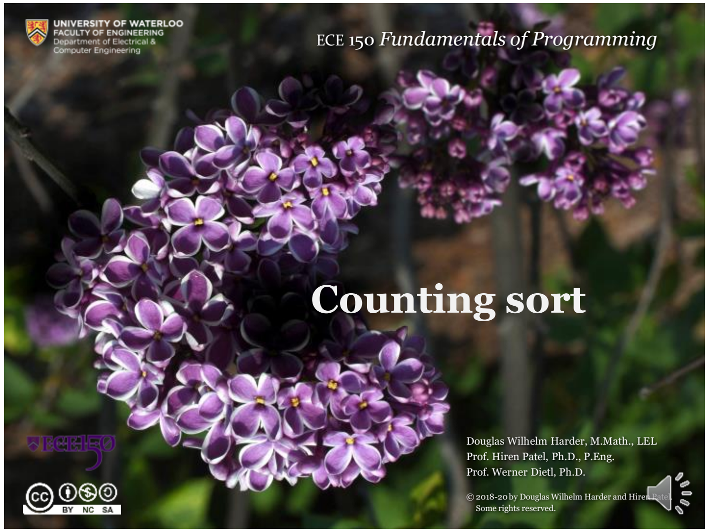


1

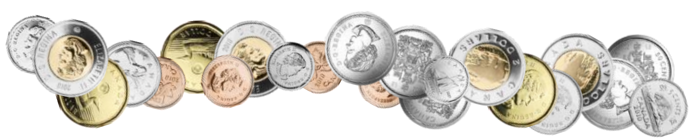
Outline

- In this lesson, we will:
  - Describe a different approach to sorting an array:
    - Counting the number of the different values
  - Consider two implementations
  - Compare and contrast this counting sort with previous sorting algorithms

2

Sorting coins

- Suppose you have sequence of coins and you'd like to sort them
  - Using insertion sort should seem to be unnecessary



  - Much easier to just count the number of pennies, nickels, dimes, quarters, 50-cent pieces, loonies, and toonies

3

Sorting coins

- Why is this faster than sorting the same number of integers?

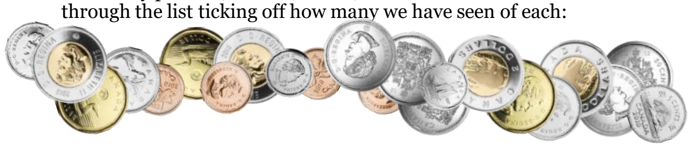
  64 69 41 74 41 73 39 1 2 39 76 22 79 71 79 62 38 82 0 25 20 9

  - The number of possible values is much more limited
  - There are likely to be duplicates
- Suppose we had these same conditions
  - For example, given an array of integers between 0 and $n - 1$
- How could we create a sorted array of a limited number of integers not using insertion or selection sort?

4

Sorting coins

- Going back to our coin example,
  here's probably the wrong way to go about it:
  - Find how many pennies there are
  - Next, find how many nickels there are, etc.
- Instead, the more reasonable approach would be to keep a tally of how many pennies, nickels, dimes, etc. there are and then walk through the list ticking off how many we have seen of each:



Pennies ✓✓✓✓
Nickels ✓✓
Dimes ✓✓
Quarters ✓✓
50-cent pieces ✓✓✓✓✓
Loonies ✓✓✓
Toonies ✓✓✓✓

Sorting coins

- This algorithm is quite straight-forward,
  and does exactly what we did with the coins:
  - Create a counting array of capacity n and initialize the entries to `0`
  - Go through the array to be sorted, and just increment the corresponding entry in your counting array

Counting the number of appearances

- Thus, suppose we are asked to sort this array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
6	1	6	0	5	0	3	4	3	0	9	4	8	9	3	8	1	2	2	8	3	5	6	8

  - Suppose we are made aware that the entries in this array do not exceed the value 9:

0	1	2	3	4	5	6	7	8	9
3	2	2	4	2	2	3	0	4	2

  - Also, at this point, we no longer need to know the capacity of the original array: the capacity equals the sum of our tally
    $3 + 2 + 2 + 4 + 2 + 2 + 3 + 0 + 4 + 2 = 24$

Counting the number of appearances

- Thus, we need to implement this algorithm:
  - Create a counting array of capacity n and initialize the entries to `0`
  - Go through the array to be sorted, and just increment the corresponding entry in your counting array

```
void counting_sort( unsigned int       array[],
                    std::size_t  const capacity,
                    unsigned int const max_value ) {
    unsigned int counting_array[max_value + 1]{};

    for ( std::size_t k{0}; k < capacity; ++k ) {
        ++counting_array[ array[k]];
    }

    // Now re-populate the array with the entries in order
}
```

Filling in the original array

- Thus, given these two arrays, we now must create a sorted array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0	0	1	1	2	2	3	3	3	3	4	4	5	5	6	6	6	8	8	8	8	9	9

0	1	2	3	4	5	6	7	8	9
3	2	2	4	2	2	3	0	4	2

- Previous, we had to swap entries
  - However, now, we proceed as follows:
    - Fill the first three entries with `0`s
    - Fill the next two with `1`s
    - Fill the next two with `2`s
    - Fill the next four with `3`s

  and so on, until the array is full

Filling in the original array

- Take a minute to try to design an algorithm to repopulate the array with the entries in order

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
6	1	6	0	5	0	3	4	3	0	9	4	8	9	3	8	1	2	2	8	3	5	6	8

0	1	2	3	4	5	6	7	8	9
3	2	2	4	2	2	3	0	4	2

```
void counting_sort( unsigned int      array[],
                    std::size_t  const capacity,
                    unsigned int const max_value ) {
    unsigned int counting_array[max_value + 1]{};

    for ( std::size_t k{0}; k < capacity; ++k ) {
        ++counting_array[         ];
    }

    // Now re-populate the array with the entries in order
}
```


Filling in the original array

- Finishing our algorithm:

```
void counting_sort( unsigned int      array[],
                    std::size_t  const capacity,
                    unsigned int const max_value ) {
    unsigned int counting_array[max_value + 1]{};

    for ( std::size_t k{0}; k < capacity; ++k ) {
        ++counting_array[ array[k] ];
    }

    std::size_t posn{0};

    for ( std::size_t k{0}; k <= max_value; ++k ) {
        for ( std::size_t count{0}; count < counting_array[k]; ++count ) {
            array[   posn    ] = k;
            ++posn;
        }
    }

    assert( posn == capacity );
}
```

0	1	2	3	4	5	6	7	8	9
3	2	2	4	2	2	3	0	4	2

Filling in the original array

- Here is another approach:
  - Create a second array, with one extra entry
  - Let the k^{th} entry of this second array be the sum of the entries from `0` to `k - 1` in the counting array

0	1	2	3	4	5	6	7	8	9
3	2	2	4	2	2	3	0	4	2

0	1	2	3	4	5	6	7	8	9	10
0	3	5	7	11	13	15	18	18	22	24

```
unsigned int cumulative_array[max_value + 2]{};

for ( std::size_t k{1}; k < max_value + 2; ++k ) {
    cumulative_array[k] = cumulative_array[k - 1] + counting_array[k- 1];
}

assert( cumulative_array[max_value + 1] == capacity );
```

Filling in the original array

- How do we use this new cumulative array?

0	1	2	3	4	5	6	7	8	9	10
0	3	5	7	11	13	15	18	18	22	24

  - This says:
    - Indices `0` to `2` should be populated with `0`
    - Indices `3` to `4` should be populated with `1`
    - Indices `5` to `6` should be poplated with `2`
    - Indices `7` through `10` should be populated with `3`
  - Note that indices `18` through `17` should be populated with `7`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0	0	1	1	2	2	3	3	3	3	4	4	5	5	6	6	6	8	8	8	8	9	9

Counting the number of appearances

```
void counting_sort( unsigned int       array[],
                    std::size_t  const capacity,
                    unsigned int const max_value ) {
    unsigned int counting_array[max_value + 1]{};

    for ( std::size_t k{0}; k < capacity; ++k ) {
        ++counting_array[array[k]];
    }

    unsigned int cumulative_array[max_value + 2]{};

    for ( std::size_t k{1}; k < max_value + 2; ++k ) {
        cumulative_array[k] = cumulative_array[k - 1] + counting_array[k- 1];
    }

    assert( cumulative_array[max_value + 1] == capacity );

    for ( std::size_t value{0}; value <= max_value; ++value ) {
        for ( std::size_t k{ cumulative_array[value] };
              k < cumulative_array[value + 1]; ++k ) {
            array[k] = value;
        }
    }
}
```


Why two approaches?

- In a sense, the first approach is superior,
  as it doesn't require a second intermediate array
  - The goal here, however, is to demonstrate there are different algorithms, and you may come across a situation where instead of the counting array, you only have the cumulative array

How much faster is counting sort?

- The answer is, of course, it depends
  - For example, which is likely faster?

    ```
    insertion_sort( array, 10 );
    counting_sort(  array, 10, 1000000 );
    ```

  - How about now?

    ```
    insertion_sort( array, 1000000 );
    counting_sort(  array, 1000000, 10 );
    ```

- In your course on algorithms and data structures,
  you will learn about asymptotic and algorithm analysis
  - You may have already seen "big-O" notation in your calculus course

One fundamental difference

- Up to this point, you have been exposed to:
  - Insertion sort
  - Selection sort
  - You may also recall the discussion on merge sort
- What is the fundamental difference between these sorting algorithms and this sorting algorithm?
  - In these first three, we compared values in the array:

```
void insert( double array[], std::size_t capacity ) {
    double value{ array[capacity - 1] };
    std::size_t k{ capacity - 1 };

    for ( ; array[k - 1] > value; --k ) {
        array[k] = array[k - 1];
    }

    array[k] = value;
}
```

  - This counting sort algorithm never compares entries to each other

17

Summary

- Following this presentation, you now:
  - Understand the idea behind a counting sort
  - Have seen two different implementations
    - The first puts the appropriate number of each value into the array
    - The second used a cumulative array to determine what goes where
  - Understand that there are circumstances where this algorithm will be faster than insertion sort, and other circumstances where it will be slower
  - Are aware that this algorithm does not compare the relative values of entries in the array, we simply count what is there

18

References

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

19

Acknowledgments

None so far.

20

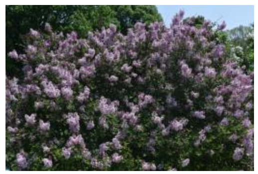
Colophon

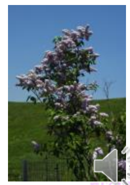
These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

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.