From Mathematics to Generic Programming

Course Slides – Part 3 of 3

Version 1.0

October 5, 2015

Lecture 17

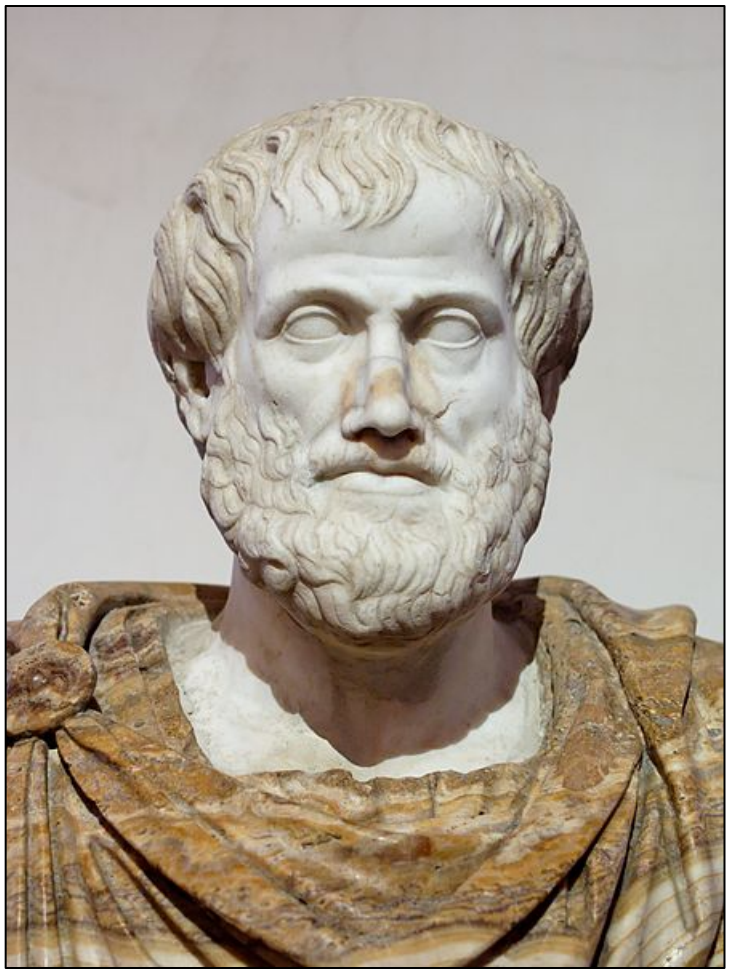
Concepts in Programming

(Covers material from Sec. 10.1-10.5)

Abstraction, Revisited

- We've been discussing abstraction in mathematics:
  - abstract algebra
- Next, we'll look at abstraction in programming:
  - generic programming and the notion of *concepts*
- Where did these ideas about abstraction originate?
  - Aristotle

Aristotle (384 BC–322 BC)



- Studied and taught at Plato's Academy for 20 years
- Tutored young prince of Macedon, who became Alexander the Great
- Founded his own school, the Lyceum
- Researched and wrote about nearly every subject, from Aesthetics to Zoology

Abstraction in Aristotle's *Categories*

Distinction between:

- Individual
- Species
  - Incorporates all the "essential" properties of a type of thing
- Genus (plural: Genera)
  - Contains many species, each identified by its *differentia* (things that differentiate it from other species in the genus)

Values and Types

- A *datum* is a sequence of bits.
  - Example: 01000001
- A *value* is a datum together with its interpretation.
  - Example: 01000001 interpreted as the character 'A'
  - Example: 01000001 interpreted as the integer 65
  - A datum without an interpretation has no meaning
- A *value type* is a set of values sharing a common interpretation.

Objects

- An *object* is a collection of bits in memory that contain a value of a given value type.
  - There is no requirement that bits of an object be contiguous. Many objects have *remote parts*.
  - An object is *immutable* if its value never changes, and *mutable* otherwise.
  - An object is *unrestricted* if it can contain any value of its value type.

Object Types

- An *object type* is a uniform method of storing and retrieving values of a given value type from a particular object, given its address.
  - What we call “types” in programming languages are object types.

Concepts

A concept in programming is:

- a way to describe a family of related object types
- (Equivalently) a set of requirements on types

Concepts are to types what types are to instances.

Equivalent Levels of Abstraction

Natural Science	Mathematics	Programming	Programming Examples
Genus	Theory	Concept	**Integral, Character**
Species	Model	Type or Class	`uint8_t`, `char`
Individual	Element	Instance	01000001 (65, 'A')

Concepts and Some of Their Types in C++

- **Integral**
  - `int8_t, uint8_t, int16_t,…`
- **UnsignedIntegral**
  - `uint8_t, uint16_t, …`
- **SignedIntegral**
  - `int8_t, int16_t, …`

Support for Concepts

- Most programming languages do not yet have support for concepts
  - i.e. the compiler cannot check concepts the way it checks types
- There have been proposals to include support for concepts in C++
  - e.g. “Concepts Lite,” which checks some constraints on templates

Concepts vs. Other Abstraction Mechanisms

- Many languages have a way to specify interface of a type to be implemented later:
  - abstract classes in C++
  - interfaces in Java
  - etc.
- However, these mechanisms impose complete type specification on argument and return types of individual methods
- Concepts allow interfaces to specify any type that meets requirements

Concepts as Predicates

A concept can be viewed as a predicate that tests whether a type *satisfies* its requirements.

Requirements concern:

- Operations
- Semantics
- Time/space complexity

Complexity Requirement

- Why space/time complexity?
  Isn't this just an implementation detail?

- Imagine a stack implemented as an array,
  where each push required moving all elements.
  - Instead of being a constant time operation,
    push would be a linear time operation.
  - That violates programmer's assumption about how stacks behave.
  - *A stack without fast push and pop is not really a stack!*

Type Functions

A *type function* is a function that, given a type, returns an affiliated type.

It would be nice to have type functions like these:

- `value_type(Sequence)`
- `coefficient_type(Polynomial)`
- `ith_element_type(Tuple, size_t)`

Even though compilers have this information, mainstream programming languages do not provide type functions.

Type Attributes

A *type attribute* is a function that, given a type, returns a value representing one of its attributes.

Examples:

- `sizeof`
- `alignment_of`
- number of members in a struct
- name of the type

Some languages provide some type attributes.

The **Regular** Concept: Operations Requirements

A type is *regular* if it supports these operations:

- copy construction
- assignment
- equality
- destruction

Having a copy constructor implies having a default constructor, since

```
T a(b);
```

should be equivalent to:

```
T a; a = b;
```

The **Regular** Concept: Semantic Requirements

$$\forall a\ \forall b\ \forall c\ : T\ a(b) \implies (b = c \implies a = c)$$
$$\forall a\ \forall b\ \forall c\ : a \leftarrow b \implies (b = c \implies a = c)$$
$$\forall f \in \text{RegularFunction}\ : a = b \implies f(a) = f(b)$$

A *regular function* is one that produces equal results given equal inputs.

The **Regular** Concept: Space/Time Complexity Requirements

Each required operation must be no worse than linear in the *area* of the object.

Area: all space occupied by an object

- header and remote parts
- data and connectors

The **Semiregular** Concept

Semiregular is like **Regular**, except that equality is not explicitly defined.

- Needed in a few situations where it's difficult to implement an equality predicate.
- Even then, equality assumed to be implicitly defined, so copying and assignment still make sense.

Iterators

An *iterator* is a concept used to express position in a sequence.

- Iterators are a generalization of pointers.
- A better name for STL iterators would have been “position” or “coordinate.”
- An iterator is “something that lets you do linear search in linear time.”

The **Iterator** Concept

Required Operations:

- Regular type operations
- Successor
- Dereference

An iterator is "a theory with successor"

- Inspired by Peano's axioms
- But, does not require all of them
  - We can have loops, and don't always have successor

Dereferencing

A way to get from an iterator to its value

Time complexity requirement:

- Must be "fast," which means *there is not a faster way to get to the data than through the iterator*.

Dereferencing is a partial function (not always defined).

Relationship Between Successor and Dereference

Restrictions:

- Dereferencing is defined on an iterator if and only if successor is defined
- If you are not at the end of a range you can dereference

Why Regular?

- Why do we need iterators to be Regular rather than Semiregular?
  - i.e. why do we need equality as a requirement?

- Because we need to be able to tell when one iterator reaches another

Iterator Categories (1)

- Input Iterators
  - Support one-directional traversal, but only once
  - `i == j` does not imply `++i == ++j`
    (Once you've consumed a character from input stream, can't consume again with another iterator)
  - Canonical model: position in *input stream*
- Forward Iterators
  - Support one-directional traversal, repeatable
  - Canonical model: position in *singly-linked list*

Iterator Categories (2)

- Bidirectional Iterators
  - Support two-directional traversal, repeatable
  - Invertible successor function:
    if *x* has successor *y*, then *y* has a predecessor *x*
  - Canonical model: position in *doubly-linked list*
- Random-Access Iterators
  - Support access to any element in constant time (both *far* and *fast*)
  - Canonical model: position in *array*

Another Common Iterator Category

- Output Iterators
  - Support alternating successor (++) and dereference (*) operations
  - Result of dereferencing an output iterator can appear only on left side of assignment operator
  - Can't define equality, because can't get to the elements once they've been output
  - Canonical model: position in *output stream*

Other Possible Iterator Categories (not included in STL)

- Linked Iterators
  - Used in situations where the successor function is mutable
  - e.g. a linked list where link structure is modified
- Segmented Iterators
  - Used where data stored in noncontiguous segments, each containing contiguous sequences
  - e.g. B-tree

Why Not in STL?

- Just because a concept (such as Linked and Segmented iterators) is not in STL does not mean it's not useful.
- STL is a set of examples needed in many common situations, not an exhaustive catalog of all useful concepts, algorithms, and data structures.

Useful Function: Distance Between Two Iterators

```
template <InputIterator I>
DifferenceType<I>
distance(I f, I l, std::input_iterator_tag) {
    // precondition: valid_range(f, l)
    DifferenceType<I> n(0);
    while (f != l) {
        ++f;
        ++n;
    }
    return n;
}
```

f and l are meant to be mnemonic for “first” and “last.”

Variation Optimized for Random Access Iterators

```
template <RandomAccessIterator I>
DifferenceType<I>
distance(I f, I l,
         std::random_access_iterator_tag) {
    // precondition: valid_range(f, l)
    return l - f;
}
```

Difference Type

The *difference type* of an iterator is an integral type large enough to encode the largest possible range.

- Ideally, we'd like a type function to obtain the difference type.
- C++ does not have a general mechanism for type functions, but it has a special set of attributes for iterators: *iterator traits*.

Iterator Traits

- `value_type`
  - Returns type of values pointed to by iterator
- `Reference`
  - (Rarely needed with current architectures)
- `Pointer`
  - (Rarely needed with current architectures)
- `difference_type`
  - Returns difference type (see previous slide)
- `iterator_category`
  - Returns iterator category (see earlier slides)

Simplifying Iterator Trait Access

- Normal syntax for iterator trait is (e.g.)

```
std::iterator_traits<I>::difference_type
```

- To simplify, we'll write our own type function:

```
template <InputIterator I>
using DifferenceType =
    typename std::iterator_traits<I>::difference_type;
```

- This allows us to use the difference type function seen in the `distance` code:

```
DifferenceType<I>
```

Category Dispatch

- Have two implementations of the iterator distance function, one for input iterators and one for random access iterators
- Can write a single top-level function that calls the right implementation depending on type of the last argument (the iterator category)
- This is called *category dispatch*
- Happens at compile time – no performance penalty

Type Function for Iterator Category

```
template <InputIterator I>
using IteratorCategory =
      typename std::iterator_traits<I>::iterator_category;
```

Top-Level Distance Function Using Category Dispatch

```
template <InputIterator I>
DifferenceType<I> distance(I f, I l) {
    return distance(f, l,
                    IteratorCategory<I>());
}
```

- A client program that needed distance between iterators would use this 2-argment version
- Last argument in return is a constructor call; create an instance of the given type (since can't just pass types themselves)

Things to Consider

- One of your goals should be to identify existing concepts in your domain.
- As a programmer:
  - You will often develop new algorithms
  - You will occasionally develop a new data structure
  - You will rarely define a new concept

Lecture 18

Searching a Range

(Covers material from Sec. 10.6-10.9)

Ranges

- A *range* is a way of specifying a contiguous sequence of elements.
- Type of ranges:
  - Closed:
    Range $[i, j]$ includes i and j
  - Semi-Open:
    Range $[i, j)$ includes i but ends just before j

Ranges in Programming Interfaces

Semi-open ranges turn out to be best for programming interfaces

- Algorithms operating on sequences of n elements need to be able to refer to $n + 1$ positions
  - e.g. can insert new item before first element, between any two elements, or after last element
- Semi-open ranges can describe empty range
- Semi-open empty ranges still specify position
  - more information than “nil” or empty list

Specifying Ranges

- *Bounded* range: Two iterators
- *Counted* range: One iterator, one integer

This gives us a total of four ways to specify ranges:

	semi-open	closed
Bounded: two iterators	$[i, j)$	$[i, j]$
Counted: iterator and integer	$[i, n)$	$[i, n]$

A closed counted range must have $n > 0$.

Zero-based Indexing

- While mathematicians index sequences from 1, computer scientists start from 0
- Originally indicated memory offset in array
- Now useful regardless of implementation:
  For a sequence with n elements:
  - Indices are in the range [0, n)
  - Any iteration is bounded by the length

Iterator Distance, Revisited

```
template <InputIterator I>
DifferenceType<I>
distance(I f, I l, std::input_iterator_tag) {
    // precondition: valid_range(f, l)
    DifferenceType<I> n(0);
    while (f != l) {
        ++f;
        ++n;
    }
    return n;
}
```

Valid Ranges

- Can we write a `valid_range` function?
  - Returns true if the range specified by two iterators is valid; false otherwise
- No…
  - If iterators point to cells in a linked list, no way to know if there's a path from one to the other
  - Even if both simple pointers, no way to know that they both point to the same contiguous block of memory

Valid Range Precondition

Since we can't write a function, we'll just say:

```
// precondition: valid_range(f, l)
```

which means the range must satisfy the conditions needed for distance to work correctly, namely:

$$\text{container}(c) \implies \text{valid}(\text{begin}(c), \text{end}(c))$$

$$\text{valid}(x, y) \ \wedge \ x \neq y \implies \text{valid}(\text{successor}(x), y)$$

All STL-style containers and C++ arrays obey these axioms.

Moving Several Positions at Once: Input Iterators

```
template <InputIterator I>
void advance(I& x, DifferenceType<I> n,
             std::input_iterator_tag) {
    while (n) {
        --n;
        ++x;
    }
}
```

Moving Several Positions at Once: Random Access Iterators

```
template <RandomAccessIterator I>
void advance(I& x, DifferenceType<I> n,
             std::random_access_iterator_tag) {
    x += n;
}
```

Moving Several Positions at Once: Top Level

```
template <InputIterator I>
void advance(I& x, DifferenceType<I> n) {
    advance(x, n, IteratorCategory<I>());
}
```

Linear Search

- Simple idea: Scan a sequence until we find a particular element.
- Generalized idea: Scan a sequence until we find an element that satisfies a given predicate.
  - Find element x (i.e. element equal to x)
  - Find first odd element
  - Find first element with no vowels
  - etc.

Linear Search with Bounded Range

```
template <InputIterator I, Predicate P>
I find_if(I f, I l, P p) {
    while (f != l && !p(*f)) ++f;
    return f;
}
```

- "Find it if it's there"
- Caller must compare returned `f` and original `l` on return; if equal, no matching item found
- Required operations: equality, derefence, successor
- Semantic requirement: value type of iterator must be the same as argument type of predicate.

Linear Search with Counted Range

```
template <InputIterator I, Predicate P>
std::pair<I, DifferenceType<I>>
find_if_n(I f, DifferenceType<I> n, P p) {
    while (n && !p(*f)) { ++f; --n; }
    return {f, n};
}
```

- Even though we could overload function name, we add "_n" to emphasize use of counted range

Why Does It Return a Pair?

```
template <InputIterator I, Predicate P>
std::pair<I, DifferenceType<I>>
find_if_n(I f, DifferenceType<I> n, P p) {
    while (n && !p(*f)) { ++f; --n; }
    return {f, n};
}
```

- Caller doesn't have "last" iterator to compare to; needs second value to see if matching item found
- If caller does find a matching item and wants to find another, it can *restart where it left off*.
  - Without this, could only ever search for *first* matching item.

Binary Search

- For sorted ranges, can find matching item(s) faster with *binary search*.
- Binary search is easy to describe, but:
  - Hard to design interface correctly
  - Hard to implement correctly

Origins of Binary Search

Intermediate Value Theorem says:

> If f is a continuous function in an interval $[a, b]$ such that $f(a) < f(b)$, then $\forall u \in [f(a), f(b)]$ there is $c \in [a, b]$ such that $u = f(c)$.

Proof consists of doing binary search: Try a point in the middle, then one half the distance to the end, etc.

- Stevin had similar idea in 1594, but used tenths
- Lagrange used binary approach in 1795
- Bolzano and Cauchy came up with general form in early 19th century

Binary Search in Programming

- 1946: Discussed by John Mauchly (co-creator of ENIAC)
- 1960: First "correct" algorithm published by D.H. Lehmer
  - But interface was incorrect
- Versions with incorrect interfaces still around!

Incorrect Interface in UNIX

According to the POSIX standard:

> *The* `bsearch()` *function shall return a pointer to a matching member of the array, or a null pointer if no match is found. If two or more members compare equal, which member is returned is unspecified.*

Is this specification correct?

If not, what's wrong with it?

Two Flaws in bsearch

1. You often do a search because you want to *insert something at the place it would go* if it were already there.
   - With this interface, if item isn't there, you have to revert to linear search from the beginning to insert it.
2. You often want to *find multiple matching items* with the same key.
   - With this interface, you need to do linear search back and forth to find the ends of the matching sequence.

Correct Interface Uses Partition Point

If we have a sequence of items $[f, l)$ arranged so that a certain predicate is true of items in the range $[f, m)$ and false for $[m, l)$, then m is the *partition point*.

Precondition of partition point function is:

$$\exists m \in [f, l) \ : \ \Big(\forall i \in [f, m) \ : \ p(i)\Big) \ \wedge \ \Big(\forall i \in [m, l) \ : \ \neg p(i)\Big)$$

(i.e. elements are partitioned as described above)

Postcondition is that it returns m.

Partition Point for Counted Range

```
template <ForwardIterator I, Predicate P>
I partition_point_n(I f, DifferenceType<I> n, P p) {
    while (n) {
        I middle(f);
        DifferenceType<I> half(n >> 1);
        advance(middle, half);
        if (!p(*middle)) {
            n = half;
        } else {
            f = ++middle;
            n = n - (half + 1);
        }
    }
    return f;
}
```

Partition Point Example
Using Predicate isVowel(c)

```
0 1 2 3 4 5 6 7 8 9 0 1 2
e i e i o u a a a c b x g
f           m
```

n = 12; half = 6; m = f + half

isVowel('a')? Yes:
f = m + 1;
n = 12 – (6 + 1) = 5;
half = 2; m = f + half

```
e i e i o u a a a c b x g
            f   m
```

isVowel('c')? No:
n = 2;
half = 1; m = f + half

```
e i e i o u a a a c b x g
            f m
```

isVowel('a')? Yes:
f = m + 1;
n = 2 – (1 + 1) = 0

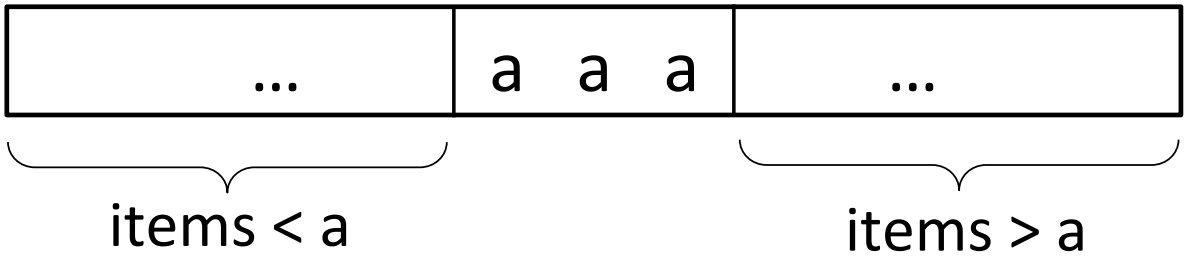
```
e i e i o u a a a c b x g
                f
```

Done!
f points to first non-vowel ('c')

Partition Point for Bounded Range

```
template <ForwardIterator I, Predicate P>
I partition_point(I f, I l, P p) {
  return partition_point_n(f, distance(f, l), p);
}
```

Binary Search Lemma



For any sorted range $[i, j)$ and a value a
(the item you're searching for), there are two iterators, lower bound b_l and upper bound b_u such that:

1. $\forall k \in [i, b_l) \quad : \; v_k < a$
2. $\forall k \in [b_l, b_u) \; : \; v_k = a$
3. $\forall k \in [b_u, j) \quad : \; v_k > a$

where v_k is the value at position k.

C++ Lambda Expressions

- When you want to pass a function as an argument, but you're only going to use it once, function object is overkill
- Use lambda expression: anonymous function object
- Declared like any function, except
  - Declaration usually occurs while calling another function
  - Use [=] instead of name
  - Return type usually inferred automatically
- Example: pass lambda expression that cubes its argument:

```
int main() {
  . . .
  int a = some_fn([=](int x) {return x * x * x; });
  . . .
```

Using Partition Point for Binary Search 1. Find *first* matching item

```
template <ForwardIterator I>
I lower_bound(I f, I l, ValueType<I> a) {
  return partition_point(f, l,
                         [=](ValueType<I> x) {
                                 return x < a; });
}
```

- This lambda expression:
  - returns true if its argument is less than *a*
  - passed as predicate used by `partition_point`
- Also, use `ValueType` type function to access iterator trait
  - Similar to what we did with `DifferenceType`

Using Partition Point for Binary Search
2. Find *last* matching item

```
template <ForwardIterator I>
I upper_bound(I f, I l, ValueType<I> a) {
  return partition_point(f, l,
                         [=](ValueType<I> x) {
                                 return x <= a; });
}
```

- Only difference is “<=“ instead of “<“ in lambda expression

Which One is the *Real* Binary Search?

It depends:

- `lower_bound` if you want the first matching item
- `upper_bound` if you want the last matching item
- `equal_range` if you want the whole range of matching items

If all you care about is *whether* there's a match, there is an STL function `binary_search`…

- …but all it's doing is calling one of the others and testing the result

Things to Consider

- Just as programs can have bugs, so can programming interfaces.
- An incorrect interface can severely limit the utility of a function.
- A correct interface can allow a function to be used in a variety of situations without loss of efficiency.

Lecture 19

Rotate

(Covers material from Sec. 11.1-11.3)

From Finding Data to Moving It

- Last time we looked at ranges and how to find data in a sequence (linear and binary search)
- This time we're going to look at some fundamental algorithms for moving data
- We start with the mathematical operations *permutation* and *transposition*

Permutation

A *permutation* is a function from a sequence of n objects onto itself.

Notation: $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 1 & 3 \end{pmatrix}$

Shorthand notation: $(2\ 4\ 1\ 3)$

Example: $(2\ 4\ 1\ 3) : \{a, b, c, d\} = \{c, a, d, b\}$

Permutations as a Group

The set of all permutations on n elements constitutes a group called the *symmetric group* S_n.

- Group operation: composition (associative)
- Inverse operation: inverse permutation
- Identity element: identity permutation

Cayley's Theorem

Every finite group is a subgroup of a symmetric group.

(For this reason, the symmetric group is one of the most important to know about.)

Transposition

A *transposition* (i, j) is a permutation that exchanges the ith and jth elements ($i \neq j$), leaving the rest in place.

Notation: $(2, 3) : \{a, b, c, d\} = \{a, c, b, d\}$

"swap the items at positions 2 and 3"

There is a simpler name for transposition in programming…

Swap

```
template <Semiregular T>
void swap(T& x, T& y) {
    T tmp(x);
    x = y;
    y = tmp;
}
```

- Swap requires the ability to copy-construct, assign, and destruct its data, but it does not test for equality
- So, arguments must be a **Semiregular** type

Transposition Lemma

Any permutation is a product of transpositions.

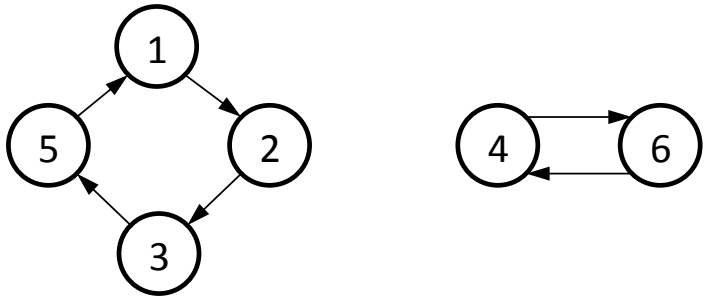
Proof:

- One transposition can put at least one element into its final destination.
- Therefore, at most $n - 1$ transpositions will put all n elements into their final destinations.

Why $n - 1$ and not n?

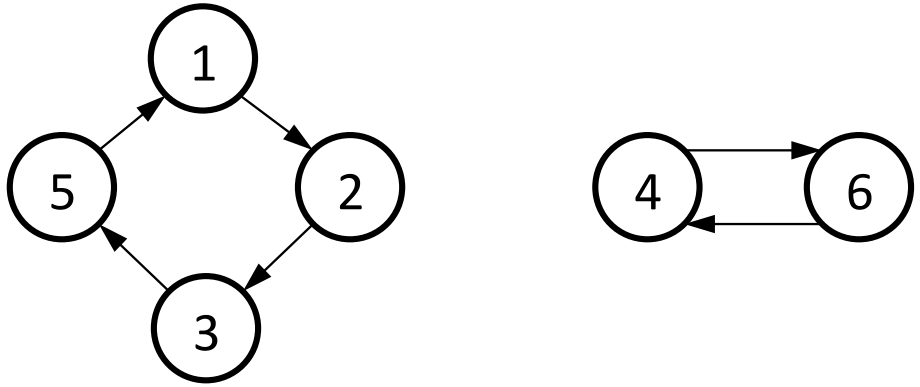
Permutations, Graphs, and Cycles

- Every permutation defines a directed graph of n elements.
- After applying a permutation enough times, an element will end up where it started, representing a cycle in the graph.
- Can decompose a permutation into cycles.



Decomposing Permutation into Cycles

- Consider permutation (2 3 5 6 1 4)
  - Element in position 4 moves to position 6; element in position 6 moves to position 4
  - So after applying twice, both elements in original position
  - Similar pattern for 1, 2, 3, 5, but takes four operations



- Notation: (2 3 5 6 1 4) = (1 2 3 5)(4 6)

Properties of Cycles

- Every permutation can be decomposed into cycles.
- A cycle containing one element is called a *trivial cycle*.
- Cycles are *disjoint*
  - If at a position in a cycle, can get to all the other positions in that cycle
  - Therefore if two cycles share one position, they share all positions (i.e. they are the same cycle)
  - So only way to have separate cycles is if they don't share any positions

Number of Assignments

The number of assignments needed to perform an arbitrary permutation in place is $n - u + v$

where

- n is the number of elements
- u is the number of trivial cycles
- v is the number of nontrivial cycles

Number of Assignments -- Proof

The number of assignments needed to perform an arbitrary permutation in place is $n - u + v$

Proof:

- Every nontrivial cycle of length k requires $k + 1$ assignments (1 per element, plus 1 to save first value)
- Since every nontrivial cycle requires one extra move, move all v cycles requires v extra moves
- Elements in trivial cycles don't need to be moved, and there are u of those
- So we need to move $n - u$ elements, plus v additional moves for the cycles.

The “Hardest” Permutation

- A common permutation that has exactly n/2 cycles is *reverse*
- It’s in some sense the hardest because it requires the most assignments

Swapping Ranges

- swap(x, y) is very useful, but sometimes want to swap more than one thing at a time
- Often want to swap all values in one range with corresponding values in another
  - Caveat: Ranges may overlap

```
while (condition) swap(*iter0++, *iter1++);
```

Swapping Ranges (when we know we won't run out)

```
template <ForwardIterator I0,ForwardIterator I1>
// ValueType<I0> == ValueType<I1>
I1 swap_ranges(I0 first0, I0 last0, I1 first1) {
    while (first0 != last0) swap(*first0++, *first1++);
    return first1;
}
```

- Why don't we require a `last1`?
  - No point in specifying end of 2nd range, because it must contain (at least) the same number of elements as first
- Why return `first1`?
  - If second range is longer, caller might want to know where the unmodified part of the second range begins

The Law of Useful Return, Restated

A procedure should return all the potentially useful information it computed.

Note:

- This does NOT mean doing extra computation
- This does NOT mean returning useless information

The Law of Separating Types

Do not assume that two types are the same when they may be different.

- Observe that we declared our function like this:

```
template <ForwardIterator I0,ForwardIterator I1>
// ValueType<I0> == ValueType<I1>
I1 swap_ranges(I0 first0, I0 last0, I1 first1)
```

- Not like this:

```
template <ForwardIterator I>
I swap_ranges(I first0, I last0, I first1)
```

Swapping Ranges (when we might run out)

```
template <ForwardIterator I0, ForwardIterator I1>
std::pair<I0, I1> swap_ranges(I0 first0, I0 last0,
                              I1 first1, I1 last1) {
    while (first0 != last0 && first1 != last1) {
        swap(*first0++, *first1++);
    }
    return {first0, first1};
}
```

- Why return a pair?
  - If one range is longer than the other, we won't necessarily have reached `last0` and `last1`.

Swapping Counted Ranges

```
template <ForwardIterator I0, ForwardIterator I1,
          Integer N>
pair<I0, I1> swap_ranges_n(I0 first0, I1 first1, N n) {
    while (n != N(0)) {
      swap(*first0++, *first1++);
      --n;
    }
    return {first0, first1};
}
```

- Instead of checking to see if we reached the end, we count down from n to 0.

The Law of Completeness

When designing an interface,
consider providing all the related procedures.

Note:

- This does NOT mean you should create a single interface for disparate operations

Rotation

A permutation of n elements by k where $k \geq 0$:

$$(k \bmod n,\ k+1 \bmod n,\ \ldots,\ k+n-2 \bmod n,\ k+n-1 \bmod n)$$

is called an *n by k rotation*.

- If you imagine all n elements laid out in a circle, the rotation shifts each one "clockwise" by k positions.
- Rotation is one of the most important algorithms you've probably never heard of.

Initial Rotation Ideas

- Naïve approach:
  - Implement with a modular shift.
  - Interface: beginning and end of range, and amount to shift
- Disadvantages:
  - Doing modular arithmetic on every operation is expensive
  - Misses the fact that rotation is equivalent to exchanging different length blocks

Better Rotation Interface: Exchanging Different Length Blocks

- Pass three iterators f, m, and l, where $[f, m)$ and $[m, l)$ are valid ranges.
  - f, m, l stand for first, middle, last
- Rotation interchanges ranges $[f, m)$ and $[m, l)$
- To rotate k positions in range $[f, l)$, pass in a value of m equal to $l - k$

Rotation Example

- Suppose we want to rotate $k = 5$ positions on a sequence specified by the range $[0, 7)$.
- Then $m = l - k = 7 - 5 = 2$:

```
0  1  2  3  4  5  6
f     m              l
```

- Which produces the result:

```
2  3  4  5  6  0  1
```

Gries-Mills Rotation Algorithm

```
template <ForwardIterator I>
void gries_mills_rotate(I f, I m, I l) {
if (f == m || m == l) return;
pair<I, I> p = swap_ranges(f, m, m, l);
  while(p.first != m || p.second != l) {
    if (p.first == m) {
      f = m; m = p.second;
    } else {
      f = p.first;
    }
    p = swap_ranges(f, m, m, l);
  }
  return;
}
```

Gries-Mills Example

```
0 1 2 3 4 5 6
f   m       l

2 3 0 1 4 5 6
    f   m   l

2 3 4 5 0 1 6
        f   m l

2 3 4 5 6 1 0
          f m l

2 3 4 5 6 0 1
          f m
            l
```

- Swap [0, 1] and [2, 3]
  Only first range is exhausted, so set `f = m` and `m = p.second`
- Swap [0, 1] and [4, 5]
  Only first range exhausted, so set `f = m` and `m = p.second`
- Swap [0] with [6]
  Only second range exhausted, so set `f = p.first`
- Swap [1] with [0]
  Both ranges, exhausted, so done.

Gries-Mills computes $\gcd(u, v)$ where u = length of $[f, m)$ and v = length of $[m, l)$

```
template <ForwardIterator I>
void gries_mills_rotate(I f, I m, I l) {
  // u = distance(f, m) && v = distance(m, l)
  if (f == m || m == l) return;      // u == 0 || v == 0
  pair<I, I> p = swap_ranges(f, m, m, l);
  while(p.first != m || p.second != l) {
    if (p.first == m) {              // u < v
      f = m; m = p.second;           // v = v - u
    } else {                         // v < u
      f = p.first;                   // u = u - v
    }
    p = swap_ranges(f, m, m, l);
  }
  return;                            // u == v
}
```

At the end, $u = v =$ GCD of lengths of initial two ranges.

Return Value of `rotate`

- Many applications benefit if `rotate` returns new middle: position where first element moved.
- Observe that

```
rotate(f, rotate(f, m, l), l)
```

  is an identity permutation.
- Challenge is to find a way to return desired value without doing any extra work.

Auxiliary Rotate

```
template <ForwardIterator I>
void rotate_unguarded(I f, I m, I l) {
    // assert(f != m && m != l)
    pair<I, I> p = swap_ranges(f, m, m, l);
    while (p.first != m || p.second != l) {
        f = p.first;
        if (m == f) m = p.second;
        p = swap_ranges(f, m, m, l);
    }
}
```

This is an “unguarded” version because it assumes the appropriate bounds checks have already been done.

Final Rotate for Forward Iterators

```
template <ForwardIterator I>
I rotate(I f, I m, I l, forward_iterator_tag) {
    if (f == m) return l;
    if (m == l) return f;
    pair<I, I> p = swap_ranges(f, m, m, l);
    while (p.first != m || p.second != l) {
        if (p.second == l) {
            rotate_unguarded(p.first, m, l);
            return p.first;
        }
        f = m;
        m = p.second;
        p = swap_ranges(f, m, m, l);
    }
    return m;
}
```

How Much Work Does Algorithm Do?

- Until last iteration of main loop, every swap puts 1 element in right place and moves another out of way.
- In final call to swap_ranges, both ranges are same length, so every swap puts both elements into their final positions – a free move for every swap.
- Total swaps = total elements n minus free swaps saved
  - Length of ranges is $\gcd(n-k, k) = \gcd(n, k)$ where $n = \text{distance}(f, l)$ and $k = \text{distance}(m, l)$.
  - So total swaps is $n - \gcd(n, k)$.
- Each swap takes 3 assignments, so number of assignments $= 3(n - \gcd(n, k))$

Things to Consider

- When you've designed an algorithm that solves a problem, don't stop when you've found the first solution…
- You may be able to:
  - Make it more efficient
  - Make it more general
  - Allow it to return additional information that would be useful to the caller

Lecture 20

Reverse

(Covers material from Sec. 11.4-11.8)

Cycles in Rotation: Example

Consider rotation of $k = 2$ for $n = 6$ elements:

0 1 2 3 4 5

$\Downarrow$

4 5 0 1 2 3

- Two nontrivial cycles
- Recall any permutation takes $n - u + v$ assignments, where n is number of elements, u is number of trivial cycles, v is number of nontrivial cycles
- So, we need $n - u + v = 6 - 0 + 2 = 8$ assignments
- Better than $3(n - \gcd(n, k)) = 3(6 - 2) = 12$

Cycles in Rotation: General Case

- Number of cycles is $\gcd(k, n)$
- So we can do any rotation in $n + \gcd(k, n)$ assignments instead of $3(n - \gcd(n, k))$
- In practice, $\gcd(n, k)$ is very small
  - It is 1 (i.e. only one cycle) about 60% of the time
- The catch:
  To exploit cycles, algorithm needs long jumps
  - Requires random access iterator

Helper Function for Rotate with Cycles

```
template <ForwardIterator I, Transformation F>
void rotate_cycle_from(I i, F from) {
    ValueType<I> tmp = *i;
    I start = i;
    for (I j = from(i); j != start; j = from(j)) {
        *i = *j;
        i = j;
    }
    *i = tmp;
}
```

- Function computes "in which position is the item that's going to move to position *x*?"
- Uses function object `from(i)` to compute "where element moving into position *i* comes from"

Function Object Passed to `rotate_cycle_from`

```
template <RandomAccessIterator I>
struct rotate_transform {
    DifferenceType<I> plus;
    DifferenceType<I> minus;
    I m1;

    rotate_transform(I f, I m, I l) :
        plus(m - f), minus(m - l), m1(f + (l - m)) {}
        // m1 divides items moving forward and backward

    I operator()(I i) const {
        return i + ((i < m1) ?  plus : minus);
    }
};
```

- Simulate rotation (including wraparound) by moving some items forward, some backward.

Cycle-Exploiting Rotate (for Random Access Iterators)

```
template <RandomAccessIterator I>
I rotate(I f, I m, I l,
         random_access_iterator_tag) {
    if (f == m) return l;
    if (m == l) return f;
    DifferenceType<I> cycles = gcd(m - f, l - m);
    rotate_transform<I> rotator(f, m, l);
    while (cycles-- > 0) {
        rotate_cycle_from(f + cycles, rotator);
    }
    return rotator.m1;
}
```

- Computes number of cycles (GCD) and constructs `rotate_transform` object
- Uses `rotate_cycle_from` to shift elements in each cycle

Cycle-Exploiting Rotate Example, Pt. 1
(rotation $k = 2$ for $n = 6$ elements)

- Take three iterators $f = 0$, $m = 4$, $l = 6$:

```
0  1  2  3  4  5
f           m     l
```

- Compute number of cycles:

$$\gcd(m - f, l - m) = \gcd(4, 2) = 2$$

- Construct rotator object, initializing state variables:

$$plus \leftarrow m - f = 4 - 0 = 4$$

$$minus \leftarrow m - l = 4 - 6 = -2$$

$$m1 \leftarrow f + (l - m) = 0 + (6 - 4) = 2$$

Cycle-Exploiting Rotate Example, Pt. 2:
what happens when rotate_cycle is called

- Pass $f + d = 0 + 2 = 2$ as first argument.
  - So inside function, $i = 2$.
- Save value at position 2 (which is also 2) to `tmp`.
- Set `start` to starting position of 2.
- Now loop as long as j is not equal to `start`:
  - Each time through loop, set $j \longleftarrow from(j)$
    - $from$ is the rotator function object
    - This adds stored values `plus` or `minus` to argument, depending on whether argument is less than $m1$.

Cycle-Exploiting Rotate Example, Pt. 3:
how array changes in `rotate_cycle_from` loop

$i \leftarrow 2, \quad j \leftarrow from(2) = 0$

```
0 1 2 3 4 5
j   i
```

$*i \leftarrow *j$

```
0 1 0 3 4 5
j   i
```

$i \leftarrow j = 0, \quad j \leftarrow from(0) = 4$

```
0 1 0 3 4 5
i     j
```

$*i \leftarrow *j$

```
4 1 0 3 4 5
i     j
```

$i \leftarrow j = 4, \quad j \leftarrow from(4) = 2$
which is `start`, so loop ends

```
4 1 0 3 4 5
    j   i
```

$*i \leftarrow tmp$

```
4 1 0 3 2 5
    j   i
```

When Is an Algorithm Faster in Practice?

- Fewer assignments does not always mean faster running time
- Ability to fit relevant data in cache is often much more significant
  - An algorithm that must make long jumps in memory may be slower than one that requires more assignments but has better *locality of reference*.

Reverse

- Informally:
  - Reverses order of elements in a sequence
- Formally:
  - Permutes a k-element list such that

    item 0 and item $k - 1$ are swapped
    item 1 and item $k - 2$ are swapped

    …

    item i and item $k - (i + 1)$ are swapped

Simple Rotate Using Reverse

```
template <BidirectionalIterator I>
void three_reverse_rotate(I f, I m, I l) {
    reverse(f, m);
    reverse(m, l);
    reverse(f, l);
}
```

- Requires bidirectional iterators
- Does not return new middle position

Example

Perform k = 2 rotation on sequence 0 1 2 3 4 5:

```
                  f           m   l
start             0 1 2 3 4 5
reverse(f, m)     3 2 1 0 4 5
reverse(m, l)     3 2 1 0 5 4
reverse(f, l)     4 5 0 1 2 3
```

Helper Function to Get New Middle

```
template <BidirectionalIterator I>
pair<I, I> reverse_until(I f, I m, I l) {
    while (f != m && m != l) swap(*f++, *--l);
    return {f, l};
}
```

- Reverses the elements until one of the iterators reaches the end
- The iterator that didn't hit the end will be pointing to the new middle

Rotate for Bidirectional Iterators

```
template <BidirectionalIterator I>
I rotate(I f, I m, I l, std::bidirectional_iterator_tag)
{
    reverse(f, m);
    reverse(m, l);
    pair<I, I> p = reverse_until(f, m, l);
    reverse(p.first, p.second);
    if (m == p.first) return p.second;
    return p.first;
}
```

- Like 3-reverse version, but splits 3rd reverse into two parts, saving new middle in between

Generic Rotate

```
template <ForwardIterator I>
I rotate(I f, I m, I l) {
    return rotate(f, m, l, IteratorCategory<I>());
}
```

- Works for any iterator type
- Use category dispatch
- Compiler chooses which of our three implementations to execute

Implementing Reverse for Bidirectional Iterators

```
template <BidirectionalIterator I>
void reverse(I f, I l, std::bidirectional_iterator_tag) {
    while (f != l && f != --l) swap(*f++, *l);
}
```

- Why do we need two comparisons each time?
  i.e. why not: `while (f != l) swap(*f++, *--l);`
  - Think about what happens if even number of items

- If we already knew how many times to execute loop (the *trip count*), we wouldn't need two comparisons…

Implementing Reverse for Bidirectional Iterators, using trip count n

```
template <BidirectionalIterator I, Integer N>
void reverse_n(I f, I l, N n) {
    n >>= 1;
    while (n-- > N(0)) {
        swap(*f++, *--l);
    }
}
```

- Now only $n/2$ comparisons

Implementing Reverse for Random Access Iterators

```
template <RandomAccessIterator I>
void reverse(I f, I l, std::random_access_iterator_tag) {
    reverse_n(f, l, l - f);
}
```

- Note how we precompute trip count, then call previous implementation

Helper Function when only have Forward Iterators

```
template <ForwardIterator I, BinaryInteger N>
I reverse_recursive(I f, N n) {
    if (n == 0) return f;
    if (n == 1) return ++f;
    N h = n >> 1;
    I m = reverse_recursive(f, h);
    if (odd(n)) ++m;
    I l = reverse_recursive(m, h);
    swap_ranges_n(f, m, h);
    return l;
}
```

- Keep partitioning range in half (h)
- Argument n keeps track of length of sequence being reversed

Implementing Reverse for Forward Iterators

```
template <ForwardIterator I>
void reverse(I f, I l, std::forward_iterator_tag) {
    reverse_recursive(f, distance(f, l));
}
```

Generic Reverse

```
template <ForwardIterator I>
void reverse(I f, I l) {
    reverse(f, l, IteratorCategory<I>());
}
```

The Law of Interface Refinement

Designing interfaces, like designing programs, is a multi-pass activity.

- We can't design an ideal interface until we have seen how the algorithm will be used
- Not all uses are immediately apparent
- Example: `std::rotate`
  - Originally returned `void`
  - 10 years to change C++ standard to return new middle

Computational Complexity

- Time complexity
  - Constant, logarithmic, quadratic, etc.
- Space complexity
  - In-place vs. not

Space complexity is often overlooked.

Space Complexity Definition

An algorithm is *in-place* (aka *polylog space*) if for an input of length n it uses $\mathrm{O}((\log n)^k)$ additional space, where k is a constant.

- In-place algorithms include divide-and-conquer techniques (e.g. quicksort) that use logarithmic extra space on stack.
- Algorithms that are not in-place usually make copy of their data.
- Can often speed up an algorithm a lot if can use extra (more than polylog) space.

Helper Function for Non-in-place Reverse

```
template <BidirectionalIterator I, OutputIterator O>
O reverse_copy(I f, I l, O result) {
    while (f != l) *result++ = *--l;
    return result;
}
```

- Copies all elements in reverse order, from back to front

Non-in-place Reverse

```
template <ForwardIterator I, Integer N,
BidirectionalIterator B>
I reverse_n_with_buffer(I f, N n, B buffer) {
    B buffer_end = copy_n(f, n, buffer);
    return reverse_copy(buffer, buffer_end, f);
}
```

- Takes only $2n$ assignments
  (instead of $3n$ for swap-based implementations)

Memory-Adaptive Algorithms

- In practice, dichotomy of in-place and not in-place algorithms is not very useful.
- While assumption of unlimited memory is not realistic, neither is assumption of only polylog extra memory.
- Usually 25%, 10%, 5%, or at least 1% of extra memory is available.
- Algorithms need to adapt to however much is available.

Memory-Adaptive Reverse: Strategy

- Pass a buffer to be used as temporary space
- If buffer is big enough for current data, use data-copying fast reverse
- If not, split sequence in half and recurse
  - Use outline of `reverse_recursive`
  - Since we only recurse with large chunks, overhead is acceptable

Memory-Adaptive Reverse

```
template <ForwardIterator I, Integer N,
          BidirectionalIterator B>
I reverse_n_adaptive(I f, N n, B buffer, N bufsize) {
    if (n == N(0)) return f;
    if (n == N(1)) return ++f;
    if (n <= bufsize) {
        return reverse_n_with_buffer(f, n, buffer);
    }
    N h = n >> 1;
    I m = reverse_n_adaptive(f, h, buffer, bufsize);
    advance(m, n & 1);
    I l = reverse_n_adaptive(m, h, buffer, bufsize);
    swap_ranges_n(f, m, h);
    return l;
}
```

A Sad Story of `get_temporary_buffer`

- `get_temporary_buffer` takes size n and returns largest buffer up to n that fits in physical memory.
- As placeholder, STL creator Stepanov wrote a simplistic and impractical implementation: call `malloc` initially with n, then halve request repeatedly until it returns valid pointer
- Years later, all major STL implementations still use this – but removed comment saying it needed to be replaced

Things to Consider

- Theoretical results (e.g. cycles in permutations) allow us to come up with more efficient algorithms.
- Practical observations (e.g. availability of space for buffer in a memory-adaptive algorithm) *also* allow us to come up with more efficient algorithms.
- Good programmers rely on knowledge of *both* theory and practice.

Lecture 21

A More Efficient GCD

(Covers material from Sec. 12.1-12.3)

Stein's Motivation

In 1961, Israeli Ph.D. student Josef Stein needed GCD computations for his research:

> "I wrote a program for the only available computer in Israel at that time — The WEIZAC at the Weizmann institute. Addition time was 57 microseconds, division took about 900 microseconds. Shift took less than addition. [Our group] had only 2 hours of computer time per week… I had the right conditions for finding that algorithm. ***Fast GCD meant survival.***"

Stein's Observations

first zero:	$\gcd(0, n) = n$
second zero:	$\gcd(n, 0) = n$
equal values:	$\gcd(n, n) = n$
even, even:	$\gcd(2n, 2m) = 2 \cdot \gcd(n, m)$
even, odd:	$\gcd(2n, 2m + 1) = \gcd(n, 2m + 1)$
odd, even:	$\gcd(2n + 1, 2m) = \gcd(2n + 1, m)$
small odd, big odd:	$\gcd(2n + 1, 2(n + k) + 1) = \gcd(2n + 1, k)$
big odd, small odd:	$\gcd(2(n + k) + 1, 2n + 1) = \gcd(2n + 1, k)$

Stein's GCD Algorithm (Part 1)

```
template <BinaryInteger N>
N stein_gcd(N m, N n) {
    if (m < N(0)) m = -m;
    if (n < N(0)) n = -n;
    if (m == N(0)) return n;
    if (n == N(0)) return m;

    // m > 0 && n > 0

    int d_m = 0;
    while (even(m)) { m >>= 1; ++d_m;}

    int d_n = 0;
    while (even(n)) { n >>= 1; ++d_n;}
```

Stein's GCD Algorithm (Part 2)

```
    // odd(m) && odd(n)

    while (m != n) {
        if (n > m) swap(n, m);
        m -= n;
        do m >>= 1; while (even(m));
    }

    // m == n

    return m << min(d_m, d_n);
}
```

Stein's Algorithm Example (Part 1)

To compute GCD(196, 42):

m	n	d_m	d_n
factor out 2s:			
196	42	0	0
98	42	1	0
49	42	2	0
49	21	2	1

Stein's Algorithm Example (Part 2)

m	n		d_m	d_n
main loop iteration:				
49	21		2	1
28	21	(by subtracting n from m)	2	1
14	21	(by shifting m)	2	1
7	21	(by shifting m)	2	1
21	7	(by swapping m and n)	2	1
14	7	(by subtracting n from m)	2	1
7	7	(by shifting m)	2	1
result:				

$$7 \times 2^{\min(2,1)} = 7 \times 2 = 14$$

Implications of Stein's Algorithm

- Stein's algorithm is faster because factoring out 2 is easy on a computer
- The notion of "factoring out 2" turns out to be more interesting when we try to generalize the algorithm beyond integers

Generalization of Euclid's Algorithm

- Positive integers: Greeks (5^{th} century BC)
- Polynomials: Stevin (ca. 1600)
- Gaussian integers: Gauss (ca. 1830)
- Algebraic integers: Dirichlet, Dedekind (ca. 1860)
- Generic version: Noether, van der Waerden (ca. 1930)

Can we do the same for Stein's algorithm?

Stein for Polynomials

x plays the role of 2; factor out powers of x

- $x^2 + x$ is "even"
- $x^2 + x + 1$ is "odd" (any zero-order coefficient)
- $x^2 + x$ "shifts" to $x + 1$

Stein's Observations Adapted for Polynomials

$$\gcd(p, 0) = \gcd(0, p) = p$$

$$\gcd(p, p) = p$$

$$\gcd(xp, xq) = x \cdot \gcd(p, q)$$

$$\gcd(xp, xq + c) = \gcd(p, xq + c)$$

$$\gcd(xp + c, xq) = \gcd(xp + c, q)$$

$$\deg(p) \geq \deg(q) \implies \gcd(xp + c, xq + d) = \gcd\left(p - \frac{c}{d}q, xq + d\right)$$

$$\deg(p) < \deg(q) \implies \gcd(xp + c, xq + d) = \gcd\left(xp + c, q - \frac{d}{c}p\right)$$

- Note how last 2 rules cancel one of zero-order coefficients, converting "odd, odd" case to "even, odd" case

$$\deg(p) \geq \deg(q) \implies \gcd(xp + c, xq + d) = \gcd\left(p - \frac{c}{d}q, xq + d\right)$$

Derivation

- If you have polynomials u and v, then $\gcd(u, v) = \gcd(u, av)$. We choose c/d as our coefficient a:

$$\gcd(xp + c, xq + d) = \gcd\left(xp + c, \frac{c}{d}(xq + d)\right)$$

- $\gcd(u, v) = \gcd(u, v - u)$, so can subtract 2nd argument from 1st:

$$\begin{aligned}\gcd(xp + c, xq + d) &= \gcd\left(xp + c - \frac{c}{d}(xq + d), xq + d\right) \\ &= \gcd\left(xp - \frac{c}{d}xq, xq + d\right)\end{aligned}$$

- If one GCD argument is divisible by x and other isn't, can drop x because resulting GCD will not contain that factor. So we can "shift" out the x, which gives our result.

Stein for Polynomials: Example

To compute $\gcd(x^3 - 3x - 2,\ x^2 - 4)$:

m	n	OPERATION
$x^3 - 3x - 2$	$x^2 - 4$	$m - (0.5x^2 - 2)$
$x^3 - 0.5x^2 - 3x$	$x^2 - 4$	$\text{shift}(m)$
$x^2 - 0.5x - 3$	$x^2 - 4$	$m - (0.75x^2 - 3)$
$0.25x^2 - 0.5x$	$x^2 - 4$	$\text{normalize}(m)$
$x^2 - 2x$	$x^2 - 4$	$\text{shift}(m)$
$x - 2$	$x^2 - 4$	$n - (2x - 4)$
$x - 2$	$x^2 - 2x$	$\text{shift}(n)$
$x - 2$	$x - 2$	$\text{GCD} : x - 2$

Further Extensions to Stein

- Gaussian integers: Weilert (2000)

  $1 + i$ plays the role of 2
- Eisenstein integers: Damgård and Frandsen (2003)

- In 2004, Agarwal and Frandsen proved that there are cases where Stein's algorithm works and Euclid's does not.
- What is the Stein domain? Still unsolved.

Generalizing Even and Odd

- Stein's algorithm relies on even and odd cases
- "Even" gets generalized to "divisible by a smallest prime"
  - Smallest prime is 2 for integers, x for polynomials, etc.

Back to Euclid's GCD

- Related to rings
- Can be extended to compute multiplicative inverse

Bézout's Identity

$$\forall a, b \; \exists x, y \; : \; xa + yb = \gcd(a, b)$$

For any two values a and b in the Euclidean domain, there are coefficients such that the linear combination gives the GCD of the original values.

Example:

- $a = 196$ and $b = 42$
- There are x and y such that $196x + 42y = \gcd(196, 42)$
- $\gcd(196, 42) = 14$, so $x = -1$ and $y = 5$

Claude Gaspar Bachet de Méziriac (1581-1638)

- Translated Diophantus' *Arithmetic* from Greek
- Wrote first book on recreational math, *Problèmes Plaisants*
- One of original members of the French Academy

Rings (Reminder)

- An algebraic structure that behaves similar to integers.
- Has both plus-like and times-like operations, but only additive inverse.

Ideals

An ideal *I* is a nonempty subset of a ring *R* such that:

$$1.\ \forall x, y \in I\ :\ x + y \in I$$

$$2.\ \forall x \in I,\ \forall a \in R\ :\ ax \in I$$

That is:

1. Ideals are closed under addition.
2. Ideals are closed under multiplication with *any* element of the ring (not necessarily an element of the ideal)

Examples of Ideals

- Even numbers (a nonempty subset of the ring of integers)
  - If you add two even numbers, you get an even number
  - If you multiply an even number by any integer (not necessarily even), you get an even number
- Univariate polynomials with root 5
- Polynomials with x and y and free coefficient 0 (e.g. $x^2 + 3y^2 + xy + x$)

Not All Subrings Are Ideals

Example:

- Integers are a subring of Gaussian integers, but not an ideal of Gaussian integers
- Multiplying an integer by imaginary number i does not produce an integer

Linear Combination Ideal

In a ring, for any two elements a and b,
the set of all elements $\{xa + yb\}$ forms an ideal.

Proof:

- First, the set is closed under addition:

$$(x_1a + y_1b) + (x_2a + y_2b) = (x_1 + x_2)a + (y_1 + y_2)b$$

- Next, it is closed under multiplication by any element:

$$z(xa + yb) = (zx)a + (zy)b$$

- Therefore it is an ideal.

Ideals in Euclidean Domains

Any ideal in a Euclidean domain is closed under the remainder operation and under Euclidean GCD.

Proof:

- Closed under remainder: By definition

$$\text{remainder}(a, b) = a - \text{quotient}(a, b) \cdot b$$

  If b is in ideal, then so is product with $\text{quotient}(a, b)$ and so is difference of that with a.

- Closed under GCD: Since GCD algorithm consists of repeatedly applying remainder, this follows from above.

Principal Ideals

An ideal I of the ring R is called a *principal ideal* if there is an element a in R, called the *principal element* of I, such that

$$x \in I \iff \exists y \in R \ : \ x = ay$$

- In other words, a principal ideal is an ideal that can be generated from one element

Examples of Principal Ideals

- Set of even numbers (2 is the principal element)
- Polynomials with root 5

But:

- Polynomials with x and and y and free coefficient 0 are ideals, but not principal ideals
  - There's no way to generate $x^2 + 3y^2 + xy + x$ starting with just x, or with just y.

Principal Ideal Domain

An integral domain is called a *principal ideal domain* (PID) if every ideal in it is a principal ideal.

- Recall that an integral domain is a ring with no zero divisors.

Example:

- Ring of integers is a PID
- Ring of multivariate polynomials over integers is not

ED $\Longrightarrow$ PID

Every Euclidean domain is a principal ideal domain.

Proof:

- Any ideal I in an ED contains an element m with a minimal positive norm. Consider element a in I.
- Either it is a multiple of m or it has remainder r:

$$a = qm + r \quad \text{where } 0 < \|r\| < \|m\|$$

- But we chose m as the smallest element, so it cannot have a smaller remainder. So a can't have a remainder:

$$a = qm$$

- So we can obtain every element from one element, which is what it means to be a PID.

Restating Bézout's Identity

- Since Bézout's identity says that there is at least one value of x and one value of y that satisfy $xa + yb = \gcd(a, b)\ldots$
- we can restate it as saying that the set of all possible linear combinations $xa + yb$ contains $\gcd(a, b)$

Bézout's Identity, Restated

A linear combination ideal $I = \{xa + yb\}$ *of a Euclidean domain contains* $\gcd(a, b)$.

Proof:

- Consider the linear combination ideal $I = \{xa + yb\}$.
- a is in I because $a = 1a + 0b$.
- Similarly, b is in I because $b = 0a + 1b$.
- We proved that every ideal in a Euclidean domain is closed under GCD.
- So $\gcd(a, b)$ is in I.

Invertibility Lemma

$$\forall a, n \in \mathbb{Z} \ : \ \gcd(a, n) = 1 \implies \exists x \in \mathbb{Z}_n \ : \ ax = xa = 1 \bmod n$$

Proof:

- By Bézout's identity:

$$\exists x, y \in \mathbb{Z} \ : \ xa + yn = gcd(a, n)$$

- So if $\gcd(a, n) = 1$, then $xa + yn = 1$.
- Therefore $xa = -yn + 1$, and

$$xa = 1 \bmod n$$

Things to Consider

- Every algorithm is based on some mathematical truth.
- Even a classical problem studied by great mathematicians may have a new solution.
- Real-world constraints are good for creativity.

Lecture 22

Extended GCD and Cryptology

(Covers material from Sec. 12.4-13.1)

Constructive vs. Nonconstructive Proofs

- Proof of Bézout's identity in last lecture proved that the coefficients x and y exist, but *doesn't tell us how to find them*.
- This is an example of a *nonconstructive* proof.
  - Mathematicians used to debate about whether these proofs were valid
  - Those who opposed them were known as *constructivists* or *intuitionists*
  - Today, nonconstructive proofs are routine

Henri Poincaré (1854–1912)

- Some considered him the greatest mathematician of his time
- Published more than 500 papers on many subjects
- Worked on establishing time zones
- The last great intuitionist; lost influence as formalist approach became dominant

Programmers Are Necessarily "Intuitionists"

- We need to have actual algorithms that compute things
- In the case of Bézout's Identity, we need an algorithm for finding x and y such that

$$xa + yb = \gcd(a, b)$$

Euclid's GCD Algorithm

```
template <EuclideanDomain E>
E gcd(E a, E b) {
    while (b != E(0)) {
        a = remainder(a, b);
        std::swap(a, b);
    }
    return a;
}
```

Remainders Computed in Euclid's Algorithm

- Each time through main loop, replace a by remainder of a and b, then swap a and b; final remainder is GCD
- Remainder sequence being computed:

$$
\begin{aligned}
r_1 &= \text{remainder}(a, b) \\
r_2 &= \text{remainder}(b, r_1) \\
r_3 &= \text{remainder}(r_1, r_2) \\
&\vdots \\
r_n &= \text{remainder}(r_{n-2}, r_{n-1})
\end{aligned}
$$

- Note how 2nd argument on iteration k becomes 1st argument on iteration $k + 1$

Remainders Computed in Euclid's Algorithm (alternate formulation)

- Since remainder of a and b is what's left over from a after dividing a by b, we can rewrite sequence as:

$$
\begin{aligned}
r_1 &= a - b \cdot q_1 \\
r_2 &= b - r_1 \cdot q_2 \\
r_3 &= r_1 - r_2 \cdot q_3 \\
&\vdots \\
r_n &= r_{n-2} - r_{n-1} \cdot q_n
\end{aligned}
$$

- q terms are corresponding quotients

Remainders Computed in Euclid's Algorithm (alternate formulation, rearranged)

- We can solve each equation for the first term on the right -- the first argument of the remainder function:

$$a = b \cdot q_1 + r_1$$
$$b = r_1 \cdot q_2 + r_2$$
$$r_1 = r_2 \cdot q_3 + r_3$$
$$\vdots$$
$$r_{n-2} = r_{n-1} \cdot q_n + r_n$$

- Earlier in course, we showed how last nonzero remainder r_n is equal to GCD of original arguments

Remainders Computed in Euclid's Algorithm (computing values for Bézout's Identity)

- We want to show that $r_n = \gcd(a, b) = xa + yb$.
- If we can write an equation in the sequence as this linear combination, we can write next one as well:

$$
\begin{aligned}
a &= 1 \cdot a + 0 \cdot b \\
b &= 0 \cdot a + 1 \cdot b \\
r_1 &= 1 \cdot a + (-q_1) \cdot b \\
r_2 &= b - r_1 q_2 \\
&= b - (a - q_1 b) q_2 \\
&= b - q_2 a + q_1 q_2 b \\
&= -q_2 a + (1 + q_1 q_2) b
\end{aligned}
$$

Recurrence in List of Remainders

- Assume we can already express two successive remainders as linear combinations:

$$r_i = x_i a + y_i b$$
$$r_{i+1} = x_{i+1} a + y_{i+1} b$$

- Substitute and rearrange to group a terms and b terms:

$$\begin{aligned} r_{i+2} &= r_i - r_{i+1} q_{i+2} \\ &= x_i a + y_i b - (x_{i+1} a + y_{i+1} b) q_{i+2} \\ &= x_i a + y_i b - x_{i+1} q_{i+2} a - y_{i+1} q_{i+2} b \\ &= x_i a - x_{i+1} q_{i+2} a + y_i b - y_{i+1} q_{i+2} b \\ &= (x_i - x_{i+1} q_{i+2}) a + (y_i - y_{i+1} q_{i+2}) b \end{aligned}$$

Observations About Recurrence

- Every entry in the sequence can be expressed as a linear combination of a and b
- Our procedure gives us coefficients x and y at each step:

$$x_{i+2} = x_i - x_{i+1}q_{i+2}$$

$$y_{i+2} = y_i - y_{i+1}q_{i+2}$$

- Coefficients on a depend only on variable x; coefficients on b depend only on variable y
- At the end, we have x and y such that

$$xa + yb = \gcd(a, b)$$

which is what we wanted.

Quick Method for Computing y

- We can rearrange $xa + yb = \gcd(a, b)$ as

$$y = \frac{\gcd(a, b) - ax}{b}$$

- So once we've gone through the recurrence to compute x, we can use this equation to immediately compute y.

Extended GCD:

returns x from Bezout's Identity, and GCD

```
template <EuclideanDomain E>
std::pair<E, E> extended_gcd(E a, E b) {
    E x0(1);
    E x1(0);
    while (b != E(0)) {
        // compute new r and x
        std::pair<E, E> qr = quotient_remainder(a, b);
        E x2 = x0 - qr.first * x1;
        // shift r and x
        x0 = x1;
        x1 = x2;
        a = b;
        b = qr.second;
    }
    return {x0, a};
}
```

Tracing `extended_gcd(196, 42)`

Initial values: a ⟵ 196, b ⟵ 42, x0 ⟵ 1, x1 ⟵ 0

- loop while b ≠ 0:
  - x2 ⟵ x0 – quotient(196, 42) * x1 = 1 – (4 * 0) = 1
  - x0 ⟵ x1 = 0; x1 ⟵ x2 = 1
  - a ⟵ b = 42; b ⟵ remainder(196, 42) = 28
- loop while b ≠ 0:
  - x2 ⟵ x0 – quotient(42, 28) * x1 = 0 – (1 * 1) = –1
  - x0 ⟵ x1 = 1; x1 ⟵ x2 = –1
  - a ⟵ b = 28; b ⟵ remainder(42, 28) = 14
- loop while b ≠ 0:
  - x2 ⟵ x0 – quotient(28, 14) * x1 = 1 – (2 * –1) = 3
  - x0 ⟵ x1 = –1; x1 ⟵ x2 = 3
  - a ⟵ b = 14; b ⟵ remainder(28, 14) = 0
- return {x0, a} = {–1, 14}

Extended GCD and Multiplicative Inverse

- Bézout's Identity says that

$$xa + yb = \gcd(a, b)$$

so

$$xa = \gcd(a, b) - yb$$

- If $\gcd(a, b) = 1$, then

$$xa = 1 - yb$$

- Thus multiplying x and a gives 1 plus a multiple of b, i.e.

$$xa = 1 \bmod b$$

- Two numbers with product 1 are multiplicative inverses, so if GCD is 1, then x (returned by extended GCD) is the multiplicative inverse of a mod b.

Applications of GCD

- Rational Arithmetic
- Symbolic Integration
- Rotation Algorithms
- Cryptography (using extended GCD for inverse)

Cryptology

- *Cryptology*: the science of secret communication.
- *Cryptography*: developing codes and ciphers.
- *Cryptanalysis*: breaking them.

Codes and Ciphers

Technically, two different things:

- Ciphers modify message at the level of its representation – letters or bits
- Codes use replacement of meaningful units (e.g. word or phrase) with code words
  - Example: using the names of animals instead of cities when discussing military plans

But informally we'll use terms interchangeably

Some Cryptographic History

- Evidence of cryptography in ancient societies such as Sparta and Persia
- Julius Caesar used “rotated” alphabet
- In 19th century, substitution ciphers appeared in stories by Poe and Conan Doyle
- Since 20th century, cryptography used extensively in war and diplomacy

Bletchley Park

- British cryptanalysis group during WWII
- Broke Nazi Enigma code, used to send messages to U-boats in the Atlantic
  - Alan Turing designed electromechanical machine called "bombe" to test combinations, based on earlier design by Marian Rejewski
- Broke Nazi Lorenz code, used by High Command
  - Tommy Flowers designed programmable vacuum-tube-based device called "Colossus" to test combinations.
  - May have been world's first programmable electronic digital computer

Cryptosystem

- Algorithms for encrypting and decrypting data
  - Plaintext: original data
  - Ciphertext: encrypted data
- *Keys* determine behavior of encryption and decryption algorithms

  ciphertext = encryption(key_0, plaintext)
  plaintext = decryption(key_1, ciphertext)
- System is *symmetric* if $key_0 = key_1$, otherwise *asymmetric*

Traditional Cryptosystems

- Symmetric and used secret keys
- Both sender and receiver must have key in advance of sending messages
- Problem: If key compromised and sender wants to switch to new one, s/he has to figure out how to secretly convey new key to the receiver.

Public-Key Cryptosystem

- An encryption scheme that uses a pair of keys:
  - Public key *pub* for encrypting
  - Private key *prv* for decrypting
- If Alice wants to send a message to Bob, she encrypts the message with Bob's public key; Bob decrypts it using his private key.

Requirements for Public-Key Cryptosystem

1. Encryption function must be one-way: easy to compute, with inverse that is hard to compute.
   - "Hard" means exponential time in size of key
2. Inverse function must be easy to compute when you have access to certain additional piece of information, known as *trapdoor*.
3. Both encryption and decryption algorithms are publicly known.

A function meeting first two requirements is called a *trapdoor one-way function*.

Who Invented Public-Key Cryptography?

- Clifford Cocks invented special case in 1973, but result was classified by British until 1997
  - Former NSA director Inman then claimed U.S. had done it in the 1960s…
- Hellman, Diffie, and Merkle initiated idea of public-key cryptography around 1976
- Rivest, Shamir, and Adleman actually came up with a trapdoor one-way function in 1977, now known as RSA algorithm

Things to Consider

- Tracing an algorithm can be instructive
  - By tracing Euclid's GCD algorithm, we discovered how to extend it to compute Extended GCD, with very few modifications.
- Most mathematical insights result from a similar process of exploring known results
  - Notebooks of great mathematicians like Gauss and Ramanujan are filled with seemingly random manipulations of existing results
  - Most of these scribblings are useless, but every so often they stumble across something profound, and recognize it

Lecture 23

A Real-World Application

(Covers material from Sec. 13.2-13.5)

Primality Testing

The problem of distinguishing prime numbers from composite… is known to be one of the most important and useful in arithmetic.

C.F. Gauss, *Disquisitiones Arithmeticae*

What Gauss Believed

1. Deciding whether a number is prime or composite is a very hard problem.
2. So is factoring a number.

Gauss was wrong about #1, but right about #2 (so far).

Useful Predicate: Is one number divisible by another?

```
template <Integer I>
bool divides(const I& i, const I& n) {
    return n % i == I(0);
}
```

- We can call this repeatedly to find the smallest divisor of a given number n.

Finding Smallest Divisor

```
template <Integer I>
I smallest_divisor(I n) {
    // precondition: n > 0
    if (even(n)) return I(2);
    for (I i(3); n >= i * i; i += I(2)) {
        if (divides(i, n)) return i;
    }
    return n;
}
```

- As with Sieve of Eratosthenes, we start with 3, advance by 2, and stop when the square of the current candidate reaches n.

Testing Primality

- Idea: If the smallest divisor of a number is the number itself, it is prime.

```
template <Integer I>
I is_prime(const I& n) {
    return n > I(1) && smallest_divisor(n) == n;
}
```

- Problem: complexity is $O(\sqrt{n}) = O(2^{(\log n)/2})$
- Exponential in *number of digits*

Different Approach: Start with Modular Multiplication

```
template <Integer I>
struct modulo_multiply {
    I modulus;
    modulo_multiply(const I& i) : modulus(i) {}

    I operator() (const I& n, const I& m) const {
        return (n * m) % modulus;
    }
};
```

Identity Element for Modular Multiplication

```
template <Integer I>
I identity_element(const modulo_multiply<I>&) {
    return I(1);
}
```

Multiplicative Inverse Modulo prime p

- Idea: As a consequence of Fermat's Little Theorem, we know inverse of integer a is a^{p-2}, where $0 < a < p$.
- So we'll raise a to that power, using our fast power function from earlier in the course:

```
template <Integer I>
I multiplicative_inverse_fermat(I a, I p) {
    // precondition: p is prime & a > 0
    return power_monoid(a, p - 2,
                        modulo_multiply<I>(p));
}
```

Idea of Fermat Test

- Fermat's Little Theorem says:

  If p is prime, then $a^{p-1} - 1$ divisible by p for any $0 < a < p$
  or, equivalently:

  If p is prime, then $a^{p-1} = 1 \bmod p$ for any $0 < a < p$

- We want to know if n is prime.
- We take an arbitrary *witness* $a < n$, raise it to $n - 1$ power ($\bmod n$), and check if the result is 1.
  - If result is not 1, we know n is not prime.
  - If result is 1, there's a good chance n is prime.
  - If we do this for lots of random witness, there's a *very* good chance n is prime.

Fermat Test

```
template <Integer I>
bool fermat_test(I n, I witness) {
  // precondition: 0 < witness < n
  I remainder(power_semigroup(witness,
                              n - I(1),
                              modulo_multiply<I>(n)));
  return remainder == I(1);
}
```

- Why did we use `power_semigroup` instead of `power_monoid`?
- Because we're not raising anything to the power 0.

Problem: Carmichael Numbers

- Pathological cases that fool the test
- Definition: A composite number $n > 1$ is a *Carmichael number* if and only if

$$\forall b > 1, \ \ \text{coprime}(b, n) \implies b^{n-1} = 1 \bmod n$$

- Example: 172081 is a Carmichael number. Its prime factorization is $7 \cdot 13 \cdot 31 \cdot 61$.

Ideas for Miller-Rabin Test (1)

- $n - 1$ is even

  (It would be silly to run a primality test on even n)
- So we can represent $n - 1$ as the product $2^k \cdot q$
- We'll test a sequence of squares

$$w^{2^0 q}, w^{2^1 q}, \ldots, w^{2^k q}$$

  where w is a random witness less than n

Ideas for Miller-Rabin Test (2)

- We'll rely on self-canceling law, restated as:

$$\text{For any } 0 < x < n \;\wedge\; \text{prime}(n),$$
$$x^2 = 1 \bmod n \implies x = 1 \vee x = -1$$

- Reminder: in modular arithmetic

$$-1 \bmod n = (n-1) \bmod n$$

- If we find some $x^2 = 1 \bmod n$ where x is neither 1 nor -1, then n is not prime

Observations for Miller-Rabin Test

1. If $x^2 = 1 \bmod n$,
   then there's no point in squaring x again,
   because the result won't change
   (if we reach 1, we're done)
2. If $x^2 = 1 \bmod n$ and x is not -1,
   then we know n is not prime
   (since we already ruled out $x = 1$)

Miller-Rabin Test

```
template <Integer I>
bool miller_rabin_test(I n, I q, I k, I w) {
    // precondition n > 1 && n - 1 = 2^kq && q is odd
    modulo_multiply<I> mmult(n);
    I x = power_semigroup(w, q, mmult);
    if (x == I(1) || x == n - I(1)) return true;
    for (I i(1); i < k; ++i) {
        // invariant x = w^{2^{i-1}q}
        x = mmult(x, x);
        if (x == n - I(1)) return true;
        if (x == I(1))     return false;
    }
    return false;
}
```

- Returns true if n probably prime, false if definitely not

Miller-Rabin Test Puzzle

- Why can we return true at the beginning if `power_semigroup` returns 1 or -1?
- Because:
  - Squaring result will give 1
  - Squaring is equivalent to multiplying exponent by 2
  - Doing this k times will make the exponent $n - 1$, the value we need for Fermat's Little Theorem
  - In other words:

$$\text{if } w^q \bmod n = 1 \text{ or } -1, \text{ then } w^{2^k q} \bmod n = w^{n-1} \bmod n = 1$$

Miller-Rabin Test Example:
Is $n = 2793$ prime?

- Choose random witness $w = 150$
  Factor $n - 1 = 2792$ into $2^3 \cdot 349$
  So $q = 349$ and $k = 3$
- Compute $x = w^q \bmod n = 150^{349} \bmod 2793 = 2019$
- Since result is neither 1 nor -1, start squaring x:

$$i = 1; \quad x^2 = 150^{2^1 \cdot 349} \bmod 2793 = 1374$$

$$i = 2; \quad x^2 = 150^{2^2 \cdot 349} \bmod 2793 = 2601$$

- Since we haven't reached 1 or -1 yet, and $i = k$ next time the loop starts, we can stop and return false; 2793 is not prime

Probabilistic Algorithm

- Like Fermat test, Miller-Rabin test is right most of the time
- Unlike Fermat test, Miller-Rabin test has provable guarantee:
  - It is right at least 75% of the time for a random witness w
- So randomly choosing 100 witnesses makes probability of error less than 1 in 2^{200}

AKS: A New Test for Primality

- Miller-Rabin is a probabilistic test, but it would also be nice to have a *deterministic* polynomial time algorithm for primality testing
- Such a test was developed in 2002 by Agrawal, Kayal, and Saxena.
- In practice, Miller-Rabin is much faster.

RSA Revisited

- First published public-key cryptosystem
- Relies on large prime numbers
  - And therefore, on primality testing
- Enabled many modern cryptography-based services
  - e.g. SecureID tokens for virtual private networks
- Basis of a major company
  - RSA Security, Inc., founded in 1982 by algorithm's inventors Rivest, Shamir, and Adleman
  - Now a division of EMC Corp., which bought it in 2006 for $2.1 billion
- Major international cryptography conference is RSA Conference

Uses of RSA Public-Key Cryptosystem

- Authentication: To prove that users, companies, websites, and other entities are who they claim to be
- Setting up secure channel: To exchange private keys, which are used for (faster) symmetric encryption

Communication Protocols Using RSA

- IPSec: security for low-level data transport
- PPTP: virtual private networks
- SET: secure electronic transactions (e.g. credit card)
- SSH: secure remote access to another computer
- SSL/TLS: secure data transfer layer
  - Used anytime you access a website with “https” address

Two Crucial Steps in RSA

1. Key Generation
   - Expensive
   - But only needs to be done very rarely
2. Encoding/Decoding
   - Relatively inexpensive
   - Needs to be done every time a message is sent or received

RSA Key Generation

Compute the following:

- Two random large primes, p_1 and p_2
- Their product $n = p_1 p_2$
- Euler function of their product:
$$\phi(p_1 p_2) = (p_1 - 1)(p_2 - 1)$$
- A random public key pub, coprime with $\phi(p_1 p_2)$
- A private key prv:
multiplicative inverse of pub modulo $\phi(p_1 p_2)$

Destroy p_1 and p_2; publish pub and n; keep prv secret

RSA Encoding/Decoding

- Choose message block size s bits so that $n > 2^s$
- Divide message into equal-sized blocks of length s, interpreted as large integers

- Encode:

```
power_semigroup(plaintext_block, pub,
                modulo_multiply<I>(n))
```

- Decode:

```
power_semigroup(ciphertext_block, prv,
                modulo_multiply<I>(n))
```

Same operation for encoding and decoding!

How RSA Works

- Encryption consists of raising message m to power pub
- Decryption consists of raising result to power prv
- Result of applying these two operations is the original message m (modulo n):

$$(m^{pub})^{prv} = m \bmod n$$

How RSA Works: Proof (1)

- We created prv to be multiplicative inverse of pub modulo n, where $n = \phi(p_1 p_2)$
- So by definition, product $pub \times prv$ is a multiple q of $\phi(p_1 p_2)$ with a remainder of 1:

$$\begin{aligned}
(m^{\text{pub}})^{\text{prv}} &= m^{\text{pub}\times\text{prv}} \\
&= m^{1+q\phi(p_1 p_2)} \\
&= m m^{q\phi(p_1 p_2)} \\
&= m(m^{\phi(p_1 p_2)})^q
\end{aligned}$$

How RSA Works: Proof (2)

- By Euler's theorem, $a^{\phi(n)} - 1$ is divisible by n, i.e. $a^{\phi(n)} = 1 + vn$. Substituting into previous equation:

$$(m^{pub})^{prv} = m(1 + vn)^q$$

- When we expand, every term on right will be a multiple of n except 1, so we can just say we have 1 plus some other multiple w of n:

$$\begin{aligned}(m^{pub})^{prv} &= m + wn \\ &= m \bmod n\end{aligned}$$

which is what we wanted.

Ensuring Coprimes

- For the step where we applied Euler's theorem to work, m must be coprime with $n = p_1 p_2$.
- Since message m could be anything, how do we know it will be coprime to $p_1 p_2$?
- In practice, probability is nearly indistinguishable from 1
- But to be sure:
  - Add a byte to end of m that isn't part of message.
  - Check if coprime; if not, add 1 to this extra byte

Why is RSA Secure?

- Factoring is hard, so computing ϕ is not feasible
- This will be true for foreseeable future
- BUT, if quantum computers become practical, RSA would no longer be secure

Final Project: RSA

1. Implement an RSA key generation library
2. Implement an RSA message encoder/ decoder that takes a string and the key as its arguments

RSA Project Hints

- If your language does not support arbitrary-precision integers, install a package that does.
- Remember that two numbers are coprime if their GCD is 1. You'll need this for a key generation step.
- You'll need two functions from earlier in the course:
  - `extended_gcd`
  - `multiplicative_inverse`

Using Extended GCD

- Recall that extended GCD returns a pair (x, y) such that $ax + ny = \gcd(a, n)$.
  - Use to check for coprimes
  - Also part of `multiplicative_inverse` implementation

Multiplicative Inverse

- Previous function `multiplicative_inverse_fermat` was only for primes
- Here's one that works for any n:

```
template <Integer I>
I multiplicative_inverse(I a, I n) {
    std::pair<I, I> p = extended_gcd(a, n);
    if (p.second != I(1)) return I(0);
    if (p.first < I(0)) return p.first + n;
    return p.first;
}
```

- Use this to get private key from public key

Things to Consider

- Results from the most theoretical, “useless” branch of mathematics – number theory – turned out to have enormous practical use.
- It is impossible to know which theoretical results will have practical applications.

Lecture 24

Summary

(Covers material from Chapter 14)

Generic Programming and Abstraction

Generic programming focuses on abstracting algorithms to their most general setting without losing efficiency.

Examples of Abstraction

- In Mathematics: Attempts to find the most general setting for Euclid's GCD algorithm led to development of abstract algebra.
- In Programming: We generalized an ancient algorithm for multiplying positive integers to a fast power function on semigroups.

The Essential Process of Generic Programming

1. Start with a specific, efficient solution.
2. Relax requirements as much as possible.

Don't Forget About Efficiency

Some techniques:

- Algebraic manipulation to reuse already computed values
- Strength reduction to avoid expensive operations
- Memory-adaptive algorithms
- Compile-time dispatch to choose fastest available implementation for given context

Often, making an algorithm more generic also makes it simpler and more efficient

Correctness of Programming Interfaces

Correct interfaces:

- enable a wider range of applications
- can improve efficiency (by returning relevant computations, avoiding duplicate effort)

Incorrect interfaces:

- cripple applications by limiting what they can do

Programming interfaces, like programs, need to be revised, debugged, and optimized.

Types and Concepts

- In mathematics, axioms in a theory are requirements that say what it means to be a certain kind of mathematical entity
- In programming, concepts are requirements on types; they say what it means to be a certain kind of computational entity

Which Concepts?

Choosing the right concepts for an algorithm or data structure is essential to good programming

- Choosing a concept with too many requirements unnecessarily limits how the algorithm can be used
- Choosing a concept with too few requirements makes it impossible to define algorithms that do anything useful

Adopting the Generic Programming Attitude

- Start with specific implementations, then revise and refine to be more efficient and more general
- Think about how pieces fit together and how to provide an interface that will still be useful in the future
- Choose concepts that provide just the right requirements on your data

Continue the Tradition

- The tradition of algorithmic thought dates back to ancient times
- You are inheriting the work of all who came before you, from Euclid to Noether to the present day
- By designing beautiful, general algorithms, you are adding your own small contribution to their work