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).
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).
Copy merge copies all elements from a sorted sequence \([\text{first}_1, \text{last}_1)\) and another sorted sequence \([\text{first}_2, \text{last}_2)\) into a single output sequence \([\text{result}, \text{result} + \left(\text{last}_1 - \text{first}_1\right) + \left(\text{last}_2 - \text{first}_2\right))\), 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:

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

Prototype2:

```cpp
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 OutputIter-
ator.

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 ascend-
ing order.
Asymptotic complexity: Let $N = (\text{last1} - \text{first1}) + (\text{last2} - \text{first2})$.

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
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)

\[
\begin{align*}
n_1 & \leftarrow q-p+1 \\
n_2 & \leftarrow r-q \\
& \text{for } i \leftarrow 1 \text{ to } n_1 \text{ do} \\
& \quad L[i] \leftarrow A[p+i-1] \\
& \text{for } j \leftarrow 1 \text{ to } n_2 \text{ do} \\
& \quad R[j] \leftarrow A[q+j] \\
& L[n_1 + 1] \leftarrow \infty \\
& R[n_2 + 1] \leftarrow \infty \\
& i \leftarrow 1 \\
& j \leftarrow 1 \\
& \text{for } k \leftarrow p \text{ to } r \text{ do} \\
& \quad \text{if } L[i] \leq R[j] \\
& \quad \quad \text{then } A[k] \leftarrow L[i] \\
& \quad \quad \quad i \leftarrow i+1 \\
& \quad \quad \text{else } A[k] \leftarrow R[j] \\
& \quad \quad \quad j \leftarrow j+1
\end{align*}
\]
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:

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

Prototype2:

```cpp
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 \([\text{first}, \text{last})\).

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

Asymptotic complexity: Let \(N = \text{last} - \text{first}\). In-place 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): \(O(N \log(N))\)

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
Two consecutive sorted sequences, contained in the same data structure, of one thousand ele-
ments each are being sorted by the version of in-place 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)

\[ n_1 \leftarrow q-p+1 \]
allocate one more cell at the end of A
\[ create \ buffer \ Tmp[1..n_1+1] \]
for i ← 1 to \( n_1 \)
do \( Tmp[i] \leftarrow A[p+i-1] \)
\( Tmp[n_1+1] \leftarrow \infty \)
\( A[r+1] \leftarrow \infty \)
i ← 1
j ← q
for k ← p to r
    do if \( Tmp[i] \leq A[j] \)
        then \( A[k] \leftarrow Tmp[i] \)
            i ← i+1
        else \( A[k] \leftarrow A[j] \)
            j ← j+1
No Auxiliary Memory Available: InplaceMerge(p,q,r,s₁,s₂)
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)

<table>
<thead>
<tr>
<th>Size1</th>
<th>Size2</th>
<th>Algorithm</th>
<th>Comparisons</th>
<th>Assignments</th>
<th>Integer Ops</th>
<th>Iterator Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Copy (P1)</td>
<td>1.999</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P2)</td>
<td>1.999</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P3)</td>
<td>1.999</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P1)</td>
<td>1.999</td>
<td>5</td>
<td>1.017</td>
<td>12.053</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P2)</td>
<td>1</td>
<td>5</td>
<td>2.016</td>
<td>10.055</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Copy (P1)</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P2)</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P3)</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P1)</td>
<td>2.999</td>
<td>7</td>
<td>1.017</td>
<td>17.053</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P2)</td>
<td>1</td>
<td>7</td>
<td>3.016</td>
<td>13.055</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>Copy (P1)</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P2)</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P3)</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P1)</td>
<td>4.999</td>
<td>11</td>
<td>1.017</td>
<td>29.053</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P2)</td>
<td>1</td>
<td>11</td>
<td>5.016</td>
<td>21.055</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Copy (P1)</td>
<td>3.999</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P2)</td>
<td>3.999</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P3)</td>
<td>3.999</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P1)</td>
<td>3.999</td>
<td>10</td>
<td>2.017</td>
<td>24.053</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P2)</td>
<td>2</td>
<td>10</td>
<td>4.016</td>
<td>20.055</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>Copy (P1)</td>
<td>15.999</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P2)</td>
<td>15.999</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P3)</td>
<td>15.999</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P1)</td>
<td>15.999</td>
<td>40</td>
<td>8.017</td>
<td>96.053</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P2)</td>
<td>8</td>
<td>40</td>
<td>16.016</td>
<td>80.055</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>Copy (P1)</td>
<td>63.999</td>
<td>64</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P2)</td>
<td>63.999</td>
<td>64</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy (P3)</td>
<td>63.999</td>
<td>64</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P1)</td>
<td>63.999</td>
<td>160</td>
<td>32.017</td>
<td>384.053</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inplace (P2)</td>
<td>32</td>
<td>160</td>
<td>64.016</td>
<td>320.055</td>
</tr>
</tbody>
</table>
Table 2: Performance of Inplace Merge on Random Sequences When No Auxiliary Memory is Available (Sizes and Operations Counts in Multiples of 1,000)

<table>
<thead>
<tr>
<th>Size1</th>
<th>Size2</th>
<th>Comparisons</th>
<th>Assignments</th>
<th>Integer Ops</th>
<th>Iterator Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3.437</td>
<td>30.474</td>
<td>51.753</td>
<td>167.164</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3.446</td>
<td>36.474</td>
<td>51.870</td>
<td>188.256</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3.455</td>
<td>45.474</td>
<td>51.987</td>
<td>222.349</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6.903</td>
<td>67.182</td>
<td>103.255</td>
<td>355.340</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>27.754</td>
<td>318.102</td>
<td>411.313</td>
<td>1589.27</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>111.334</td>
<td>1468.04</td>
<td>1640.39</td>
<td>7028.64</td>
</tr>
</tbody>
</table>
References

