CPSC 221 - Assignment #2

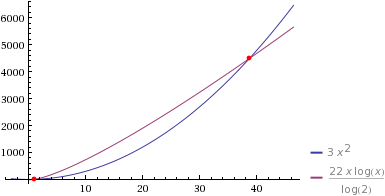
1. (a) Insertion sort has Θ(n2) so for 100000 values, it takes t seconds. So for double that value, 200000, it would take **4t seconds** as the input is doubled, 22.

(b) Merge sort Θ(nlgn) so for 100000 values, it takes t seconds. So for double that value, 200000, it would take **2t seconds** as the input is doubled, 2lg2 = 2.

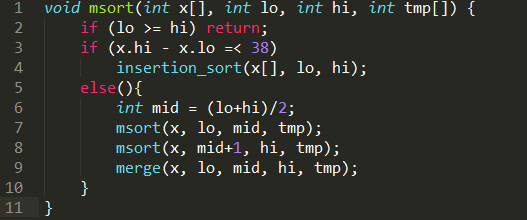
(c) Average runtime of insertion sort -> Ti(n) = 3n2

Average runtime of merge sort -> Ti(n) = 22nlgn

Using wolfram alpha to find when the two functions intersect:



So insert sort is faster from 1.11 < n < 38.67.

(d) 

2. (a) best case: Ω(n2)

Worst case: Ο(n2)

Average case: Θ(n2)

Explanation: The way bubble sort is written, that even in its best case (where the array is sorted) the code still requires that the entire list is tranversed through n2, since there are no switch statements made that will shorten run time, each element is compared in bubble sort. This n2 comes from the outer for loop running (n-1) times and the outer also running (n-1) times, so the best, worst, and average runtimes are all n2.

(b)

3. (a) Base case: At the start of the loop, j = n-1 so x[j … n-1] is the same as x[j … n-1] vacuously, there is one value to compare n-1 and they are the same value.

Inductive step: Assuming the loop invariant holds at the top of the loop, the subarray at the bottom of the loop will either no change (if x[j] > x[j-1]), meaning that the subarray will stay the same as x[j .. n-1] at the time the loop started and then j decrements, adding the same element to both subarrays. If there is a swap (if x[j] < x[j-1]), then the subarray will swap j with j-1, with increases the subarray and temporarily breaks the loop invariant but j will decrement at the end of the swap which adds i-1, to the original subarray thus restoring the loop invariant, so that if it becomes a permutation of the original.

(b) Base case: At the start of the loop, j = n-1 so there is only one element in x[j ... n-1], so it must be the smallest value.

Inductive step: Assume the loop invariant holds at the top of the loop. There are two cases: If there is no swap, then that means x[j-1] is smaller than x[j]. At the bottom of the loop j decrements to include (what was) x[j-1] and it becomes the new j, which is smaller than the previous j and there the smallest element in the array. If there is a swap then x[j] is smaller than x[j-1], j decrements to include x[j], where it is in the x[j-1] position after it is swapped. There the new j, x[j] must be the smallest element in the subarray because it was swapped (smaller than the previous smallest element).

(c) The loop terminates because j is always being decremented at the end of the for loop and it will eventually equal i (the termination condition, because i doesn’t change during that loop. x[i-1] contains the smallest value assuming proof (b) is true, because it will always be the smaller of x[j] and x[j-1], x[i-1 … n-1] contains a permutation of the data originally in x[i-1 .. n-1] assuming proof (a) is true, because j will decrement to include x[j-1] or x[j] if there is a swap.

(d) Base case: i=1, x[0 .. i-2] is x[ 0… -1] which contains 0 elements and therefore zero of the smallest values of 0 x[0 … n-1]. ie. At the top of the array, nothing is sorted, x[0… n-1] will contain a permutation of the original because nothing is sorted it will be exactly the same as the original.

Inductive step: Assume that the loop invariant holds at the top of the loop. At the end of the 2nd loop, by proof b, x[j-1] is x[j] and x[j] is the smallest element of the array. At the end of the second loop, I increments so x[i-1] becomes x[i-2]. x[0… i-2] contains x[i-2], which was x[j], so we have added the sorted element to the subarray. Because of I’s increment, x[0.. i-2] contains one more (i-1) at the smallest value in the original list. Assuming this is true, the loop invariant will hold until i = n-1 as x[0..i-2] gains one sorted element each time i increments, so it contains i-1 of the smallest value in the original list. At the end of the outer for loop the array will be the same array as the one at the end of the inner for loop. Therefore by proof (a), the array will be a permutation of the original array, and this is maintained at the end of the outer loop.

(e) The outer loop terminates because I is being incremented at the end each time. It will eventually be equal to n(the termination condition) because n does not change. The entire array is sorted correctly because at the last iteration when i = n-1, I is the last thing in the array. By proof (d), x[0 .. i-2] contains elements that are in sorted order. By proof (b) , i-1 is sorted because it is the lesser of x[j] (x[i]) and x[j-1] (x[i-1]). Therefore because everything before i is in sorted order, i must also be sorted because it is the last element in the array and it must be the largest element of the array. After this iteration i becomes n and the loop terminates, leaving the entire array sorted.