Closest distance

Consider the following problem:

> Write a function that takes two arguments, an array of double and its capacity. It will return the smallest distance between any two values in the array.

An unsorted list

When finding the minimum of an array, it was easy enough to begin by assuming that the first entry was the minimum:

```
double minimum{ array[0] };

// Check if any entries from array[1] to array[capacity - 1] are
// less than the current value of minimum
for ( std::size_t k{1}; k < capacity; ++k ) {
    if ( array[k] < minimum ) {
        minimum = array[k];
    }
}

return minimum;
```

Now to find the minimum distance between two entries in an array, you have to compare all pairs in the array:

```
double minimum{ some-initial-value };

for ( std::size_t i{0}; i < capacity; ++i ) {
    // We will now compare the distance from array[i] to each other
    // entry in the array
    //  - Of course, we cannot compare array[i] to itself, so we must
    //    be sure to exclude this case
    for ( std::size_t j{0}; j < capacity; ++j ) {
        if ( i != j ) {
            double dist{ std::abs( array[i] - array[j] ) };

            if ( dist < minimum ) {
                minimum = dist;
            }
        }
    }
}

return minimum;
```

Now, the calculation we are making is $|a_i - a_j|$, and from secondary school, you know that the absolute value function is *symmetric*: that is, $|a_i - a_j| = |a_j - a_i|$. Consequently, if we have already calculated $|a_i - a_j|$, there is no reason to calculate $|a_j - a_i|$. We could therefore significantly simplify these loops having the inner loop start at $i + 1$, so we are always guaranteed that $j > i$.

```
double minimum{ some-initial-value };

for ( std::size_t i{0}; i < capacity; ++i ) {
    // We will now compare the distance from array[i] to each other
    // entry in the array
    for ( std::size_t j{i + 1}; j < capacity; ++j ) {
        double dist{ std::abs( array[i] - array[j] ) };

        if ( dist < minimum ) {
            minimum = dist;
        }
    }
}

return minimum;
```

Note that this is only possible if the operation we are performing is symmetric. Suppose you had 20 intersections in the City of Waterloo, and you wanted to find the two intersections that are closest when driving. Initially, you may think that this is a symmetric calculation, but in Waterloo, there is the occasional one-way street; consequently, while for most pairs of intersections, the distance while driving from one to another equals the distance going in the other direction, in some cases, an intermediate street being one way may change the distance significantly. This is therefore not a symmetric calculation, and thus, it would be necessary to calculate the distance from Intersection i to Intersection j and it would also be necessary to calculate the distance from Intersection j to Intersection i.

Next, what do we initialize the local variable `minimum` with? Your initial thought may be to initialize it with the first distance:

```
double minimum{ std::abs( array[1] - array[0] ) };
```

When finding the minimum value in an array, this was acceptable, because we could easily adjust the loop to avoid looking at the first entry a second time: we started the loop at index 1. In this case, however, there is no way to easily change the limits of the loop so as to avoid checking this. You can, as is shown here:

```
double minimum{ std::abs( array[1] - array[0] ) };

for ( std::size_t i{0}; i < capacity; ++i ) {
    // We will now compare the distance from array[i] to each other
    // entry in the array
    for ( std::size_t j{std::min( 2, i + 1 )}; j < capacity; ++j ) {
        double dist{ std::abs( array[i] - array[j] ) };

        if ( dist < minimum ) {
            minimum = dist;
        }
    }
}

return minimum;
```

This is a disaster waiting to happen, because every person who reads this code from now on will probably spend at least two minutes considering how this works, even if you add comments. It is also not obvious that this is the purpose of this code. Finally, with each iteration of the inner loop, you are now making one additional function call, so you are doing a lot of extra work just to avoid comparing the distance between the first two entries. Why is this a disaster? If employees are paid $50 per hour and it costs a company an additional $50 per hour to pay for benefits, equipment, insurance, etc., such obfuscated code may actually end up costing more in human-resource hours than you may ever initially imagine: every minute spent contemplating what this bizarre `std::min( 2, i + 1 )` is supposed to achieve is a waste of $2.

On the other hand, recalculating the first distance is not that much extra work in this case, so perhaps it is best to simply ignore this. Unfortunately, if the calculation being performed is more difficult, say finding the minimum distance between two intersections in a city while driving, then unnecessarily performing this calculation twice may actually be a significant waste of time.

Your next thought may be to just set the initial minimum to a really big number:

```
double minimum{ 123456.0 };    // 123456 m
```

This works great until the distances you are calculating all greater than this value. Code has a lifetime long after its initial authoring, and it will often be repurposed for other tasks in the future, so while this may work for calculating distances between intersections in the City of Waterloo, if we now consider distances in Ontario, it would be necessary to increase this to a larger number, but this would not be done until someone submitted a bug: “Your program, in calculating the distance between the Yonge St. and Bloor St. in Toronto and Rideau St. and Mackenzie Ave. in Ottawa, gives a distance of 123.456 km…” To fix this bug, the developer may first spend a significant amount of time looking at the algorithm, although, an intelligent

developer may look at the result, 123.456 km, and deduce that such a regular number could not have come as a result of a calculation, and thus, may more quickly look at the initial value.

The incorrect fix for this bug is to now set it to something larger:

```
double minimum{ 12345678901234567890.0 }; // 1305 light years...
```

Such magic numbers are undesirable, although the comment here may actually improve morale: anyone reading this will get a good laugh when reading the comment.

However, more seriously, we should find a better initial value. Consider this property:

$$\min\{ a, b, \ldots, z \} = \min\{ a, \min\{b, \ldots, z\} \}$$

That is, the minimum of n values is the minimum of the first value and the minimum of everything else that is being considered. Thus, taking this to the lower extreme,

$$\min\{ a, b, c \} = \min\{ a, \min\{b, c\} \}$$

$$\min\{ a, b \} = \min\{ a, \min\{b\} \}$$

$$\min\{ a \} = \min\{ a, \min\{\} \}$$

Now, the minimum of one value is that value, so $\min\{a\} = a$, but what about $\min\{\}$? Well, we require that $\min\{ a, \min\{\} \}$ equals a, so $\min\{\}$ must take on such a value so that it is guaranteed that $a \leq \min\{\}$, and the only reasonable value therefore ∞ .

Thus, we could find the minimum entry of an array using the following:

```
// Set the minimum to infinity
double minimum{ std::numeric_limits<double>::infinity() };

for ( std::size_t i{0}; i < capacity; ++i ) {
    // We will now compare the distance from array[i] to each other
    // entry in the array
    for ( std::size_t j{i + 1}; j < capacity; ++j ) {
        double dist{ std::abs( array[i] - array[j] ) };

        if ( dist < minimum ) {
            minimum = dist;
        }
    }
}

return minimum;
```

This is satisfying and about as good as one can get if you have no additional information about the array.

A sorted array

Suppose, however, you know the array is sorted. In this case, there is a lot less work:

```
// Set the minimum to infinity
double minimum{ std::numeric_limits<double>::infinity() };

for ( std::size_t i{1}; i < capacity; ++i ) {
    double dist{ array[i] - array[i - 1] };

    if ( dist < minimum ) {
        minimum = dist;
    }
}

return minimum;
```

If the array is sorted, first, we are guaranteed that $|a_3 - a_2| = a_3 - a_2$, so if we always subtract the entry that appears earlier in the array from the one that appears later, we don't need the absolute value function. Also, because, for example, $a_3 \geq a_2$, $a_3 - a_1 \geq a_2 - a_1$ (just subtract a_1 from both sides), we only have to compare entries that are right next to each other in the array.

If the array has 1000 entries, and you know it is sorted, then the above loop requires only 999 comparisons. In the previous implementation where we had two nested for loops, where we do not know if the array is sorted, this would require us to make

$$\binom{1000}{2} = \frac{1000!}{2!998!} = \frac{1000 \cdot 999}{2} = 499500,$$

or half-a-million comparisons. If the array is sorted, there is significantly less work

A reverse-sorted array

A reverse-sorted array is one in which the largest entry appears first, and each subsequent entry in the array is less than or equal to the previous. Rework the previous loop so that it finds the minimum distance between two entries in a reverse-sorted array.

Should you end early?

If you ever find that two entries in the array are equal, should you simply return `0.0`, because once two entries are equal, we are guaranteed that nothing will be smaller than this value. Taking one of the earlier pieces of code, we could achieve this with the following:

```
// Set the minimum to infinity
double minimum{ std::numeric_limits<double>::infinity() };

for ( std::size_t i{0}; i < capacity; ++i ) {
    // We will now compare the distance from array[i] to each other
    // entry in the array
    for ( std::size_t j{i + 1}; j < capacity; ++j ) {
        double dist{ std::abs( array[i] - array[j] ) };

        // If the distance is 0, we are done, as 0 is the minimum
        // possible distance between two entries in the array.
        if ( dist == 0.0 ) {
            return 0.0;
        } else if ( dist < minimum ) {
            minimum = dist;
        }
    }
}

return minimum;
```

Notice that this check must be made every time two entries are compared. Thus, if you were performing this calculation on an array of capacity 1000, you would thus be required to perform this check $\binom{1000}{2} = \frac{1000!}{2!998!} = \frac{1000 \cdot 999}{2} = 499500$, or half-a-million times. If most of the entries of the array are usually guaranteed to be different, this would be a wasted calculate. It would only be useful if:

1. There is a high certainty that two entries in the array will be equal.
2. It is necessary to get an answer as soon as possible.

If the first is false, this additional check will almost always slow down the time that is required to find this minimum distance.

An array that may be sorted or reverse sorted?

Suppose you are asked to author an algorithm that does not assume that the array is sorted or reverse sorted, but does less work if the array is sorted or reverse sorted. Your initial inclination may be to author two additional functions:

```
// Set the minimum to infinity
double minimum{ std::numeric_limits<double>::infinity() };

if ( is_sorted( array, capacity ) == capacity ) {
    // Find the minimum distance under the assumption the array is sorted
} else if ( is_reverse_sorted( array, capacity ) == capacity ) {
    // Find the minimum distance under the
    // assumption the array is reverse sorted
} else {
    // Find the minimum distance under the assumption that the array
    // is not sorted--this requires the two nested loops.
}

return minimum;
```

This may be sufficient, but the two calls to `is_sorted(…)` and `is_reverse_sorted(…)` do require two additional loops, although, is there a way you could avoid these two function calls and determine if the array is either sorted or reverse sorted while you are doing the calculations of finding the minimum distance?

The next page contains this author's implementation. Try to author such a function yourself first, as the struggle of getting a good solution will help you appreciate what this author considers to be a reasonable solution.

Some hints, before you proceed to the next page:

1. Have two Boolean-valued flags, `is_sorted` and `is_reverse_sorted`, and set both of these to `true`, only setting them to `false` if you find evidence that the array is not sorted or not reverse sorted, respectively.
2. Have one loop that simply compares adjacent entries while simultaneously checking for evidence that the array is not sorted or reverse sorted.
3. If you determined that the array is sorted or reverse sorted, then the current minimum you have found must be the overall minimum, so just return it.
4. Otherwise, check the distance between all other pairs of entries.

```
// Set the minimum to infinity
double minimum{ std::numeric_limits<double>::infinity() };
// Assume the array is sorted and reverse sorted
// until we find evidence to the contrary.
bool is_sorted{ true };
bool is_reverse_sorted{ true };

// This loop only compares adjacent entries in the array,
// while simultaneously determining whether or not the array
// is either sorted or reverse sorted.
for ( std::size_t i{1}; i < capacity; ++i ) {
    double dist{};

    // If the next entry is greater than the previous,
    //   the array is not reverse sorted;
    // and if the next entry is less than the previous,
    //   the array is not sorted;
    // otherwise, the adjacent entries are equal, so just return 0.0.
    if ( array[i] > array[i - 1] ) {
        dist = array[i] - array[i - 1];
        is_reverse_sorted = false;
    } else if ( array[i] < array[i - 1] ) {
        dist = array[i - 1] - array[i];
        is_sorted = false;
    } else {
        assert( array[i] == array[i - 1] );
        return 0.0;
    }

    if ( dist < minimum ) {
        minimum = dist;
    }
}

// If either of these flags is still set to true, the currently calculated
// minimum distance is the overall minimum distance, so we are done.
if ( is_sorted || is_reverse_sorted ) {
    return minimum;
}

// Otherwise, we must check all other pairs.
for ( std::size_t i{0}; i < capacity; ++i ) {
    // We have already compared array[i] and array[i + 1],
    // so we can start the inner loop with 'j' starting at 'i + 2'
    for ( std::size_t j{i + 2}; j < capacity; ++j ) {
        double dist{ std::abs( array[i] - array[j] ) };

        if ( dist < minimum ) {
            minimum = dist;
        }
    }
}

return minimum;
```

A comment on the author's solution: in a previous section, we discouraged checking if we ever get a distance of `0.0` and then immediately returning. In the above code, however, we almost get this observation for free, because to determine if the array is either sorted or reverse sorted, we must check both if $a_i > a_{i-1}$ or if $a_i < a_{i-1}$ anyway, so if neither of these is true, $a_i = a_{i-1}$.

Online

Once you have tried this yourself, you can look at the complete solution at `repl.it` with the URL

https://repl.it/@dwharder/Minimum-distance-between-array-entries.