18.1 — Sorting an array using selection sort

A case for sorting

Sorting an array is the process of arranging all of the elements in the array in a particular order. There are many different cases in which sorting an array can be useful. For example, your email program generally displays emails in order of time received, because more recent emails are typically considered more relevant. When you go to your contact list, the names are typically in alphabetical order, because it's easier to find the name you are looking for that way. Both of these presentations involve sorting data before presentation.

Sorting an array can make searching an array more efficient, not only for humans, but also for computers. For example, consider the case where we want to know whether a name appears in a list of names. In order to see whether a name was on the list, we'd have to check every element in the array to see if the name appears. For an array with many elements, searching through them all can be expensive.

However, now assume our array of names is sorted alphabetically. In this case, we only need to search up to the point where we encounter a name that is alphabetically greater than the one we are looking for. At that point, if we haven't found the name, we know it doesn't exist in the rest of the array, because all of the names we haven't looked at in the array are guaranteed to be alphabetically greater!

It turns out that there are even better algorithms to search sorted arrays. Using a simple algorithm, we can search a sorted array containing 1,000,000 elements using only 20 comparisons! The downside is, of course, that sorting an array is comparatively expensive, and it often isn't worth sorting an array in order to make searching fast unless you're going to be searching it many times.

In some cases, sorting an array can make searching unnecessary. Consider another example where we want to find the best test score. If the array is unsorted, we have to look through every element in the array to find the greatest test score. If the list is sorted, the best test score will be in the first or last position (depending on whether we sorted in ascending or descending order), so we don't need to search at all!

How sorting works

Sorting is generally performed by repeatedly comparing pairs of array elements, and swapping them if they meet some predefined criteria. The order in which these elements are compared differs depending on which sorting algorithm is used. The criteria depends on how the list will be sorted (e.g. in ascending or descending order).

To swap two elements, we can use the std::swap() function from the C++ standard library, which is defined in the utility header.

```
#include <iostream>
#include <utility>

int main()
{
    int x{ 2 };
    int y{ 4 };
    std::cout << "Before swap: x = " << x << ", y = " << y << '\n';
    std::swap(x, y); // swap the values of x and y
    std::cout << "After swap:  x = " << x << ", y = " << y << '\n';

    return 0;
}
```

This program prints:

```
Before swap: x = 2, y = 4
After swap:  x = 4, y = 2
```

Note that after the swap, the values of x and y have been interchanged!

Selection sort

There are many ways to sort an array. Selection sort is probably the easiest sort to understand, which makes it a good candidate for teaching even though it is one of the slower sorts.

Selection sort performs the following steps to sort an array from smallest to largest:

1. Starting at array index 0, search the entire array to find the smallest value
2. Swap the smallest value found in the array with the value at index 0
3. Repeat steps 1 & 2 starting from the next index

In other words, we're going to find the smallest element in the array, and swap it into the first position. Then we're going to find the next smallest element, and swap it into the second position. This process will be repeated until we run out of elements.

Here is an example of this algorithm working on 5 elements. Let's start with a sample array:

{ 30, 50, 20, 10, 40 }

First, we find the smallest element, starting from index 0:

{ 30, 50, 20, **10**, 40 }

We then swap this with the element at index 0:

{ **10**, 50, 20, **30**, 40 }

Now that the first element is sorted, we can ignore it. Now, we find the smallest element, starting from index 1:

{ *10*, 50, **20**, 30, 40 }

And swap it with the element in index 1:

{ *10*, **20**, **50**, 30, 40 }

Now we can ignore the first two elements. Find the smallest element starting at index 2:

{ *10*, *20*, 50, **30**, 40 }

And swap it with the element in index 2:

{ *10*, *20*, **30**, **50**, 40 }

Find the smallest element starting at index 3:

{ *10*, *20*, *30*, 50, **40** }

And swap it with the element in index 3:

{ *10*, *20*, *30*, **40**, **50** }

Finally, find the smallest element starting at index 4:

{ *10*, *20*, *30*, *40*, **50** }

And swap it with the element in index 4 (which doesn't do anything):

{ *10*, *20*, *30*, *40*, **50** }

Done!

{ 10, 20, 30, 40, 50 }

Note that the last comparison will always be with itself (which is redundant), so we can actually stop 1 element before the end of the array.

Selection sort in C++

Here's how this algorithm is implemented in C++:

```
#include <iostream>
#include <iterator>
#include <utility>

int main()
{
	int array[]{ 30, 50, 20, 10, 40 };
	constexpr int length{ static_cast<int>(std::size(array)) };

	// Step through each element of the array
	// (except the last one, which will already be sorted by the time we get
there)
	for (int startIndex{ 0 }; startIndex < length - 1; ++startIndex)
	{
		// smallestIndex is the index of the smallest element we've
encountered this iteration
		// Start by assuming the smallest element is the first element of
this iteration
		int smallestIndex{ startIndex };

		// Then look for a smaller element in the rest of the array
		for (int currentIndex{ startIndex + 1 }; currentIndex < length;
++currentIndex)
		{
			// If we've found an element that is smaller than our
previously found smallest
			if (array[currentIndex] < array[smallestIndex])
				// then keep track of it
				smallestIndex = currentIndex;
		}

		// smallestIndex is now the index of the smallest element in the
remaining array
		// swap our start element with our smallest element (this sorts it
into the correct place)
		std::swap(array[startIndex], array[smallestIndex]);
	}

	// Now that the whole array is sorted, print our sorted array as proof it
works
	for (int index{ 0 }; index < length; ++index)
		std::cout << array[index] << ' ';

	std::cout << '\n';

	return 0;
}
```

The most confusing part of this algorithm is the loop inside of another loop (called a **nested loop**). The outside loop (startIndex) iterates through each element one by one. For each iteration of the outer loop, the inner loop (currentIndex) is used to find the smallest element in

the remaining array (starting from startIndex+1). smallestIndex keeps track of the index of the smallest element found by the inner loop. Then smallestIndex is swapped with startIndex. Finally, the outer loop (startIndex) advances one element, and the process is repeated.

Hint: If you're having trouble figuring out how the above program works, it can be helpful to work through a sample case on a piece of paper. Write the starting (unsorted) array elements horizontally at the top of the paper. Draw arrows indicating which elements startIndex, currentIndex, and smallestIndex are indexing. Manually trace through the program and redraw the arrows as the indices change. For each iteration of the outer loop, start a new line showing the current state of the array.

Sorting names works using the same algorithm. Just change the array type from int to std::string, and initialize with the appropriate values.

std::sort

Because sorting arrays is so common, the C++ standard library includes a sorting function named `std::sort`. `std::sort` lives in the <algorithm> header, and can be invoked on an array like so:

```
#include <algorithm> // for std::sort
#include <iostream>
#include <iterator> // for std::size

int main()
{
	int array[]{ 30, 50, 20, 10, 40 };

	std::sort(std::begin(array), std::end(array));

	for (int i{ 0 }; i < static_cast<int>(std::size(array)); ++i)
		std::cout << array[i] << ' ';

	std::cout << '\n';

	return 0;
}
```

By default, std::sort sorts in ascending order using operator< to compare pairs of elements and swapping them if necessary (much like our selection sort example does above).

We'll talk more about `std::sort` in a future chapter.

Quiz time

Question #1

Manually show how selection sort works on the following array: { 30, 60, 20, 50, 40, 10 }. Show the array after each swap that takes place.

Show Solution

Question #2

Rewrite the selection sort code above to sort in descending order (largest numbers first). Although this may seem complex, it is actually surprisingly simple.

Show Solution

Question #3

This one is going to be difficult, so put your game face on.

Another simple sort is called “bubble sort”. Bubble sort works by comparing adjacent pairs of elements, and swapping them if the criteria is met, so that elements “bubble” to the end of the array. Although there are quite a few ways to optimize bubble sort, in this quiz we’ll stick with the unoptimized version here because it’s simplest.

Unoptimized bubble sort performs the following steps to sort an array from smallest to largest:
A) Compare array element 0 with array element 1. If element 0 is larger, swap it with element 1.
B) Now do the same for elements 1 and 2, and every subsequent pair of elements until you hit the end of the array. At this point, the last element in the array will be sorted.
C) Repeat the first two steps again until the array is sorted.

Write code that bubble sorts the following array according to the rules above:

```
int array[]{ 6, 3, 2, 9, 7, 1, 5, 4, 8 };
```

Print the sorted array elements at the end of your program.

Hint: If we’re able to sort one element per iteration, that means we’ll need to iterate roughly as many times as there are numbers in our array to guarantee that the whole array is sorted.
Hint: When comparing pairs of elements, be careful of your array’s range.

Show Solution

Question #4

Add two optimizations to the bubble sort algorithm you wrote in the previous quiz question:

- Notice how with each iteration of bubble sort, the biggest number remaining gets bubbled to the end of the array. After the first iteration, the last array element is sorted. After the second iteration, the second to last array element is sorted too. And so on… With each iteration, we don't need to recheck elements that we know are already sorted. Change your loop to not re-check elements that are already sorted.
- If we go through an entire iteration without doing a swap, then we know the array must already be sorted. Implement a check to determine whether any swaps were made this iteration, and if not, terminate the loop early. If the loop was terminated early, print on which iteration the sort ended early.

Your output should match this:

```
Early termination on iteration 6
1 2 3 4 5 6 7 8 9
```

Show Solution