016

1 Problem 1

Describe a recursive algorithm to reconstruct an arbitrary binary tree, given its preorder and inorder node sequences as input.

First, recall that pre–order and in–order are two ways of traversing a tree. Both are a combination of three (for binary trees) basic actions one can perform at a node:

1. read the information that the node contains.
2. Move to the left child.
3. Move to the right child.

The difference between in–order and pre–order is just about the order in which these operations are performed. The in–order traversal first moves to the left child, then read the content of the node, and finally moves to the right child. The pre-order traversal first read the content of the node, then moves to the left child, and finally to the right child. Each of these traversals can be recorded in an array, where the entries are the nodes of the tree, recorded in the order they are visited. Note also that both these traversals can be implemented recursively:

1. InOrder(T) = InOrder(T_L) + $[r]$ + InOrder(T_R)
2. PreOrder(T) = $[r]$ + PreOrder(T_L) + PreOrder(T_R)

where r is the root of the tree T, T_L (T_R) is the sub–tree of T with root the left (right) child of r, and the plus sign means "array concatenation".

The following observation follows directly from the definition of in–order and pre–order traversal:

a. In a pre–order traversal the first node in the list is always the root node.

b. The in–order traversal has the nodes of the sub–tree rooted at the left child of the root, appearing all before the root node. Similarly the nodes of the sub–tree rooted at the right child of the root, appear after the root node.

An algorithm which reconstruct a three given its in–order and pre–order traversals can be designed as follows:

– The input is given by two lists, both containing all the nodes of the tree under consideration, T. One list contains the nodes in the in–order order $I = I[1, 2, \ldots, n]$, the other in pre–order order $P = P[1, 2, \ldots, n]$. Here n is the number of nodes in the tree we want to reconstruct.
Note also that we can write: $I = I(T) =$ InOrder(T_L) + $[r]$ + InOrder(T_R) and $P = P(T) = [r] +$ PreOrder(T_L) + PreOrder(T_R).

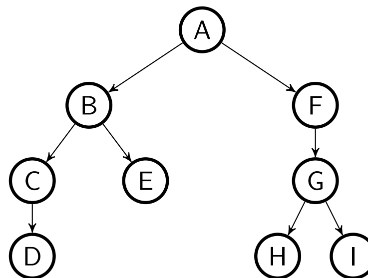
– The root of the tree is the first entry in P.

– Let k be the position of $P[1]$, in the list I.

– Recursively reconstruct the left and right sub–trees linked to the root node by passing as input of the algorithm the following two pairs of lists: the left sub–tree will be reconstructed on the input $I_L = I[1, 2, \ldots, k-1]$, P_L, and the right sub–tree on the input $I_R = I[k+1, k+2, \ldots, n]$, P_R. The list P_L (P_R) is the sublist of P which contains the nodes appearing in I_L (I_R).
  Alternatively we can write: $I_L = \text{InOrder}(T_L)$, $I_R = \text{InOrder}(T_R)$, $P_L = \text{PreOrder}(T_L)$, and $P_R = \text{PreOrder}(T_R)$.

– Finally the base case, if the input to the algorithm consists in a pair of empty lists, then the output is the empty tree.

Let's give an example to clarify how the construction goes.
Consider the following binary tree:



Then its in–order array is $I = [D, C, B, E, A, H, G, I, F]$, and its pre–order array is $P = [A, B, C, D, E, F, G, H, I]$. Since the first node in the pre–order list is A, this means that A is the root node. Then we look in the in–order list I and define $I_L = [D, C, B, E]$, these are the nodes in the three which belong to the sub-tree with root the left child of A. Similarly $I_R = [H, G, I, F]$ contains the nodes of the sub–tree rooted at the right child of A. Finally define $P_L = [B, C, D, E]$ ($P_R = [F, G, H, I]$), since this is the sub–list of P containing the nodes in I_L (I_R). The pairs (I_L, P_L) and (I_R, P_R) are passed as input to the algorithm to reconstruct the sub–trees rooted at the left and right child of A.

It is not clear to me if a proof of correctness should be provided or not. In any case here it is.
The proof proceeds by induction on the number of elements in the lists[1].
As base case we consider inputs of length one. In this case the tree consists only in the root node and there is nothing to show. One can also consider the base case to be given by two empty lists, in which case the algorithm returns the empty tree. Again there is nothing to be checked.
Now assume that the algorithm is correct for input lists of length at most $n-1$, we want to show that the algorithm is correct for input lists of length n. The algorithm picks as root of the tree the first entry of P. As noted above this is always the root of the tree. Then the input to the recursive call are (I_L, P_L) and (I_R, P_R). These lists are all of length at most $n-1$, since the root of the original tree has been removed. By the observation above, the list I_L (I_R) contains the nodes of the sub–tree rooted at the left (right) child of the root tree, therefore by induction hypothesis the algorithm correctly reconstructs this tree.

It is also not clear if we have to analyze the running time, hopefully if something is wrong in my argument, and this is not required for the solution, no points will be deducted.
If the tree T has all $log(n)$ levels full, then since at each level $O(n)$ work is done (we have to search the list I for the root node), then the running time is $O(n \cdot log(n))$. If the tree T has n levels, i.e it is a linked list, then the running time is $O(n^2)$: there are n levels and at each level we still have to search for the root node in I, which has $O(n)$ elements.

[1] Note that we assume that the lists provided as input to the algorithm are legitimate lists, i.e. they have the same length, they contains the same items, and that no element in I_R appears in P before all the elements of I_L have been listed (in P).

2 Problem 2

Suppose you are given two sets of n points, one set $\{p_1, p_2, ..., p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, ..., q_n\}$ on the line $y = 1$. Create a set of n line segments by connecting each point p_i to the corresponding point q_i . Suppose no three line segments intersect in a point.
Describe an algorithm to determine how many pairs of these line segments intersect. Prove that your algorithm is correct, and analyze its running time. For full credit your algorithm should work in $O(n \cdot log(n))$ time. A correct $O(n^2)$ time algorithm receives 10 points.

First note that it is not important that the points are on the lines $y = 0$ and $y = 1$, what matters is the label associated to each point. To start with we assume that the labels of the points p_i are in ascending order, i.e. the label of p_i is i. This allows us to think of the set of points q_i as a permutation of the integers $\{1, 2, \dots, n\}$. We can also assume that the input of the algorithm is an array with entries being in the set $\{1, 2, \dots, n\}$, arranged in the same order as the labels of the q's, i.e. it is a permutation of the set $\{1, 2, \dots, n\}$. From now on when we say permutation we mean permutation on $\{1, 2, \dots, n\}$.
Finally we have to say what the intersecting lines in the problem correspond to in terms of permutations. Given a permutation σ, an *inversion* is given by a pair of integer $i, j \in \{1, 2, \dots, n\}$, $i < j$, such that $\sigma(i) > \sigma(j)$. A simple observation is that if i and j form an inversion for the permutation defined by the q's, then the lines p_i–q_i, p_j–q_j intersect. Therefore the problem of counting the number of intersection points between pairs of lines reduces to the problem of counting inversion in the permutation defined by the q's.
In general one can define an inversion in an array of numbers: two indexes $i < j$ form an inversion if $A[i] > A[j]$.

Here is an algorithm to compute the number of inversions in an array $A[1, 2, \dots, n]$:

InversionCount($A[1, 2, \dots, n]$):
  if $n == 1$:
    return 0
  else:
    $m \longleftarrow \frac{n}{2}$
    $(i_1, L) \longleftarrow$ InversionCount($A[1, 2, \dots, m]$)
    $(i_2, M) \longleftarrow$ InversionCount($A[m+1, m+2, \dots, n]$)
    $(i_3, N) \longleftarrow$ MergeCount(L, M)
    return $(i_1 + i_2 + i_3, N)$

MergeCount(L, M) merges and sorts the arrays L and M, and counts the number of inversions between elements of L and elements of M. The merge and sort part is done as in merge–sort. In addition for each element a in L one checks how many elements in M are smaller than a. Each of this generates an inversion. Since one has to check all elements in L for inversions, which takes linear time, the search in M can be done either with linear search or binary–search. The merging part of the algorithm takes also linear time as we saw in class. In total the whole algorithm's running time is the same as merge–sort, $O(n \cdot log(n))$: the recursion tree has $O(log(n))$ levels and at each level $O(n)$ work is done.

3 Problem 3

Design an algorithm that given an array of integers $A[1..n]$ (that contains at least one positive integer), computes

$$\max_{1 \leq i \leq j \leq n} \sum_{k=i}^{j} A[k]$$

For example, if $A = [31, -41, 59, 26, -53, 58, 97, -93, -23, 84]$, the output must be 187. Prove that your algorithm is correct, and analyze its running time. For full credit your algorithm should work in linear time. For any credit (more than "I dont know") you algorithm should work in $O(n^2)$ time.

A linear algorithm which computes the maximum of the summation over sub-arrays of $A[1, \ldots, n]$ can be designed as follows:

1. Define two variables max and sum and set them equal to zero. These two variables depend on the index $i \in \{1, 2, \ldots, n\}$.
2. Loop over the entries of A, at step i we start with $max[i-1]$ and $sum[i-1]$.
   Then we update both variables as follows:
   (a) $sum[i] = \max\{0, sum[i-1] + A[i]\}$. This is the maximum for the sum of consecutive elements of A which ends at $A[i]$.
   (b) $max[i] = \max\{max[i-1], sum[i]\}$
3. The algorithm returns $max[n]$.

Correctness of the algorithm:
This can be done by induction on the length of the input array.
If the input array has length one, then $A[1]$ is positive, since at least one entry is positive by hypothesis, and so $sum = max = A[1]$. There is nothing to proof in this case.
Now, assume that the algorithm returns the correct answer for input arrays of length at most $n-1$, we want to show that the algorithm is correct also on arrays of length n. By induction hypothesis we know that $sum[n-1]$ is the maximum of the sums of consecutive elements of $A[1, 2, \ldots, n-1]$ ending at $A[n-1]$, while $max[n-1]$ is the maximum value for partial sums of consecutive elements in $A[1, 2 \ldots, n-1]$. Now, if $A[n]$ is positive, $sum[n] = sum[n-1] + A[n]$, and $max[n] = \max\{max[n-1], sum[n]\}$, which will return the maximum for partial sums of consecutive elements in A. If $A[n]$ is negative, or zero, then $sum[n]$ cannot be bigger than $sum[n-1]$, and so $max[n] = max[n-1]$. Then the induction hypothesis tells us that we already have the right answer (since the sub–array we are looking for is contained in $A[1, 2, \ldots, n-1]$).

Running time:
This algorithm runs in $O(n)$ time since it considers each element of the array once and at each step it performs a sum, two comparisons and possibly two variables update, and all these are constant time operations.

Source: wikipedia.

4 Problem 4

A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA. Describe an algorithm to find the length of the longest subsequence of a given string that is also a palindrome. For example, the longest palindrome subsequence of MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM is MHYMRORMYHM, so given that string as input, your algorithm should output the number 11. Prove that your algorithm is correct, and analyze its running time.

We are going to think of a string as an array of characters, therefore the input of our algorithm is an array $A[1, 2, \ldots, n]$. The output is a sub–string of $A[1, 2, \ldots, n]$ which is the longest palindrome sub–string contained in $A[1, 2, \ldots, n]$. If we are interested just in the length of the longest palindrome contained in $A[1, 2, \ldots, n]$ we can modify the algorithm below to return the length of the longest palindrome instead of the string itself.
Then the solution is given by a recursive algorithm:

```
LongestPalindrome(A[1,2,...,n]):
  if n == 0:
    return empty string
  else if n == 1:
    return A[1]
  else:
    if A[1] == A[n]:
      return A[1] + LongestPalindrome(A[2,3,...n-1]) + A[n]
    else:
      return max{ LongestPalindrome(A[1,2,...,n-1]),
                  LongestPalindrome(A[2,3,...,n]) }
```

The algorithm returns the empty string if the input is empty, and the input string if this has only one character (since a single character is a particular instance of a palindrome).
If the input string has more than one character the algorithm compares the first and the last character, if they are the same then it looks recursively for the longest palindrome in the sub–string of A obtained from A by removing the first and the last character. It returns the string found by the recursive call to which are attached the first character, at the beginning, and the last character, at the end.
If the first and the last character are not equal, then it looks recursively for the longest palindrome in the sub–strings $A[1, 2, \ldots, n-1]$ and $A[2, 3, \ldots, n]$, and returns the longest between the two.

Correctness of the algorithm:
the correctness of the algorithm can be proved by induction on the length of the input string.

If the input has length one, than the algorithm returns the input string itself, which is the longest palindrome sub–string.

Next, assume that the algorithm is correct for input strings of length at most $n-1$, we want to show that the algorithm is correct for input strings of length n.
If the first and the last characters are not equal then, by induction hypothesis, the algorithm finds the longest sub–string which is a palindrome for either strings $A[1, 2, \ldots, n-1]$ and $A[2, 3, \ldots, n]$. By taking the longest of the two we are guaranteed to have found the longest palindrome sub–string of $A[1, 2, \ldots, n]$[2].
If the first and the last character of the input string $A[1, 2, \ldots, n]$ are equal, then the induction hypothesis guarantees that the algorithm finds the longest palindrome of $A[2, 3, \ldots, n-1]$, and so $A[1]$ + LongestPalindrome($A[2, 3, \ldots n-1]$) + $A[n]$ is also a palindrome. This will be the longest palindrome sub–string of $A[1, 2, \ldots, n]$. In fact the only thing which can happen is that there is a palindrome sub–string of $A[2, 3, \ldots, n]$ (or $A[1, 2, \ldots, n-1]$)[3]. Then, either this palindrome sub–string ends with $A[n]$ or not. If $A[n]$ is not the last character of the longest palindrome

[2]Note that we cannot make the palindrome sub–string found by the algorithm for $A[1, 2, \ldots, n-1]$ (resp. $A[2, 3, \ldots, n]$) longer by adding the last (resp. first) character, without compromising the fact that that string is palindrome.
[3]We consider $A[2, 3, \ldots, n]$ here, the argument for $A[1, 2, \ldots, n-1]$ is similar with $A[n]$ replaced by $A[1]$

sub–string of $A[2, 3, \ldots, n]$, then the palindrome is actually a sub–string of $A[2, 3, \ldots, n-1]$ and so $A[1] +$ LongestPalindrome($A[2, 3, \ldots n-1]$) $+ A[n]$ is the longest palindrome sub–string of $A[1, 2, \ldots, n]$. On the other hand, if $A[n]$ is the last character of a palindrome sub–string of $A[2, 3, \ldots, n]$, $s[1, 2, \ldots, m]$, then both $s[1]$ and $s[m]$, will be also equal to $A[n]$, and so $s[2, 3, \ldots, m-1]$ is a palindrome sub–string of $A[2, 3, \ldots, n-1]$. Its length will be at most the same as the length of LongestPalindrome($A[2, 3, \ldots n-1]$), by induction hypothesis. This completes the proof.

Running Time:
The algorithm as presented above has an exponential running time since it performs some computations several time. For example in computing the longest palindrome of $A[2, 3, \ldots, n]$ and $A[1, 2, \ldots, n-1]$, the longest palindrome of $A[2, 3, \ldots, n-1]$ will be computed twice. To avoid this problem, and reduce the running time, one can store the results of the new computation and look up the one for the cases already considered. If this is done, each step of the iteration takes $O(n)$ time (the time to write the longest palindrome sub–string, which is the worst case is of order n), and the running time of the algorithm is given by the time needed to fill the storage table, multiplied by the time required to write one cell of the table. In our case the storage table has $O(n^2)$ elements, and so the total running time is $O(n^3)$.
The running time of the algorithm which computes the length of the longest palindrome sub–string is $O(n^2)$, since now the time which takes to fill a cell of the table is constant (the time to write a number).
For completeness here is the algorithm which returns the length of the longest palindrome sub–string instead of the string itself:

```
LongestPalindromeLen(A[1, 2, ..., n]):
  if n == 0:
    return 0
  else if n == 1:
    return 1
  else:
    if A[1] == A[n]:
      return LongestPalindromeLen(A[2, 3, ... n − 1]) + 2
    else:
      return max{ LongestPalindromeLen(A[1, 2, ..., n − 1]),
                  LongestPalindromeLen(A[2, 3, ..., n]) }
```