ECE 150 Algorithms

Douglas Harder

December 2023

1 Algorithms

In our introduction to algorithms, we focused on a number of algorithms associated with arrays and the data stored in those arrays.
We defined a sorted array as any array where each subsequent entry is greater than or equal to the previous entry, and if this is false for even one pair, we say that the array is not sorted.
If an array is sorted, there are many algorithms that can be executed much more efficiently, but it requires more work to verify that an array is sorted, to modify an array that is sorted, and to sort an unsorted array, producing one that is sorted.
An array, whether it is a local array (a local variable declared to be an array with the declaration `typename identifier[capacity];`) or a dynamically allocated array (allocated by calling `new typename[capacity]`), the value of the identifier is an address, and when an array is passed to a function, it is the address of the array that is passed. Consequently, any change to an entry in an array within a function changes the original array that was passed. Thus, if you want to pass an array to a function, but you do not want that function to change the entries of the array, you must declare the parameter to be `const`.

2 Queries on arrays

The three functions we wrote that simply look at the entries of an array include:

- Find the maximum entry of an array.
- Determine if an array is sorted.
- Find if a value is in an array.
- Find if a value is in a sorted array (binary search).

2.1 Finding the maximum entry of an array

To find the maximum entry of an array, we assume that the maximum entry is at `std::size index{0};`. We then proceed to check each subsequent entry, and if a subsequent entry is greater than the entry at `index`, we update that index variable. We then return the index and the user can that access the array at that index to determine the actual maximum.

```
std::size_t find_maximum(
    double      const array[],
    std::size_t const 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;
}
```

Of course, if the array is sorted, the maximum entry is the last entry of the array, so at index `capacity - 1`.

2.2 Determining if an array is sorted

To determine if an array is sorted, we must compare each pair of entries, so indices `0` and `1`, `1` and `2`, all the way to `capacity - 2` and `capacity - 1`. Notice that the second index we are checking starts from `1` and goes to `capacity - 1`, so this be a for loop:

```
for ( std::size_t k{ 1 }; k < capacity; ++k ) {
```

We then simply check if `array[k - 1] <= array[k]`, and if this is ever false, the array is not sorted. Remember that `!(array[k - 1] <= array[k])` is logically equivalent to Remember that `array[k - 1] > array[k]`. We could have this function simply return `true` or `false`, but this doesn't provide much information to the program calling this function. Instead, we will return the index of the first entry that is unsorted, and if the array is sorted, we will return the capacity. Thus, rather than checking if this function returns true if the array is sorted, we will check if this function returns the capacity.

```
std::size_t is_sorted(
    double      const array[],
    std::size_t const capacity
```

```
) {
    for ( std::size_t k{1}; k < capacity; ++k ) {
        if ( array[k - 1] > array[k] ) {
            return k;
        }
    }

    return capacity;
}
```

Thus, if all a program cares about is if the array is sorted or not, the code could look like the following:

```
if ( is_sorted( data, cap ) == cap ) {
    // Code to execute if the array is sorted
} else {
    // Code to execute if the array is not sorted
}
```

2.3 Finding an entry in an array

If we want to find if there is an entry in an array that contains a specific value, we must search through each entry of the array and compare each entry with the value we are searching for. We could simply return `true` or `false`, but this gives little useful information to the program calling this function, so instead, if we find the value, we will return the index of the first entry that contains that value, and if we do not find the value, we will return the array capacity.

```
std::size_t find(
    double      const array[],
    std::size_t const capacity,
    double      const value
) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        if ( array[k] == value ) {
            return k;
        }
    }

    return capacity;
}
```

Thus, code calling this function would check:

```
std::size_t position{ find( data, cap, some_value ) };

if ( position == cap ) {
```

```
    // Code to execute if the value was not found
} else {
    // Code to execute give that the value was
    // found at index 'position'
}
```

2.4 Finding an entry in a sorted array (binary search)

A binary search checks the middle entry of an array, and if that entry equals what we are searching for, we return that index. If what we are searching for is less than that index, we now only have to search the first half of the array, and if what we are searching for is greater (the only other possibility), we now only have to search the second half. However, there are some annoying implementation details.

```
std::size_t binary_search(
    double      const array[],
    std::size_t const capacity,
    double      const value
) {
    // During testing, ensure that the
    // array is indeed sorted...
    assert( is_sorted( array, capacity ) == capacity );

    std::size_t left{0};
    std::size_t right{capacity - 1};

    while ( (left <= right) && (right < capacity) ) {
        std::size_t middle{ (left + right)/2 };

        if ( array[middle] == value ) {
            return middle;
        } else if ( array[middle] < value ) {
            left = middle + 1;
        } else {
            right = middle - 1;
        }
    }

    // If we did not find the value, return 'capacity'
    return capacity;
}
```

Not on the examination

In the above code, we calculate `(left + right)/2`; however this might result in an overflow (when a carry occurs when adding the most significant bits) when we add `left + right`, which would then give the wrong answer. Thus, it is preferable to use `left + (right - left)/2`, because we know that `right >= left`, so the cannot be an issue when calculating `right - left`.

Not on the examination

Instead of always checking if `right < capacity`, we can just ensure we are never searching for something less than the first entry of the array:

```
std::size_t binary_search(
    double      const array[],
    std::size_t const capacity,
    double      const value
) {
    // During testing, ensure that the
    // array is indeed sorted...
    assert( is_sorted( array, capacity ) == capacity );

    // If what we are searching for is less
    // than the first entry, just return 'capacity'
    if ( value < array[0] ) {
        return capacity;
    }

    std::size_t left{0};
    std::size_t right{capacity - 1};

    while (left <= right) {
        std::size_t middle{ (left + right)/2 };

        if ( array[middle] == value ) {
            return middle;
        } else if ( array[middle] < value ) {
            left = middle + 1;
        } else {
            right = middle - 1;
        }
    }

    // If we did not find the value, return 'capacity'
    return capacity;
}
```

3 Modifying an array

We looked at three algorithms that modify the entries of an array:

1. Insert a value into an existing sorted array.
2. Sort an array using selection sort.

3. Sort an array using insertion sort.

3.1 Inserting a value into an existing sorted array

To insert a new value into an existing sorted array, we must find where in the existing sorted array the new entry should be, and then move over all entries larger than that new entry to the right by one. To do this, we will assume that the first `capacity - 1` entries (at indices `0` through `capacity - 2`) of a given array are already sorted, and that the new entry we wish to insert into the sorted array is at location `capacity - 1`. This ensures that the array we are inserting the new value into has the required capacity.

```
void insert(
    double            array[],
    std::size_t const capacity
) {
    // During testing, ensure that the
    // first 'capacity - 1' entries are
    // indeed sorted.
    assert( is_sorted( array, capacity - 1 )
                          == (capacity - 1) );

    // Temporarily store the new value
    double new_value{ array[capacity - 1] };

    // We need this index after the loop,
    // so it must be declared outside the loop.
    std::size_t k{ capacity - 1 };

    for ( ; (k > 0) && (array[k - 1] > new_value); --k ) {
        // If the entry is greater than the new value,
        // move it to the right by one index.
        array[k] = array[k - 1];
    }

    array[k] = new_value;
}
```

Notice that we use short-circuit evaluation in the condition of the for-loop: if `k == 0`, then the first comparison returns `false`, so we do not even evaluate the second comparison Notice that we use short-circuit evaluation in the condition of the for-loop: if `k == 0`, then the first comparison returns `false`, so we do not even evaluate the second comparison: this is critical, because if the second evaluation was performed, then we would be accessing `array[-1]`, which is outside the bounds of the array. Note that if the first condition is `false`, then in the previous step, we copied what was at index `0` to `array[1]`, meaning that

the new value was smaller than all entries in the sorted array, and when the for-loop exists, the new value will be assigned to `array[0]`.

3.2 Sorting an array using selection sort

Selection sort works as follows:

1. Find the index from `0` to `capacity - 2` of the entry that contains the largest value, and if that entry is greater than `array[capacity - 1]`, swap the two.
2. Find the index from `0` to `capacity - 3` of the entry that contains the largest value, and if that entry is greater than `array[capacity - 2]`, swap the two.
3. Keep doing this until we are left with only the first entry.

Because we already implemented the `std::size_t find_maximum(...)` function, we can using this in our selection sort algorithm. The only other function we require is one to swap two entries; however, here we will simply call the implementation in the standard library.

```
void selection_sort(
    double            array[],
    std::size_t const 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_maximum( 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 );
}
```

If you look at most on-line implementations, you will find that they try to do everything in one function, including finding the maximum entry. This can lead to more difficult bugs, as you must ensure everything is working. By writing a separate helper function `std::size_t find_maximum(...)`, you can ensure that that function works correctly, and this leads to a much simpler structure to the selection sort function.

Not on the examination

This implementation of selection sort is superior to all other sorting algorithms with respect to only one criterion: it is guaranteed to have the minimum number of assignments. This generally is not useful except with these arrays are being stored in solid-state memory where integrity of the memory is paramount (expeditions to outer space, or other remote locations) and the capacity of the arrays being sorted is not too large. It is definitely a reasonable algorithm if the capacity of the array is under 100, and could still be reasonable for larger arrays.

3.3 Sorting an array using insertion sort

Insertion sort works as follows:

1. When considered as an array by itself, the first entry is sorted.
2. Insert the entry at index `1` into the previously sorted array from index `0` to `0` (an array with one entry), resulting in a sorted array with two entries.
3. Insert the entry at index `2` into the previously sorted array from index `0` to `2` (an array with two entries), resulting in a sorted array with three entries.
4. Continue until we have inserted the entry at index `capacity - 1` into the previously sorted array from index `0` to `capacity - 2` (an array with `capacity - 1` entries, resulting in a sorted array with `capacity` entries.

Because we already implemented the `void insert(...)` function, we can using this in our insertion sort algorithm; it is only necessary to remember that the call `insert( array, N )` inserts the entry at `N - 1` into the assumed-to-be-sorted array entries from index `0` through `N - 2`, producing an array that is sorted of capacity `N`.

```
void insertion_sort(
    double            array[],
    std::size_t const capacity
) {
    for ( std::size_t k{2}; k <= capacity; ++k ) {
        // Insert the entry at 'k - 1' into
        // the array entries assumed to be sorted
        // at entries '0' through 'k - 2'.
        insert( array, k );
    }

    // During testing, ensure that the
    // resulting array is actually sorted.
    assert( is_sorted( array, capacity ) == capacity );
}
```

Not on the examination

This implementation of insertion sort is superior to all other sorting algorithms if the array is almost sorted, meaning that there are only a few entries (relatively speaking) that are out of place. It does, however, have more writes than selection sort.
This algorithm, however, is much slower than many other sorting algorithms that you will learn in second year.