Why arrays and linked lists? One of the most common use cases for all sorts of computati is to put data in a "bin" for analysis and/or repetitive tasks. Often one wants a sorted list like when one needs to search through a list multiple times.

Our first sorting algorithm is Selection Sort.

Sort.

Input: An array a of values. ~~either [illegible]~~.

Output: An array s.

The assumption is that the values in a are totally ordered, that is, for each pair of entries, say, a[i] and a[j] at least one of following is true:

- $a[i] \leq a[j]$,
- $a[j] \leq a[i]$.

} The values could be strings and $\leq$ could be the "lexicographical" (ie. dictionary) order.

The output array s has the same length as a and the same entries, possibly in a different order such that

$$s[i] \leq s[i+1],$$

for all relevant i.

(code for selection sort.)

What is the complexity of the algorithm?

- The outer loop runs through all n entries.
- The inner loop goes through n-1, n-2, ..., 2, 1, 0 entries. Since

$$n - i = O(n)$$

for $i = 1, 2, \ldots, n$, the inner loop runs through $O(n)$ entries.

Because each entry in the outer loop does an $O(n)$-operation and the outer loop has n entries, the algorithm is $O(n^2)$.

This is not the fastest sorting algorithm.

(Bubble sort code.)

What's happening? How is the algorithm sorting the list? (Is it actually sorting?!)

It is sorting, and it is making swaps of adjacent entries. The algorithm performs a number of passes through the list. In the worst-case, the first pass puts the last element in place. The second pass puts the second-to-last element in place, and so on.