Lab 1: Sorting Integers

Sam Harder

1. While the computationally expensive process in selection sort is the comparisons needed to find a max in the unsorted portion of an array, in insertion sort the swapping of terms is the most costly process. Swapping terms in an array is more time consuming than comparing terms, therefore for extremely random lists, where lots of swapping is needed, insertion sort takes much longer than selection sort.

2. The swapping of terms, which is where insertion sort takes the most time, is conditional on there being terms to swap. If no terms need a swap, then the most costly part of insertion sort is never executed. Selection sort on the other hand executes the same number of comparisons even if the list is already sorted. It is probable that this advantage would persist even for partially sorted arrays when we observe the huge gap in performance for already sorted lists and the only slight difference in performance for unsorted lists (presumably partially sorted lists are somewhere along this continuum of efficiency).

3. The number of times the for loop executes in selection sort for an array of length n is almost exactly equal to the sum of the numbers 1 to n, or $n(n+1)/2$ or $(n^2+n)/2$. The most notable term in this equation is the n^2, since for large n, $n(n+1) \approx n^2$. So we may say that the time is directly proportional to the number of comparisons, i.e. $t = kn^2$. If we substitute in cn for n we get kc^2n^2, which is just $c^2(kn^2) = c^2t$. Therefore when we multiply the length of an array by c we increase the time to sort by c^2.

4. The runtime for insertion sort is dependent on the number of swaps performed. This is clear from the very fast runtime when no swaps are performed (on already sorted arrays). It is hard to say on average how many swaps are performed, yet we can put a lower bound (worst case) on its performance. The array which leads to the least efficient insertion sort is an array in descending order (if insertion sort is coded to create an ascending array). In this case each term encountered by insertion sort must move all the way to "the front of the line" so to speak, or must be swapped the maximum number of times with the term to its left without throwing an indexOutOfBounds exception. Therefore the worst case scenario for number of swaps is that each term at index i is swapped i times (note that the term at index 0 always requires 0 swaps). Therefore the total

swaps is the sum of all the indices, 1 to $n-1$ which is $(n(n-1))/2$. For sufficiently large n, this is approximately n^2 is the only term which significantly affects the growth of time. We assume that the time is directly proportional to the number of swaps, i.e. $t = kn^2$. If we substitute in cn for n we get kc^2n^2, which is just $c^2(kn^2) = c^2t$. Therefore when we multiply the length of an array by c we increase the time to sort by c^2.

5. The case where the array is already sorted is no different than for a random array. Selection sort performs the same number of comparisons regardless of whether the list is sorted or not. Therefore if we scale the array by c the time to sort will scale by c^2.

6. When the list is perfectly sorted then the for loop to swap terms never executes. Therefore the runtime is dependent on how many comparisons between adjacent terms insertion sort performs. For an array of length n, $n-1$ comparisons are required. It is pretty clear that for each additional term one more comparison is needed. This a linear relationship between computer work (number of swaps) and the length of the array. Therefore, given an initial array of length n and sorting time t, the time to sort an already sorted list of length cn is simply ct. The data supports this approximation (my computer could not handle input beyond 500,000,000):

Array Length	Time to Sort
1.0×10^8	0.044044964
2.0×10^8	0.100644517
5.0×10^8	0.239353141

7. I would recommend insertion sort because insertion sort has the same worst case as selection sort but has a much better best case when the list is already sorted. The only time that selection sort would be much better is if one knew that the input arrays were always perfectly random.