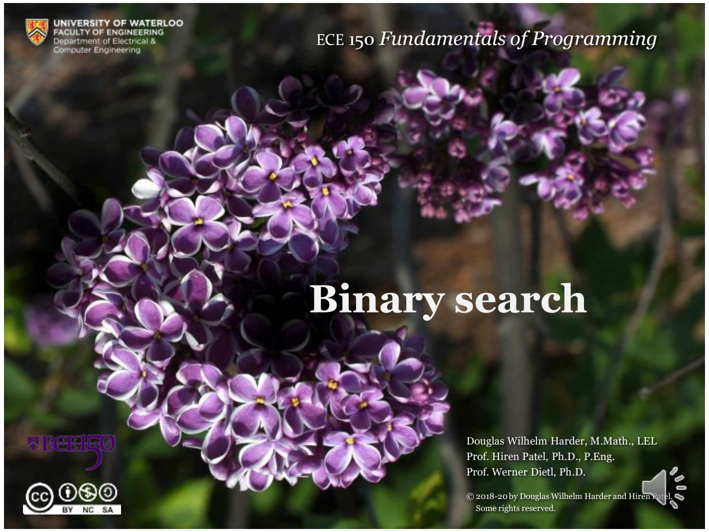

Outline

- In this lesson, we will:
  - Describe a binary search in terms of the high-low game
  - Implement the binary search
  - Consider the conditions necessary to stop looping
    - We will be looking at different cases

Searching sorted arrays

- Suppose an array is sorted,
  and you'd like to find an entry in that array
  - You could use a linear search from a previous topic
- A linear search is necessary if an array is not sorted,
  for the entry you're looking for may be anywhere
- Question: can we speed up the search if the array is sorted?
  - Imagine if you had a book, and you had to find page 147
    - If the book was 308 pages, would you start with page 1?

Searching sorted arrays

- Remember the high-low game:
  - Your friend has thought of a number between `0` and `99`,
    and you have to guess that number
- Suppose you guess `(0 + 99)/2 == 49`
  - In the off chance you're correct, great!
  - If your friend says "low", you're too low,
    so your next guess is `(50 + 99)/2 == 74`
  - If your friend says "high", you're too high,
    so your next guess is `(0 + 48)/2 == 24`
  - Note that we are using integer division, as used in C++
- Can we use this strategy with searching a sorted array?

Implementing the binary search

- Like our linear search,
  the binary search will have a similar function declaration

```
std::size_t binary_search( double const array[],
                           std::size_t const capacity,
                           double const value );
```


Searching sorted arrays

- Suppose you are searching an array with capacity `100` for `9237.1`
  - The value could be anywhere between index `0` and `99`
- Suppose you guess the index `(0 + 99)/2 == 49`
  - Check if `array[49] == value`
    - In the off chance this entry equals the value, great!
  - Check if `array[49] < 9237.1`
    - We're too low, so we are restricted to `50` to `99`
    - Next time, inspect index `(50 + 99)/2 == 74`
  - Otherwise, `array[49] > 9237.1`
    - We're too high, so we are restricted to `0` to `48`
    - Next time, inspect index `(0 + 48)/2 == 24`

Implementing the binary search

- We will start by checking the array is sorted,
  and setting the bounds

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

std::size_t lower_index{0};
std::size_t upper_index{capacity - 1};
```


Implementing the binary search

- We cannot use a for loop, so we'll have to use a while loop
  - We'll hold off on the condition

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

std::size_t lower_index{0};
std::size_t upper_index{capacity - 1};

while ( ... ) {

}
```

Implementing the binary search

- First, we will calculate the average of the lower and upper indices

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

std::size_t lower_index{0};
std::size_t upper_index{capacity - 1};

while ( ... ) {
    std::size_t average_index{ (lower_index + upper_index)/2 };






}
```


Implementing the binary search

- Next, we will check if this entry contains what we are searching for

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

std::size_t lower_index{0};
std::size_t upper_index{capacity - 1};

while ( ... ) {
    std::size_t average_index{ (lower_index + upper_index)/2 };

    if ( array[average_index] == value ) {
        return average_index;
    }



}
```


Implementing the binary search

- If the entry at this point is too small
  - We'll update the lower index

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

std::size_t lower_index{0};
std::size_t upper_index{capacity - 1};

while ( ... ) {
    std::size_t average_index{ (lower_index + upper_index)/2 };

    if ( array[average_index] == value ) {
        return average_index;
    } else if ( array[average_index] < value ) {
        lower_index = average_index + 1;
    }


}
```


Implementing the binary search

- Otherwise, the only other possibility is that the entry is too large
  - We'll update the upper index

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

std::size_t lower_index{0};
std::size_t upper_index{capacity - 1};

while ( ... ) {
    std::size_t average_index{ (lower_index + upper_index)/2 };

    if ( array[average_index] == value ) {
        return average_index;
    } else if ( array[average_index] < value ) {
        lower_index = average_index + 1;
    } else {
        upper_index = average_index - 1;
    }
}
```

While loop condition

- Now for the more interesting question:
  – When do we stop searching?

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

std::size_t lower_index{0};
std::size_t upper_index{capacity - 1};

while ( ... ) {
    std::size_t average_index{ (lower_index + upper_index)/2 };

    if ( array[average_index] == value ) {
        return average_index;
    } else if ( array[average_index] < value ) {
        lower_index = average_index + 1;
    } else {
        upper_index = average_index - 1;
    }
}
```


While loop condition

- Question: What is the necessary condition for the while loop?
  – If we are searching for something in the array, no halting condition is necessary
    - This algorithm is guaranteed to find the entry
- Thus, the condition would be

```
while ( true ) {
    // Find the average and update accordingly
}
```


While loop condition

- Question: What is the necessary halting condition?
  – What if we are searching for something not in the array?
- With every iteration of the loop where we don't find the value, either
  – The value of `lower_index` will be increasing, or
  – The value of `upper_index` will be decreasing

While loop condition

- Question: What is the necessary halting condition?
  – What if we are searching for something not in the array?

Case 1

- Suppose that at one step, `lower_index == upper_index`
  – If this is the case, `average_index` will equal this value, so if `array[average_index] < value`,
    `uppex_index == lower_index - 1`
    and if `array[average_index] > value`,
    `lower_index == upper_index + 1`
- In either case, `lower_index > upper_index`

While loop condition

- Question: What is the necessary halting condition?
  – What if we are searching for something not in the array?

Case 2

- Suppose that at one step, `lower_index + 1 == upper_index`
  – If this is the case, `average_index` will equal `lower_index`,
    so if `array[average_index] < value`,
    `uppex_index == lower_index - 1`
    and if `array[average_index] > value`,
    `lower_index == upper_index`

- In the first case, `lower_index > upper_index`
  and in the second, we are back to Case 1

While loop condition

- Question: What is the necessary halting condition?
  – What if we are searching for something not in the array?

Case 3

- Suppose that at one step, `lower_index + 2 == upper_index`
  – If this is the case, `average_index` will equal `lower_index + 1`,
    so if `array[average_index] < value`,
    `uppex_index == lower_index`
    and if `array[average_index] > value`,
    `lower_index == upper_index`

- In both possibilities, we are back to Case 1

While loop condition

- Thus, we should stop looping as soon as
  `lower_index > upper_index`

- Thus, we should continue iterating so long as:

```
while ( lower_index <= upper_index ) {
    // Find the average and update accordingly
}

return capacity;
```


Our binary search

```
std::size_t binary_search( double const array[], std::size_t const capacity,
                           double const value ) {
    assert( is_sorted( array, capacity ) == capacity );

    std::size_t lower_index{0};
    std::size_t upper_index{capacity - 1};

    while ( lower_index <= upper_index ) {
        std::size_t average_index{ (lower_index + upper_index)/2 };

        if ( array[average_index] == value ) {
            return average_index;
        } else if ( array[average_index] < value ) {
            lower_index = average_index + 1;
        } else {
            upper_index = average_index - 1;
        }
    }

    return capacity;
}
```

A test

- Here is a test:

```
int main() {
    std::size_t const CAPACITY{5};
    double data[CAPACITY]{ 0.0, 1.2, 1.5, 2.7, 4.6 };

    for ( int k{0}; k <= CAPACITY*10; ++k ) {
        double x{ k/10.0 };

        std::size_t index{ binary_search( data, CAPACITY, x ) };

        assert( (0 <= index) && (index <= CAPACITY) );

        if ( index == CAPACITY ) {
            std::cout << x << " not found" << std::endl;
        } else {
            std::cout << x << "\t== data[" << index << "] = "
                      << data[index] << std::endl;
        }
    }

    return 0;
}
```


A test

- Here is the output:

```
0      == data[0] = 0        2 not found              4 not found
0.1 not found                2.1 not found            4.1 not found
0.2 not found                2.2 not found            4.2 not found
0.3 not found                2.3 not found            4.3 not found
0.4 not found                2.4 not found            4.4 not found
0.5 not found                2.5 not found            4.5 not found
0.6 not found                2.6 not found            4.6      == data[4] = 4.6
0.7 not found                2.7      == data[3] = 2.7  4.7 not found
0.8 not found                2.8 not found            4.8 not found
0.9 not found                2.9 not found            4.9 not found
1 not found                  3 not found              5 not found
1.1 not found                3.1 not found
1.2      == data[1] = 1.2    3.2 not found
1.3 not found                3.3 not found
1.4 not found                3.4 not found
1.5      == data[2] = 1.5    3.5 not found
1.6 not found                3.6 not found
1.7 not found                3.7 not found
1.8 not found                3.8 not found
1.9 not found                3.9 not found
```

  – Note that each value was found in the appropriate location:

```
double data[5]{ 0.0, 1.2, 1.5, 2.7, 4.6 };
```


Weakness in our test

- Did any of you notice a weakness in the test?

```
int main() {
    std::size_t const CAPACITY{5};
    double data[CAPACITY]{ 0.0, 1.2, 1.5, 2.7, 4.6 };

    for ( int k{0}; k <= CAPACITY*10; ++k ) {
        double x{ k/10.0 };
```

We never searched for a value less than the first entry!

Weakness in our test

- We can fix this:

```
int main() {
    std::size_t const CAPACITY{5};
    double data[CAPACITY]{ 0.3, 1.2, 1.5, 2.7, 4.6 };

    for ( int k{0}; k <= CAPACITY*10; ++k ) {
        double x{ k/10.0 };
```

  – This should indicate `0`, `0.1` and `0.2` are not found
  – On my computer, however, I get:

```
Segmentation fault (core dumped)
```

Weakness in our implementation

- If both `lower_index` and `upper_index` equal `0`,
  `average_index = 0`,
  and if `array[0] > value`, then
  `upper_index = 0 - 1;`
- Problem: `std::size_t` is unsigned, so `0 - 1` causes a carry,
  so now `upper_index == 0xffff···f`
- How do we test for this?
  – Note that this maximum value is greater than or equal to the capacity of the array
  – Thus, if `upper_index == 0xffff···f`,
  then `upper_index >= capacity`

Weakness in our implementation

- Thus we should not continue if either of these conditions is `true`:
  `lower_index > upper_index`
  `upper_index >= capacity`
- Thus we should continue if both of these conditions are `false`:
  `lower_index > upper_index`
  `upper_index >= capacity`
- Thus we should continue if both of these conditions are `true`:
  `lower_index <= upper_index`
  `upper_index < capacity`

Our complete implementation

```
std::size_t binary_search( double const array[], std::size_t const capacity,
                           double const value ) {
    assert( is_sorted( array, capacity ) == capacity );

    std::size_t lower_index{0};
    std::size_t upper_index{capacity - 1};

    while ( (lower_index <= upper_index)
         && (upper_index < capacity) ) {
        std::size_t average_index{ (lower_index + upper_index)/2 };

        if ( array[average_index] == value ) {
            return average_index;
        } else if ( array[average_index] < value ) {
            lower_index = average_index + 1;
        } else {
            upper_index = average_index - 1;
        }
    }

    return capacity;
}
```


What type of error is this?

- This error in our function was not a problem with binary search
  – If `upper_index` could take on the value `-1`,
  the implementation would work as expected
  – It is because the index is `unsigned` that this error appears
- We call such an error a *semantic error*
  – We expect integer arithmetic to work as it does in the real world
  – This is not what happens with unsigned integers in C++

Another weakness

- Interesting observation:
  - This fix also fixes the problem if capacity is `0`
  - The initial value of `upper_limit` is `0 - 1`
- We seem to have taken care of two weaknesses with one fix

A better test

- Our test requires that we inspect the output
  - Can we write a test that does not require us to do so?

```
int main() {
    std::size_t const CAPACITY{5};
    double data[CAPACITY]{ 0.3, 1.2, 1.5, 2.7, 4.6 };
    std::size_t found_count{0};

    for ( int k{0}; k <= CAPACITY*10; ++k ) {
        double x{ k/10.0 };

        std::size_t index{ binary_search( data, CAPACITY, x ) };

        assert( (0 <= index) && (index <= CAPACITY) );

        if ( index != CAPACITY ) {
            assert( data[index] == x );
            ++found_count;
        }
    }

    assert( found_count == CAPACITY );

    return 0;
}
```


How often must we search the array?

- Question:

  What is the maximum number of entries of the array that we must inspect?

How often must we search the array?

- Suppose the array has a capacity of 127
  - In this case, we must search entries from `0` to `126`
    - We are searching 127 entries
  - If it is not at index `63`
    - We must search the indices from `0` to `62` or `64` to `126`
  - In both cases, we are restricted to searching 63 entries
    - In the first case, if it is not at index `31`
      - We must search the indices from `0` to `30` or `32` to `62`
    - In the second case, if it is not at index `95`
      - We must search the indices from `64` to `94` or `96` to `126`
  - In all four cases, we are restricted to searching 31 entries
  - Note these values are $2^7 - 1 = 127$, $2^6 - 1 = 63$, $2^5 - 1 = 31$,
  - You may correctly deduce that this pattern will continue:

    $2^4 - 1 = 15$, $2^3 - 1 = 7$, $2^2 - 1 = 3$, $2^1 - 1 = 1$

How often must we search the array?

- In your course on algorithms and data structures,
  you will prove that a binary search will insepct no more than
  $\log_2(n) + 1$
  entries of the array
- A linear search on an array of capacity one million
  may require up to searching one million entries
- A binary search on a sorted array of capacity one million
  will require no more than $\log_2(1000000) + 1 = 20.93156857$
  - That is, inspecting no more than 20 entries of the array

Summary

- Following this presentation, you now:
  - Understand how to implement a binary search
  - Are aware that you must thoroughly test your implementation
  - Understand that there may be issues with implementing the algorithm as described
    - In our case, for an unsigned integer, we calculated `0 - 1`,
      which resulted not in `-1`, but rather at `0xfff···f`
  - Are aware that a binary search is relatively fast compared to a linear search, but the array must be sorted

References

[1] Wikipedia,
https://en.wikipedia.org/wiki/Binary_search
[2] Dictionary of Algorithms and Data Structures (DADS)
https://xlinux.nist.gov/dads/HTML/binarySearch.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

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.