CMPS 102 Solutions to Homework 1

Lindsay Brown, lbrown@soe.ucsc.edu

September 29, 2005

Problem 1. 1.2-2 p. 13
For inputs of size n insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

We want to find where $8n^2 = 64n \lg n$, this reduces to $n = 8 \lg n$.

n	$8 \lg n$
1	0
2	8
4	16
8	24
16	32
32	40
64	48

We can see from this table that insertion sort beats merge sort for all values of n below 32, and for values of n above 64 merge sort beats insertion sort. Evaluating the functions for n in the interval $(32, 64)$, we find that for all values of $n < 44$ insertion sort beats merge sort. *See figure 1.*

Problem 2. Induction

Claim: For integers $n \geq 1$

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{(n+1)}$$

Proof by induction on n, the upper limit of the sum.

Base case: Let $n = 1$.

$$\sum_{i=1}^{1} \frac{1}{i(i+1)} = \frac{1}{1(1+1)} = \frac{1}{2}$$

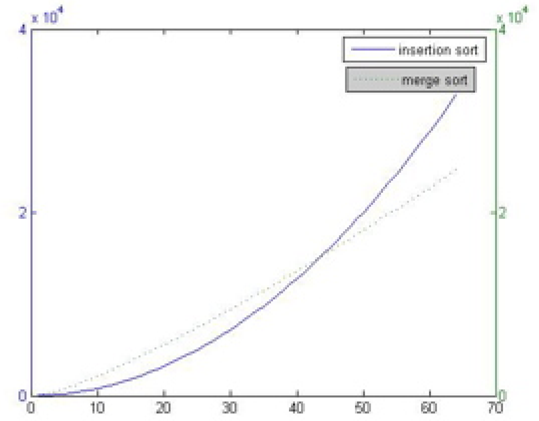

Figure 1: Merge sort and insertion sort running times.

and

$$\frac{n}{n+1} = \frac{1}{2}$$

Inductive step: Suppose it is true for all k such that $1 \le k \le n-1$. We want to show it is true for n. Let $k = n-1$. Then using the inductive hypothesis:

$$\begin{aligned}\sum_{i=1}^{n} \frac{1}{i(i+1)} &= \sum_{i=1}^{n-1} \frac{1}{i(i+1)} + \frac{1}{n(n+1)} \\ &= \frac{n-1}{n} + \frac{1}{n(n+1)} \\ &= \frac{(n+1)(n-1)+1}{n(n+1)} = \frac{n^2}{n(n+1)} = \frac{n}{n+1}\end{aligned}$$

as desired.

Problem 3. 2.3-7 p. 37
Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

There is more than one algorithm that satisfies this problem. You should have described an algorithm and done the necessary proof for correctness, as well as proved the asymptotic running time. If your algorithm utilizes an algorithm covered in the text, you may assume it is correct and use the results of its run-time analysis.

First run merge sort on S to get the sorted array $S' = s'_1, s'_2, ..., s'_n$, where $s'_i \leq s'_{i+1}$ for all i. As proved in Cormen, this runs in $\Theta(n \lg n)$-time.

```
SUM-X($S, x$)
  $MERGE-SORT(S)$
  $i \leftarrow 1$
  $j \leftarrow n$
  **while** $i < j$
      **do if** $S[i] + S[j] < x$
              **then** $i \leftarrow i + 1$
              **else**  **if** $S[i] + S[j] > x$
                          **then** $j \leftarrow j - 1$
                          **else**  **return** $S[i], S[j]$
  **if** $i = j$
     **then return** NIL
```

Proof of correctness: Assume that the set S is sorted, so that $s_1 < s_2 < ... < s_n$. For $1 \leq p < q \leq n$, call (p, q) an x pair if $s_p + s_q = x$. There may be many such x pairs. Initially $i = 1$ and $j = n$. SUM-X maintains a pair (i, j) and either increases i or decreases j until it finds an x pair, or $i = j$. If $s_i + s_j > x$ then j is decremented, and if $s_i + s_j < x$ then i is incremented.

If there are no x pairs in S then SUM-X terminates without finding such a pair. Assume there is at least one x pair. Let (i, j) be the first loop at which either the left index i or the right index j agrees with some x pair. Without loss of generality suppose that $j = q$ and $i \leq p$, thus $s_i + s_q \leq x$. This means that $s_i + s_q = x$ and $i = p$, or $s_i + s_q < x$ and i is increased until $i = p$. Hence, SUM-X finds an x pair.

The worst-case run time of SUM-X is $O(n \lg n)$. After the set is sorted, the procedure to find the two elements that sum to x has a worst-case time of $O(n)$.

A second solution is to run merge sort to get the sorted array S'. Then for each element in S' compute $d_i = s'_i - x$ and do a binary search for d_i in S'. In the worst-case we search for n elements using binary search. The worst-case run time analysis of binary search is $O(\lg n)$. Therefore, the algorithm is $\Theta(n \lg n)$.

Problem 4. 2.2-2 p. 38 Bubble Sort
a. To prove BUBBLESORT is correct you must proved that it terminates, that the output is a permutation of the input array in sorted order, and that the loop invariants for both **for** loops hold.

b. **Loop invariant:** At the start of each iteration of the for loop in lines 2-4, there are no elements of lesser value to the subarray to the right of $A[j]$, that is $A[j] \leq A[k]$ for all k where $j + 1 \leq k \leq n$.
Initialization: When $j = length[A]$ there are no higher indexed elements.
Maintenance: If $A[j-1] \geq A[j]$ then the elements are exchanged, otherwise the elements remain unchanged. The counter j is only decremented after the

comparison.
Termination: The loop ends when $j = i + 1$, thus $A[i]$ and $A[i + 1]$ are compared and exchanged if necessary.

c. **Loop invariant:** At the start of each iteration of the for loop in lines 1-4 the elements in the subarray $A[1]...A[i - 1]$ are in sorted order.
Initialization: Prior to the first iteration $i = 1$, so the subarray $A[1]...A[i - 1]$ is empty.
Maintenance: The smallest element in the subarray $A[i]...A[length[A]]$ is moved to $A[i]$, and the elements in the subarray $A[1]...A[i - 1]$ are unchanged, therefore the loop invariant is maintained.
Termination: The loop ends when $i = length[A]$, thus $A[1] \leq A[2] \leq ... \leq A[length[A]]$.
d. The worst case running time for bubble sort occurs if the input array is in reverse sorted order. In this case the algorithm must compare and swap

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

elements. The worst case for insertion sort is also when the input array is in reverse sorted order. Again, the algorithm must do $(n-1)n/2$ comparisons and swaps.

Problem 5. 2.4 a-c p. 39 Inversions
a. List the five inversions of the array (2, 3, 8, 6, 1).
(2,1), (3,1), (8,1), (6, 1), (8,6)

b. What array with elements from the set $\{1, 2, ..., n\}$ has the most inversions and how many inversions does it have?

The reverse sorted array $n, n - 1, ..., 2, 1$ has the most inversions. There are $n - 1$ inversions with the element 1, $n - 2$ inversions with the element 2, ..., and 1 inversion with the element $n - 1$. Therefore, the number of inversions is the sum:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

c. Claim: The number of swaps done executing insertion sort on array A is equal to the the number of inversions in A, and the number of comparisons done executing insertion sort is also equal to the number of inversions in A.

Proof: Let A be the input array. $A = a_1, a_2, ..., a_n$. Suppose the first $i - 1$ elements are in sorted order and we are at the ith iteration of the for loop, so $key = a_i$.
Let (p, q) be an inversion pair such that $a_p < a_q$ and $p > q$. For all inversion pairs (i, q) we can say that $q < i$. Suppose there are $i-k$ such elements q. If (i, q)

was an inversion before running insertion sort on the subarray $A[1], ..., A[i-1]$, then (i, q) is still an inversion of the resulting array, because the first $i-1$ loops only modify the position of the first $i-1$ elements.
The i^{th} loop will insert $key = a_i$ into the sorted subarray $A'[1], ..., A'[i]$ at position k so that $a'_1 \leq ... \leq a'_k = key \leq ... \leq a'_i$, for $i \leq k \leq i$. The key will be compared to $(i-1)-(k-1) = i-k$ elements. All elements $a_k, ..., a_{i-1}$ will be shifted one position. Therefore, the number of elements that change position in the array when inserting element i is $(i-1)-k+1 = i-k$ (we add 1 because the key is also moved).
We assumed the number of inversions for element i was $i-k$ and showed that the number of comparisons made and the number of elements that moved to insert element i was $i-k$. Therefore, the running time of insertion sort is proportional to the number of inversions in A.

Extra Credit. 2.4 d p. 40
Design an algorithm such that:
Input: Array A, $length[A] = n$
Output: Number of inversions of A
Worst-Case running time: $\Theta(n \lg n)$
Modify the combine procedure of merge sort (line 12-17 p. 29). Each time we combine two arrays we add to the total inversion count the number of inversions that have been fixed when combining the two subarrays. When an element from the right array is added to A, add to the inversion count the number of elements remaining in the left array. Add this number of inversions to the total number of inversions of A. Merge sort runs in time $\Theta(nlogn)$, and we have only increased the number of operations by a constant factor.

Another way to count the number of inversions of an n element array A is to build a graph G with n vertices. Number the vertices $1, ..., n$ to represent the indices of A. For every inversion pair (p, q) in A, such that $a_p < a_q$ and $p > q$ add an edge from vertex p to vertex q in G. Then the total number of inversions is the number of edges.
Use a recursive algorithm COUNT-EDGES to count the edges of G. The input is a graph with n vertices. Divide the n vertices into two sets $(1, ..., n/2)$ and $(n/2+1, ..., n)$. Count the edges by recursively calling COUNT-EDGES on subproblems of size $n/2$. Count the number of edges between the edges in the two sets and combine the sets of vertices. Add the edge count of the subarray to the total edge count of G.