**Sorting Intuition**

* Imagine some data that you can examine all at once.
* To sort it, you could select the largest item and put it in its place, select the next largest and put it in its place, and so on.
* For a card player, this process is analogous to looking at an entire hand of cards and ordering it by selecting cards one at a time in their proper order.
* The selection sort formalizes these intuitive notions.

**Selection Sort Overview**

1. To sort an array into ascending order, you first search it for the largest item.
2. Because you want the largest item to be in the last position of the array, you swap the last item with the largest item, even if these items happen to be identical.
3. Now, ignoring the last—and largest—item of the array, you search the rest of the array for its next largest item and swap it with its last item, which is the next-to-last item in the original array.
4. You continue until you have selected and swapped *n* – 1 of the *n* items in the array.
5. The remaining item, which is now in the first position of the array, is in its proper order, so it is not considered further.

**Example Walkthrough**

* Figure 11-1 provides an example of a selection sort.
* As the items in this figure are ordered, they are shaded in blue and no longer touched.
* This convention will be used throughout this chapter.

1. Beginning with five integers, you select the largest—37—and swap it with the last integer—13.
2. Next, you select the largest integer of the still unsorted part of the array (the first four integers in the array). The next largest integer is 29, and it is swapped with the top index of the unsorted array (next-to-last integer in the array), which holds the value 13.
3. Notice that the next selection—14—is already in its proper position, but the algorithm ignores this fact and performs a swap of 14 with itself. It is more efficient in general to occasionally perform an unnecessary swap than it is to continually ask whether the swap is necessary.
4. Finally, you select the 13 and swap it with the item in the second position of the array—10. The array is now sorted into ascending order.

**A picture containing text, electronics

Description automatically generated**

**Selection Sort Algorithm**

You can sort an array using the selection sort algorithm by either:

1. Repeatedly finding the element with the **minimum value** (ascending order) from the remaining (unsorted) subarray and switching it with the beginning element of the remaining subarray.
2. Repeatedly finding the element with the **maximum value** (descending order) from the unsorted part of the array and switching it with the end element of the remaining subarray.

The algorithm maintains **two subarrays** in a given array.

1. The subarray which is already **sorted**.
2. Remaining subarray which is **unsorted**.

The algorithm requires two general steps:

1. The **minimum/maximum value** (considering ascending order) is found by iterating over the unsorted subarray.
2. The minimum element is **swapped** with the **beginning/end** of the sorted subarray (beginning/end of the unsorted array).

The following example explains the above steps and constraints:

* First array represents array before swapping minimum element with beginning of unsorted array. Second array represents the array after the swap.
* Gray elements are the minimum array elements chosen to be swapped.
* Blue elements comprise the sorted part of the array.

int arr[] = { 64 25 12 22 11 };

// Find the minimum element in arr[0...4] and place it at beginning

64 25 12 22 11

**11** 25 12 22 64

// Find the minimum element in arr[1...4] and place it at beginning of arr[1...4]

**11** 25 12 22 64

11 **12** 25 22 64

// Find the minimum element in arr[2...4] and place it at beginning of arr[2...4]

11 **12** 25 22 64

11 12 **22** 25 64

// Find the minimum element in arr[3...4] and place it at beginning of arr[3...4]

11 12 **22** 25 64

11 12 22 **25** 64

25 is already in its proper position, but the algorithm ignores this fact, and the element performs a swap on itself.

It is more efficient in general to occasionally perform an unnecessary swap than it is to continually ask whether the swap is necessary.

**Analysis**

* As you can see from the previous algorithm, sorting in general
  + compares,
  + exchanges,

or

* + moves items.
* As a first step in analyzing such algorithms, you should count these operations.
* Generally, such operations are more expensive than ones that control loops or manipulate array indexes, particularly when the data to be sorted is more complex than integers or characters.
* Thus, our approach ignores these incidental operations. You should convince yourself that by ignoring such operations we do not affect our final result. (See Exercise 1.)
* There is a nested for loop in the function selectionSort().
  + The outer loop executes *n* – 1 times. This loop calls a function findMinElement() and swap() *n* – 1 times.
  + The inner loop is inside the findMinElement() function, which iterates over the array n – 1 times to find the smallest value in the array (each subsequent call to this function will be 1 smaller because we will only be passing the index to the last part of the sorted part of the array).

Thus, findMinElement() executes a total of times

* + The *n* – 1 calls to swap result in *n* – 1 exchanges. Each exchange requires three assignments, each of which take O(1) time.

Thus, the calls to swap requires

moves

Together, a selection sort of *n* items requires the following number of operations:



major operations. By applying the properties of the growth-rate functions given in Chapter 10, you can ignore low-order terms to get O(*n*2 / 2) and then ignore the multiplier 1/2 to get O(*n*2). **Thus, the selection sort is O(*n*2).**

**O(*n*2) is Not Optimal – But Can Be Used on Small Datasets**

Although a selection sort does not depend on the initial arrangement of the data, which is an advantage of this algorithm, it is appropriate only for small *n* because O(*n*2) grows rapidly.

While the algorithm requires O(*n*2) comparisons, it **requires only O(*n*) data moves**.

A selection sort could be a good choice over other approaches when **data moves are costly but comparisons are not**.

Such might be the case if each data item is lengthy but the sort key is short.