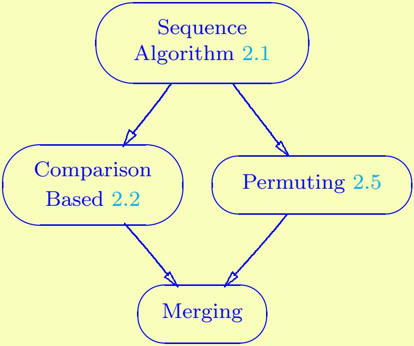
4. Merging Algorithm Concepts

Section authors: David R. Musser, Alessandro Assis, Amir Youssefi, Michal Sofka.

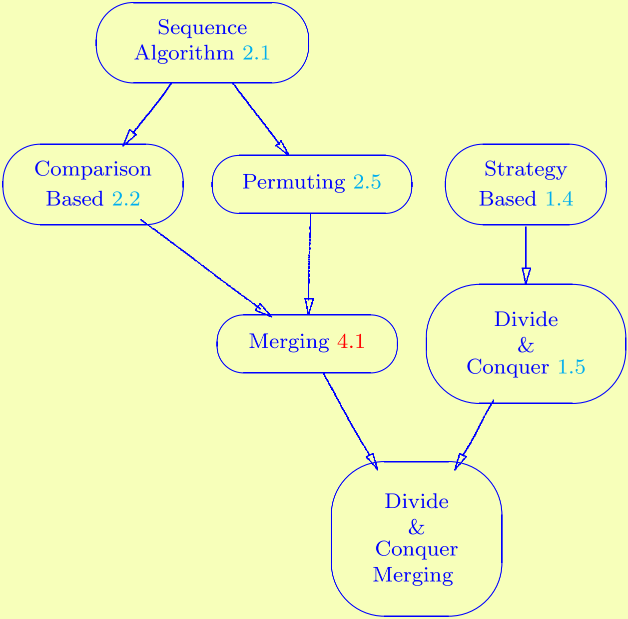
4.1. Merging Algorithm



A *merging algorithm* is an algorithm (§1.2) that takes two sorted sequences (§2.1) as inputs and combines them into a single sorted sequence.

Refinement of: Comparison Based (§2.2), Permuting (§2.5), Sequence Algorithm (§2.1).

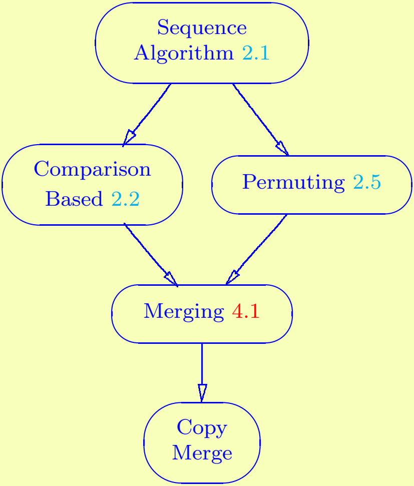
4.2. Divide-and-Conquer Merging Algorithm



A *divide-and-conquer-merge-algorithm* is a merging algorithm (§4.1) whose computation is based on the divide-and-conquer (§1.5) strategy.

Refinement of: Merging Algorithm (§4.1) Divide-and-Conquer Algorithm (§1.5), therefore of Comparison Based (§2.2), Permuting (§2.5), Sequence Algorithm (§2.1), Strategy Based Algorithms (§1.4).

4.3. Copy Merge Algorithm



Copy merge copies all elements from a sorted sequence `[first1,last1)` and another sorted sequence `[first2, last2)` into a single output sequence `[result,result+[last1-first1)+[last2-first2)` ,such that the resulting sequence is sorted in ascending order.

Refinement of: Merging Algorithm (§4.1), therefore of Comparison Based (§2.2), Permuting (§2.5), Sequence Algorithm (§2.1).

Prototype1:

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

Prototype2:

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

Prototype3:

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

Prototype3 is a special case of Prototype1, where the BackInsertIterator is used as the OutputIterator.

Input: Iterators `first1` and `last1` delimiting a sorted sequence `[first1,last1)`. Iterators `first2` and `last2` delimiting a sorted sequence `[first2,last2)`.

Output: Iterator `result` delimiting a sorted sequence `[result,result+[last1-first1)+[last2-first2)`.

Effects: The elements in `[first1,last1)` and `[first2, last2)` are concatenated and compared such that the resulting sequence `[result,result+(last1-first1)+(last2 - first2)]` is sorted in ascending order.

Asymptotic complexity: Let $N =$ (last1 - first1)$+$ (last2 - first2).

Average case (random data): $O(N)$

Worst case (random data): $O(N)$

Complexity in terms of operation counts:

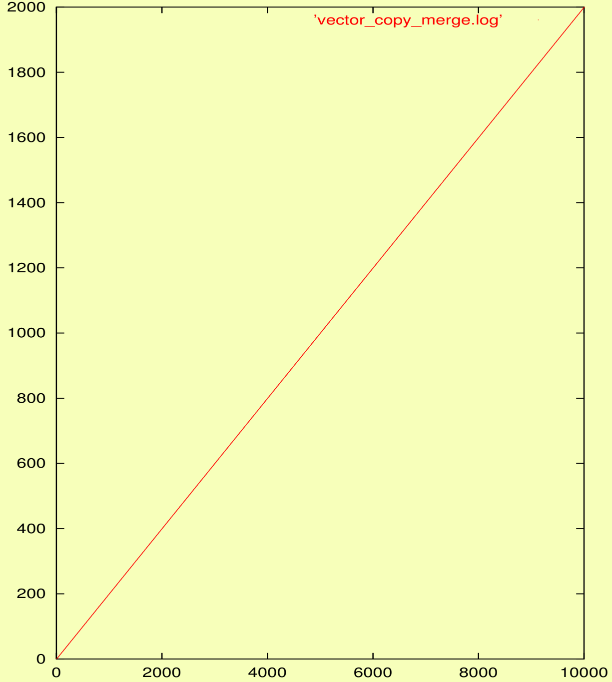
Worst case:

Value comparisons:	$N - 1$
Value assignments:	N
Iterator operations:	0
Integer operations:	0

See also Merging Algorithm Operation Counts (§4.5) for sample counts on random data for *copy merge* and inplace merge (§4.4) algorithms.

Animation: See Copy Merge Animation

Iterator trace plot:

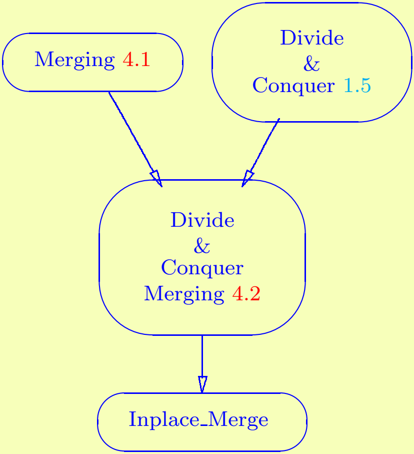


Two sorted sequences of one thousand elements each are being merged by the version of copy merge implemented in SGI STL. During all the time, the minimum values from each sequence are compared and the minimum element of those two is placed on the output sequence.

Pseudocode: [1] Merge(A,p,q,r)

```
n_1 ← q-p+1
n_2 ← r-q
for i ← 1 to n_1 do
    L[i] ← A[p+i-1]
for j ← 1 to n_2 do
    R[j] ← A[q+j]
L[n_1 + 1] ← ∞
R[n_2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r do
    if L[i] ≤ R[j]
    then A[k] ← L[i]
           i ← i+1
    else A[k] ← R[j]
           j ← j+1
```

4.4. Inplace Merge Algorithm



An *inplace merge algorithm* is a merging algorithm (§4.1) that takes two consecutive sorted sequences `[first,middle)` and `[middle,last)` and combines them into a single sorted sequence `[first,last)`. The execution is based on the divide-and-conquer (§1.5) strategy.

Refinement of: Divide-and-Conquer Merging Algorithm (§4.2), therefore of Divide-and-Conquer (§1.5) and Merging (§4.1).

Prototype1:

```
template <class BidirectionalIterator>
inline void inplace_merge
(BidirectionalIterator first,
BidirectionalIterator middle,
BidirectionalIterator last);
```

Prototype2:

```
template <class BidirectionalIterator,
          class StrictWeakOrdering>
inline void inplace_merge
(BidirectionalIterator first,
BidirectionalIterator middle,
BidirectionalIterator last,
StrictWeakOrdering comp);
```

Input: Iterators `first`, `middle` and `last` delimiting two consecutive sorted sequences `[first,middle)` and `[middle,last)`.

Output: Iterators `first` and `last` delimiting a single sorted sequence `[first,last)`.

Effects: The two consecutive sorted sequences `[first, middle)` and `[middle,last)` are rearranged such that the resulting sequence `[first,last)` is entirely sorted in ascending order.

Asymptotic complexity: Let $N = \text{last} - \text{first}$. Inplace merge is an adaptive algorithm: it attempts to allocate a temporary memory buffer,and its running-time complexity depends on how much memory is available.

Best Case (if sufficient auxiliary memory is available): $O(N)$

Worst case (if no auxiliary memory is available): $O(N \lg(N))$

Complexity in terms of operation counts:

- Auxiliary Memory Available:

Value comparisons:	$N - 1$
Value assignments:	$2.5N$
Iterator operations:	$6N + 0.053$
Integer operations:	$N/2 + 0.017$

- No Auxiliary Memory Available:

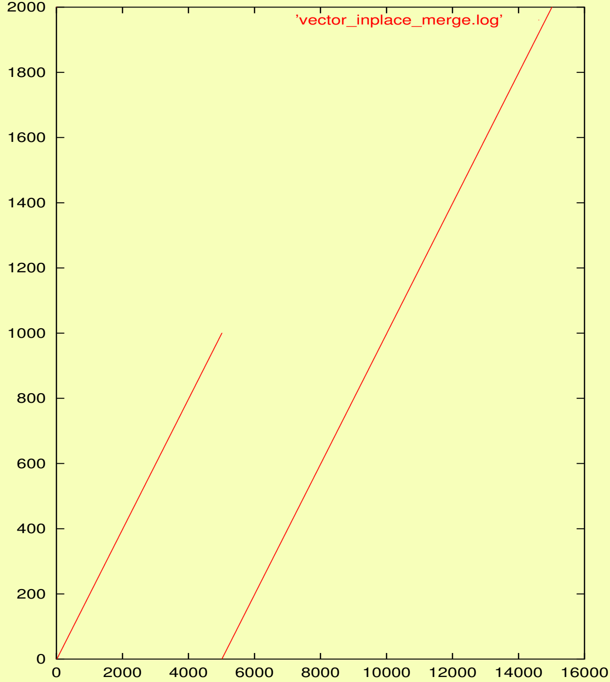
Value comparisons:	$3.47N$
Value assignments:	$3.03N\log_2 N + 30.76N$
Iterator operations:	$10.50N\log_2 N + 167.17N$
Integer operations:	$51.43N$

- See also Merging Algorithm Operation Counts (§4.5) for sample counts on random data for copy merge (§4.3) and *inplace merge* algorithms.

Animation: See Inplace Merge Animation

Iterator trace plot:

Auxiliary Memory Available



Two consecutive sorted sequences, contained in the same data structure, of one thousand elements each are being sorted by the version of inplace merge implemented in SGI STL. From time 0 to about time 5,000 the algorithm is copying the first 1,000 elements to auxiliary memory. Then, for the rest of the time, it is repeatedly comparing the minimum element of the second sequence with the minimum element in the auxiliary memory and placing the minimum element of these two in the data structure.

No Auxiliary Memory Available

See the plot here

From time 0 to about time 15,000, the algorithm is rotating the elements and calling InplaceMerge recursively (divide and conquer strategy). During the rest of the time, the final sorted sequence is merged.

Pseudocode:

```
Auxiliary Memory Available: InplaceMerge(A,p,q,r)
    n_1 ← q-p+1
    allocate one more cell at the end of A
    create buffer Tmp[1..n_1+1]
    for i ← 1 to n_1
        do Tmp[i] ← A[p+i-1]
    Tmp[n_1 + 1] ← ∞
    A[r+1] ← ∞
    i ← 1
    j ← q
    for k ← p to r
        do if Tmp[i] ≤ A[j]
            then A[k] ← Tmp[i]
                 i ← i+1
            else A[k] ← A[j]
                 j ← j+1
```

No Auxiliary Memory Available: InplaceMerge(p,q,r,s_1,s_2)

```
if (lenght(s1=0)) or (lenght(s2=0))
then return
if (lenght(s1) + lenght(s2) = 2)
then if (a[middle] < a[first])
        then swap(first,middle)
return
first-cut ← first
second-cut ← middle
x ← 0; y ← 0
if (lenght(s1)) > (lenght(s2))
then x ← (lenght(s1)/2)
        first-cut ← first-cut + x
        second-cut ← lower-bound(middle,last,a[first-cut])
        y ← y + second-cut - middle
else y ← (lenght(s2)/2)
        second-cut ← second-cut + y
        first-cut ← upper-bound(first,middle,a[second-cut])
        x ← x + first-cut - first
new-middle = rotate(first-cut, middle, second-cut)
InplaceMerge(first, first-cut, new-middle, x, y)
InplaceMerge(new-middle,second-cut,last,(lenght(s1)-x),(lenght(s2)-y))
```

4.5. Operation Counts for Copy Merge and Inplace Merge Algorithms

Table 1: Performance of Copy Merge and Inplace Merge on Random Sequences (Sizes and Operations Counts in Multiples of 1,000)

Size1	Size2	Algorithm	Comparisons	Assignments	Integer Ops	Iterator Ops
1	1	Copy (P1)	1.999	2	0	0
		Copy (P2)	1.999	2	0	0
		Copy (P3)	1.999	2	0	0
		Inplace (P1)	1.999	5	1.017	12.053
		Inplace (P2)	1	5	2.016	10.055
1	2	Copy (P1)	2	3	0	0
		Copy (P2)	2	3	0	0
		Copy (P3)	2	3	0	0
		Inplace (P1)	2.999	7	1.017	17.053
		Inplace (P2)	1	7	3.016	13.055
1	4	Copy (P1)	2	5	0	0
		Copy (P2)	2	5	0	0
		Copy (P3)	2	5	0	0
		Inplace (P1)	4.999	11	1.017	29.053
		Inplace (P2)	1	11	5.016	21.055
2	2	Copy (P1)	3.999	4	0	0
		Copy (P2)	3.999	4	0	0
		Copy (P3)	3.999	4	0	0
		Inplace (P1)	3.999	10	2.017	24.053
		Inplace (P2)	2	10	4.016	20.055
8	8	Copy (P1)	15.999	16	0	0
		Copy (P2)	15.999	16	0	0
		Copy (P3)	15.999	16	0	0
		Inplace (P1)	15.999	40	8.017	96.053
		Inplace (P2)	8	40	16.016	80.055
32	32	Copy (P1)	63.999	64	0	0
		Copy (P2)	63.999	64	0	0
		Copy (P3)	63.999	64	0	0
		Inplace (P1)	63.999	160	32.017	384.053
		Inplace (P2)	32	160	64.016	320.055

Table 2: Performance of Inplace Merge on Random Sequences When No Auxiliary Memory is Available (Sizes and Operations Counts in Multiples of 1,000)

Size1	Size2	Comparisons	Assignments	Integer Ops	Iterator Ops
1	1	3.437	30.474	51.753	167.164
1	2	3.446	36.474	51.870	188.256
1	4	3.455	45.474	51.987	222.349
2	2	6.903	67.182	103.255	355.340
8	8	27.754	318.102	411.313	1589.27
32	32	111.334	1468.04	1640.39	7028.64

References

[1] Cormen, T., Leiserson, C., Rivest, R., Stein, T. *Introduction to Algorithms*, Second Edition, MIT Press, Cambridge, MA, 2001.

[2] Lamport, L., *Latex: A Document Preparation System*, Second Edition, Addison-Wesley, New York, NY, 1994.

[3] Goosens, M., Mittelbach, F., Samarin, A. *The Latex Companion*, Addison Wesley, Reading, MA, 1994.