A Quick Refresher on Sets

A set is defined as a collection of unique, ordered data (no duplicates) that is used together for some purpose. Most commonly in C++, this is exemplified in the std::set container.

"std::set is an associative container that contains a sorted set of unique objects of type Key. Sorting is done using the key comparison function Compare. Search, removal, and insertion operations have logarithmic complexity. Sets are usually implemented as red-black trees" --cppreference.com

Example: Two small, sorted sets containing integers.

Set 0:	0	1	2	3	end
Set 1:	-1	0	1	2	end

Intersection of Set 0 and Set 1:

0	1	2	end

Example: Two small, sorted sets that have no intersection:

Set 0:	1	3	5	9	end
Set 1:	2	4	6	8	end

N Sets Intersection Iterator

A method for calculating the intersection of N sorted data sets in linear time.
By Andrew Aldwell and Allan Deutsch

Core methodology:

1. For all sets, obtain an iterator to the beginning element in each set.

0	1	2	3	end
1	3	5	9	end
-1	0	1	3	end

2. Find the first/next common element, advancing iterators towards the max value found so far as a lower bound.

0	1	2	3	end
1	3	5	9	end
-1	0	1	3	end

Return these iterators as a tuple.

Iterator 0	Points at data: 1
Iterator 1	Points at data: 1
Iterator 2	Points at data: 1

3. If a set reaches the end without finding a common element , we can conclude that there are no more intersections to be found.

5	6	7	8	end
1	2	5	9	end
-1	0	1	3	end

4. The user can increment the iterator to find additional intersectional elements.

Additional Info

Algorithm Pseudocode:

```
1   max = *iterators[0];
2   while no iterators are at end:
3     for each iterator:
4       if *iterator < max
5         ++iterator;
6       else if *iterator > max
7         max = *iterator;
8         break;
9       else if iterator is iterators.back
10        return {iterators...};
11  for each iterator:
12    iterator = end;
13  return {iterators...};
```

Background data and potential use cases:

This is an efficient solution to a common search problem: "Find all the items with only these similar traits." It has applications of efficiently finding data in any data-base, game engine objects, online shopping inventory, financial investments, and more.

The algorithm was initially designed for a game engine to iterate over only the objects containing a specified list of components. This is the generalized form. The only constraint of the algorithm is that the data be sorted first.

Where this algorithm really shines is when your sets consist of pairs or similar. One part of the pair can be a tag or ID, using a predicate that only looks at the ID. The other part can be some associated data, which the user can access and/or modify. The use case we have used it for is in a game engine as a sort of sparse Structure of Arrays (SoA).

By using the tag instead of an index, you can create a sparse SoA in which not all the elements contained by it have data in every array. In the engine it was developed for, entities are represented as a composition. The compositions are made from different types with different data, and not all entities need all that data. Rather than having empty or unused memory in all those arrays, we can tag the dense data with IDs corresponding to the entity it belongs to, and iterate over only the elements sharing the same ID. In this manner all similar data is stored contiguously.