Searching and Sorting 1-D Arrays

Linear Search of an Array

A linear search is when one moves through an entire 1-dimensional array, element-by-element, to search for a matching value. The search is linear because in the worst-case scenario, you might have to search through the entire array to find what you are looking for. In other words, for an array of length N, you have to search N items and the time it takes to search is directly proportional (or linear) to the array length.

A 2-dimensional array of size N x N would require, in the worst-case scenario, N^2 elements to be searched! There are clever algorithms that can search faster, but this topic is beyond the scope of this course. At the end of this tutorial, I placed a reference to an interesting YouTube video where sorting algorithms are visualized. You should check these out!

Here is a code example for searching through an array to find the maximum value.

```
// assume first value inarray is max value
int max = myArray[0];

// check all other values in array to determine if they are larger
for (int index = 1; index < arrayLength; index++)
{
      if (myArray[index] > max)
      {
            max = myArray[index];
      }
}
```

Sorting an Array

Typical examples of sorting are

- arranging string or character data in alphabetical order
- arranging numerical data in ascending or descending order

It would be impossible to sort data, as they are being read from a file, without having a place to store the data, so arrays--or other types of data containers--are essential in a sorting task.

To sort an array, you have to search the array--finding a minimum or maximum value. Even sorting a data-set alphabetically is really comparing the relative “sizes” of strings. For example, when using the relational operators, greater than (>) and less than (<) in C++, “apple” is less than “zebra”, and “bat” is less than “bar”.

There are numerous ways to sort data. Some sorting methods are very easy to understand and implement, but they take a long time (alot of operations). Other sorting methods are very efficient, but they are very difficult to understand and implement.

Selection Sort

One of the simplest sorting methods to understand is the selection sort algorithm. The idea is that for every position in the array, you search for the next appropriate size data that belongs there and then you place the correct data in that position. The "next appropriate data" depends on how you want the data to be sorted.

Let's take the example of integer data, initially in random order, that we want to be sorted into ascending order.

Unsorted array

7	5	10	1	3

Array sorted in ascending order

1	3	5	7	10

The selection sort algorithm for ascending order says:

for each position in the array starting with the first position

-find the smallest number contained in the subarray starting at that position

-swap the data in those two positions

Let us see this algorithm in action.

First pass: Take the first element and swap it with the smallest element.

before

7	5	10	1	3
First element			smallest	

after

1	5	10	7	3

Now the first element is the smallest value, so leave it alone!

Second pass: Take the second element and swap it with the smallest element

before

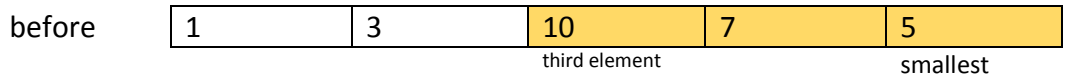
1	5	10	7	3
	Second element			smallest

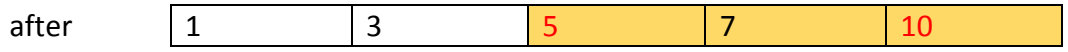
after

1	3	10	7	5

Now the first TWO elements are in their proper places, so leave them alone!

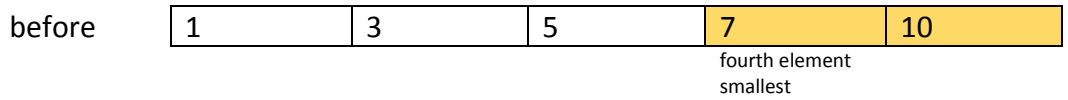
Third pass: Take the third element and swap it with the smallest element.





Now the first THREE elements are in their proper places, so leave them alone!

Fourth pass: Take the fourth element swap it with the smallest element.



The fourth element was the smallest element so the value was "swapped" with itself!

An alternate way of looking at this algorithm is to consider the array as having a sorted portion and an unsorted portion. After every pass, the sorted portion becomes one element longer and the unsorted portion becomes one element shorter. The sorting is complete after the algorithm compares the last two elements in the array. In our example, the unsorted portion was colored yellow in each step. Note that the array was 5 elements long and we needed 4 passes to sort it. You need N-1 passes to sort an array with N elements.

There are many variations on the selection sort. For example, the algorithm could start at the other end of the array and search for the largest values. There are several videos on line that demonstrate the insertion sort. Here are a couple that I like:

https://www.youtube.com/watch?v=DFG-XuyPYUQ (this one is a very good and quick explanation by a person)

https://www.youtube.com/watch?v=8oJS1BMKE64 (this is just a visualization)

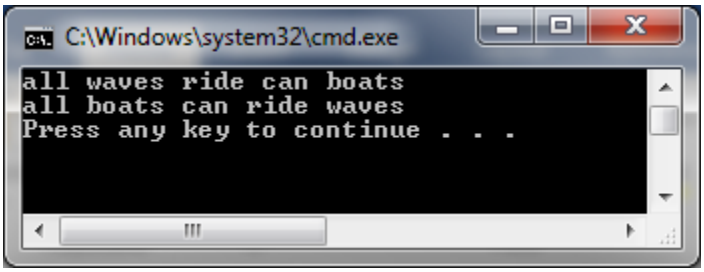
https://www.youtube.com/watch?v=kPRA0W1kECg (visual and audio!)

An example selection sort is on the next page. The example sorts a string array in alphabetical order.

```
///  SelectionSort        This function sorts a string array into alphabetical order.
///                       This function assumes that the strings are all one case, and
///                       does not distinguish between upper and lower case. If a string
///                       contains UPPER case letters, those letters are lower value than
///                       the lower case letters, so may be out of alphabetical order.
///
void SelectionSort(string strArray[], int arrayLength)
{
      int currentElement;
      int smallest;

      /// go through all array elements up to the second to last element. On last iteration,
      ///   there are only two elements left in the unsorted array to compare.
      for (int currentElement = 0; currentElement < arrayLength - 1; currentElement++)
      {
            // for the sub-array of currentElement through end-of-array, find the position
            //   of the smallest value.

            //  Initialize smallest to the first element in the sub array.
            smallest = currentElement;

            /// check ALL other elements in sub-array against current smallest position
            for (int i = currentElement + 1; i < arrayLength; i++)
            {
                  if (strArray[i] < strArray[smallest])
                  {
                        smallest = i;
                  }

            }
            //  now that sub-array has been searched, we have the index of the smallest value
            //   exchange the values contained in current element and smallest element.
            Exchange(strArray[currentElement], strArray[smallest]);
      }

}
```

The following short program calls the sort function, and the output is shown below:

```
const int LENGTH = 5;

int main(void)
{
	string strArray[LENGTH] = { "all", "waves", "ride", "can", "boats" };

	PrintArray(strArray, LENGTH);
	cout << endl;

	SelectionSort(strArray, LENGTH);

	PrintArray(strArray, LENGTH);
	cout << endl;

	return 0;
}
```



If we change the string array to include one string with a capital letter (“waves” is now “Waves”), then the array is not properly alphabetized. You should be aware of the uppercase vs. lower case issue.

```
string strArray[LENGTH] = { "all", "Waves", "ride", "can", "boats" };
```