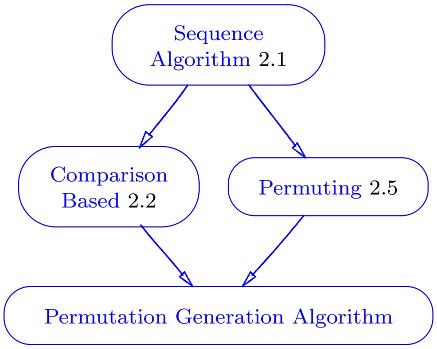
11.1 Permutation Generation

Section authors: Jeff Czarnowski, Andy Schechter, Tom Smith.



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

Prototype: template ⟨*class BidirectionalIterator*⟩

```
bool next_permutation(
       BidirectionalIterator first,
       BidirectionalIterator last);

template ⟨class BidirectionalIterator,       class StrictWeakOrdering⟩
bool next_permutation(
       BidirectionalIterator first,
       BidirectionalIterator last,
       StrictWeakOrdering comp);

template ⟨class BidirectionalIterator⟩
bool prev_permutation(
       BidirectionalIterator first,
       BidirectionalIterator last);

template ⟨class BidirectionalIterator,       class StrictWeakOrdering⟩
bool prev_permutation(
       BidirectionalIterator first,
       BidirectionalIterator last,
       StrictWeakOrdering comp);
```

Input: A sequence of comparable elements. See Sequence Algorithm (§2.1).

Output: Next_permutation transforms the range of elements into the next lexicographically greater permutation of the elements. If this permutation exists, the function returns true. If not, then it transforms the given range into the lexicographically smallest permutation and returns false. This is a mutable algorithm, so the container is altered to reflect the changes caused by the algorithm. Prev_permutation similarly generates the next lexicographically lesser permutation of the elements.

Effects: Standard effects of a Sequence Permuting Algorithm (§2.5).

This algorithm will change the container between the two given iterators to assign the values of the next greatest permutation. For example, if a range initially contained {1, 2, 3, 4}, this would transform it to {1, 2, 4, 3}. Called once again on the new data, it would yield {1, 3, 2, 4}. If the next greatest permutation does not exist, then the range would equal the lowest permutation. For example, {4, 3, 2, 1} would be changed to {1, 2, 3, 4}. This is the only case where the function would return false. Thus, this can be used for a very slow sorting routine that would call the next_permutation function repeatedly until it returns false. Of course, since there are $N!$ different permutations, the running time of this sort will be of the order $O(N!)$.

Asymptotic complexity: Let $N = \text{last} - \text{first}$.

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

Complexity in terms of operation counts:

- Average case:

next_permutation:	
total:	0.0564
iterator assign.	0.0198
iterator comp.:	0.0048
value assign.:	0.0060
value comp.:	0.0034
prev_permutation:	
total:	0.0500
iterator assign.	0.0192
iterator comp.:	0.0042
value assign.:	0.0060
value comp.:	0.0022
reverse:	
total:	$7.5N + 0.0055$
iterator assign.	$2.5N + 0.0042$
iterator comp.:	$0.5N + 0.0010$
value assign.:	$1.5N$
value comp.:	0

- Worst case:

permutation generation:	
total:	$13.5N + 0.0079$
iterator assign.	$3.5N + 0.0068$
iterator comp.:	$1.5N$
value assign.:	$1.5N$
value comp.:	$N - 0.0011$
prev_permutation:	
total:	$13.5N + 0.0079$
iterator assign.	$3.5N + 0.0068$
iterator comp.:	$1.5N$
value assign.:	$1.5N$
value comp.:	$N - 0.0011$
reverse:	
Same as average case.	

- Note: reverse is included for comparison because both next_permutation and prev_permutation rely on it. Results are scaled by 1000 and based upon the Operation Count tables below.

Table 1: Performance of Permutation Generation Algorithms on Random Sequences (Sizes and Operations Counts in Multiples of 1,000)

Size	Alg	Iterator	Iterator	Value	Value	Other	Total
		Comp	Assign	Comp	Assign	Ops	Ops
1	next_p	0.005	0.02	0.003	0.006	0.021	0.055
	prev_p	0.004	0.019	0.002	0.006	0.018	0.049
	reverse	0.501	2.504	0	1.5	3	7.505
4	next_p	0.005	0.02	0.004	0.006	0.024	0.059
	prev_p	0.004	0.019	0.002	0.006	0.018	0.049
	reverse	2.001	10.004	0	6	12	30.005
16	next_p	0.004	0.019	0.002	0.006	0.018	0.049
	prev_p	0.005	0.02	0.003	0.006	0.021	0.055
	reverse	8.001	40.004	0	24	48	120.005
64	next_p	0.005	0.02	0.004	0.006	0.024	0.059
	prev_p	0.004	0.019	0.002	0.006	0.018	0.049
	reverse	32.001	160.004	0	96	192	480.005
256	next_p	0.005	0.02	0.004	0.006	0.024	0.059
	prev_p	0.004	0.019	0.002	0.006	0.018	0.049
	reverse	128.001	640.004	0	384	768	1920.01
1024	next_p	0.004	0.019	0.002	0.006	0.018	0.049
	prev_p	0.005	0.02	0.003	0.006	0.021	0.055
	reverse	512.001	2560	0	1536	3072	7680.01

Note: A worst case sequence for next_permutation is a sequence sorted in descending order. A worst case sequence for prev_permutation is a sequence sorted in ascending order.

```
NEXT-PERMUTATION($first$, $last$)
if $first = last$ then
    return FALSE
end if
$iteratorA \leftarrow first$
$iteratorA \leftarrow iteratorA + 1$
if $iterator1 = last$ then
    return FALSE
```

Table 2: Performance of Permutation Generation Algorithms on Worst-Case Sequences (Sizes and Operations Counts in Multiples of 1,000)

Size	Alg	Iterator	Iterator	Value	Value	Other	Total
		Comp	Assign	Comp	Assign	Ops	Ops
1	next_p	1.502	3.507	0.999	1.5	5.999	13.507
	prev_p	1.502	3.507	0.999	1.5	5.999	13.507
4	next_p	6.002	14.007	3.999	6	23.999	54.007
	prev_p	6.002	14.007	3.999	6	23.999	54.007
16	next_p	24.002	56.007	15.999	24	95.999	216.01
	prev_p	24.002	56.007	15.999	24	95.999	216.01
64	next_p	96.002	224.01	63.999	96	384	864.01
	prev_p	96.002	224.01	63.999	96	384	864.01
256	next_p	384.00	896.01	256	384	1536	3456.01
	prev_p	384.00	896.01	256	384	1536	3456.01
1024	next_p	1536	3584.01	1024	1536	6144	13824
	prev_p	1536	3584.01	1024	1536	6144	13824

```
end if
$iteratorA \leftarrow last$
$iteratorA \leftarrow iteratorA - 1$
while TRUE do
    $iteratorB \leftarrow iteratorA$
    $iteratorA \leftarrow iteratorA - 1$
    if $value[iteratorA] < value[iteratorB]$ then
        $iteratorC \leftarrow last$
        while $value[iteratorA] > value[iteratorC]$ do
            $iteratorC \leftarrow iteratorC - 1$
        end while
        ITER-SWAP($iteratorA, iteratorC$)
        REVERSE($iteratorB, last$)
        return TRUE
    end if
    if $iteratorA = first$ then
        REVERSE($first$, $last$)
        return FALSE
    end if
end while
```

```
PREV-PERMUTATION($first$, $last$)
if $first = last$ then
    return FALSE
end if
$iteratorA \leftarrow first$
$iteatorA \leftarrow iteratorA + 1$
if $iterator1 = last$ then
    return FALSE
end if
$iteratorA \leftarrow last$
$iteratorA \leftarrow iteratorA - 1$
while TRUE do
    $iteratorB \leftarrow iteratorA$
    $iteratorA \leftarrow iteratorA - 1$
    if $value[iteratorB] < value[iteratorB]$ then
        $iteratorC \leftarrow last$
        while $value[iteratorC] > value[iteratorA]$ do
            $iteratorC \leftarrow iteratorC - 1$
        end while
        ITER-SWAP($iteratorA, iteratorC$)
        REVERSE($iteratorB, last$)
        return TRUE
    end if
    if $iteratorA = first$ then
        REVERSE($first$, $last$)
        return FALSE
    end if
end while
```

```
REVERSE($first$, $last$)
while TRUE do
    $temp \leftarrow last - 1$
    if $first = last$ or $first = temp$ then
        $last \leftarrow last - 1$ RETURN
    else
```

```
        $first \leftarrow first + 1$
        ITER-SWAP($first$, $last$)
    end if
end while
```

```
ITER-SWAP($iterator1$, $iterator2$)
$temp \leftarrow value[iterator1]$
$value[iterator1] \leftarrow value[iterator2]$
$value[iterator2] \leftarrow temp$
```