Baby Steps are Slow
Terminology: Given any array $A$ of $n$ data values, we will say that a pair is any two entries $(A[i], A[j])$ with $i < j$. We say that a pair $(A[i], A[j])$ is inverted if $A[i] > A[j]$. 
Note that with \( n \) data values there are \( n(n-1)/2 \) pairs: \( A[0] \) is paired with \( (n-1) \) entries, \( A[1] \) is paired with \( (n-2) \) entries and so forth. The numbers

\[
(n-1) + (n-2) + \ldots + 1
\]

sum to

\[
n(n-1)/2
\]
Now, suppose we start with $n$ distinct data values. Think of all of the different ways we could order them. Each ordering has a reversal (just put the data in the opposite order). A pair that is not inverted in one of these orderings is inverted in its reversal. If we sum the inversions in any ordering and in its reversal we get $n(n-1)/2$ because each pair is inverted in one of the two orderings.

This means the average number of inversions over all possible orderings is $n(n-1)/4$. 
**Theorem:** Any sorting algorithm that sorts by interchanging adjacent data elements (BubbleSort, InsertionSort) or that moves an element k places only after doing k comparisons (SelectionSort) has an average-case running time at least $\Omega(n^2)$.

**Proof:** A data interchange of adjacent elements will correct only one inversion, and on average there are $n(n-1)/4$ inversions to correct. An interchange of elements k steps apart corrects at most k inversions.
Moral: If we want to do better than $O(n^2)$ in sorting, we need to find better ways to move the data around. One step per comparison won't do the trick.

Moral: One step per comparison is the only option for sorting linked lists in place, so sorting a linked list as a linked list is inherently $O(n^2)$. If you have a large linked list it would be faster to copy the data into an array, sort the array efficiently, and copy the data back into the linked list.