Homework 2

Zhehao Wang 404380075 (Dis 1B)

Apr 10, 2016

1 Number of inversions remains unchanged for any permutation

Proof by induction on the number of position swaps in the permutation: we denote the original lists as $A = a_1...a_n$ and $B = b_1...b_n$. Each permutation consists of a number of position swaps for songs in both list A and B. We call a pair (a_i, a_j) *flipped* if it used to be (a_i, a_j) before the permutation, and becomes (a_j, a_i) after the permutation, and the songs a_i, a_j to be *involved* in the flip.

Base case: consider the case where only one pair of songs in both A and B have their positions swapped. Denote the pair as (a_i, a_j) in the original list A, and (b_m, b_n) in the original list B, we have $i < j$, $m < n$, $a_i = b_m$ and $a_j = b_n$.

The *flipped* pairs in A caused by this permutation include: (a_i, a_j), (a_p, a_j) and (a_i, a_p), where $i < p < j$. Similarly, *flipped* pair in B include: (b_m, b_n), (b_q, b_n) and (b_m, b_q), where $m < q < n$. Since $a_i = b_m$ and $a_j = b_n$, number of inversions is not changed by A and B both having the (a_i, a_j), (b_m, b_n) flips. Thus we consider each a_p and b_q *involved* in the flip, and the total number of inversions does not change if each *involved* a_p and b_q do not cause changes in the number of inversions. Case analysis on the position of each b_l in B where each $b_l = a_p$.

- If $l < m$, then list B used to have (b_l, b_m) and (b_l, b_n), list A used to have (a_i, a_p) and (a_p, a_j), number of inversions used to be 1. After the permutation, B's pairs involving b_l, b_m, b_n are not flipped, and A has (a_p, a_i), (a_j, a_p). Number of inversions is still 1.
- If $l > n$, the case is similar with above. The number of inversions before and after the permutation are both 1.
- If $m < l < n$, then B used to have (b_m, b_l) and (b_l, b_n), A used to have (a_i, a_p) and (a_p, a_j), and number of inversions used to be 0. After the permutation, B has (b_l, b_m) and (b_n, b_l), A has (a_p, a_i) and (a_j, a_p). The number of inversions is still 0.

Similar case analysis can be done for each a_k in list A where each $a_k = b_q$. Thus we have the number of inversions does not change when only one pair is swapped in the permutation.

Induction case: assume that the conclusion holds for any permutation involving n position swaps. For any permutation involving $n + 1$ position swaps, by the induction hypothesis, we know that the conclusion holds for its sub-permutation with one pair excluded. By applying the analysis of the base case on the results of the sub-permutation, we know that the conclusion holds for any permutations involving $n + 1$ position swaps as well.

2 Number of intersection and inversions

(a)

Proof: an inversion if defined by a pair (i, j) such that q_i is before q_j in list q, but p_j is before p_i in list p.

For each pair (i, j), consider the sequences of p_i, p_j and q_i, q_j, and the lines $(p_i, q_i), (p_j, q_j)$.

- If there's an intersection between the two lines, the sequences of p_i, p_j and q_i, q_j have to be different in list q and p, as illustrated in figure 1. Thus for each intersection, there is at least one corresponding inversion. Number of inversions $\geq$ Number of intersections.
- If the sequences of p_i, p_j and q_i, q_j are different in list q and p, there has to be an intersection between the lines $(p_i, q_i)$and (p_j, q_j), as illustrated in figure 1. Thus for each inversion, there is at least one corresponding intersection. Number of intersections $\geq$ Number of inversions.

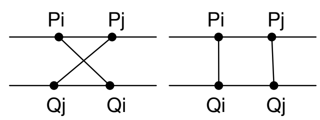

Figure 1: Two-node cases of inversion vs intersection

Summarizing the two cases, we have the number of intersections = number of inversions for each pair (i, j), thus the theorem holds.

(b)

Given the conclusion in 1, permutations do not cause the number of inversions to change, this algorithm uses one list as the standard, takes in the other list, and calculates its number of inversions when compared against the standard.

Time complexity: this algorithm adds a constant-time inversion count to a recursive merge sort, thus the complexity is $O(n \log n)$

Correctness:

3 Celebrity iterative

4 Diameter of tree

(a)

Define the **height** of a rooted directed tree as the number of edges on the longest path from the root to a leaf. Algorithm is given in Alg 2.

This recursive algorithm takes in the root of a tree and produces the height of the tree, by each time removing the root and finding the maximum height among all resulting sub trees. The diameter of the tree would be the sum of the heights of two highest subtrees. Initial call to the algorithm should look like $findHeightOrDiameter(root, false, nil)$. This algorithm is $O(n)$, where n is the number of nodes in the tree, because each node in the tree will be visited exactly once.

(b)

The iterative version of the algorithm is given in Alg 3.

This algorithm is $O(n)$, where n is the number of nodes in the tree, because each node in the tree will be visited exactly once.

Algorithm 1 Number of inversions for one list against a permutated standard

```
function numberOfInversions(array)
    n ← len(array)
    if n < 2 then
        return 0
    l1 ← numberOfInversions(array[0]..array[n/2])
    l2 ← numberOfInversions(array[n/2 + 1]..array[n])
    return l1 + l2 + countInversions(array, array[0]..array[n/2], array[n/2 + 1]..array[n])
function countInversions(original,l1,l2)
    inversions ← 0
    n ← len(l1)
    pos ← 0
    while l1.hasNext() or l2.hasNext() do
        if l1.hasNext() = false then
            original[pos] ← l2.next()
        else if l2.hasNext() = false then
            original[pos] ← l1.next()
        else if l1.next() < l2.next() then
            original[pos] ← l1.next()
        else
            original[pos] ← l2.next()
            i ← n − l2.next()'s position in l1
            inversions ← inversions + i
        pos ← pos + 1
    return inversions
```

Algorithm 2 Diameter of a rooted directed tree's underlying undirected tree, recursive

```
function findHeightOrDiameter(root, findHeight, prevRoot)
    if degree(root) = 1 then
        return 0
    heights ← []
    for each {n|n ∈ V, (n, root) ∈ E, n ≠ prevRoot} do
        heights.push(1 + findHeightOrDiameter(n, true, root))
    if findHeight then
        return max(heights)
    else
        return max(heights) + 2^nd highest(heights)
```

Algorithm 3 Diameter of a rooted directed tree's underlying undirected tree, iterative

```
function findHeightOrDiameter(root)
    queue ← [root]
    height0 ← 0
    height1 ← 0
    while True do
        nodeCount ← queue.size()
        if nodeCount = 0 then
            return height0 + height1
        height ← height + 1
        while nodeCount > 0 do
            r ← queue.dequeue()
            r.visited ← true
            if degree(r) = 1 then
                if height > height0 then
                    height1 ← height0
                    height0 ← height
                else if height > height1 then
                    height1 ← height
            else
                for each {n|n ∈ V, (n, r) ∈ E, n.visited = false} do
                    queue.enqueue(n)
            nodeCount ← nodeCount − 1
```