We will start with something simple--the problem of sorting, or arranging values in ascending order.

Sorting appears in a lot of scenarios in our everyday lives: alphabetizing books, sorting playing cards, and indexing in directories are among them.

But in the simplest form the problem is as follows: Given an array A = [a(1) ... a(n)] determine a permutation B = [b(1) ... b(n)] such that if $p \le q$ then $b(p) \le b(q)$. Essentially, rearranging the elements of A in ascending order.

One simple way to achieve this: scan the array and find the minimum value. Said value goes in position 1. Then, among the other n - 1 elements (indices 2 ... n) repeat: find the minimum among them, place in position 2. Then repeat for position 3 and so on until you reach the end.

In writing, this algorithm looks a bit like this:

```
SELECTION-SORT(A):
    for i = 1 to A.length - 1:
        ind = i
        for j = i+1 to A.length:
            if (A[j] < A[ind]) ind = j
        endfor
        swap(A[ind], A[i])
    endfor
    return A
```

How can we show that this always works? We proceed inductively using iterations of the outer loop (i). In iteration 1, the algorithm finds the smallest element of the array and places it in position 1.

Now, assume that the smallest k - 1 elements have been correctly placed in positions [1 ... (k - 1)]. Therefore, all other elements are larger than element A[k - 1]. A[k] is naturally the smallest element among them.

This is exactly what the inner loop does: it scans the rest of the array, from k to n, and records the index of the minimum element. Finally, once that loop completes, we swap element k with element (index), placing the desired element in position k.

Therefore we can see that the first (n - 1) iterations of the outer loop correctly place the lowest n - 1 elements into the correct positions. The last element falls naturally into position n.

But how good is this algorithm? The outer loop (i) ranges from 1 to n - 1. The inner loop, for each i, ranges from (i + 1) to (n). The inner loop contains a fixed amount of work. If we add this up we get that the number of times the inner loop runs is 1 + 2 + ... (n - 1) = n(n - 1) / 2.

This measure is a quadratic polynomial in n. What this means is that the runtime of this algorithm is roughly proportional to the square of the input size: doubling the input size quadruples the runtime.

While this is a good start, there are algorithms that can sort values quicker, where increasing the input size does not quadratically increase the runtime. More specifically, the runtime will be proportional to n log n, which is a function that grows a lot slower than n^2.