G++ 2.91.57，cygnus\cygwin-b20\include\g++\stl_algo.h 完整列表

```
/*
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1996
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

/* NOTE: This is an internal header file, included by other STL headers.
 *   You should not attempt to use it directly.
 */

#ifndef __SGI_STL_INTERNAL_ALGO_H
#define __SGI_STL_INTERNAL_ALGO_H

#include <stl_heap.h> // make_heap(), push_heap(), pop_heap(), sort_heap

__STL_BEGIN_NAMESPACE

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma set woff 1209
#endif

// 傳回 a,b,c 之居中者
template <class T>
inline const T& __median(const T& a, const T& b, const T& c) {
  if (a < b)
    if (b < c)          // a < b < c
      return b;
    else if (a < c)     // a < b, b >= c, a < c
```

```
      return c;
    else
      return a;
  else if (a < c)      // c > a >= b
    return a;
  else if (b < c)      // a >= b, a >= c, b < c
    return c;
  else
    return b;
}

template <class T, class Compare>
inline const T& __median(const T& a, const T& b, const T& c, Compare comp) {
  if (comp(a, b))
    if (comp(b, c))
      return b;
    else if (comp(a, c))
      return c;
    else
      return a;
  else if (comp(a, c))
    return a;
  else if (comp(b, c))
    return c;
  else
    return b;
}

template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f) {
  for ( ; first != last; ++first)
    f(*first);
  return f;
}

template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value) {
  while (first != last && *first != value) ++first;
  return first;
}

template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred) {
  while (first != last && !pred(*first)) ++first;
  return first;
}

// 搜尋相鄰的重複元素。版本一
```

```
template <class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last) {
  if (first == last) return last;
  ForwardIterator next = first;
  while(++next != last) {
    if (*first == *next) return first; // 如果找到相鄰的元素值相同，就結束
    first = next;
  }
  return last;
}

// 搜尋相鄰的重複元素。版本二
template <class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                              BinaryPredicate binary_pred) {
  if (first == last) return last;
  ForwardIterator next = first;
  while(++next != last) {
    // 如果找到相鄰的元素符合外界指定條件，就結束
    if (binary_pred(*first, *next)) return first;
    first = next;
  }
  return last;
}

// 這是舊版的 count()
template <class InputIterator, class T, class Size>
void count(InputIterator first, InputIterator last, const T& value,
         Size& n) {
  for ( ; first != last; ++first)    // 整個範圍走一遍
    if (*first == value)             // 如果元素值和 value 相等
      ++n;                           // 計數器累加 1
}

// 這是舊版的 count_if()
template <class InputIterator, class Predicate, class Size>
void count_if(InputIterator first, InputIterator last, Predicate pred,
            Size& n) {
  for ( ; first != last; ++first)    // 整個範圍走一遍
    if (pred(*first))                // 如果元素帶入pred 的運算結果為 true
      ++n;                           // 計數器累加 1
}

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION

// 這是新版的 count()
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value) {
  // 以下宣告一個計數器 n
  typename iterator_traits<InputIterator>::difference_type n = 0;
  for ( ; first != last; ++first)    // 整個範圍走一篇
    if (*first == value)             // 如果元素值和 value 相等
      ++n;                           // 計數器累加 1
  return n;
}

// 這是新版的 count_if()
template <class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred) {
  // 以下宣告一個計數器 n
  typename iterator_traits<InputIterator>::difference_type n = 0;
  for ( ; first != last; ++first)    // 整個範圍走一篇
    if (pred(*first))           // 如果元素帶入pred 的運算結果為 true
      ++n;                      // 計數器累加 1
  return n;
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

template <class ForwardIterator1, class ForwardIterator2, class Distance1,
        class Distance2>
ForwardIterator1 __search(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2,
                      Distance1*, Distance2*) {
  Distance1 d1 = 0;
  distance(first1, last1, d1);
  Distance2 d2 = 0;
  distance(first2, last2, d2);

  if (d1 < d2) return last1;  // 如果第二序列大於第一序列，不可能成為其子序列。

  ForwardIterator1 current1 = first1;
  ForwardIterator2 current2 = first2;

  while (current2 != last2)     // 走訪整個第二序列
    if (*current1 == *current2) {    // 如果這個元素相同
      ++current1;                    // 調整，以便比對下一個元素
      ++current2;
    }
    else {                      // 如果這個元素不等
      if (d1 == d2)             //   如果兩序列一樣長
        return last1;           //   表示不可能成功了
      else {                    // 兩序列不一樣長（至此肯定是序列一大於序列二）
        current1 = ++first1;    //   調整第一序列的標兵，
        current2 = first2;      //   準備在新起點上再找一次
        --d1;          // 已經排除了序列一的一個元素，所以序列一的長度要減 1
      }
    }
  return first1;
}

// 搜尋子序列首次出現地點
// 版本一
template <class ForwardIterator1, class ForwardIterator2>
inline ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2, ForwardIterator2 last2)
{
  return __search(first1, last1, first2, last2, distance_type(first1),
                  distance_type(first2));
}

template <class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate, class Distance1, class Distance2>
ForwardIterator1 __search(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
                          BinaryPredicate binary_pred, Distance1*, Distance2*) {
  Distance1 d1 = 0;
  distance(first1, last1, d1);
  Distance2 d2 = 0;
  distance(first2, last2, d2);

  if (d1 < d2) return last1;

  ForwardIterator1 current1 = first1;
  ForwardIterator2 current2 = first2;

  while (current2 != last2)
    if (binary_pred(*current1, *current2)) {
      ++current1;
      ++current2;
    }
    else {
      if (d1 == d2)
        return last1;
      else {
        current1 = ++first1;
        current2 = first2;
        --d1;
      }
    }
  return first1;
}

// 版本二
template <class ForwardIterator1, class ForwardIterator2,
        class BinaryPredicate>
inline ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
                          BinaryPredicate binary_pred) {
  return __search(first1, last1, first2, last2, binary_pred,
               distance_type(first1), distance_type(first2));
}

// 版本一。
// 搜尋「元素 value 連續出現count次」所形成的那個子序列，傳回其發生位置。
template <class ForwardIterator, class Integer, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Integer count, const T& value) {
  if (count <= 0)
    return first;
  else {
    first = find(first, last, value); // 首先找出 value 第一次出現點
    while (first != last) {         // 繼續搜尋餘下元素
      Integer n = count - 1;        // value 還應出現 n 次
      ForwardIterator i = first;    // 從上次出現點接下去搜尋
      ++i;
      while (i != last && n != 0 && *i == value) { // 下個元素是value，good.
        ++i;
        --n;                 // 既然找到了，「value 應再出現次數」便可減 1。
      }                      // 回到內迴圈內繼續搜尋
      if (n == 0)       // n==0 表示確實找到了「元素值出現n次」的子序列。功德圓滿。
        return first;
      else              // 功德尚未圓滿…
        first = find(i, last, value); // 找value 的下一個出現點，並準備回到外迴圈。
    }
    return last;
  }
}

// 版本二。
// 搜尋「連續count個元素皆滿足指定條件」所形成的那個子序列的起點，傳回其發生位置。
template <class ForwardIterator, class Integer, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Integer count, const T& value,
                         BinaryPredicate binary_pred) {
  if (count <= 0)
    return first;
  else {
    while (first != last) {
      if (binary_pred(*first, value)) break; // 首先找出第一個符合條件的元素
      ++first;                                 // 找到就離開。
    }
    while (first != last) {         // 繼續搜尋餘下元素
      Integer n = count - 1;        // 還應有n個連續元素符合條件
      ForwardIterator i = first;     // 從上次出現點接下去搜尋
      ++i;
      // 以下迴圈確定接下來 count-1 個元素是否都符合條件
      while (i != last && n != 0 && binary_pred(*i, value)) {
        ++i;
        --n;  // 既然這個元素符合條件，「應符合條件的元素個數」便可減 1。
      }
      if (n == 0) // n==0 表示確實找到了count個符合條件的元素。功德圓滿。
        return first;
      else {      // 功德尚未圓滿…
        while (i != last) {
          if (binary_pred(*i, value)) break;    // 搜尋下一個符合條件的元素
          ++i;
        }
        first = i;                              // 準備回到外迴圈
      }
    }
    return last;
  }
}

// 將兩段等長範圍內的元素互換。
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2) {
  for ( ; first1 != last1; ++first1, ++first2)
    iter_swap(first1, first2);
  return first2;
}

// 版本一
template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryOperation op) {
  for ( ; first != last; ++first, ++result)
    *result = op(*first);
  return result;
}

// 版本二
template <class InputIterator1, class InputIterator2, class OutputIterator,
          class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryOperation binary_op) {
  for ( ; first1 != last1; ++first1, ++first2, ++result)
    *result = binary_op(*first1, *first2);
  return result;
}
```

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last, const T& old_value,
             const T& new_value) {
  // 將範圍內的所有 old_value 都以 new_value 取代
  for ( ; first != last; ++first)
    if (*first == old_value) *first = new_value;
}

template <class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last, Predicate pred,
                const T& new_value) {
  for ( ; first != last; ++first)
    if (pred(*first)) *first = new_value;
}

template <class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
                            OutputIterator result, const T& old_value,
                            const T& new_value) {
  for ( ; first != last; ++first, ++result)
    // 如果舊序列上的元素等於 old_value，就放new_value到新序列中，
    // 否則就將元素拷貝一份放進新序列中。
    *result = *first == old_value ? new_value : *first;
  return result;
}

template <class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
                               OutputIterator result, Predicate pred,
                               const T& new_value) {
  for ( ; first != last; ++first, ++result)
    // 如果舊序列上的元素被 pred 評估為true，就放new_value到新序列中，
    // 否則就將元素拷貝一份放進新序列中。
    *result = pred(*first) ? new_value : *first;
  return result;
}

template <class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen) {
  for ( ; first != last; ++first)    // 整個序列範圍
    *first = gen();
}

template <class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator gen) {
  for ( ; n > 0; --n, ++first)  // 只限 n 個元素
    *first = gen();
  return first;
```

```
}

template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                       OutputIterator result, const T& value) {
  for ( ; first != last; ++first)
    if (*first != value) {     // 如果不相等
      *result = *first;        //   就指派給新容器
      ++result;                //   新容器前進一個位置
    }
  return result;
}

template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                         OutputIterator result, Predicate pred) {
  for ( ; first != last; ++first)
    if (!pred(*first)) {   // 如果pred核定為false，
      *result = *first;    //   就指派給新容器。
      ++result;            //   新容器前進一個位置。
    }
  return result;
}

template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                   const T& value) {
  first = find(first, last, value); // 利用循序搜尋法找出第一個相等元素
  ForwardIterator next = first;     // 以 next 標示出來
  // 以下利用「remove_copy()允許新舊容器重疊」的性質，做移除動作，
  // 並將結果放到原容器中。
  return first == last ? first : remove_copy(++next, last, first, value);
}

template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                      Predicate pred) {
  first = find_if(first, last, pred); // 利用循序搜尋法找出第一個吻合者。
  ForwardIterator next = first;       // 以 next 標記出來。
  // 以下利用「remove_copy_if()允許新舊容器重疊」的性質，做移除動作，
  // 並將結果放到原容器中。
  return first == last ? first : remove_copy_if(++next, last, first, pred);
}

// 版本一輔助函式，forward_iterator_tag 版
template <class InputIterator, class ForwardIterator>
ForwardIterator __unique_copy(InputIterator first, InputIterator last,
                         ForwardIterator result, forward_iterator_tag) {
  *result = *first;            // 登錄第一個元素
  while (++first != last)        // 走訪整個區間
  // 以下，元素不同，就再登錄，否則（元素相同），就跳過。
    if (*result != *first) *++result = *first;
  return ++result;
}

// 由於 output iterator 為 write only，無法像 forward iterator 那般可以讀取，
// 所以不能有類似 *result != *first 這樣的判斷動作，所以才需要設計此一版本。
// 例如，ostream_iterator 就是一個 output iterator.
template <class InputIterator, class OutputIterator, class T>
OutputIterator __unique_copy(InputIterator first, InputIterator last,
                             OutputIterator result, T*) {
  // T 為 output iterator 的 value type
  T value = *first;
  *result = value;
  while (++first != last)
    if (value != *first) {
      value = *first;
      *++result = value;
    }
  return ++result;
}

// 版本一輔助函式，output_iterator_tag 版
template <class InputIterator, class OutputIterator>
inline OutputIterator __unique_copy(InputIterator first, InputIterator last,
                                    OutputIterator result,
                                    output_iterator_tag) {
  // 以下，output iterator有一些功能限制，所以必須先知道其 value type.
  return __unique_copy(first, last, result, value_type(first));
}

// 版本一
template <class InputIterator, class OutputIterator>
inline OutputIterator unique_copy(InputIterator first, InputIterator last,
                                  OutputIterator result) {
  if (first == last) return result;
  // 以下，根據result 的 iterator category，做不同的處理
  return __unique_copy(first, last, result, iterator_category(result));
}

template <class InputIterator, class ForwardIterator, class BinaryPredicate>
ForwardIterator __unique_copy(InputIterator first, InputIterator last,
                              ForwardIterator result,
                              BinaryPredicate binary_pred,
                              forward_iterator_tag) {
  *result = *first;
  while (++first != last)
    if (!binary_pred(*result, *first)) *++result = *first;
  return ++result;
}

template <class InputIterator, class OutputIterator, class BinaryPredicate,
          class T>
OutputIterator __unique_copy(InputIterator first, InputIterator last,
                             OutputIterator result,
                             BinaryPredicate binary_pred, T*) {
  T value = *first;
  *result = value;
  while (++first != last)
    if (!binary_pred(value, *first)) {
      value = *first;
      *++result = value;
    }
  return ++result;
}

template <class InputIterator, class OutputIterator, class BinaryPredicate>
inline OutputIterator __unique_copy(InputIterator first, InputIterator last,
                                    OutputIterator result,
                                    BinaryPredicate binary_pred,
                                    output_iterator_tag) {
  return __unique_copy(first, last, result, binary_pred, value_type(first));
}

// 版本二
template <class InputIterator, class OutputIterator, class BinaryPredicate>
inline OutputIterator unique_copy(InputIterator first, InputIterator last,
                                  OutputIterator result,
                                  BinaryPredicate binary_pred) {
  if (first == last) return result;
  return __unique_copy(first, last, result, binary_pred,
                       iterator_category(result));
}

// 版本一
template <class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last) {
  first = adjacent_find(first, last);          // 首先找到相鄰重複元素的起點
  return unique_copy(first, last, first);      // 利用 unique_copy 完成。
}

// 版本二
template <class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                       BinaryPredicate binary_pred) {
  first = adjacent_find(first, last, binary_pred);
  return unique_copy(first, last, first, binary_pred);
```

```
}

// reverse 的 bidirectional iterator 版
template <class BidirectionalIterator>
void __reverse(BidirectionalIterator first, BidirectionalIterator last,
               bidirectional_iterator_tag) {
  while (true)
    if (first == last || first == --last)
      return;
    else
      iter_swap(first++, last);
}

// reverse 的 random access iterator 版
template <class RandomAccessIterator>
void __reverse(RandomAccessIterator first, RandomAccessIterator last,
               random_access_iterator_tag) {
  // 以下，頭尾兩兩互換，然後頭部累進一個位置，尾部累退一個位置。兩者交錯時即停止。
  // 注意，以下的 first < last 判斷動作，只適用於 random iterators.
  while (first < last) iter_swap(first++, --last);
}

// 分派函式（dispatch function）
template <class BidirectionalIterator>
inline void reverse(BidirectionalIterator first, BidirectionalIterator last) {
  __reverse(first, last, iterator_category(first));
}

template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result) {
  while (first != last) {  // 整個序列走一遍
    --last;                // 尾端前移一個位置
    *result = *last;       // 將尾端所指元素複製到 result 所指位置
    ++result;              // result 前進一個位置
  }
  return result;
}

// rotate 的 forward iterator 版
template <class ForwardIterator, class Distance>
void __rotate(ForwardIterator first, ForwardIterator middle,
              ForwardIterator last, Distance*, forward_iterator_tag) {
  for (ForwardIterator i = middle; ;) {
    iter_swap(first, i);    // 前段、後段的元素一一交換
    ++first;                // 雙雙前進 1
    ++i;
    // 以下判斷是前段[first, middle)先結束還是後段[middle,last)先結束
    if (first == middle) {      // 前段結束了
      if (i == last) return;     // 如果後段同時也結束，整個就結束了。
      middle = i;                // 否則調整，對新的前、後段再作交換。
    }
    else if (i == last)     // 後段先結束
      i = middle;           // 調整，準備對新的前、後段再作交換。
  }
}

// rotate 的 bidirectional iterator 版
template <class BidirectionalIterator, class Distance>
void __rotate(BidirectionalIterator first, BidirectionalIterator middle,
              BidirectionalIterator last, Distance*,
              bidirectional_iterator_tag) {
  reverse(first, middle);
  reverse(middle, last);
  reverse(first, last);
}

// 最大公因數，利用輾轉相除法。
// __gcd() 應用於 __rotate() 的 random access iterator 版
template <class EuclideanRingElement>
EuclideanRingElement __gcd(EuclideanRingElement m, EuclideanRingElement n)
{
  while (n != 0) {
    EuclideanRingElement t = m % n;
    m = n;
    n = t;
  }
  return m;
}

template <class RandomAccessIterator, class Distance, class T>
void __rotate_cycle(RandomAccessIterator first, RandomAccessIterator last,
                    RandomAccessIterator initial, Distance shift, T*) {
  T value = *initial;
  RandomAccessIterator ptr1 = initial;
  RandomAccessIterator ptr2 = ptr1 + shift;
  while (ptr2 != initial) {
    *ptr1 = *ptr2;
    ptr1 = ptr2;
    if (last - ptr2 > shift)
      ptr2 += shift;
    else
      ptr2 = first + (shift - (last - ptr2));
  }
  *ptr1 = value;
}
```

```
// rotate 的 random access iterator 版
template <class RandomAccessIterator, class Distance>
void __rotate(RandomAccessIterator first, RandomAccessIterator middle,
              RandomAccessIterator last, Distance*,
              random_access_iterator_tag) {
  // 以下迭代器的相減動作，只適用於 random access iterators
  // 取全長和前段長度的最大公因數。
  Distance n = __gcd(last - first, middle - first);
  while (n--)
    __rotate_cycle(first, last, first + n, middle - first,
                   value_type(first));
}

// 分派函式（dispatch function）
template <class ForwardIterator>
inline void rotate(ForwardIterator first, ForwardIterator middle,
                   ForwardIterator last) {
  if (first == middle || middle == last) return;
  __rotate(first, middle, last, distance_type(first),
           iterator_category(first));
}

template <class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result) {
  return copy(first, middle, copy(middle, last, result));
}

template <class RandomAccessIterator, class Distance>
void __random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                      Distance*) {
  if (first == last) return;
  for (RandomAccessIterator i = first + 1; i != last; ++i)
#ifdef __STL_NO_DRAND48
    iter_swap(i, first + Distance(rand() % ((i - first) + 1)));
#else
    iter_swap(i, first + Distance(lrand48() % ((i - first) + 1)));
#endif
// 注意，在我的GCC2.91.57 中，__STL_NO_DRAND48 是未定義的，因此上述實作碼會採用
// lrand48() 那個版本。但編譯時卻又說 lrand48() undeclared.
}

template <class RandomAccessIterator>
inline void random_shuffle(RandomAccessIterator first,
                           RandomAccessIterator last) {
  __random_shuffle(first, last, distance_type(first));
}

template <class RandomAccessIterator, class RandomNumberGenerator>
```

```
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    RandomNumberGenerator& rand) { // 注意，by reference
  if (first == last) return;
  for (RandomAccessIterator i = first + 1; i != last; ++i)
    iter_swap(i, first + rand((i - first) + 1));
}

template <class ForwardIterator, class OutputIterator, class Distance>
OutputIterator random_sample_n(ForwardIterator first, ForwardIterator last,
                               OutputIterator out, const Distance n)
{
  Distance remaining = 0;
  distance(first, last, remaining);
  Distance m = min(n, remaining);

  while (m > 0) {
#ifdef __STL_NO_DRAND48
    if (rand() % remaining < m) {
#else
    if (lrand48() % remaining < m) {
#endif
      *out = *first;
      ++out;
      --m;
    }

    --remaining;
    ++first;
  }
  return out;
}

template <class ForwardIterator, class OutputIterator, class Distance,
          class RandomNumberGenerator>
OutputIterator random_sample_n(ForwardIterator first, ForwardIterator last,
                               OutputIterator out, const Distance n,
                               RandomNumberGenerator& rand)
{
  Distance remaining = 0;
  distance(first, last, remaining);
  Distance m = min(n, remaining);

  while (m > 0) {
    if (rand(remaining) < m) {
      *out = *first;
      ++out;
      --m;
    }
    --remaining;
    ++first;
  }
  return out;
}

template <class InputIterator, class RandomAccessIterator, class Distance>
RandomAccessIterator __random_sample(InputIterator first, InputIterator last,
                                     RandomAccessIterator out,
                                     const Distance n)
{
  Distance m = 0;
  Distance t = n;
  for ( ; first != last && m < n; ++m, ++first)
    out[m] = *first;

  while (first != last) {
    ++t;
#ifdef __STL_NO_DRAND48
    Distance M = rand() % t;
#else
    Distance M = lrand48() % t;
#endif
    if (M < n)
      out[M] = *first;
    ++first;
  }

  return out + m;
}

template <class InputIterator, class RandomAccessIterator,
          class RandomNumberGenerator, class Distance>
RandomAccessIterator __random_sample(InputIterator first, InputIterator last,
                                     RandomAccessIterator out,
                                     RandomNumberGenerator& rand,
                                     const Distance n)
{
  Distance m = 0;
  Distance t = n;
  for ( ; first != last && m < n; ++m, ++first)
    out[m] = *first;

  while (first != last) {
    ++t;
    Distance M = rand(t);
    if (M < n)
      out[M] = *first;
    ++first;
  }

  return out + m;
}

template <class InputIterator, class RandomAccessIterator>
inline RandomAccessIterator
random_sample(InputIterator first, InputIterator last,
              RandomAccessIterator out_first, RandomAccessIterator out_last)
{
  return __random_sample(first, last, out_first, out_last - out_first);
}

template <class InputIterator, class RandomAccessIterator,
        class RandomNumberGenerator>
inline RandomAccessIterator
random_sample(InputIterator first, InputIterator last,
              RandomAccessIterator out_first, RandomAccessIterator out_last,
              RandomNumberGenerator& rand)
{
  return __random_sample(first, last, out_first, rand, out_last - out_first);
}

// 所有被 pred 判定為 true 的元素，都被放到前段，
// 被pred 判定為 false 的元素，都被放到後段。
// 不保證保留原相對位置。（not stable）
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last,
                                Predicate pred) {
  while (true) {
    while (true)
      if (first == last)        // 頭指標等於尾指標
        return first;           // 所有動作結束。
      else if (pred(*first))    // 頭指標所指的元素符合不搬移條件
        ++first;                // 不搬移；頭指標前進 1
      else                      // 頭指標所指元素符合搬移條件
        break;                  // 跳出迴圈
    --last;                     // 尾指標回溯 1
    while (true)
      if (first == last)        // 頭指標等於尾指標
        return first;           // 所有動作結束。
      else if (!pred(*last))    // 尾指標所指的元素符合不搬移條件
        --last;                 // 不搬移；尾指標回溯 1
      else                      // 尾指標所指元素符合搬移條件
        break;                  // 跳出迴圈
    iter_swap(first, last);     // 頭尾指標所指元素彼此交換
    ++first;                    // 頭指標前進 1，準備下一個外迴圈迭代
  }
```

```
}

template <class ForwardIterator, class Predicate, class Distance>
ForwardIterator __inplace_stable_partition(ForwardIterator first,
                                           ForwardIterator last,
                                           Predicate pred, Distance len) {
  if (len == 1) return pred(*first) ? last : first;
  ForwardIterator middle = first;
  advance(middle, len / 2);
  ForwardIterator
    first_cut = __inplace_stable_partition(first, middle, pred, len / 2);
  ForwardIterator
    second_cut = __inplace_stable_partition(middle, last, pred,
                                            len - len / 2);
  rotate(first_cut, middle, second_cut);
  len = 0;
  distance(middle, second_cut, len);
  advance(first_cut, len);
  return first_cut;
}

template <class ForwardIterator, class Pointer, class Predicate,
          class Distance>
ForwardIterator __stable_partition_adaptive(ForwardIterator first,
                                            ForwardIterator last,
                                            Predicate pred, Distance len,
                                            Pointer buffer,
                                            Distance buffer_size) {
  if (len <= buffer_size) {
    ForwardIterator result1 = first;
    Pointer result2 = buffer;
    for ( ; first != last ; ++first)
      if (pred(*first)) {
        *result1 = *first;
        ++result1;
      }
      else {
        *result2 = *first;
        ++result2;
      }
    copy(buffer, result2, result1);
    return result1;
  }
  else {
    ForwardIterator middle = first;
    advance(middle, len / 2);
    ForwardIterator first_cut =
      __stable_partition_adaptive(first, middle, pred, len / 2,
                                  buffer, buffer_size);
    ForwardIterator second_cut =
      __stable_partition_adaptive(middle, last, pred, len - len / 2,
                                  buffer, buffer_size);

    rotate(first_cut, middle, second_cut);
    len = 0;
    distance(middle, second_cut, len);
    advance(first_cut, len);
    return first_cut;
  }
}

template <class ForwardIterator, class Predicate, class T, class Distance>
inline ForwardIterator __stable_partition_aux(ForwardIterator first,
                                              ForwardIterator last,
                                              Predicate pred, T*, Distance*) {
  temporary_buffer<ForwardIterator, T> buf(first, last);
  if (buf.size() > 0)
    return __stable_partition_adaptive(first, last, pred,
                                       Distance(buf.requested_size()),
                                       buf.begin(), buf.size());
  else
    return __inplace_stable_partition(first, last, pred,
                                      Distance(buf.requested_size()));
}

template <class ForwardIterator, class Predicate>
inline ForwardIterator stable_partition(ForwardIterator first,
                                        ForwardIterator last,
                                        Predicate pred) {
  if (first == last)
    return first;
  else
    return __stable_partition_aux(first, last, pred,
                                  value_type(first), distance_type(first));
}

// 版本一
template <class RandomAccessIterator, class T>
RandomAccessIterator __unguarded_partition(RandomAccessIterator first,
                                           RandomAccessIterator last,
                                           T pivot) {
  while (true) {

    while (*first < pivot) ++first;  // first 找到 >= pivot 的元素，就停下來
    --last;                          // 調整
    while (pivot < *last) --last;    // last 找到 <= pivot 的元素，就停下來
    // 注意，以下first < last 判斷動作，只適用於random iterator
    if (!(first < last)) return first;    // 交錯，結束迴圈。
    iter_swap(first, last);                // 大小值交換
    ++first;                               // 調整
  }
}

// 版本二
template <class RandomAccessIterator, class T, class Compare>
RandomAccessIterator __unguarded_partition(RandomAccessIterator first,
                                           RandomAccessIterator last,
                                           T pivot, Compare comp) {
  while (1) {
    while (comp(*first, pivot)) ++first;
    --last;
    while (comp(pivot, *last)) --last;
    // 注意，以下的first < last 判斷動作，只適用於random iterator
    if (!(first < last)) return first;
    iter_swap(first, last);
    ++first;
  }
}

const int __stl_threshold = 16;


// 版本一
template <class RandomAccessIterator, class T>
void __unguarded_linear_insert(RandomAccessIterator last, T value) {
  RandomAccessIterator next = last;
  --next;
  // insertion sort 的內迴圈
  // 注意，一旦不出現逆轉對（inversion），迴圈就可以結束了。
  while (value < *next) {  // 逆轉對（inversion）存在
    *last = *next;         // 轉正
    last = next;           // 調整迭代器
    --next;                // 前進一個位置
  }
  *last = value;
}

// 版本二
template <class RandomAccessIterator, class T, class Compare>
void __unguarded_linear_insert(RandomAccessIterator last, T value,
                               Compare comp) {
  RandomAccessIterator next = last;
  --next;
  while (comp(value , *next)) {
    *last = *next;
    last = next;
    --next;
  }
  *last = value;
}

// 版本一
template <class RandomAccessIterator, class T>
inline void __linear_insert(RandomAccessIterator first,
                            RandomAccessIterator last, T*) {
  T value = *last;          // 記錄尾元素
  if (value < *first) {     // 尾比頭還小（那就別一個個比較了，一次做完…）
    copy_backward(first, last, last + 1); // 將整個範圍向右遞移一個位置
    *first = value;         // 令頭元素等於原先的尾元素值
  }
  else
    __unguarded_linear_insert(last, value);
}

// 版本二
template <class RandomAccessIterator, class T, class Compare>
inline void __linear_insert(RandomAccessIterator first,
                       RandomAccessIterator last, T*, Compare comp) {
  T value = *last;
  if (comp(value, *first)) {
    copy_backward(first, last, last + 1);
    *first = value;
  }
  else
    __unguarded_linear_insert(last, value, comp);
}

// 版本一
template <class RandomAccessIterator>
void __insertion_sort(RandomAccessIterator first, RandomAccessIterator last) {
  if (first == last) return;
  for (RandomAccessIterator i = first + 1; i != last; ++i)  // 外迴圈
    __linear_insert(first, i, value_type(first));  // first,i形成一個子範圍
}

// 版本二
template <class RandomAccessIterator, class Compare>
void __insertion_sort(RandomAccessIterator first,
                  RandomAccessIterator last, Compare comp) {
  if (first == last) return;
  for (RandomAccessIterator i = first + 1; i != last; ++i)
    __linear_insert(first, i, value_type(first), comp);
}

// 版本一
template <class RandomAccessIterator, class T>
```

```
void __unguarded_insertion_sort_aux(RandomAccessIterator first,
                                    RandomAccessIterator last, T*) {
  for (RandomAccessIterator i = first; i != last; ++i)
    __unguarded_linear_insert(i, T(*i));
}

// 版本一
template <class RandomAccessIterator>
inline void __unguarded_insertion_sort(RandomAccessIterator first,
                                RandomAccessIterator last) {
  __unguarded_insertion_sort_aux(first, last, value_type(first));
}

// 版本二
template <class RandomAccessIterator, class T, class Compare>
void __unguarded_insertion_sort_aux(RandomAccessIterator first,
                                    RandomAccessIterator last,
                                    T*, Compare comp) {
  for (RandomAccessIterator i = first; i != last; ++i)
    __unguarded_linear_insert(i, T(*i), comp);
}

// 版本二
template <class RandomAccessIterator, class Compare>
inline void __unguarded_insertion_sort(RandomAccessIterator first,
                                       RandomAccessIterator last,
                                       Compare comp) {
  __unguarded_insertion_sort_aux(first, last, value_type(first), comp);
}

// 版本一
template <class RandomAccessIterator>
void __final_insertion_sort(RandomAccessIterator first,
                            RandomAccessIterator last) {
  if (last - first > __stl_threshold) {
    __insertion_sort(first, first + __stl_threshold);
    __unguarded_insertion_sort(first + __stl_threshold, last);
  }
  else
    __insertion_sort(first, last);
}

// 版本二
template <class RandomAccessIterator, class Compare>
void __final_insertion_sort(RandomAccessIterator first,
                            RandomAccessIterator last, Compare comp) {
  if (last - first > __stl_threshold) {
    __insertion_sort(first, first + __stl_threshold, comp);
    __unguarded_insertion_sort(first + __stl_threshold, last, comp);
  }
  else
    __insertion_sort(first, last, comp);
}

// 找出 2^k <= n 的最大值k。例，n=7，得k=2，n=20，得k=4，n=8，得k=3。
template <class Size>
inline Size __lg(Size n) {
  Size k;
  for (k = 0; n > 1; n >>= 1) ++k;
  return k;
}

// 版本一
// 注意，本函式內的許多迭代器運算動作，都只適用於RandomAccess Iterators.
template <class RandomAccessIterator, class T, class Size>
void __introsort_loop(RandomAccessIterator first,
                      RandomAccessIterator last, T*,
                      Size depth_limit) {
  // 以下，__stl_threshold 是個全域常數，稍早定義為 const int 16。
  while (last - first > __stl_threshold) {
    if (depth_limit == 0) {                // 至此，切割惡化
      partial_sort(first, last, last);    // 改用 heapsort
      return;
    }
    --depth_limit;
    // 以下是 median-of-three partition，選擇一個夠好的樞軸並決定切割點。
    // 切割點將落在迭代器 cut 身上。
    RandomAccessIterator cut = __unguarded_partition
      (first, last, T(__median(*first, *(first + (last - first)/2),
                                 *(last - 1))));
    // 對右半段遞迴進行 sort.
    __introsort_loop(cut, last, value_type(first), depth_limit);
    last = cut;
    // 現在回到while 迴圈，準備對左半段遞迴進行 sort.
    // 這種寫法可讀性較差，效率並沒有比較好。
    // RW STL 採用一般教科書寫法（直觀地對左半段和右半段遞迴），較易閱讀。
  }
}

// 版本二
template <class RandomAccessIterator, class T, class Size, class Compare>
void __introsort_loop(RandomAccessIterator first,
                      RandomAccessIterator last, T*,
                      Size depth_limit, Compare comp) {
  while (last - first > __stl_threshold) {
    if (depth_limit == 0) {
      partial_sort(first, last, last, comp);
      return;
    }
    --depth_limit;
    RandomAccessIterator cut = __unguarded_partition
      (first, last, T(__median(*first, *(first + (last - first)/2),
                               *(last - 1), comp)), comp);
    __introsort_loop(cut, last, value_type(first), depth_limit, comp);
    last = cut;
  }
}

// 版本一
// 千萬注意：sort()只適用於 RandomAccessIterator
template <class RandomAccessIterator>
inline void sort(RandomAccessIterator first, RandomAccessIterator last) {
  if (first != last) {
    __introsort_loop(first, last, value_type(first), __lg(last - first) * 2);
    __final_insertion_sort(first, last);
  }
}

// 版本二
// 千萬注意：sort()只適用於 RandomAccessIterator
template <class RandomAccessIterator, class Compare>
inline void sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp) {
  if (first != last) {
    __introsort_loop(first, last, value_type(first), __lg(last - first) * 2,
                     comp);
    __final_insertion_sort(first, last, comp);
  }
}


template <class RandomAccessIterator>
void __inplace_stable_sort(RandomAccessIterator first,
                           RandomAccessIterator last) {
  // 注意，以下的last-first < 15 判斷動作，只適用於random iterator
  if (last - first < 15) {
    __insertion_sort(first, last);
    return;
  }
  RandomAccessIterator middle = first + (last - first) / 2;
  __inplace_stable_sort(first, middle);
  __inplace_stable_sort(middle, last);
  __merge_without_buffer(first, middle, last, middle - first, last - middle);
}

template <class RandomAccessIterator, class Compare>
void __inplace_stable_sort(RandomAccessIterator first,
                          RandomAccessIterator last, Compare comp) {
  // 注意，以下的last-first < 15 判斷動作，只適用於random iterator
  if (last - first < 15) {
    __insertion_sort(first, last, comp);
    return;
  }
  RandomAccessIterator middle = first + (last - first) / 2;
  __inplace_stable_sort(first, middle, comp);
  __inplace_stable_sort(middle, last, comp);
  __merge_without_buffer(first, middle, last, middle - first,
                        last - middle, comp);
}

template <class RandomAccessIterator1, class RandomAccessIterator2,
          class Distance>
void __merge_sort_loop(RandomAccessIterator1 first,
                       RandomAccessIterator1 last,
                       RandomAccessIterator2 result, Distance step_size) {
  Distance two_step = 2 * step_size;

  while (last - first >= two_step) {
    result = merge(first, first + step_size,
                   first + step_size, first + two_step, result);
    first += two_step;
  }

  step_size = min(Distance(last - first), step_size);
  merge(first, first + step_size, first + step_size, last, result);
}

template <class RandomAccessIterator1, class RandomAccessIterator2,
          class Distance, class Compare>
void __merge_sort_loop(RandomAccessIterator1 first,
                       RandomAccessIterator1 last,
                       RandomAccessIterator2 result, Distance step_size,
                       Compare comp) {
  Distance two_step = 2 * step_size;

  while (last - first >= two_step) {
    result = merge(first, first + step_size,
                   first + step_size, first + two_step, result, comp);
    first += two_step;
  }
  step_size = min(Distance(last - first), step_size);

  merge(first, first + step_size, first + step_size, last, result, comp);
}

const int __stl_chunk_size = 7;
```

```
template <class RandomAccessIterator, class Distance>
void __chunk_insertion_sort(RandomAccessIterator first,
                       RandomAccessIterator last, Distance chunk_size) {
  while (last - first >= chunk_size) {
    __insertion_sort(first, first + chunk_size);
    first += chunk_size;
  }
  __insertion_sort(first, last);
}

template <class RandomAccessIterator, class Distance, class Compare>
void __chunk_insertion_sort(RandomAccessIterator first,
                       RandomAccessIterator last,
                       Distance chunk_size, Compare comp) {
  while (last - first >= chunk_size) {
    __insertion_sort(first, first + chunk_size, comp);
    first += chunk_size;
  }
  __insertion_sort(first, last, comp);
}

template <class RandomAccessIterator, class Pointer, class Distance>
void __merge_sort_with_buffer(RandomAccessIterator first,
                         RandomAccessIterator last,
                         Pointer buffer, Distance*) {
  Distance len = last - first;
  Pointer buffer_last = buffer + len;

  Distance step_size = __stl_chunk_size;
  __chunk_insertion_sort(first, last, step_size);

  while (step_size < len) {
    __merge_sort_loop(first, last, buffer, step_size);
    step_size *= 2;
    __merge_sort_loop(buffer, buffer_last, first, step_size);
    step_size *= 2;
  }
}

template <class RandomAccessIterator, class Pointer, class Distance,
        class Compare>
void __merge_sort_with_buffer(RandomAccessIterator first,
                         RandomAccessIterator last, Pointer buffer,
                         Distance*, Compare comp) {
  Distance len = last - first;
  Pointer buffer_last = buffer + len;

  Distance step_size = __stl_chunk_size;
  __chunk_insertion_sort(first, last, step_size, comp);

  while (step_size < len) {
    __merge_sort_loop(first, last, buffer, step_size, comp);
    step_size *= 2;
    __merge_sort_loop(buffer, buffer_last, first, step_size, comp);
    step_size *= 2;
  }
}

template <class RandomAccessIterator, class Pointer, class Distance>
void __stable_sort_adaptive(RandomAccessIterator first,
                            RandomAccessIterator last, Pointer buffer,
                            Distance buffer_size) {
  Distance len = (last - first + 1) / 2;
  RandomAccessIterator middle = first + len;
  if (len > buffer_size) {
    __stable_sort_adaptive(first, middle, buffer, buffer_size);
    __stable_sort_adaptive(middle, last, buffer, buffer_size);
  } else {
    __merge_sort_with_buffer(first, middle, buffer, (Distance*)0);
    __merge_sort_with_buffer(middle, last, buffer, (Distance*)0);
  }
  __merge_adaptive(first, middle, last, Distance(middle - first),
                   Distance(last - middle), buffer, buffer_size);
}

template <class RandomAccessIterator, class Pointer, class Distance,
          class Compare>
void __stable_sort_adaptive(RandomAccessIterator first,
                            RandomAccessIterator last, Pointer buffer,
                            Distance buffer_size, Compare comp) {
  Distance len = (last - first + 1) / 2;
  RandomAccessIterator middle = first + len;
  if (len > buffer_size) {
    __stable_sort_adaptive(first, middle, buffer, buffer_size,
                           comp);
    __stable_sort_adaptive(middle, last, buffer, buffer_size,
                           comp);
  } else {
    __merge_sort_with_buffer(first, middle, buffer, (Distance*)0, comp);
    __merge_sort_with_buffer(middle, last, buffer, (Distance*)0, comp);
  }
  __merge_adaptive(first, middle, last, Distance(middle - first),
                   Distance(last - middle), buffer, buffer_size,
                   comp);
}

template <class RandomAccessIterator, class T, class Distance>
```

```
inline void __stable_sort_aux(RandomAccessIterator first,
                              RandomAccessIterator last, T*, Distance*) {
  temporary_buffer<RandomAccessIterator, T> buf(first, last);
  if (buf.begin() == 0)
    __inplace_stable_sort(first, last);
  else
    __stable_sort_adaptive(first, last, buf.begin(), Distance(buf.size()));
}

template <class RandomAccessIterator, class T, class Distance, class Compare>
inline void __stable_sort_aux(RandomAccessIterator first,
                              RandomAccessIterator last, T*, Distance*,
                              Compare comp) {
  temporary_buffer<RandomAccessIterator, T> buf(first, last);
  if (buf.begin() == 0)
    __inplace_stable_sort(first, last, comp);
  else
    __stable_sort_adaptive(first, last, buf.begin(), Distance(buf.size()),
                           comp);
}

template <class RandomAccessIterator>
inline void stable_sort(RandomAccessIterator first,
                        RandomAccessIterator last) {
  __stable_sort_aux(first, last, value_type(first), distance_type(first));
}

template <class RandomAccessIterator, class Compare>
inline void stable_sort(RandomAccessIterator first,
                        RandomAccessIterator last, Compare comp) {
  __stable_sort_aux(first, last, value_type(first), distance_type(first),
                    comp);
}

template <class RandomAccessIterator, class T>
void __partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                    RandomAccessIterator last, T*) {
  make_heap(first, middle);
  // 注意，以下的i < last 判斷動作，只適用於random iterator
  for (RandomAccessIterator i = middle; i < last; ++i)
    if (*i < *first)
      __pop_heap(first, middle, i, T(*i), distance_type(first));
  sort_heap(first, middle);
}

// 版本一
template <class RandomAccessIterator>
inline void partial_sort(RandomAccessIterator first,
                         RandomAccessIterator middle,
                     RandomAccessIterator last) {
  __partial_sort(first, middle, last, value_type(first));
}

template <class RandomAccessIterator, class T, class Compare>
void __partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                    RandomAccessIterator last, T*, Compare comp) {
  make_heap(first, middle, comp);
  // 注意，以下的i < last 判斷動作，只適用於random iterator
  for (RandomAccessIterator i = middle; i < last; ++i)
    if (comp(*i, *first))
      __pop_heap(first, middle, i, T(*i), comp, distance_type(first));
  sort_heap(first, middle, comp);
}

// 版本二
template <class RandomAccessIterator, class Compare>
inline void partial_sort(RandomAccessIterator first,
                         RandomAccessIterator middle,
                         RandomAccessIterator last, Compare comp) {
  __partial_sort(first, middle, last, value_type(first), comp);
}

template <class InputIterator, class RandomAccessIterator, class Distance,
          class T>
RandomAccessIterator __partial_sort_copy(InputIterator first,
                                         InputIterator last,
                                         RandomAccessIterator result_first,
                                         RandomAccessIterator result_last,
                                         Distance*, T*) {
  if (result_first == result_last) return result_last;
  RandomAccessIterator result_real_last = result_first;
  while(first != last && result_real_last != result_last) {
    *result_real_last = *first;
    ++result_real_last;
    ++first;
  }
  make_heap(result_first, result_real_last);
  while (first != last) {
    if (*first < *result_first)
      __adjust_heap(result_first, Distance(0),
                    Distance(result_real_last - result_first), T(*first));
    ++first;
  }
  sort_heap(result_first, result_real_last);
  return result_real_last;
}

// 版本一
```

```
template <class InputIterator, class RandomAccessIterator>
inline RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
               RandomAccessIterator result_first,
               RandomAccessIterator result_last) {
  return __partial_sort_copy(first, last, result_first, result_last,
                        distance_type(result_first), value_type(first));
}

template <class InputIterator, class RandomAccessIterator, class Compare,
        class Distance, class T>
RandomAccessIterator __partial_sort_copy(InputIterator first,
                                  InputIterator last,
                                  RandomAccessIterator result_first,
                                  RandomAccessIterator result_last,
                                  Compare comp, Distance*, T*) {
  if (result_first == result_last) return result_last;
  RandomAccessIterator result_real_last = result_first;
  while(first != last && result_real_last != result_last) {
    *result_real_last = *first;
    ++result_real_last;
    ++first;
  }
  make_heap(result_first, result_real_last, comp);
  while (first != last) {
    if (comp(*first, *result_first))
      __adjust_heap(result_first, Distance(0),
                 Distance(result_real_last - result_first), T(*first),
                 comp);
    ++first;
  }
  sort_heap(result_first, result_real_last, comp);
  return result_real_last;
}

// 版本二
template <class InputIterator, class RandomAccessIterator, class Compare>
inline RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
               RandomAccessIterator result_first,
               RandomAccessIterator result_last, Compare comp) {
  return __partial_sort_copy(first, last, result_first, result_last, comp,
                        distance_type(result_first), value_type(first));
}

// 版本一輔助函式
template <class RandomAccessIterator, class T>
void __nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                  RandomAccessIterator last, T*) {
```

```
  while (last - first > 3) {    // 長度超過 3
    RandomAccessIterator cut = __unguarded_partition
      (first, last, T(__median(*first, *(first + (last - first)/2),
                                *(last - 1))));
    if (cut <= nth)
      first = cut;
    else
      last = cut;
  }
  __insertion_sort(first, last);
}

// 版本一
template <class RandomAccessIterator>
inline void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                        RandomAccessIterator last) {
  __nth_element(first, nth, last, value_type(first));
}

// 版本二輔助函式
template <class RandomAccessIterator, class T, class Compare>
void __nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, T*, Compare comp) {
  while (last - first > 3) {
    RandomAccessIterator cut = __unguarded_partition
      (first, last, T(__median(*first, *(first + (last - first)/2),
                                *(last - 1), comp)), comp);
    if (cut <= nth)
      first = cut;
    else
      last = cut;
  }
  __insertion_sort(first, last, comp);
}

template <class RandomAccessIterator, class Compare>
inline void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                        RandomAccessIterator last, Compare comp) {
  __nth_element(first, nth, last, value_type(first), comp);
}

// 這是版本一的 forward_iterator 版本
template <class ForwardIterator, class T, class Distance>
ForwardIterator __lower_bound(ForwardIterator first, ForwardIterator last,
                              const T& value, Distance*,
                              forward_iterator_tag) {
  Distance len = 0;
  distance(first, last, len);   // 求取整個範圍的長度 len
  Distance half;
  ForwardIterator middle;

  while (len > 0) {
    half = len >> 1;             // 除以 2
    middle = first;              // 這兩行令middle 指向中間位置
    advance(middle, half);
    if (*middle < value) {       // 如果中間位置的元素值 < 標的值
      first = middle;            // 這兩行令 first 指向 middle 的下一位置
      ++first;
      len = len - half - 1;      // 修正 len，回頭測試迴圈的結束條件
    }
    else
      len = half;                // 修正 len，回頭測試迴圈的結束條件
  }
  return first;
}

// 這是版本一的 random_access_iterator 版本
template <class RandomAccessIterator, class T, class Distance>
RandomAccessIterator __lower_bound(RandomAccessIterator first,
                                   RandomAccessIterator last, const T& value,
                                   Distance*, random_access_iterator_tag) {
  Distance len = last - first;  // 求取整個範圍的長度 len
  Distance half;
  RandomAccessIterator middle;

  while (len > 0) {
    half = len >> 1;             // 除以 2
    middle = first + half;       // 令middle 指向中間位置
    if (*middle < value) {       // 如果中間位置的元素值 < 標的值
      first = middle + 1;        // 令 first 指向 middle 的下一位置
      len = len - half - 1;      // 修正 len，回頭測試迴圈的結束條件
    }
    else
      len = half;                // 修正 len，回頭測試迴圈的結束條件
  }
  return first;
}

// 這是版本一
template <class ForwardIterator, class T>
inline ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                                   const T& value) {
  return __lower_bound(first, last, value, distance_type(first),
                       iterator_category(first));
}

// 這是版本二的 forward_iterator 版本
template <class ForwardIterator, class T, class Compare, class Distance>
```

```
ForwardIterator __lower_bound(ForwardIterator first, ForwardIterator last,
                              const T& value, Compare comp, Distance*,
                              forward_iterator_tag) {
  Distance len = 0;
  distance(first, last, len);
  Distance half;
  ForwardIterator middle;

  while (len > 0) {
    half = len >> 1;
    middle = first;
    advance(middle, half);
    if (comp(*middle, value)) {
      first = middle;
      ++first;
      len = len - half - 1;
    }
    else
      len = half;
  }
  return first;
}

// 這是版本二的 random_access_iterator 版本
template <class RandomAccessIterator, class T, class Compare, class Distance>
RandomAccessIterator __lower_bound(RandomAccessIterator first,
                                   RandomAccessIterator last,
                                   const T& value, Compare comp, Distance*,
                                   random_access_iterator_tag) {
  Distance len = last - first;
  Distance half;
  RandomAccessIterator middle;

  while (len > 0) {
    half = len >> 1;
    middle = first + half;
    if (comp(*middle, value)) {
      first = middle + 1;
      len = len - half - 1;
    }
    else
      len = half;
  }
  return first;
}

// 這是版本二
template <class ForwardIterator, class T, class Compare>
inline ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                                 const T& value, Compare comp) {
  return __lower_bound(first, last, value, comp, distance_type(first),
                       iterator_category(first));
}

// 這是版本一的 forward_iterator 版本
template <class ForwardIterator, class T, class Distance>
ForwardIterator __upper_bound(ForwardIterator first, ForwardIterator last,
                              const T& value, Distance*,
                              forward_iterator_tag) {
  Distance len = 0;
  distance(first, last, len);   // 求取整個範圍的長度 len
  Distance half;
  ForwardIterator middle;

  while (len > 0) {
    half = len >> 1;            // 除以 2
    middle = first;             // 這兩行令middle 指向中間位置
    advance(middle, half);
    if (value < *middle)        // 如果中間位置的元素值 > 標的值
      len = half;               // 修正 len，回頭測試迴圈的結束條件
    else {
      first = middle;           // 這兩行令 first 指向 middle 的下一位置
      ++first;
      len = len - half - 1;     // 修正 len，回頭測試迴圈的結束條件
    }
  }
  return first;
}

// 這是版本一的 random_access_iterator 版本
template <class RandomAccessIterator, class T, class Distance>
RandomAccessIterator __upper_bound(RandomAccessIterator first,
                                   RandomAccessIterator last, const T& value,
                                   Distance*, random_access_iterator_tag) {
  Distance len = last - first;  // 求取整個範圍的長度 len
  Distance half;
  RandomAccessIterator middle;

  while (len > 0) {
    half = len >> 1;            // 除以 2
    middle = first + half;      // 令middle 指向中間位置
    if (value < *middle)        // 如果中間位置的元素值 > 標的值
      len = half;               // 修正 len，回頭測試迴圈的結束條件
    else {
      first = middle + 1;       // 令 first 指向 middle 的下一位置
      len = len - half - 1;     // 修正 len，回頭測試迴圈的結束條件
    }
  }
  return first;
}

// 這是版本一
template <class ForwardIterator, class T>
inline ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                             const T& value) {
  return __upper_bound(first, last, value, distance_type(first),
                   iterator_category(first));
}

// 這是版本二的 forward_iterator 版本
template <class ForwardIterator, class T, class Compare, class Distance>
ForwardIterator __upper_bound(ForwardIterator first, ForwardIterator last,
                         const T& value, Compare comp, Distance*,
                         forward_iterator_tag) {
  Distance len = 0;
  distance(first, last, len);
  Distance half;
  ForwardIterator middle;

  while (len > 0) {
    half = len >> 1;
    middle = first;
    advance(middle, half);
    if (comp(value, *middle))
      len = half;
    else {
      first = middle;
      ++first;
      len = len - half - 1;
    }
  }
  return first;
}

// 這是版本二的 random_access_iterator 版本
template <class RandomAccessIterator, class T, class Compare, class Distance>
RandomAccessIterator __upper_bound(RandomAccessIterator first,
                             RandomAccessIterator last,
                             const T& value, Compare comp, Distance*,
                             random_access_iterator_tag) {
  Distance len = last - first;
  Distance half;
  RandomAccessIterator middle;

  while (len > 0) {
    half = len >> 1;
    middle = first + half;
    if (comp(value, *middle))
      len = half;
    else {
      first = middle + 1;
      len = len - half - 1;
    }
  }
  return first;
}

// 這是版本二
template <class ForwardIterator, class T, class Compare>
inline ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                                   const T& value, Compare comp) {
  return __upper_bound(first, last, value, comp, distance_type(first),
                       iterator_category(first));
}

// 版本一的 forward_iterator 版本
template <class ForwardIterator, class T, class Distance>
pair<ForwardIterator, ForwardIterator>
__equal_range(ForwardIterator first, ForwardIterator last, const T& value,
              Distance*, forward_iterator_tag) {
  Distance len = 0;
  distance(first, last, len);
  Distance half;
  ForwardIterator middle, left, right;

  while (len > 0) {
    half = len >> 1;
    middle = first;
    advance(middle, half);
    if (*middle < value) {
      first = middle;
      ++first;
      len = len - half - 1;
    }
    else if (value < *middle)
      len = half;
    else {
      left = lower_bound(first, middle, value);
      advance(first, len);
      right = upper_bound(++middle, first, value);
      return pair<ForwardIterator, ForwardIterator>(left, right);
    }
  }
  return pair<ForwardIterator, ForwardIterator>(first, first);
}
```

```
// 版本一的 random_access_iterator 版本
template <class RandomAccessIterator, class T, class Distance>
pair<RandomAccessIterator, RandomAccessIterator>
__equal_range(RandomAccessIterator first, RandomAccessIterator last,
              const T& value, Distance*, random_access_iterator_tag) {
  Distance len = last - first;
  Distance half;
  RandomAccessIterator middle, left, right;

  while (len > 0) {             // 整個區間尚未走訪完畢
    half = len >> 1;            // 找出中央位置
    middle = first + half;      // 設定中央迭代器
    if (*middle < value) {      // 如果中央元素 < 指定值
      first = middle + 1;       // 將運作區間縮小（移至後半段），以提高效率
      len = len - half - 1;
    }
    else if (value < *middle)         // 如果中央元素 > 指定值
      len = half;                     // 將運作區間縮小（移至前半段）以提高效率
    else {           // 如果中央元素 == 指定值
      // 在前半段找 lower_bound
      left = lower_bound(first, middle, value);
      // 在後半段找 lower_bound
      right = upper_bound(++middle, first + len, value);
      return pair<RandomAccessIterator, RandomAccessIterator>(left,right);
    }
  }
  // 整個區間內都沒有吻合的值，那麼應該傳回一對迭代器，指向第一個大於value 的元素。
  return pair<RandomAccessIterator, RandomAccessIterator>(first, first);
}

// 版本一
template <class ForwardIterator, class T>
inline pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& value) {
  // 根據迭代器的種類型（category），採用不同的策略。
  return __equal_range(first, last, value, distance_type(first),
                           iterator_category(first));
}

// 版本二的 forward_iterator 版本
template <class ForwardIterator, class T, class Compare, class Distance>
pair<ForwardIterator, ForwardIterator>
__equal_range(ForwardIterator first, ForwardIterator last, const T& value,
                Compare comp, Distance*, forward_iterator_tag) {
  Distance len = 0;
  distance(first, last, len);
  Distance half;
  ForwardIterator middle, left, right;
  while (len > 0) {
    half = len >> 1;
    middle = first;
    advance(middle, half);
    if (comp(*middle, value)) {
      first = middle;
      ++first;
      len = len - half - 1;
    }
    else if (comp(value, *middle))
      len = half;
    else {
      left = lower_bound(first, middle, value, comp);
      advance(first, len);
      right = upper_bound(++middle, first, value, comp);
      return pair<ForwardIterator, ForwardIterator>(left, right);
    }
  }
  return pair<ForwardIterator, ForwardIterator>(first, first);
}

// 版本二的 random_access_iterator 版本
template <class RandomAccessIterator, class T, class Compare, class Distance>
pair<RandomAccessIterator, RandomAccessIterator>
__equal_range(RandomAccessIterator first, RandomAccessIterator last,
              const T& value, Compare comp, Distance*,
              random_access_iterator_tag) {
  Distance len = last - first;
  Distance half;
  RandomAccessIterator middle, left, right;

  while (len > 0) {
    half = len >> 1;
    middle = first + half;
    if (comp(*middle, value)) {
      first = middle + 1;
      len = len - half - 1;
    }
    else if (comp(value, *middle))
      len = half;
    else {
      left = lower_bound(first, middle, value, comp);
      right = upper_bound(++middle, first + len, value, comp);
      return pair<RandomAccessIterator, RandomAccessIterator>(left,
                                                               right);
    }
  }
  return pair<RandomAccessIterator, RandomAccessIterator>(first, first);
}
```

```
// 版本二
template <class ForwardIterator, class T, class Compare>
inline pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& value,
            Compare comp) {
  return __equal_range(first, last, value, comp, distance_type(first),
                       iterator_category(first));
}

// 版本一
template <class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value) {
  ForwardIterator i = lower_bound(first, last, value);
  return i != last && !(value < *i);
}

// 版本二
template <class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last, const T& value,
                   Compare comp) {
  ForwardIterator i = lower_bound(first, last, value, comp);
  return i != last && !comp(value, *i);
}

// 版本一
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result) {
  while (first1 != last1 && first2 != last2) { // 兩個序列都尚未走完
    if (*first2 < *first1) {    // 序列二的元素比較小
      *result = *first2;        // 登記序列二的元素
      ++first2;                 // 序列二前進 1
    }
    else {                      // 序列二的元素不比較小
      *result = *first1;        // 登記序列一的元素
      ++first1;                 // 序列一前進 1
    }
    ++result;
  }
  // 最後剩餘元素以 copy 複製到目的端。以下兩個序列一定至少有一個為空。
  return copy(first2, last2, copy(first1, last1, result));
}

// 版本二
template <class InputIterator1, class InputIterator2, class OutputIterator,
          class Compare>
```

```
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp) {
  while (first1 != last1 && first2 != last2) { // 兩個序列都尚未走完
    if (comp(*first2, *first1)) {    // 比較兩序列的元素
      *result = *first2;             // 登記序列二的元素
      ++first2;                      // 序列二前進 1
    }
    else {
      *result = *first1;             // 登記序列一的元素
      ++first1;                      // 序列一前進 1
    }
    ++result;
  }
  // 最後剩餘元素以 copy 複製到目的端。以下兩個序列一定至少有一個為空。
  return copy(first2, last2, copy(first1, last1, result));
}

template <class BidirectionalIterator, class Distance>
void __merge_without_buffer(BidirectionalIterator first,
                            BidirectionalIterator middle,
                            BidirectionalIterator last,
                            Distance len1, Distance len2) {
  if (len1 == 0 || len2 == 0) return;
  if (len1 + len2 == 2) {
    if (*middle < *first) iter_swap(first, middle);
    return;
  }
  BidirectionalIterator first_cut = first;
  BidirectionalIterator second_cut = middle;
  Distance len11 = 0;
  Distance len22 = 0;
  if (len1 > len2) {
    len11 = len1 / 2;
    advance(first_cut, len11);
    second_cut = lower_bound(middle, last, *first_cut);
    distance(middle, second_cut, len22);
  }
  else {
    len22 = len2 / 2;
    advance(second_cut, len22);
    first_cut = upper_bound(first, middle, *second_cut);
    distance(first, first_cut, len11);
  }
  rotate(first_cut, middle, second_cut);
  BidirectionalIterator new_middle = first_cut;
  advance(new_middle, len22);
  __merge_without_buffer(first, first_cut, new_middle, len11, len22);
  __merge_without_buffer(new_middle, second_cut, last, len1 - len11,
                          len2 - len22);
}

template <class BidirectionalIterator, class Distance, class Compare>
void __merge_without_buffer(BidirectionalIterator first,
                             BidirectionalIterator middle,
                             BidirectionalIterator last,
                             Distance len1, Distance len2, Compare comp) {
  if (len1 == 0 || len2 == 0) return;
  if (len1 + len2 == 2) {
    if (comp(*middle, *first)) iter_swap(first, middle);
    return;
  }
  BidirectionalIterator first_cut = first;
  BidirectionalIterator second_cut = middle;
  Distance len11 = 0;
  Distance len22 = 0;
  if (len1 > len2) {
    len11 = len1 / 2;
    advance(first_cut, len11);
    second_cut = lower_bound(middle, last, *first_cut, comp);
    distance(middle, second_cut, len22);
  }
  else {
    len22 = len2 / 2;
    advance(second_cut, len22);
    first_cut = upper_bound(first, middle, *second_cut, comp);
    distance(first, first_cut, len11);
  }
  rotate(first_cut, middle, second_cut);
  BidirectionalIterator new_middle = first_cut;
  advance(new_middle, len22);
  __merge_without_buffer(first, first_cut, new_middle, len11, len22, comp);
  __merge_without_buffer(new_middle, second_cut, last, len1 - len11,
                          len2 - len22, comp);
}

template <class BidirectionalIterator1, class BidirectionalIterator2,
          class Distance>
BidirectionalIterator1 __rotate_adaptive(BidirectionalIterator1 first,
                                          BidirectionalIterator1 middle,
                                          BidirectionalIterator1 last,
                                          Distance len1, Distance len2,
                                          BidirectionalIterator2 buffer,
                                          Distance buffer_size) {
  BidirectionalIterator2 buffer_end;
  if (len1 > len2 && len2 <= buffer_size) {
    // 緩衝區足夠安置序列二（較短）
    buffer_end = copy(middle, last, buffer);
    copy_backward(first, middle, last);
    return copy(buffer, buffer_end, first);
  } else if (len1 <= buffer_size) {
    // 緩衝區足夠安置序列一
    buffer_end = copy(first, middle, buffer);
    copy(middle, last, first);
    return copy_backward(buffer, buffer_end, last);
  } else  {
    // 緩衝區仍然不足. 改用 rotate 演算法（不需緩衝區）
    rotate(first, middle, last);
    advance(first, len2);
    return first;
  }
}

template <class BidirectionalIterator1, class BidirectionalIterator2,
          class BidirectionalIterator3>
BidirectionalIterator3 __merge_backward(BidirectionalIterator1 first1,
                                        BidirectionalIterator1 last1,
                                        BidirectionalIterator2 first2,
                                        BidirectionalIterator2 last2,
                                        BidirectionalIterator3 result) {
  if (first1 == last1) return copy_backward(first2, last2, result);
  if (first2 == last2) return copy_backward(first1, last1, result);
  --last1;
  --last2;
  while (true) {
    if (*last2 < *last1) {
      *--result = *last1;
      if (first1 == last1) return copy_backward(first2, ++last2, result);
      --last1;
    }
    else {
      *--result = *last2;
      if (first2 == last2) return copy_backward(first1, ++last1, result);
      --last2;
    }
  }
}

template <class BidirectionalIterator1, class BidirectionalIterator2,
          class BidirectionalIterator3, class Compare>
BidirectionalIterator3 __merge_backward(BidirectionalIterator1 first1,
                                        BidirectionalIterator1 last1,
                                        BidirectionalIterator2 first2,
                                        BidirectionalIterator2 last2,
                                        BidirectionalIterator3 result,
                                        Compare comp) {
  if (first1 == last1) return copy_backward(first2, last2, result);
  if (first2 == last2) return copy_backward(first1, last1, result);
  --last1;
  --last2;
  while (true) {
    if (comp(*last2, *last1)) {
      *--result = *last1;
      if (first1 == last1) return copy_backward(first2, ++last2, result);
      --last1;
    }
    else {
      *--result = *last2;
      if (first2 == last2) return copy_backward(first1, ++last1, result);
      --last2;
    }
  }
}

// 版本一的輔助函式。有緩衝區的情況下。
template <class BidirectionalIterator, class Distance, class Pointer>
void __merge_adaptive(BidirectionalIterator first,
                      BidirectionalIterator middle,
                      BidirectionalIterator last,
                      Distance len1, Distance len2,
                      Pointer buffer, Distance buffer_size) {
  if (len1 <= len2 && len1 <= buffer_size) {
    // case1. 緩衝區足夠安置序列一
    Pointer end_buffer = copy(first, middle, buffer);
    merge(buffer, end_buffer, middle, last, first);
  }
  else if (len2 <= buffer_size) {
    // case 2. 緩衝區足夠安置序列二
    Pointer end_buffer = copy(middle, last, buffer);
    __merge_backward(first, middle, buffer, end_buffer, last);
  }
  else {     // case3.  緩衝區空間不足安置任何一個序列
    BidirectionalIterator first_cut = first;
    BidirectionalIterator second_cut = middle;
    Distance len11 = 0;
    Distance len22 = 0;
    if (len1 > len2) {     // 序列一比較長
      len11 = len1 / 2;
      advance(first_cut, len11);
      second_cut = lower_bound(middle, last, *first_cut);
      distance(middle, second_cut, len22);
    }
    else {                 // 序列二比較不短
      len22 = len2 / 2;    // 計算序列二的一半長度
      advance(second_cut, len22);
      first_cut = upper_bound(first, middle, *second_cut);
      distance(first, first_cut, len11);
    }
    BidirectionalIterator new_middle =
      __rotate_adaptive(first_cut, middle, second_cut, len1 - len11,
                        len22, buffer, buffer_size);
    // 針對左段，遞迴呼叫。
    __merge_adaptive(first, first_cut, new_middle, len11, len22, buffer,
                     buffer_size);
    // 針對右段，遞迴呼叫。
    __merge_adaptive(new_middle, second_cut, last, len1 - len11,
                     len2 - len22, buffer, buffer_size);
  }
}

template <class BidirectionalIterator, class Distance, class Pointer,
          class Compare>
void __merge_adaptive(BidirectionalIterator first,
                      BidirectionalIterator middle,
                      BidirectionalIterator last, Distance len1, Distance len2,
                      Pointer buffer, Distance buffer_size, Compare comp) {
  if (len1 <= len2 && len1 <= buffer_size) {
    Pointer end_buffer = copy(first, middle, buffer);
    merge(buffer, end_buffer, middle, last, first, comp);
  }
  else if (len2 <= buffer_size) {
    Pointer end_buffer = copy(middle, last, buffer);
    __merge_backward(first, middle, buffer, end_buffer, last, comp);
  }
  else {
    BidirectionalIterator first_cut = first;
    BidirectionalIterator second_cut = middle;
    Distance len11 = 0;
    Distance len22 = 0;
    if (len1 > len2) {
      len11 = len1 / 2;
      advance(first_cut, len11);
      second_cut = lower_bound(middle, last, *first_cut, comp);
      distance(middle, second_cut, len22);
    }
    else {
      len22 = len2 / 2;
      advance(second_cut, len22);
      first_cut = upper_bound(first, middle, *second_cut, comp);
      distance(first, first_cut, len11);
    }
    BidirectionalIterator new_middle =
      __rotate_adaptive(first_cut, middle, second_cut, len1 - len11,
                        len22, buffer, buffer_size);
    __merge_adaptive(first, first_cut, new_middle, len11, len22, buffer,
                     buffer_size, comp);
    __merge_adaptive(new_middle, second_cut, last, len1 - len11,
                     len2 - len22, buffer, buffer_size, comp);
  }
}

// 版本一的輔助函式
template <class BidirectionalIterator, class T, class Distance>
inline void __inplace_merge_aux(BidirectionalIterator first,
                                BidirectionalIterator middle,
                                BidirectionalIterator last,
                                 T*, Distance*) {
  Distance len1 = 0;
  distance(first, middle, len1);     // len1 表示序列一的長度
  Distance len2 = 0;
  distance(middle, last, len2);      // len2 表示序列二的長度

  // 注意，本演算法會使用額外的記憶體空間（暫時緩衝區）
  temporary_buffer<BidirectionalIterator, T> buf(first, last);
  if (buf.begin() == 0)     // 記憶體配置失敗
    __merge_without_buffer(first, middle, last, len1, len2);
  else       // 在有暫時緩衝區的情況下進行
    __merge_adaptive(first, middle, last, len1, len2,
                     buf.begin(), Distance(buf.size()));
}

// 版本二的輔助函式
template <class BidirectionalIterator, class T,
          class Distance, class Compare>
inline void __inplace_merge_aux(BidirectionalIterator first,
                          BidirectionalIterator middle,
                          BidirectionalIterator last, T*, Distance*,
                          Compare comp) {
  Distance len1 = 0;
  distance(first, middle, len1);
  Distance len2 = 0;
  distance(middle, last, len2);

  temporary_buffer<BidirectionalIterator, T> buf(first, last);
  if (buf.begin() == 0)
    __merge_without_buffer(first, middle, last, len1, len2, comp);
  else
    __merge_adaptive(first, middle, last, len1, len2,
                  buf.begin(), Distance(buf.size()),
                  comp);
}

// 版本一。合併並取代（覆寫）
template <class BidirectionalIterator>
```

```
inline void inplace_merge(BidirectionalIterator first,
                          BidirectionalIterator middle,
                          BidirectionalIterator last) {
  // 只要有任何一個序列為空，就什麼都不必做。
  if (first == middle || middle == last) return;
  __inplace_merge_aux(first, middle, last, value_type(first),
                      distance_type(first));
}

// 版本二。合併並取代（覆寫）
template <class BidirectionalIterator, class Compare>
inline void inplace_merge(BidirectionalIterator first,
                          BidirectionalIterator middle,
                          BidirectionalIterator last, Compare comp) {
  if (first == middle || middle == last) return;
  __inplace_merge_aux(first, middle, last, value_type(first),
                      distance_type(first), comp);
}

// 版本一。判斷區間二的每個元素值是否都存在於區間一。
// 前提：區間一和區間二都是 sorted ranges.
template <class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2) {
  while (first1 != last1 && first2 != last2) // 兩個區間都尚未走完
    if (*first2 < *first1)      // 序列二的元素小於序列一的元素
      return false;              // 「涵蓋」的情況必然不成立
    else if(*first1 < *first2)  // 序列二的元素大於序列一的元素
      ++first1;                  //序列一前進 1
    else                        // *first1 == *first2
      ++first1, ++first2;        // 兩序列各自前進 1

  return first2 == last2;  // 有一個序列走完了，判斷最後一關
}

// 版本二。判斷序列一內是否有個子序列，其與序列二的每個對應元素都滿足二元運算 comp。
// 前提：序列一和序列二都是 sorted ranges.
template <class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2, Compare comp) {
  while (first1 != last1 && first2 != last2)
    if (comp(*first2, *first1))
      return false;
    else if(comp(*first1, *first2))
      ++first1;
    else
      ++first1, ++first2;

  return first2 == last2;
```

```
}

// 聯集，求存在於[first1,last1) 或存在於 [first2,last2) 的所有元素。
// 注意，set 是一種 sorted range。這是以下演算法的前提。
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result) {
  // 當兩個區間都不為空白區間時，執行以下動作…
  while (first1 != last1 && first2 != last2) {
    // 在兩區間內分別移動迭代器。首先將元素值較小者（假設為A區）記錄於標的區，
    // 然後移動A區迭代器使之前進；同時間之另一個區迭代器不動。然後，再進行
    // 新一次的比大小、記錄小值、迭代器移動…。直到兩區中有一區為空白。
    if (*first1 < *first2) {
      *result = *first1;
      ++first1;
    }
    else if (*first2 < *first1) {
      *result = *first2;
      ++first2;
    }
    else { // *first2 == *first1
      *result = *first1;
      ++first1;
      ++first2;
    }
    ++result;
  }

  // 只要兩區中有一區成為空白，就結束上述的 while 迴圈。
  // 以下將剩餘的（非空白的）區間的所有元素拷貝到目的端。
  // 此刻的 [first1,last1)和[first2,last2)之中有一個是空白區間。
  return copy(first2, last2, copy(first1, last1, result));
}

template <class InputIterator1, class InputIterator2, class OutputIterator,
          class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp) {
  while (first1 != last1 && first2 != last2) {
    if (comp(*first1, *first2)) {
      *result = *first1;
      ++first1;
    }
    else if (comp(*first2, *first1)) {
      *result = *first2;
      ++first2;
    }
    else {
      *result = *first1;
      ++first1;
      ++first2;
    }
    ++result;
  }
  return copy(first2, last2, copy(first1, last1, result));
}

// 交集，求存在於[first1,last1) 且存在於 [first2,last2) 的所有元素。
// 注意，set 是一種 sorted range。這是以下演算法的前提。
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                OutputIterator result) {
  // 當兩個區間都不為空白區間時，執行以下動作…
  while (first1 != last1 && first2 != last2)
    // 在兩區間內分別移動迭代器，直到遇到元素值相同，暫停，將該值記錄於標的區，
    // 再繼續移動迭代器…。直到兩區中有一區為空白。
    if (*first1 < *first2)
      ++first1;
    else if (*first2 < *first1)
      ++first2;
    else {
      *result = *first1;
      ++first1;
      ++first2;
      ++result;
    }
  return result;
}

template <class InputIterator1, class InputIterator2, class OutputIterator,
          class Compare>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                OutputIterator result, Compare comp) {
  while (first1 != last1 && first2 != last2)
    if (comp(*first1, *first2))
      ++first1;
    else if (comp(*first2, *first1))
      ++first2;
    else {
      *result = *first1;
      ++first1;
      ++first2;
      ++result;
    }
  return result;
}

// 差集，求存在於[first1,last1) 且不存在於 [first2,last2) 的所有元素。
// 注意，set 是一種 sorted range。這是以下演算法的前提。
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result) {
  // 當兩個區間都不為空白區間時，執行以下動作…
  while (first1 != last1 && first2 != last2)
    // 在兩區間內分別移動迭代器。當第一區間的元素等於第二區間的元素（表示此值
    // 同時存在於兩區），就讓兩區同時前進；當第一區間的元素大於第二區間的元素，
    // 就讓第二區間前進；有了這兩種處理，就保證當第一區間的元素小於第二區間的
    // 元素時，第一區間的元素只存在於第一區間中，不存在於第二區間。於是將它
    // 記錄於目標區。
    if (*first1 < *first2) {
      *result = *first1;
      ++first1;
      ++result;
    }
    else if (*first2 < *first1)
      ++first2;
    else {   // *first2 == *first1
      ++first1;
      ++first2;
    }
  return copy(first1, last1, result);
}

template <class InputIterator1, class InputIterator2, class OutputIterator,
          class Compare>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result, Compare comp) {
  while (first1 != last1 && first2 != last2)
    if (comp(*first1, *first2)) {
      *result = *first1;
      ++first1;
      ++result;
    }
    else if (comp(*first2, *first1))
      ++first2;
    else {
      ++first1;
      ++first2;
    }
  return copy(first1, last1, result);
}
```

```
// 對稱差集，求存在於[first1,last1) 且不存在於 [first2,last2) 的所有元素，
// 以及存在於[first2,last2) 且不存在於 [first1,last1) 的所有元素。
// 注意，上述定義只有在「元素值獨一無二」的情況下才成立。如果將 set 一般化，
// 允許出現重複元素，那麼 set-symmetric-difference 的定義應該是：
// 如果某值在[first1,last1) 出現n次，在 [first2,last2) 出現m次，
// 那麼它在 result range 中應該出現 abs(n-m) 次。
// 注意，set 是一種 sorted range。這是以下演算法的前提。
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference(InputIterator1 first1,
                                        InputIterator1 last1,
                                        InputIterator2 first2,
                                        InputIterator2 last2,
                                        OutputIterator result) {
  // 當兩個區間都不為空白區間時，執行以下動作…
  while (first1 != last1 && first2 != last2)
    // 在兩區間內分別移動迭代器。當兩區間內的當值元素相等，就讓兩區同時前進；
    // 當兩區間內的當值元素不等，就記錄較小值於目標區，並令較小值所在區間前進。
    if (*first1 < *first2) {
      *result = *first1;
      ++first1;
      ++result;
    }
    else if (*first2 < *first1) {
      *result = *first2;
      ++first2;
      ++result;
    }
    else {   // *first2 == *first1
      ++first1;
      ++first2;
    }
  return copy(first2, last2, copy(first1, last1, result));
}

template <class InputIterator1, class InputIterator2, class OutputIterator,
          class Compare>
OutputIterator set_symmetric_difference(InputIterator1 first1,
                                        InputIterator1 last1,
                                        InputIterator2 first2,
                                        InputIterator2 last2,
                                        OutputIterator result, Compare comp) {
  while (first1 != last1 && first2 != last2)
    if (comp(*first1, *first2)) {
      *result = *first1;
      ++first1;
      ++result;
    }
    else if (comp(*first2, *first1)) {
      *result = *first2;
      ++first2;
      ++result;
    }
    else {
      ++first1;
      ++first2;
    }
  return copy(first2, last2, copy(first1, last1, result));
}

// 版本一
template <class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last) {
  if (first == last) return first;
  ForwardIterator result = first;
  while (++first != last)
    if (*result < *first) result = first;
  return result;
}

// 版本二
template <class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                  Compare comp) {
  if (first == last) return first;
  ForwardIterator result = first;
  while (++first != last)
    if (comp(*result, *first)) result = first;
  return result;
}

// 版本一
template <class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last) {
  if (first == last) return first;
  ForwardIterator result = first;
  while (++first != last)
    if (*first < *result) result = first;
  return result;
}

// 版本二
template <class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                             Compare comp) {
  if (first == last) return first;
  ForwardIterator result = first;
  while (++first != last)
```

```
    if (comp(*first, *result)) result = first;
  return result;
}

// 版本一
template <class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last) {
  if (first == last) return false;   // 空範圍
  BidirectionalIterator i = first;
  ++i;
  if (i == last) return false;       // 只有一個元素
  i = last;  // i 指向尾端
  --i;

  for(;;) {
    BidirectionalIterator ii = i;
    --i;
    // 以上，鎖定一組（兩個）相鄰元素
    if (*i < *ii) {    // 如果前一個元素小於後一個元素
      BidirectionalIterator j = last;     // 令 j指向尾端
      while (!(*i < *--j));      // 由尾端往前找，直到遇上比 *i 大的元素
      iter_swap(i, j);           // 交換 i, j
      reverse(ii, last);         // 將 ii 之後的元素全部逆向重排
      return true;
    }
    if (i == first) {            // 進行至最前面了
      reverse(first, last);      // 全部逆向重排
      return false;
    }
  }
}

// 版本二
template <class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last,
                      Compare comp) {
  if (first == last) return false;   // 空範圍
  BidirectionalIterator i = first;
  ++i;
  if (i == last) return false;       // 只有一個元素
  i = last;  // i 指向尾端
  --i;

  for(;;) {
    BidirectionalIterator ii = i;
    --i;
    // 以上，鎖定一組（兩個）相鄰元素
    if (comp(*i, *ii)) {    // 如果前一個元素與後一個元素滿足 comp 條件
```

```
      BidirectionalIterator j = last;     // 令 j指向尾端
      while (!comp(*i, *--j));  // 由尾端往前找，直到遇上符合條件的元素
      iter_swap(i, j);          // 交換 i, j
      reverse(ii, last);        // 將 ii 之後的元素全部逆向重排
      return true;
    }
    if (i == first) {           // 進行至最前面了
      reverse(first, last);     // 全部逆向重排
      return false;
    }
  }
}

// 版本一
template <class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                  BidirectionalIterator last) {
  if (first == last) return false;   // 空範圍
  BidirectionalIterator i = first;
  ++i;
  if (i == last) return false;       // 只有一個元素
  i = last;  // i 指向尾端
  --i;

  for(;;) {
    BidirectionalIterator ii = i;
    --i;
    // 以上，鎖定一組（兩個）相鄰元素
    if (*ii < *i) {    // 如果前一個元素大於後一個元素
      BidirectionalIterator j = last;     // 令 j指向尾端
      while (!(*--j < *i));     // 由尾端往前找，直到遇上比 *i 小的元素
      iter_swap(i, j);          // 交換 i, j
      reverse(ii, last);        // 將 ii 之後的元素全部逆向重排
      return true;
    }
    if (i == first) {           // 進行至最前面了
      reverse(first, last);     // 全部逆向重排
      return false;
    }
  }
}

// 版本二
template <class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last,
                  Compare comp) {
  if (first == last) return false;
  BidirectionalIterator i = first;
  ++i;
  if (i == last) return false;
  i = last;
  --i;

  for(;;) {
    BidirectionalIterator ii = i;
    --i;
    if (comp(*ii, *i)) {
      BidirectionalIterator j = last;
      while (!comp(*--j, *i));
      iter_swap(i, j);
      reverse(ii, last);
      return true;
    }
    if (i == first) {
      reverse(first, last);
      return false;
    }
  }
}

// 版本一
template <class InputIterator, class ForwardIterator>
InputIterator find_first_of(InputIterator first1, InputIterator last1,
                            ForwardIterator first2, ForwardIterator last2)
{
  for ( ; first1 != last1; ++first1) // 遍訪序列一
    // 以下，根據序列二的每個元素
    for (ForwardIterator iter = first2; iter != last2; ++iter)
      if (*first1 == *iter) // 如果序列一的元素等於序列二的元素
        return first1;      // 找到了，結束。
  return last1;
}

// 版本二
template <class InputIterator, class ForwardIterator, class BinaryPredicate>
InputIterator find_first_of(InputIterator first1, InputIterator last1,
                            ForwardIterator first2, ForwardIterator last2,
                            BinaryPredicate comp)
{
  for ( ; first1 != last1; ++first1) // 遍訪序列一
    // 以下，根據序列二的每個元素
    for (ForwardIterator iter = first2; iter != last2; ++iter)
      if (comp(*first1, *iter)) // 如果序列一和序列二的元素滿足comp 條件
        return first1;          // 找到了，結束。
  return last1;
}
```

```
// 搜尋[first1, last1) 中的子序列 [first2, last2) 的最後出現點
// 以下是forward iterators 版
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 __find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            forward_iterator_tag, forward_iterator_tag)
{
  if (first2 == last2)      // 如果搜尋目標是空的，
    return last1;           // 傳回 last1 表示該「空子序列」的最後出現點
  else {
    ForwardIterator1 result = last1;
    while (1) {
      // 以下利用search()搜尋某個子序列的首次出現點。找不到的話傳回last1
      ForwardIterator1 new_result = search(first1, last1, first2, last2);
      if (new_result == last1)   // 沒找到
        return result;
      else {
        result = new_result;     // 調動一下標兵，準備下一個搜尋行動
        first1 = new_result;
        ++first1;
      }
    }
  }
}

template <class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate>
ForwardIterator1 __find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            forward_iterator_tag, forward_iterator_tag,
                            BinaryPredicate comp)
{
  if (first2 == last2)
    return last1;
  else {
    ForwardIterator1 result = last1;
    while (1) {
      ForwardIterator1 new_result = search(first1, last1, first2, last2, comp);
      if (new_result == last1)
        return result;
      else {
        result = new_result;
        first1 = new_result;
        ++first1;
      }
    }
  }
}
```

```
// 以下是bidirectional iterators 版，需用到partial specialization.
#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator1
__find_end(BidirectionalIterator1 first1, BidirectionalIterator1 last1,
           BidirectionalIterator2 first2, BidirectionalIterator2 last2,
           bidirectional_iterator_tag, bidirectional_iterator_tag)
{
  // 由於搜尋的是「最後出現地點」，因此反向搜尋比較快。利用reverse_iterator.
  typedef reverse_iterator<BidirectionalIterator1> reviter1;
  typedef reverse_iterator<BidirectionalIterator2> reviter2;

  reviter1 rlast1(first1);
  reviter2 rlast2(first2);
  // 搜尋時，序列和子序列統統逆轉方向
  reviter1 rresult = search(reviter1(last1), rlast1, reviter2(last2), rlast2);

  if (rresult == rlast1)   // 沒找到
    return last1;
  else {                   // 找到了
    BidirectionalIterator1 result = rresult.base(); // 轉回正常（非逆向）迭代器
    advance(result, -distance(first2, last2)); // 調整回到子序列的起頭處
    return result;
  }
}

template <class BidirectionalIterator1, class BidirectionalIterator2,
          class BinaryPredicate>
BidirectionalIterator1
__find_end(BidirectionalIterator1 first1, BidirectionalIterator1 last1,
           BidirectionalIterator2 first2, BidirectionalIterator2 last2,
           bidirectional_iterator_tag, bidirectional_iterator_tag,
           BinaryPredicate comp)
{
  typedef reverse_iterator<BidirectionalIterator1> reviter1;
  typedef reverse_iterator<BidirectionalIterator2> reviter2;

  reviter1 rlast1(first1);
  reviter2 rlast2(first2);
  reviter1 rresult = search(reviter1(last1), rlast1, reviter2(last2), rlast2,
                            comp);

  if (rresult == rlast1)
    return last1;
  else {
    BidirectionalIterator1 result = rresult.base();
    advance(result, -distance(first2, last2));
    return result;
  }
```

```
}
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// 分派函式（Dispatching functions）
template <class ForwardIterator1, class ForwardIterator2>
inline ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2)
{
#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
  typedef typename iterator_traits<ForwardIterator1>::iterator_category
          category1;
  typedef typename iterator_traits<ForwardIterator2>::iterator_category
          category2;
  return __find_end(first1, last1, first2, last2, category1(), category2());
#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
  return __find_end(first1, last1, first2, last2,
                    forward_iterator_tag(), forward_iterator_tag());
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */
}

template <class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate>
inline ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2,
         BinaryPredicate comp)
{
#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
  typedef typename iterator_traits<ForwardIterator1>::iterator_category
          category1;
  typedef typename iterator_traits<ForwardIterator2>::iterator_category
          category2;
  return __find_end(first1, last1, first2, last2, category1(), category2(),
                    comp);
#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
  return __find_end(first1, last1, first2, last2,
                    forward_iterator_tag(), forward_iterator_tag(),
                    comp);
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */
}

template <class RandomAccessIterator, class Distance>
bool __is_heap(RandomAccessIterator first, RandomAccessIterator last,
               Distance*)
{
  const Distance n = last - first;

  Distance parent = 0;
  for (Distance child = 1; child < n; ++child) {
    if (first[parent] < first[child])
      return false;
    if ((child & 1) == 0)
      ++parent;
  }
  return true;
}

template <class RandomAccessIterator>
inline bool is_heap(RandomAccessIterator first, RandomAccessIterator last)
{
  return __is_heap(first, last, distance_type(first));
}


template <class RandomAccessIterator, class Distance, class StrictWeakOrdering>
bool __is_heap(RandomAccessIterator first, RandomAccessIterator last,
               StrictWeakOrdering comp,
               Distance*)
{
  const Distance n = last - first;

  Distance parent = 0;
  for (Distance child = 1; child < n; ++child) {
    if (comp(first[parent], first[child]))
      return false;
    if ((child & 1) == 0)
      ++parent;
  }
  return true;
}

template <class RandomAccessIterator, class StrictWeakOrdering>
inline bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
                    StrictWeakOrdering comp)
{
  return __is_heap(first, last, comp, distance_type(first));
}


template <class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last)
{
  if (first == last)
    return true;

  ForwardIterator next = first;
  for (++next; next != last; first = next, ++next) {
    if (*next < *first)
      return false;
  }

  return true;
}

template <class ForwardIterator, class StrictWeakOrdering>
bool is_sorted(ForwardIterator first, ForwardIterator last,
            StrictWeakOrdering comp)
{
  if (first == last)
    return true;

  ForwardIterator next = first;
  for (++next; next != last; first = next, ++next) {
    if (comp(*next, *first))
      return false;
  }

  return true;
}

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma reset woff 1209
#endif

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_ALGO_H */

// Local Variables:
// mode:C++
// End:
```