1

Outline

- In this lesson, we will:
  - Implement selection and insertion sort recursively
  - Print a number in binary
  - Implement a binary search recursively
  - Print all subsets of a given set of unique objects
    - Determine all appropriate combinations of jumpers
  - Explore the merge sort recursive algorithm

2

Selection sort

- Let's re-interpret selection sort recursively:
  - If the array has capacity of `0` or `1`, we are done
  - Given an array of capacity $n > 1$:
    - Find the largest entry and swap it with the last entry
    - Apply selection sort to the first $n - 1$ entries

```
void selection_sort( double array[], std::size_t capacity ) {
    if ( capacity <= 1 ) {
        return;
    } else {
        std::size_t max_index{ find_max( array, capacity - 1 ) };

        if ( array[max_index] > array[capacity - 1] ) {
            std::swap( array[max_index], array[capacity - 1] );
        }

        selection_sort( array, capacity - 1 );
    }
}
```

3

Insertion sort

- Let's re-interpret insertion sort recursively:
  - If the array has capacity of `0` or `1`, we are done
  - Given an array of capacity n:
    - Sort the first $n - 1$ entries using insertion sort
    - Insert the last entry into the now-sorted array

- This is also straight-forward:

```
void insertion_sort( double array[], std::size_t capacity ) {
    if ( capacity <= 1 ) {
        return;
    } else {
        insertion_sort( array, capacity - 1 );
        insert( array, capacity );
    }
}
```

4

Printing a number in binary

- Suppose we want to print a number in binary:
  - For example, we know that $n = 13$ in binary is `1101`
  - It is easy to print the least-significant digit:
    `std::cout << (n%2);`
  - But, before we print that last "`1`", we must print "`110`"
  - How do we get the value `110`?
    - Recall that integer division discards the fractional part, so `0b1101/2 == 0b110`
  - Thus,
    - To print `1101`, we must print `110` and then print the `1`
    - To print `110`, we must print `11` and then print the `0`
    - To print `11`, we must print `1` and then print the `1`
    - To print a single bit `1`, we just print that bit

Printing a number in binary

- Thus, the implementation of this function is quite straight-forward:

```
void print_binary( int const n ) {
    if ( n <= 1 ) {
        std::cout << n;
    } else {
        print_binary( n/2 );
        std::cout << (n%2);
    }
}
```

`101010`

Printing a number in binary

- To test this algorithm, we could author the following program:

```
#include <iostream>

// Function declarations
int main();
void print_binary( int const n );

// Function definitions
int main() {
    for ( int k{0}; k <= 17; ++k ) {
        print_binary( k );
    }

    return 0;
}

// Other definitions...
```

Output:

```
0
1
10
11
100
101
110
111
1000
1001
1010
1011
1100
1101
1110
1111
10000
10001
```

Printing a number in binary

- Suppose we test our code with negative numbers:

```
#include <iostream>

// Function declarations
int main();
void print_binary( int const n );

// Function definitions
int main() {
    for ( int k{0}; k <= 17; ++k ) {
        print_binary( -k );
        std::cout << std::endl;
    }

    return 0;
}

// Other definitions...
```

Output:

```
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
-10
-11
-12
-13
-14
-15
-16
-17
```

Printing a number in binary

- What is the problem?

```
void print_binary( int const n ) {
    if ( (n<=10) || (n == 1) ) {
        std::cout << n;
    } else {
        print_binary( n/2 );
        std::cout << (n%2);
    }
}
```

Printing a number in binary

- Suppose we test our code with negative numbers:

```
#include <iostream>

// Function declarations
int main();
void print_binary( int const n );

// Function definitions
int main() {
    for ( int k{0}; k <= 17; ++k ) {
        print_binary( -k );
        std::cout << std::endl;
    }

    return 0;
}

// Other definitions...
```

Output:

```
0
0-1
0-10
0-1-1
0-100
0-10-1
0-1-10
0-1-1-1
0-1000
0-100-1
0-10-10
0-10-1-1
0-1-100
0-1-10-1
0-1-1-10
0-1-1-1-1
0-10000
0-1000-1
```

Printing a number in binary

- Question: Why are we getting `-1` printed?
  - You will recall that `n%2` is that number such that

```
n == (2*(n/2) + (n%2))
```

  - For example,

```
 7 = 2*  3  + 1
 6 = 2*  3  + 0
 6 = 2*(-3) + 0
-7 = 2*(-3) - 1
```

Output:

```
0
0-1
0-10
0-1-1
0-100
0-10-1
0-1-10
0-1-1-1
0-1000
0-100-1
0-10-10
0-10-1-1
0-1-100
0-1-10-1
0-1-1-10
0-1-1-1-1
0-10000
0-1000-1
```

Printing a number in binary

- Question: How should we print –13 in binary?
  - The most reasonable format would be `-1101`
  - Thus, to print –13, print "`-`" and then print 13

```
void print_binary( int const n ) {
    if ( n < 0 ) {
        std::cout << "-";
        print_binary( -n );
    } else if ( (n == 0) || (n == 1) ) {
        std::cout << n;
    } else {
        print_binary( n/2 );
        std::cout << (n%2);
    }
}
```

Printing a number in binary

- Here is an interesting case:

```
int main() {
    //         31
    // Print -2  -- the largest possible negative integer
    //  - it should print as -10000000000000000000000000000000
    print_binary( -2147483648 );

    return 0;
}
```

- The output may be interesting; either
  Nothing

```
--------------------------------------------------------------
--------------------------------------------------------------
--------------------------------------------------------------
--------------------------------------------------------------
--------------------------------------------------------------
--------------------------------------------------------------
--------------------------------------------------------------
```


Binary search

- Recall our function declaration of the binary search:

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

- The behavior is as follows:
  - If the value is found, the location is returned
    - An index between `0` and `capacity - 1`
  - Otherwise, `capacity` is returned

Binary search

- Let us describe binary search recursively:
  - If the array has capacity two or less,
    - Just check the existing entries
  - If the capacity is three or greater,
    check the middle entry
    - If the middle entry equals the value, return its index
    - If the middle entry is greater than the value,
      recurse on the array up to the middle entry
    - Otherwise, the middle entry is less than the value,
      recurse on the array starting at the next entry

0	1	2	3	4	5	6	7	8	9	10	11	12
0.3	0.9	1.2	1.9	2.3	2.5	2.8	3.5	3.7	4.5	4.9	6.3	9.5

Binary search

- Implementing the base case is easy:

```
std::size_t binary_search( double array[], std::size_t capacity,
                           double value ) {
    if ( capacity <= 2 ) {
        for ( std::size_t k{0}; k < capacity; ++k ) {
            if ( array[k] == value ) {
                return k;
            }
        }

        return capacity;
    } else {
        // recursive case
    }
}
```

Binary search

- Next, we can find and check the middle entry:

```
std::size_t binary_search( double array[], std::size_t capacity,
                           double value ) {
    if ( capacity <= 2 ) {
        // Base case...
    } else {
        std::size_t middle_index{ capacity/2 };

        if ( array[middle_index] == value ) {
            return middle_index;
        } else ...
```

Binary search

- How will we recurse if `array[middle_index] != value`?
  - To determine the correct approach, let us suppose the capacity is 9:

0	1	2	3	4	5	6	7	8
2.3	3.7	5.6	6.1	7.0	8.4	9.5	10.9	11.2

  - We can calculate the middle index with `capacity/2`
    - In this case, the middle index is 4

Binary search

- Suppose we find that `value < array[middle_index]`:

0	1	2	3	4	5	6	7	8
2.3	3.7	5.6	6.1	7.0	8.4	9.5	10.9	11.2

  - The appropriate call is therefore

```
binary_search( array, middle_index, value );
```

- Now, suppose we were searching for the entry 3.7
  - This call would return 1
  - This is the appropriate value to return
- Now, suppose we were searching for the entry 4.6
  - The value is not found, so 4 is returned
  - But to flag that an entry is not found in an array of capacity 9, we must return 9

Binary search

- We simply have to check the value returned

```
if ( capacity <= 2 ) {
    // Base case...
} else {
    std::size_t middle_index{ capacity/2 };

    if ( array[middle_index] == value ) {
        return middle_index;
    } else if ( value < array[middle_index] ) {
        std::size_t returned_index{ binary_search( array, middle_index,
                                                   value ) };

        if ( returned_index == middle_index ) {
            return capacity;
        } else {
            return returned_index;
        }
    } else {
```

Binary search

- Suppose we find that `array[middle_index] < value`:

  - We want to search the second half the array
    - The first entry is at `5`
      - That is, `middle_index + 1`
    - The capacity is `4`
      - That is, `capacity - middle_index - 1`
  - The address of the entry at index `5` is either:
    `&array[5]` or `&array[middle_index + 1]`
    `array + 5` or `array + middle_index + 1`
  - The appropriate call is therefore

```
binary_search( array + middle_index + 1,
               capacity - middle_index - 1,
               value );
```


Binary search

- Suppose we make this recursive call:


```
binary_search( array + middle_index + 1,
               capacity - middle_index - 1,
               value );
```

  - What do we do with the returned index?
- Now, suppose we were searching for the entry 10.9
  - This call would return `2`
  - We want to return `7`, so this would be `2 + middle_index + 1`
- Now, suppose we were searching for the entry 12.6
  - The value is not found, so `4` is returned
  - But to flag that an entry is not found in an array of capacity `9`, we must return `4 + middle_index + 1`
- In both cases, we add `middle_index + 1` to the returned index

Binary search

- Thus, for the alternative body:

```
std::size_t middle_index{ capacity/2 };

if ( array[middle_index] == value ) {
    return middle_index;
} else if ( value < array[middle_index] ) {
    std::size_t returned_index{ binary_search( array, middle_index,
                                               value ) };

    if ( returned_index == middle_index ) {
        return capacity;
    } else {
        return returned_index;
    }
} else {
    return binary_search( array + middle_index + 1,
                          capacity - middle_index - 1,
                          value ) + middle_index + 1;

}
```


Binary search

```
std::size_t binary_search( double array[], std::size_t capacity, double value ) {
    if ( capacity <= 2 ) {
        for ( std::size_t k{0}; k < capacity; ++k ) {
            if ( array[k] == value ) {
                return k;
            }
        }

        return capacity;
    } else {
        std::size_t middle_index{ capacity/2 };

        if ( array[middle_index] == value ) {
            return middle_index;
        } else if ( value < array[middle_index] ) {
            std::size_t returned_index{ binary_search( array, middle_index, value ) };

            if ( returned_index == middle_index ) {
                return capacity;
            } else {
                return returned_index;
            }
        } else {
            return binary_search( array + middle_index + 1,
                                  capacity - middle_index - 1, value ) + middle_index + 1;
        }
    }
}
```

All subsets

- Given the set {1, 2, 3}
  - All subsets include

{}
{1}, {2}, {3}
{1,2}, {1,3}, {2,3}
{1,2,3}

Output:

```
1
2
3
1 2
1 3
2 3
1 2 3
```

- Given the set {1, 2, 3, 4}
  - All subsets include

{}
{1}, {2}, {3}, {4}
{1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4}
{1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}
{1,2,3,4}

Output:

```
1
2
3
4
1 2
1 3
1 4
2 3
2 4
3 4
1 2 3
1 2 4
1 3 4
2 3 4
1 2 3 4
```


All subsets

- Given the set {1, 2, 3, 4, 5}
  - All subsets include

{}
{1}, {2}, {3}, {4}, {5}
{1,2}, {1,3}, {1,4}, {1,5}, {2,3}, {2,4}, {2,5}, {
{1,2,3}, {1,2,4}, {1,2,5}, {1,3,4}, {1,3,5}, {1,4,5}, {2,3,4
{1,2,3,4}, {1,2,3,5}, {1,2,4,5}, {1,3,4,5}
{1,2,3,4,5}

Output:

```
1
2
3
4
5
1 2
1 3
1 4
1 5
2 3
2 4
2 5
3 4
3 5
4 5
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
2 3 4
2 3 5
3 4 5
1 2 3 4
1 2 3 5
1 2 4 5
1 3 4 5
2 3 4 5
1 2 3 4 5
```

5}
5}, {3,4,5}

All subsets

- How print all subsets of the set {1, 2, …, *n*}?
  - Suppose we had an array of *n* Boolean values
  - If the k^{th} entry is `true`, then *k* is the subset
  - Otherwise, it is not
- To print this, we could use:

```
for ( unsigned int k{0}; k < n; ++k ) {
    if ( membership_array[k] ) {
        std::cout << (k + 1) << " ";
    }
}
```

- For example, if the membership array was:

0	1	2	3	4	5	6	7
true	false	false	true	true	false	false	true

this would print `1 4 5 8`

All subsets

- Thus we have a function that has at least the following:

```
void print_subsets( bool membership_array[], std::size_t capacity );
```

  - On the first recursive call,
    we will flag "1" as being printed (`true`) or not (`false`)
    `membership_array[0] = true` or `false`;
  - On the next recursive call,
    we will flag "2" as being printed or not
    `membership_array[1] = true` or `false`;
  - On the n^{th} recursive call,
    we will flag "*n*" as being printed or not
    `membership_array[capacity - 1] = true` or `false`;
  - Finally, with the next recursive call, each array entry has been assigned either `true` or `false`, so we can print the array

All subsets

- What more information must we pass down?
  - Which index are we currently at

```
void print_subsets( bool membership_array[], std::size_t capacity,
                    std::size_t current_index );
```

  - Thus, we must
    - Create an appropriately sized array

```
bool array[n];
```

    - Call

```
print_subsets( array, n, 0 );
```

All subsets

- Thus, we could implement this as follows:

```
void print_subsets( bool membership_array[], std::size_t capacity,
                                             std::size_t current_index ) {
    if ( current_index == capacity ) {
        for ( unsigned int k{0}; k < capacity; ++k ) {
            if ( membership_array[k] ) {
                std::cout << (k + 1) << " ";
            }
        }

        std::cout << std::endl;
    } else {
        membership_array[current_index] = true;
        print_subsets( membership_array, capacity, current_index + 1 );
        membership_array[current_index] = false;
        print_subsets( membership_array, capacity, current_index + 1 );
    }
}
```

All subsets

- The only thing left is to make this easier for the user:
  - The users should not have to allocate an array

```
void print_subsets( unsigned int n ) {
    bool membership_array[n]{};
    print_subsets( membership_array, n, 0 );
}
```

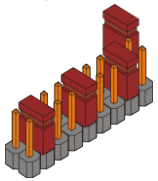
  - Thus, a test for this recursive function may be:

```
int main() {
    for ( unsigned int k{0}; k <= 10; ++k ) {
        std::cout << "-------- " << k << " --------"
                  << std::endl;
        print_subsets( k );
    }

    return 0;
}
```

Jumpers

- A pair of jumper pins on a circuit board (e.g., a motherboard) can be optionally connected with a jumper to configure the circuit board
  - A circuit board may have *n* pairs of jumper pins
  - Each pair can be open or physically shorted by adding a jumper



Jumpers

- Suppose you use a circuit board in your product,
  and certain pairs of jumpers must be open or shorted,
  but the user can use other pairs for additional configurations

- You need to your software for all possible user configurations
  - Let `0` represent open, `1` closed and `?` user configurable

```
1 ? 0 ? 1 1 0 1 ? ? 0 1 0 0 1 0
```

33

Jumpers

- When you test your software,
  you must test it against all possible user configurations
  - Here we see four user-configurable pins

```
"1?0?1101??010010"
```

- All possible configurations include:

```
1000110100010010   1100110100010010
1000110101010010   1100110101010010
1000110110010010   1100110110010010
1000110111010010   1100110111010010
1001110100010010   1101110100010010
1001110101010010   1101110101010010
1001110110010010   1101110110010010
1001110111010010   1101110111010010
```

34

Jumpers

- Your goal is to author a recursive function

```
void print_configurations( char pins[] );
```

  - It takes a C-style string (null-character-terminated) as an argument
    - The characters should only be `0`, `1` or `?`
  - It prints out all possible user configurations
  - You may modify the string entries before you send it recursive

- For example, this program output should match the previous slide:

```
int main() {
    char pin_setup[17]{ "1?0?1101??010010" };
    print_configurations( pin_setup );

    return 0;
}
```

  - The order doesn't matter

35

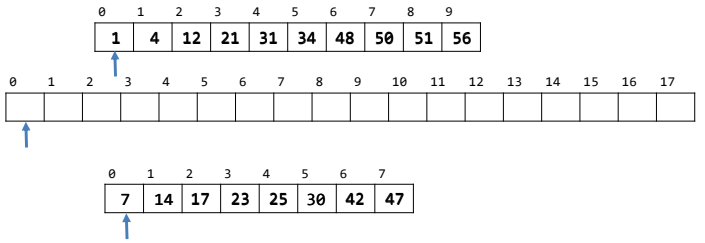
Merging

- Suppose you had two piles of sorted cards:
  - How could you create a single pile of sorted cards?

36

Merging

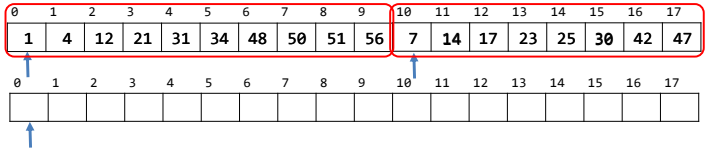
- Suppose you had two sorted arrays
  - How could we create a single sorted array out of these?



37

Merging

- Suppose we had a single array:
  - The first *m* entries of which are sorted
  - The second *n* entries of which are also sorted
- How could we create a single sorted array out of these?



38

Merging

- Here is the implementation of merging one such array

```
void merge( double array[], std::size_t cap_1,
                            std::size_t cap_2 ) {
assert( is_sorted( array, cap_1 ) == cap_1 );
assert( is_sorted( array + cap_1, cap_2 ) == cap_2 );
double tmp_array[cap_1 + cap_2]{};
std::size_t k1{0};
std::size_t k2{cap_1};
std::size_t j{0};
```

Beyond the scope of this course!

```
while ( (k1 < cap_1) && (k2 < cap_1 + cap_2 ) ) {
    if ( array[k1] <= array[k2] ) {
        tmp_array[j] = array[k1];
        ++k1;
    } else {
        tmp_array[j] = array[k2];
        ++k2;
    }

    ++j;
}
```

39

Merging

- Here is the implementation of merging one such array

```
while ( k1 < cap_1 ) {
    tmp_array[j] = array[k1];
    ++k1;
    ++j;
}
```

```
while ( k2 < cap_1 + cap_2 ) {
    tmp_array[j] = array[k2];
    ++k2;
    ++j;
}
```

Beyond the scope of this course!

```
for ( std::size_t k{0}; k < (cap_1 + cap_2); ++k ) {
    array[k] = tmp_array[k];
}

assert( is_sorted( array, cap_1 + cap_2 ) == (cap_1 + cap_2) );
}
```

40

Merge sort

- Now for the most simple recursive sorting algorithm out there:
  - If an array has only one or zero entries in it, it is sorted
  - Otherwise, divide the array into two,
    sort each recursively,
    and then merge the results

Merge sort

- Here is the implementation:

```
void merge_sort( double array[], std::size_t capacity ) {
    if ( capacity <= 1 ) {
        return;
    } else {
        std::size_t capacity_1{ capacity/2 };
        std::size_t capacity_2{ capacity - capacity_1 };

        merge_sort( array, capacity_1 );
        merge_sort( array + capacity_1, capacity_2 );

        merge( array, capacity_1, capacity_2 );
    }
}
```

Summary

- Following this presentation, you now:
  - Understand that implementing selection sort and insertion sort is quite straight-forward
  - Know how to print a number in binary recursively
  - Understand that binary search is also straight-forward to implement recursively
  - Know the problem of finding all subsets of a given set of values
  - Know that this can be used to, for example, find all possible combinations of jumper pins given certain restrictions
  - Are aware of the merge sort algorithm

References

[1] Wikipedia,
https://en.wikipedia.org/wiki/Insertion_sort
https://en.wikipedia.org/wiki/Selection_sort
https://en.wikipedia.org/wiki/Binary_number
https://en.wikipedia.org/wiki/Binary_search_algorithm
https://en.wikipedia.org/wiki/Power_set
https://en.wikipedia.org/wiki/Jumper_(computing)
https://en.wikipedia.org/wiki/Merge_sort

Acknowledgments

None so far.

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