



STL Algorithms

Principles and Practice

Victor Ciura - Technical Lead

Gabriel Diaconița - Senior Software Developer

March 2020

Introduction to Algorithms

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

March, 1990

**A classic, priceless book
30 years after publication**





Student solutions for Homeworks

- Anghel Matei
- Alex Costinel
- Marina Rusu
- Mircea Aurelian Badoi
- Alexandra Catalina Barcan
- Bogdan-Andrei Zanfir
- Cristiana-Mădălina Prodan
- Bogdan Draghici
- Luchian Ionica
- Daniel Constantin

You can still send us your solutions until April 1st

Email solutions to pca@caphyon.com



Homework

Server Nodes

We have a huge network of server nodes.

Each server node contains a copy of a particular **data Value** (not necessarily unique).

`class Value` is a **Regular** type.

{ Assignable + Constructible + EqualityComparable + LessThanComparable }

The network is constructed in such a way that the nodes are **sorted ascending** with respect to their **Value** but their sequence might be **rotated** (left) by some offset.

Eg.

For the **ordered** node values:

{ **A, B, C, D, E, F, G, H** }

The **actual network** configuration might look like:

{ **D, E, F, G, H, A, B, C** }



Homework


Server Nodes

The network exposes the following APIs:

```
// gives the total number of nodes -  $O(1)$ 
size_t Count() const;

// retrieves the data from a given node -  $O(1)$ 
const Value & GetData(size_t index) const;

// iterator interface for the network nodes
vector<Value>::const_iterator BeginNodes() const;
vector<Value>::const_iterator EndNodes() const;
```

 Implement a new API for the network, that efficiently finds a server node (address) containing a given data **Value**.

```
size_t GetNode(const Value & data) const
{
    // implement this
}
```



Homework

Fun with squares²

“You have a vector of integers sorted in non-decreasing order; compute the squares of each number, also in sorted non-decreasing order.”

-5 -2 0 6 8 => 0 4 25 36 64

A naive solution:

```
for (auto & e : vec)
    e *= e;
std::sort(begin(vec), end(vec));
```

- Can you implement a solution avoiding the **cost of sorting** ?
- Explore **in-place** solutions (like the naive one), as well as using a **separate** vector for the result
- Can you trade **space** for **speed** in this example ?
- What STL **algorithms** can you identify as useful here ?
- Can you find a solution that is **linear** or *better* ?
- **Tip:** take advantage of all the **constraints** of the problem and the initial conditions

 **A deceptively simple problem**

Recap<T>

STL and Its Design Principles

Generic Programming



- algorithms are associated with a **set of common properties**
Eg. op { +, *, min, max } => associative operations => reorder operands
=> parallelize + reduction (std::accumulate)
- find the most general representation of algorithms (**abstraction**)
- exists a **generic algorithm** behind every WHILE or FOR loop
- natural extension of 4,000 years of **mathematics**

Alexander Stepanov (2002),

<https://www.youtube.com/watch?v=COuHLky7E2Q>

Generic Programming Drawbacks

- abstraction penalty
- implementation in the interface
- early binding
- horrible error messages (*no formal specification* of interfaces, **yet**)
- duck typing
- algorithm could work on some data types, but fail to work/compile on some other new data structures (different iterator category, no copy semantics, etc)

We need to fully specify **requirements** on algorithm types => **Concepts**

Named Requirements

Some examples from STL:

- `DefaultConstructible`, `MoveConstructible`, `CopyConstructible`
- `MoveAssignable`, `CopyAssignable`,
- `Destructible`
- `EqualityComparable`, `LessThanComparable`
- `Predicate`, `BinaryPredicate`
- `Compare`
- `FunctionObject`
- `Container`, `SequenceContainer`, `ContiguousContainer`,
`AssociativeContainer`
- `Iterator`
 - `InputIterator`, `OutputIterator`
 - `ForwardIterator`, `BidirectionalIterator`, `RandomAccessIterator`

Named Requirements

Named requirements are used in the normative text of the C++ standard to define the **expectations** of the standard library.

Some of these requirements were formalized in **C++20** using **concepts**.

If you're not using C++20 yet, the **burden is on YOU** to ensure that library templates are instantiated with template arguments that satisfy these requirements.

What Is A **Concept**, Anyway ?

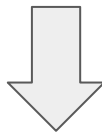
Formal specification of concepts makes it possible to **verify** that template arguments satisfy the **expectations** of a template or function during overload resolution and template specialization (requirements).

Each concept is a **predicate**, evaluated at *compile time*, and becomes a part of the **interface of a template** where it is used as a constraint.

What Is A **Concept**, Anyway ?

The whole STL has been **conceptified**, starting with C++20

```
template< class InputIt, class UnaryPredicate >
constexpr bool any_of( InputIt first, InputIt last, UnaryPredicate pred );
```



std::ranges::

```
template< std::input_iterator I, std::sentinel_for<I> S,
          class Proj = std::identity,
          std::indirect_unary_predicate<std::projected<I, Proj>> Pred >
constexpr bool any_of( I first, S last, Pred pred, Proj proj = {} );
```

} **constraints**

<https://en.cppreference.com/w/cpp/language/constraints>

What's the Practical Upside ?

If I'm not a library writer 🤓,
Why Do I Care ?

What's the Practical Upside ?

Using STL algorithms & data structures

Designing & exposing your own vocabulary types
(interfaces, APIs)

Compare Concept

Why is this one special ?

Because ~50 STL facilities (algorithms & data structures) expect a *Compare* type.

```
template< class RandomIt, class Compare >  
void sort( RandomIt first, RandomIt last, Compare comp );
```

Concept relations:

Compare << *BinaryPredicate* << *Predicate* << *FunctionObject* << *Callable*

Compare Concept

What are the *requirements* for a `Compare` type ?

`Compare` << `BinaryPredicate` << `Predicate` << `FunctionObject` << `Callable`

```
bool comp(*iter1, *iter2);
```

But what kind of **ordering** relationship is needed for the *elements* of the collection ?



https://en.cppreference.com/w/cpp/named_req/Compare

Compare Concept

But what kind of *ordering* relationship is needed



Irreflexivity	$\forall a, \text{comp}(a, a) == \text{false}$
Antisymmetry	$\forall a, b, \text{if } \text{comp}(a, b) == \text{true} \Rightarrow \text{comp}(b, a) == \text{false}$
Transitivity	$\forall a, b, c, \text{if } \text{comp}(a, b) == \text{true} \text{ and } \text{comp}(b, c) == \text{true} \Rightarrow \text{comp}(a, c) == \text{true}$

Compare Examples

```
vector<string> v = { ... };
```

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

```
sort(v.begin(), v.end(), less<>());
```

```
sort(v.begin(), v.end(), [](const string & s1, const string & s2)
{
    return s1 < s2;
});
```

```
sort(v.begin(), v.end(), [](const string & s1, const string & s2)
{
    return strcmp(s1.c_str(), s2.c_str()) < 0;
});
```

Compare Examples

```
struct Point { int x; int y; };  
vector<Point> v = { ... };  
  
sort(v.begin(), v.end(), [](const Point & p1,  
                             const Point & p2)  
{  
    return (p1.x < p2.x) && (p1.y < p2.y);  
});
```

Is this a good *Compare* predicate for 2D points ?

Compare Examples

Definition:

if `comp(a,b) == false` **&&** `comp(b,a) == false`
=> **a** and **b** are **equivalent**

Let { P1, P2, P3 }

$x_1 < x_2$; $y_1 > y_2$;

$x_1 < x_3$; $y_1 > y_3$;

$x_2 < x_3$; $y_2 < y_3$;

=>

P2 and P1 are unordered (P2 ? P1) `comp(P2,P1) == false` **&&** `comp(P1,P2) == false`

P1 and P3 are unordered (P1 ? P3) `comp(P1,P3) == false` **&&** `comp(P3,P1) == false`

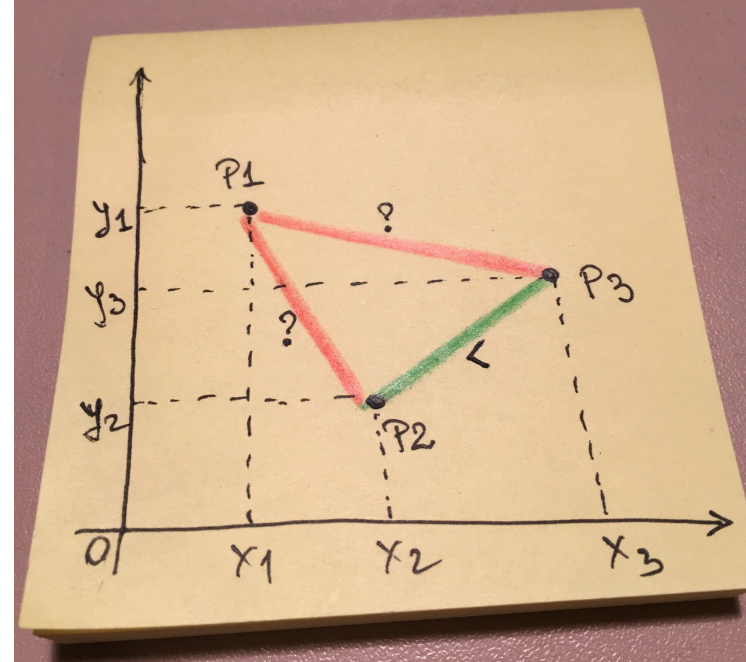
P2 and P3 are ordered (P2 < P3) `comp(P2,P3) == true` **&&** `comp(P3,P2) == false`

=>

P2 is **equivalent** to P1

P1 is **equivalent** to P3

P2 is **less than** P3



Compare Concept

Partial ordering relationship is not enough :(

Compare needs a **stronger** constraint

Strict weak ordering = *Partial ordering* + **Transitivity of Equivalence**

where:

equiv(a,b) : `comp(a,b)==false && comp(b,a)==false`

Strict weak ordering

Partial ordering relationship: *Irreflexivity* + *Antisymmetry* + *Transitivity*

Strict weak ordering relationship: *Partial ordering* + *Transitivity of Equivalence*

Total ordering relationship: **Strict weak ordering** + **equivalence** must be the same as **equality**

Irreflexivity	$\forall a, \text{comp}(a, a) == \text{false}$
Antisymmetry	$\forall a, b, \text{if } \text{comp}(a, b) == \text{true} \Rightarrow \text{comp}(b, a) == \text{false}$
Transitivity	$\forall a, b, c, \text{if } \text{comp}(a, b) == \text{true} \text{ and } \text{comp}(b, c) == \text{true} \Rightarrow \text{comp}(a, c) == \text{true}$
Transitivity of equivalence	$\forall a, b, c, \text{if } \text{equiv}(a, b) == \text{true} \text{ and } \text{equiv}(b, c) == \text{true} \Rightarrow \text{equiv}(a, c) == \text{true}$

where:

$\text{equiv}(a, b) : \text{comp}(a, b) == \text{false} \ \&\& \ \text{comp}(b, a) == \text{false}$

https://en.wikipedia.org/wiki/Weak_ordering#Strict_weak_orderings

Compare Examples

```
struct Point { int x; int y; };  
vector<Point> v = { ... };  
  
sort(v.begin(), v.end(), [](const Point & p1,  
                             const Point & p2)  
{  
    return (p1.x * p1.x + p1.y * p1.y) <  
           (p2.x * p2.x + p2.y * p2.y);  
});
```

Is this a good Compare predicate for 2D points ?

Compare Examples

```
struct Point { int x; int y; };  
vector<Point> v = { ... };  
  
sort(v.begin(), v.end(), [](const Point & p1,  
                             const Point & p2)  
{  
    if (p1.x < p2.x) return true;  
    if (p2.x < p1.x) return false;  
  
    return p1.y < p2.y;  
});
```

Is this a good Compare predicate for 2D points ?

Compare Examples

The general idea is to pick an **order** in which to compare *elements/parts* of the object.
(in our example we first compared by **x** coordinate, and then by **y** coordinate for equivalent **x**)

This strategy is analogous to how a **dictionary** works, so it is often called "*dictionary order*", or "*lexicographical order*".

The STL implements dictionary ordering in at least three places:

std::pair<T, U> - defines the six comparison operators in terms of the corresponding operators of the pair's components

std::tuple< ... Types> - generalization of pair

std::lexicographical_compare() algorithm

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- ...

Compare



The **Spaceship** has landed !

operator <=>

- Consistent comparison
- Relationship strength (comparison category traits):
 - `strong_ordering`
 - `weak_ordering`
 - `partial_ordering`
 - `strong_equality`
 - `weak_equality`
- STL implementation for containers & utility classes
- Examples

Jonathan Müller “Using C++20's Three-way Comparison <=>”
<https://www.youtube.com/watch?v=8jNXy3K2Wpk>

STL Algorithms - Principles and Practice

“Show me the code”

A common task...

Remove elements matching a predicate.

Given:

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7 };
```

How do we remove all **even** numbers ?

A common task...

Remove elements matching a predicate.

<https://en.cppreference.com/w/cpp/container/vector/erase>

```
iterator vector::erase(const_iterator first, const_iterator last);
```

<https://en.cppreference.com/w/cpp/algorithm/remove>

```
template< class ForwardIt, class UnaryPredicate >  
ForwardIt std::remove_if(ForwardIt first, ForwardIt last, UnaryPredicate p);
```

Erase-Remove Idiom

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7 };
```

```
v.erase( std::remove_if(v.begin(), v.end(),  
                        [] (int i) { return (i & 1) == 0; } ),  
        v.end() );
```

How do you think this works ?

“`remove_if()` moves all the elements you want to remove to the **end** of the vector, then the `erase` gets rid of them.”

```
v = { 1, 3, 5, 7, 2, 4, 6 }
```

WRONG !

Erase-Remove Idiom

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7 };
```

```
v.erase( std::remove_if(v.begin(), v.end(),  
                        [] (int i) { return (i & 1) == 0; }),  
        v.end() );
```

This **isn't** what `std::remove_if()` does !

If it did that – which is **more work** than it does – it would in fact be `std::partition()`.

What `std::remove()` does is move the elements that **won't** be removed **to the beginning**.

Erase-Remove Idiom

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7 };
```

```
v.erase( std::remove_if(v.begin(), v.end(),  
                        [] (int i) { return (i & 1) == 0; } ),  
        v.end() );
```

What about the elements at the **end** of the vector ?

GARBAGE !

They get *overwritten* in the process of `std::remove()` algorithm.

Before `erase()` is called: `v = { 1, 3, 5, 7, 5, 6, 7 }`



where the **iterator** returned by `remove_if()` points

Prefer Member Functions To Similarly Named Algorithms

The following member functions are available for *associative containers*:

- `.count()`
- `.find()`
- `.equal_range()`
- `.lower_bound()` // only for ordered containers
- `.upper_bound()` // only for ordered containers

The following member functions are available for `std::list`

- `.remove()` `.remove_if()`
- `.unique()`
- `.sort()`
- `.merge()`
- `.reverse()`

These member functions are always **faster** than their similarly named generic algorithms.

Why? They can leverage the *implementation details* of the underlying data structure.

Prefer Member Functions To Similarly Named Algorithms

`std::list<>` specific algorithms

`std::sort()` doesn't work on lists (Why ?)
=> call `.sort()` member function

`.remove()` and `.remove_if()` don't need to use the **erase/remove idiom**.
They directly remove matching elements from the list.

`.remove()` and `.remove_if()` are more efficient than the generic algorithms,
because they just relink nodes with the need to copy or move elements.

Prefer Member Functions To Similarly Named Algorithms

```
std::set<string> s = {...}; // 1 million elements
```

```
// worst case: 1 million comparisons
```

```
// average:  $\frac{1}{2}$  million comparisons
```

```
auto it = std::find(s.begin(), s.end(), "stl");
```

```
if (it != s.end()) {...}
```

```
// worst case: 40 comparisons
```

```
// average: 20 comparisons
```

```
auto it = s.find("stl");
```

```
if (it != s.end()) {...}
```

Why ?

Don't Trust Your Intuition: Always Benchmark !

```
static void StdFind(benchmark::State & state)
{
    std::set<std::string> items;
    for (int i = COUNT_ELEM; i >= 0; --i)
        items.insert("string #" + std::to_string(i));

    // Code before the loop is not measured
    for (auto _ : state)
    {
        auto it = std::find(items.begin(), items.end(), "STL");
        if (it != items.end())
            std::cout << "Found: " << *it << std::endl;
    }
}

BENCHMARK(StdFind);
```

```
static void SetFind(benchmark::State & state)
{
    std::set<std::string> items;
    for (int i = COUNT_ELEM; i >= 0; --i)
        items.insert("string #" + std::to_string(i));

    // Code before the loop is not measured
    for (auto _ : state)
    {
        auto it = items.find("STL");
        if (it != items.end())
            std::cout << "Found: " << *it << std::endl;
    }
}

BENCHMARK(SetFind);
```

<http://quick-bench.com>

Try increasing values for `COUNT_ELEM` : 500 >>> 500'000 >>> ...

Don't Trust Your Intuition: Always Benchmark !

```
static void ListFind(benchmark::State & state)
{
    std::list<std::string> items;
    for (int i = COUNT_ELEM; i >= 0; --i)
        items.push_back("string #" + std::to_string(i));

    // Code before the loop is not measured
    for (auto _ : state)
    {
        auto it = std::find(items.begin(), items.end(), "STL");
        if (it != items.end())
            std::cout << "Found: " << *it << std::endl;
    }
}

BENCHMARK(ListFind);
```

```
static void VectorFind(benchmark::State & state)
{
    std::vector<std::string> items;
    for (int i = COUNT_ELEM; i >= 0; --i)
        items.push_back("string #" + std::to_string(i));

    // Code before the loop is not measured
    for (auto _ : state)
    {
        auto it = std::find(items.begin(), items.end(), "STL");
        if (it != items.end())
            std::cout << "Found: " << *it << std::endl;
    }
}

BENCHMARK(VectorFind);
```


<http://quick-bench.com>

Try increasing values for COUNT_ELEM : 500 >>> 500'000 >>> ...

Binary search operations (on *sorted* ranges)

```
binary_search() // helper (incomplete interface - Why ?)
lower_bound()  // returns an iter to the first element not less than the given value
upper_bound()  // returns an iter to the first element greater than the certain value

equal_range() = { lower_bound(), upper_bound() }

// properly checking return value
auto it = lower_bound(v.begin(), v.end(), 5);
if ( it != v.end() && (*it == 5) )  Why do we need to check the value we searched for ?
{
    // found item, do something with it
}
else // not found, insert item at the correct position
{
    v.insert(it, 5);
}
```

Binary search operations (on *sorted* ranges)

Counting elements equal to a given value

```
vector<string> v = { ... }; // sorted collection
size_t num_items = std::count(v.begin(), v.end(), "stl");
```

Instead of using `std::count()` generic algorithm, use **binary search** instead.

```
auto range = std::equal_range(v.begin(), v.end(), "stl");
size_t num_items = std::distance(range.first, range.second);
```

Extend STL With Your Generic Algorithms

Eg.

```
template<class Container, class Value>
bool name_this_algorithm(Container & c, const Value & v)
{
    return std::find(begin(c), end(c), v) != end(c);
}
```


Extend STL With Your Generic Algorithms

Eg.

```
template<class Container, class Value>
void name_this_algorithm(Container & c, const Value & v)
{
    if ( std::find(begin(c), end(c), v) == end(c) )
        c.emplace_back(v);
}
```

Extend STL With Your Generic Algorithms

Eg.

```
template<class Container, class Value>
bool name_this_algorithm(Container & c, const Value & v)
{
    auto found = std::find(begin(c), end(c), v);
    if (found != end(c))
    {
        c.erase(found); // call 'erase' from STL container
        return true;
    }
    return false;
}
```

Consider Adding Range-based Versions of STL Algorithms

```
namespace range { // our <algorithm_range.h> has ~150 wrappers for std algorithms

template< class InputRange, class T > inline
typename auto find(InputRange && range, const T & value)
{
    return std::find(begin(range), end(range), value);
}

template< class InputRange, class UnaryPredicate > inline
typename auto find_if(InputRange && range, UnaryPredicate pred)
{
    return std::find_if(begin(range), end(range), pred);
}

template< class RandomAccessRange, class BinaryPredicate > inline
void sort(RandomAccessRange && range, BinaryPredicate comp)
{
    std::sort(begin(range), end(range), comp);
}

}
```

Until C++20

Consider Adding Range-based Versions of STL Algorithms

Eg.

```
vector<string> v = { ... };
```

```
auto it = range::find(v, "stl");  
string str = *it;
```

```
auto chIt = range::find(str, 't');
```

```
auto it2 = range::find_if(v, [](const auto & val) { return val.size() > 5; });
```

```
range::sort(v);
```

```
range::sort(v, [](const auto & val1, const auto & val2)  
            { return val1.size() < val2.size(); } );
```

Until C++20

STL for Competitive Programming and Software Development



Coding Test

- Test NETROM (March 25, 5:45pm)
- Test CAPHYON (April 1st, 5:45pm)
- aprox 2h
- open-books, internet
- bring your laptop
- your assigned homeworks will help prepare you for this



Demo: FUN WITH STL

We'll take on the challenge of building a fun game in C++ using the STL toolbox available at our disposal

Take it away, Gabi !

// FIN