6.4 — Sorting an array using selection sort

BY ALEX ON JULY 3RD, 2007 | LAST MODIFIED BY ALEX ON SEPTEMBER 7TH, 2017

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 algorithm header. For efficiency reasons, std::swap() was moved to the utility header in C++11.

```
#include <algorithm> // for std::swap, use <utility> instead if C++11
#include <iostream>

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';
}
```

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++:

```
1  #include <algorithm> // for std::swap, use <utility> instead if C++11
```

```cpp
#include <iostream>

int main()
{
    const int length = 5;
    int array[length] = { 30, 50, 20, 10, 40 };

    // 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 smallest element in the remaining array
                // swap our start element with our smallest element (this sorts it into the correct pla
ce)
        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] << ' ';

    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:

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

int main()
{
    const int length = 5;
    int array[length] = { 30, 50, 20, 10, 40 };

    std::sort(array, array+length);

```

```
    for (int i=0; i < length; ++i)
        std::cout << array[i] << ' ';

    return 0;
}
```

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

Quiz

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.

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.

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:

```
const int length(9);
int array[length] = { 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 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.

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
```

Quiz solutions

1) **Hide Solution**

30 60 20 50 40 10
10 60 20 50 40 **30**
10 **20 60** 50 40 30
10 20 **30** 50 40 **60**
10 20 30 **40 50** 60

10 20 30 40 **50** 60 (self-swap)
10 20 30 40 50 **60** (self-swap)

2) **Hide Solution**

Simply change:

```
        if (array[currentIndex] < array[smallestIndex])
```

to:

```
        if (array[currentIndex] > array[smallestIndex])
```

smallestIndex should probably be renamed largestIndex as well.

```
#include <algorithm> // for std::swap, use <utility> instead if C++11
#include <iostream>

int main()
{
    int array[] = { 30, 50, 20, 10, 40 };
    const int length = sizeof(array) / sizeof(array[0]);

    // Step through each element of the array except the last
    for (int startIndex = 0; startIndex < length - 1; ++startIndex)
    {
        // largestIndex is the index of the largest element we've encountered so far.
        int largestIndex = startIndex;

        // Search through every element starting at startIndex + 1
        for (int currentIndex = startIndex + 1; currentIndex < length; ++currentIndex)
        {
            // If the current element is smaller than our previously found largest
            if (array[currentIndex] > array[largestIndex])
                // This is the new largest number for this iteration
                largestIndex = currentIndex;
        }

        // Swap our start element with our largest element
        std::swap(array[startIndex], array[largestIndex]);
    }

    // Now print our sorted array as proof it works
    for (int index = 0; index < length; ++index)
        std::cout << array[index] << ' ';

    return 0;
}
```

3) **Hide Solution**

```
#include <algorithm> // for std::swap, use <utility> instead if C++11
#include <iostream>

int main()
{
    int array[] = { 6, 3, 2, 9, 7, 1, 5, 4, 8 };
    const int length = sizeof(array) / sizeof(array[0]);

    // Step through each element of the array except the last
    for (int iteration = 0; iteration < length-1; ++iteration)
    {
        // Search through all elements up to the end of the array - 1
        // The last element has no pair to compare against
        for (int currentIndex = 0; currentIndex < length - 1; ++currentIndex)
        {
            // If the current element is larger than the element after it, swap them
            if (array[currentIndex] > array[currentIndex+1])
```

```
                std::swap(array[currentIndex], array[currentIndex + 1]);
        }
    }

    // Now print our sorted array as proof it works
    for (int index = 0; index < length; ++index)
        std::cout << array[index] << ' ';

    return 0;
}
```

4) **Hide Solution**

```
#include <algorithm> // for std::swap, use <utility> instead if C++11
#include <iostream>

int main()
{
    int array[] = { 6, 3, 2, 9, 7, 1, 5, 4, 8 };
    const int length = sizeof(array) / sizeof(array[0]);

    // Step through each element of the array except the last
    for (int iteration = 0; iteration < length-1; ++iteration)
    {
        // Account for the fact that the last element is already sorted with each subsequent iteration
        // so our array "ends" one element sooner
        int endOfArrayIndex(length - iteration);

        bool swapped(false); // Keep track of whether any elements were swapped this iteration

        // Search through all elements up to the end of the array - 1
        // The last element has no pair to compare against
        for (int currentIndex = 0; currentIndex < endOfArrayIndex - 1; ++currentIndex)
        {
            // If the current element is larger than the element after it
            if (array[currentIndex] > array[currentIndex + 1])
            {
                // Swap them
                std::swap(array[currentIndex], array[currentIndex + 1]);
                swapped = true;
            }
        }

        // If we haven't swapped any elements this iteration, we're done early
        if (!swapped)
        {
            // iteration is 0 based, but counting iterations is 1-based.  So add 1 here to adjust.
            std::cout << "Early termination on iteration: " << iteration+1 << '\n';
            break;
        }
    }

    // Now print our sorted array as proof it works
    for (int index = 0; index < length; ++index)
        std::cout << array[index] << ' ';

    return 0;
}
```



6.5 -- Multidimensional Arrays


Index