Lecture 26 — Sorting

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 27, 2016

Acknowledgments: W.D. Bishop

The Sorting Problem

We saw in the discussion about searching that sorting is worthwhile because it dramatically improves the speed of searching.

Sorted sequences of objects can often be manipulated quickly.

Example: the binary search algorithm.

We might have an intuitive idea about what it means for data to be sorted, but let's agree on a formal definition.

The Sorting Problem

The sorting problem, formally:

Input: A sequence of n numbers $\{a_1, a_2, \ldots, a_n\}$

Output: A reordering $\{a'_1, a'_2, \ldots a'_n\}$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

In less formal terms, a solution to the sorting problem:

takes a sequence of n numbers as input; and
returns those numbers in non-decreasing order.

For input of $\{31, 41, 59, 26, 41\}$, the output is $\{26, 31, 41, 41, 59\}$

The step-by-step process by which we reorder these values is the sorting algorithm.

More General Sorting

Of course, it's possible to sort things other than numbers (e.g., names) and it might be desirable to sort in another order (e.g., descending).

Thus, a more general definition of a sorting algorithm:
A sorting algorithm places a sequence of values in a predefined order.

To keep things simple while learning about sorting, we'll use:

- numbers; and
- non-descending order

Sorting Algorithm Discussion

Many algorithms exist for sorting data:

- Bogo sort
- Insertion sort
- Selection sort
- Shellsort
- Heapsort
- Quicksort
- Bubblesort
- Merge sort

... and more.

Sorting Algorithm Discussion

Ultimately, which algorithm is best for a given application may depend on a number of factors.

The above algorithms all have different characteristics:

- Algorithm complexity
- Space complexity (storage needed to sort data)
- Performance (how long it takes)
- Data structure (what types it can be used on)

A sorting algorithm is correct if, for every input, it halts with the correct output.

An incorrect algorithm might not halt at all for some input sequences (infinite loop) or might return the output not sorted as expected.

Simple Sorting Techniques

In this course, we examine only a few of the sorting techniques:

1. Bogo sort
2. Selection sort
3. Insertion sort

The algorithms are (relatively) easy to understand.

The algorithms can be applied to arrays and linked lists.

More complex algorithms are introduced & analyzed in ECE 250.

Bogo Sort

The Bogo sort can be summarized as follows:

While the array is unsorted, randomly order the entries.

Through random chance, we'll eventually end up with a sorted array.
But how long will it take...?

This isn't a serious sorting algorithm. It's just used as a baseline to contrast with more realistic algorithms.

Selection Sort

Imagine that you have been taking notes for a course on loose-leaf paper and have been numbering the pages.

You accidentally knock over the pile of pages and they scatter and fly everywhere in your room, landing all over.

You gather them up, but they're out of order & you want to sort them.

A strategy you may choose: selection sort.

Selection Sort

Selection Sort is based on searching: find the minimum value.

You have an unsorted pile and a (currently-empty) sorted pile.

Search the unsorted pile until you find page 1.
    Add it to the sorted pile.

Then look for page 2; add it to the sorted pile after page 1.

Repeat this procedure until all the pages are sorted again.

Selection Sort Algorithm

Given an array of n entries:

For *index* from 0 to $(n - 1)$

1. Find the index of the minimum value (*minIndex*) in the subset of the array from *index* to $(n - 1)$.
2. Swap the value at *minIndex* with the value at *index*

To find the minimum index, we must use a linear search (because the binary search requires sorted data and we're trying to sort...).

Selection Sort Illustration

Consider sorting $\{12, 36, 20, -1\}$ using the selection sort.

After iteration *x*, the first *x* values have been sorted.
Progress is shown by the red highlighting.

Iteration 1: Array Contents = $\{12, 36, 20, -1\}$

- Find minimum value between index 0 and (length - 1)
- Minimum value is -1, at index 3
- Swap minimum value of -1 with value of 12 in index 0.

Selection Sort Illustration

Iteration 2: Array Contents = $\{-1, 36, 20, 12\}$
- Find minimum value between index 1 and (length - 1)
- Minimum value is 12, at index 3
- Swap minimum value of 12 with value of 36 in index 1

Iteration 3: Array Contents = $\{-1, 12, 20, 36\}$
- Find minimum value between index 2 and (length - 1)
- Minimum value is 20, at index 2
- Swap minimum value of 20 with value of 20 in index 2

The array, after sorting, contains: $\{-1, 12, 20, 36\}$.

Larger Array Example

Consider sorting {12, 36, 42, -1, 99, 20, 10, 19, 70}.

{12, 36, 42, -1, 99, 20, 10, 19, 70} // Initial array values
{-1, 36, 42, 12, 99, 20, 10, 19, 70} // After iteration 1
{-1, 10, 42, 12, 99, 20, 36, 19, 70} // After iteration 2
{-1, 10, 12, 42, 99, 20, 36, 19, 70} // After iteration 3
{-1, 10, 12, 19, 99, 20, 36, 42, 70} // After iteration 4
{-1, 10, 12, 19, 20, 99, 36, 42, 70} // After iteration 5
{-1, 10, 12, 19, 20, 36, 99, 42, 70} // After iteration 6
{-1, 10, 12, 19, 20, 36, 42, 99, 70} // After iteration 7
{-1, 10, 12, 19, 20, 36, 42, 70, 99} // After iteration 8

Note that a total of 8 iterations are required to sort 9 values.

In general, selection sort requires $n - 1$ iterations to sort n values.

Selection Sort Code

Here is a simple code implementation of selection sort:

```
void selection_sort( int[] data, int length )
{
    int minIndex;

    for( int index = 0; index < length - 1; index++ )
    {
        minIndex = min( data, length, index );
        swap( data, minIndex, index );
    }
}
```

Implementations of `min` and `swap` are shown on the next slide

...but we (should) already know how to implement those!

Selection Sort Code

```
int min( int[] data, int length, int start )
{
    int minIndex = start;

    for( int index = start + 1; index < length; index++ )
    {
        if( data[index] < data[minIndex] )
        {
            minIndex = index;
        }
    }
    return minIndex;
}

void swap( int[] data, int index1, int index2 )
{
    int tmp = data[index1];
    data[index1] = data[index2];
    data[index2] = tmp;
}
```

Alternative Implementation

The previous code example is written in such a way as to separate out the different parts, such as finding the minimum and the swap.

Consider a more compact implementation of selection sort:

```
void selection_sort( int[] data, int length )
{
  int minIndex;
  int temp;
  for (int i = 0; i < length; i++) {
    for (int j = i, minIndex = i; j < length; j++) {
        if (data[j] < data[minIndex])
        {
            minIndex = j;
        }
    }
    temp = data[i];
    data[i] = data[minIndex];
    data[minIndex] = temp;
  }
}
```

In-Place Sorting

The implementation of selection sort we have examined is an example of an in-place sort: no additional memory is allocated.

In-place means we don't have to allocate a new array for output.
There is only one array and we re-order the values within that.

As an alternative strategy that takes more memory:

For an input array of capacity *n*, allocate an output array of capacity *n*. Then, for each entry of the input array:

1. Find the minimum value in the input array
2. Remove it from the input array
3. Put it in the output array

In-Place Sorting

Why would we choose the alternative that uses more memory?

There are very often space-time tradeoffs.

- An algorithm that takes more memory may run faster;
- An algorithm that doesn't use as much memory may run slower.