UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

Non-binary partitions

Douglas Wilhelm Harder, M.Math. LEL
Prof. Hiren Patel, Ph.D., P.Eng.
Prof. Werner Dietl, Ph.D.

Outline

- In this lesson, we will:
  - Discuss our previous implementation of partitioning an array
    - That was a binary partition
  - Describe having multiple partitions
    - We will divide the entries into *decades*
  - Look at:
    - A simple and straight-forward implementation
    - A better implementation that doesn't require a second array
    - A faster implementation that requires a second array
    - An even faster implementation that requires only an array of size ten
  - Discuss how design decisions and choice of algorithms can lead to sometimes busier code, but also faster code

Non-binary partitions

- We have seen algorithms for partitioning entries of an array so that all entries satisfying a condition come first, and all entries that do not come second, or vice versa
  - This is a *binary* partition: only two possibilities

- Suppose there multiple partitions:
  - For example, given n numbers in the range $[0, 100)$, partition the numbers so:
    - Those in $[0, 10)$ come first,
    - Those in $[10, 20)$ come next,

    and so on until those in $[90, 100)$ come last
  - Recall that $[a, b)$ includes all numbers x such that $a \leq x < b$

Non-binary partitions

- Here is the function prototype:

```
void decade_partition( double      array[],
                       std::size_t bounds[11],
                       std::size_t capacity );
```

- The behavior of the function is as follows:
  - The array that is passed will be partitioned in place
  - The bounds array will contain indices so that we can iterate through all entries falling between $[10k,\ 10(k+1))$ with

```
for ( std::size_t i{ bounds[k] }; i < bounds[k + 1]; ++i ) {
    std::cout << array[i] << " ";
}
```

Initial approach

- Consider the following approach:
  - Loop through the array and find all entries on [0, 10) and copy them into a new array, recording how many there were
  - Loop again, but now do the same with entries on [10, 20)
  - Repeat this until all decades have been partitioned

- We will need a new array containing as many entries as the original
  - A local variable will store the next location to place a value
- Our outer loop will iterate from 0 to 9:
  - On the i^{th} iteration, it will find all numbers x such that

$$10\,i \leq x < 10\,(i+1)$$

Initial approach

```
void decade_partition( double      array[],
                       std::size_t bounds[11],
                       std::size_t capacity ) {
    double partition[capacity];
    std::size_t next_index{ 0 };

    for ( std::size_t idx{ 0 }; idx < 10; ++idx ) {
        bounds[idx] = next_index;

        for ( std::size_t k{ 0 }; k < capacity; ++k ) {
            if ( (10.0*idx <= array[k]) && (array[k] < 10.0*(idx + 1)) ) {
                partition[next_index] = array[k];
                ++next_index;
            }
        }
    }

    assert( next_index == capacity );
    bounds[10] = capacity;
```

Initial approach

```
    // Copy all the entries back to the original array

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

Using only one array?

- Question: Can you do this without a second array?
  - Suppose that we are checking if `array[k]` should be moved back to position `array[next_index]`
  - In this case, whatever is at `array[next_index]` is not in the correct location
    - How about just swapping them?
    - We could use:

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

    - We will use `std::swap(…)`

Using only one array?

```
void decade_partition( double      array[],
                       std::size_t bounds[10],
                       std::size_t capacity ) {
    std::size_t next_index{ 0 };

    for ( std::size_t idx{ 0 }; idx < 10; ++idx ) {
        bounds[idx] = next_index;

        for ( std::size_t k{ 0 }; k < capacity; ++k ) {
            if ( (10.0*idx <= array[k]) && (array[k] < 10.0*(idx + 1)) ) {
                std::swap( array[next_index], array[k] );
                ++next_index;
            }
        }
    }

    assert( next_index == capacity );
    bounds[10] = capacity;
}
```

Reducing the number of checks?

- Notice that, after the first loop,
  the entries 0 through `next_index - 1` are all their correct location
  - There is no point in checking these again!

- Thus, we really only need start the loop at `next_index`, not `0`:

```
for ( std::size_t idx{ 0 }; idx < 10; ++idx ) {
        bounds[idx] = next_index;

        for ( std::size_t k{ next_index }; k < capacity; ++k ) {
            if ( (10.0*idx <= array[k]) && (array[k] < 10.0*(idx + 1)) ) {
                std::swap( array[next_index], array[k] );
                ++next_index;
            }
        }
    }
```

Reducing the number of checks?

- How does this help us?
  - Suppose we are partitioning an array with capacity n
  - If all the entries are in the first partition, we will only check n entries
  - If all the entries are in the last partition, we will check $10n$ entries
  - Suppose that each partition has approximation 10% of the entries
    - The first time, we will check n
    - Next, 10% are partitioned, so we will check only 90% or $0.9n$
    - Next, 20% are partitioned, so we will only check 80%, and so on…
  - Thus, we will check:

$$\begin{aligned} & n + 0.9n + 0.8n + 0.7n + 0.6n + 0.5n + 0.4n + 0.3n + 0.2n + 0.1n \\ &= (1 + 0.9 + 0.8 + 0.7 + 0.6 + 0.5 + 0.4 + 0.3 + 0.2 + 0.1)n \\ &= 5.5n \end{aligned}$$

  - This is about 50% of the worst-case scenario,
    but engineers must worry about the worst case

Issues with this approach

- How expensive is this algorithm?
  - We must loop through the array up to ten times
- Suppose we wanted to partition such numbers, but on percentiles: [0, 1), [1, 2), [2, 3), …, [98, 99), [99, 100)?
  - We would need to loop through the array up to one hundred times…
- This could get very expensive, very fast…
- Can we do this without a loop inside a loop?
  - Hint: We will use the `bounds` array

Issues with this approach

- Pause this video, and try this on your own
  - Hint: Start by counting how many items fall into each partition
  - From this, can you get entries of the `bounds` array?
  - Can you use the bounds array to build up a partition?

- Try this with

7.5	2.3	4.6	5.7	0.3	2.9	2.2	9.9	7.3	4.4	0.2	4.8	8.8	9.8	3.8	1.7

  - Remember, the first two entries can be `0.2 0.3`, or `0.3 0.2`
    - Order does not matter within a partition

0	1	2	3	4	5	6	7	8	9	10
0	2	3	6	7	10	11	11	13	14	16

1. Counting the entries in the partitions

- First, count the number of entries that fall into each of the partitions

```
// Set all entries to 0
for ( std::size_t idx{ 0 }; idx <= 10; ++idx ) {
    bounds[idx] = 0;
}

// Determine which partition an entry falls in and
// then increment the count for that partition
for ( std::size_t k{ 0 }; k < capacity; ++k ) {
    std::size_t idx{ std::floor( array[k]/10.0 ) };
    ++bounds[idx];
}
```

2. Calculate a running sum

- With 25 items, given that the bounds array is now:

  { 3, 1, 2, 3, 2, 0, 5, 2, 3, 4, 0}

  We must convert this to:

  { 0, 3, 4, 6, 9, 11, 11, 16, 18, 21, 25}

  – Pause and try to do this on your own

```
std::size_t running_sum{ 0 };

for ( std::size_t idx{ 0 }; idx < 10; ++idx ) {
    running_sum += bounds[idx];
    bounds[idx] = running_sum - bounds[idx];
}

assert( running_sum == capacity );
bounds[10] = capacity;
```

2. Calculate a running sum

- With 25 items, given that the bounds array is now:

```
{ 3,  1,  2,  3,  2,  0,  5,  2,  3,  4,  0}
```

We must convert this to:

```
{ 0,  3,  4,  6,  9, 11, 11, 16, 18, 21, 25}
```

  - Sometimes, however, there are clearer implementations

```
bounds[10] = capacity;

for ( std::size_t idx{ 9 }; idx > 0; --idx ) {
    bounds[idx] = bounds[idx + 1] - bounds[idx];
}

assert( bounds[0] == bounds[1] );
bounds[0] = 0;
```

3. Placing items into the correct location

- Consider the array

```
{ 0,  3,  4,  6,  9, 11, 11, 16, 18, 21, 25}
```

  - This says:
    - Items in the first decade belong in `array[0]`, …, `array[2]`
    - The sixth decade is empty
    - Items in the seventh decade belong in `array[11]`, …, `array[15]`
  - Thus, the first item we find in the 7^{th} decade belongs at

```
bounds[6]
```

  - The second in the 7^{th} decade belongs at

```
bounds[6] + 1
```

- Place an item into `bounds[idx]` and increment that value

3. Placing items into the correct location

- Thus, we have:

```
// Copy into a new array of the appropriate size
double partition[capacity];

for ( std::size_t k{ 0 }; k < capacity; ++k ) {
    std::size_t idx{ std::floor( array[k]/10.0 ) };
    partition[bounds[idx]] = array[k];
    ++bounds[idx];
}

assert( bounds[9] == capacity );
```

4. Clean up…

- First, we have to copy the entries back to the original array:

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

- Next, the `bounds` array now looks like:

```
{ 3,  4,  6,  9, 11, 11, 16, 18, 21, 25, 25}
```

  - We must shift these entries back

```
for ( std::size_t idx{ 9 }; idx > 0; --idx ) {
    bounds[idx] = bounds[idx - 1];
}

bounds[0] = 0;
```

Our two approaches

- Compare these two functions:

```
void decade_partition( double      array[],
                       std::size_t bounds[10],
                       std::size_t capacity ) {
    std::size_t next_index{ 0 };
    double partition[capacity];

    for ( std::size_t idx{ 0 }; idx < 10; ++idx ) {
        bounds[idx] = next_index;

        for ( std::size_t k{ next_index }; k < capacity; ++k ) {
            if ( (10.0*idx <= array[k]) && (array[k] < 10.0*(idx + 1)) ) {
                std::swap( partition[next_index], array[k] );
                ++next_index;
            }
        }
    }
}
```

```
void decade_partition( double      array[],
                       std::size_t bounds[11],
                       std::size_t capacity ) {
  for ( std::size_t idx{ 0 }; idx < 10; ++idx ) {
    bounds[idx] = 0;
  }

  for ( std::size_t k{ 0 }; k < capacity; ++k ) {
    std::size_t idx{ std::floor( array[k]/10.0 ) };
    ++bounds[idx];
  }

  bounds[10] = capacity;

  for ( std::size_t idx{ 9 }; idx > 0; --idx ) {
    bounds[idx] = bounds[idx + 1] - bounds[idx];
  }

  bounds[0] = 0;

  double partition[capacity];

  for ( std::size_t k{ 0 }; k < capacity; ++k ) {
    std::size_t idx{ std::floor( array[k]/10.0 ) };
    partition[bounds[idx]] = array[k];
    ++bounds[idx];
  }

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

  for ( std::size_t idx{ 9 }; idx > 0; --idx ) {
    bounds[idx] = bounds[idx - 1];
  }

  bounds[0] = 0;
}
```


Our two approaches

- Why implement a function that is significantly:
  - Longer, and
  - More complex?

- Consider the total number of iterations to partition 10 000 numbers
  - We will count the number of executions of a loop body:
    - The first requires 10 loops of 10 0000, so 100 000 executions
    - The second requires $3 \times 10 + 3 \times 10\,000 = 30\,030$ executions
  - Suppose we were partitioning n numbers into m partitions:
    - The first requires mn executions
    - The second requires $m + n + m + n + n + m = 3(m + n)$ executions
  - If $n = 100\,000$ and $m = 100$ partitions:
    - The first requires 10 000 000 executions
    - The second requires 300 300 executions or about 3%

Our two approaches

- Is the second approach really more complex?
  - Not really, as each individual step is easy and straight-forward

- The real problem with such a multi-step approach is if there is a single bug in any of the one algorithms,
  it may be difficult to isolate exactly where the bug is
  - Solution?
    - Start with a small array where you know what the solution is
    - Work out all the results by hand
    - Make sure the program has the same values

Our two approaches

- For example,

```
{?, ?, ?, ?, ?, ?, ?, ?, ?, ?}
{56.0, 76.0, 3.5, 86.8, 86.7, 96.7, 100.0, 55.6, 36.4, 98.8}
```

  - Initialize the `bounds` array:

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

  - Count the items in each decade:

```
{1, 0, 0, 1, 0, 2, 0, 1, 2, 3, 0}
```

  - Calculate the running sum:

```
{0, 1, 1, 1, 2, 2, 4, 4, 5, 7, 10}
```

  - Place items in the correct location:

```
{3.5, 36.4, 56.0, 55.6, 76.0, 86.8, 86.7, 96.7, 100.0, 98.8}
```

  - The running sum should now look like:

```
{1, 1, 1, 2, 2, 4, 4, 5, 7, 10, 10}
```

    - Revert it back and we're done

Another approach?

- Our algorithm now requires:
  - An additional array with `capacity` entries
  - Three passes through the array:
    - To count, to partition, and to copy back

- Can we do better?
  - We'll step through another approach that requires:
    - An additional array of capacity `10`
    - Only two passes through the array of entries to be partitioned

Another approach?

- Let's make a copy of most of the `bounds[11]` array:
  - Call it `next_index[10]`
- We will use `next_index[k]` to determine where to place the next entry that appears in the k^{th} decade
  - If `next_index[k] != bounds[k + 1]`, the item at that location has not yet been moved to its correct partition
    - Determine where that entry should be (call it `idx`) and then swap `next_index[k]` and `next_index[idx]`, then increment `next_index[idx]`
  - Continue until `next_index[k] == bounds[k + 1]` for all k

Final approach

```
void decade_partition( double      array[],
                       std::size_t bounds[11],
                       std::size_t capacity ) {
    for ( std::size_t idx{ 0 }; idx < 10; ++idx ) {
        bounds[idx] = 0;
    }

    for ( std::size_t k{ 0 }; k < capacity; ++k ) {
        std::size_t idx{ std::round(
            array[k]/10.0
        ) };

        ++bounds[idx];
    }

    bounds[10] = capacity;

    for ( std::size_t idx{ 9 }; idx > 0; --idx ) {
        bounds[idx] = bounds[idx + 1] - bounds[idx];
    }

    bounds[0] = 0;

    // Copy the 'bounds' array
    std::size_t next_index[10];

    for ( std::size_t idx{ 0 }; idx < 10; ++idx ) {
        next_index[idx] = bounds[idx];
    }

    // Keep going until all partitions are full
    for ( std::size_t k{ 0 }; k < 10; ++k ) {
        while ( next_index[k] != bounds[k + 1] ) {
            std::size_t idx{ std::round(
                array[next_index[k]]/10.0
            ) };

            std::swap(
                array[next_index[k]],
                array[next_index[idx]]
            );

            ++next_index[idx];
        }
    }
}
```

Generalization

- We hard-coded the partitioning algorithm into our routine
  - Can we do better?

```
void decade_partition(
    double      array[],
    std::size_t capacity,
    std::size_t bounds[],
    std::size_t num_partitions,
    std::function<std::size_t(double)> to_index );
```

  - The `to_index(...)` function takes a double and returns a partition number between `0` and `num_partitions - 1`

Summary

- Following this lesson, you now
  - Described non-binary partitioning
  - Observed that the most obvious solution is not always the best
  - Seen how there are many different implementations for different algorithms, sometimes even within another algorithm
  - Seen that a more efficient algorithm may not always be more complex
    - The first and second approaches were similar
      - The second did not require a second array
    - The third approach was better than the second, but much longer
    - The last approach was better than the third, and more sussinct

References

[1] https://en.wikipedia.org/wiki/Partition_of_a_set

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.