Алгоритмы

Соглашения об именовании параметров

В качестве типов для параметров-итераторов *first, last, result* указываются:

- *InIter* — итератор чтения (input),
- *OutIter* — итератор записи (output),
- *FwdIter* — однонаправленный итератор (forward),
- *BidiIter* — двунаправленный итератор (bidirectional),
- *RandIter* — итератор произвольного доступа (random).

В качестве типа значения для последовательностей указывается *T*. Для итератора из диапазона используется переменная *p*.

Во всех алгоритмах, связанных с копированием данных, предполагается, что в результирующей последовательности зарезервировано достаточно места для размещения всех получаемых элементов.

Всюду при определении сложности под *N* понимается разность итераторов *distance*(*first, last*) (если *N* имеет индекс, то подразумевается, что такой же индекс имеют и итераторы). Если сложность является постоянной, то она не указывается.

Алгоритмы перечисляются в алфавитном порядке их имен.

Алгоритмы общего назначения (заголовочный файл <algorithm>)

```
FwdIter adjacent_find(FwdIter first, FwdIter last[,
    BinaryPredicate pred])
```

Находит первую пару соседних элементов из диапазона [*first, last*), которые равны (или, при наличии предиката *pred*(*p, *(*p* + 1)), для которых данный предикат возвращает *true*). Возвращает итератор на первый элемент найденной пары или *last*, если пара не найдена.

Сложность линейная (не более *N* + 1 вызовов *pred*).

```
bool binary_search(FwdIter first, FwdIter last,
    const T& value[, Compare comp])
```

Использует двоичный поиск для проверки того, содержится ли в диапазоне [*first, last*) значение *value* (если значение найдено, то возвращает *true*, иначе *false*). Содержимое диапазона должно быть предварительно отсортировано в соответствии с порядком, задаваемым предикатом *comp*(**p1, *p2*) или (по умолчанию) операцией <.

Сложность логарифмическая (не более log *N* + 2 сравнений).

```
OutIter copy(InIter first, InIter last,
    OutIter result)
```

Копирует элементы из [*first, last*) в итератор вывода *result* и возвращает позицию за последним скопированным элементом. Итератор *result* не может находиться в исходном диапазоне [*first, last*), но другие части выходного диапазона могут накладываться на исходный диапазон.

Сложность линейная (*N* присваиваний).

```
BidiIter2 copy_backward(BidiIter1 first,
    BidiIter1 last, BidiIter2 result)
```

Выполняет те же действия (и имеет те же ограничения), что и *copy*, но перебирает исходные данные в обратном порядке: от элемента, предшествующего *last*, до *first*. Итератор *result* должен указывать на элемент, следующий за концом выходной последовательности; возвращаемое значение — это итератор, указывающий на первый элемент выходной последовательности.

Сложность линейная (*N* присваиваний).

```
difference_type count(InIter first, InIter last,
    const T& value)
```

Возвращает количество элементов в диапазоне [*first, last*), равных *value*.

Сложность линейная (*N* сравнений).

```
difference_type count_if(InIter first, InIter last,
    Predicate pred)
```

Возвращает количество элементов в диапазоне [*first, last*), для которых выражение *pred*(**p*) равно *true*.

Сложность линейная (*N* вызовов *pred*).

```
bool equal(InIter1 first1, InIter1 last1,
    InIter2 first2[, BinaryPredicate pred])
```

Возвращает *true*, если два диапазона содержат одни и те же элементы в одинаковом порядке. Первый диапазон — [*first1, last1*), второй начинается с *first2* и имеет такую же длину; диапазоны могут перекрываться. Для сравнения используется предикат *pred*(**p1, *p2*) или (по умолчанию) операция ==.

Сложность линейная (не более *N1* сравнений).

```
pair<FwdIter, FwdIter> equal_range(FwdIter first,
    FwdIter last, const T& value[, Compare comp])
```

Проверяет, содержится ли в диапазоне [*first, last*) значение *value*, и возвращает пару итераторов, которые указывают на начало диапазона, содержащего *value*, и на элемент за концом этого диапазона (если значение не найдено, то оба итератора указывают на позицию в диапазоне, в которую можно вставить *value*, не нарушая порядка сортировки). Содержимое диапазона должно быть предварительно отсортировано в соответствии с порядком, задаваемым предикатом *comp*(**p1, *p2*) или (по умолчанию) операцией <.

Сложность логарифмическая (не более 2*log *N* + 1 сравнений).

```
void fill(FwdIter first, FwdIter last,
    const T& value)
```

Заполняет выходной диапазон [*first, last*) значениями *value*.

Сложность линейная (*N* присваиваний).

```
void fill_n(OutIter first, Size n, const T& value)
```

Заполняет выходной диапазон из *n* элементов, начиная с *first*, значениями *value*.

Сложность линейная (*n* присваиваний).

```
InIter find(InIter first, InIter last,
    const T& value)
```

Возвращает итератор, указывающий на первое вхождение элемента *value* в диапазоне [*first, last*), или *last*, если элемент *value* отсутствует. Для сравнения элементов используется операция ==.

Сложность линейная (не более *N* сравнений).

```
FwdIter1 find_end(FwdIter1 first1, FwdIter1 last1,
    FwdIter2 first2, FwdIter2 last2[,
    BinaryPredicate pred])
```

Находит последнюю (самую правую) подпоследовательность [*first2, last2*) в диапазоне [*first1, last1*). Возвращает итератор, который указывает на начало найденной подпоследовательности, или *last1*, если подпоследовательность не найдена. Для сравнения элементов используется предикат *pred*(**p1, *p2*) или (по умолчанию) операция ==.

Сложность линейная (не более *N1*N2* сравнений).

```
FwdIter1 find_first_of(FwdIter1 first1,
    FwdIter1 last1, FwdIter2 first2, FwdIter2 last2[,
    BinaryPredicate pred])
```

Находит первое вхождение любого элемента подпоследовательности [*first2, last2*) в диапазон [*first1, last1*); возвращает итератор, который указывает на найденный элемент, или *last1*, если элемент не найден. Для сравнения элементов используется предикат *pred*(**p1, *p2*) или (по умолчанию) операция ==.

Сложность линейная (не более *N1*N2* сравнений).

```
InIter find_if(InIter first, InIter last,
    Predicate pred)
```

Возвращает итератор, указывающий для диапазона [*first, last*) на первое вхождение элемента, для которого выражение

pred(**p*) возвращает *true*; если требуемые элементы отсутствуют, то возвращает *last*.

Сложность линейная (не более *N* вызовов *pred*).

```
Func for_each(InIter first, InIter last, Func f)
```

Вызывает функциональный объект *f* (как *f*(**p*)) для всех элементов из диапазона [*first, last*) и возвращает этот же функциональный объект.

Сложность линейная (*N* вызовов *f*).

```
void generate(FwdIter first, FwdIter last,
    Generator gen)
```

Заполняет диапазон [*first, last*), последовательно присваивая элементам диапазона результат вызова функционального объекта *gen* (как *gen*()).

Сложность линейная (*N* вызовов *gen*).

```
void generate_n(OutIter first, Size n,
    Generator gen)
```

Заполняет последовательность, начинающуюся с позиции *first*, *n* элементами, полученными в результате вызова функционального объекта *gen* (как *gen*()).

Сложность линейная (*n* вызовов *gen*).

```
bool includes(InIter1 first1, InIter1 last1,
    InIter2 first2, InIter2, last2[, Compare comp])
```

Возвращает *true*, если все элементы предварительно отсортированной последовательности [*first2, last2*) содержатся в предварительно отсортированной последовательности [*first1, last1*), и *false* в противном случае (фактически ищется вхождение подпоследовательности [*first2, last2*) в диапазон [*first1, last1*)). Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (не более 2*(*N1* + *N2*) – 1 сравнений).

```
void inplace_merge(BidiIter first, BidiIter middle,
    BidiIter last[, Compare comp])
```

Выполняет слияние двух предварительно отсортированных частей [*first, middle*) и [*middle, last*) последовательности на месте, в результате чего создается единый отсортированный диапазон [*first, last*). Слияние является устойчивым; кроме того, в полученном диапазоне равные элементы из первого диапазона будут располагаться перед равными им элементами из второго диапазона. Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (*N* + 1 сравнений) или (при нехватке памяти) *N**log *N* сравнений.

```
void iter_swap(FwdIter1 a, FwdIter2 b)
```

Меняет местами значения, на которые указывают итераторы *a* и *b*.

```
bool lexicographical_compare(InIter1 first1, InIter1
    last1, InIter2 first2, InIter2 last2[,
    Compare comp])
```

Возвращает *true*, если последовательность [*first1, last1*) «меньше» (в лексикографическом смысле), чем последовательность [*first2, last2*), и *false* в противном случае (в частности, если последовательности равны, то возвращается *false*, а если первая последовательность является собственным префиксом второй, то возвращается *true*). Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (не более min{*N1, N2*} сравнений).

```
FwdIter lower_bound(FwdIter first, FwdIter last,
    const T& value[, Compare comp])
```

Проверяет, содержится ли в диапазоне [*first, last*) значение *value*, и возвращает итератор, который указывает на первое вхождение *value* (если значение не найдено, то итератор указывает на позицию в диапазоне, в которую можно вставить *value*, не нарушая порядка сортировки). Содержимое диапазона должно быть предварительно отсортировано в соответствии с порядком, задаваемым предикатом *comp*(**p1, *p2*) или (по умолчанию) операцией <.

Сложность логарифмическая (не более log *N* + 1 сравнений).

```
void make_heap(RandIter first, RandIter last[,
    Compare comp])
```

Переупорядочивает элементы диапазона [*first, last*), получая из него *кучу* (т. е. очередь с приоритетом, для которой первый элемент всегда больше остальных, а добавление нового элемента или удаление первого элемента может быть произведено за логарифмическое время, и результат тоже будет кучей). Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (не более 3**N* сравнений).

```
const T& max(const T& a, const T& b[, Compare comp])
```

Возвращает большее из значений *a* и *b* (при их равенстве возвращается *a*). Для сравнения элементов используется предикат *comp*(*a, b*) или (по умолчанию) операция <.

```
FwdIter max_element(FwdIter first, FwdIter last[,
    Compare comp])
```

Возвращает позицию первого наибольшего элемента в диапазоне [*first, last*). Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (max{*N* – 1, 0} сравнений).

```
OutIter merge(InIter1 first1, InIter1 last1,
    InIter2 first2, InIter2 last2, OutIter result[,
    Compare comp])
```

Выполняет слияние двух предварительно отсортированных диапазонов [*first1, last1*) и [*first2, last2*) и копирование результата в последовательность, начиная с *result* (в выходной последовательности должно быть достаточно места для полученного набора данных). Возвращает выходной итератор, указывающий на позицию за концом добавленного набора данных. Выходной диапазон не должен накладываться ни на один из исходных диапазонов. Слияние является устойчивым; кроме того, в полученном диапазоне равные элементы из первого диапазона будут располагаться перед равными им элементами из второго диапазона. Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (не более *N1* + *N2* – 1 сравнений).

```
const T& min(const T& a, const T& b[, Compare comp])
```

Возвращает меньшее из значений *a* и *b* (при их равенстве возвращается *a*). Для сравнения элементов используется предикат *comp*(*a, b*) или (по умолчанию) операция <.

```
FwdIter min_element(FwdIter first, FwdIter last[,
    Compare comp])
```

Возвращает позицию первого наименьшего элемента в диапазоне [*first, last*). Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (max{*N* – 1, 0} сравнений).

```
pair<InIter1, InIter2> mismatch(InIter1 first1,
    InIter1 last1, InIter2 first2[,
    BinaryPredicate pred])
```

Выполняет попарное сравнение элементов из диапазона [*first1, last1*) и диапазона, начинающегося с *first2* и имеющего длину, не меньшую, чем длина первого диапазона. Возвращает первую пару различных элементов (или, если все элементы первого диапазона совпадают с соответствующими элементами второго диапазона, пару из итератора *last1* и соответствующего итератора для второго диапазона). Для сравнения элементов используется предикат *pred*(**p1, *p2*) или (по умолчанию) операция ==.

Сложность линейная (не более *N1* сравнений).

```
bool next_permutation(BidiIter first, BidiIter last[,
    Compare comp])
```

Переупорядочивает содержимое диапазона [*first, last*), создавая следующую перестановку из набора лексикографически упорядоченных перестановок элементов данного диапазона. Возвращает *true*, если перестановка была создана успешно, или *false*, если исходный диапазон представлял собой последнюю (в лексикографическом порядке) перестановку; в этом последнем случае генерируется первая в лексикографическом порядке перестановка (в которой все элементы расположены в порядке возрастания). Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (не более *N*/2 перемещений).

```
void nth_element(RandIter first, RandIter nth,
    RandIter last[, Compare comp])
```

Переупорядочивает диапазон [*first, last*) таким образом, чтобы в позиции *nth* размещался элемент, который стоял бы на этом месте в случае, если бы весь диапазон был отсортирован. Кроме того, в результате выполнения данного алгоритма все элементы в диапазоне [*first, nth*) не будут превосходить элементы из диапазона [*nth, last*). Алгоритм не является устойчивым: если имеется несколько элементов, которые при сортировке могли бы оказаться на позиции *nth*, то нельзя сказать, какой из них будет перемещен на эту позицию. Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность в среднем линейная (около *N* сравнений).

```
void partial_sort(RandIter first, RandIter middle,
    RandIter last[, Compare comp])
```

Частично сортирует элементы диапазона [*first, last*), размещая отсортированные элементы в диапазоне [*first, middle*), оставшиеся элементы никак не упорядочиваются. Алгоритм не является устойчивым. Для сравнения элементов используется предикат comp(**p1, *p2*) или (по умолчанию) операция <.

Сложность: примерно *N**log(*middle* – *first*) сравнений.

```
RandIter partial_sort_copy(InIter first1,
    InIter last1, RandIter first2, RandIter last2[,
    Compare comp])
```

Частично сортирует элементы из диапазона [*first1, last1*) и копирует отсортированную часть в диапазон [*first2, last2*). Размер сортируемой части определяется размером второго диапазона: если он меньше первого, то сортируется часть первого диапазона, если он больше или равен размеру первого диапазона, то сортируется весь первый диапазон.

Возвращается итератор второго диапазона, указывающий на позицию за концом отсортированного набора данных, добавленного из первого диапазона. Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность: примерно *N1**log(min{*N1, N2*}) сравнений.

```
BidiIter partition(BidiIter first, BidiIter last,
    Predicate pred)
```

Меняет местами элементы диапазона [*first, last*) так, чтобы все элементы, удовлетворяющие предикату *pred*, были расположены перед теми, которые ему не удовлетворяют. Относительный порядок следования элементов не сохраняется. Алгоритм возвращает итератор, указывающий на первый элемент, для которого *pred* возвращает *false* (или итератор *last*, если таких элементов нет).

Сложность линейная (*N* вызовов *pred* и не более *N*/2 перемещений элементов).

```
void pop_heap(RandIter first, RandIter last[,
    Compare comp])
```

При условии, что диапазон [*first, last*) является кучей, перемещает первый (наибольший) элемент этой кучи в конец этого диапазона (т. е. в элемент *(*last* – 1)) и гарантирует, что элементы, оставшиеся в диапазоне [*first, last* – 1), образуют кучу. Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность логарифмическая (не более 2*log *N* сравнений).

```
bool prev_permutation(BidiIter first, BidiIter last[,
    Compare comp])
```

Переупорядочивает содержимое диапазона [*first, last*), создавая предыдущую перестановку из набора лексикографически упорядоченных перестановок элементов данного диапазона. Возвращает *true*, если перестановка была создана успешно, или *false*, если исходный диапазон представлял собой первую (в лексикографическом порядке) перестановку; в этом последнем случае генерируется последняя в лексикографическом порядке перестановка (в которой все элементы расположены в порядке убывания). Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (не более *N*/2 перемещений).

```
void push_heap(RandIter first, RandIter last[,
    Compare comp])
```

При условии, что диапазон [*first, last* – 1) является кучей, добавляет в эту кучу элемент, расположенный в позиции *last* – 1, формируя тем самым кучу в диапазоне [*first, last*). Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность логарифмическая (не более log *N* сравнений).

```
void random_shuffle(RandIter first, RandIter last[,
    RandomGenerator rand])
```

Случайным образом изменяет порядок элементов из диапазона [*first, last*). Для генерации случайных чисел по умолчанию используется встроенный генератор случайных чисел с равномерным распределением; может также использоваться явно заданный генератор *rand*(*n*), возвращающий целое случайное число в диапазоне [0, *n*).

Сложность линейная (*N* + 1 перемещений элементов).

```
FwdIter remove(FwdIter first, FwdIter last,
    const T& value)
```

«Удаляет» из диапазона [*first, last*) элементы, равные *value* (для сравнения элементов используется операция ==). «Удаляемые» элементы перемещаются в конец данного диапазона. Возвращает итератор *middle*, расположенный на первом «удаленном» элементе (таким образом, преобразованный набор без удаленных элементов размещается в диапазоне [*first, middle*)). Относительный порядок оставшихся в диапазоне [*first, middle*) элементов сохраняется.

Сложность линейная (*N* сравнений).

```
OutIter remove_copy(InIter first, InIter last,
    OutIter result, const T& value)
```

Копирует в диапазон, начинающийся с *result*, все элементы диапазона [*first, last*), кроме элементов, равных *value* (для сравнения элементов используется операция ==). Возвращает позицию за последним скопированным элементом в полученном диапазоне. Относительный порядок элементов в полученном диапазоне сохраняется. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная (*N* сравнений).

```
OutIter remove_copy_if(InIter first, InIter last,
    OutIter result, const T& value, Predicate pred)
```

Копирует в диапазон, начинающийся с *result*, все элементы диапазона [*first, last*), кроме элементов, для которых *pred* возвращает *true*. Возвращает позицию за последним скопированным элементом в полученном диапазоне. Относительный порядок элементов в полученном диапазоне сохраняется. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная (*N* сравнений).

```
FwdIter remove_if(FwdIter first, FwdIter last,
    Predicate pred)
```

«Удаляет» из диапазона [*first, last*) элементы, для которых *pred* возвращает *true*. «Удаляемые» элементы перемещаются в конец данного диапазона. Возвращает итератор *middle*, расположенный на первом «удаленном» элементе (таким образом, преобразованный набор без удаленных элементов размещается в диапазоне [*first, middle*)). Относительный порядок оставшихся в диапазоне [*first, middle*) элементов сохраняется.

Сложность линейная (*N* сравнений).

```
void replace(FwdIter first, FwdIter last,
    const T& old_value, const T& new_value)
```

Заменяет в диапазоне [*first, last*) все вхождения значения *old_value* на *new_value*.

Сложность линейная (*N* сравнений).

```
OutIter replace_copy(InIter first, InIter last,
    OutIter result, const T& old_value,
    const T& new_value)
```

Копирует элементы диапазона [*first, last*) в диапазон, начинающийся с *result*, заменяя при этом все элементы, равные *old_value*, на *new_value*. Возвращает позицию за последним скопированным элементом в полученном диапазоне. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная (*N* сравнений).

```
OutIter replace_copy_if(InIter first, InIter last,
    OutIter result, Predicate pred,
    const T& new_value)
```

Копирует элементы диапазона [*first, last*) в диапазон, начинающийся с *result*, заменяя при этом все элементы, для которых *pred* возвращает *true*, на *new_value*. Возвращает позицию за последним скопированным элементом в полученном диапазоне. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная (*N* сравнений).

```
void replace_if(FwdIter first, FwdIter last,
    Predicate pred, const T& new_value)
```

Заменяет в диапазоне [*first, last*) все элементы, для которых *pred* возвращает *true*, на *new_value*.

Сложность линейная (*N* сравнений).

```
void reverse(BidiIter first, BidiIter last)
```

Переставляет элементы диапазона [*first, last*) в обратном порядке.

Сложность линейная (*N*/2 перестановок).

```
OutIter reverse_copy(BidiIter first, BidiIter last,
    OutIter result)
```

Копирует в обратном порядке элементы диапазона [*first, last*) в диапазон, начинающийся с *result*. Возвращает позицию за последним скопированным элементом в полученном диапазоне. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная (*N* присваиваний).

```
void rotate(FwdIter first, FwdIter middle,
    FwdIter last)
```

Циклически сдвигает элементы диапазона [*first, last*) так, что элементы из диапазона [*middle, last*) оказываются в начале новой последовательности (элементы из диапазона [*first, middle*) передвигаются в конец последовательности).

Сложность линейная (не более *N* перестановок)

```
OutIter rotate_copy(FwdIter first, FwdIter middle,
    FwdIter last, OutIter result)
```

Копирует в диапазон, начинающийся с *result*, вначале элементы из диапазона [*middle, last*), а затем — элементы из диапазона [*first, middle*). Возвращает позицию за последним скопированным элементом в полученном диапазоне. Исходный и результирующий диапазоны не должны перекрываться.

Сложность линейная (*N* присваиваний).

```
FwdIter1 search(FwdIter1 first1, FwdIter1 last1,
    FwdIter2 first2, FwdIter2 last2[,
    BinaryPredicate pred])
```

Находит первую (самую левую) подпоследовательность [*first2, last2*) в диапазоне [*first1, last1*). Возвращает итератор, который указывает на начало найденной подпоследовательности, или *last1*, если подпоследовательность не найдена. Для сравнения элементов используется предикат *pred*(**p1, *p2*) или (по умолчанию) операция ==.

Сложность линейная (не более *N1*N2* сравнений).

```
FwdIter search_n(FwdIter first, FwdIter last, Size
    count, const T& value[, BinaryPredicate pred])
```

Находит первую (самую левую) подпоследовательность из *count* соседних одинаковых значений *value* в диапазоне [*first, last*). Возвращает итератор, который указывает на начало найденной подпоследовательности, или *last*, если подпоследовательность не найдена. Для сравнения элементов используется предикат *pred*(**p1, *p2*) или (по умолчанию) операция == (фактически второй параметр в предикате всегда полагается равным *value*).

Сложность линейная (не более *N*count* сравнений).

```
OutIter set_difference(InIter1 first1,
    InIter1 last1, InIter2 first2, InIter2 last2,
    OutIter result[, Compare comp])
```

Копирует элементы предварительно отсортированного диапазона [*first1, last1*) в диапазон, начинающийся с *result*; при этом копируются только те элементы, которых нет в диапазоне [*first2, last2*) (этот диапазон также должен быть предварительно отсортирован). Возвращает позицию за последним скопированным элементом в полученном диапазоне. Таким образом, алгоритм реализует аналог теоретико-множественной операции разности, при котором в исходных наборах допускаются одинаковые, с точки зрения операции сравнения, элементы. Например, при обработке наборов (10, 10, 10, 20, 20, 40) и (10, 10, 30, 40) будет получен набор (10, 20, 20). Результирующий диапазон не должен перекрываться с любым из исходных. Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (не более 2*(*N1 + N2*) – 1 сравнений).

```
OutIter set_intersection(InIter1 first1,
    InIter1 last1, InIter2 first2, InIter2 last2,
    OutIter result[, Compare comp])
```

Копирует элементы предварительно отсортированного диапазона [*first1, last1*) в диапазон, начинающийся с *result*; при этом копируются только те элементы, которые присутствуют в диапазоне [*first2, last2*) (этот диапазон также должен быть предварительно отсортирован). Возвращает позицию за последним скопированным элементом в полученном диапазоне. Таким образом, алгоритм реализует аналог теоретико-множественной операции пересечения, при котором в исходных наборах допускаются одинаковые, с точки зрения операции сравнения, элементы. Например, при обработке наборов (10, 10, 10, 20, 20, 40) и (10, 10, 30, 40) будет получен набор (10, 10, 40). Результирующий диапазон не должен перекрываться с любым из исходных. Для сравнения элементов используется предикат comp(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (не более 2*(*N1 + N2*) – 1 сравнений).

```
OutIter set_symmetric_difference(InIter1 first1,
    InIter1 last1, InIter2 first2, InIter2 last2,
    OutIter result[, Compare comp])
```

Выполняет слияние предварительно отсортированных диапазонов [*first1, last1*) и [*first2, last2*) и копирует полученные отсортированные элементы в диапазон, начинающийся с *result*; при этом копируются только те элементы, которые присутствуют только в одном из исходных диапазонов. Возвращает позицию за последним скопированным элементом в полученном

диапазоне. Таким образом, алгоритм реализует аналог теоретико-множественной операции симметричной разности, при котором в исходных наборах допускаются одинаковые, с точки зрения операции сравнения, элементы. Например, при обработке наборов (10, 10, 10, 20, 20, 40) и (10, 10, 30, 40) будет получен набор (10, 20, 20, 30). Используемый алгоритм слияния обладает теми же свойствами, что и алгоритм *merge*; в частности, он является устойчивым. Результирующий диапазон не должен перекрываться с любым из исходных. Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (не более 2*(*N1* + *N2*) – 1 сравнений).

```
OutIter set_union(InIter1 first1, InIter1 last1,
    InIter2 first2, InIter2 last2, OutIter result[,
    Compare comp])
```

Выполняет слияние предварительно отсортированных диапазонов [*first1, last1*) и [*first2, last2*) и копирует полученные отсортированные элементы в диапазон, начинающийся с *result*; при этом равные элементы, присутствующие в каждом из исходных диапазонов, копируются только один раз (копируется элемент из первого диапазона). Возвращает позицию за последним скопированным элементом в полученном диапазоне. Таким образом, алгоритм реализует аналог теоретико-множественной операции объединения, при котором в исходных наборах допускаются одинаковые, с точки зрения операции сравнения, элементы. Например, при обработке наборов (10, 10, 10, 20, 20, 40) и (10, 10, 30, 40) будет получен набор (10, 10, 10, 20, 20, 30, 40). Используемый алгоритм слияния обладает теми же свойствами, что и алгоритм *merge*; в частности, он является устойчивым. Результирующий диапазон не должен перекрываться с любым из исходных. Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность линейная (не более 2*(*N1* + *N2*) – 1 сравнений).

```
void sort(RandIter first, RandIter last[,
    Compare comp])
```

Сортирует элементы в диапазоне [*first, last*). Сортировка не является устойчивой, т. е. равные элементы могут не сохранять свой исходный порядок. Для сравнения элементов используется предикат comp(**p1, *p2*) или (по умолчанию) операция <.

Сложность: в среднем *N**log *N* сравнений.

```
void sort_heap(RandIter first, RandIter last[,
    Compare comp])
```

Сортирует элементы в диапазоне [*first, last*) в предположении, что исходный диапазон образует кучу. Сортировка не является устойчивой, т. е. равные элементы могут не сохранять свой исходный порядок. Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность: не более *N**log *N* сравнений.

```
BidiIter stable_partition(BidiIter first,
    BidiIter last, Predicate pred)
```

Меняет местами элементы диапазона [*first, last*) так, чтобы все элементы, удовлетворяющие предикату *pred*, были расположены перед теми, которые ему не удовлетворяют. Относительный порядок следования элементов в каждой группе сохраняется. Алгоритм возвращает итератор, указывающий на первый элемент, для которого *pred* возвращает *false* (или итератор *last*, если таких элементов нет).

Сложность линейная (*N* вызовов *pred* и не более *N**log *N* перестановок даже в случае ограниченного объема памяти).

```
void stable_sort(RandIter first, RandIter last[,
    Compare comp])
```

Сортирует элементы в диапазоне [*first, last*). Сортировка является устойчивой, т. е. равные элементы сохраняют свой исходный порядок. Для сравнения элементов используется предикат *comp*(**p1, *p2*) или (по умолчанию) операция <.

Сложность: в среднем *N**log *N* сравнений, если достаточно памяти (при ограниченном объеме памяти — не более *N**log *N**log *N* сравнений).

```
void swap(T& a, T& b)
```

Меняет местами значения *a* и *b*.

```
FwdIter2 swap_ranges(FwdIter1 first1,
    FwdIter1 last1, FwdIter2 first2)
```

Меняет местами элементы диапазона [*first1, last1*) с элементами диапазона, начинающегося с *first2* и имеющего ту же длину, что и первый диапазон. Возвращает позицию за последним элементом второго диапазона. Диапазоны не должны перекрываться.

Сложность линейная (*N1* перестановок).

```
OutIter transform(InIter first, InIter last,
    OutIter result, UnaryOp unop)
OutIter transform(InIter1 first1, InIter1 last1,
    InIter2 first2, OutIter result, BinaryOp binop)
```

Присваивает новые значения всем элементам диапазона, начинающегося с *result*. В первой версии алгоритма используются значения *unop*(**p*), где *p* — итератор из диапазона [*first, last*); во второй версии алгоритма используются значения *binop*(**p1, *p2*), где *p1* — итератор из диапазона [*first1, last1*), а *p2* — итератор из диапазона, начинающегося с *first2* и имеющего ту же длину, что и первый диапазон. Возвращает позицию за последним элементом полученного диапазона. Результирующий диапазон может совпадать с любым из входных диапазонов.

Сложность линейная (*N* вызовов *unop* или *N1* вызовов *binop*).

```
FwdIter unique(FwdIter first, FwdIter last[,
    BinaryPredicate pred])
```

«Удаляет» из диапазона [*first, last*) повторные вхождения соседних равных элементов (удаляется вся последовательность соседних равных элементов, кроме ее начального элемента). «Удаляемые» элементы перемещаются в конец данного диапазона. Возвращает итератор *middle*, расположенный на первом «удаленном» элементе (таким образом, преобразованный набор без удаленных элементов размещается в диапазоне [*first, middle*)). Относительный порядок оставшихся в диапазоне [*first, middle*) элементов сохраняется. Если исходный диапазон [*first, last*) предварительно отсортирован, то в полученном диапазоне [*first, middle*) содержатся уникальные значения (также отсортированные). Для сравнения элементов используется предикат *pred*(**p1, *p2*) или (по умолчанию) операция ==.

Сложность линейная (max{0, *N* – 1} сравнений).

```
OutIter unique_copy(InIter first, InIter last,
    OutIter result[, BinaryPredicate pred])
```

Копирует элементы диапазона [*first, last*) в диапазон, начинающийся с *result*, удаляя повторные вхождения соседних равных элементов (удаляется вся последовательность соседних равных элементов, кроме ее начального элемента). Возвращает позицию за последним элементом полученного диапазона. Для сравнения элементов используется предикат *pred*(**p1, *p2*) или (по умолчанию) операция ==.

Сложность линейная (max{0, *N* – 1} сравнений).

```
FwdIter upper_bound(FwdIter first, FwdIter last,
    const T& value[, Compare comp])
```

Проверяет, содержится ли в диапазоне [*first, last*) значение *value*, и возвращает итератор, который указывает на элемент, расположенный после последнего элемента со значением *value* (если значение не найдено, то итератор указывает на позицию в диапазоне, в которую можно вставить *value*, не нарушая порядка сортировки). Содержимое диапазона должно быть предварительно отсортировано в соответствии с порядком, задаваемым предикатом *comp*(**p1, *p2*) или (по умолчанию) операцией <.

Сложность логарифмическая (не более log *N* + 1 сравнений).

Численные алгоритмы (заголовочный файл <numeric>)

Все функциональные объекты, используемые в алгоритмах данной группы, не должны иметь побочных эффектов.

```
T accumulate(InIter first, InIter last, T init[,
    BinaryOp binop])
```

Обрабатывает все значения из диапазона [*first, last*), добавляя их к переменной *tmp* типа *T*, инициализированной значением *init*, и возвращает полученный результат. При добавлении используется операция *binop* (в виде *tmp* = *binop*(*tmp*, **p*)) или (по умолчанию) операция +.

Сложность линейная (*N* вызовов *binop*).

```
OutIter adjacent_difference(InIter first,
    InIter last, OutIter result[, BinaryOp binop])
```

Обрабатывает пары соседних элементов из диапазона [*first, last*) с помощью операции *binop* (в виде *binop*(*(*p*+1), **p*)) или (по умолчанию) бинарной операции «–» (*(*p*+1) – **p*) и записывает полученные значения в диапазон вывода, начиная с *result* (первым записывается значение **first*, затем результат *binop*(*(*first*+1), **first*) и т. д.). Возвращает позицию за последним элементом полученного диапазона. Итератор *result* может совпадать с *first*.

Сложность линейная (*N* – 1 вызовов *binop*).

```
T inner_product(InIter1 first1, InIter1 last1,
    InIter2 first2, T init[, BinaryOp1 binop1,
    BinaryOp2 binop2])
```

Вычисляет сумму попарных произведений («внутреннее произведение») элементов двух диапазонов: [*first1, last1*) и диапазона такой же длины, начинающегося с *first2* (переменная суммирования *tmp* инициализируется значением *init*). Вместо бинарных операций «+» и «*» могут использоваться операции *binop1* и *binop2* соответственно: *tmp* = *binop1*(*tmp*, *binop2*(**p1*, **p2*)).

Сложность линейная (*N* вызовов *binop1* и *binop2*).

```
OutIter partial_sum(InIter first, InIter last,
    OutIter result[, BinaryOp binop])
```

Записывает частичные суммы (**first*, **first* + *(*first*+1), (**first* + *(*first*+1)) + *(*first*+2), …) в диапазон, начинающийся с *result*. Вместо операции по умолчанию «+» может использоваться бинарная операция *binop*. Возвращает позицию за последним элементом полученного диапазона. Итератор *result* может совпадать с *first*.

Сложность линейная (*N* – 1 вызовов *binop*).

Дополнение: итераторы вставки

С помощью обычных итераторов модифицирующие алгоритмы могут лишь изменять содержимое существующих элементов. *Итераторы вставки* позволяют вставлять новые элементы в контейнеры при выполнении модифицирующих алгоритмов. Имеются три вида итераторов вставки:

- *back_insert_iterator* (для вставки в конец контейнера)

Доступен для всех последовательных контейнеров. Может создаваться с помощью функции *back_inserter*(*c*), где *c* — контейнер. Каждое обращение к подобному итератору для записи в контейнер *c* элемента *e* приводит к вызову функции-члена *c.push_back*(*e*).

- *front_insert_iterator* (для вставки в начало контейнера)

Доступен для тех контейнеров, для которых реализована функция-член *push_front*(), т. е. для дека и списка. Может создаваться с помощью функции *front_inserter*(*c*), где *c* — контейнер. Каждое обращение к подобному итератору для записи в контейнер *c* элемента *e* приводит к вызову функции-члена *c.push_front*(*e*). Последовательность добавляемых элементов размещается в начале контейнера в *обратном* порядке.

- *insert_iterator* (для вставки в любую позицию контейнера)

Доступен для всех контейнеров. Может создаваться с помощью функции *inserter*(*c, pos*), где *c* — контейнер, а *pos* — итератор данного контейнера, определяющий место вставки. Каждое обращение к подобному итератору для записи в контейнер *c* элемента *e* приводит к выполнению операторов *pos* = *c.insert*(*pos, e*); *++pos*; таким образом, следующая вставка будет выполняться за вставленным ранее элементом. Вставка с помощью итератора *inserter*(*c, c.begin*()) отличается от вставки с помощью итератора *front_inserter*(*c*) тем, что в первом случае вставляемые элементы будут располагаться в контейнере в исходном порядке. Применение итераторов *inserter*(*c, c.end*()) и *back_inserter*(*c*) приводит к одинаковому результату.

С осторожностью следует использовать итераторы вставки при добавлении данных в тот же контейнер, из которого они извлекаются, поскольку операция вставки может приводить к тому, что некоторые (или даже все) итераторы, используемые для чтения, окажутся недействительными. Напомним, что любое действие по вставке элементов в дек делает недействительными все его итераторы. Безопасным контейнером в этом отношении является список, так как вставка в него новых данных оставляет корректными все связанные с ним итераторы. Промежуточное положение занимает вектор, который сохраняет корректными все итераторы до позиции вставки. Однако в случае перераспределения памяти (увеличения емкости) делаются недействительными все итераторы вектора.

Впрочем, проблемы могут возникнуть даже при работе со списками. Рассмотрим задачу дублирования всех элементов списка *l* путем их записи в его конец. Казалось бы, достаточно использовать следующий оператор:

```
copy(l.begin(), l.end(), back_inserter(l).
```

При использовании компилятора gcc этот оператор решает поставленную задачу, однако в Visual Studio C++ он приводит к зависанию программы.

Если требуется продублировать все элементы списка, записав их в конец в обратном порядке, то вариант

```
copy(l.rbegin(), l.rend(), back_inserter(l));
```

опять же решает задачу в gcc, а в Visual Studio создает список, в котором последний элемент исходного списка появляется не 2, а 3 раза. Например, 1, 2, 3 преобразуется в 1, 2, 3, 3, 3, 2, 1.

В то же время, при дублировании элементов списка в его начало проблем не возникает; в частности, первая из рассмотренных задач успешно решается в Visual Studio оператором

```
copy(l.rbegin(), l.rend(), front_inserter(l));
```

а для решения второй можно использовать два действия:

```
copy(l.begin(), l.end(), front_inserter(l));
l.reverse();
```

Рекомендации по использованию итераторов вставки:

- если алгоритм используется для преобразования и вставки данных из одного контейнера в другой, то такое действие всегда является безопасным;
- если требуется преобразовать и вставить данные из одной части контейнера в другую его часть, то следует принимать во внимание вид контейнера и расположение позиции вставки и исходных данных. Если позиция вставки и диапазон исходных данных «отграничены» друг от друга, то в случае списка действие всегда является безопасным, а в случае вектора оно будет безопасным, если, в дополнение к предыдущему условию, диапазон исходных данных расположен до позиции вставки и, кроме того, емкость вектора позволит завершить процедуру вставки без перераспределения памяти. В случае дека указанное действие никогда не является безопасным;
- если действие не является безопасным, то желательно выполнять его с использованием вспомогательного контейнера, инициализировав его требуемым диапазоном данных из исходного контейнера и затем использовав вспомогательный контейнер в алгоритме в качестве источника данных.