11.10. Sorted-Range Algorithms

Sorted-range algorithms require that the source ranges have the elements sorted according to their sorting criterion. These algorithms may have significantly better performance than similar algorithms for unsorted ranges (usually logarithmic instead of linear complexity). You can use these algorithms with iterators that are not random-access iterators. However, in this case, the algorithms have linear complexity because they have to step through the sequence element-by-element. Nevertheless, the number of comparisons may still have logarithmic complexity.

According to the standard, calling these algorithms for sequences that are not sorted on entry results in undefined behavior. However, for most implementations, calling these algorithms also works for unsorted sequences. Nevertheless, to rely on this fact is not portable.

Associative and unordered containers provide special member functions for some of the searching algorithms presented here. When searching for a special value or key, you should use them.

11.10.1. Searching Elements

The following algorithms search certain values in sorted ranges.

Checking Whether One Element Is Present

Click here to view code image

```
bool
binary_search (ForwardIterator beg, ForwardIterator end, const T& value)

bool
binary_search (ForwardIterator beg, ForwardIterator end, const T& value,
               BinaryPredicate op)
```

- Both forms return whether the sorted range [*beg* , *end*) contains an element equal to *value*.
- *op* is an optional binary predicate to be used as the sorting criterion:

  op(*elem1*,*elem2*)

- To obtain the position of an element for which you are searching, use `lower_bound()` , `upper_bound()` , or `equal_range()` (see pages 611 and 613).
- The caller has to ensure that the ranges are sorted according to the sorting criterion on entry.
- Complexity: logarithmic for random-access iterators, linear otherwise (at most, `log(` *numElems* `) +2` comparisons; but for other than random-access iterators, the number of operations to step through the elements is linear, making the total complexity linear).

The following program demonstrates how to use `binary_search()` :

Click here to view code image

```
// algo/binarysearch1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll);

    // check existence of element with value 5
    if (binary_search(coll.cbegin(), coll.cend(), 5)) {
        cout << "5 is present" << endl;
    }
    else {
        cout << "5 is not present" << endl;
    }

    // check existence of element with value 42
    if (binary_search(coll.cbegin(), coll.cend(), 42)) {
        cout << "42 is present" << endl;
    }
    else {
        cout << "42 is not present" << endl;
    }
}
```

The program has the following output:

```
1 2 3 4 5 6 7 8 9
5 is present
42 is not present
```

Checking Whether Several Elements Are Present

Click here to view code image

```
bool
includes (InputIterator1 beg, InputIterator1 end,
          InputIterator2 searchBeg, InputIterator2 searchEnd)

bool
includes (InputIterator1 beg, InputIterator1 end,
          InputIterator2 searchBeg, InputIterator2 searchEnd,
          BinaryPredicate op)
```

- Both forms return whether the sorted range [*beg* , *end*) contains all elements in the sorted range [*searchBeg* , *searchEnd*) . That is, for each element in [*searchBeg* , *searchEnd*) , there must be an equal element in [*beg* , *end*) . If elements in [*searchBeg* , *searchEnd*) are equal, [*beg* , *end*) must contain the same number of elements. Thus, [*searchBeg* , *searchEnd*) must be a subset of [*beg* , *end*) .
- *op* is an optional binary predicate to be used as the sorting criterion:

  op (*elem1* , *elem2*)

- The caller has to ensure that both ranges are sorted according to the same sorting criterion on entry.
- Complexity: linear (at most, `2*(` *numElems* `+` *numSearchElems* `)- 1` comparisons).

The following program demonstrates the usage of `includes()` :

Click here to view code image

```
// algo/includes1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;
    vector<int> search;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll:    ");

    search.push_back(3);
    search.push_back(4);
    search.push_back(7);
    PRINT_ELEMENTS(search,"search:   ");

    // check whether all elements in search are also in coll
    if (includes (coll.cbegin(), coll.cend(),
                  search.cbegin(), search.cend())) {
        cout << "all elements of search are also in coll"
             << endl;
    }
    else {
        cout << "not all elements of search are also in coll"
             << endl;
    }
}
```

The program has the following output:

Click here to view code image

```
coll:   1 2 3 4 5 6 7 8 9
search: 3 4 7
```

```
all elements of search are also in coll
```

Searching First or Last Possible Position

Click here to view code image

```
ForwardIterator
lower_bound (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator
lower_bound (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)

ForwardIterator
upper_bound (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator
upper_bound (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)
```

- `lower_bound()` returns the position of the first element that has a value equal to or greater than *value*. This is the first position where an element with value *value* could get inserted without breaking the sorting of the range `[`*beg*`,`*end*`)`.
- `upper_bound()` returns the position of the first element that has a value greater than *value*. This is the last position where an element with value *value* could get inserted without breaking the sorting of the range `[`*beg*`,`*end*`)`.
- All algorithms return *end* if there is no such value.
- *op* is an optional binary predicate to be used as the sorting criterion:

  op(*elem1*,*elem2*)

- The caller has to ensure that the ranges are sorted according to the sorting criterion on entry.
- To obtain the result from both `lower_bound()` and `upper_bound()`, use `equal_range()`, which returns both (see the next algorithm).
- Associative containers provide equivalent member functions that provide better performance (see Section 8.3.3, page 405).
- Complexity: logarithmic for random-access iterators, linear otherwise (at most, `log(`*numElems*`)+1` comparisons; but for other than random-access iterators, the number of operations to step through the elements is linear, making the total complexity linear).

The following program demonstrates how to use `lower_bound()` and `upper_bound()`:

Click here to view code image

```
// algo/bounds1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    INSERT_ELEMENTS(coll,1,9);
    coll.sort ();
    PRINT_ELEMENTS(coll);

    //print first and last position 5 could get inserted
    auto pos1 = lower_bound (coll.cbegin(), coll.cend(),
                             5);
    auto pos2 = upper_bound (coll.cbegin(), coll.cend(),
                             5);

    cout << "5 could get position "
         << distance(coll.cbegin(),pos1) + 1
         << " up to "
         << distance(coll.cbegin(),pos2) + 1
         << " without breaking the sorting" << endl;

    //insert 3 at the first possible position without breaking the sorting
    coll.insert (lower_bound(coll.begin(),coll.end(),
                             3),
                 3);

    //insert 7 at the last possible position without breaking the sorting
    coll.insert (upper_bound(coll.begin(),coll.end(),
                              7),
                  7);

        PRINT_ELEMENTS(coll);
    }
```

The program has the following output:

[Click here to view code image]

```
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 could get position 9 up to 11 without breaking the sorting
1 1 2 2 3 3 3 4 4 5 5 6 6 7 7 7 8 8 9 9
```

`pos1` and `pos2` have type

```
list<int>::const_iterator
```

Searching First and Last Possible Positions

[Click here to view code image]

```
pair<ForwardIterator,ForwardIterator>
equal_range (ForwardIterator beg, ForwardIterator end, const T& value)

pair<ForwardIterator,ForwardIterator>
equal_range (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)
```

- Both forms return the range of elements that is equal to *value*. The range comprises the first and the last position an element with value *value* could get inserted without breaking the sorting of the range [*beg*,*end*).
- This is equivalent to:

  ```
  make_pair(lower_bound(...),upper_bound(...))
  ```

- *op* is an optional binary predicate to be used as the sorting criterion:

  op(*elem1*,*elem2*)

- The caller has to ensure that the ranges are sorted according to the sorting criterion on entry.
- Associative and unordered containers provide an equivalent member function that has better performance (see Section 8.3.3, page 406).
- Complexity: logarithmic for random-access iterators, linear otherwise (at most, `2*log(`*numElems*`)+1` comparisons; but for other than random-access iterators, the number of operations to step through the elements is linear, making the total complexity linear).

The following program demonstrates how to use `equal_range()`:

[Click here to view code image]

```
// algo/equalrange1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    INSERT_ELEMENTS(coll,1,9);
    coll.sort ();
    PRINT_ELEMENTS(coll);

    //print first and last position 5 could get inserted
    pair<list<int>::const_iterator,list<int>::const_iterator> range;
    range = equal_range (coll.cbegin(), coll.cend(),
                         5);
    cout << "5 could get position "
         << distance(coll.cbegin(),range.first) + 1
         << " up to "
         << distance(coll.cbegin(),range.second) + 1
         << " without breaking the sorting" << endl;
}
```

The program has the following output:

Click here to view code image

```
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 could get position 9 up to 11 without breaking the sorting
```

11.10.2. Merging Elements

The following algorithms merge elements of two ranges. The algorithms process the sum, the union, the intersection, and so on.

Processing the Sum of Two Sorted Sets

Click here to view code image

```
OutputIterator
merge (InputIterator source1Beg, InputIterator source1End,
       InputIterator source2Beg, InputIterator source2End,
       OutputIterator destBeg)

OutputIterator
merge (InputIterator source1Beg, InputIterator source1End,
       InputIterator source2Beg, InputIterator source2End,
       OutputIterator destBeg, BinaryPredicate op)
```

- Both forms merge the elements of the sorted source ranges [*source1Beg,source1End*) and [*source2Beg,source2End*) so that the destination range starting with `destBeg` contains all elements that are in the first source range plus those that are in the second source range. For example, calling `merge()` for

  ```
  1 2 2 4 6 7 7 9
  ```

  and

  ```
  2 2 2 3 6 6 8 9
  ```

  results in

  ```
  1 2 2 2 2 2 3 4 6 6 6 7 7 8 9 9
  ```

- All elements in the destination range are in sorted order.
- Both forms return the position after the last copied element in the destination range (the first element that is not overwritten).
- *op* is an optional binary predicate to be used as the sorting criterion:

  op(*elem1*,*elem2*)

- The source ranges are not modified.
- According to the standard, the caller has to ensure that both source ranges are sorted on entry. However, in most implementations, this algorithm also merges elements of two unsorted source ranges into an unsorted destination range. Nevertheless, for unsorted ranges, you should call `copy()` twice, instead of `merge()`, to be portable.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- The destination range should not overlap the source ranges.
- Lists and forward lists provide a special member function, `merge()`, to merge the elements of two lists (see Section 8.8.1, page 423).
- To ensure that elements that are in both source ranges end up in the destination range only once, use `set_union()` (see page 616).
- To process only the elements that are in both source ranges, use `set_intersection()` (see page 617).
- Complexity: linear (at most, *numElems1* + *numElems2* -1 comparisons).

The following example demonstrates how to use `merge()`:

Click here to view code image

```
// algo/merge1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll1;
    set<int> coll2;
```

```
        // fill both collections with some sorted elements
        INSERT_ELEMENTS(coll1,1,6);
        INSERT_ELEMENTS(coll2,3,8);

        PRINT_ELEMENTS(coll1,"coll1:  ");
        PRINT_ELEMENTS(coll2,"coll2:  ");

        // print merged sequence
        cout << "merged: ";
        merge (coll1.cbegin(), coll1.cend(),
               coll2.cbegin(), coll2.cend(),
               ostream_iterator<int>(cout," "));
        cout << endl;
    }
```

The program has the following output:

```
coll1:  1 2 3 4 5 6
coll2:  3 4 5 6 7 8
merged: 1 2 3 3 4 4 5 5 6 6 7 8
```

See page 620 for another example. It demonstrates how the various algorithms that are provided to combine sorted sequences differ.

Processing the Union of Two Sorted Sets

Click here to view code image

```
OutputIterator
set_union (InputIterator source1Beg, InputIterator source1End,
           InputIterator source2Beg, InputIterator source2End,
           OutputIterator destBeg)

OutputIterator
set_union (InputIterator source1Beg, InputIterator source1End,
           InputIterator source2Beg, InputIterator source2End,
           OutputIterator destBeg, BinaryPredicate op)
```

- Both forms merge the elements of the sorted source ranges [*source1Beg,source1End*) and [*source2Beg,source2End*) so that the destination range starting with `destBeg` contains all elements that are in the first source range, in the second source range, or in both. For example, calling `set_union()` for

  ```
  1 2 2 4 6 7 7 9
  ```

  and

  ```
  2 2 2 3 6 6 8 9
  ```

  results in

  ```
  1 2 2 2 3 4 6 6 7 7 8 9
  ```

- All elements in the destination range are in sorted order.
- Elements that are in both ranges are in the union range only once. However, duplicates are possible if elements occur more than once in one of the source ranges. The number of occurrences of equal elements in the destination range is the maximum of the number of their occurrences in both source ranges.
- Both forms return the position after the last copied element in the destination range (the first element that is not overwritten).
- *op* is an optional binary predicate to be used as the sorting criterion:

  op (*elem1* , *elem2*)

- The source ranges are not modified.
- The caller has to ensure that the ranges are sorted according to the sorting criterion on entry.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- The destination range should not overlap the source ranges.
- To obtain all elements of both source ranges without removing elements that are in both, use `merge()` (see page 614).
- Complexity: linear (at most, `2*(` *numElems1* `+` *numElems2* `)- 1` comparisons).

See page 620 for an example of the use of `set_union()` . This example also demonstrates how it differs from other algorithms that combine elements of two sorted sequences.

Processing the Intersection of Two Sorted Sets

Click here to view code image

```
OutputIterator
set_intersection (InputIterator source1Beg, InputIterator source1End,
                  InputIterator source2Beg, InputIterator source2End,
                  OutputIterator destBeg)

OutputIterator
set_intersection (InputIterator source1Beg, InputIterator source1End,
                  InputIterator source2Beg, InputIterator source2End,
                  OutputIterator destBeg, BinaryPredicate op)
```

- Both forms merge the elements of the sorted source ranges [*source1Beg,source1End*) and [*source2Beg,source2End*) so that the destination range starting with `destBeg` contains all elements that are in both source ranges. For example, calling `set_intersection()` for

  ```
  1 2 2 4 6 7 7 9
  ```

  and

  ```
  2 2 2 3 6 6 8 9
  ```

  results in

  ```
  2 2 6 9
  ```

- All elements in the destination range are in sorted order.
- Duplicates are possible if elements occur more than once in both source ranges. The number of occurrences of equal elements in the destination range is the minimum number of their occurrences in both source ranges.
- Both forms return the position after the last merged element in the destination range.
- *op* is an optional binary predicate to be used as the sorting criterion:

  op (*elem1*, *elem2*)

- The source ranges are not modified.
- The caller has to ensure that the ranges are sorted according to the sorting criterion on entry.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- The destination range should not overlap the source ranges.
- Complexity: linear (at most, `2*(` *numElems1* `+` *numElems2* `)-1` comparisons).

See page 620 for an example of the use of `set_intersection()` . This example also demonstrates how it differs from other algorithms that combine elements of two sorted sequences.

Processing the Difference of Two Sorted Sets

Click here to view code image

```
OutputIterator
set_difference (InputIterator source1Beg, InputIterator source1End,
                InputIterator source2Beg, InputIterator source2End,
                OutputIterator destBeg)

OutputIterator
set_difference (InputIterator source1Beg, InputIterator source1End,
                InputIterator source2Beg, InputIterator source2End,
                OutputIterator destBeg, BinaryPredicate op)
```

- Both forms merge the elements of the sorted source ranges [*source1Beg,source1End*) and [*source2Beg,source2End*) so that the destination range starting with `destBeg` contains all elements that are in the first source range but not in the second source range. For example, calling `set_difference()` for

  ```
  1 2 2 4 6 7 7 9
  ```

  and

  ```
  2 2 2 3 6 6 8 9
  ```

  results in

  ```
  1 4 7 7
  ```

- All elements in the destination range are in sorted order.
- Duplicates are possible if elements occur more than once in the first source range. The number of occurrences of equal elements in

the destination range is the difference between the number of their occurrences in the first source range less the number of occurrences in the second source range. If there are more occurrences in the second source range, the number of occurrences in the destination range is zero.

- Both forms return the position after the last merged element in the destination range.
- *op* is an optional binary predicate to be used as the sorting criterion:

  op (*elem1* , *elem2*)

- The source ranges are not modified.
- The caller has to ensure that the ranges are sorted according to the sorting criterion on entry.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- The destination range should not overlap the source ranges.
- Complexity: linear (at most, `2*(` *numElems1* `+` *numElems2* `)-1` comparisons).

See page 620 for an example of the use of `set_difference()` . This example also demonstrates how it differs from other algorithms that combine elements of two sorted sequences.

Click here to view code image

```
OutputIterator
set_symmetric_difference (InputIterator source1Beg, InputIterator source1End,
                          InputIterator source2Beg, InputIterator source2End,
                          OutputIterator destBeg)

OutputIterator
set_symmetric_difference (InputIterator source1Beg, InputIterator source1End,
                          InputIterator source2Beg, InputIterator source2End,
                          OutputIterator destBeg, BinaryPredicate op)
```

- Both forms merge the elements of the sorted source ranges [*source1Beg,source1End*) and [*source2Beg,source2End*) so that the destination range starting with `destBeg` contains all elements that are either in the first source range or in the second source range but not in both. For example, calling `set_symmetric_difference()` for

  ```
  1 2 2 4 6 7 7 9
  ```

  and

  ```
  2 2 2 3 6 6 8 9
  ```

  results in

  ```
  1 2 3 4 6 7 7 8
  ```

- All elements in the destination range are in sorted order.
- Duplicates are possible if elements occur more than once in one of the source ranges. The number of occurrences of equal elements in the destination range is the difference between the number of their occurrences in the source ranges.
- Both forms return the position after the last merged element in the destination range.
- *op* is an optional binary predicate to be used as the sorting criterion:

  op (*elem1* , *elem2*)

- The source ranges are not modified.
- The caller has to ensure that the ranges are sorted according to the sorting criterion on entry.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- The destination range should not overlap the source ranges.
- Complexity: linear (at most, `2*(` *numElems1* `+` *numElems2* `)-1` comparisons).

See the following subsection for an example of the use of `set_symmetric_difference()` . This example also demonstrates how it differs from other algorithms that combine elements of two sorted sequences.

Example of All Merging Algorithms

The following example compares the various algorithms that combine elements of two sorted source ranges, demonstrating how they work and differ:

Click here to view code image

```
// algo/sorted1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> c1 = { 1, 2, 2, 4, 6, 7, 7, 9 };

    deque<int>  c2 = { 2, 2, 2, 3, 6, 6, 8, 9 };

    // print source ranges
    cout << "c1:                         " ;
    copy (c1.cbegin(), c1.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;
    cout << "c2:                         " ;
    copy (c2.cbegin(), c2.cend(),
          ostream_iterator<int>(cout," "));
    cout << '\n' << endl;

    // sum the ranges by using merge()
    cout << "merge():                    ";
    merge (c1.cbegin(), c1.cend(),
           c2.cbegin(), c2.cend(),
           ostream_iterator<int>(cout," "));
    cout << endl;

    // unite the ranges by using set_union()
    cout << "set_union():                ";
    set_union (c1.cbegin(), c1.cend(),
               c2.cbegin(), c2.cend(),
               ostream_iterator<int>(cout," "));
    cout << endl;

    // intersect the ranges by using set_intersection()
    cout << "set_intersection():       ";
    set_intersection (c1.cbegin(), c1.cend(),
                      c2.cbegin(), c2.cend(),
                      ostream_iterator<int>(cout," "));
    cout << endl;

    // determine elements of first range without elements of second range
    // by using set_difference()
    cout << "set_difference():         ";
    set_difference (c1.cbegin(), c1.cend(),
                    c2.cbegin(), c2.cend(),
                    ostream_iterator<int>(cout," "));
    cout << endl;

    // determine difference the ranges with set_symmetric_difference()
    cout << "set_symmetric_difference(): ";
    set_symmetric_difference (c1.cbegin(), c1.cend(),
                              c2.cbegin(), c2.cend(),
                              ostream_iterator<int>(cout," "));
    cout << endl;
}
```

The program has the following output:

Click here to view code image

```
c1:                          1 2 2 4 6 7 7 9
c2:                          2 2 2 3 6 6 8 9

merge():                     1 2 2 2 2 2 3 4 6 6 6 7 7 8 9 9
set_union():                 1 2 2 2 3 4 6 6 7 7 8 9
set_intersection():          2 2 6 9
set_difference():            1 4 7 7
set_symmetric_difference():  1 2 3 4 6 7 7 8
```

Merging Consecutive Sorted Ranges

Click here to view code image

void
inplace_merge (BidirectionalIterator *beg1*, BidirectionalIterator *end1beg2*,
BidirectionalIterator *end2*)

void
inplace_merge (BidirectionalIterator *beg1*, BidirectionalIterator *end1beg2*,
BidirectionalIterator *end2*, BinaryPredicate *op*)

- Both forms merge the consecutive sorted source ranges [*beg1* , *end1beg2*) and [*end1beg2,end2*) so that the range [*beg1,end2*) contains the elements as a sorted summary range.
- Complexity: linear (*numElems* -1 comparisons) if enough memory available, or n-log-n otherwise (*numElems* *log(*numElems*) comparisons).

The following program demonstrates the use of `inplace_merge()` :

Click here to view code image

```
// algo/inplacemerge1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // insert two sorted sequences
    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(coll,1,8);
    PRINT_ELEMENTS(coll);

    // find beginning of second part (element after 7)
    list<int>::iterator pos;
    pos = find (coll.begin(), coll.end(),    // range
                7);                          // value
    ++pos;

    // merge into one sorted range
    inplace_merge (coll.begin(), pos, coll.end());

    PRINT_ELEMENTS(coll);
}
```

The program has the following output:

```
    1 2 3 4 5 6 7 1 2 3 4 5 6 7 8
    1 1 2 2 3 3 4 4 5 5 6 6 7 7 8
```