CS 515: Written Assignment #1

Due on Due Tue, 10/11/16

Amir Nayyeri
6:00-7:20pm, TR

Chen Hang, 932-727-711
Kaibo Liu, 932-976-427

10/08/2016

Contents

Problem 1

Describe a recursive algorithm to reconstruct an arbitrary binary tree, given its preorder and inorder node sequences as input. See Wikipedia (`https://en.wikipedia.org/wiki/Tree_traversal`) if you have forgotten preorder/inorder traversals.

1. **Analysis**

   For a binary tree, the root node is always the place we start to construct. For the preorder and inorder node sequences, the root appears in particular position where we can explore. And root node's every child (if existed) can be treated as a new root, so recursion is a good way to go.

   We use preorder[1..n] and inorder[1..n] as current preorder and inorder node sequences, to build a binary tree. For any preorder node sequences, the first element is always the root node for the current sequence. And this root shall appear in inorder node sequence, dividing the inorder into a left part and a right part(if the root doesn't have child in left or right side, the part will be zero). Then we divide both of them(preorder[1..n] and inorder[1..n]) into two parts, the left parts have the same length and so do the right parts.

2. **Proof**

   It's easy to prove from the definition of preorder and inorder. In current preorder node sequences, the first element is always the root node for the sequence. And this root shall appear in inorder node sequence, dividing the inorder into a left part and a right part(if the root doesn't have child in left or right side, the part will be zero). We can find the right position p, then inorder becomes [1..p-1,root,p+1..n], and preorder becomes [root,2..p,p+1..n]. Here is a simple example in Fig 1. Then we can use recursion for left side (preorder[2..p], inorder[1..p-1]) and right side(preorder[p+1..n], inorder[p+1..n]), respectively.

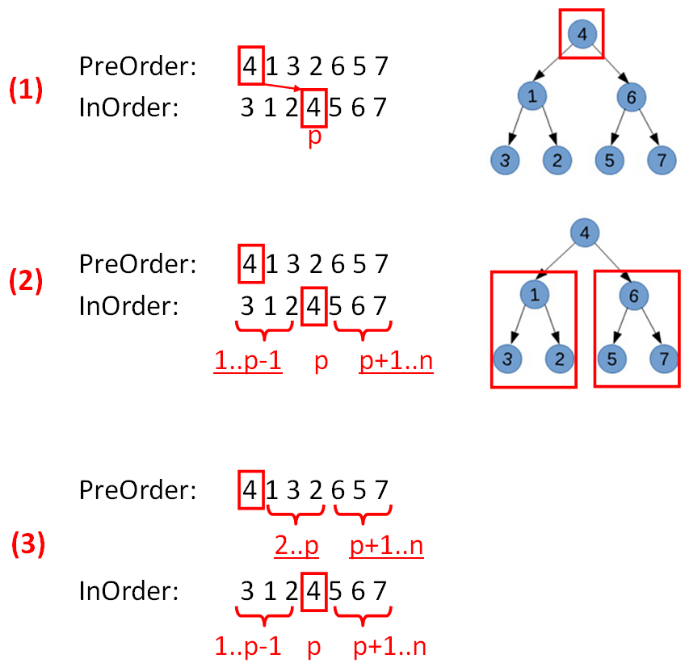

Figure 1: Sample procedure for preorder and inorder

3. **Pseudocode**

Listing 1 shows the pseudocode for the algorithm above.

Listing 1: Binary Tree Reconstruction

```
Node(preorder[1..n],inorder[1..n])
    if (n = 1) return preorder[1]
    else
        p ← Position of preorder[1] in inorder[1..n]
    if (p > 1)
        preorder[1].left ← Node(preorder[2..p],inorder[1..p-1])
    if (p < n)
        preorder[1].right  ← Node(preorder[p+1..n],inorder[p+1..n])
    return preorder[1]
```

4. **Time Complexity**

Every recursion constructs one node and its children, so there are n rounds of recursion. For an arbitrary binary tree, the inorder sequence is unsorted, so finding preorder[1]'s position in inorder needs O(n) runtime for one round of recursion.
Here is a extreme example, if the tree has only left child, shown in Fig.2, every round we need to search to the end of inorder sequence to find the current root. Then the entire runtime becomes O(n^2).
If the tree is balanced, the runtime for position search becomes O($n/2$).

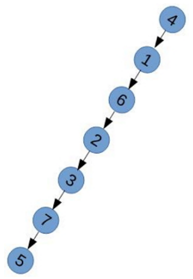

Figure 2: Binary tree only has left child

The conclusion is:
For a balanced tree, T(n)=O($nlogn$)
For a left side tree, T(n)=O(n^2)
For a right side tree, T(n)=O(n)

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 intersec. Prove that your algorithm is correct, and analyze its running time. For full credit your algorithm should work in O($n$logn) time. A correct O(n^2) time algorithm receives 10 points.

Problem 2 continued on next page. . .

1. **Reduce to count array inversions**
   Step 1: Set connections between q[i] and p[i].
   Step 2: Sort array p[] by value of p increasingly. If p[i]=p[j] and $i! = j$, then sort p[i] and p[j] by q decreasingly. If p[i]=p[j] and $i! = j$ and q[i]=q[j], sort p[i] and p[j] increasingly. While swapping p[i] and p[j], just swap q[i] and q[j].
   Step 3: Use inversion function.
   Proof
   There are conditions for intersecting line segments:
   C1: $i! = j$**,** $p[i] < p[j]$ **and** $q[i] >= q[j]$**.**
   C2: $i < j$ **and** $p[i] = p[j]$**.**
   If any (i,j) pairs satisfy C1 or C2, then line segments p[i]q[i] and p[j]q[j] must intersect with each other. Therefore, our task is to count how many pairs of (i,j) satisfying one of these two conditions. This problem is very similar to array inversions. The array inversions problem aims at counting index pairs (i,j) in an array A[1..n] where $i < j$ and $A[i] > A[j]$. According to "Count Inversions in an array - Set 1(Using Merge Sort)" on GeeksForGeeks[1], there is an algorithm solving array inversions problem in O($nlogn$). If we can reduce intersections problem to array inversion problem, it's possible to solve intersections problem in O($nlogn$) time.
   However, in this problem we need to look at two arrays(p and q). Thus we need to simplify the conditions and even revise the array inversions problem.
   First, redefine array inversion.
   Definition 1, an array inversion in array A[1..n] is a pair of indices i and j where $i < j$ **and** $A[i] >= A[j]$**(the not revised version is** $A[i] > A[j]$**).**
   Then, assume we use array q to count inversions.
   To simplify C1, we sort array p by values in p increasingly. Therefore, if $i < j$, then $p[i] <= p[j]$. However, while swapping p[i] and p[j], we must also swap q[i] and q[j]. Only by doing so, the connections always keep correct. (Otherwise, after swapping p[i] and p[j], line segments p[i]q[i] and p[j]q[j] become p[i]q[j] and p[j]q[i] in reality.) Then C1 is reduced to C3 ($i < j$ and $q[i] >= q[j]$).
   To reduce C2 to C3, we can add a rule while sorting array p. This rule is that if p[i] = p[j], then sort p[i] and p[j] by values in array q decreasingly, finally by value of index(i and j) increasingly. By doing so, after sorting array p and resetting array q, if $i < j$ and p[i] = p[j], then $q[i] >= q[j]$. Without doing this part, we should spend other time to count how many pairs of (i,j) satisfying ($q[i] < q[j]$, $i < j$ and p[i] = p[j]).
   However, if ($q[i] = q[j]$, $i < j$ and p[i] = p[j]), it doesn't matter to put p[i] at first or put p[j] at first. We just place smaller index at first.
   By ensuring ($p[j] >= p[i]$ when $i < j$) and ($q[j] <= q[i]$ when ($i < j$ and $p[j] = p[i]$)), it can be concluded that if $i < j$ and $q[i] >= q[j]$, then p[i]q[i] intersects with p[j]q[j].
   Therefore, the problem is reduced to how many pairs of (i,j) in array q satisfying $i < j$ and $q[i] >= q[j]$.

2. **Solve inversions problem in O(nlogn)**

   - Step1: If the size of q is 1, return 0. Otherwise, divide q[1..n] into two halves. Recursively get number of inversions(by Definition 1) of left half and right half, then set data structures for merging two halves.
   - Step2: Before merging two halves, set indices i(from 1 to $n/2$) and j(from $n/2 + 1$ to n) for comparing elements in two halves. Then set an empty array B[n] to restore sorted array. Later, set index k for array B to set element. Finally set variable "merge" as 0 to restore inversions found while merging two halves.
   - Step3: Insert every element to correct place.
     Case 1: If j=n+1 or ($j <= n$, $i <= n/2$ and $q[i] < q[j]$)
     B[k]=q[i], then i pluses one.

Case 2: If i=n/2+1 or ($j <= n$, $i <= n/2$ and $q[i] >= q[j]$)
B[k]=q[j],then j pluses one. Then, "merge" adds (n/2-i+1).
Finally:
k pluses one. If k is less than n, repeat Step3. Otherwise, do Step4.

- Step4: Replace q with B. Return (merge + left result + right result).

Proof

- Base case:
  If n(size of q) is 1, there is no intersecting line segment pairs. Thus, return 0.
- Induction hypothesis:
  Suppose this algorithm can sort array and count inversions in array q[k] where $n-1 >= k >= 1$, then $n >= 2$. In such case, $n/2 <= n-1$. Thus, it also means this algorithm should work for q[n/2].
- Given an array q[n], then divide q[n] into 2 parts(q[1..mid] and q[mid+1..n] and mid=n/2).
  By recursively computing number of inversions in both left half and right half, we can also sort left half and sort right half.(By IH)
  Then merge two halves. First, set indices i(from 1 to mid is meaningful) and j(from mid+1 to n is meaningful). Then set an empty array B(restore the sorted array), an index k(from 1 to n) and variable "merge" as 0(inversions found in merging two halves).
  Before doing merging work, "merge" must be 0 because we didn't find any inversion at beginning.
  To prove correctness of Step3(merging two halves), we can break 2 cases into 4 cases.
  - Case 1: $j <= n$, $i <= n/2$ and $q[i] < q[j]$
    In such case, q[i] is the smallest uninserted element in left half of q. Besides q[j] is the smallest uninserted in right half of q. Additionally, $q[i] < q[j]$. Thus, the smallest uninserted is q[i]. Then q[i] should be inserted into B.
  - Case 2: $j <= n$, $i <= n/2$ and $q[i] >= q[j]$
    In such case, q[i] is the smallest uninserted element in left half of q. Besides q[j] is the smallest uninserted in right half of q. Additionally, $q[i] >= q[j]$. Thus, the smallest uninserted is q[j]. Then q[j] should be inserted into B.
    Besides, $q[i] >= q[j]$ is obviously a inversion. Because left half is sorted, $q[i1] >= q[i]$(where $i <= i1 <= mid$). Combining $q[i1] >= q[i]$(where $i < i1 <= mid$) and $q[i] >= q[j]$, it can be concluded that $q[i1] >= q[j]$(where $i <= i1 <= mid$). Thus for q[j], there are mid-i+1 inversions. So, "merge" should plus mid-i+1.
  - Case 3: j=n+1
    In this case, all elements in right half are added. Only elements in q[i..mid] are not inserted. So the smallest uninserted element is q[i]. Then q[i] should be inserted into B.
  - Case 4: i=n/2+1
    In this case, all elements in left half are added. Only elements in q[j..n] are not inserted. So the smallest uninserted element is q[j]. Then q[j] should be inserted into B. Besides, for q[j], there are no inversions left.
    Besides, "merge" pluses mid-i+1. In this case, mid=n/2+1. In fact, "merge" doesn't change.
    After adding q[i] to B[k], both i and k should plus one so that we can update the smallest uninserted element in left half and the place in B where should be placed in next insert operation. It's similar to add 1 to both j and k after adding q[j] to B[k].

  Because we append B by adding smallest uninserted element in the end of B, B must be sorted. Beside, we find inversions by counting all inversions for each element in right half, thus, all inversions between left half and right half are founded.

Problem 2 continued on next page. . .

Finally, all inversions are composed by inversions just in left half, inversions just in right half, and inversions between left half and right half.
By successfully reducing intersections problem into inversions problem and successfully solving the inversions problem, the whole algorithm is correct.

3. **Pseudocode**

   Listing 2 shows the pseudocode for the algorithm above.

Listing 2: Intersections

```
Intersections(p[1..n],q[1..n])
    sort p[1..n]
          (First, values of p is increasingly. Second, for elements
             whose p values are same, there q values should be
             decreasingly. For elements who have same p values and
             same q values, indices should be increasingly. Swap q[i]
             and q[j] while swapping p[i] and p[j])
    return Count (q[1..n],n)

Count(q[1..n],n)
    if (n = 1)
        return 0
    set empty array B[n]
    mid ← n/2
    left ← count(q[1..mid],mid)
    right ← count (q[mid+1..n],n-mid)
    i ← 1
    j ← mid+1
    merge ← 0
    for (k ← 1 to n)
        if ((j<n+1 && i<mid+1 && q[i] < q[j]) or j>n)
            B[k] ← q[i]
            i++
        else
            merge ← merge+mid-i+1
            B[k] ← q[j]
            j++
    q ← B
    return merge+left+right
```

4. **Time Complexity**

   Sort array p and swap q: O($nlogn$)
   Use merge sort and count intersection pairs: O($nlogn$)
   Total: $c_1 nlogn + c_2 nlogn = (c_1 + c_2)nlogn =$O($nlogn$)

Problem 3

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

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

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 doní know”) your algorithm should work in O(n^2) time.

1. **Analysis**
   Set an array MAX[1..n] where MAX[i]($1 <= i <= n$) means the biggest subsequence sum by using A[i] as the last element. Then set a variable MAXsum to record the best subsequence sum while iterating index i.
   At first, just initialize MAX[1] and MAXsum with A[1]. Then, for all i that $2 <= i <= n$, use equation MAX[i]=max{A[i],MAX[i-1]+A[i]} to update MAX[i]. Then use MAXsum=max{MAXsum,MAX[i]} to update MAXsum. After comparing MAXsum and MAX[n], just return MAXsum.

2. **Proof**

   - Base case:
     In this case, size of a given array is 1. Because MAX[1] means the biggest subsequence sum by using A[1] as the last element(by defition of MAX[i]), then MAXsum=MAX[1]=A[1]. Because this question computes $\max\limits_{1 \le i \le j \le n} \sum_{k=i}^{j} A[k]$, which means empty subsequence is not allowed, so MAXsum cannot be MAX{0,A[1]}.
   - Induction hypothesis:
     Suppose for $k >= 1$, this algorithm can correctly compute (largest subsequence sum of using A[k] as last element, MAX[k]) and (largest subsequence sum of the whole A[1..k], MAXsum).
   - Induction case:
     For array A[1..k+1], there are two ways to compute MAX[k+1](Combine A[k+1] with front elements or only use A[k]). If combine A[k+1] with front elements, because the subsequence should be continous, A[k] must be included. By induction hypothesis, the maximum of subsequence sum by using A[k] as last element is MAX[k]. Thus, MAX[k]+A[k+1] is the largest subsequence sum by combining A[k+1] with front elements. However, MAX[k] might be negative, so A[k+1] might be larger than MAX[k]+A[k+1]. So, MAX[k+1]=$max\{A[k+1], A[k+1]+MAX[k]\}$ is correct. Besides, by induction hypothesis, before comparing MAXsum(referring to A[1..k]) and A[k+1], MAXsum is the largest subsequence sum of A[1..k]. After MAX[k+1] is computed, MAXsum has two cases: use A[k+1] as last element(MAX[k+1]), or not use A[k+1] as last element(MAXsum). Thus, MAXsum for A[1..k+1] should be max{MAXsum for A[1..k], MAX[k+1]}.
     Therefore, the induction case is also proved. To sum up, this algorithm is correct.

3. **Pseudocode**
   Listing 3 shows the pseudocode for the algorithm above.

Listing 3: Max Sum of Successive Subarray

```
FindMaxSum(A[1..n])
    MAXsum ← A[1]
    MAX[1] ← A[1]
    for (i ← 2 to n)
```

Problem 3 continued on next page. . .

```
        MAX[i] ← max{MAX[i-1]+A[i],A[i]}
        MAXsum ← max{MAX[i],MAXsum}
    return MAXsum
```

4. **Time Complexity**
   Compute array MAX and $MAXsum$: O(n)
   Total: O(n)

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.

1. **Analysis**
   LPA[i][j] means length of longest palindrome in str[i..j].
   There are four rules to compute LPA[i][j].
   LPA[i][j]=if i==j then 1
   LPA[i][j]=if ($i < j$ and str[i]==str[j]) then LPA[i+1][j-1]+2
   LPA[i][j]=if ($i < j$ and str[i]!=str[j]) then max(LPA[i+1][j],LPA[i][j-1])
   LPA[i][j]=if $i > j$ then 0 We want the value of LPA[1][n].

2. **Proof**
   Definition of palindrome.
   If str[1..n] is a palindrome, str[i]=str[n+1-i] and $1 <= i <= n-1$.
   LP means longest palindrome. Value of LPA[i][j] means the longest length of palindrome inside str[i..j].
   If LPA[i][j]=0, it means the unit LPA[i][j] is not computed and even not meaningful.
   LPAss is an assistant function to compute the longest palindrome length inside str[i..j].
   First of all, if a string contains only one char, it forms a palindrome whose length is 1. Thus, if i==j, LPA[i][j]=1.
   Other elements should be computed by assistant function.
   In this algorithm, i means start and j means end. If $i > j$, it's meaningless. Thus, just set LPA[i][j] as 0. This value cannot be -1, which will be proved in proving rule for (str[i]==str[j] and i!=j).
   Suppose $j > i$, in this case, length of str[i..j] is at least 2.

   Proof of rule for $str[i] == str[j]$ and $i < j$

   - Base case:
     Suppose str[2] is 'aa'. LPA[1][2] should be LPA[2][1]+2=0+2=2.
     If LPA[2][1] is -1, this algorithm is wrong.
     Suppose str[3] is 'aba'. LPA[1][3]=2+LPA[2][2]=2+1=3.
   - Induction case:
     Assume that the algorithm works for str[1..n] and the longest palindrome in str[1..n] is pl[1..k].
     If we add a char x both at head and tail of str[1..n], then we gain a new string str1[1..n+2]. We also add x at head and tail of pl[1..k] to form pl1[1..k+2].Therefore, str1[2..n+1]=str[1..n] and pl1[2..k+1]=pl[1..k]. Since pl is a palindrome, pl1 satisfies pl1[i]=pl1[(k+2)+1-i](2 $<= i <=$

$k + 1)$), combine with pl1[1]=pl1[k+2], pl1 satisfies pl1[i]=pl1[(k+2)+1-i]($1 <= i <= k + 2$). Thus pl1 is a palindrome. Though pl1 is a palindrome, it's still not sure whether it's the longest palindrome.

Assume pl1 is not the longest palindrome inside str1[1..n+2]. Then, there should be another palindrome pl2 inside str1[1..n+2] whose length is larger than pl1. This means pl2 has at least 3 more elements than pl.

- Case 1(Containing top x and last x):
  If pl2 has two xs(x is at both top and tail char of pl1) at top and tail, pl2 can delete two xs(at top and tail of pl2) to form another palindrome inside str[1..n] whose length is larger than pl. It's contradict to pl is the largest palindrome in str[1..n].
- Case 2(Containing no top x nor tail x):
  If pl2 is inside str[1..n], its best case is just as long as pl. In fact, pl2 is smaller than pl1. It's contradict to pl2 is the largest palindrome in str2[1..n+2].
- Case 3(Containing only one top x or tail x):
  If pl2 uses str1[1](which is x) as head but not use str1[n+2](which is also x) as end, then if there is no x in str1[2..n+1], length pl2 should only be 1. On the other hand, if there is at least an x inside str1[2..n+1], using anyone x in str1[1..n+2]. Set this x point as str1[q]. Thus, there must be a palindrome inside str1[1..q] called pl6. pl2 equals to str1[1] appending pl6 then appending str1[q]. Then pl6 must locate in str1[2..n+1]. Since pl is the longest palindrome inside str1[2..n+1] , then pl6 would not be longer than pl. Thus, pl2(pl6+2) is not longer than pl1(pl+2).
  If last element in pl2 is x, this situation is the same as pl2[1] is x.
  Therefore, there is no pl2 whose length is longer than pl1.
  Thus, if pl is the longest palindrome in str[1..n], pl1 should be the longest palindrome in str1[1..n+2].
  In this proof, we can know that if the rule works for n, it works for n+2.

By combining the two correct cases (size of 2 and size of 3), it works for size of both even number and odd number.Thus, the rule is correct.

Proof of rule for ($str[i]! = str[j]$ and $i < j$)
The last rule is that when ($str[i]! = str[j]$ and $i < j$), the length of longest palindrome in str[i..j] is the maximum length of LPAss(str[i+1..j],i+1,j) or LPAss(str[i..j-1],i,j-1).
If a str[2] is 'ab', LPA[1][2]=max(LPA[1][1],LPA[2][2])=max(1,1). This base case is obviously true.
If there are 3 chars in string str[1..3] whose value is "aab", obviously, LPAss("aa",1,2)=2, LPAss("ab",2,3)=1. The biggest length of palindrome in "aab" should be 2. This is another base case of rule for ($str[i]! = str[j]$ and $i < j$).
Suppose there is a string str[1..n] whose first char is different from its last char. Then we suppose the whole algorithm works for all strings whose length is less than or equals to n-1($n >= 2$). Suppose the longest palindrome in str[1..n-1] is pl3. Then suppose the longest palindrome in str[2..n] is pl4. Finally, suppose pl5 is the longest palindrome in str[1..n].
Then because str[1]!=str[n], then pl5 should not use str[1] as top and str[n] as end. Thus pl5 is either inside str[1..n-1] or str[2..n]. If pl5 is inside str[1..n-1], pl5 is pl3. Otherwise, pl5 is inside str[2..n], which means pl5 is pl4. Thus, pl5 should get maximum of pl4 and pl3 to ensure that pl5 is the longest palindrome inside str[1..n].
Finally, all rules are proved.

3. **Pseudocode**

   Listing 4 shows the pseudocode for the algorithm above.

Listing 4: Longest Palindrome

```
   LPA[][](declare but not implement this empty 2D array)

   LP(str[1..n],n)
       LPA[n][n]
5      set all elements in LPA as 0 to show they are not computed
       for (k ← 1 to n)
           LPA[k][k] ← 1
       return LPAss(str[1..n],1,n)

10 LPAss(str[1..n],i,j)
       if (LPA[i][j] != 0)
           return LPA[i][j]
       if (i > j)
           return 0
15     if (i < j)
           if (str[i] = str[j])
           LPA[i][j] ← LPAss(str[1..n],i+1,j-1)+2
           return LPA[i][j]
       else
20         LPA[i][j] ← max{LPAss(str[1..n],i+1,j),LPAss(str[i..j-1],i,j
               -1)}
           return LPA[i][j]
```

4. **Time Complexity**

   Runtime:
   Initialize LPA: O(n^2)
   LPAss: LPAss(str[i..j],i,j) is used to calculate LPA[i][j], which means the length of the longest palindrome inside str[i][j]. Because LPA can store the temperate result, thus many recursive calls consume only O(1) time. It's not reasonable to say "$T(n) = 2T(n-1) + O(1)$" by just looking at line 16 of pseudocode 4. At most $(1+n)n/2$ elements need to be computed(because we only compute LPA[i][j] where $1 <= i <= j <= n$). To compute each element, they have at most 4 steps(find rule, take at most 2 numbers from the bottum level in O(1) and compute). Averagely, compute each block takes O(1) time. Therefore, the runtime for LPAss is O(n^2).
   Total: $c_1n^2 + c_2n^2 = (c_1 + c_2)n^2$=O($n^2$)

Citation

[1] Source 1:
Title: Count Inversions in an array — Set 1 (Using Merge Sort)
Website: http://www.geeksforgeeks.org/counting-inversions/
Author: Not mentioned in the website