Outline

- In this lesson, we will:
  - Describe a selection-sort algorithm to sort an array
  - Consider a straight-forward implementation
  - Observe that it makes more sense to use *helper functions*
  - Make a simple modification that improves its function
  - Consider the appropriate tests for this algorithm
  - Briefly describe the execution time and benefits of this algorithm

Searching sorted arrays

- Suppose you start with an unsorted array of values
  - How can you rearrange the entries so that they are sorted?
- Selection sort describes an algorithm that takes an unsorted array,
  and after a fixed number of steps,
  rearranges the entries so that they are sorted

Searching sorted arrays

- As with linear search and binary search,
  selection sort can be described independent of
  any programming language
  - Given an array of N entries,
    - Find the largest of the first N entries, and swap it with the last entry
    - Find the largest of the first $N - 1$ entries,
      and swap it with the second-last entry
    - Find the largest of the first $N - 2$ entries,
      and swap it with the third-last entry

    and so on, until you are finding the largest of only one entry

Searching sorted arrays

- To implement this in C++, we will use the function declaration

```
void selection_sort( double array[], std::size_t capacity );
```

  - Remember that when you call a function with an array,
    it is the address that is passed,
    so any change to the array entries changes the original array, too

Searching sorted arrays

- Thus, suppose you were to sort an array of capacity `10`:
  - On the first iteration, you would find the largest entry between indices `0` and `9` inclusive, and swap that entry with entry `9`
  - You would proceed as follows

k

Search starting at index	Search up to and including index	Swap with index	
0	9	9	capacity - 1
0	8	8	
0	7	7	
0	6	6	
0	5	5	
0	4	4	
0	3	3	
0	2	2	
0	1	1	k > 0

Searching sorted arrays

- Thus, our implementation would start:

```
for ( std::size_t k{capacity - 1}; k > 0; --k ) {
    // Find the index of the maximum
    // entry between indices 0 and 'k'


    // Swap that entry with the entry at index 'k'

}
```


Searching sorted arrays

- Your first thought might be to implement both these ideas:

```
void selection_sort( double array[], std::size_t capacity ) {
    for ( std::size_t k{capacity - 1}; k > 0; --k ) {
        // Assume the maximum is at index 0:
        std::size_t max_index{0};

        // If any entry between indices 1 and 'k' is larger,
        //    update 'max_index'
        for ( std::size_t j{1}; j <= k; ++j ) {
            if ( array[j] > array[max_index] ) {
                max_index = j;
            }
        }

        // Swap the two entries
        double tmp{array[max_index]};
        array[max_index] = array[k];
        array[k] = tmp;
    }

    assert( is_sorted( array, capacity ) == capacity );
}
```

Searching sorted arrays

- Suppose, however, we had authored two additional functions:

```
void swap( double &first, double &second );
std::size_t find_max( double array[], std::size_t capacity );
```


Searching sorted arrays

- For example, swapping two values would be easy:

```
void swap( double &first, double &second ) {
    double tmp{ first };
    first = second;
    second = tmp;
}
```

- This is implemented in the standard template library, under the name `std::swap`
  - You would have to include the library `#include <utility>`

Searching sorted arrays

- Similarly, finding the maximum entry of an array is easy, too:

```
std::size_t find_max( double array[], std::size_t capacity ) {
    assert( capacity > 0 );

    std::size_t max_index{0};

    for ( std::size_t k{1}; k < capacity; ++k ) {
        if ( array[k] > array[max_index] ) {
            max_index = k;
        }
    }

    return max_index;
}
```


Searching sorted arrays

- Thus, using these two *helper functions*, we can simplify our implementation

```
for ( std::size_t k{capacity - 1}; k > 0; --k ) {
    // Find the index of the maximum
    // entry between indices 0 and 'k'
    std::size_t max_index{ find_max( array, k + 1 ) };

    // Swap that entry with the entry at index 'k'
    std::swap( array[max_index], array[k] );
}
```

Searching sorted arrays

- Thus, our final implementation is

```
void selection_sort( double array[], std::size_t capacity ) {
    for ( std::size_t k{capacity - 1}; k > 0; --k ) {
        // Find the maximum entry between 0 and 'k'
        std::size_t max_index{ find_max( array, k + 1 ) };

        // Swap that entry with the entry at index 'k'
        std::swap( array[max_index], array[k] );
    }

    assert( is_sorted( array, capacity ) == capacity );
}
```


Searching sorted arrays

```
void selection_sort( double array[], std::size_t capacity ) {
    for ( std::size_t k{capacity - 1}; k > 0; --k ) {
        std::size_t max_index{0};

        for ( std::size_t j{1}; j <= k; ++j ) {
            if ( array[j] > array[max_index] ) {
                max_index = j;
            }
        }

        double tmp{array[max_index]};
        array[max_index] = array[k];
        array[k] = tmp;
    }
}
```

```
void selection_sort( double array[], std::size_t capacity ) {
    for ( std::size_t k{capacity - 1}; k > 0; --k ) {
        std::size_t max_index{ find_max( array, k + 1 ) };

        std::swap( array[max_index], array[k] );
    }
}
```


Implementing the binary search

- We have finished our algorithm; however, is it possible to make a small incremental improvement?
  - With each iteration of the loop:
    - We are finding the maximum entry from index `0` to index `k`
    - We swap what is at that index with the entry at index `k`
  - If the maximum entry is already in the last index, do we have to perform a swap?
  - How about the following:
    - Find the maximum entry from index `0` to index `k - 1`
    - If that maximum entry is greater than the entry `array[k]`, only then swap the two

Searching sorted arrays

- This only changes our implementation slightly:

```
void selection_sort( double array[], std::size_t capacity ) {
    for ( std::size_t k{capacity - 1}; k > 0; --k ) {
        // Find the maximum entry between 0 and 'k - 1'
        std::size_t max_index{ find_max( array, k ) };

        // If the largest entry before 'array[k]' is
        // greater than the last entry, swap them
        if ( array[max_index] > array[k] ) {
            std::swap( array[max_index], array[k] );
        }
    }

    assert( is_sorted( array, capacity ) == capacity );
}
```

Testing our implementation

- We should author a test for our implementation:
  - How to proceed?
  1. Sort an array of capacity 0
  2. Sort an array of capacity 1
  3. Sort three arrays of capacity 2:
     `{4.5, 7.2}   {8.2, 8.2}   {6.1, -9.0}`
  4. Sort all possible arrays of capacity 3:
     ```
     {0,0,0}
     {0,0,1} {0,1,0} {1,0,0}
     {0,1,1} {1,0,1} {1,1,0}
     {0,1,2} {0,2,1} {1,0,2} {1,2,0} {2,0,1} {2,1,0}
     ```
  5. Sort three arrays of capacity 100:
     ```
     {-7.5, -0.3, 0.0, 1.2, 1.5, 2.70, …, 89.2}
     {32.5, 29.5, 25.9, 24.8, 24.5, …, -18.3, -18.7}
     {5.7, 19.3, -18.2, 24.9, 58.2, 16.8, …, 35.2}
     ```


How much work is involved?

- You will note that the algorithm takes the exact same number of steps no matter what the array looks like
  - In your algorithms and data structures course, you will see that the run time can be calculated by counting the number of statements that are executed
  - In this case, if the capacity is n, it is approximately a scalar multiple of:

$$(n-1) + (n-2) + (n-2) + \ \dots + 3 + 2 + 1 = \frac{(n-1)n}{2}$$

- This implementation has one benefit over all other such algorithms
  - It has the minimum number of changes to entries of the array
  - In our second implementation, if the array is sorted, no changes are made to the array

Summary

- Following this presentation, you now:
  - Understand how to implement a selection sort
  - Know that it is better to create helper functions that perform common tasks
    - In this case, finding the maximum entry and swapping two values
  - Are aware that smaller changes can have benefits to the implementation
  - Have seen a reasonable set of tests for a sorting algorithm
  - Have an overview of the idea of execution and one benefit of this algorithm

References

[1] Wikipedia,
https://en.wikipedia.org/wiki/Selection_sort

[2] Dictionary of Algorithms and Data Structures (DADS)
https://xlinux.nist.gov/dads/HTML/selectionSort.html

Acknowledgments

Proof read by Dr. Thomas McConkey and Charlie Liu.

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

Disclaimer