```
/*
 * STL Algorithm Walkthrough:
 *      Non-modifying Algorithms
 *      count, min and max, compare, linear search,
attribute
 */


// C++ 11 Lambda Function:
num = count_if(vec.begin(), vec.end(), [](int x){return x<10;});

bool lessThan10(int x) {
   return x<10;
}

vector<int> vec = {9,60,90,8,45,87,90,69,69,55,7};
vector<int> vec2 = {9,60,70,8,45,87};
vector<int>::iterator itr, itr2;
pair<vector<int>::iterator, vector<int>::iterator> pair_of_itr;

// C++ 03: some algorithms can be found in tr1 or boost

vector<int> vec = {9,60,90,8,45,87,90,69,69,55,7};

// 1. Counting
//     Algorithm   Data              Operation
int n = count(vec.begin()+2, vec.end()-1, 69);   // 2
int m = count_if(vec.begin(), vec.end(), [](int x){return x==69;}); // 3
int m = count_if(vec.begin(), vec.end(), [](int x){return x<10;}); // 3

// 2.  Min and Max
itr = max_element(vec.begin()+2, vec.end());  // 90
// It returns the first max value
itr = max_element(vec.begin(), vec.end(),
                  [](int x, int y){ return (x%10)<(y%10);}); // 9

// Most algorithms have a simple form and a generalized form

itr = min_element(vec.begin(), vec.end());  // 7
// Generalized form: min_element()

pair_of_itr = minmax_element(vec.begin(), vec.end(),  // {60, 69}
                                  [](int x, int y){ return
(x%10)<(y%10);});
// returns a pair, which contains first of min and last of max

// 3. Linear Searching (used when data is not sorted)
//    Returns the first match
itr = find(vec.begin(), vec.end(), 55);

itr = find_if(vec.begin(), vec.end(), [](int x){ return x>80; });
```

```
itr = find_if_not(vec.begin(), vec.end(), [](int x){ return x>80; });

itr = search_n(vec.begin(), vec.end(), 2, 69);  // Consecutive 2 items of
69
// Generalized form: search_n()

// Search subrange
vector<int> sub = {45, 87, 90};
itr = search( vec.begin(), vec.end(), sub.begin(), sub.end());
      // search first subrange
itr = find_end( vec.begin(), vec.end(), sub.begin(), sub.end());
      // search last subrange
// Generalized form: search(), find_end()

// Search any_of
vector<int> items  = {87, 69};
itr = find_first_of(vec.begin(), vec.end(), items.begin(), items.end());
      // Search any one of the item in items
itr = find_first_of(vec.begin(), vec.end(), items.begin(), items.end(),
                          [](int x, int y) { return x==y*4;});
      // Search any one of the item in items that satisfy: x==y*4;

// Search Adjacent
itr = adjacent_find(vec.begin(), vec.end());  // find two adjacent items
that
                                              // are same
itr = adjacent_find(vec.begin(), vec.end(), [](int x, int y){ return
x==y*4;});
           // find two adjacent items that satisfy: x==y*4;

// 4. Comparing Ranges
if (equal(vec.begin(), vec.end(), vec2.begin())) {
  cout << "vec and vec2 are same.\n";
}

if (is_permutation(vec.begin(), vec.end(), vec2.begin())) {
      cout << "vec and vec2 have same items, but in differenct order.\n";

}

pair_of_itr = mismatch(vec.begin(), vec.end(), vec2.begin());
// find first difference
// pair_of_itr.first is an iterator of vec
// pair_of_itr.second is an iterator of vec2

//Lexicographical Comparison: one-by-one comparison with "less than"
lexicographical_compare(vec.begin(), vec.end(), vec2.begin(),
vec2.end());
// {1,2,3,5} < {1,2,4,5}
// {1,2}     < {1,2,3}

// Generalized forms:
//   equal(), is_permutation(), mismatch(), lexicographical_compare()
```

```
// 5. Check Attributes
is_sorted(vec.begin(), vec.end());  // Check if vec is sorted

itr = is_sorted_until(vec.begin(), vec.end());
// itr points to first place to where elements are no longer sorted
// Generalized forms: is_sorted(), is_sorted_until()

is_partitioned(vec.begin(), vec.end(), [](int x){return x>80;} );
                  // Check if vec is partitioned according to the
condition of (x>80)

is_heap(vec.begin(), vec.end());  // Check if vec is a heap
itr = is_heap_until(vec.begin(), vec.end());  // find the first place
where it
                                                  // is no longer a heap
// Generalized forms: is_heap(), is_heap_until()

// All, any, none
all_of(vec.begin(), vec.end(), [](int x) {return x>80} );
// If all of vec is bigger than 80

any_of(vec.begin(), vec.end(), [](int x) {return x>80} );
// If any of vec is bigger than 80

none_of(vec.begin(), vec.end(), [](int x) {return x>80} );
// If none of vec is bigger than 80
```

```
/*
 * Algorithm Walkthrough:
 *   Value-changing Algorithm - Changes the element
values
 *   copy, move, transform, swap, fill, replace,
remove
 */

vector<int> vec = {9,60,70,8,45,87,90};     // 7 items
vector<int> vec2 = {0,0,0,0,0,0,0,0,0,0,0}; // 11 items
vector<int>::iterator itr, itr2;
pair<vector<int>::iterator, vector<int>::iterator> pair_of_itr;

vector<int> vec = {9,60,70,8,45,87,90};     // 7 items
vector<int> vec2 = {0,0,0,0,0,0,0,0,0,0,0}; // 11 items

// 1. Copy
copy(vec.begin(), vec.end(), // Source
       vec2.begin());          // Destination

copy_if(vec.begin(), vec.end(),      // Source
```

```
            vec2.begin(),                 // Destination
            [](int x){ return x>80;});    // Condition
// vec2: {87, 90, 0, 0, 0, 0, 0, 0, 0, 0, 0}

copy_n(vec.begin(),  4, vec2.begin());
// vec2: {9, 60, 70, 8, 0, 0, 0, 0, 0, 0, 0}

copy_backward(vec.begin(),  vec.end(),  // Source
                  vec2.end());              // Destination
// vec2: {0, 0, 0, 0, 9, 60, 70, 8, 45, 87, 90}

// 2. Move
vector<string> vec = {"apple", "orange", "pear", "grape"}; // 4 items
vector<string> vec2 = {"", "", "", "", "", ""};            // 6 items

move(vec.begin(), vec.end(), vec2.begin());
// vec:  {"", "", "", ""}  // Undefined
// vec2: {"apple", "orange", "pear", "grape", "", ""};
//
// If move semantics are defined for the element type, elements are moved
over,
// otherwise they are copied over with copy constructor, just like
copy().

move_backward(vec.begin(), vec.end(), vec2.end());
// vec2: {"", "", "apple", "orange", "pear", "grape"};

vector<int> vec = {9,60,70,8,45,87,90};     // 7 items
vector<int> vec2 = {9,60,70,8,45,87,90};     // 7 items
vector<int> vec3 = {0,0,0,0,0,0,0,0,0,0,0}; // 11 items

// 3. Transform
transform(vec.begin(), vec.end(),    // Source
               vec3.begin(),              // Destination
                  [](int x){ return x-1;});  // Operation

transform(vec.begin(), vec.end(),           // Source #1
          vec2.begin(),                     // Source #2
               vec3.begin(),                      // Destination
             [](int x, int y){ return x+y;});  // Operation
         // Add items from vec and vec2 and save in vec3
         // vec3[0] = vec[0] + vec2[0]
         // vec3[1] = vec[1] + vec2[1]
         // ...

// 4. Swap - two way copying
swap_ranges(vec.begin(), vec.end(), vec2.begin());

// 5. Fill
vector<int> vec = {0, 0, 0, 0, 0};

fill(vec.begin(), vec.end(), 9); // vec: {9, 9, 9, 9, 9}

fill_n(vec.begin(), 3, 9);       // vec: {9, 9, 9, 0, 0}
```

```
generate(vec.begin(), vec.end(), rand);
generate_n(vec.begin(), 3, rand);

// 6. Replace
replace(vec.begin(), vec.end(),  // Data Range
             6,                        // Old value condition
             9);                       // new value

replace_if(vec.begin(), vec.end(),     // Data Range
                  [](int x){return x>80;},    // Old value condition
                  9);                          // new value

replace_copy(vec.begin(), vec.end(),  // Source
                  vec2.begin(),              // Destination
                  6,                         // Old value condition
                  9);                        // new value
  // Generalized form: replace_copy_if()

// 7. Remove
remove(vec.begin(), vec.end(), 3);   // Remove all 3's
remove_if(vec.begin(), vec.end(), [](int x){return x>80;});
       // Remove items bigger than 80

remove_copy(vec.begin(), vec.end(),  // Source
                  vec2.begin(),            // Destination
                       6);                      // Condition
   // Remove all 6's, and copy the remain items to vec2
   // Generalized form: remove_copy_if()

unique(vec.begin(), vec.end());   // Remove consecutive equal elements

unique(vec.begin(), vec.end(), less<int>());
        // Remove elements whose previous element is less than itself

unique_copy(vec.begin(), vec.end(), vec2.begin());
// Remove consecutive equal elements, and then copy the uniquified items
to vec2
// Generalized form: unique_copy()

/*
 * Order-Changing Algorithms:
 *   - reverse, rotate, permute, shuffle
 *
 * They changes the order of elements in container, but not necessarily the
 * elements themselves.
 */

vector<int> vec =  {9,60,70,8,45,87,90};     // 7 items
vector<int> vec2 = {0,0,0,0,0,0,0};     // 7 items

// 1. Reverse
reverse(vec.begin()+1, vec.end()-1);
// vec: {9,87,45,8,70,60,90};     // 7 items
```

```
reverse_copy(vec.begin()+1, vec.end()-1, vec2.begin());
// vec2: {87,45,8,70,60,0,0};


// 2. Rotate
rotate(vec.begin(), vec.begin()+3, vec.end());
// vec: {8,45,87,90,9,60,70};     // 7 items

rotate_copy(vec.begin(), vec.begin()+3, vec.end(),  // Source
            vec2.begin());                                  // Destination
       // Copy vec to vec2 in rotated order
       // vec is unchanged

// 3. Permute
next_permutation(vec.begin(), vec.end());
                            //Lexicographically next greater permutation
prev_permutation(vec.begin(), vec.end());
                            //Lexicographically next smaller permutation
// {1,2,3,5} < {1,2,4,4}
// {1,2}     < {1,2,3}

//Sorted in ascending order:  {8, 9, 45, 60, 70, 87, 90}
//                            - Lexicographically smallest
//
//Sorted in descending order: {90, 87, 70, 60, 45, 9, 8}
//                            - Lexicographically greatest

// Generalized form: next_permutation(), prev_permutation()

// 4. Shuffle
//    - Rearrange the elements randomly
//      (swap each element with a randomly selected element)
random_shuffle(vec.begin(), vec.end());
random_shuffle(vec.begin(), vec.end(), rand);

// C++ 11
shuffle(vec.begin(), vec.end(), default_random_engine());
// Better random number generation

/*
 * Sorted Data Algorithms
 *   - Algorithms that require data being pre-sorted
 *   - Binary search, merge, set operations
 */

// Note: Every sorted data algorithm has a generalized form with a same
name.

vector<int> vec = {8,9,9,9,45,87,90};     // 7 items

// 1. Binary Search
// Search Elements
```

```
bool found = binary_search(vec.begin(), vec.end(), 9);  // check if 9 is
in vec

vector<int> s = {9, 45, 66};
bool found = includes(vec.begin(), vec.end(),     // Range #1
                      s.begin(), s.end());        // Range #2
// Return true if all elements of s is included in vec
// Both vec and s must be sorted

// Search Position
itr = lower_bound(vec.begin(), vec.end(), 9);  // vec[1]
// Find the first position where 9 could be inserted and still keep the
sorting.

itr = upper_bound(vec.begin(), vec.end(), 9);  // vec[4]
// Find the last position where 9 could be inserted and still keep the
sorting.

pair_of_itr = equal_range(vec.begin(), vec.end(), 9);
// Returns both first and last position

// 2. Merge
vector<int> vec = {8,9,9,10};
vector<int> vec2 = {7,9,10};
merge(vec.begin(), vec.end(),      // Input Range #1
          vec2.begin(), vec2.end(),    // input Range #2
          vec_out.begin());               // Output
      // Both vec and vec2 should be sorted (same for the set operation)
      // Nothing is dropped, all duplicates are kept.
// vec_out: {7,8,9,9,9,10,10}

vector<int> vec = {1,2,3,4,1,2,3,4,5}  // Both part of vec are already
sorted
inplace_merge(vec.begin(), vec.begin()+4, vec.end());
// vec: {1,1,2,2,3,3,4,4,5}  - One step of merge sort

// 3. Set operations
//    - Both vec and vec3 should be sorted
//    - The resulted data is also sorted
vector<int> vec = {8,9,9,10};
vector<int> vec2 = {7,9,10};
vector<int> vec_out[5];
set_union(vec.begin(), vec.end(),      // Input Range #1
              vec2.begin(), vec2.end(),    // input Range #2
              vec_out.begin());               // Output
// if X is in both vec and vec2, only one X is kept in vec_out
// vec_out: {7,8,9,9,10}

set_intersection(vec.begin(), vec.end(),      // Input Range #1
                     vec2.begin(), vec2.end(),    // input Range #2
                     vec_out.begin());               // Output
// Only the items that are in both vec and vec2 are saved in vec_out
// vec_out: {9,10,0,0,0}
```

```
vector<int> vec = {8,9,9,10};
vector<int> vec2 = {7,9,10};
vector<int> vec_out[5];
set_difference(vec.begin(), vec.end(),      // Input Range #1
                    vec2.begin(), vec2.end(),    // input Range #2
                    vec_out.begin());               // Output
// Only the items that are in vec but not in vec2 are saved in vec_out
// vec_out: {8,9,0,0,0}

set_symmetric_difference(vec.begin(), vec.end(),      // Input Range #1
                    vec2.begin(), vec2.end(),       // input Range #2
                    vec_out.begin());               // Output
// vec_out has items from either vec or vec2, but not from both
// vec_out: {7,8,9,0,0}

/*
 *  Numeric Algorithms (in <numeric>)
 *   - Accumulate, inner product, partial sum, adjacent difference
 */

// 1. Accumulate

int x = accumulate(vec.begin(), vec.end(), 10);
// 10 + vec[0] + vec[1] + vec[2] + ...

int x = accumulate(vec.begin(), vec.end(), 10, multiplies<int>());
// 10 * vec[0] * vec[1] * vec[2] * ...

// 2. Inner Product
//vector<int> vec = {9,60,70,8,45,87,90};     // 7 items
int x = inner_product(vec.begin(), vec.begin()+3,  // Range #1
                           vec.end()-3,                  // Range #2
                                10);                          // Init
Value
// 10 + vec[0]*vec[4] + vec[2]*vec[5] + vec[3]*vec[6]

int x = inner_product(vec.begin(), vec.begin()+3,  // Range #1
                            vec.end()-3,                  // Range #2
                                 10,                           // Init
Value
                                 multiplies<int>(),
                                 plus<int>());
// 10 * (vec[0]+vec[4]) * (vec[2]+vec[5]) * (vec[3]+vec[6])

// 3. Partial Sum
partial_sum(vec.begin(), vec.end(), vec2.begin());
// vec2[0] = vec[0]
// vec2[1] = vec[0] + vec[1];
// vec2[2] = vec[0] + vec[1] + vec[2];
// vec2[3] = vec[0] + vec[1] + vec[2] + vec[3];
// ...

partial_sum(vec.begin(), vec.end(), vec2.begin(), multiplies<int>());
```

```
// 4. Adjacent Difference
adjacent_difference(vec.begin(), vec.end(), vec2.begin());
// vec2[0] = vec[0]
// vec2[1] = vec[1] - vec[0];
// vec2[2] = vec[2] - vec[1];
// vec2[3] = vec[3] - vec[2];
// ...

adjacent_difference(vec.begin(), vec.end(), vec2.begin(), plus<int>());
```