



# STL

## -CHAPTER6-

SOULSEEK



# 목차

- 
- 1.** 정렬 알고리즘
  - 2.** 정렬된 범위 알고리즘
  - 3.** 수치 알고리즘

# 정렬 알고리즘

# 1. 정렬 알고리즘

- 변경 알고리즘의 특수한 형태
- 특정 정렬 기준으로 원소의 순서를 변경하며 정렬한다.

알고리즘	설명(설명에 사용되는 $p$ 는 구간 $[b,e)$ 의 반복자)
<b>p=partition(b,e,f)</b>	구간 $[b,e)$ 의 순차열 중 $f(*p)$ 가 참인 원소는 $[b,e)$ 의 순차열에 거짓인 원소는 $[p,e)$ 의 순차열로 분류한다.
<b>stable_partition(b,e,f)</b>	<b>partition()</b> 알고리즘과 같고 원소의 상대적인 순서를 유지한다.
<b>make_heap(b,e)</b>	힙을 생성한다. 구간 $[b,e)$ 의 순차열을 힙 구조로 변경한다.
<b>make_heap(b,e,f)</b>	힙을 생성한다. 구간 $[b,e)$ 의 순차열을 힙 구조로 변경한다. $f$ 는 조건자로 비교에 사용한다.
<b>push_heap(b,e)</b>	힙에 원소를 추가한다. 보통 <b>push_back()</b> 멤버 함수와 같이 사용되며, 구간 $[b,e)$ 의 순차열을 다시 힙 구조가 되게 한다.
<b>push_heap(b,e,f)</b>	힙에 원소를 추가한다. 보통 <b>push_back()</b> 멤버 함수와 같이 사용되며, 구간 $[b,e)$ 의 순차열을 다시 힙 구조가 되게 한다. $f$ 는 조건자로 비교에 사용한다.
<b>pop_heap(b,e)</b>	힙에서 원소를 제거한다. 구간 $[b,e)$ 의 순차열의 가장 큰 원소(첫 원소)를 제거한다.
<b>pop_heap(b,e,f)</b>	힙에서 원소를 제거한다. 구간 $[b,e)$ 의 순차열의 가장 큰 원소(첫 원소)를 제거한다. $f$ 를 이용한 비교
<b>sort_heap(b,e)</b>	힙을 정렬한다. 구간 $[b,e)$ 의 순차열을 힙 구조를 이용해 정렬한다.
<b>sort_heap(b,e,f)</b>	힙을 정렬한다. 구간 $[b,e)$ 의 순차열을 힙 구조를 이용해 정렬한다. $f$ 는 조건자로 비교에 사용한다.
<b>nth_element(b,m,e)</b>	구간 $[b,e)$ 의 원소 중 $m-b$ 개 만큼 선별된 원소를 $[b,m)$ 순차열에 놓이게 한다.
<b>nth_element(b,m,e,f)</b>	구간 $[b,e)$ 의 원소 중 $m-b$ 개 만큼 선별된 원소를 $[b,m)$ 순차열에 놓이게 한다.
<b>sort(b,e)</b>	퀵 정렬을 기반으로 정렬한다. 구간 $[b,e)$ 를 정렬한다.

# 1. 정렬 알고리즘

알고리즘	설명(설명에 사용되는 $p$ 는 구간 $[b, e]$ 의 반복자)
<b>srot(b,e,f)</b>	퀵 정렬을 기반으로 정렬한다. 구간 $[b, e]$ 를 조건자 $f$ 를 사용해 정렬한다.
<b>stable_sort(b,e)</b>	머지 정렬을 기반으로 정렬한다. 구간 $[b, e]$ 를 정렬하되 값이 같은 원소의 상대적인 순서를 유지한다.
<b>stable_sort(b,e,f)</b>	머지 정렬을 기반으로 정렬한다. 구간 $[b, e]$ 를 정렬하되 값이 같은 원소의 상대적인 순서를 유지한다. $F$ 는 조건자로 비교에 사용한다.
<b>partial_sort(b,m,e)</b>	힙 정렬을 기반으로 정렬한다. 구간 $[b, e]$ 의 원소 중 $m-b$ 개 만큼의 상위 원소를 정렬하여 $[b, m]$ 순차열에 놓는다.
<b>partial_sort(b,m,e,f)</b>	힙 정렬을 기반으로 정렬한다. 구간 $[b, e]$ 의 원소 중 $m-b$ 개 만큼의 상위 원소를 정렬하여 $[b, m]$ 순차열에 놓는다. $f$ 는 조건자로 비교에 사용한다.
<b>partial_sort_copy(b,e,b2,e2)</b>	힙 정렬을 기반으로 정렬한다. 구간 $[b, e]$ 의 원소 중 상위 $e2-b2$ 개의 원소 정도만 정렬하여 $[b2, e2]$ 로 복사한다.
<b>partial_sort_copy(b,e,b2,e2,f)</b>	힙 정렬을 기반으로 정렬한다. 구간 $[b, e]$ 의 원소 중 상위 $e2-b2$ 개의 원소 정도만 정렬하여 $[b2, e2]$ 로 복사한다. $f$ 는 조건자로 비교 사용한다.

# 1. 정렬 알고리즘

**make\_heap()** 알고리즘 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v;  
    v.push_back(10);  
    v.push_back(20);  
    v.push_back(30);  
    v.push_back(40);  
    v.push_back(50);  
    v.push_back(60);
```

```
    cout << "v : ";  
    for (vector<int>::size_type i = 0; i < v.size(); ++i)  
        cout << v[i] << " ";  
    cout << endl;
```

```
    make_heap(v.begin(), v.end());  
    cout << "v[힙 생성] : ";  
    for (vector<int>::size_type i = 0; i < v.size(); ++i)  
        cout << v[i] << " ";  
    cout << endl;
```

```
}
```

조건1. 부모보다 자식이 우선순위가 낮다.  
조건2. 내림차순 정렬이 디폴트다

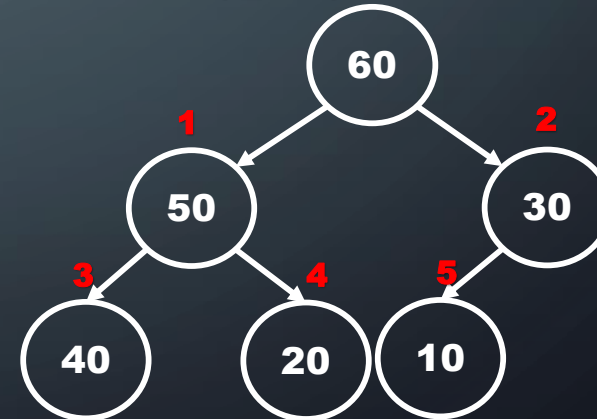
힙 구조 생성 전



힙 구조 생성 후



순차열 순서: 0



# 1. 정렬 알고리즘

## push\_heap() 알고리즘 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v;  
    v.push_back(10);  
    v.push_back(20);  
    v.push_back(30);  
    v.push_back(40);  
    v.push_back(50);  
    v.push_back(60);
```

push\_back같은 삽입 멤버 함수와 함께 사용되어  
push\_back으로 추가된 멤버를 heap 정렬한다.

```
    make_heap(v.begin(), v.end());
```

```
    cout << "v[힙 생성] : ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
    v.push_back(35);
```

```
    cout << "v 순차열에 35 추가 : ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
    push_heap(v.begin(), v.end());
```

```
    cout << "v[힙 추가] 연산 수행 :";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

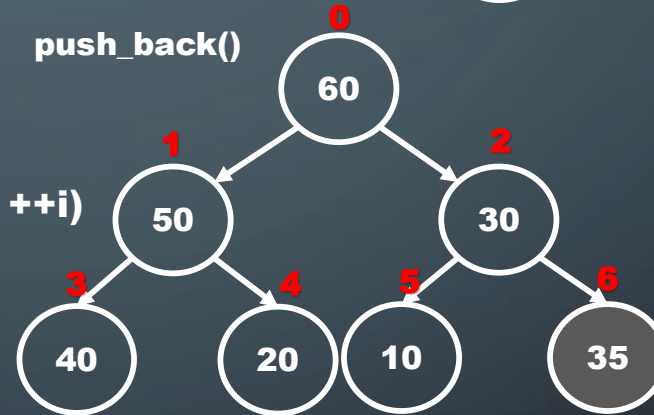
```
        cout << v[i] << " ";
```

```
    cout << endl;
```

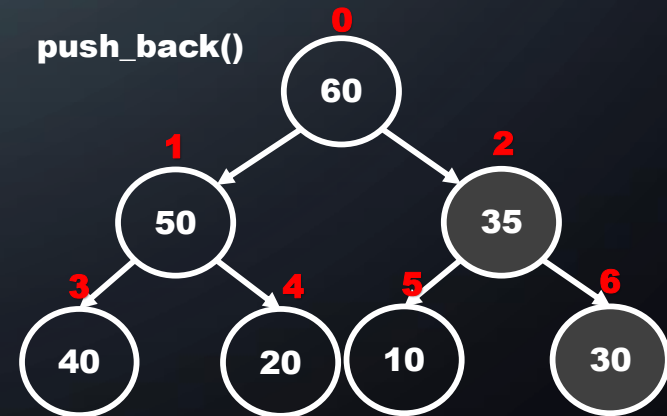
heap 생성 후



push\_back()



push\_back()



# 1. 정렬 알고리즘

## pop\_heap() 알고리즘 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v;  
    v.push_back(10);  
    v.push_back(20);  
    v.push_back(30);  
    v.push_back(40);  
    v.push_back(50);  
    v.push_back(60);
```

```
    make_heap(v.begin(), v.end());
```

```
    cout << "v[힙 생성] : ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
    pop_heap(v.begin(), v.end());
```

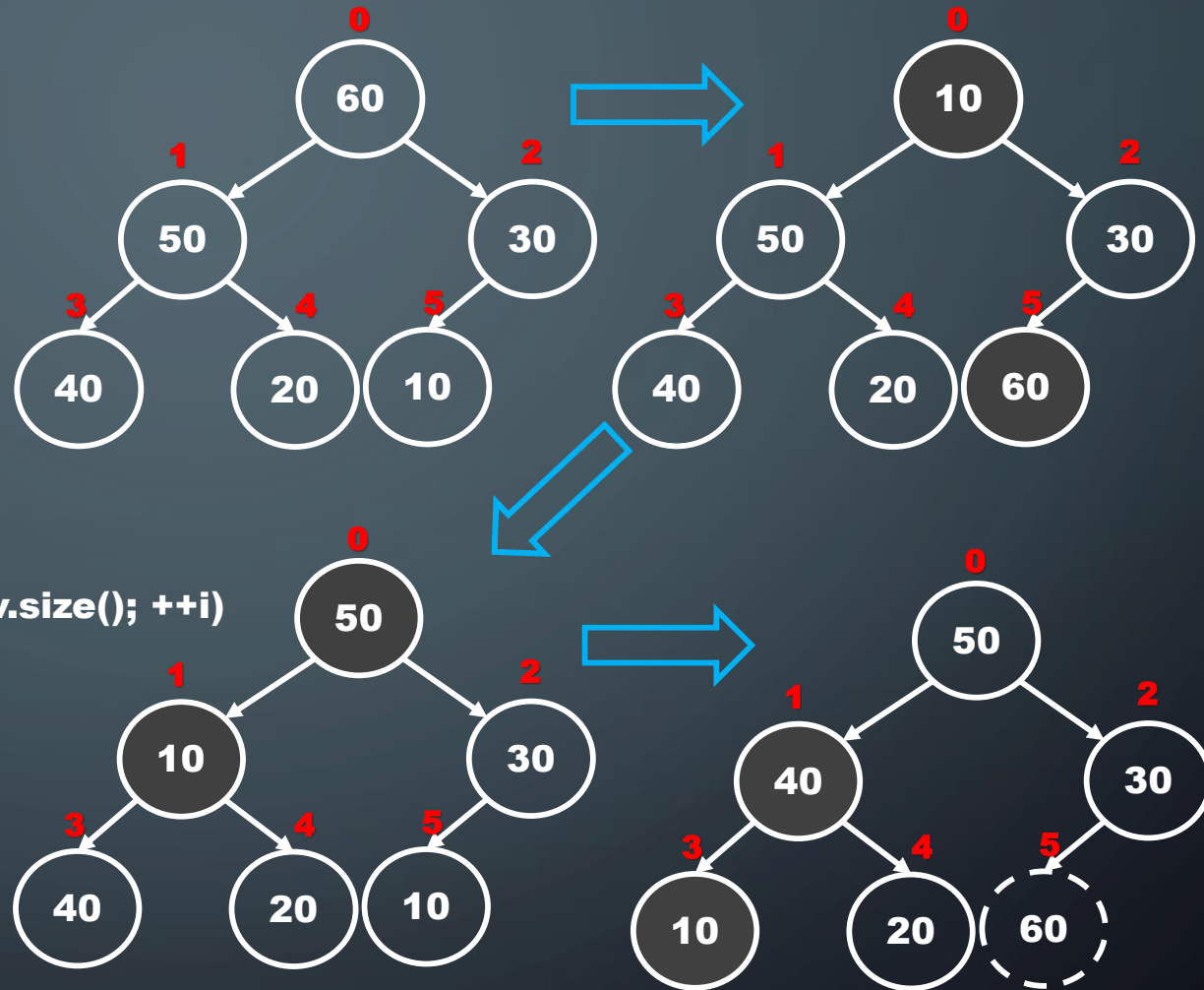
```
    cout << "v[힙 삭제] 연산 수행 :";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
}
```



**pop**으로 제거할 루트 노드를 **heap**에서 배제하는 위치로 변경만 한다. 제거 알고리즘도 아니며 정렬만 수행한다  
원소삭제 멤버 함수를 활용해서 제거 할 수 있다.



# 1. 정렬 알고리즘

## sort\_heap() 알고리즘 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v;
```

```
    v.push_back(10);
```

```
    v.push_back(20);
```

```
    v.push_back(30);
```

```
    v.push_back(40);
```

```
    v.push_back(50);
```

```
    v.push_back(60);
```

```
    make_heap(v.begin(), v.end());
```

```
    cout << "v[힙 생성] : ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
    sort_heap(v.begin(), v.end());
```

```
    cout << "v[힙 정렬] : ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
}
```

sort\_heap()은 오름차순이 디폴트다.

# 1. 정렬 알고리즘

## 조건자를 적용한 힙 생성 예제..

```
void main()
{
    vector<int> v;
    v.push_back(40);
    v.push_back(10);
    v.push_back(50);
    v.push_back(30);
    v.push_back(20);
    v.push_back(60);

    //부모 노드가 모든 자식 노드보다 작은 힙을 생성한다.
    make_heap(v.begin(), v.end(), greater<int>());

    cout << "v[힙 생성] : ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    v.push_back(35);
    push_heap(v.begin(), v.end(), greater<int>());

    cout << "v[힙 추가] : ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
```

```
sort_heap(v.begin(), v.end(), greater<int>());
```

```
cout << "v[힙 정렬] : ";
for (vector<int>::size_type i = 0; i < v.size(); ++i)
    cout << v[i] << " ";
cout << endl;
```

**heap**의 생성할 때와 같은 조건자를  
가지고 원소를 추가 해야 한다.

# 1. 정렬 알고리즘

## **nth\_element()** 알고리즘 사용 예제..

```
void main()
{
    vector<int> v;

    for (int i = 0; i < 100; ++i)
        v.push_back(rand() % 1000);

    nth_element(v.begin(), v.begin() + 20, v.end());

    cout << "v[상위 20개] : ";

    for (vector<int>::size_type i = 0; i < 20; ++i)
        cout << v[i] << " ";

    cout << endl;

    cout << "v[하위 80개] : ";

    for (vector<int>::size_type i = 20; i < v.size(); ++i)
        cout << v[i] << " ";

    cout << endl;
}
```

**v.begin() ~ v.begin() + 20**의 구간. 즉, 상위 **20**개를  
디폴트인 오름 차순으로 정렬

# 1. 정렬 알고리즘

## **sort()** 알고리즘의 랜덤 정수 정렬 예제..

```
void main()
{
    vector<int> v;

    for (int i = 0; i < 100; ++i)
        v.push_back(rand() % 1000);

    cout << "v[정렬 전] : ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    sort(v.begin(), v.end());
    cout << "v[정렬 less] : ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    sort(v.begin(), v.end(), Greater);
    //sort(v.begin(), v.end(), greater<int>() );

    cout << "v[정렬 greater] : ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
}

bool Greater(int left, int right)
{
    return left > right;
}
```

# 1. 정렬 알고리즘

## stable\_sort() 알고리즘 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v;
```

```
    v.push_back(30);
```

```
    v.push_back(50);
```

```
    v.push_back(30);
```

```
    v.push_back(20);
```

```
    v.push_back(40);
```

```
    v.push_back(10);
```

```
    v.push_back(40);
```

```
    cout << "v[정렬 전] : ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
    stable_sort(v.begin(), v.end());
```

```
    cout << "v[정렬 less] : ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
    stable_sort(v.begin(), v.end(), Greater);
```

```
    //sort(v.begin(), v.end(), greater<int>() );
```

```
    cout << "v[정렬 greater] : ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
}
```

```
bool Greater(int left, int right)
```

```
{
```

```
    return left > right;
```

```
}
```

정렬 전

30A	50	30B	20	40A	10	40B	N
-----	----	-----	----	-----	----	-----	---

```
    stable_sort(v.begin(), v.end())
```

10	20	30A	30B	40A	40B	50	N
----	----	-----	-----	-----	-----	----	---

```
    stable_sort(v.begin(), v.end(), greater<int>())
```

50	40A	40B	30A	30B	20	10	N
----	-----	-----	-----	-----	----	----	---

# 1. 정렬 알고리즘

## **partial\_sort()** 알고리즘 사용 예제..

```
void main()
{
    vector<int> v;

    for (int i = 0; i < 100; ++i)
        v.push_back(rand() % 1000);

    partial_sort(v.begin(), v.begin() + 20, v.end());

    cout << "v[상위 정렬 20개] : ";
    for (vector<int>::size_type i = 0; i < 20; ++i)
        cout << v[i] << " ";

    cout << endl;

    cout << "v[하위 80개] : ";
    for (vector<int>::size_type i = 20; i < v.size(); ++i)
        cout << v[i] << " ";

    cout << endl;
}
```

# 1. 정렬 알고리즘

## **partial\_sort\_copy()** 알고리즘 사용 예제..

```
void main()
{
    vector<int> v1;

    for (int i = 0; i < 100; ++i)
        v1.push_back(rand() % 1000);

    cout << "[v1 정렬 전] : ";
    for (vector<int>::size_type i = 0; i < v1.size(); ++i)
        cout << v1[i] << " ";
    cout << endl;

    vector<int> v2(20); //size : 20의 vector 생성

    partial_sort_copy(v1.begin(), v1.end(), v2.begin(), v2.end());
    cout << "[v2 less]: ";
    for (vector<int>::size_type i = 0; i < v2.size(); ++i)
        cout << v2[i] << " ";
    cout << endl;

    partial_sort_copy(v1.begin(), v1.end(), v2.begin(), v2.end(), greater<int>());
    cout << "[v2 greater] : ";
    for (vector<int>::size_type i = 0; i < v2.size(); ++i)
        cout << v2[i] << " ";
    cout << endl;
}
```



# 정렬된 범위 알고리즘



## 2. 정렬된 범위 알고리즘

- 정렬된 범위 안에서만 동작하는 알고리즘을 말한다.
- 입력 순차열이 반드시 정렬돼 있어야 한다.
- 원소가 같음을 표시할 때 **a '==' b** 연산자가 아닌 **!(a<b) && !(b<a)**를 사용한다.

알고리즘	설명(설명에 사용되는 p는 구간[b,e)의 반복자)
<b>binary_search(b,e,x)</b>	구간 <b>[b,e)</b> 의 순차열에 <b>x</b> 와 같은 원소가 있는가?
<b>binary_search(b,e,x,f)</b>	구간 <b>[b,e)</b> 의 순차열에 <b>x</b> 와 같은 원소가 있는가? <b>f</b> 를 비교에 사용한다.
<b>includes(b,e,b2,e2)</b>	구간 <b>[b2,e2)</b> 의 모든 원소가 구간 <b>[b,e)</b> 에도 있는가?
<b>includes(b,e,b2,e2,f)</b>	구간 <b>[b2,e2)</b> 의 모든 원소가 구간 <b>[b,e)</b> 에도 있는가? <b>f</b> 를 비교에 사용한다.
<b>p=lower_bound(b,e,x)</b>	<b>p</b> 는 구간 <b>[b,e)</b> 의 순차열에서 <b>x</b> 와 같은 첫 원소의 반복자다.
<b>p=lower_bound(b,e,x,f)</b>	<b>p</b> 는 구간 <b>[b,e)</b> 의 순차열에서 <b>x</b> 와 같은 첫 원소의 반복자다. <b>f</b> 를 비교에 사용한다.
<b>p=upper_bound(b,e,x)</b>	<b>p</b> 는 구간 <b>[b,e)</b> 의 순차열에서 <b>x</b> 보다 큰 첫 원소의 반복자다.
<b>p=upper_bound(b,e,x,f)</b>	<b>p</b> 는 구간 <b>[b,e)</b> 의 순차열에서 <b>x</b> 보다 큰 첫 원소의 반복자다. <b>f</b> 를 비교에 사용한다.
<b>pair(p1,p2)=equal_range(b,e,x)</b>	구간 <b>[p1,p2)</b> 의 순차열 구간 <b>[b,e)</b> 의 순차열에서 <b>x</b> 와 같은 원소의 구간(순차열)이다. <b>[lower_bound(), upper_bound())</b> 의 순차열과 같다.
<b>pair(p1,p2)=equal_range(b,e,x,f)</b>	구간 <b>[p1,p2)</b> 의 순차열 구간 <b>[b,e)</b> 의 순차열에서 <b>x</b> 와 같은 원소의 구간(순차열)이다. <b>[lower_bound(), upper_bound())</b> 의 순차열과 같다. <b>f</b> 를 비교에 사용한다.
<b>p=merge(b,e,b2,e2,t)</b>	구간 <b>[b,e)</b> 의 순차열과 구간 <b>[b2,e2)</b> 의 순차열을 합병해 <b>[t,p)</b> 에 저장한다.
<b>p=merge(b,e,b2,e2,t,f)</b>	구간 <b>[b,e)</b> 의 순차열과 구간 <b>[b2,e2)</b> 의 순차열을 합병해 <b>[t,p)</b> 에 저장한다. <b>f</b> 를 비교에 사용한다.

## 2. 정렬된 범위 알고리즘

알고리즘	설명(설명에 사용되는 p는 구간[b,e)의 반복자)
<b>inplace_merge(b,m,e)</b>	정렬된 <b>[b,m)</b> 순차열과 <b>[m,e)</b> 순차열을 <b>[b,e)</b> 순차열로 합병한다.
<b>inplace_merge(b,m,e,f)</b>	정렬된 <b>[b,m)</b> 순차열과 <b>[m,e)</b> 순차열을 <b>[b,e)</b> 순차열로 합병한다. <b>f</b> 를 비교에 사용한다.
<b>p=set_union(b,e,b2,e2,t)</b>	구간 <b>[b,e)</b> 의 순차열과 <b>[b2,e2)</b> 의 순차열을 정렬된 합집합으로 <b>[t,p)</b> 에 저장한다.
<b>p=set_union(b,e,b2,e2,t,f)</b>	구간 <b>[b,e)</b> 의 순차열과 <b>[b2,e2)</b> 의 순차열을 정렬된 합집합으로 <b>[t,p)</b> 에 저장한다. <b>f</b> 를 비교에 사용
<b>p=set_intersection(b,e,b2,e2,t)</b>	구간 <b>[b,e)</b> 의 순차열과 <b>[b2,e2)</b> 의 순차열을 정렬된 교집합으로 <b>[t,p)</b> 에 저장한다.
<b>p=set_intersection(b,e,b2,e2,t,f)</b>	구간 <b>[b,e)</b> 의 순차열과 <b>[b2,e2)</b> 의 순차열을 정렬된 합집합으로 <b>[t,p)</b> 에 저장한다. <b>f</b> 를 비교에 사용
<b>p=set_difference(b,e,b2,e2,t)</b>	구간 <b>[b,e)</b> 의 순차열과 <b>[b2,e2)</b> 의 순차열을 정렬된 차집합으로 <b>[t,p)</b> 에 저장한다.
<b>p=set_difference(b,e,b2,e2,t,f)</b>	구간 <b>[b,e)</b> 의 순차열과 <b>[b2,e2)</b> 의 순차열을 정렬된 차집합으로 <b>[t,p)</b> 에 저장한다. <b>f</b> 를 비교에 사용
<b>p=set_symmetric_difference(b,e,b2,e2,t)</b>	구간 <b>[b,e)</b> 의 순차열과 <b>[b2,e2)</b> 의 순차열을 정렬된 대칭 차집합으로 <b>[t,p)</b> 에 저장한다.
<b>p=set_symmetric_difference(b,e,b2,e2,t,f)</b>	구간 <b>[b,e)</b> 의 순차열과 <b>[b2,e2)</b> 의 순차열을 정렬된 대칭 차집합으로 <b>[t,p)</b> 에 저장한다. <b>f</b> 를 비교에 사용

## 2. 정렬된 범위 알고리즘

### **binary\_search()** 알고리즘 사용 예제..

```
void main()
{
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    if (binary_search(v.begin(), v.end(), 20))
        cout << "20 원소가 존재합니다." << endl;
    else
        cout << "20 원소가 존재하지 않습니다." << endl;
}
```

## 2. 정렬된 범위 알고리즘

조건자를 통한 **binary\_search()** 알고리즘 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v;
```

```
    v.push_back(40);
```

```
    v.push_back(46);
```

```
    v.push_back(12);
```

```
    v.push_back(80);
```

```
    v.push_back(10);
```

```
    v.push_back(47);
```

```
    v.push_back(30);
```

```
    v.push_back(55);
```

```
    v.push_back(90);
```

```
    v.push_back(53);
```

```
    cout << "[v 원본]: ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
    sort(v.begin(), v.end(), Pred);
```

```
    cout << "[v: 정렬]: ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
    if (binary_search(v.begin(), v.end(), 32, Pred))
```

```
        cout << "32 원소가 존재합니다." << endl;
```

```
    else
```

```
        cout << "32 원소가 존재하지 않습니다." << endl;
```

```
    if (binary_search(v.begin(), v.end(), 35, Pred))
```

```
        cout << "35 원소가 존재합니다." << endl;
```

```
    else
```

```
        cout << "35 원소가 존재하지 않습니다." << endl;
```

```
}
```

```
bool Pred(int left, int right)
```

```
{
```

```
    return 3 < right - left;
```

```
}
```

오름차순 정렬을 하되 주어진 조건이 만족할 경우  
매개변수로 사용한 원소끼리 교환이 이루어지는 것.

원본 정렬

40	46	12	80	10	47	30	55	90	53	N
----	----	----	----	----	----	----	----	----	----	---

sort() - 두 원소의 차이가 3보다 크면 다음 원소로 정렬

12	10	30	40	46	47	30	55	90	53	N
----	----	----	----	----	----	----	----	----	----	---

binary\_search() - 두 원소의 차이가 3이하이면 같은 원소 - !Pred(30,32) && !Pred(32,30)

12	10	30	40	46	47	30	55	90	53	N
----	----	----	----	----	----	----	----	----	----	---

## 2. 정렬된 범위 알고리즘

### binary\_search() 알고리즘의 조건자 사용 예제..

```
void main()
{
    vector<int> v;

    v.push_back(10);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);
    v.push_back(20);

    cout << "[v 원본]: ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    //기본 정렬 기준 less 사용
    sort(v.begin(), v.end());

    cout << "[v: less 정렬]: ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    //비교 조건자 less 지정(일반 버전 binary_search() 가능)
    cout << "비교 less 찾기: " << binary_search(v.begin(), v.end(), 20, less<int>()) << endl;

    //정렬 기준 greater 지정
    sort(v.begin(), v.end(), greater<int>());
    cout << "[v: greater 정렬]: ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    //조건자 정렬 기준 greater 지정
    cout << "비교 greater 찾기: ";
    cout << binary_search(v.begin(), v.end(), 20, greater<int>());
    cout << endl;
}
```

## 2. 정렬된 범위 알고리즘

### Includes() 알고리즘 사용 예제..

```
void main()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
    v1.push_back(50);

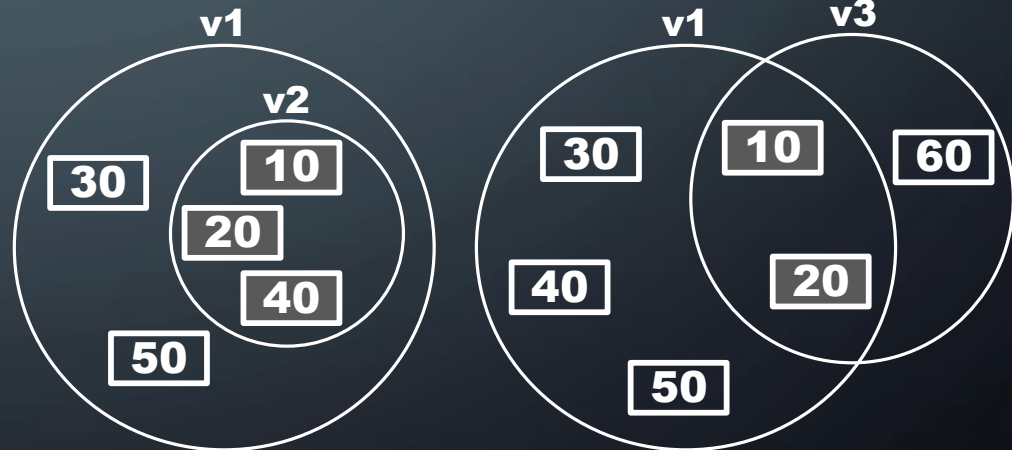
    vector<int> v2;
    v2.push_back(10);
    v2.push_back(20);
    v2.push_back(40);

    vector<int> v3;
    v3.push_back(10);
    v3.push_back(20);
    v3.push_back(60);

    if (includes(v1.begin(), v1.end(), v2.begin(), v2.end()))
        cout << "v2는 v1의 부분 집합" << endl;
    else
        cout << "v2는 v1의 부분 집합 아님" << endl;

    //정렬 기준을 greater<int> 설정
    sort(v1.begin(), v1.end(), greater<int>());
    sort(v2.begin(), v2.end(), greater<int>());

    //비교 기준을 greater<int> 설정
    if (includes(v1.begin(), v1.end(), v2.begin(), v2.end(), greater<int>()))
        cout << "greater정렬 순차열: v2는 v1의 부분 집합" << endl;
}
```





## 2. 정렬된 범위 알고리즘

**lower\_bound(), upper\_bound()** 알고리즘 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v;  
    v.push_back(10);  
    v.push_back(20);  
    v.push_back(30);  
    v.push_back(30);  
    v.push_back(30);  
    v.push_back(40);  
    v.push_back(50);
```

```
    vector<int>::iterator iter_lower, iter_upper;
```

```
    iter_lower = lower_bound(v.begin(), v.end(), 30);
```

```
    iter_upper = upper_bound(v.begin(), v.end(), 30);
```

```
    cout << "30 원소의 순차열 [iter_lower, iter_upper): ";
```

```
    for (vector<int>::iterator iter = iter_lower; iter != iter_upper; ++iter)
```

```
        cout << *iter << " ";
```

```
    cout << endl;
```

```
}
```



## 2. 정렬된 범위 알고리즘

**equal\_range()** 알고리즘 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v;  
    v.push_back(10);  
    v.push_back(20);  
    v.push_back(30);  
    v.push_back(30);  
    v.push_back(30);  
    v.push_back(40);  
    v.push_back(50);
```

```
    //vector<int>::iterator iter_lower, iter_upper;  
    pair<vector<int>::iterator, vector<int>::iterator> iter_pair;
```

```
    iter_pair = equal_range(v.begin(), v.end(), 30);
```

```
    cout << "30 원소의 순차열 [iter_pair.first, iter_pair.second): ";
```

```
    for (vector<int>::iterator iter = iter_pair.first; iter != iter_pair.second; ++iter)  
        cout << *iter << " ";
```

```
    cout << endl;
```

```
}
```





## 2. 정렬된 범위 알고리즘

### merge() 알고리즘 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v1;  
    v1.push_back(10);  
    v1.push_back(20);  
    v1.push_back(30);  
    v1.push_back(40);  
    v1.push_back(50);
```

```
    vector<int> v2;  
    v2.push_back(20);  
    v2.push_back(30);  
    v2.push_back(60);
```

```
    vector<int> v3(10); //size: 10인 vector 생성
```

```
    cout << "v1: ";  
    for (vector<int>::size_type i = 0; i < v1.size(); ++i)  
        cout << v1[i] << " ";  
    cout << endl;  
    cout << "v2: ";  
    for (vector<int>::size_type i = 0; i < v2.size(); ++i)  
        cout << v2[i] << " ";  
    cout << endl;  
    cout << "v3: ";
```

```
    for (vector<int>::size_type i = 0; i < v3.size(); ++i)  
        cout << v3[i] << " ";  
    cout << endl;
```

```
    vector<int>::iterator iter_end;  
    //v1의 순차열과 v2의 순차열을 합병하여 [v3.begin(),  
    iter_end)의 순차열에 저장한다.  
    iter_end = merge(v1.begin(), v1.end(), v2.begin(),  
    v2.end(), v3.begin());  
    cout << "v1: ";  
    for (vector<int>::size_type i = 0; i < v1.size(); ++i)  
        cout << v1[i] << " ";  
    cout << endl;  
    cout << "v2: ";  
    for (vector<int>::size_type i = 0; i < v2.size(); ++i)  
        cout << v2[i] << " ";  
    cout << endl;  
    cout << "v3: ";  
    for (vector<int>::size_type i = 0; i < v3.size(); ++i)  
        cout << v3[i] << " ";  
    cout << endl;
```

## 2. 정렬된 범위 알고리즘

### **inplace\_merge()** 알고리즘 사용 예제..

```
void main()
{
```

```
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);
```

```
    v.push_back(20);
    v.push_back(30);
    v.push_back(60);
```

```
    cout << "v의 두 구간으로 정렬된 하나의 순차열" << endl;
    cout << "[v.begin(), v.begin()+5) + [v.begin()+5, v.end())" << endl;
    cout << "v: ";
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
// 두 구간으로 정렬된 하나의 순차열을 한 구간으로 정렬한다.
inplace_merge(v.begin(), v.begin() + 5, v.end());
```

```
    cout << "v: ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
}
```

v.begin()



v.begin()+5



v.end()



10	20	30	40	50	20	30	60	N
----	----	----	----	----	----	----	----	---

**inplace\_merge** 실행 후

10	20	30	40	50	20	30	60	N
----	----	----	----	----	----	----	----	---

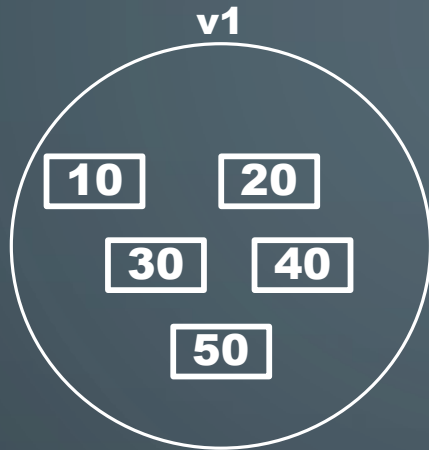
## 2. 정렬된 범위 알고리즘

### set\_union() 알고리즘 사용 예제..

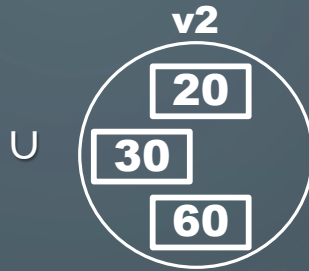
```
void main()
```

```
{
```

```
vector<int> v1;  
v1.push_back(10);  
v1.push_back(20);  
v1.push_back(30);  
v1.push_back(40);  
v1.push_back(50);
```



```
vector<int> v2;  
v2.push_back(20);  
v2.push_back(30);  
v2.push_back(60);
```



```
vector<int> v3(10); //size: 10인 vector 생성  
vector<int>::iterator iter_end;
```

v3.begin()



iter\_end



```
iter_end = set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());
```

```
cout << "[v3.begin(), iter_end): ";
```

```
for (vector<int>::iterator iter = v3.begin(); iter != iter_end; ++iter)
```

```
    cout << *iter << " ";
```

```
cout << endl;
```

```
cout << "v3: ";
```

```
for (vector<int>::size_type i = 0; i < v3.size(); ++i)
```

```
    cout << v3[i] << " ";
```

```
cout << endl;
```

```
}
```

## 2. 정렬된 범위 알고리즘

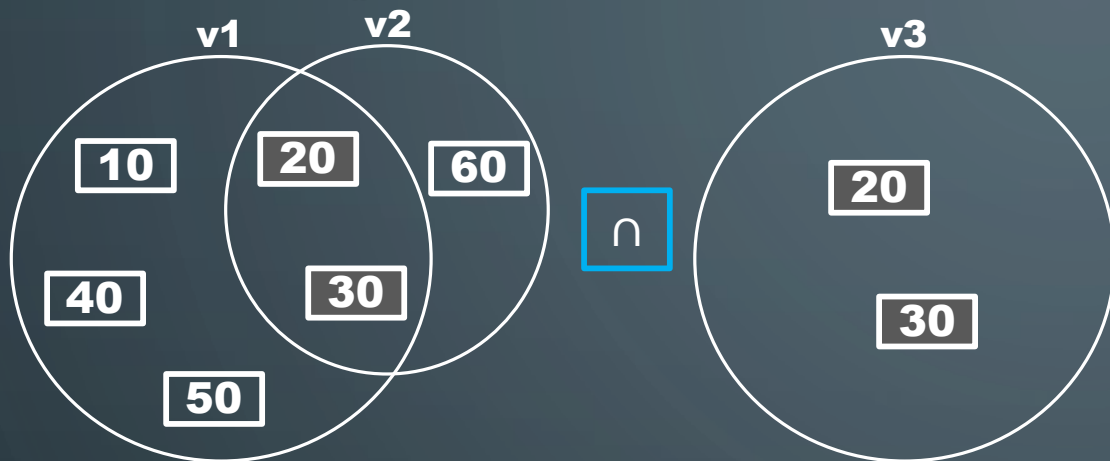
**set\_intersection(), set\_difference(), set\_symmetric\_difference()** 알고리즘 사용 예제..

```
void main()
```

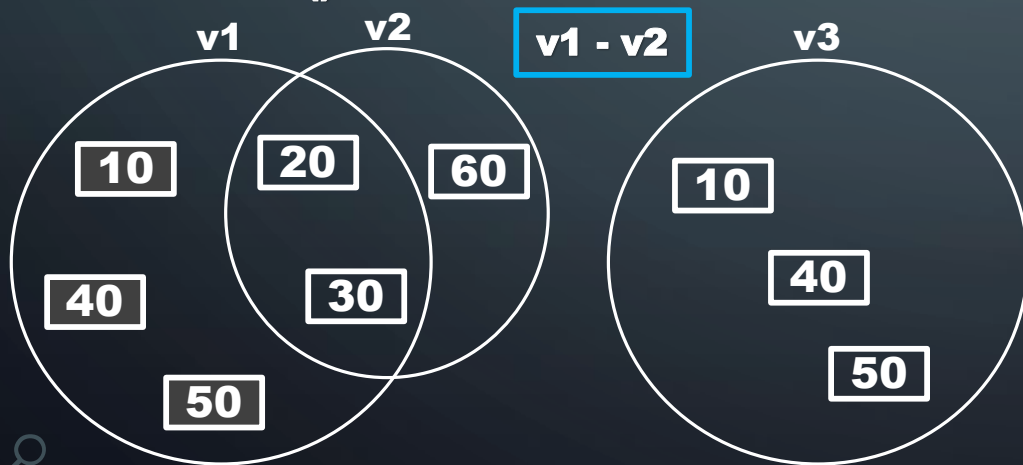
```
{  
    vector<int> v1;  
    v1.push_back(10);  
    v1.push_back(20);  
    v1.push_back(30);  
    v1.push_back(40);  
    v1.push_back(50);  
    iter_end = set_symmetric_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),  
    v3.begin());  
    cout << "대칭차집합[v3.begin(), iter_end): ";  
    for (vector<int>::iterator iter = v3.begin(); iter != iter_end; ++iter)  
        cout << *iter << " ";  
    cout << endl;  
    }  
  
    vector<int> v2;  
    v2.push_back(20);  
    v2.push_back(30);  
    v2.push_back(60);  
  
    vector<int> v3(10); //size: 10인 vector 생성  
    vector<int>::iterator iter_end;  
  
    iter_end = set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());  
    cout << "교집합[v3.begin(), iter_end): ";  
    for (vector<int>::iterator iter = v3.begin(); iter != iter_end; ++iter)  
        cout << *iter << " ";  
    cout << endl;  
    iter_end = set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());  
    cout << "차집합[v3.begin(), iter_end): ";  
    for (vector<int>::iterator iter = v3.begin(); iter != iter_end; ++iter)  
        cout << *iter << " ";  
    cout << endl;
```

## 2. 정렬된 범위 알고리즘

**set\_intersection()** - 교집합

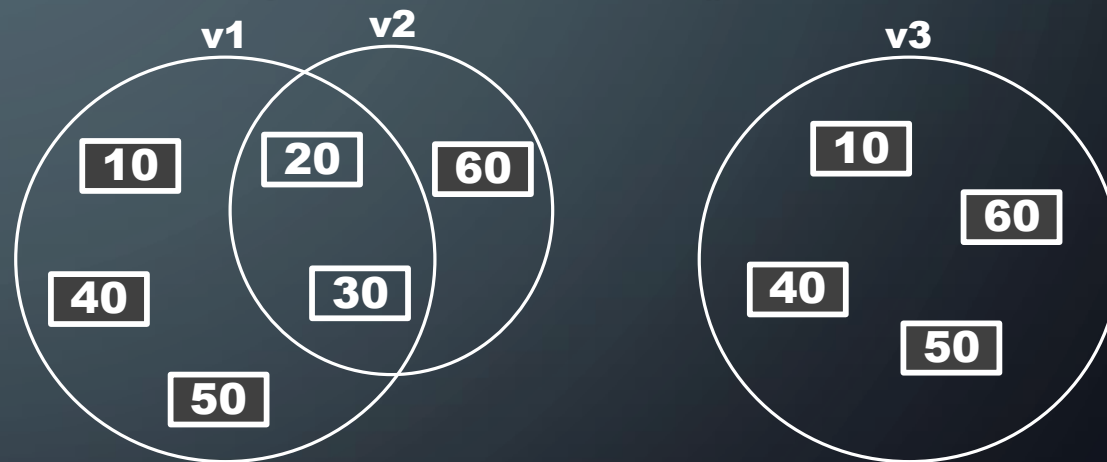


**set\_difference()** - 차집합



$$v1 \Delta v2 = (v1 - v2) \cup (v2 - v1)$$

**set\_symmetric\_difference()** - 차집합의 합집합



# 수치 알고리즘



### 3. 수치 알고리즘

- 변경 알고리즘에서 수치 변화를 하는 알고리즘이다.
- **<numeric>**헤더에 따로 정의 되어 있다.

알고리즘	설명(설명에 사용되는 p는 구간[b,e)의 반복자)
<b>x2=axxumulate(b,e,x)</b>	<b>x2</b> 는 <b>x</b> 를 초기값으로 시작한 구간 <b>[b,e)</b> 순차열 원소의 합이다.
<b>x2=axxumulate(b,e,x,f)</b>	<b>x2</b> 는 <b>x</b> 를 초기값으로 시작한 구간 <b>[b,e)</b> 순차열 원소의 누적이다. <b>f</b> 를 누적에 사용한다.
<b>x2=inner_product(b,e,b2,x)</b>	<b>x2</b> 는 <b>x</b> 를 초기값으로 시작한 구간 <b>[b,e)</b> 와 구간 <b>[b2,b2+e-b)</b> 의 내적(두 순차열의 곱의 합)이다
<b>x2=inner_product(b,e,b2,x,f1,f2)</b>	<b>x2</b> 는 <b>x</b> 를 초기값으로 시작한 구간 <b>[b,e)</b> 와 구간 <b>[b2,b2+e-b)</b> 의 모든 원소끼리 <b>f2</b> 연산 후 <b>f1</b> 연산으로 총 연산한 결과다.
<b>p=adjacent_difference(b,e,t)</b>	구간 <b>[b,e)</b> 의 인접 원소와의 차를 순차열 <b>[t,p)</b> 에 저장한다.
<b>p=adjacent_difference(b,e,t,f)</b>	구간 <b>[b,e)</b> 의 인접 원소와의 연산을 순차열 <b>[t,p)</b> 에 저장한다. <b>f</b> 를 연산에 사용한다.
<b>p=partial_sum(b,e,t)</b>	구간 <b>[b,e)</b> 의 현재 원소까지의 합을 순차열 <b>[t,p)</b> 에 저장한다.
<b>p=partial_sum(b,e,t,f)</b>	구간 <b>[b,e)</b> 의 현재 원소까지의 연산을 순차열 <b>[t,p)</b> 에 저장한다. <b>f</b> 를 연산에 사용한다.

### 3. 수치 알고리즘

**accumulate()** 알고리즘 사용 예제..

```
void main()
{
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    cout << "v: ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    //0+10+20+30+40+50
    cout << accumulate(v.begin(), v.end(), 0) << endl;
    //100+10+20+30+40+50
    cout << accumulate(v.begin(), v.end(), 100) << endl;
}
```



### 3. 수치 알고리즘

**accumulate()** 알고리즘 조건 함수 사용 예제..

```
void main()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    v.push_back(5);

    cout << "v: ";
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    // 사용자 합 연산 0+1+2+3+4+5
    cout << accumulate(v.begin(), v.end(), 0, Plus<int>()) << endl;
    // plus 합 연산 0+1+2+3+4+5
    cout << accumulate(v.begin(), v.end(), 0, plus<int>()) << endl;
    // multiplies 곱 연산 1*1*2*3*4*5
    cout << accumulate(v.begin(), v.end(), 1, multiplies<int>()) << endl;
}
```

```
template <typename T>
struct Plus
{
    T operator()(const T& left, const T& right)
    {
        return left + right;
    }
};
```

### 3. 수치 알고리즘

**inner\_product()** 알고리즘 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v1;  
    v1.push_back(1);  
    v1.push_back(2);  
    v1.push_back(3);  
    v1.push_back(4);  
    v1.push_back(5);
```

```
    vector<int> v2;  
    v2.push_back(2);  
    v2.push_back(2);  
    v2.push_back(2);  
    v2.push_back(2);  
    v2.push_back(2);
```

```
    //  $0 + 1*2 + 2*2 + 3*2 + 4*2 + 5*2$ 
```

```
    cout << inner_product(v1.begin(), v1.end(), v2.begin(), 0) << endl;
```

```
    //  $100 + 1*2 + 2*2 + 3*2 + 4*2 + 5*2$ 
```

```
    cout << inner_product(v1.begin(), v1.end(), v2.begin(), 100) << endl;
```

```
}
```

### 3. 수치 알고리즘

**inner\_product()** 알고리즘 조건 함수 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v1;  
    v1.push_back(10);  
    v1.push_back(20);  
    v1.push_back(30);  
    v1.push_back(40);  
    v1.push_back(50);
```

```
    vector<int> v2;  
    v2.push_back(2);  
    v2.push_back(2);  
    v2.push_back(2);  
    v2.push_back(2);  
    v2.push_back(2);
```

```
    // 0 + 10-2 + 20-2 + 30-2 + 40-2 + 50-2 사용자 함수 사용  
    cout << inner_product(v1.begin(), v1.end(), v2.begin(), 0, Plus, Minus) << endl;
```

```
    // 0 + 10-2 + 20-2 + 30-2 + 40-2 + 50-2 STL 함수자 사용  
    cout << inner_product(v1.begin(), v1.end(), v2.begin(), 0, plus<int>(), minus<int>()) << endl;
```

```
}
```

```
int Plus(int left, int right)
```

```
{
```

```
    return left + right;
```

```
}
```

```
int Minus(int left, int right)
```

```
{
```

```
    return left - right;
```

```
}
```

### 3. 수치 알고리즘

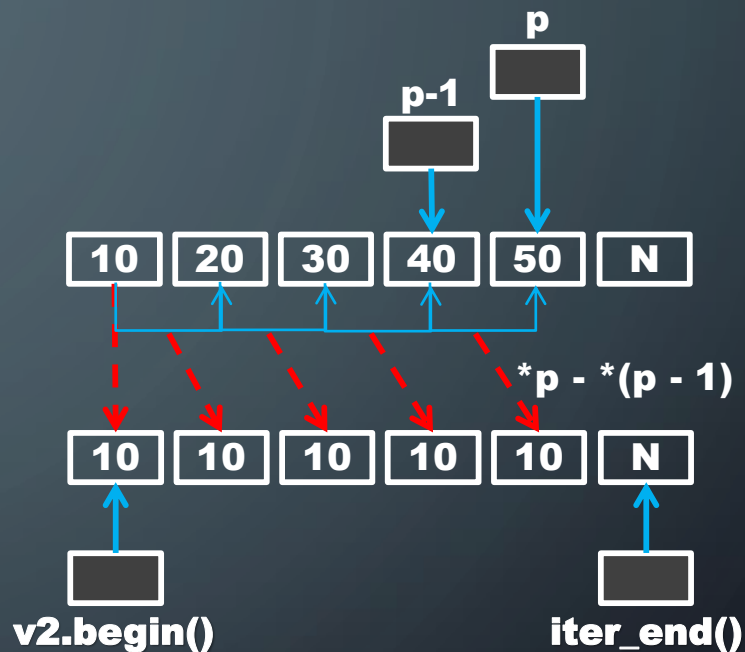
#### adjacent\_difference 알고리즘 사용 예제..

```
void main()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
    v1.push_back(50);

    cout << "v1: ";
    for (vector<int>::size_type i = 0; i < v1.size(); ++i)
        cout << v1[i] << " ";
    cout << endl;

    vector<int> v2(5); //size: 5인 vector 생성
    vector<int>::iterator iter_end;
    iter_end = adjacent_difference(v1.begin(), v1.end(), v2.begin());

    cout << "v2: ";
    for (vector<int>::size_type i = 0; i < v2.size(); ++i)
        cout << v2[i] << " ";
    cout << endl;
}
```



### 3. 수치 알고리즘

**adjacent\_difference** 알고리즘 조건 함수 사용 예제..

```
void main()
```

```
{
```

```
    vector<int> v1;  
    v1.push_back(10);  
    v1.push_back(20);  
    v1.push_back(30);  
    v1.push_back(40);  
    v1.push_back(50);
```

```
    cout << "v1: ";  
    for (vector<int>::size_type i = 0; i < v1.size(); ++i)  
        cout << v1[i] << " ";  
    cout << endl;
```

```
    vector<int> v2(5); //size: 5인 vector 생성
```

```
    vector<int>::iterator iter_end;
```

```
    //iter_end = adjacent_difference(v1.begin(), v1.end(), v2.begin(), plus<int>());
```

```
    iter_end = adjacent_difference(v1.begin(), v1.end(), v2.begin(), Plus);
```

```
    cout << "v2: ";  
    for (vector<int>::size_type i = 0; i < v2.size(); ++i)  
        cout << v2[i] << " ";  
    cout << endl;
```

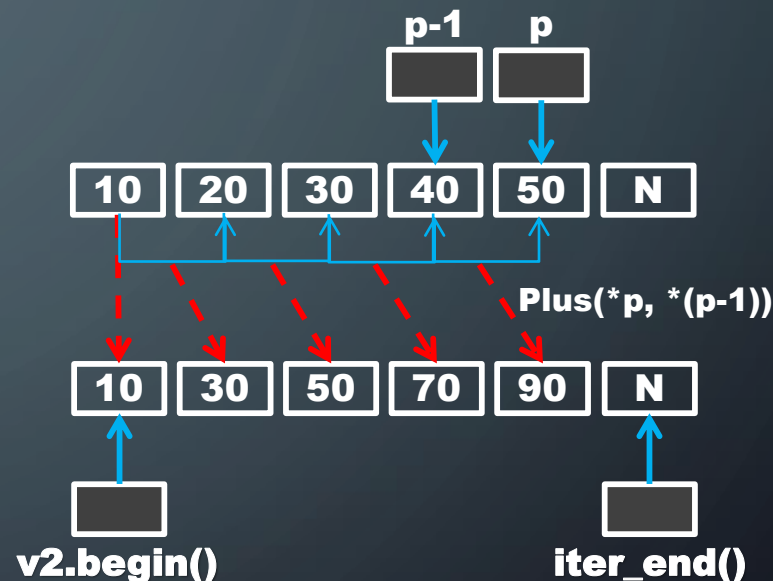
```
}
```

```
int Plus(int left, int right)
```

```
{
```

```
    return left + right;
```

```
}
```



### 3. 수치 알고리즘

#### partial\_sum() 알고리즘 사용 예제..

```
void main()
```

```
{
```

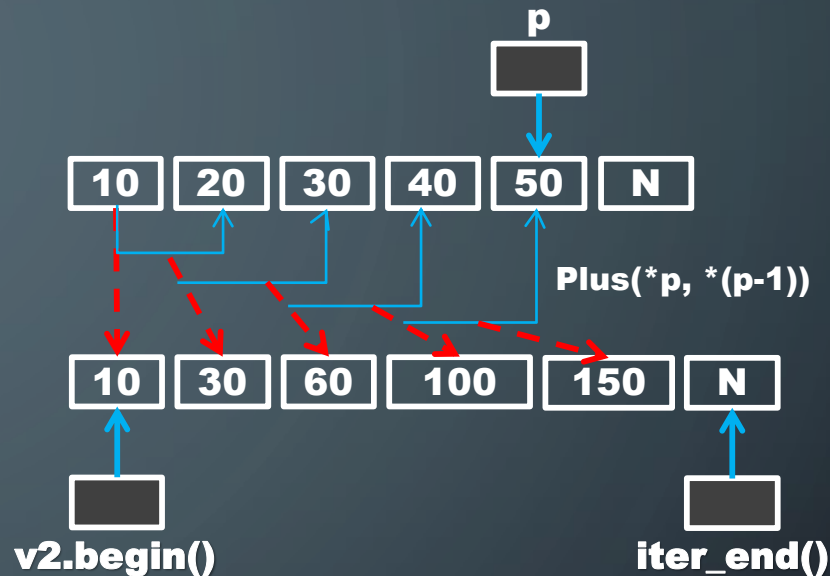
```
    vector<int> v1;  
    v1.push_back(10);  
    v1.push_back(20);  
    v1.push_back(30);  
    v1.push_back(40);  
    v1.push_back(50);
```

```
    cout << "v1: ";  
    for (vector<int>::size_type i = 0; i < v1.size(); ++i)  
        cout << v1[i] << " ";  
    cout << endl;
```

```
    vector<int> v2(5); //size: 5인 vector 생성  
    vector<int>::iterator iter_end;  
    iter_end = partial_sum(v1.begin(), v1.end(), v2.begin());
```

```
    cout << "v2: ";  
    for (vector<int>::size_type i = 0; i < v2.size(); ++i)  
        cout << v2[i] << " ";  
    cout << endl;
```

```
}
```





### 3. 수치 알고리즘

**partial\_sum()** 알고리즘 조건 함수 사용 예제..

```
void main()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
    v1.push_back(50);

    cout << "v1: ";
    for (vector<int>::size_type i = 0; i < v1.size(); ++i)
        cout << v1[i] << " ";
    cout << endl;

    vector<int> v2(5); //size: 5인 vector 생성
    vector<int>::iterator iter_end;
    //iter_end = partial_sum(v1.begin(), v1.end(), v2.begin(), multiplies<int>());
    iter_end = partial_sum(v1.begin(), v1.end(), v2.begin(), Multi);

    cout << "v2: ";
    for (vector<int>::size_type i = 0; i < v2.size(); ++i)
        cout << v2[i] << " ";
    cout << endl;
}
```

```
int Multi(int left, int right)
{
    return left * right;
}
```