Homework 2

Matt Forbes

October 21, 2010

1 Problem One

a)

found is true. For this case to occur, we had to be within some iteration of the loop. In order to be in an interation, the expression (low < high) AND !found must be true. Therefore, low does not equal high, because low must be less than high. The first if statement within the loop body checks if $A[low] + a[high] == x$ is true, and will set found to true if that is the case. Therefore, if found is true, then low and high are indices of A whose elements' sum is x.

b)

low $\geq$ high. I will prove the loop invariant provided to show that if low $\geq$ high, there does not exist two disting elements in **A** that sum to x.

<u>Basis</u>

At the start of the first iteration,
$low = 0, high = n - 1,$
$S = \{A[0] \dots A[-1]\} \cup \{A[n] \dots A[n-1]\} = \{\}$
So there are no distinct pairs in S sum to x. The L.I. holds before we enter the loop.

<u>Maintenance</u>

Assuming the L.I. held for all iterations up to this iteration j, then:

- Right before this loop started, there was no distinct pair of elements in the set $S = \{A[0] \dots A[low-1]\} \cup \{A[high+1] \dots A[n-1]\}$ whose sum = x.
- During this iteration, $A[low] + a[high]$ could be:
  equal to x: We found a distinct pair that sums x, done.

  less than x: low is incremented, and thus $A[low]$ is 'added' to S in the next iteration, in which case the L.I. would still hold for these reasons: $A[low] + A[i], i = 0 \dots low - 1$ will always be less than x, and $A[low] + a[j], j =$

$high + 1 \dots n - 1$ will always be greater than x. Therefore there will be no pair in S whose sum is exactly x.

greater than x: high is decremented, and this $A[high]$ is 'added' to S in the next iteration, in which case the L.I. would still hold for these reasons: $A[high] + A[j], j = high + 1 \dots n - 1$ will always be greater than x, and $A[i] + A[high], i = 0 \dots low$ will always be less than x. Therefore there will be no pair in S whose sum is exactly x.

- The L.I. will always hold after this iteration, granted that it held up to this point for the reasons listed above.

Termination

- After each iteration, either low is incremented or high is decremented, so they have to converge at one point as long as **found** is never set to true, so the loop is guaranteed to terminate.
- According to the L.I. at the end of the last iteration, there is no distinct pair of elements in $S = \{A[0] \dots A[low - 1]\} \cup \{A[high + 1] \dots A[n - 1]\}$ whose sum is x. Well at the end of the loop, S is equal to all of the elements in **A**. Therefore, there is no distinct pair of elements in **A** whose sum is equal to x.

2 Problem Two

a)

(1,5) (2,5) (3,4) (3,5) (4,5)

b)

The array [n . . . 1] has the most inversions.
A[1] is greater than n-1 elements on its right, so it has n-1 inverions.
A[2] is greater than n-2 elements on its right, so it has n-2 inversion.
. . .
A[n-1] is grater than n element on its right so it has 1 inversion.
So, the number of inversions: $\sum_{i=1}^{n-1} = \dfrac{n(n-1)}{2}$

c)

The number of writes insertion sort performs is equal to the number of inversions there were in the intial form of array. Each inversion in the array implies a shift of an element to the left.

d)

```
// 'merge' function from text modified to not use infinity
// and keep track of inversions by counting the number of positions
// each element 'jumps' over (i.e. number of inversions)
merge_and_count(A, p, q, r):
    let L = new Array(q - p + 1),
        R = new Array(r-q),
        inversions = 0,
        i = 1, j = 1
    L = A[1..q]     //copy first half into L
    R = A[q+1..r]     //copy last half into R
    for k = p to r:
        if j > r:
            switch_left = true
        else if i > q:
            switch_left = false
        else:
            switch_left = L[i] < R[j]

        if switch_left:
            A[k] = R[j]
            i++
        else:
            A[k] = R[j]
            j++
            // (q - i + 1) number of positions this element jumps left
            inversions += (q - i + 1)
    return inversions

count_inversions(A, p, r):
    inversions = 0
    if p < r:
        q = (p+r) / 2
        inversions += count_inversions(A, p, q)
        inversions += count_inversions(A, q+1, r)
        inversions += merge_and_count(A, p, q, r)
    return inversions
```

Explanation

So say we are using the above code, and are at some level of recursion. We have an array that has two 'sorted subarrays' L and R, within it, where L is the first half and R is the last half. When we 'merge' these arrays, we comapre the smallest element of each subarray and place the minimum of the two in the next slot of the whole array. If the minimum element comes from R, then it is essentially 'jumping' over all of the elements still currently in L. Each 'jump' is essentially an inversion because the elements that were jumped were all greater than it. So if we count the number of jumps we are actually the number of inversions.

3 Problem Three

Algorithm Description

The algorithm's correctness stems from the following point being true: If we take the median card from both friends' hand, then the absolute median's value must be within the value of these two cards.

If we take the median card from both hands, and call card **a** the smaller of the two, and **b** the greater. There are at most $n - \frac{n}{2}$ cards greater than **a** in its own hand. In **b**'s hand, there are at most $n - \frac{n}{2} - 1$ cards greater than **a**, because **a** > **b**. In total, there are at most $n - \frac{n}{2} + n - \frac{n}{2} - 1 = 2n - n - 1 = n - 1$ cards greater than **a**. Therefore, **a** must be greater than or equal to the absolute median of the 2n cards.

Likewise for **b**.

In the algorithm, **L** and **R** represent the two hands of cards. **lbound** and **rbound** are the minimum and maximum values of the absolute median that we know up to that point. **cuts** is the number of cards that we have ruled out and know are less than the absolute median.

Pseudo Code

```
median(L, R, n):
    return median_helper(L, R, 0, 0, 0, n)

median_helper(L, R, lbound, rbound, cuts, n):
    if L is 'empty' or R is 'empty':
        'remove' (n-cuts-1) cards from the non-empty hand
        return first 'remaining' card in the non-empty hand    # (+1)
    if both L and R have exactly one card:
        ask for both remaining cards    # (+2)
        if cuts = n-1:
            return minimum of the 'remaining' cards
        else:
            return maximum of the 'remaining' cards

    ask for median card from the 'remaining' set of both L and R    # (+2)
    set min = smaller of the two cards that were just asked for,
        max = greater of the two cards that were just asked for,
        min-hand = the hand that holds min
        max-hand = the hand that holds max
    if lbound < min < rbound:
        'remove' all cards in min-hand that are less than min
        set lbound = min,
            cuts = cuts + number of cards just 'removed'
    else:
        if min < lbound:
            'remove' min and all cards in min-hand lower than min from min-
               hand
            cuts = cuts + number of cards just 'removed'
        else:
            'remove' min and all cards in min-hand greater than min
    if lbound < max < rbound:
        'remove' all cards in max-hand that are greater than max
    else:
        if max < lbound:
            'remove' max and all cards in max-hand that are less than max
            cuts = cuts + number of cards 'removed'
        else:
            'remove' max and all cards in max-hand that are greatre than
               max

    median_helper(L, R, lbound, rbound, cuts, n)
```

Running time of algorithm

Let T(n) be the running time of this algorithm, where n is the total number of cards we are searching through.

$$T(n) = \left\{ \begin{array}{cc} T(\frac{n}{2}) + O(1), & \text{for n} > 2 \\ \text{O}(1), & \text{for n} \leq 2 \end{array} \right\}$$

An upper bound guess for T is $T(n) \leq c\log_2(n) + O(1)$

$$\begin{aligned} T(n) &\leq c\log_2(\frac{n}{2}) + O(1) \\ &\leq c\log_2(n) - \log_2(2) + O(1) \\ &= c\log_2(n) + O(1) \end{aligned}$$

So, $T(n)$ is $O(\log_2(n))$.