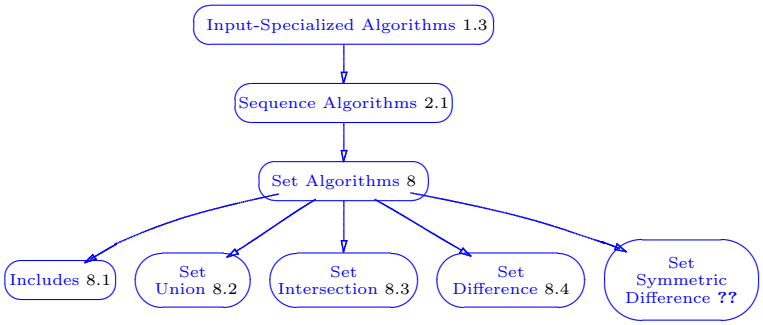
8 Set Algorithms

Section authors: Metin Inanc, Lingling Shen, Rongrong Jiang, Qian Huang



Set algorithms are input-specialized algorithms that deal with sets.

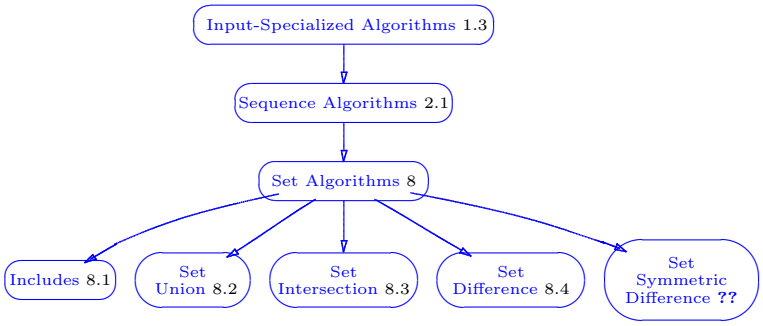
They implement basic mathematical set operations over sets with generic element types. STL implements set containers with red-black trees. The reason for this is that operations with sets require fast and bounded search on set members. This can be achieved with binary search on red-black trees. Red-black trees are one of the ways of getting balanced binary trees and $O(\log N)$ bounded binary search time, the other alternatives being AVL trees and B-trees. AVL trees are better balanced than red-black trees; however, they require more operations to maintain the balance. B-trees would be a better choice with huge sets. Having set elements in binary search trees assures the precondition that all set elements should be sorted. Set algorithms can be applied on container classes other than sets but in this case programmer should take care of the sorting.

There are five set algorithms:

- includes
- set_union
- set_intersection

- set_difference
- set_symmetric_difference

8.1 Includes



Refinement of: Set Algorithms (§8), therefore refinement of Sequence Algorithms §2.1, therefore refinement of Input-Specialized Algorithms §1.3.

Prototype: Includes is overloaded and has two versions:

```
template <class InputIterator1,
          class InputIterator2>
bool includes(InputIterator1 first1,
              InputIterator1 last1,
              InputIterator2 first2,
              InputIterator2 last2);

template <class InputIterator1,
          class InputIterator2,
          class StrictWeakOrdering>
bool includes(InputIterator1 first1,
              InputIterator1 last1,
              InputIterator2 first2,
              InputIterator2 last2,
              StrictWeakOrdering comp);
```

Input: Inputs are two valid ranges of iterators. The range of elements should be sorted in ascending order. This is guaranteed if set container is used.

Output: A boolean indicating whether the second range is included in the first one.

Effects: (Taken from http://www.sgi.com/tech/stl)
Includes tests whether the first sorted range includes the second one.

The semantics of the includes operation are slightly different depending on whether the input contains duplicate keys. When there are no duplicate keys in both the input sets (e.g., $set < Key >$), it returns true if and only if, for every element in [first2, last2), an equivalent element is also present in [first1, last1). When there are duplicate keys in either input sets or both (e.g., $multiset < Key >$), then if a key appears m times in the first set, and n times in the second set (either m or n could be $zero$), it returns false if $m < n$.

The two versions of the includes are different in that the first version uses operator$<$ to compare objects, and the second using a function object $comp$.

Asymptotic Complexity: Lets first give the required bounds from the STL library.

Let $N =$ last1 $-$ first1, $M =$ last2 $-$ first2.

- Worst Case: $O(N + M)$
- Average Case: $O(N + M)$

Complexity in terms of operation counts :

(Average Case) The average case for this algorithm depends on the application at hand. The running time of the algorithm greatly depends on the probability of having one of the elements of the second range not included in the first range. If we define a r.v. X as the rank of the first element in the second range which is not included in the first range, the expected value of X $E[X]$ would be proportional to the running time of the algorithm. Distribution of r.v. X greatly depends on the application which uses includes() function.

(Worst Case) The numbers in the tables below are performed with the worst case input. The second set is completely included in the first set. Thus there is no possibility for the algorithm to encounter a key in the second set which is not in the first set and exit preliminary returning false. Neither of the sets contains duplicated entries.

Includes (N=1000, M=1000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	6	2.002	8.006
value_cnt	0	0	2	2
Total	0.004	6	4.002	10.006

Includes (N=2000, M=1000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	10.99	3.999	14.993
value_cnt	0	0	3.996	3.996
Total	0.004	10.99	7.995	18.989

Includes (N=2000, M=2000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	12	4.002	16.006
value_cnt	0	0	4	4
Total	0.004	12	8.002	20.006

Includes (N=4000, M=1000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	21	8.002	29.006
value_cnt	0	0	8	8
Total	0.004	21	16.002	37.006

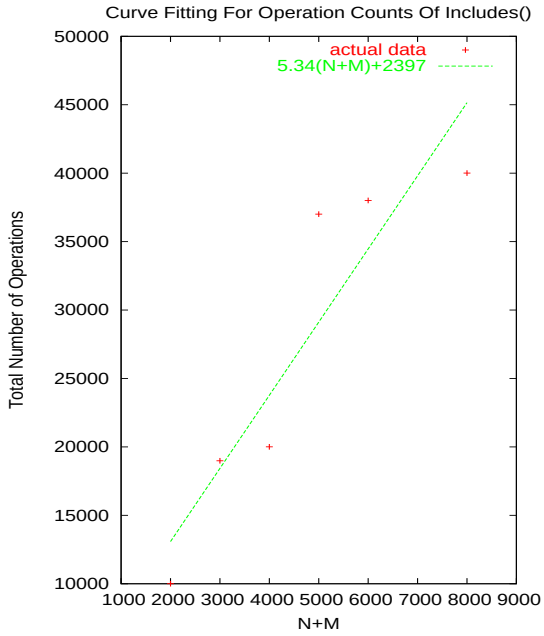
Includes (N=4000, M=2000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	22	8.002	30.006
value_cnt	0	0	8	8
Total	0.004	22	16.002	38.006

Includes (N=4000, M=4000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	24	8.002	32.006
value_cnt	0	0	8	8
Total	0.004	24	16.002	40.006

Curve Fitting for the Total Operation Counts: Curve fitting was performed using polyfit() function of Matlab. All of the figures were drawn with Gnuplot.



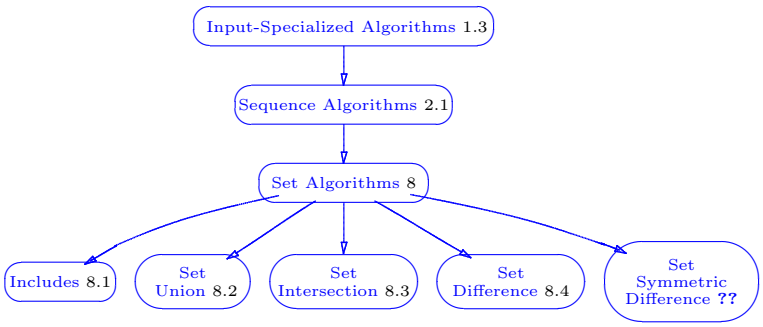
Worst Case Formula: $5.3441(N + M) + 2397.5$

Pseudocode (Actual Code):

```
while (first1 != last1 && first2 != last2)
    if (*first2 < *first1)
       return false;
    else if(*first1 < *first2)
       ++first1;
    else
```

```
        ++first1, ++first2;
    return first2 == last2;
```

8.2 Set Union



Refinement of: Set Algorithms (§8), therefore refinement of Sequence Algorithms §2.1, therefore refinement of Input-Specialized Algorithms §1.3.

Prototype: Set_union is overloaded and has two versions:

```
template <class InputIterator1,
          class InputIterator2,
          class OutputIterator>
OutputIterator
set_union(InputIterator1 first1,
          InputIterator1 last1,
          InputIterator2 first2,
          InputIterator2 last2,
          OutputIterator result);

template <class InputIterator1,
          class InputIterator2,
          class OutputIterator,
          class StrictWeakOrdering>
```

```
OutputIterator
set_union(InputIterator1 first1,
          InputIterator1 last1,
          InputIterator2 first2,
          InputIterator2 last2,
          OutputIterator result,
          StrictWeakOrdering comp);
```

Input: Inputs are two valid ranges of iterators and an output iterator. The range of elements should be sorted in ascending order. This is guaranteed if set container is used.

Output: None. However there are side-effects.

Effects: Set_union outputs a set with the sorted range which is the union of the sorted ranges [first1, last1) and [first2, last2).

The semantics of the union operation are slightly different depending on whether the input contains duplicate keys. When there are no duplicate keys in both the input sets (e.g., $Set < Key >$), set_union's output is a copy of every element that is contained in [first1, last1), [first2, last2), or both. When there are duplicate keys in either input sets or both (e.g., $Multiset < Key >$), then if a key appears m times in the first set, and n times in the second set (either m or n could be $zero$), it appears $max(m, n)$ times in the output.

Set_union is stable. The relative order of elements with each input set is preserved and if an element appears in both input sets then it is only copied from the first set.

The two versions of the set_union are different in that the first version uses operator$<$ to compare objects, and the second using a function object $comp$.

Asymptotic complexity: Let $N =$ last1 $-$ first1, $M =$ last2 $-$ first2.

- Average case: $O(N + M)$
- Worst case: $O(N + M)$

Complexity in terms of operation counts: The numbers in these tables are derived from operation count experiments performed with inputs randomly drawn from a set which is ten times bigger than the input sets.

Input sets do not contain duplicate keys. All the keys drawn for the first input set are put back, shuffling is performed and then keys for the second input set are drawn randomly. Thus two input sets are overlapping with very high probability. The key type used is integer.

Set_union (N=1000, M=1000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.024	9.69	3.798	13.512
value_cnt	1.898	0	6.69	8.588
Total	1.922	9.69	10.488	22.1

Set_union (N=1000, M=2000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.024	15.358	5.575	20.957
value_cnt	2.786	0	10.356	13.142
Total	2.81	15.358	15.931	34.099

Set_union (N=1000, M=4000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.024	26.813	9.215	36.052
value_cnt	4.607	0	17.815	22.422
Total	4.631	26.813	27.03	58.474

Set_union (N=2000, M=2000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.024	19.368	7.585	26.977
value_cnt	3.792	0	13.37	17.162
Total	3.816	19.368	20.955	44.139

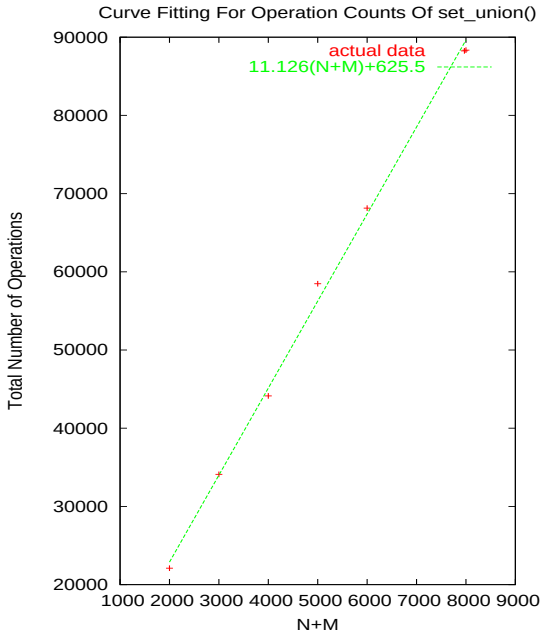
Set_union (N=2000, M=4000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.024	30.707	11.141	41.872
value_cnt	5.569	0	20.705	26.274
Total	5.593	30.707	31.846	68.146

Set_union (N=4000, M=4000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.024	38.769	15.184	53.977
value_cnt	7.591	0	26.769	34.36
Total	7.615	38.769	41.953	88.337

Curve Fitting for the Total Operation Counts:



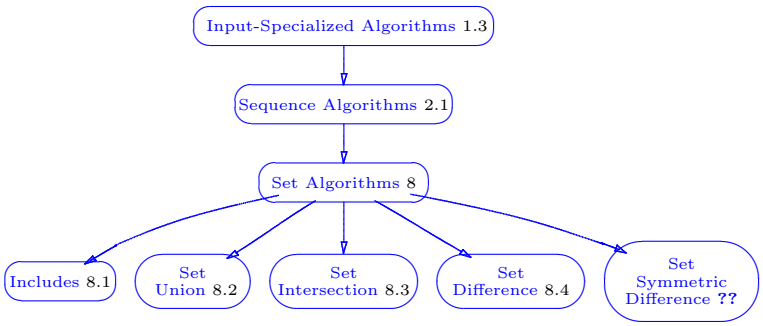
Average Case Formula: $11.126(N+M)+625.5$

Pseudocode (Actual Code):

```
while (first1 != last1 && first2 != last2) {
    if (*first1 < *first2) {
        *result = *first1;
        ++first1;
    }
    else if (*first2 < *first1) {
        *result = *first2;
        ++first2;
    }
```

```
  else {
     *result = *first1;
     ++first1;
     ++first2;
  }
  ++result;
  }
return copy(first2, last2,
            copy(first1, last1, result));
```

8.3 Set Intersection



Refinement of: Set Algorithms (§8), therefore refinement of Sequence Algorithms §2.1, therefore refinement of Input-Specialized Algorithms §1.3.

Prototype: Set_intersection is overloaded and has two versions:

```
template <class InputIterator1,
          class InputIterator2,
          class OutputIterator>
OutputIterator
set_intersection(InputIterator1 first1,
                 InputIterator1 last1,
                 InputIterator2 first2,
```

```
                    InputIterator2 last2,
                    OutputIterator result);

template <class InputIterator1,
          class InputIterator2,
          class OutputIterator,
          class StrictWeakOrdering>
OutputIterator
set_intersection(InputIterator1 first1,
                    InputIterator1 last1,
                    InputIterator2 first2,
                    InputIterator2 last2,
                    OutputIterator result,
                    StrictWeakOrdering comp);
```

Input: Inputs are two valid ranges of iterators and an output iterator. The range of elements should be sorted in ascending order. This is guaranteed if set container is used.

Output: None. However there are side-effects.

Effects: Set_intersection outputs a set with the sorted range which is the intersection of the sorted ranges [first1, last1) and [first2, last2).

The semantics of the intersection operation are slightly different depending on whether the input contains duplicate keys. When there are no duplicate keys in both the input sets (e.g., $Set < Key >$), set_intersection's output is a copy of every element that is contained in both [first1, last1) and [first2, last2). When there are duplicate keys in either input sets or both (e.g., $Multiset < Key >$), then if a key appears m times in the first set, and n times in the second set (either m or n could be $zero$), it appears $min(m, n)$ times in the output.

Set_intersection is stable. The relative order of elements with each input set is preserved and if an element appears in both input sets then it is only copied from the first set.

The two versions of the set_intersection are different in that the first version uses operator$<$ to compare objects, and the second using a function object $comp$.

Asymptotic complexity: Let $N =$ last1 $-$ first1, $M =$ last2 $-$ first2.

- Average case: $O(N + M)$
- Worst case: $O(N + M)$

Complexity in terms of operation counts: The numbers in these tables are derived from operation count experiments performed with inputs randomly drawn from a set which is ten times bigger than the input sets. Input sets do not contain duplicate keys. All the keys drawn for the first input set are put back, shuffling is performed and then keys for the second input set are drawn randomly. Thus two input sets are overlapping with very high probability. The key type used is integer.

Set_intersection (N=1000, M=1000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	7.893	3.795	11.692
value_cnt	0	0	4.372	4.372
Total	0.004	7.893	8.167	16.064

Set_intersection (N=1000, M=2000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	12.786	5.573	18.363
value_cnt	0	0	7.987	7.987
Total	0.004	12.786	13.56	26.35

Set_intersection (N=1000, M=4000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	22.597	9.211	31.812
value_cnt	0	0	14.741	14.741
Total	0.004	22.597	23.952	46.553

Set_intersection (N=2000, M=2000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	15.782	7.581	23.367
value_cnt	0	0	8.975	8.975
Total	0.004	15.782	16.556	32.342

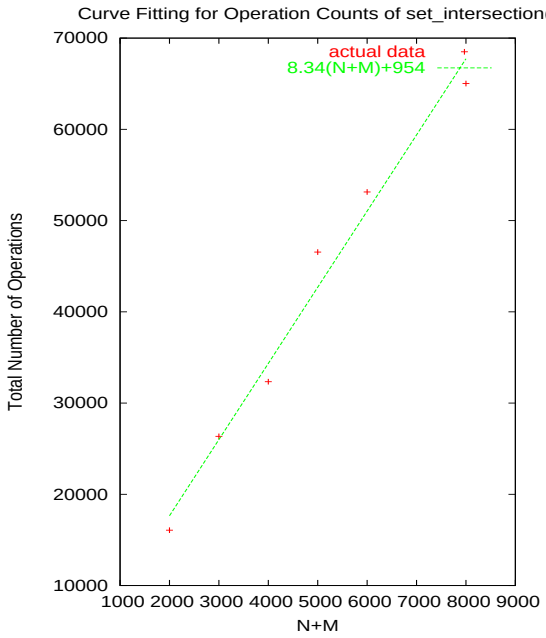
Set_intersection (N=2000, M=4000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.004	25.569	11.139	36.712
value_cnt	0	0	16.426	16.426
Total	0.004	25.569	27.565	53.138

Set_intersection (N=4000, M=4000)

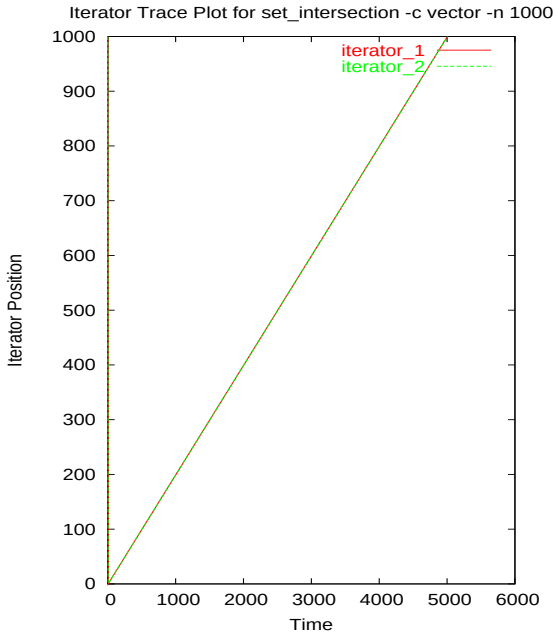
Type	Assign	Other	Compare	Total
iterator_cnt	0.004	31.586	15.181	46.771
value_cnt	0	0	18.247	18.247
Total	0.004	31.586	33.428	65.018

Curve Fitting for the Total Operation Counts:



Average Case Formula: $8.3477(N+M)+954.9$

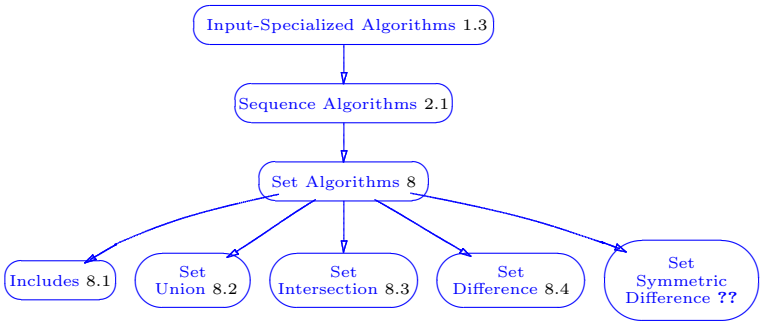
Iterator Trace Plot for set_intersection() Iterator trace plots are get from the output of the program *itrace*, the parameters used are -a set_intersection -c vector -n 1000.



Pseudocode (Actual Code):

```
while (first1 != last1 && first2 != last2)
    if (*first1 < *first2)
       ++first1;
    else if (*first2 < *first1)
       ++first2;
    else {
       *result = *first1;
       ++first1;
       ++first2;
       ++result;
    }
return result;
```

8.4 Set Difference



Refinement of: Set Algorithms (§8), therefore refinement of Sequence Algorithms §2.1, therefore refinement of Input-Specialized Algorithms §1.3.

Prototype: Set_difference is overloaded and has two versions:

```
template <class InputIterator1,
          class InputIterator2,
          class OutputIterator>
OutputIterator
set_difference(InputIterator1 first1,
               InputIterator1 last1,
               InputIterator2 first2,
               InputIterator2 last2,
               OutputIterator result);

template <class InputIterator1,
          class InputIterator2,
          class OutputIterator,
          class StrictWeakOrdering>
OutputIterator
set_difference(InputIterator1 first1,
               InputIterator1 last1,
               InputIterator2 first2,
               InputIterator2 last2,
```

```
OutputIterator result,
StrictWeakOrdering comp);
```

Input: Inputs are two valid ranges of iterators and an output iterator. The range of elements should be sorted in ascending order. This is guaranteed if set container is used.

Output: None. However there are side-effects.

Effects: Set_difference outputs a set with the sorted range which is the set difference of the sorted ranges [first1, last1) and [first2, last2).

The semantics of the set difference operation are slightly different depending on whether the input contains duplicate keys. When there are no duplicate keys in both the input sets (e.g., $Set < Key >$), set_difference's output is a copy of every element that is contained in [first1, last1) but not in [first2, last2). When there are duplicate keys in either input sets or both (e.g., $Multiset < Key >$), then if a key appears m times in the first set, and n times in the second set (either m or n could be $zero$), it appears $max(m - n, 0)$ times in the output.

Set_difference is stable. The relative order of elements with each input set is preserved and if an element appears in both input sets then it is only copied from the first set.

The two versions of the set_difference are different in that the first version uses operator$<$ to compare objects, and the second using a function object $comp$.

Asymptotic complexity: Let $N =$ last1 $-$ first1, $M =$ last2 $-$ first2.

- Average case: $O(N + M)$
- Worst case: $O(N + M)$

Complexity in terms of operation counts: The numbers in these tables are derived from operation count experiments performed with inputs randomly drawn from a set which is ten times bigger than the input sets. Input sets do not contain duplicate keys. All the keys drawn for the first input set are put back, shuffling is performed and then keys for the second input set are drawn randomly. Thus two input sets are overlapping with very high probability. The key type used is integer.

Set_difference (N=1000, M=1000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.014	8.689	3.796	12.499
value_cnt	0	0	15.724	15.724
Total	0.014	8.689	19.52	28.223

Set_difference (N=1000, M=2000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.014	13.358	5.574	18.946
value_cnt	0	0	16.415	16.415
Total	0.014	13.358	21.989	35.361

Set_difference (N=1000, M=4000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.014	22.811	9.212	32.037
value_cnt	0	0	18.168	18.168
Total	0.014	22.811	27.38	50.205

Set_difference (N=2000, M=2000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.014	17.366	7.582	24.962
value_cnt	0	0	33.176	33.176
Total	0.014	17.366	40.758	58.138

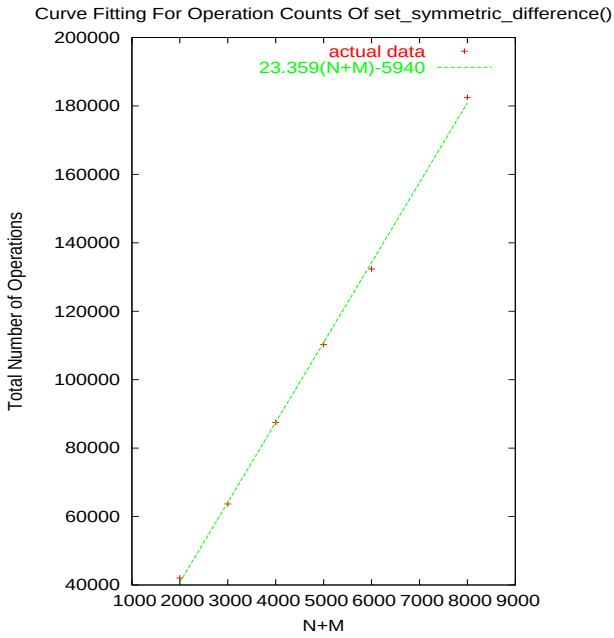
Set_difference (N=2000, M=4000)

Type	Assign	Other	Compare	Total
iterator_cnt	0.014	26.707	11.14	37.861
value_cnt	0	0	34.411	34.411
Total	0.014	26.707	45.551	72.272

Set_difference (N=4000, M=4000)

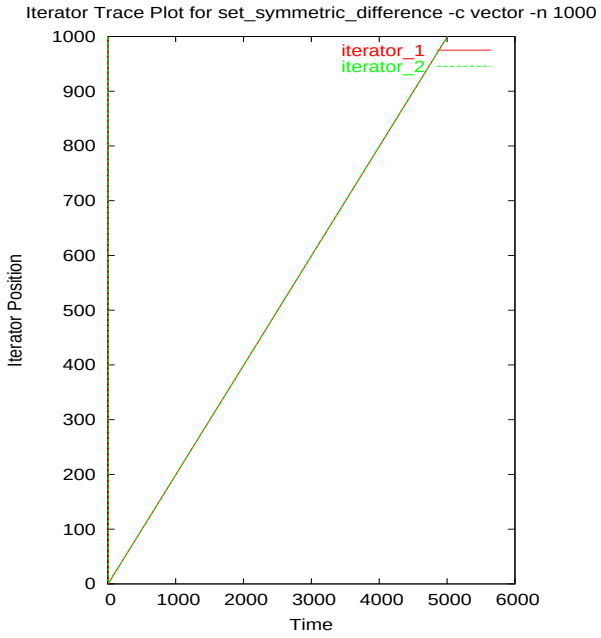
Type	Assign	Other	Compare	Total
iterator_cnt	0.014	34.768	15.182	49.964
value_cnt	0	0	70.326	70.326
Total	0.014	34.768	85.508	120.29

Curve Fitting for the Total Operation Counts:



Average Case Formula: $23.359(N+M)-5940.5$

Itrace for set_symmetric_difference() function Iterator trace plots are get from the output of the program *itrace*, the parameters used are -a set_symmetric_difference -c vector -n 1000.



Pseudocode (Actual Code) : `while (first1 != last1 && first2 != last2)`

```
    if (*first1 < *first2) {
       *result = *first1;
       ++first1;
       ++result;
    }
    else if (*first2 < *first1) {
       *result = *first2;
       ++first2;
       ++result;
    }
    else {
       ++first1;
       ++first2;
    }
  return copy(first2, last2,
              copy(first1, last1, result));
```