Four Journeys

Alexander Stepanov

Four Journeys: Journey One

Lecture 1

Alexander Stepanov

Ἀεὶ ὁ θεὸς γεωμετρεῖ.

Πλάτων

ἀλλὰ πάντα μέτρῳ καὶ ἀριθμῷ καὶ σταθμῷ διέταξας.

Σοφία Σολομώντος

...impossibile est res huius mundi sciri nisi sciatur mathematica.

Rogerius Baco

Objectives

- To show deep connections between Programming and Mathematics.
- To recognize well-known mathematical abstractions in regular programming tasks.
- To write reusable, efficient algorithms defined on these abstractions.
- To use a small number of algorithms in a variety of applications.

Approach

The course traces four fundamental algorithmic ideas and shows how they led to discoveries in both computing and mathematics and how they can be applied to a variety of problems.

Historical Approach

- To understand something we need to know its history.

- “Great Men, taken up in any way, are profitable company.”

Thomas Carlyle

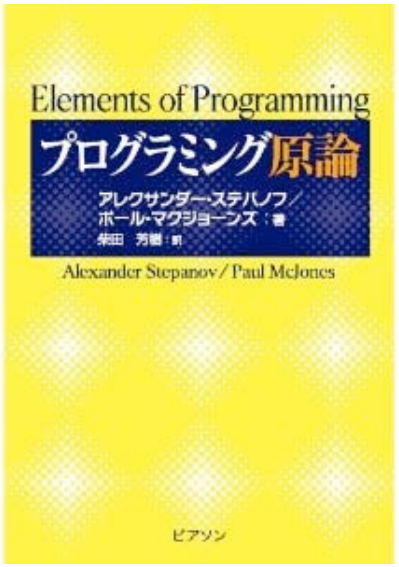
Programming Aesthetics

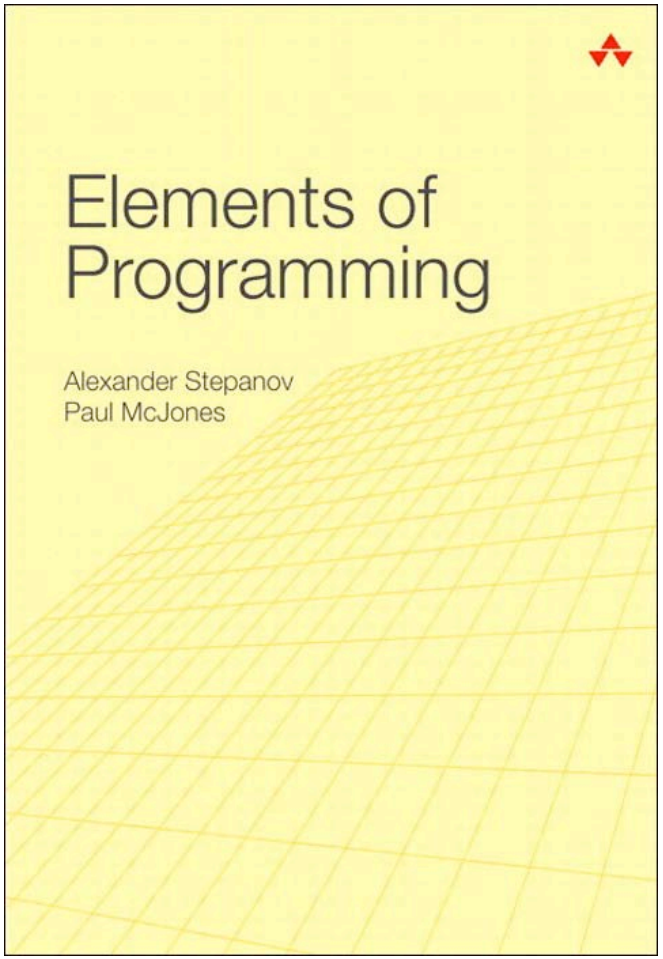
- Sense of beauty is important for building large systems and controlling complexity.
- Study of mathematics develops this sense.

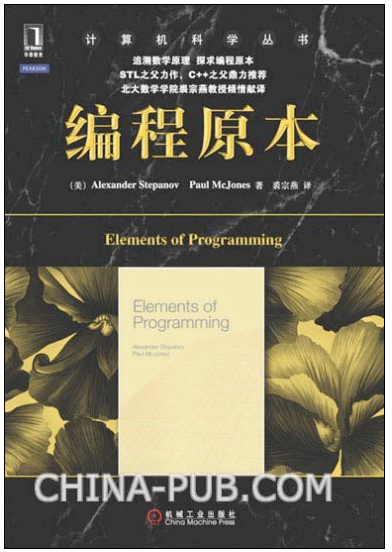
Lincoln and Jefferson

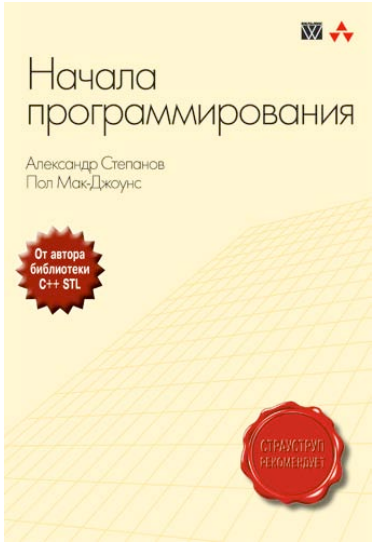
- Lincoln
  - "He studied and nearly mastered the Six-books of Euclid, since he was a member of Congress."
  - "…to improve his logic and language."
  - Use of Euclid in Lincoln-Douglas debates.
- Jefferson
  - "I have given up newspapers in exchange for Tacitus and Thucydides, for Newton and Euclid; and I find myself much the happier."
  - A thorough knowledge of Euclid should be the admission requirement to the University of Virginia.

The Auxiliary Text: *EoP*









Deductive vs. Historical Approach

- *EoP* presents facts.
- *Four Journeys* presents the intuition behind the facts.
- They complement each other.
  - It is not necessary to read EoP.
  - But it might be really helpful for some.

First Journey:
The Spoils of the Egyptians

How elementary properties of commutativity and associativity of addition and multiplication led to fundamental algorithmic and mathematical discoveries

Self-evident propositions

From time immemorial people knew that addition obeys:

$$a + b = b + a$$
$$(a + b) + c = a + (b + c)$$

and multiplication obeys:

$$ab = ba$$
$$(ab)c = a(bc)$$

Are proofs important?

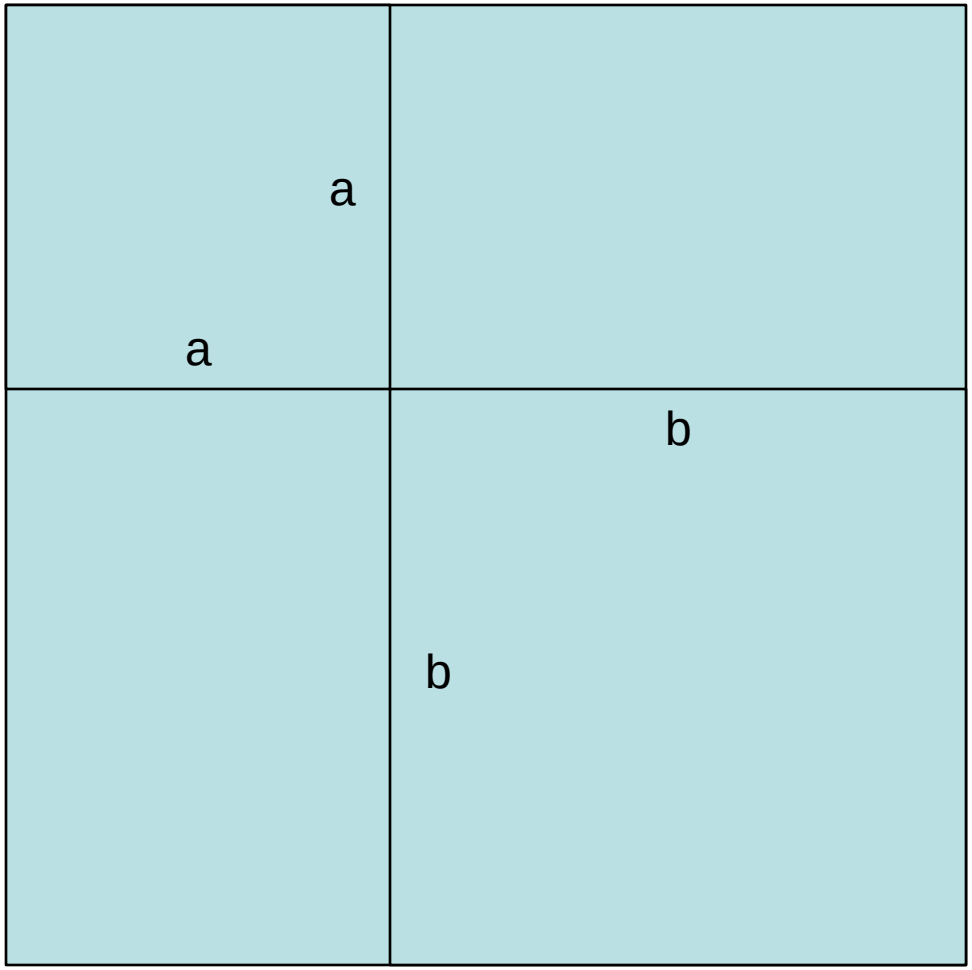
The first “rigorous” proof
of commutativity and associativity of plus and times
was published in 1861 by Hermann Grassmann (and ignored)
and rediscovered in 1888 by Richard Dedekind.

We will encounter the proof during our fourth journey.

Geometric Proofs

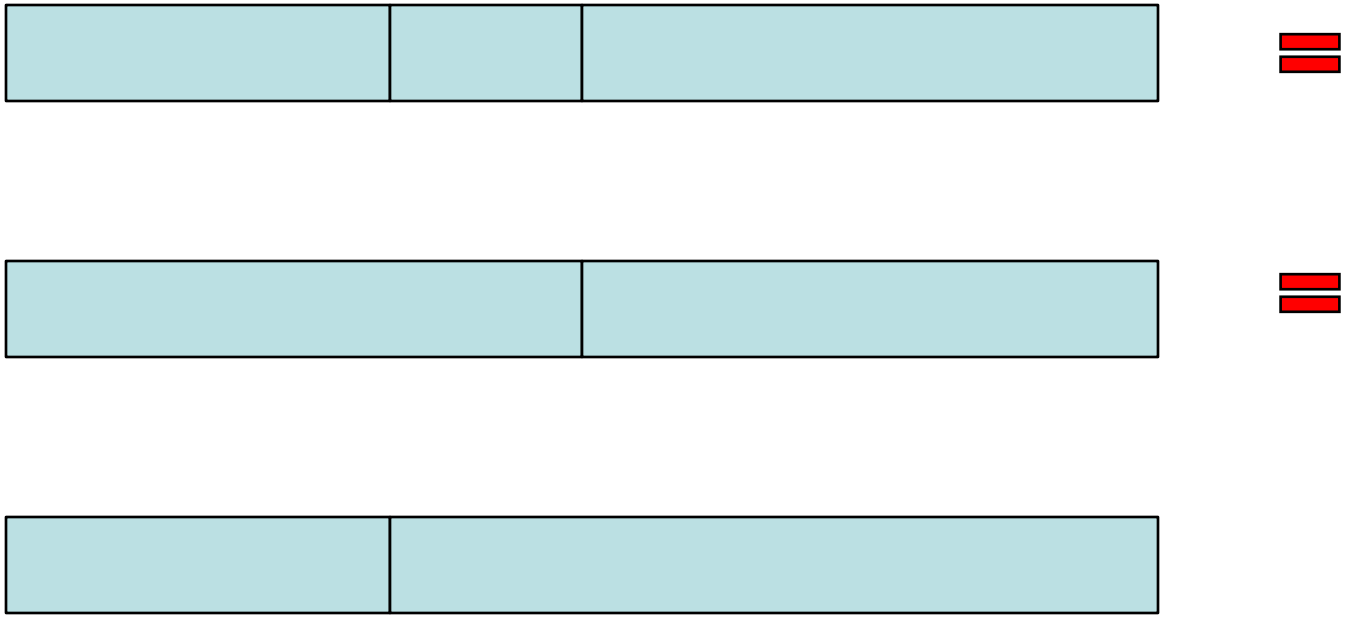
- For millennia mathematicians relied on geometric proofs
- They appeal to our innate intuition of space
  - Innate?

$$(a+b)^2 = a^2 + 2ab + b^2$$

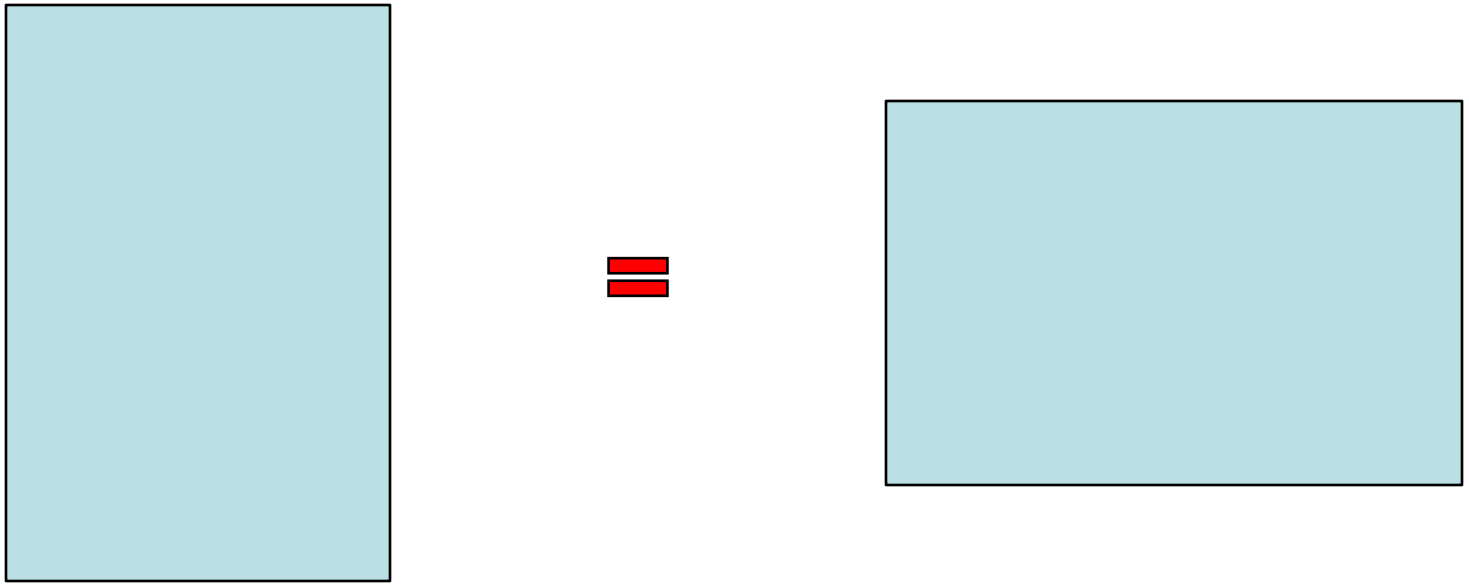


Commutativity of addition

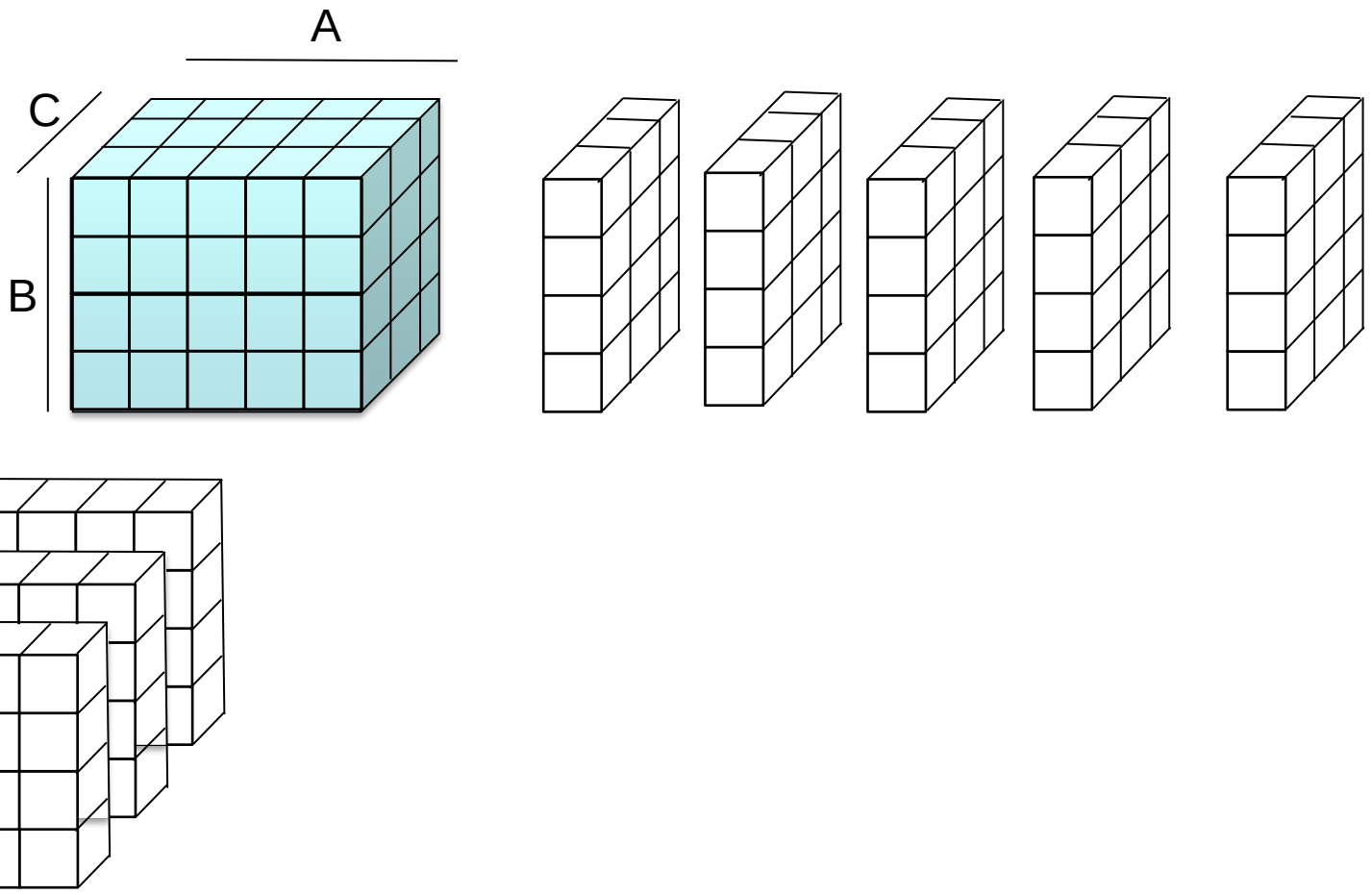
Associativity of addition



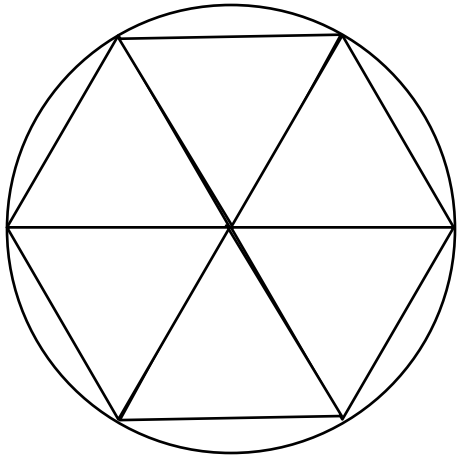
Commutativity of multiplication



Associativity of multiplication



$$\pi > 3$$



Problem 21
Design geometric proofs

$$(a-b)^2 = a^2 - 2ab + b^2$$

$$a^2 - b^2 = (a+b)(a-b)$$

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

$$(a-b)^3 = a^3 - 3a^2b + 3ab^2 - b^3$$

Problem 22

- Using geometry find formulas for
  - Sum of the first n positive integers
  - Sum of the first n positive odd integers
- Find geometric upper bound for π.

What is a proof?

A proof of a proposition is

- an argument
- accepted by the mathematical community
- to establish the proposition as valid

What is a valid proof today, might not be a valid proof three hundred years from now.

Reading 24

- Richard A. De Millo, Richard J. Lipton, Alan J. Perlis (1979), “Social processes and proofs of theorems and programs”, *Communications of the ACM*, Volume 22, Issue 5

Number systems

- All civilizations developed number systems
- Fundamental requirement for
  - Collecting taxes
  - Calendar computation to determine cultivation dates

Common Source?

- How could they all discover Pythogorean triples?
- Van der Waerden conjecture
  - Common Neolithic source (3000 BC)
  - Spreads through
    - Babylonia: Plimpton 322
    - China: Nine Chapters
      - (九章算術 *Jiǔzhāng Suànshù*)
    - India: Baudha-yana's Sulvasutra

Parallel Evolution

- Many similar mathematical concepts have been (re-)discovered independently
- That shows their fundamental nature

Geographical origins

- India and China had early mathematical traditions
  - Liu Hui (劉徽)
    - on inability to find volume of the sphere.
  - Aryabhata (आर्यभट)
- European mathematical tradition (which included Arabs) is the one from which Computer Science came

“Egyptians we take to be the most ancient of men”

- …the mathematical arts were founded in Egypt...

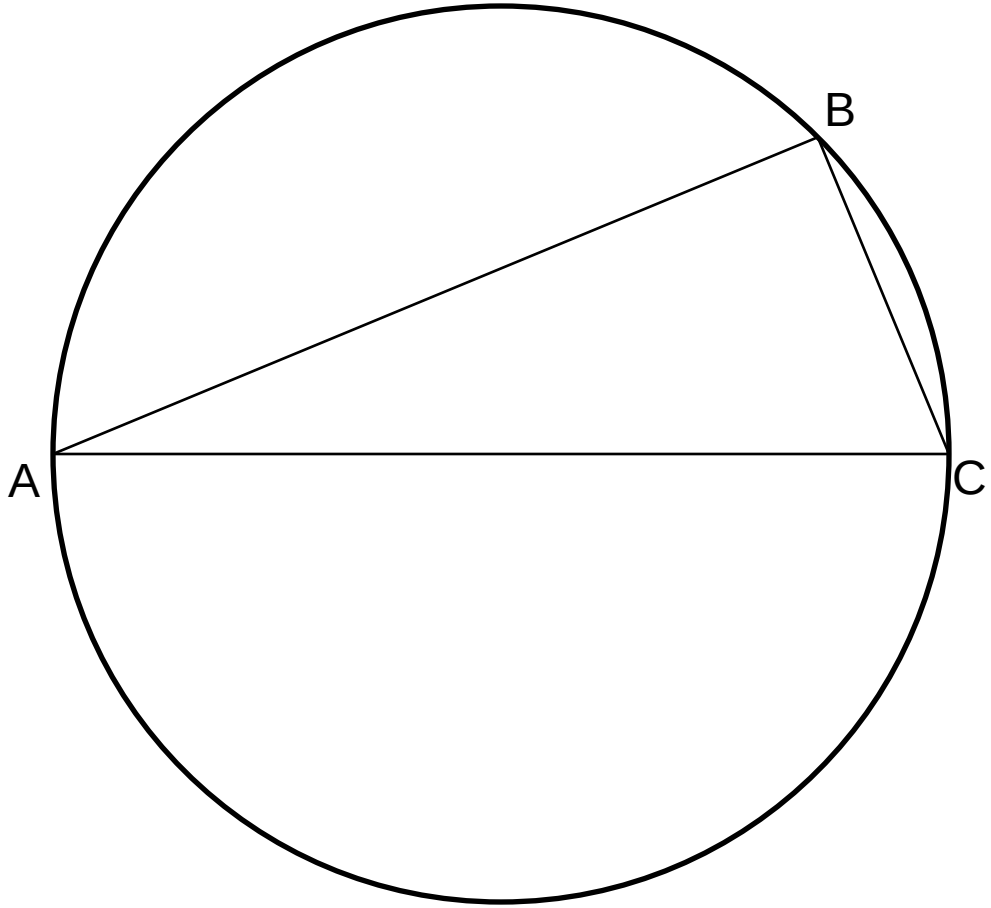
  Aristotle, *Metaphysics*, A 981b23-2

- [Moses] speedily learned arithmetic, and geometry... This knowledge he derived from the Egyptians, who study mathematics above all things...

  Philo, *De Vita Mosis*, I, 23-24

Thales of Miletus (635BC – 545BC)

Thales Theorem:
AC is diameter => ∠ABC = 90˚



The Proof of Thales Theorem

$\angle DAB = \angle DBA$

$\angle DCB = \angle DBC$

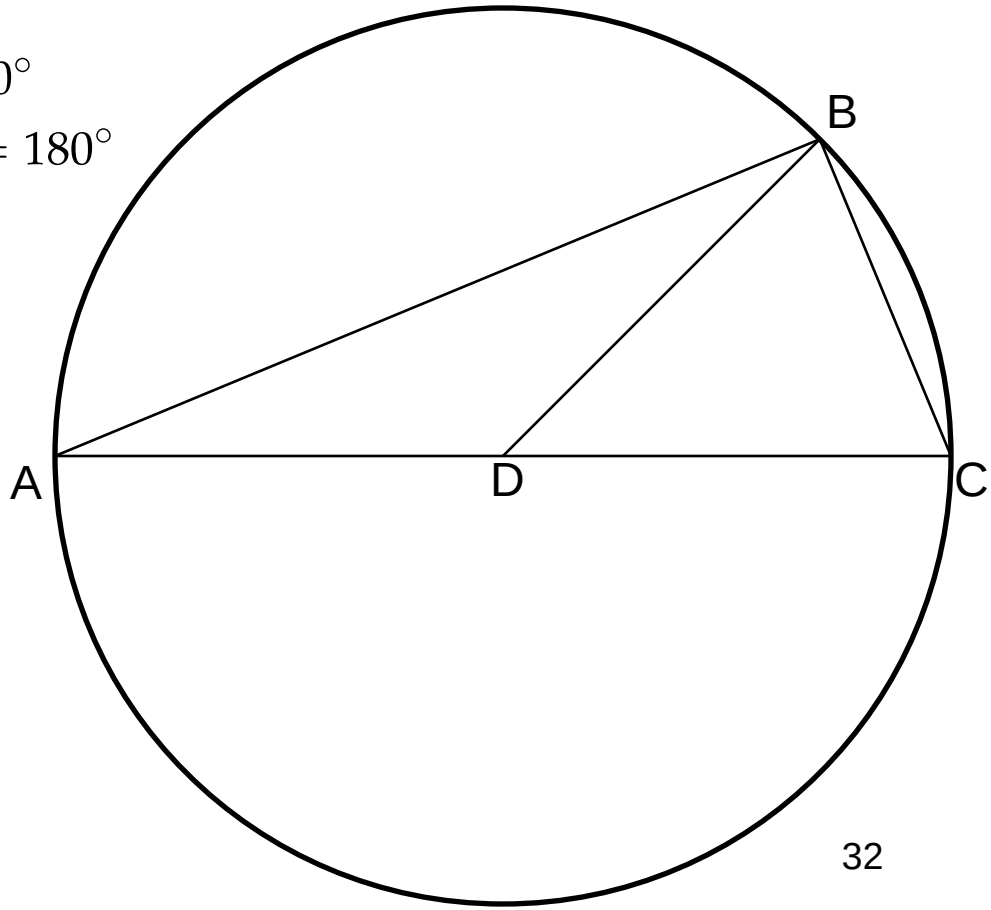
$\angle DAB + \angle DCB = \angle DBA + \angle DBC$

$\angle DAB + \angle DCB + \angle DBA + \angle DBC = 180^\circ$

$(\angle DBA + \angle DBC) + (\angle DBA + \angle DBC) = 180^\circ$

$\angle DBA + \angle DBC = 90^\circ$

$\angle CBA = 90^\circ$



Surviving Egyptian Mathematics

- Rhind papyrus
  - Written by the scribe Ahmes
  - 1650BC (copy of a much older text ≈1850BC)
  - Fast multiplication algorithm
    - Greeks knew it as *Egyptian multiplication*
    - Russian Peasant Algorithm
  - Fast division algorithm
    - We will meet it during the next journey

Distributivity

- Every multiplication algorithm depends on:

$$(n+m)a = na + ma$$

- It allows us to *reduce* the problem

The Inductive Base

$$1a = a$$

- Could we prove it?

Inductive Step

$$(n+1)a = na + a$$

- Depends on distributivity?

Slow Multiply Algorithm

```
int multiply0(int n, int a) {
  if (n == 1) return a;
  return multiply0(n – 1, a) + a;
}
```

- Both *a* and *n* are positive

The Algorithmic Insight

$$4a = ((a + a) + a) + a$$
$$= (a + a) + (a + a)$$

Ahmes Algorithm: 41 × 59

1	✓	59
2		118
4		236
8	✓	472
16		944
32	✓	1888

Halving n

$$even(n) \implies n = \frac{n}{2} + \frac{n}{2}$$

$$odd(n) \implies n = \frac{n-1}{2} + \frac{n-1}{2} + 1$$

Fast Multiply Algorithm

```
int multiply1(int n, int a) {
  if (n == 1) return a;
  int result = multiply1(half(n), a + a);
  if (odd(n)) result = result + a;
  return result;
}
```

requirement for half:

$$odd(n) \implies half(n) = half(n-1)$$

Problem 42

Implement an iterative version of fast multiply.

Number of additions

$$\#_+(n) = \lfloor \log n \rfloor + \nu(n) - 1$$

where $\nu(n)$ is the number of 1s in the binary representation of n
(the *population count* or *popcount*)

Is it optimal?

$$\#_+(15) = 3 + 4 - 1 = 6$$

multiply_by_15

```
int multiply_by_15(int a) {
   int b = (a + a) + a; // b == 3*a
   int c = b + b;       // c == 6*a
   return (c + c) + b;  // 12*a + 3*a
}
```

5 additions!

We discovered an optimal *addition chain* for 15.

Problem 46
Optimal Addition Chains

Find optimal addition chains for $n < 100$.

Further Reading 47

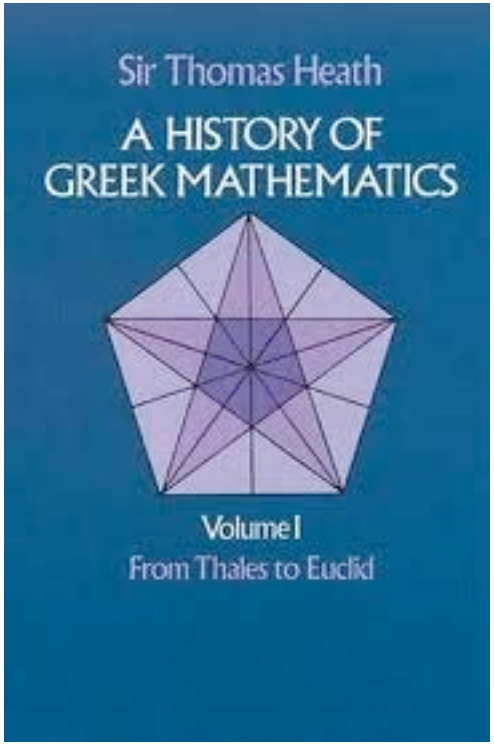
Donald Knuth, *The Art of Computer Programming*, Volume 2: Seminumerical Algorithms, pages 465-481.

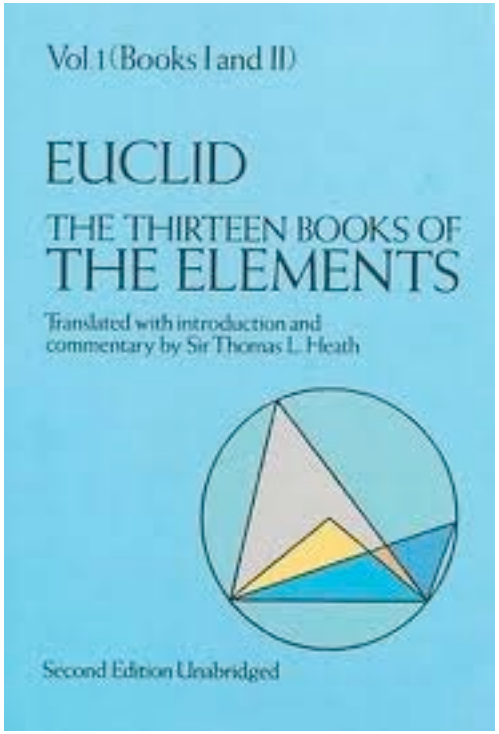
Four Journeys: Journey One

Lecture 2

Alexander Stepanov

Dover Treasures





Fast Multiply Algorithm

```
int multiply1(int n, int a) {
  if (n == 1) return a;
  int result = multiply1(half(n), a + a);
  if (odd(n)) result = result + a;
  return result;
}
```

requirement for half:

$$odd(n) \implies half(n) = half(n-1)$$

Program transformations

- The code of multiply1 is a good implementation of Egyptian Multiplication as far as the number of additions
- It also does $\lfloor \log n \rfloor$ recursive calls
  - function call is much more expensive than plus

multiply-accumulate

- It is often easier to do more than less

$$r + na$$

mult_acc0

```
int mult_acc0(int r, int n, int a) {
  if (n == 1) return r + a;
  if (odd(n)) {
    return mult_acc0(r + a, half(n), a + a);
  } else {
    return mult_acc0(r, half(n), a + a);
    }
}
```

Invariant: $r + na = r_0 + n_0 a_0$

mult_acc1

```
int mult_acc1(int r, int n, int a) {
  if (n == 1) return r + a;
  if (odd(n)) r = r + a;
  return mult_acc1(r, half(n), a + a);
}
```

- n is usually not 1.
- if n is even, it is not 1.
- We can reduce the number of comparisons with 1 by a factor of 2.

mult_acc2

```
int mult_acc2(int r, int n, int a) {
  if (odd(n)) {
    r = r + a;
    if (n == 1) return r;
  }
  return mult_acc2(r, half(n), a + a);
}
```

A *strict tail-recursive* procedure is one in which all the tail-recursive calls are done with the formal parameters of the procedure being the corresponding arguments.

mult_acc3

```
int mult_acc3(int r, int n, int a) {
  if (odd(n)) {
    r = r + a;
    if (n == 1) return r;
  }
  n = half(n);
  a = a + a;
  return mult_acc3(r, n, a);
}
```

Now it is easy to make it iterative.

mult_acc4

```
int mult_acc4(int r, int n, int a) {
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}
```

multiply2

```
int multiply2(int n, int a) {
  if (n == 1) return a;
  return mult_acc4(a, n - 1, a);
}
```

When n is 16, it does 7 addition instead of 4.

We do not want to subtract 1 from an even number.

Let us make it odd!

multiply3

```
int multiply3(int n, int a) {
  while (!odd(n)) {
    a = a + a;
    n = half(n);
  }
  if (n == 1) return a;
  return mult_acc4(a, n - 1, a);
}
```

mult_acc4 does an unnecessary test for 1.

multiply4

```
int multiply4(int n, int a) {
  while (!odd(n)) {
    a = a + a;
    n = half(n);
  }
  if (n == 1) return a;
  // even(n - 1) => n - 1 != 1
  return mult_acc4(a, half(n), a + a);
}
```

Coloring mult_acc4

```
int mult_acc4(int r, int n, int a) {
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}
```

Syntactically generalized multiply_accumulate

```
template <typename A, typename N>
A multiply_accumulate(A r, N n, A a) {
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}
```

Syntactic Requirements on *A*

- a + b
  - operator+
- can be passed by value
  - copy constructor
- assignment
  - operator=

A semantic requirement on A: Associativity of +

$$A(T) \implies \forall a, b, c \in T : a + (b + c) = (a + b) + c$$

In computer + may be partial

- We often deal with operations that are partially defined.
- We require axioms to hold inside the *domain of definition*.
- Much research work is needed to extend axioms to computer models of real numbers.
  - *approximate associativity*

Implicit syntactic requirements induced by intended semantics

- Equality
  - operator==
- Inequality
  - operator!=

Equational Reasoning

- Equality on type T should satisfy our expectations.
  - inequality is negation of equality
  - equality is reflexive, symmetric and transitive

$$a = a$$

$$a = b \implies b = a$$

$$a = b \wedge b = c \implies a = c$$

  - Substitutability

$$\forall \mathbb{P}, a, b : a = b \implies \mathbb{P}(a) = \mathbb{P}(b)$$

Regular types:

Connection between construction, assignment and equality:

- T a = b; assert(a == b);
- a = b; assert(a == b);
- T a = b; <=> T a; a = b;
- “normal” destructor

Regularity is the default

All the types in this course will be regular unless stated otherwise.

Problem 70

It seems that all of the C/C++ built-in types are regular. Unfortunately, one of the most important logical axioms is sometimes violated: the law of excluded middle. Sometimes the following is false:

$$a = b \vee a \neq b$$

Find the case; analyze it and propose a solution.

Problem 71

There is a sizable body of work on constructive real numbers: the real numbers that can be computed with arbitrary precision. Apparently, the classes that implement them are not regular. Why?

If you are interested to learn more

- Hans-Juergen Boehm: Constructive real interpretation of numerical programs. PLDI 1987: 214-221
- or use his calculator
  - http://www.hpl.hp.com/personal/Hans_Boehm/crcalc/

Requirements on **A**

- Regular type
- Associative +

- **A** is an *additive semigroup*:
  - *semigroup*: a set with an associative binary operation
  - *additive*: binary operation is +
    - but by convention, + commutes

Rules for overloading +

- If a set has one binary operation and it is associative and commutative, call it $+$.
- If a set has one binary operation and it is associative and not commutative, call it $*$.
- Kleene uses ab to denote string concatenation (in mathematics $*$ is elided).

Overloading Dilemma

- We can drop commutativity requirement
  - Allows people to use multiply with strings
    - The algorithm is used by Hans Boehm in his implementation of ropes
      - http://www.sgi.com/tech/stl/Rope.html
- We can try to enforce an established mathematical terminology

Language as a tool of thought

1. If there is an established term, use it.

2. Do not use an established term inconsistently with its accepted meaning.

Problem 76

What would be a better name for std::vector?

Solution: Explicit Weakening

- We do not want to change the meaning of an established mathematical concept *Additive Semigroup.*
- We want our algorithm to work with strings.
- We weaken *Additive Semigroup* into *Noncommutative Additive Semigroup.*
- We require commutativity of + unless otherwise specified.

Examples of Additive Semigroups

- Positive even numbers
- Negative integers
- Planar vectors
- Boolean functions
  - what is +?
- Line segments
  - what is +?

Non-requirement is not a requirement

- If an algorithm requires Noncommutative Semigroup, it will work on Commutative Semigroup.
- Noncommutative only means that it is not *required* to be Commutative.

Syntactically generalized multiply_accumulate

```
template <NonCommutativeAdditiveSemigroup A,
          typename N>
A multiply_accumulate(A r, N n, A a) {
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}
```

Syntactic Requirements on *N*

- Regular type with
  - half
  - odd
  - == 1
  - copy constructor
  - assignment

Semantic Requirements on N

even(n) => half(n) + half(n) == n

odd(n) => even(n – 1)

odd(n) => half(n – 1) == half(n)

axiom:

n == 1 || half(n) == 1 ||

half(half(n)) == 1 || ...

What kind of a mathematical concept is N?

Disjunctive Requirements

- Sometimes we define requirements not through axioms but by defining a set of *intended models* – types on which we want our algorithm to work.
- N = {uint8_t, int8_t, ..., uint64_t, int64_t, ...}
- We will call N *Integer.*
- At the end of the course we will be able to define it axiomatically.

Auxiliary procedures

```
// poor man's concepts:
#define Integer typename

template <Integer N>
N half(N n) { return n >> 1; }

template <Integer N>
bool odd(N n) { return n & 1; }
```

multiply_accumulate for Additive Semigroup

```
template
<NoncommutativeAdditiveSemigroup A, Integer N>
A multiply_accumulate_semigroup(A r, N n, A a) {
  precondition(n >= 0);
  if (n == 0) return r;
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}
```

multiply
for Additive Semigroup

```
template
<NoncommutativeAdditiveSemigroup A, Integer N>
A multiply_semigroup(N n, A a) {
  precondition(n > 0);
  while (!odd(n)) {
    a = a + a;
    n = half(n);
  }
  if (n == 1) return a;
  return multiply_accumulate_semigroup(a, half(n),
                                       a + a);
}
```

Why n > 0 not n >= 0?

$$a = 1a = (1 + 0)a = 1a + 0a = a + 0a$$

- $0a$ is the right additive identity

$$a = 1a = (0 + 1)a = 0a + 1a = 0a + a$$

- $0a$ is the left additive identity
- $0a$ is the additive identity, a zero
- A semigroup might not have an identity

Monoids

- *Monoid* is a semigroup that contains an identity element *id*:

$$a \circ id = id \circ a = a$$

- *Additive monoid* is an additive semigroup where the identity element is *0*:

$$a + 0 = 0 + a = a$$

Monoid multiply

```
template <NoncommutativeAdditiveMonoid A, Integer N>
A multiply_monoid(N n, A a) {
  precondition(n >= 0);
  if (n == 0) return A(0);
  return multiply_semigroup(n, a);
}
```

Groups

- *Group* is a monoid with inverse operation

$$a \circ inv(a) = inv(a) \circ a = id$$

- *Additive group* is an additive monoid with unary – (minus) such that

$$a + -a = -a + a = 0$$

“A new star of unimaginable brightness in the heavens of pure mathematics”

Group multiply

```
template <AdditiveGroup A, SignedInteger N>
A multiply_group(N n, A a) {
  if (n < 0) {
    n = -n;
    a = -a;
  }
  return multiply_monoid(n, a);
}
```

From multiply to power

If we replace + with * we compute a^n instead of na.

- replace doubling with squaring

power_accumulate for Multiplicative Semigroup

```
template <MultiplicativeSemigroup A, Integer N>
A power_accumulate_semigroup(A r, A a, N n) {
  precondition(n >= 0);
  if (n == 0) return r;
  while (true) {
    if (odd(n)) {
      r = r * a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a * a;
  }
}
```

we compute ra^n

power
for Multiplicative Semigroup

```
template <MultiplicativeSemigroup A, Integer N>
A power_semigroup(A a, N n) {
  precondition(n > 0);
  while (!odd(n)) {
    a = a * a;
    n = half(n);
  }
  if (n == 1) return a;
  return power_accumulate_semigroup(a, a * a,
                                    half(n));
}
```

power
for Multiplicative Monoid

```
template <MultiplicativeMonoid A, Integer N>
A power_monoid(A a, N n) {
  precondition(n >= 0);
  if (n == 0) return A(1);
  return power_semigroup(a, n);
}
```

power
for Multiplicative Group

```
template <MultiplicativeGroup A>
A multiplicative_inverse(A a) {
  return A(1) / a;
}

template <MultiplicativeGroup A, Integer N>
A power_group(A a, N n) {
  if (n < 0) {
    n = -n;
    a = multiplicative_inverse(a);
  }
  return power_monoid(a, n);
}
```

Multiple semigroups on T

- There could be multiple associative operations on the same type T
  - additive
  - multiplicative
  - …
- We often need to pass an operation to an algorithm

accumulate for an arbitrary semigroup

```
template <Regular A, Integer N, SemigroupOperation Op>
requires (Domain<Op, A>) // parens will go away
A power_accumulate_semigroup(A r, A a, N n, Op op) {
  precondition(n >= 0);
  if (n == 0) return r;
  while (true) {
    if (odd(n)) {
      r = op(r, a);
      if (n == 1) return r;
    }
    n = half(n);
    a = op(a, a);
  }
}
```

power for an arbitrary semigroup

```
template <Regular A, Integer N, SemigroupOperation Op>
requires(Domain<Op, A>)
A power_semigroup(A a, N n, Op op) {
  precondition(n > 0);
  while (!odd(n)) {
    a = op(a, a);
    n = half(n);
  }
  if (n == 1) return a;
  return power_accumulate_semigroup(a, op(a, a),
                                    half(n), op);
}
```

power for an arbitrary monoid

```
template <Regular A, Integer N, MonoidOperation Op>
requires(Domain<Op, A>)
A power_monoid(A a, N n, Op op) {
  precondition(n >= 0);
  if (n == 0) return identity_element(op);
  return power_semigroup(a, n, op);
}
```

identity_element examples

```
template <NoncommutativeAdditiveMonoid T>
T identity_element(std::plus<T>) {
  return T(0);
}

template <MultiplicativeMonoid T>
T identity_element(std::multiplies<T>) {
  return T(1);
}
```

power for an arbitrary group

```
template <Regular A, Integer N, GroupOperation Op>
requires(Domain<Op, A>)
A power_group(A a, N n, Op op) {
  if (n < 0) {
    n = -n;
    a = inverse_operation(op)(a);
  }
  return power_monoid(a, n, op);
}
```

examples of inverse_operation

```
template <AdditiveGroup T>
std::negate<T> inverse_operation(std::plus<T>) {
  return std::negate<T>();
}

template <MultiplicativeGroup T>
reciprocal<T> inverse_operation(std::multiplies<T>) {
  return reciprocal<T>();
}
```

reciprocal

```
template <MultiplicativeGroup T>
struct reciprocal
  : public std::unary_function<T, T> {
  T operator()(const T& x) const {
    return T(1) / x;
  }
};
```

Algorithms defined on abstract concepts

An ancient algorithm can be used on an abstract mathematical concept and then applied to a myriad of different situations.

Even more important algorithm on semigroups

- Additive semigroup

$$\sum$$

- Multiplicative semigroup

$$\prod$$

- Arbitrary semigroup
  - *reduction*

Reduction

- APL (Ken Iverson – 1962) (reduce)

  + / 1 2 3

- FP (John Backus – 1973) (insert)

  (/ +): <1, 2, 3>

- Tecton (Kapur et al – 1981)
  - semigroups, monoids, parallel reduction
- MapReduce (Dean et al – 2004)
  - commutativity

Problem 109

Using our work on multiply and power design a library version of reduction algorithm.

Reading 110

- Kenneth Iverson, *Notation as a Tool of Thought*, CACM, Vol. 23(8), Aug. 1980
- John Backus, *Can programming be liberated from the von Neumann style?*, CACM, Vol. 21(8), Aug. 1978
- Deepak Kapur et al, *Operators and Algebraic Structures*, Proceedings FPCA 1981
- Jeffrey Dean et al, *MapReduce: simplified data processing on large clusters*, Proceedings OSDI 2004

Four Journeys: Journey One

Lecture 3

Alexander Stepanov

Fibonacci Numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817, 39088169,…

Computation of Fibonacci numbers

$$F_0 = 0$$

$$F_1 = 1$$

$$F_i = F_{i-1} + F_{i-2}$$

“Obvious” Implementation

```
int fib0(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib0(n – 1) + fib0(n – 2);
}
```

fib0(5)

$$
\begin{aligned}
&F_5 = \\
&F_4 + F_3 = \\
&(F_3 + F_2) + (F_2 + F_1) = \\
&((F_2 + F_1) + (F_1 + F_0)) + ((F_1 + F_0) + F_1) = \\
&(((F_1 + F_0) + F_1) + (F_1 + F_0)) + ((F_1 + F_0) + F_1)
\end{aligned}
$$

Problem 116

How many additions are needed to compute fib0(n)?

Iterative Fibonacci

```
int fibonacci_iterative(int n) {
  if (n == 0) return 0;
  std::pair<int, int> v(0, 1);
  for (int i = 1; i < n; ++i)
    v = std::make_pair(v.second,
                       v.first + v.second);
  return v.second;
}
```

Fibonacci transformation

$$\begin{bmatrix} v_{i+1} \\ v_i \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} v_i \\ v_{i-1} \end{bmatrix}$$

*n*th Fibonacci vector

$$\begin{bmatrix} v_n \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Matrix multiplication is associative

We use power algorithm to get *n*th Fibonacci number.

Problem 121

Implement computing Fibonacci numbers using power.

Reading 122

EoP (pages 45 – 46) shows how to reduce the number of operations needed to multiply two *Fibonacci matrices*.

Linear Recurrences

A *linear recurrence function of order k* is a function *f* such that

$$f(y_0, \ldots, y_{k-1}) = \sum_{i=0}^{k-1} a_i y_i$$

A *linear recurrence sequence* is a sequence generated by such function from initial *k* values.

Linear Recurrence via power

$$\begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-k+1} \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & \dots & a_{k-2} & a_{k-1} \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \end{bmatrix}^{n-k+1} \begin{bmatrix} x_{k-1} \\ x_{k-2} \\ x_{k-3} \\ \vdots \\ x_0 \end{bmatrix}$$

Further Reading 125
Reducing number of operations

- Charles M. Fiduccia: An Efficient Formula for Linear Recurrences. SIAM J. Comput. 14(1): 106-112 (1985)

Matrices

- We combine power with matrix multiplication to compute linear recurrences.
- It is possible to use this technique for many other algorithms if we use more general notion of matrix multiplication.

Inner Product

$$\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} x_i y_i$$

Matrix-vector Product

$$\vec{w} = \left[x_{ij}\right] \vec{v}$$

$$w_i = \sum_{j=1}^{n} x_{ij} v_j$$

Matrix-matrix Product

$$[z_{ij}] = [x_{ij}]\ [y_{ij}]$$

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$$

General setting for matrix multiplication

- Where do coefficients come from?
- Semiring
  - two operations
    - $\oplus$ - associative and commutative
    - $\otimes$ - associative
  - distributivity

$$a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$$

$$(b \oplus c) \otimes a = b \otimes a \oplus c \otimes a$$

Transitive closure of a relation

- Take an $n \times n$ matrix with Boolean values.
- Use matrix multiplication generated by *Boolean* or $\{\vee, \wedge\}$-semiring.
- Raise it to *n-1* power.

Shortest path

- Take an $n \times n$ matrix with edge length values.
  - a_{ij} is the distance from node *i* to node *j*
- Use matrix multiplication generated by *Tropical* or {*min*, +} semiring

$$b_{ij} = \min_{k=1}^{n}(a_{ik} + a_{kj})$$

- Raise the matrix to *n-1* power.

Problem 133

1. Implement power-based shortest path algorithm.
2. Modify it to return not just the shortest distance but the shortest path (a sequence of edges).

Pythagoras (570BC - 490BC)

Life of Pythagoras

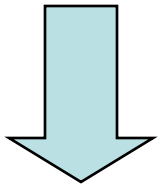
- Samos
  - Polycrates, his tunnel and his ring.
- Travels
  - Miletus, Egypt, Babylon, (India?), Croton
- Pythagorean brotherhood
  - Discipline
  - Program of study (μάθημα)
  - Political influence

He maintained that “the principles of mathematics are principles of all existing things.”

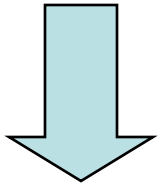
Aristotle

Pythagorean Quadrivium

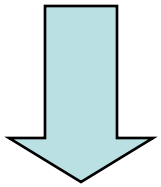
ASTRONOMY



GEOMETRY



NUMBER THEORY



MUSIC

Pythagorean Arithmetic

“He attached supreme importance to the study of arithmetic, which he advanced and took out of the region of commercial utility.”

Aristoxenus

Nicomachus of Gerasa

Introduction to Arithmetic

Ἀριθμητικὴ εἰσαγωγή

Triangular numbers

Triangular numbers

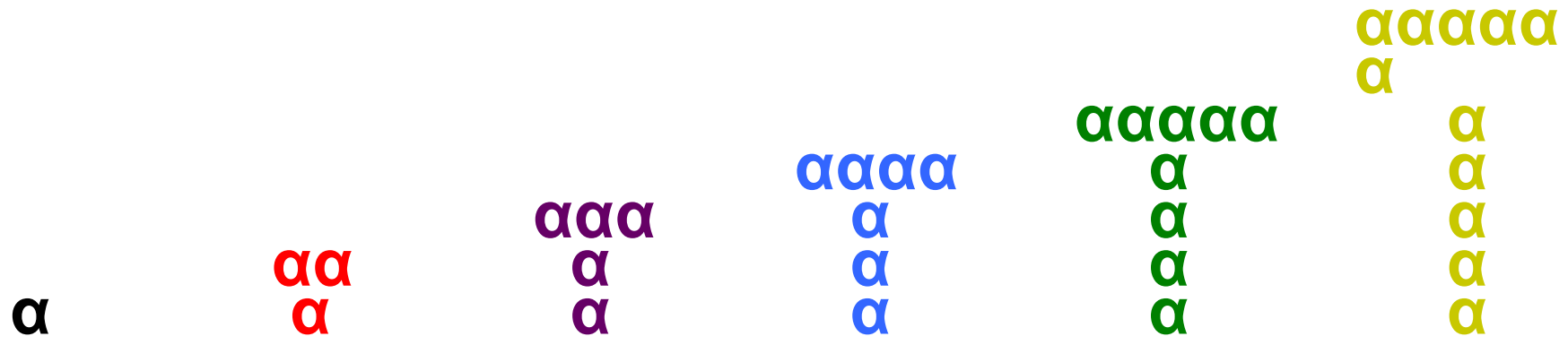
1	3	6	10	15	21

$$\triangle_n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Oblong numbers

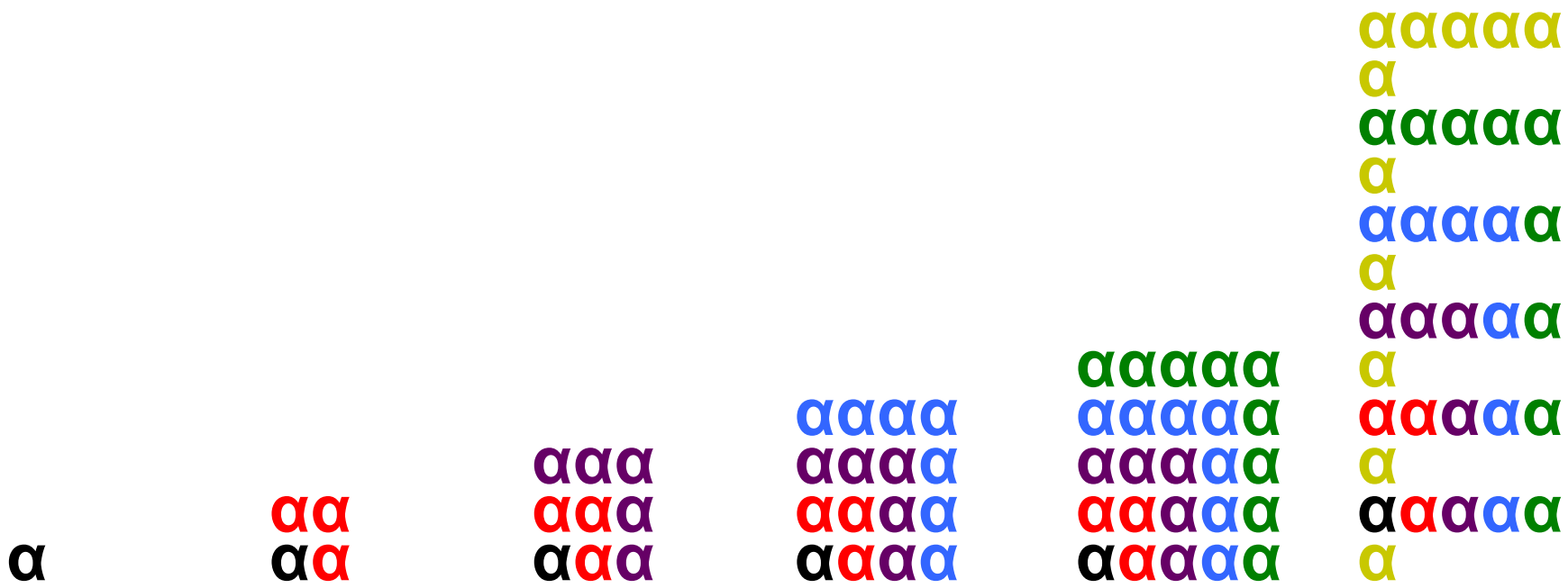
$$\square_n = 2\triangle_n = 2\sum_{i=1}^{n} i = n(n+1)$$

Gnomons



$$gnomon_n = 2n - 1$$

Square numbers



$$\square_n = \sum_{i=1}^{n} (2i - 1) = n^2$$

1 4 9 16 25 36

Problem 145

Find a geometric proof of the following:

Take any triangular number, multiply it by 8 and add 1. The result is a square number.

Plutarch, *Platonic Questions*

Prime numbers

Numbers that are not products of smaller numbers.

2, 3, 5, 7, 11, 13 …

Euclid VII, 32

Any number is either prime or divisible by some prime.

If not, “an infinite sequence of numbers will divide the number, each of which is less than the other; and this is impossible.”

Euclid, IX, 20

For any sequence of primes $\{p_1, \ldots, p_n\}$ there is a prime p not in the sequence.

Consider the number $q = 1 + \prod_{i=1}^{n} p_i$.

It is not divisible by any p_i. Its smallest prime factor is the desired p.

Therefore, there are infinitely many primes.

Eratosthenes (284BC – 195BC)

Life of Eratosthenes

- Born in Cyrene (Κυρήνη), Libya
- Studied in Athens (with Zeno?)
- Around 244BC Ptolemy Euergetes invites him to Alexandria as a tutor.
  - Duplication of the cube, column and poem
- Measuring the Earth (within 2%!).
- A friend of Archimedes.
- Still just a beta…

Sieve (*κόσκινον*) of Eratosthenes

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53

step 3

3 5 7 **9** 11 13 **15** 17 19 **21** 23 25 **27** 29 31 **33** 35 37 **39** 41 43 **45** 47 49 **51** 53

step 5

3 5 7 **9** 11 13 **15** 17 19 **21** 23 **25** **27** 29 31 **33** **35** 37 **39** 41 43 **45** 47 49 **51** 53

step 7

3 5 7 **9** 11 13 **15** 17 19 **21** 23 **25** **27** 29 31 **33** **35** 37 **39** 41 43 **45** 47 **49** **51** 53

mark_sieve

```
template <RandomAccessIterator I,
          Integer N>
void mark_sieve(I first, I last, N step) {
  *first = false;
  while (last - first > step) {
    first = first + step;
    *first = false;
  }
}
```

Sifting lemmas

- The square of the smallest prime factor of a composite number n is less or equal than n.
- Any composite number less than p^2 is sifted by a prime less than p.
- When sifting by p, start marking at p^2.
- If the table is of size m, stop sifting when $p^2 \geq m$.

Sifting formulas

$$\begin{aligned}
\text{value at position } i: \ \operatorname{val}(i) &= 3 + 2i = 2i + 3 \\
\text{position of value } v: \ \operatorname{idx}(v) &= \frac{v-3}{2} \\
\text{step between multiples of } p: \ \operatorname{step}(i) &= \operatorname{idx}((n+2)p) - \operatorname{idx}(np) \\
&= \frac{np+2p-3}{2} - \frac{np-3}{2} \\
&= p = 2i+3
\end{aligned}$$

$$\begin{aligned}
\text{position of square of value at } i: \ \operatorname{idx}(\operatorname{val}^2(i)) &= \frac{(2i+3)^2-3}{2} \\
&= \frac{4i^2+12i+9-3}{2} \\
&= 2i^2+6i+3
\end{aligned}$$

sift0

```
template <RandomAccessIterator I, Integer N>
void sift0(I first, N n) {
   std::fill(first, first + n, true);
   N i(0);
   N square(3);
   while (square < n) {
     // invariant: square = 2i^2 + 6i + 3
     if (first[i]) {
       mark_sieve(first + square,
                  first + n,  // last
                  i + i + 3); // step
     }
     ++i;
     square =  3 + 2*i*(i + 3);
   }
}
```

sift1

```
template <RandomAccessIterator I, Integer N>
void sift1(I first, N n) {
  I last = first + n;
  std::fill(first, last, true);
  N i(0);
  N square(3);
  N step(3);
  while (square < n) {
    // invariant: square = 2i² + 6i + 3, step = 2i + 3
    if (first[i]) mark_sieve(first + square, last, step);
    ++i;
    step   = i + i + 3;
    square =  3 + 2*i*(i + 3);
  }
}
```

Strength reduction

From Wikipedia, the free encyclopedia

Strength reduction is a *compiler optimization* where expensive operations are replaced with equivalent but less expensive operations. The classic example of strength reduction converts "strong" multiplications inside a loop into "weaker" additions – something that frequently occurs in array addressing.

Examples of strength reduction include:

- replacing a multiplication within a loop with an addition
- replacing an exponentiation within a loop with a multiplication

Increment computation

Replace

step = i + i + 3;
square = 3 + 2*i*(i + 3);

with

step += δ_{step};
square += δ_{square};

δ computations

$$\delta_{step}: \quad (2(i+1)+3)-(2i+3)=2$$

$$\begin{aligned}\delta_{square}: \quad & (2(i+1)^2+6(i+1)+3)-(2i^2+6i+3)\\ & =4i+8=(2i+3)+(2i+2+3)\\ & =(2i+3)+(2(i+1)+3)\\ & =\mathrm{step}(i)+\mathrm{step}(i+1)\end{aligned}$$

sift

```
template <RandomAccessIterator I, Integer N>
void sift(I first, N n) {
  I last = first + n;
  std::fill(first, last, true);
  N i(0);
  N square(3);
  N step(3);
  while (square < n) {
    // invariant: square = 2i^2 + 6i + 3, step = 2i + 3
    if (first[i]) mark_sieve(first + square, last, step);
    ++i;
    square += step;
    step   += N(2);
    square += step;
  }
}
```

Problem 161

Time the sieve using different data sizes:

- bool (use std::vector<bool>)
- uint8_t
- uint16_t
- uint32_t
- uint64_t

Problem 162

Using the sieve, graph the function

$\pi(n)$ = the number of primes < n

for n up to 10^7 and find its analytic approximation.

Palindromic Primes

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 **101** 103 107 109 113 127 **131** 137 139
149 **151** 157 163 167 173 179 **181 191** 193 197 199 211
223 227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 **313** 317 331 337 347 349 **353** 359 367
373 379 **383** 389 397 401 409 419 421 431 433 439 443
449 457 461 463 467 479 487 491 499 503 509 521 523
541 547 557 563 569 571 577 587 593 599 601 607 613
617 619 631 641 643 647 653 659 661 673 677 683 691
701 709 719 **727** 733 739 743 751 **757** 761 769 773 **787**
797 809 811 821 823 827 829 839 853 857 859 863 877
881 883 887 907 911 **919 929** 937 941 947 953 967 971
977 983 991 997

No Palindromic Primes ?!

1009 1013 1019 1021 1031 1033 1039 1049 1051 1061
1063 1069 1087 1091 1093 1097 1103 1109 1117 1123
1129 1151 1153 1163 1171 1181 1187 1193 1201 1213
1217 1223 1229 1231 1237 1249 1259 1277 1279 1283
1289 1291 1297 1301 1303 1307 1319 1321 1327 1361
1367 1373 1381 1399 1409 1423 1427 1429 1433 1439
1447 1451 1453 1459 1471 1481 1483 1487 1489 1493
1499 1511 1523 1531 1543 1549 1553 1559 1567 1571
1579 1583 1597 1601 1607 1609 1613 1619 1621 1627
1637 1657 1663 1667 1669 1693 1697 1699 1709 1721
1723 1733 1741 1747 1753 1759 1777 1783 1787 1789
1801 1811 1823 1831 1847 1861 1867 1871 1873 1877
1879 1889 1901 1907 1913 1931 1933 1949 1951 1973
1979 1987 1993 1997 1999

Problem 165

- Are there palindromic primes > 1000?
- What is the reason for the lack of them in the interval [1000, 2000]?
- What happens if we change our base to 16? To an arbitrary *n*?

Four Journeys: Journey One

Lecture 4

Alexander Stepanov

Perfect numbers

- An *aliquot part* of a positive integer is a *proper* divisor of the integer.
- The *aliquot sum* of a positive integer is the sum of its *aliquot parts*.
- A positive integer is *perfect* if it is equal to its *aliquot sum*.

Perfect numbers known to the Greeks

$$
\begin{aligned}
6 &= 1+2+3 \\
28 &= 1+2+4+7+14 \\
496 &= 1+2+4+8+16+31+62+124+248 \\
8128 &= 1+2+4+8+16+32+64+127 \\
&\quad +254+508+1016+2032+4064
\end{aligned}
$$

Their prime factorization

$$6 = 2 \cdot 3$$

$$28 = 4 \cdot 7$$

$$496 = 16 \cdot 31$$

$$8128 = 64 \cdot 127$$

Further decomposition

$$6 = 2 \cdot 3 = 2^1 \cdot (2^2 - 1)$$

$$28 = 4 \cdot 7 = 2^2 \cdot (2^3 - 1)$$

$$496 = 16 \cdot 31 = 2^4 \cdot (2^5 - 1)$$

$$8128 = 64 \cdot 127 = 2^6 \cdot (2^7 - 1)$$

Other powers of 2

$$6 = 2 \cdot 3 = 2^1 \cdot (2^2 - 1)$$

$$28 = 4 \cdot 7 = 2^2 \cdot (2^3 - 1)$$

$$120 = 8 \cdot 15 = 2^3 \cdot (2^4 - 1) \text{ not perfect}$$

$$496 = 16 \cdot 31 = 2^4 \cdot (2^5 - 1)$$

$$2016 = 32 \cdot 63 = 2^5 \cdot (2^6 - 1) \text{ not perfect}$$

$$8128 = 64 \cdot 127 = 2^6 \cdot (2^7 - 1)$$

Euclid, *Elements*, Book IX, Proposition 36

If $\sum_{i=0}^{n} 2^i$ is prime then $2^n \sum_{i=0}^{n} 2^i$ is perfect.

Difference of powers

$$
\begin{aligned}
x^2 - y^2 &= (x-y)(x+y) \\
x^3 - y^3 &= (x-y)(x^2+xy+y^2) \\
&\vdots \\
x^{n+1} - y^{n+1} &= (x-y)(x^n + x^{n-1}y + \cdots + xy^{n-1} + y^n)
\end{aligned}
$$

$$
\begin{aligned}
x(x^n + x^{n-1}y + \cdots + xy^{n-1} + y^n) &= x^{n+1} + x^n y + x^{n-1}y^2 + \cdots + xy^n \\
y(x^n + x^{n-1}y + \cdots + xy^{n-1} + y^n) &= \qquad x^n y + x^{n-1}y^2 + \cdots + xy^n + y^{n+1}
\end{aligned}
$$

Sum of odd powers

$$
\begin{aligned}
x^{2n+1} + y^{2n+1} &= x^{2n+1} - -y^{2n+1} \\
&= x^{2n+1} - (-y)^{2n+1} \\
&= (x - (-y))(x^{2n} + x^{2n-1}(-y) + \cdots + (-y)^{2n}) \\
&= (x + y)(x^{2n} - x^{2n-1}y + \cdots - xy^{2n-1} + y^{2n})
\end{aligned}
$$

Restatement of Euclid's theorem

$$(2-1)(2^{n-1}+2^{n-2}+\cdots+2+1)=\sum_{i=0}^{n-1}2^i=2^n-1$$

If 2^n-1 is prime then $2^{n-1}(2^n-1)$ is perfect.

σ(*n*) - sum of the divisors

$$n = p_1^{a_1} p_2^{a_2} \cdots p_m^{a_m}$$

$$\sigma(n) = \prod_{i=1}^{m} (1 + p_i + p_i^2 + \cdots + p_i^{a_i})$$

$$= \prod_{i=1}^{m} \frac{p_i^{a_i+1} - 1}{p_i - 1}$$

Problem 177

Prove that if *n* and *m* are *co-prime* (have no common prime factors) then

$$\sigma(nm) = \sigma(n)\sigma(m)$$

α(*n*) - aliquot sum

$$\alpha(n) = \sigma(n) - n$$

Proof of Euclid, IX, 36

$$Q = 2^{n-1}(2^n - 1)$$

$$m = 2,\ p_1 = 2,\ a_1 = n - 1,\ p_2 = 2^n - 1,\ a_2 = 1$$

$$\sigma(Q) = \frac{2^{(n-1)+1} - 1}{1} \cdot \frac{(2^n - 1)^2 - 1}{(2^n - 1) - 1}$$

$$= (2^n - 1)((2^n - 1) + 1)$$

$$= 2^n(2^n - 1) = 2 \cdot 2^{n-1}(2^n - 1) = 2Q$$

$$\alpha(Q) = \sigma(Q) - Q = Q$$

The inverse of IX, 36

- In XVIII century Leonard Euler proved that every *even* perfect number is of the form

$$2^{p-1}(2^p - 1)$$

- Odd perfect numbers?

Problem 181

Prove that every even perfect number is a triangular number.

Problem 182

Prove that the sum of the reciprocals of the divisors of a perfect number is always *2*.

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{6} = 2$$

Practical years

- In 212 BC the study of Mathematics in the Latin West takes a long break.

- In the third journey we will see how it comes back around 1200 AD.

Noli turbare circulos meos!

An admission of a practical fellow

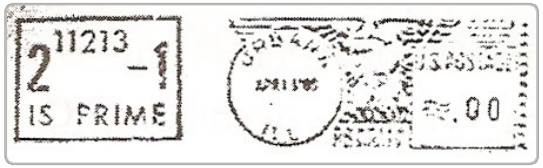
In summo apud illos honore geometria fuit, itaque nihil mathematicis inlustrius; at nos metiendi ratiocinandique utilitate huius artis terminavimus modum.

Among the Greeks geometry was held in highest honor; nothing could outshine mathematics. But we have limited the usefulness of this art to measuring and calculating.

Marcus Tullius Cicero,*Tusculan Disputations*

The tomb of Archimedes.

Marin Mersenne (1588 – 1648) “walking scientific journal”

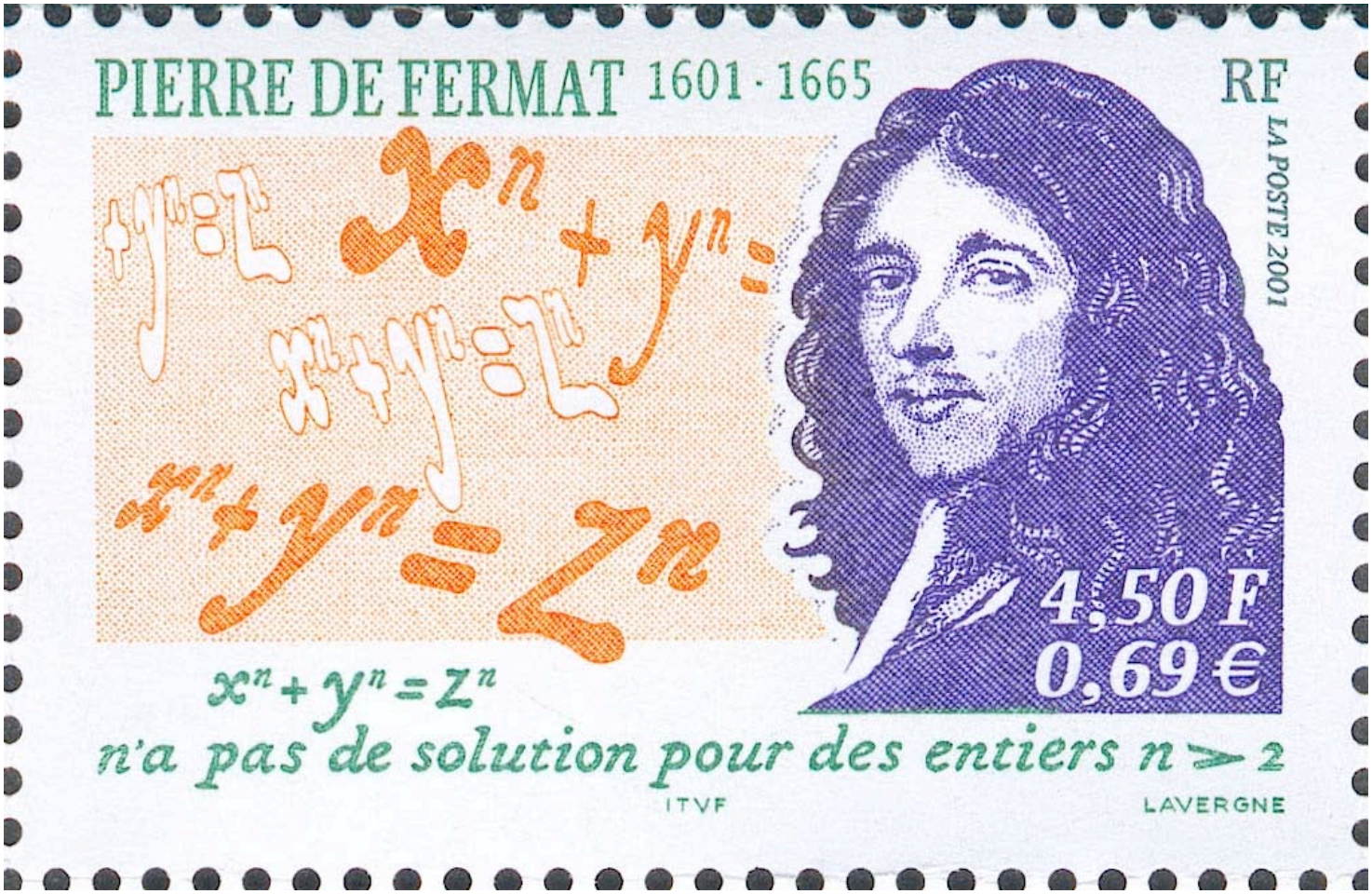


Wikipedia on Mersenne

His most important contribution to the advance of learning was his extensive correspondence with mathematicians and other scientists in many countries. At a time when the scientific journal had not yet come into being, Mersenne was the center of a network for exchange of information.

A few of Mersenne's friends

- Galileo Galilei
- René Descartes
- Pierre de Fermat
- Blaise Pascal


PIERRE DE FERMAT 1601 - 1665
RF
LA POSTE 2001
4,50 F
0,69 €
$x^n + y^n = z^n$
n'a pas de solution pour des entiers $n > 2$
ITVF
LAVERGNE

When is $2^n - 1$ prime?

- Greeks: true for $n = 2, 3, 5, 7, 13$
- Hudalricus Regius (1536): false for $n = 11$

  $2^{11} - 1 = 2047 = 23 \times 89$
- Pietro Cataldi (1603): true for $n = 17, 19$
- true *for* $n = 23, \mathbf{29}, 31, 37$
- Fermat discovered that

$$2^{23} - 1 = 8388607 = 47 \times 178481$$

$$2^{37} - 1 = 137438953471 = 223 \times 61631877$$

Mersenne primes

In 1644 in his book *Cogitata Physico Mathematica*, Mersenne states that if $n \leq 257$ then 2^n-1 is prime if and only if n =

2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257

67 might be a misprint for *61*.

Mersenne missed: 89, 107

Letter from Fermat to Mersenne

In June 1640 Fermat states that his factorization of $2^{37} - 1$ depends on the following three discoveries:

1. If n is not a prime, $2^n - 1$ is not a prime.
2. If n is a prime, $2^n - 2$ is a multiple of $2n$.
3. If n is a prime, and p is a prime divisor of $2^n - 1$, then $p - 1$ is a multiple of n.

Factoring $2^{37} - 1$

- It has a prime factor p.
- $p - 1$ is divisible by 37.
- $p = 37n + 1$
- p is odd
- $p = 74m + 1$
- $m = 1$, but 75 is not a prime
- $m = 2$, 149 is prime, but does not divide
- $m = 3$, 223 is prime and does divide!

If $2^n - 1$ is prime then n is prime

Proof: Suppose n is not prime. Then,

$$n = uv, \quad u > 1, v > 1$$

$$\begin{aligned} 2^n - 1 &= 2^{uv} - 1 \\ &= (2^u)^v - 1 \\ &= (2^u - 1)((2^u)^{v-1} + (2^u)^{v-2} + \cdots + (2^u) + 1) \end{aligned}$$

$$u > 1 \implies 1 < 2^u - 1$$

$$1 < (2^u)^{v-1} + (2^u)^{v-2} + \cdots + (2^u) + 1$$

Contradiction!

Fermat's discoveries 2 and 3

"… he would send [the proof] if he did not fear being too long."

Fermat to Frenicle

Little Fermat Theorem

If p is prime, $a^{p-1} - 1$ is divisible by p for any $0 < a < p$.

Delayed proof

- Fermat claimed to have the proof in 1640.
- Leibnitz discovered the proof some time between 1676 an 1680, but did not publish.
- Euler published two different proofs in 1742 and 1750.

Another Conjecture by Fermat

$$2^n + 1 \text{ is prime} \iff n = 2^i$$

$2^n + 1$ is prime $\implies n = 2^i$

Proof:

If $n \neq 2^i$ then n has an odd factor $2q + 1 > 1$ and $n = m(2q + 1)$.

Using the formula for sum of odd powers, we factor $2^n + 1$:

$$\begin{aligned} 2^n + 1 &= (2^m)^{2q+1} + 1^{2q+1} \\ &= (2^m + 1)((2^m)^{2q} - (2^m)^{2q-1} + \cdots + 1) \end{aligned}$$

$2^{2^i} + 1$ is prime

- Fermat states that 3, 5, 17, 257, 65537, 4294967297 and 18446744073709551617 are primes.
- Fermat conjectured that so are they all.
- In 1732 Euler shows that

  $2^{32} + 1 = 4294967297 = 641 \times 6700417$

- For $5 \leq i \leq 32$ they are composite.
- Are there any more *Fermat primes*?

Leonhard Euler, *c'est notre maître à tous*

Euclid, *VII, 30*

The product of two integers smaller than a prime *p* is not divisible by *p*.

The proof of *VII, 30*

$$p \text{ is prime } \wedge\ 0 < a, b < p \implies ab = mp + r \wedge 0 < r < p$$

Proof:

Assume the contrary. Then for a given a let b be the smallest integer such that $ab = mp$. Then since p is prime

$$p = bu + v \wedge 0 < v < b$$

$$ap = abu + av$$

$$ap - mpu = av$$

$$av = (a - mu)p \wedge v < b$$

Contradiction!

Notational convention

For the rest of the journey, *p* is always assumed to be prime.

Permutation of Remainders Lemma

For any $0 < a < p$,

$$\begin{aligned} a \cdot \{1, \ \ldots, \ p-1\} &= \{a, \ \ldots, \ a(p-1)\} \\ &= \{q_1 p + r_1, \ \ldots, \ q_{p-1} p + r_{p-1}\} \end{aligned}$$

where $0 < r_i < p \wedge i \neq j \implies r_i \neq r_j$.

Proof: Suppose $r_i = r_j$ and $i < j$

$$(q_j p + r_j) - (q_i p + r_i) = (q_j - q_i)p = aj - ai = a(j-i)$$

Contradicts VII, 30

Cancellation Law

For any $0 < a < p$ there is $0 < b < p$ such that $ab = mp + 1$.

Proof:

By the Permutation of Remainders Lemma we know that one of

$$a \cdot \{1, \ \ldots, \ p-1\}$$

maps to 1.

Self-cancelling elements

1 and *p−1* cancel themselves.

Self-cancelling Lemma

For any $0 < a < p,\ a^2 = mp + 1 \implies a = 1 \vee a = p - 1$
Proof:

$$a \neq 1 \wedge a \neq p - 1 \implies 1 < a < p - 1$$

$$a^2 - 1 = (a - 1)(a + 1) = mp$$

And, since $0 < a - 1, a + 1 < p,$ contradiction with VII, 30.

Wilson Theorem

- Was introduced by Waring who (incorrectly) attributes it to Wilson.
- Waring stated that he cannot prove it since he did not have the right notation.
- Gauss remarks: ”One needs notion, not notation.”
  - At nostro quidem iudicio huiusmodi veritates ex notionibus potius quam ex notationibus hauriri debebant.

Wilson's Theorem

$$(p-1)! = (p-1) + mp$$

Proof:

Every number except *1* and *p−1* is cancelled by its inverse.

Problem 211

Prove that if $n > 4$ is composite then $(n-1)!$ is a multiple of n.

The proof of Little Fermat

By Wilson Theorem:

$$\prod_{i=1}^{p-1} ai = a^{p-1} \prod_{i=1}^{p-1} i = a^{p-1}((p-1) + up) = a^{p-1}(1+u)p - a^{p-1}$$

by the Permutation of Remainders Lemma:

$$\prod_{i=1}^{p-1} ai = \prod_{i=1}^{p-1} (q_i p + r_i) = vp + \prod_{i=1}^{p-1} r_i = (p-1) + vp = -1 + (v+1)p$$

$$(a^{p-1} + u)p - a^{p-1} = -1 + (v+1)p$$

$$a^{p-1} = 1 + np$$

Multiplicative inverse modulo *n*

For integer $n > 1$ and integer $u > 0$,
we call v a *multiplicative inverse modulo n*
if there is an integer q such that
$uv = 1 + qn$.

a^{p-2} is an inverse of a

Trivial.

Non-invertibility Lemma

If $n = uv \wedge 1 < u, v$ then u is not invertible modulo n.

Proof:

Let $uv = n$ and w be an inverse of u. Then

$$wn = wuv = (1 + mn)v = v + mvn$$

and

$$(w - mv)n = zn = v$$

Contradiction.

Converse of Little Fermat

If for all a in $0 < a < n$, $a^{n-1} = 1 + q_a n$
then n is prime.

Proof:

If $n = uv$ then u is not invertible.
But $u^{n-2}u = u^{n-1} = 1 + q_u n$. Contradiction.

“Useful” mathematics

The search for (so far) useless perfect numbers led to the discovery of one of the most practically useful theorems in all of mathematics.

In particular, this theorem helps us to find large primes used daily in e-commerce.

Four Journeys: Journey One

Lecture 5

Alexander Stepanov

Multiplication Table *modulo 7*

$$1 \times i : 1,2,3,4,5,6 = 1,2,3,4,5,6$$

$$2 \times i : 2,4,6,1+1\times 7,3+1\times 7,5+1\times 7 = 2,4,6,1,3,5$$

$$3 \times i : 3,6,2+1\times 7,5+1\times 7,1+2\times 7,4+2\times 7 = 3,6,2,5,1,4$$

$$4 \times i : 4,1+1\times 7,5+1\times 7,2+2\times 7,6+2\times 7,3+2\times 7 = 4,1,5,2,6,3$$

$$5 \times i : 5,3+1\times 7,1+2\times 7,6+2\times 7,4+3\times 7,2+4\times 7 = 5,3,1,6,4,2$$

$$6 \times i : 6,5+1\times 7,4+2\times 7,3+3\times 7,2+4\times 7,1+5\times 7 = 6,5,4,3,2,1$$

Multiplication modulo 7 with inverses

$1 \times i:$	$\mathbf{1}, 2, 3, 4, 5, 6$	1
$2 \times i:$	$2, 4, 6, \mathbf{1}, 3, 5$	4
$3 \times i:$	$3, 6, 2, 5, \mathbf{1}, 4$	5
$4 \times i:$	$4, \mathbf{1}, 5, 2, 6, 3$	2
$5 \times i:$	$5, 3, \mathbf{1}, 6, 4, 2$	3
$6 \times i:$	$6, 5, 4, 3, 2, \mathbf{1}$	6

Wilson's Theorem *modulo 7*

$$
\begin{aligned}
\mathbf{6!} &= 1 \times (2 \times 4) \times (3 \times 5) \times 6 \\
&= 1 \times (1 + 1 \times 7) \times (1 + 2 \times 7) \times 6 \\
&= (1 + 1 \times 7) \times (1 + 2 \times 7) \times 6 \\
&= (1 + 2 \times 7 + 1 \times 7 + 2 \times 7 \times 7) \times 6 \\
&= (1 + 17 \times 7) \times 6 \\
&= 6 + (17 \times 6) \times 7 \\
&= \mathbf{6} + 102 \times 7
\end{aligned}
$$

Wilson *modulo 7*

$$\begin{aligned} \mathbf{6!} &= 1 \times (2 \times 4) \times (3 \times 5) \times 6 \\ &= 1 \times (1 + 1 \times 7) \times (1 + 2 \times 7) \times 6 \\ &= \mathbf{6} + 102 \times 7 \end{aligned}$$

Fermat *modulo 7*

$$
\begin{aligned}
2^6 \times 6! &= (2\times 1)\times(2\times 2)\times(2\times 3)\times(2\times 4)\times(2\times 5)\times(2\times 6)\\
&= 2\times 4\times 6\times(1+1\times 7)\times(3+1\times 7)\times(5+1\times 7)\\
&= 6! + 7m \quad \text{dropping factorial by Wilson we get}\\
2^6 \times 6 &= 6 + 7u \quad \text{multiplying both sides by 6 we get}\\
2^6 &= 1 + 7v
\end{aligned}
$$

Multiplication *modulo 10* with inverses

$1 \times i:$	$1,2,3,4,5,6,7,8,9$	1
$2 \times i:$	$2,4,6,8,0,2,4,6,8$	
$3 \times i:$	$3,6,9,2,5,8,1,4,7$	7
$4 \times i:$	$4,8,2,6,0,4,8,2,6$	
$5 \times i:$	$5,0,5,0,5,0,5,0,5$	
$6 \times i:$	$6,2,8,4,0,6,2,8,4$	
$7 \times i:$	$7,4,1,8,5,2,9,6,3$	3
$8 \times i:$	$8,6,4,2,0,8,6,4,2$	
$9 \times i:$	$9,8,7,6,5,4,3,2,1$	9

Reducing the Multiplication Table

$1 \times i:$	$1, 2, 3, 4, 5, 6, 7, 8, 9$	1
$2 \times i:$	$2, 4, 6, 8, 0, 2, 4, 6, 8$	
$3 \times i:$	$3, 6, 9, 2, 5, 8, 1, 4, 7$	7
$4 \times i:$	$4, 8, 2, 6, 0, 4, 8, 2, 6$	
$5 \times i:$	$5, 0, 5, 0, 5, 0, 5, 0, 5$	
$6 \times i:$	$6, 2, 8, 4, 0, 6, 2, 8, 4$	
$7 \times i:$	$7, 4, 1, 8, 5, 2, 9, 6, 3$	3
$8 \times i:$	$8, 6, 4, 2, 0, 8, 6, 4, 2$	
$9 \times i:$	$9, 8, 7, 6, 5, 4, 3, 2, 1$	9

Euler Totient function

$$\phi(n) = |\{0 < i < n \wedge \text{coprime}(i, n)\}|$$

Totient of prime

$$\phi(p) = p - 1$$

Euler's Theorem

For $0 < a < n$,
coprime a and $n \iff a^{\phi(n)} = 1 + mn$

Problem 229

- Prove Euler's theorem by modifying the proof of the Little Fermat Theorem
- Steps
  - Replace Permutation of Remainders Lemma with Permutation of Coprime Remainders Lemma
  - Prove that every coprime remainder has a multiplicative inverse
  - Use the product of all coprime remainders where that proof of Little Fermat uses the product of all non-zero remainders.

Abstraction

Generalizing code is like generalizing theorems and their proofs.

Totient of power of prime

$$\begin{aligned}
\phi(p^m) &= (p^m - 1) - |\{p, 2p, \ldots, p^m - p\}| \\
&= (p^m - 1) - |\{1, 2, \ldots, p^{m-1} - 1\}| \\
&= (p^m - 1) - (p^{m-1} - 1) \\
&= p^m - p^{m-1} \\
&= p^m(1 - \frac{1}{p})
\end{aligned}$$

Totient of $n = p^u q^v$

$$\begin{aligned}
\phi(n) &= (n-1) - (\frac{n}{p} - 1) - (\frac{n}{q} - 1) + (\frac{n}{pq} - 1) \\
&= n - \frac{n}{p} - \frac{n}{q} + \frac{n}{pq} \\
&= n(1 - \frac{1}{p} - \frac{1}{q} + \frac{1}{pq}) \\
&= n((1 - \frac{1}{p}) - \frac{1}{q}(1 - \frac{1}{p})) \\
&= n(1 - \frac{1}{p})(1 - \frac{1}{q}) \\
&= p^u(1 - \frac{1}{p})q^v(1 - \frac{1}{q}) = \phi(p^u)\phi(q^v)
\end{aligned}$$

General case totient formula

$$\phi(n) = \phi(\prod_{i=1}^{m} p_i^{e_i})$$

$$= n\prod_{i=1}^{m}(1 - \frac{1}{p_i})$$

$$= \prod_{i=1}^{m} \phi(p_i^{e_i})$$

Primality Testing

The problem of distinguishing prime numbers from composite and of resolving the latter into their prime factors is known to be one of the most important and useful in arithmetic.

C.F. Gauss

Disquisitiones Arithmeticae, article 329

Divides

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

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;
}
```

is_prime

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

Complexity: $O(\sqrt{n}) = O(2^{(\log n)/2})$

Exponential in *number of digits*.

Modulo Multiplication

```
template <Integer I>
struct modulo_multiply {
  I modulus;
  modulo_multiply(const I& i) : modulus(i) {}
  I operator() const (const I& n,
                      const I& m) {
    return (n * m) % modulus;
  }
};
```

Modulo Multiplication Identity

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

Multiplicative inverse - Fermat

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

fermat_test

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

Why don’t I use power_monoid?

Co-prime

Two integers are *co-prime* if and only if they do not share a prime divisor.

Carmichael Numbers

A composite number $n > 1$ is a *Carmichael number* iff

$$\forall\, b > 1,\ \text{coprime}(b, n) \implies \exists m,\ b^{n-1} = 1 + mn$$

or

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

Problem 244

1. Implement a function

   bool is_carmichael(n)

1. Find the first seven Carmichael numbers.

```
template <Integer I>
bool miller_rabin_test(I n, I q, I k, I w) {

  //precondition n > 1 ∧ n − 1 = 2^k q ∧ q is odd
  modulo_multiply<I> op(n);
  I x = power_semigroup(w, q, op);
  I i(1);
  while (x != I(1) && x != n - I(1)) {

    //invariant x = w^(2^(i−1) q)
    if (i >= k) return false;
    ++i;
    x = op(x, x);
  }
  return x != I(1) || i == I(1);
}
```

```
template <Integer I>
bool miller_rabin_test(I n, I q, I k, I w) {
    //precondition  n > 1  ∧  n − 1 = 2^k q  ∧  q  is odd
  modulo_multiply<I> op(n);
  I x = power_semigroup(w, q, op);
  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 = op(x, x);
    if (x == n – I(1)) return true;
    if (x == I(1))     return false;
  }
  return false;
}
```

Miller-Rabin Guarantee

- miller_rabin_test is wrong with probability at most 25% for a random *w*.
- Randomly choosing, say, 100 witnesses makes the probability of error less than 2^{-200}
- “It is much more likely that our computer has dropped a bit, due […] to cosmic radiations.”

Donald E. Knuth

Primes is in P (2002)

Manindra Agrawal, Neeraj Kayal, and Nitin Saxena

Full treatment of AKS

There is a pdf of Andrew Granville's expository paper on the wiki. It is very well written, but still quite difficult for a non-specialist.

Cryptology

- *Cryptography* develops ciphers.
- *Cryptanalysis* breaks ciphers.

- For a brief history of Cryptology, read my notes on the wiki.

Cryptosystem

- plaintext => encryption(key_0) => ciphertext
- ciphertext => decryption(key_1) => plaintext
- the system is *symmetric* if $key_0 = key_1$
- the system is *asymmetric* otherwise

Public Key Cryptosystem

- An encryption scheme in which users would have a pair of keys, a public key *pub* for encrypting, and a private key *prv* for decrypting.
- When Alice wants to send a message to Bob she encrypts with Bob's public key. Bob uses his private key to decipher her message.

Requirements for a Public Key Cryptosystem

- The encryption function needs to be a *one-way function:* easy to compute with an inverse that is hard to compute.
  - hard means exponential in the size of the key
- The inverse function has to be easy to compute when you have access to a certain additional piece of information (*trapdoor*).
- A function meeting these two requirements is known as a *trapdoor one-way function.*
- Both encryption and decryption functions are publicly known.

Inventors of Public Key Cryptosystems

- Whit Diffie, Martin Hellman, Ralph Merkle
  - preceded by James Ellis
- Ron Rivest, Adi Shamir and Len Adleman
  - preceded by Clifford Cocks
    - preceded by unnamed NSA researcher

Clifford Cocks (1950 -) the secret inventor of RSA

RSA key generation

- Compute
  - random large primes p_1 and p_2
  - $n = p_1 p_2$
  - $\varphi(p_1 p_2) = (p_1 - 1)(p_2 - 1)$
  - random *pub*, coprime with $\varphi(p_1 p_2)$
  - *prv*, the multiplicative inverse of *pub* modulo $\varphi(p_1 p_2)$
    - wait for the 2nd journey for the algorithm
- Destroy: p_1 , p_2
- Publish: *(pub, n)*
- Keep secret: *prv*

Encoding/decoding

- Encode:

```
power_semigroup(plain_text_block,
        pub,
        modulo_multiply<I>(n));
```

- Decode:

```
power_semigroup(cipher_text_block,
        prv,
        modulo_multiply<I>(n));
```

Justification of RSA

$$
\begin{aligned}
\left(m^{\mathrm{pub}}\right)^{\mathrm{prv}} &= m^{\mathrm{pub}\times\mathrm{prv}} \\
&= m^{1+q\phi(p_1p_2)} \\
&= mm^{q\phi(p_1p_2)} \\
&= m\left(m^{\phi(p_1p_2)}\right)^{q} \\
&= m + wn
\end{aligned}
$$

Presupposition

- Factoring is hard and, therefore, computing φ is not feasible.

- The presupposition is not true if quantum computers are realizable.

extended_gcd

```
template <EuclideanDomain I>
std::pair <I, I> extended_gcd(I a , I b) {
  if (b == 0) return std::make_pair(I(1), a);
  I u(1);
  I v(0);
  while (true) {
    I q = a / b;
    a -= q * b;  // a = a % b;
    if (a == I(0)) return std::make_pair(v, b);
    u -= q * v;

    q = b / a;
    b -= q * a;  // b = b % a;
    if (b == I(0)) return std::make_pair(u, a);
    v -= q * u;
  }
}
```

multiplicative_inverse

```
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;
}
```

returns multiplicative inverse of *a* modulo *n* if it exists or 0 if it does not.

Project 262

- Implement RSA key generation library.

- Implement RSA message encoder/decoder that takes a string and the key as its arguments.

“Spoils of the Egyptians”?

Let us take from mathematicians what is eternal and beautiful, while leaving behind their passing fads.