# NOTES ON THE FOUNDATIONS OF PROGRAMMING

ALEX STEPANOV AND MAT MARCUS

Disclaimer: Please do not redistribute. Instead, requests for a current draft should go to Mat Marcus. These notes are a work in progress and do not constitute a book. In particular, most of the current effort is directed towards writing up new material. As a consequence little time remains for structuring, refinement, or clean up, so please be patient. Nevertheless, suggestions, comments and corrections are welcomed. Please reply to mmarcus@adobe.com and stepanov@adobe.com.

*What thou lovest well remains,*
*the rest is dross*
—Ezra Pound, *Cantos*

## CONTENTS

## 1. Introduction

Modern science is fragmented. We have a split not only between natural sciences and humanities, not only between different natural sciences, but even inside every scientific discipline. It is remarkable how quickly Computer Science succumbed to this. We still have with us the representatives of the great founding generation of programming, people like Donald Knuth and Nicklaus Wirth, who knew everything. They could design a computer, write a compiler, invent an algorithm, prove a theorem. Not so now. Do not expect a person specializing in algorithms to know anything about programming languages. Do not expect a specialist in computer architecture to know anything about theory of computation. It would be good for us all to recall an ancient warning about the house divided. This course will attempt to ignore the boundaries and present a view of programming as a unified discipline.

Programming has become a disreputable, lowly activity. More and more programmers try to become managers, product managers, architects, evangelists – anything but writing code. It is possible now to find a professor of Computer Science who never wrote a program. And it is almost impossible to find a professor who actually writes code that is used by anyone: the task of writing code is delegated to graduate students. Programming is wonderful. The best job in the world is to be a computer programmer. Code can be as beautiful as the periodic table or Bach's Well Tempered Clavier. We hope to develop an aesthetics of programming; the aesthetics to guide your designs.

In this course we teach you to produce programming artifacts which can be used by many people. As a matter of fact, lots of code that we present is used by hundreds of thousands or even millions of people.

We use C++, but this is not a C++ course. This is a course about programming. To program one needs to use a real programming language and we chose C++. C++ has a wonderful machine model, inherited from C. It allows us to honestly look at the behavior of our algorithms. This machine model allows us to count how many instructions an algorithm generates in order to analyze its performance. If you need help with C++ we recommend the C++ standard [1].

Once upon a time programmers loved mathematics and knew it well. One needs only to look at the HAKMEM – a remarkable collection of MIT hacker's lore – to see how mathematically sophisticated early hackers were. Nowadays, we have programmers – even senior, principal and chief programmers – who are proud not to know high school mathematics. It is becoming fashionable to boast of being practical, with mathematics being viewed as academic mumbo-jumbo.

We believe that the separation of programming from mathematics is suicidal for programming. Mathematically illiterate people do not innovate. This course will attempt to present an integrated view of mathematics and Computer Science. We will not just use mathematics to analyze programs. We will use mathematics to derive programs. We will use mathematics to specify programs. We will use mathematics to optimize programs.

For Computer Science to be a true science it must be mathematical. Quoting from Dijkstra [2]:

*"As soon as programming emerges as a battle against unmastered complexity it is quite natural that one turns to that mental discipline whose main purpose has been since centuries to apply effective structuring to otherwise unmastered complexity. That mental discipline is more or less familiar to all of us, it is called Mathematics. If we take the existence of the impressive body of Mathematics as the experimental evidence for the opinion that for the human mind the mathematical method is indeed the most effective way to come to*

*grips with complexity, we have no choice any longer: we should reshape our field of programming in such a way that the mathematician's methods become equally applicable to our programming problems, for there are no other means."*

The point that mathematics is the only way is controversial. People say that this does not apply to software. But you could not possibly write software to build airplanes without mathematics. You cannot write software that does 3D graphics without it. You cannot write database software without mathematics. You cannot write a compiler without mathematics. What kind of programs are not mathematical? Programs that are insecure, programs that crash, programs that cannot be easily modified are examples of non-mathematical software. There is, of course, a solid economic reason for producing such programs: they assure world domination on a large scale and job security on a small scale.

People often confuse mathematics with formal methods. We contend that mathematics is not a science of formal structures. Quoting from Euler, "Mathematics, in general, is the*science of quantity*; or; the science which investigates the means of measuring quantity." Mathematics is a science of numbers and figures. Its task is to discover new truths about them. Formalization comes at the end of the creative period of any mathematical discipline. It is quite unfortunate that many computer scientists think of Category Theory as the part of mathematics that will help Computer Science. They should be reminded that one of the fathers of modern formalistic mathematics, André Weil, urged professors of mathematics to use Euler's *Introductio ad Analysin Infinitorum* instead of modern texts. Mathematical formalists want to derive mathematical theorems from first principles – Computer Science formalists want to derive programs from first principles. That is not how theorems or algorithms are discovered. Theorems and algorithms do not come from heaven to be proved by induction. They emerge from experimental, often flawed, transformations. Only after the community accepts them as true and useful, formalists develop their mechanisms. People were doing arithmetic long before Peano's axioms were published in 1889. And all of the essential theorems about Euclidean geometry were understood well before Hilbert's first rigorous set of axioms in 1898.

It is impossible to understand things unless we understand how they came to be. It is essential to show science as a human activity in its historical context. We have an old-fashioned view of history as a story. When we know a good story we tell it.

You won't need any books for mathematics beyond high school level – we develop what we need from algebra and number theory in the course. However we highly recommend George Chrystal's "Textbook of Algebra" [3].

While our bibliography contains many references and we occasionally recommend books and papers in the main text, there is one special book that pertains to most of our material: Knuth's *The Art of Computer Programming* [4]. Every computer scientist needs to acquire some fluency in Knuth. One does not need to know all of it by heart, but it is important to know what is there.

## 2. COMBINATORICS

2.1. **Blaise Pascal.** The man who invented combinatorics, was named Blaise Pascal (1623-1662). In the 17th century pre-science was transformed into modern science. The century "starts" with Kepler (or Gallileo even though he was a bit earlier). It ended with Newton (who lived well into the next century, but who was essentially a 17th century person). Prior to that, especially because of the Renaissance and the infatuation with Greeks, people still focused on how the ancients would think of something. But then there was a major scientific revolution, and one of the leading figures in this revolution was Pascal. By the end of the century the foundations of the modern scientific outlook on the world were firmly established.

France was a wonderful center of intellectual activity and the greatest European power. Paris was the “center of the world”. France prior to this time was going through a horrid religious war. Then at the end of the 16th and the beginning of the 17th century things stabilized under Henry IV, the first Bourbon King, and under his son Louis XIII. When you think of Pascal you might also think of his contemporaries, such as D’Artagnian, or cardinal Richelieu. This was the classical age of French literature, the age of Corneille, Moliere and Racine. It was then when the great revolution in mathematics, the move from synthetic geometry of Euclid to analytic geometry and analysis was initiated by Pascal’s contemporaries and compatriots Descartes and Fermat.

Pascal’s father was a prominent lawyer with novel ideas on childhood education: a child should not study mathematics until he is 15. He should first acquire full mastery of Greek and Latin, and only then start learning mathematics. So the father hid all of the mathematical books. Pascal was intrigued by the secrecy, and when he was 12 he came to his father and said that he just proved that the sum of the angles of a triangle is equal to two right angles. The father relented and let him read Euclid. When he turned 16, at which time they had moved to Paris, his father introduced him to Fr. Marin Mersenne who was the center of the newly emerging circle of European scientific intelligentsia. (Later on we will encounter him again when we learn about Mersenne Numbers and primality testing.) One day, while visiting Mersenne, Pascal brought a piece of paper and put it on the table without saying a word: on it were a number of amazing proofs of theorems in projective geometry.

His dad was appointed to Rouen to be a tax supervisor. To help his dad, Pascal decided to invent a mechanical computer. He actually built the first calculator. Well, some German invented such a calculator 10 years earlier, but no one had heard of it, so Pascal did his independently. It was a difficult thing to design since at the time the French were using as bizarre a currency system as the British. (In Britain you had to go from twelve pence to 1 shilling, 20 shillings to a pound. This system was abolished during the French Revolution – decimal currency system was introduced. Revolutions do some good, occasionally.) Then, when Pascal was 20, he started a business, made 600 of these machines – sold 50. Pascal invented the notion of vacuum. Prior to that everybody “knew” that vacuums could not exist – Aristotle said so and it was self-evident: nature abhors a vacuum. When Pascal tried to prove the existence of a vacuum to the greatest mind of the time, Rene Descartes, he wrote in one of his letters, “Pascal has a vacuum in his head”. Later Descartes claimed that he suggested the idea to Pascal – which was false. Then Pascal developed principle of hydrostatics – pressure propagates in any direction uniformly. Then Pascal’s triangle – we shall look at it shortly. He carried out a famous correspondence with Fermat where he invented probability theory. Not bad for a life of only 39 years. He was also the father of modern French prose. He joined the Jansenist – a 17th century version of traditionalist Catholics, rigorists who were more Catholic than the Pope – and wrote the Provincial Letters, a remarkable work of religious polemics still worth reading. A witty, devastating work, even while he is sometimes unfair to his opponents, the Jesuits. It is commonly considered to be a monument of French prose. His last unfinished work, which established him as one of the great philosophers and one of the founders of existentialism, was called Pensees. (In it he introduced a probabilistic argument for the existence of God know as Pascal’s wager.) Even his scientific treatises are worth reading – they are written in an elegant flowing style. His treatise on the (Pascal) triangle is especially wonderful.

2.2. **Permutations.** In 1654 Antoine Gombaud (a.k.a Chevalier de Méré) asked Pascal to help him figure out why he kept losing when gambling at a certain game of dice. Gombaud reasoned that since the odds of obtaining a pair of sixes when throwing two dice was $1/36$ it should follow that the chance of obtaining a pair of sixes after 24 throws should be 24 times as likely, or $24/36$. He couldn't understand why he kept losing money when wagering on this. This started Pascal's interest in combinatorics and probability. These subjects will also be important later in the book when we study algorithm performance. Pascal started looking at how things go together and how many of them there could be.

First we ask, given $n$ things, how many *permutations* of $k$ elements can there be. That is, suppose that we have $n = 5$ things. How many ways can we choose $k = 3$ elements, where we remain sensitive to order? Well we must begin by choosing a first element. There are $n$ possible ways to do so. Then we must select a second element form the $n - 1$ remaining. Finally we pick the third element from the $n-2$ that remain. Thus we answer that there are

$$n\,(n-1)\,(n-2) = 5 \cdot 4 \cdot 3 = 60 \text{ ways to do so.}$$

**Definition 2.1.** In general the number of permutations of $k$ things from $n$, respecting order is known as ${}_nP_k$.

It is easy to see that

$$\begin{aligned} {}_nP_1 &= n \\ {}_nP_2 &= n\,(n-1) \end{aligned}$$

and extrapolating

$${}_nP_k = n\,(n-1)\ldots(n-k+1)$$

It is sometimes useful to write it as

$${}_nP_k = \frac{n!}{(n-k)!} \tag{2.1}$$

despite the fact that it is not efficient for computation. A widely useful formula that we absolutely must know is:

**Theorem 2.2.** *Stirling's Approximation for $n!$*

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n. \tag{2.2}$$

Stirling's Approximation tells us how to get a value of $n!$ quickly. Please memorize Equation 2.2.. Remember that this is also equal to:

$$\sqrt{2\pi n}e^{n\ln n-n}$$

Why is this form important? When we study sorting we will see that it is related to permutations since $\lg n!$ bits of information are encoded in a permutation. This where the $n \lg n$ comes from in the complexity analysis of sorting functions. The exact number is $\lg n!$. For a proof of Stirling's formula see Chrystal [3] v. 2, p.368.

So far we have been talking about permutations, which are order sensitive. Now we will consider *combinations*, which are not order sensitive.

### 2.3. **Combinations.**

**Definition 2.3.** We define "*n* choose *k*": $\binom{n}{k}$ =the number of combinations of *k* elements out of *n* elements, disregarding order. Alternatively, it is the number of different *k* element subsets of size *n*. That is, we take all the permutations and factor out those with the same elements:

$$\binom{n}{k} = \frac{{}_nP_k}{{}_kP_k} = \frac{n!/\,(n-k)!}{k!} = \frac{n!}{k!\,(n-k)!} \tag{2.3}$$

Equation 2.3 is the canonical formula. It is not efficient for computation, so we don't use it to compute directly, but we will find several applications for it.

One great thing about Knuth is that he is willing to teach you which things need to be learned by heart. For example, he claims that you need to memorize 10! by heart (about 3.5 million). Also $\lg 1000 = 10$. These are important to remember, for quick mental conversions from the natural logarithm, ln, to the binary logarithm lg. Anyway, he also suggests memorizing:

$$\binom{n}{0} = 1, \binom{n}{1} = n, \binom{n}{2} = \frac{n(n-1)}{2}$$

You must know the above three facts by heart. You should also go on to memorize all of them up to and including $n = 4$, shown below with *n* on the vertical axis and *k* on the horizontal axis.

$$\binom{0}{0} = 1$$

$$\binom{1}{0} = 1, \binom{1}{1} = 1$$

$$\binom{2}{0} = 1, \binom{2}{1} = 2, \binom{2}{2} = 1$$

$$\binom{3}{0} = 1, \binom{3}{1} = 3, \binom{3}{2} = 3, \binom{3}{3} = 1$$

$$\binom{4}{0} = 1, \binom{4}{1} = 4, \binom{4}{2} = 6, \binom{4}{3} = 4, \binom{4}{3} = 1$$

The triangle formed by these coefficients is known as *Pascal's Triangle* ([3], vol 1, p.67) a portion of which is shown below:

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}$$

Entries in a given line of this table are constructed by adding the nearest pair of elements from the line above. The k-th entry in row n+1 gives the binomial coefficient $\binom{n}{k}$. Obtaining the binomial coefficients without carrying out the tedious multiplications was solved originally by the Chinese mathematician Yang Hui in the 13th century. This construct is known as Pascal's triangle everywhere in the world except in Italy (Italians call it Il Triangolo di Tartaglia, Tartaglia's triangle, and their terminology is historically more accurate since

Tartaglia discovered the triangle more than 100 years before Pascal). But it was Pascal whose name became permanently and rightly associated with this discovery. After all the mathematical community understood its fundamental importance through Pascal's work *Traite du Triangle Arithmetic* (Treatise on the Arithmetical Triangle – see for example [5]).

There are some simple mathematical facts that we need to remember. First: the law of symmetry – there as many ways as picking $k$ things from $n$ as there are of picking $n - k$ things. That is,

$$\binom{n}{k} = \binom{n}{n-k}$$

Also note that

$$\binom{n}{k} = \frac{n\,(n-1)\ldots(n-k+1)}{k\,(k-1)\ldots 1} = \frac{n}{k}\frac{n-1}{k-1}\ldots\frac{n-k+1}{1} = \frac{n}{k}\cdot\binom{n-1}{k-1}$$

This is very important. This also leads us to a remarkable divisibility rule telling us that $k!$ always divides $(n+1)\ldots(n+k)$ since

$$\frac{(n+1)\ldots(n+k)}{k!} = \binom{n+k}{k}$$

is a whole number. And of course, if we try we can also derive:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

(Try deriving it).

Now we ask how fast these grow. Well, the average of these "$n$ choose $k$" terms, otherwise known as *binomial coefficients*

$$\frac{\sum_{k=0}^{n}\binom{n}{k}}{n+1}$$

We don't need to do the sum – thinking works. We know the sum is equal to the size of the power set (the set of all subsets) so without doing the sum we know

$$\begin{aligned}\frac{\sum_{k=0}^{n}\binom{n}{k}}{n+1} &= \frac{1}{n+1}\left(\sum_{k=0}^{n}\text{the number of } k \text{ element subsets of a set of size } n\right)\\ &= \frac{1}{n+1}\,(\text{the total number of subsets of a set of size } n)\\ &= \frac{2^n}{n+1}\end{aligned}$$

One way to understand this is to recognize that each element of an $n$ element set corresponds to a bit in an $n$ bit word. So each subset corresponds to some $n$ bit number. Therefore the total number of subsets is the same as the total number of $n$ bit words, namely $2^n$. This also applies to binary numbers: What is the probability of 11 ones occurring in a 32 bit integer? It is

$$\frac{\text{The number of 11 element subsets of a set of size 32}}{\text{The total number of subsets of a set of size 32}} = \frac{\binom{32}{11}}{2^n}$$

2.3.1. *Implementing choose.* We now turn our attention to implementing the *choose* function. One approach is to make use of Pascal's triangle with a naive implementation of *choose(n,k):*

```
choose(n,k) = choose(n-1,k-1) + choose(n-1,k)
```

Unfortunately, this results in an exponential number of function calls. (Exercise: Why?)

A second idea is to use the factorial form of $\binom{n}{k}$, namely $\frac{n!}{k!(n-k)!}$.Actually, there is no reason to compute both $n!$ and $(n-k)!$. Instead we use the form:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{n!/k!}{(n-k)!} = \frac{n\,(n-1)\,...\,(n-k+1)}{(n-k)!}$$

At first this appears to suggest a linear time algorithm. But such an algorithm is not actually practical, due to overflow considerations. Further, the algorithm is actually quadratic, since the multiplying two numbers causes the length of the number to double. So we abandon these two obvious implementations and employ a generic approach. We will borrow the notion of iterators from later in the course and implement a quadratic algorithm using Pascal's triangle. The idea is to scan a line of the triangle (from some input iterator first to last) and output the next line (for some output iterator result). We want to illustrate the thought processes that goes into the design of algorithms One point worth noting is that we avoid top down design. Instead we focus on the key inner loop, and decide on the appropriate data structures later.

```
while(first != last){
  value_type new_value = *first;
  *result++ = old_value + new_value;
  old_value = new_value;
}
```

We don't know or care yet what value_type is. Having started with the inner loop we now need to step back and set up the appropriate initial conditions. This is very often the case. In very many algorithms you need to write a nice central loop, then it will tell you how to organize your messy initial conditions. We also need to decide where to increment first. Here is a more complete version of the code:

```
//Concepts: I Models InputIterator, O Models OutputIterator
template <typename I, typename O>
O add_adjacent_pairs(I first,
           I last,
           O result)
{
  typedef typename iterator_traits<I>::value_type
value_type;

  value_type old_value = 0;

  while (first != last) {
    value_type new_value = *first;
    *result = new_value + old_value;
    old_value = new_value;
    ++first;
    ++result;
  }
  return result;
}
```

**Exercise 2.4.** Implement a quadratic (e.g. $n * k$ operations) version of `choose(n, k)`. Test it for small values of $n$ and $k$.

2.3.2. *Another implementation of choose().* We can improve our implementation in the case when we will be calling `choose()` to calculate a single binomial coefficient instead of the whole row. We start with the fact that:

$$\binom{n}{k} = \frac{n\,(n-1)\,...\,(n-k+1)}{k\,(k-1)\,...1} = \frac{n}{k}\frac{n-1}{k-1}...\frac{n-k+1}{1} \tag{2.4}$$

This translates pretty directly into a recursive implementation:

```
//Concepts: T models Integer
template <typename T>
T choose_recursive(T n,    T k)
{
  if (n < k ) return T(0); //view elements out of the triangle as
0
  if (is_zero(k)) return T(1);
  return T((n * choose_recursive(predecessor(n), predecessor(k)))
/ k);
}
```

We turn this into an iterative version below (again using Equation 3 but here we multiply from right to left):

```
template <typename T>
T choose(T n, T k)
{
  if (n < k ) return T(0);
  n -= k;
  if (n < k) swap(n, k);
  T result(1);
  T i(0);
  while (i < k) {
    ++i;
    ++n;
    result *= n;
    result /= i;
  }
  return result;
}
```

Notice that in the code above we divide after multiplying, so there is some chance that we will suffer from overflow on the last time through the loop, even though the final result might be within range. It is possible to produce a version that squeezes out an extra bit of range at the cost of the calculation of the gcd (suggested to us by Mike Schuster):

```
//Concepts: T models integer
template <typename T>
```

```
T choose_extended_range(T n, T k)
{
  if (n < k ) return T(0);
  n -= k;
  if (n < k) swap(n, k);
  T result(1);
  T i(0);
  while (i < k) {
    ++i;
    ++n;
    T d(gcd(n, i));
    result /= (i/d);
    result *= (n/d);
  }
  return result;
}
```

Finally, in the case when $n$ is large , but not so large that the result can't be held in a `long double,` and when inexact results are acceptable, we can provide an implementation that makes use of Stirling's approximation ([3], vol II, p.368) to $n!$

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{2.5}$$

So that,

$$\binom{n}{k} = \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\sqrt{2\pi k}\left(\frac{k}{e}\right)^k \sqrt{2\pi (n-k)}\left(\frac{n-k}{e}\right)^{n-k}} = \sqrt{\frac{n}{2\pi k\,(n-k)}}\left(\frac{n-k}{k}\right)^k \left(\frac{n}{n-k}\right)^n$$

That is,

```
#define PI               3.14159265358979323846
long double choose_approximate(long double n, long double k)
{
  if (n < k ) return 0.0;
  return sqrt(n/(2 * PI * k * (n - k)))
         * pow((n - k)/k, k)
         * pow(n/(n - k), n);
}
```

2.4. **Binomial Theorem.** Newton was able to connect combinatorics with algebra. This was a marvelous insight. We will talk more about Newton later, but here we note in passing that he was the greatest physicist of all time, the inventor of calculus. It is less widely understood that Newton was one of the founders of modern algebra. This was not just because of the binomial theorem, but instead due to a number of results, not the least of which is his wonderful work concerning symmetric functions. His only subpar areas were theology – his numerological interpretations of the Apocalypse gives a chuckle to a modern Biblical scholar – and alchemy on which he, unfortunately, spent more time than on his Physics.

He observed this fundamental connection between algebra and combinatorics via an examination of:

$$(x + y)^n$$

He might have been the first one to look at it for arbitrary $n$.Pascal looked into this, and even Tartaglia knew the formula up to some point. Newton was the first person who considered this for an arbitrary $n$ and then connected it to the choose function.

$$(x+y)^n = (x+y)\,(x+y)\ldots(x+y)$$

If we multiply this out we can ask how many monomials of the form $x^i y^{n-i}$ we end up with. To answer this question, we see that for each parenthesized term, either $x$ or $y$ will make a contribution to the monomial. That is, the number of terms for which $x$ will participate $k$ times in the monomial = the number of subsets of size $k$ of $i$ things = $\binom{n}{i}$ = the coefficient of $x^i y^{n-i}$ in $(x+y)^n$. This provides us with the formula known as the

**Theorem 2.5.** *Binomial Theorem*

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i} \tag{2.6}$$

**Exercise 2.6.** What is the expected distribution of bytes with $k$ 1 bits turned on in a file? Write a filter (small console application) that will take a real file and compare the distribution to the expected distribution

### 2.5. **Derangements.**

**Definition 2.7.** A *derangement* is a permutation that does not leave any element in place.

A remarkable result regarding derangements is given below. It is commonly believed that this result was first introduced by Euler in 1753. Indeed, he did wrote a very beautiful paper about it then while in Berlin, where there was a strong interest in the card game Rencontre. The game requires two people and two decks of cards. Each person is given one deck. Each player places a card from their deck on the table. If the cards have the same value then player A wins. If both players go through the whole deck then player B wins. Euler's paper explained the chances of each player winning the game. But in this case Euler was not actually the first to explain matters. Nor was it de Moivre. In 1708 Pierre Rémond de Montmort published a result about the odds in the game of Treize (Treize means thirteen in French). In Treize there is a single deck and everybody bets against the dealer. For simplicity's sake we can describe the rules as follows. The dealer calls out "Ace". Then he turns places the top card on the table. If the card is in fact an Ace then he wins. If not, the play continues by the dealer calling out "two" and then turning over another card. If it is a two he wins otherwise play continues as above through King. De Montmort published a 500 page book on the theory of card games. The first edition contained no proofs, but the second edition (1713) did (probably done by Nicolas Bernoulli). We demonstrate how to understand the chances involved in both of these games. This amounts to proving the following theorem.

**Theorem 2.8.** *The number of derangements of n elements is:*

$$D_n = n! \sum_{i=0}^{n} (-1)^i \frac{1}{i!} \approx \frac{n!}{e}. \tag{2.7}$$

Now the dealer only loses in our simplified version of Treize when the way the deck was shuffled (permuted) happens to be a derangement of (A 2 3 4 ... K). This tells us the probability that the dealer will lose in Treize is the number of derangements over the total number of cards played:

$$\frac{D_n}{{}_nP_n} \approx \frac{\frac{n!}{e}}{n!} = \frac{1}{e} \approx 36\%$$

This is surprising since it doesn't depend on $n$ for the most part. Before we prove the above we will explain and prove the very important combinatorial Theorem, known as the *Inclusion-Exclusion Principle.* To get a feel for the principle we consider the following examples. First, suppose there are some people in a room, of which 16 are blond and 19 are tall.. We put the question: how many are blond or tall. The answer is at most the number of blond people + the number of tall people = 16 + 19 = 35. To obtain the actual answer, we must avoid counting twice those who happen to be both blond and tall. That is, the number who are blonde or tall is equal to the number who are blonde plus the number who are tall minus the number who are "blonde and tall". We can also state this mathematically. The *cardinality* (number of elements) of a set $A$ is written as $|A|$. Our example becomes: given finite sets $A_1$ and $A_2$, $|A_1 \cup A_2| = |A_1| + |A_2| - |A_1 \cap A_2|$. Now let us further assume that there are 11 fat people in the room and we ask how many are blond or tall or fat. Again we see the maximum answer is 16+19+11. But to obtain the correct answer this time we must compensate further. Let $A_1$ be the set of blond people and $A_2$ be the set of tall people and $A_3$ be the set of fat people. Then our initial estimate of $|A_1 \cup A_2 \cup A_3|$ is $|A_1| + |A_2| + |A_3|$. But then we must subtract the number who are blond and tall, blond and fat, or tall and fat to get

$$|A_1| + |A_2| + |A_3| - |A_1 \cap A_2| - |A_1 \cap A_3| - |A_2 \cap A_3|.$$

But we are still not quite done, because we have eliminated the number of people who are simultaneously blond and tall and fat three times, so that they are now not counted at all. Adding them back in we get:

$$|A_1 \cup A_2 \cup A_3| = |A_1| + |A_2| + |A_3| - |A_1 \cap A_2| - |A_1 \cap A_3| - |A_2 \cap A_3| + |A_1 \cap A_2 \cap A_3|.$$

We suggest drawing a Venn diagram to see this more clearly. In general then, to count the number of elements in the set $A_1 \cup A_2 \cup \ldots \cup A_n$, we first add the cardinality of each $A_i$, then subtract the cardinalities of all possible intersections of two of the sets, add the cardinalities of all possible intersections of three sets, and so on.

**Theorem 2.9.** *(Inclusion/Exclusion Principle).* Let $A_1, A_2, ..., A_n$ *be finite sets. Then*
(2.8)

$$|\bigcup_{i=1}^{n} A_i| = \sum_{i=1}^{n} |A_i| - \sum_{1 \le i < j \le n} |A_i \cap A_j| + \sum_{1 \le i < j < k \le n} |A_i \cap A_j \cap A_k| - ... + (-1)^{n-1} |A_1 \cap A_2 \cap \ldots \cap A_n|.$$

*(Note that there is no summation before the last term since there is only one possible intersection of nelements.)*

*Proof.* If $x$ is in at least one of the $A_i$'s then it will contribute 1 to the left hand side of Equation 2.8. We must show that it will also contribute exactly 1 to the right hand side. Suppose that $x$ is in exactly $k$ of the $A_i$'s. Then $x$ will contribute $k = \binom{k}{1}$ to the first term of the right hand side, minus one for each intersection that contains $x$ to the second term i.e. $-\binom{k}{2}$, and $\binom{k}{3}$ to the third term and so on. So $x$'s overall contribution on the right hand side will be

$$\binom{k}{1} - \binom{k}{2} + \binom{k}{3} + \ldots + \binom{k}{k}(-1)^{k-1} = \sum_{i=1}^{k} \binom{k}{i}(-1)^{i-1}.$$

Now, with the help of the Binomial Theorem we can see that

$$0 = (1-1)^k = \sum_{i=0}^{k} \binom{k}{i}(-1)^i = \binom{k}{0} + \sum_{i=1}^{k} \binom{k}{i}(-1)^i = 1 + \sum_{i=1}^{k} \binom{k}{i}(-1)^i = 1 - \sum_{i=1}^{k} \binom{k}{i}(-1)^{i-1}.$$

Therefore $x$'s overall contribution to the right hand side is

$$\sum_{i=1}^{k} \binom{k}{i} (-1)^{i-1} = 1.$$

□

Now we are ready to prove the derangement formula Theorem 2.8.

*Proof.* (of derangement formula). Let $A_i$ denote the set of permutations of $n$ elements that leaves the $i$-th element fixed. Of course these sets are not necessarily disjoint. Since there are $n!$ permutations of $n$ elements, it is easy to see that for each $A_i$, $|A_i| = (n-1)!$. Also, $|A_i \cap A_j| = (n-2)!$ since $A_i \cap A_j$ is simply the collection of permutations fixing two elements. By the same reasoning, we see that in the general case, the cardinality of the intersection of $k$ disjoint $A_i$'s will be $(n-k)!$. Furthermore, we know that there are $\binom{n}{k}$ such intersections, so by Theorem 2.9 we have

$$\begin{aligned}
|\bigcup_{i=1}^{n} A_i| &= \sum_{k=1}^{n} (-1)^{k+1} \binom{n}{k} (n-k)! \\
&= \sum_{k=1}^{n} (-1)^{k+1} \frac{n!}{k!\,(n-k)!} (n-k)! \\
&= \sum_{k=1}^{n} (-1)^{k+1} \frac{n!}{k!} \\
&= n! \sum_{k=1}^{n} (-1)^{k+1} \frac{1}{k!} \\
&= n! \left( 1 + \sum_{k=0}^{n} (-1)^{k+1} \frac{1}{k!} \right) \\
&= n! \left( 1 - \sum_{k=0}^{n} (-1)^{k} \frac{1}{k!} \right)
\end{aligned}$$

Now the number of derangements of $n$ of a finite set of $n$ elements is equal to the total number of permutations, $n!$ minus the number of permutations that leave some element fixed. That is

$$D_n = n! - |\bigcup_{i=1}^{n} A_i| \approx n! - n! \left( 1 - \sum_{k=0}^{n} (-1)^k \frac{1}{k!} \right) = n! \sum_{k=0}^{n} (-1)^k \frac{1}{k!} \tag{2.9}$$

Now, the Taylor formula for $e^x = \sum_{i=0}^{\infty} \frac{x^n}{i!}$. Notice that it is easily seen that it is equal to its own derivative. Substituting $x = -1$ gives $\frac{1}{e} = \sum_{i=0}^{\infty} (-1)^i \frac{1}{i!}$. Furthermore, this alternate series converges very quickly, the error being bounded by the first omitted term, so we have:

$$\sum_{i=0}^{n} (-1)^i \frac{1}{i!} \quad \approx \frac{1}{e}$$

and we can also rewrite Equation 2.9 as:

$$D_n \approx \frac{n!}{e}.$$

□

**Exercise 2.10.** Write a function *is_derangement* which takes a sequence of the integers from 1 to $n$ in some order and tells you whether it is a derangement or not. For example, if the sequence is (1 0 2) it is not a derangement (since 2 is a fixed point).

**Exercise 2.11.** Verify the above formula for the number of derangements up to some $n$ using the STL function *next_permutation* and the *is_derangement* function from the previous exercise.

**Exercise 2.12.** Using the STL function called *random_shuffle*, try to generate $e$. That is, shuffle the sequence 1 2 ... $n$ a number of times. Each time check whether the permutation is a derangement. Then calculate the ratio of the number of derangements over the number of tries, and use the derangement formula to estimate $e$. This is not very fast, but it is an interesting way of generating $e$. This was the first combinatorial problem to illustrate the profound connection between probability theory and $e^x$.

**Exercise 2.13.** Figure out a way to generate $\pi$. Hint: Assume that you could get uniformly distributed set of points from this square using random number for each coordinate. For a given point, the probability that it is more within 1 of the origin is $\frac{\pi}{4}$.

2.6. **Harmonic Series.** We will consider the performance of the following algorithm, *max_element:*

```
template <class I> //I models ForwardIterator
I max_element(I first, I last)
{
  if (first == last) return first;
  I result = first;
  while (++first != last)
    if (*result < *first) result = first;
  return result;
}
```

There is no way to avoid performing $n - 1$ comparisons for a range of length $n$, since each comparison eliminates one element from the set of candidates for max. Notice that we return the position of the maximal element so as not to throw away a possibly useful computation, not to mention that this gives a something to return when the range is empty. Given that we know the number of comparisons required, we will focus on the average number of assignments to *result*.

If a sequence happens to be sorted in ascending order the implementation above will perform $n - 1$ assignments. But the odds of that occurring or $1/n!$. In statistics this problem is known as the record problem: if we keep track of the weather each year for 100 years, how often will there be a year with a record high temperature (assuming a random distribution of temperatures)? For a uniformly distributed range of elements, the likelihood that the $i$th element is the largest (of the first $i$ elements) is $1/i$. Indeed, there are $i$ places where it could reside, all of them equally likely. So at the $n$th step we will expect to have performed, on average,

$$H_n = \sum_{i=1}^{n} 1/i$$

assignments to *result*.

$H_n$ is known as the $n$th *harmonic number,* and for $n = \infty$ the series is called the *harmonic series.* The term *harmonic* goes back to Pythagoras, who was studying mathematics and music. In considering the mathematical structure of the seven tone scale, he observed that the sounds made by a C followed by a G that is one fifth lower, followed by the next lower C, sounded harmonious. The relative string lengths needed to produce those notes are 1, 2/3, 1/2. Then he realized that if he inverted each of the fractions in that series he would end up with an arithmetic series 1, 3/2, 2. In this case, 3/2 is the arithmetic mean of 1 and 2. Pythagoras came up with the idea of defining, for a given $A$ and $B$

$$\frac{1}{C} = \frac{1}{2}\left(\frac{1}{A} + \frac{1}{B}\right)$$

which works out to give the *harmonic mean, C,* of two numbers $A$ and $B$

$$C = \frac{2\,(AB)}{A + B}$$

**Exercise 2.14.** A car is driving from point $x$ to point $y$ and back. The average velocity of the car when traveling from $x$ to $y$ is $v_1$, and the average velocity on the return trip is $v_2$. Show that the average velocity over the whole trip is equal to the harmonic mean of $v_1$ and $v_2$.

**Exercise 2.15.** The harmonic series consists of the reciprocals of an arithmetic series. Prove that any term is the harmonic mean of its two neighbors, that is, show that for $i > 1$, $\frac{1}{i}$ is the harmonic mean of $\frac{1}{i-1}$ and $\frac{1}{i+1}$.

Recall that the harmonic series does not converge. The divergence of the harmonic series was proved by the French scholastic philosopher and theologian Nicole Oresme (ca. 1323-1382). He was one of the great men from the University of Paris (Sorbonne), the leading educational institution in the middle ages. Founded in 1258, within fifty years the Sorbonne became the center of European learning. In the first 20 years its professors included Thomas Aquinas, Bonaventure, and slightly later on John Duns Scotus. Oresme was the first person, after the Greeks, to suggest and argue convincingly that the solar system is heliocentric. He even offered sound theological arguments that the Bible is not the place to look for natural laws – these arguments could have saved the church much trouble later on. His opinion was later restated by Galileo in the famous words, "the Bible doesn't tell us how the heavens go, but how to go to heaven."

Oresme's proof of divergence is quite short. Consider the harmonic series with terms grouped as follows:

$$1 + 1/2 + (1/3 + 1/4) + (1/5 + 1/6 + 1/7 + 1/8) + ...$$

Each parenthesized group of terms has sum greater than 1/2. Since there are an unbounded number of groups we conclude that the harmonic series diverges. Each parenthesized group of terms has sum less than 1 and the $i$th parenthesized group of terms has $2^i$ terms, so we conclude that the growth of $H_n$ is logarithmic since it takes at least $2^n$ additional terms to increase $H$ by 1.

There is a formula for the sum of the first $n$ terms of the harmonic series

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + O\left(\frac{1}{n}\right) \tag{2.10}$$

where $\gamma \approx .57721566$ is the *Euler constant* (sometimes known as the*Euler-Mascheroni* constant). We postulate that the probability that a result in Analysis was first discovered by Euler is $1/e$.

**Exercise 2.16.** Approximate the value of $\gamma$ using a simulated deck of cards. Make use of the fact that, on average, we expect $H_n$ assignments to *result* in *max_element.* Then solve for $\gamma$ in making use of Equation 2.10. It's ok to use the *log* standard library function. This exercise will give you some idea of how $H_n$ slowly grows.

**Exercise 2.17.** Try to compute the harmonic series on your computer. That is, keep computing $1/i$ until you find an $i$ that so that $1/i$ yields 0. See how big $H_i$ is. You might want to use a 32 bit integer for $i$, to speed up your computations.

The harmonic series is part of a general family of series, given by the *Riemann zeta function*

$$\zeta(s) = \sum_{i=1}^{\infty} \frac{1}{i^s}.$$

Euler discovered that, for real numbers $s > 1$, the series converges. To see this we use the following grouping

$$1 + \left(\frac{1}{2^s} + \frac{1}{3^s}\right) + \left(\frac{1}{4^s} + \frac{1}{5^s} + \frac{1}{6^s} + \frac{1}{7^s}\right) + ...$$

Notice that the $i$th group, containing $2^i$ terms, has a sum bounded above by

$$2^i \times \frac{1}{(2^i)^s} = \left(\frac{1}{2^s}\right)^{i-1}$$

Thus the sum of our original series is bounded above, i.e.

$$\sum_{i=1}^{\infty} \frac{1}{i^s} \leq \sum_{i=1}^{\infty} \left(\frac{1}{2^s}\right)^{i-1}.$$

The right hand series is geometric with ratio $1/2^{s-i}$ so we have convergence when $s > 1$. (See Chrystal [3] v.2, 120-127). Euler also demonstrated that

$$\sum_{i=1}^{n} \frac{1}{p_i} = \frac{1}{2} + \frac{1}{3} + \frac{1}{5} + ...$$

where $p_i$ is the $i$th prime number. This diverges very slowly – it practically converges. The above sum is approximately $\ln(\ln n)$.

## 3. Experimental Science

### 3.1. A simple example.

Many algorithms turn out to be inductive in nature, so we begin here by recalling the

**Axiom 3.1.** Axiom of Induction

Let there be given a set $P$ of natural numbers with the following properties:

(1) 0 belongs to $P$
(2) if $n$ belongs to $P$ then so does $n + 1$

Then $P$ contains all the natural numbers.

Often we choose to think of $P$ as a property of natural numbers. That is, if $n$ is a natural number, then for we say that the property $P$ is true for $n$, or more succinctly, $P(n)$ when $n \in P$. Then another way to state the Axiom of Induction is that in order to verify that a given property holds for all natural numbers we must carry out two proof obligations:

(1) The *base step* is to verify that $P(0)$ is true
(2) The *inductive step* requires us to verify the following: for any $n$, given that we can assume that $P(n)$is true we must be able to then prove that $P(n + 1)$is true. The assumption that $P(n)$ is true is known as the *inductive hypothesis*

**Example 3.2.** We wish to show that:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{3.1}$$

We will prove this identity using mathematical induction. To do so we first verify that the base step holds. That is we must show that:

$$\sum_{i=0}^{0} i = \frac{0\,(0+1)}{2} \tag{3.2}$$

This is straightforward since the left and right side of the above equation are easily seen to be equal to 0 and so are equal to each other. For our second proof obligation we may assume that

$$\sum_{i=0}^{j-1} i = \frac{(j-1)\,j}{2} \tag{3.3}$$

and from this we must show that

$$\sum_{i=0}^{j} i = \frac{j(j+1)}{2}$$

But

$$\sum_{i=0}^{j} i = j + \sum_{i=0}^{j-1} i \tag{3.4}$$

By the inductive hypothesis and Equation 3.3 the right hand side can be rewritten as:

$$j + \sum_{i=0}^{j-1} i = j + \frac{(j-1)\,j}{2} \tag{3.5}$$

Combining this with Equation 3.4 we get:

$$\sum_{i=0}^{j} i = j + \sum_{i=0}^{j-1} i = j + \frac{(j-1)\,(j)}{2} = \frac{2j}{2} + \frac{j^2 - j}{2} = \frac{2j + j^2 - j}{2} = \frac{j + j^2}{2} = \frac{j(j+1)}{2} \tag{3.6}$$

Now our second proof obligation is complete. We have shown that the base step holds and that the inductive step holds, and so by induction we may conclude that the formula 3.1 holds for all natural numbers.

3.2. **Mathematics is an experimental science.** The above section may seem like a pretty familiar treatment of induction. But we claim that there is something profoundly unsatisfying here. It is true that induction lets us verify that a given formula holds. The problem with the way that mathematics is taught right now is that this method gives no indication of where the formula came from. You have to somehow know the solution to prove anything. If you don't know the solution then induction doesn't help. Knuth [4] on page 32 of volume 1 states:

*"Induction is of course a perfectly valid procedure, but it does not give any insight into how on earth a person could ever have dreamed the formula up in the first place except by some lucky guess."*

Mathematics is not about proving. It is about discovery. It is an experimental science. Gauss is a wonderful example of an experimental mathematician. One of his major accomplishments is the prime number theorem which establishes a function which counts the number of primes less than some integer $n$. Gauss somehow guessed, at age 15, that the number of primes less than $n$ is approximated by $\frac{n}{\ln n}$. The proof of this was quite difficult and didn't appear until 100 years later (it was proved independently by Hadamard and de la Vallée Poussin in 1896). How did Gauss come up with the prime counting function? Apparently for years, when he had spare time, he would pick a random range of a thousand numbers, then he would calculate the number of primes in that range. Over time he accumulated the density of primes in different 1000 number ranges of primes, then he would plot the results. Eventually he realized that the curve looked very much like $\frac{n}{\ln n}$. But he couldn't prove it.

There is a second story about Gauss that is relevant here. Here is one version. At the age of 10 Gauss was a show-off in arithmetic class at St Catherine elementary school. One day his teacher, one Buttner, entered the room and asked the boys to find the sum of the whole numbers from 1 to 100. After a few seconds Gauss said "I have the answer". Buttner laughed but in a blink Gauss pronounced: 5050. His teacher was astonished. What was Gauss's trick? In his mind he apparently pictured writing the summation sequence twice forward and backward, one sequence above the other

$$\begin{array}{ccccc} 1 & 2 & ... & 99 & 100 \\ 100 & 99 & ... & 2 & 1 \end{array}$$

Gauss realized that you could add the numbers vertically instead of horizontally. There are 100 vertical pairs each summing to 101. So the answer is 100 x 101 divided by 2 since each number is counted twice... and easily in his head he got the right answer $\frac{10100}{2}$=5050

3.3. **Some non-inductive proofs.** Let us revisit the problem of summing the first $n$ numbers, and others like it. Instead of verifying formulas given from out of the blue, we will try to motivate how one could generate such formulas. Interest in this sort of problems goes back to a time far before Gauss was born. The ancient Greeks were very interested in numbers and their properties.

A number which can be represented by a regular geometrical arrangement of equally spaced points is known as a figured (or figurate) number [6]. If the arrangement forms a regular polygon, the number is called a polygonal number. The Greeks were very interested in certain polygonal numbers, such as square numbers:

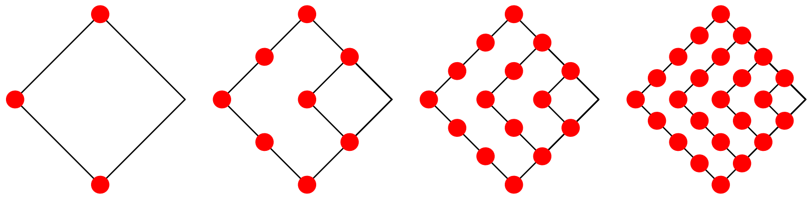


FIGURE 1. The first 5 square numbers: $1^2, 2^2 = 4, 3^2 = 9, 4^2 = 16, 5^2 = 25$

and triangular numbers:

In fact (a centered version of the) pattern for the triangular number 10 was known as the sacred "tetractys" (literally: "group of four things") the symbol that the Pythagoreans used as their emblem for the essence of the universe. In any case, we note that the triangular

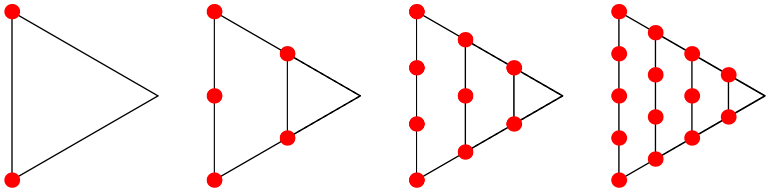


FIGURE 2. The first five triangular numbers: 1, 3, 6, 10, 15

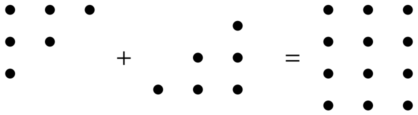
number of height $n$ corresponds to the sum:

$$\sum_{i=0}^{n} i \tag{3.7}$$

e.g., looking at the base of the image corresponding to the triangular number 10 above we see that

$$10 = 1 + 2 + 3 + 4 = \sum_{i=0}^{4} i$$

We note that it is possible to place a triangular number of height $n$ on top of another of the same size (after rotating by 180 degrees) to obtain a rectangle of dimensions $n$ by$(n + 1)$> For example, when $n = 6$ we have



The numbers so obtained were known to the Greeks as an oblong (also called pronic) numbers. From this we can see geometrically that the .

$$\begin{aligned} &\text{number of points in triangular number of height } n = \\ &\qquad \text{half the number in the oblong number of dimensions } n \text{ by } (n+1) \\ &\qquad = \frac{n(n+1)}{2} \end{aligned} \tag{3.8}$$

The two different accountings for the number of points in a triangular number given in Equation 3.7 and 3.8 serve as a geometric proof that

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

We find this method of proof to be a great deal more satisfying then the inductive verification of an arbitrary formula given at the start of this section. We started with some elementary geometric facts and put them together to come up with the formula. This is reminiscent of a the way that a Russian mathematician Vladimir Arnold prefers to demonstrate commutativity of multiplication [7] :

"*Attempts to create "pure" deductive-axiomatic mathematics have led to the rejection of the scheme used in physics (observation - model - investigation of the model - conclusions - testing by observations) and its substitution by the scheme: definition - theorem - proof. It is impossible to understand an unmotivated definition but this does not stop the criminal algebraists-axiomatisators. For example, they would readily define the product of natural* *numbers by means of the long multiplication rule. With this the commutativity of multiplication becomes difficult to prove but it is still possible to deduce it as a theorem from the axioms. It is then possible to force poor students to learn this theorem and its proof (with the aim of raising the standing of both the science and the persons teaching it). It is obvious that such definitions and such proofs can only harm the teaching and practical work.*

*It is only possible to understand the commutativity of multiplication by counting and re-counting soldiers by ranks and files or by calculating the area of a rectangle in the two ways. Any attempt to do without this interference by physics and reality into mathematics is sectarianism and isolationism which destroy the image of mathematics as a useful human activity in the eyes of all sensible people."*

There is another very important kind of number, closely related to square numbers. Figurate numbers of the form, $2i + 1$ are known to us as odd numbers and to the Greeks as Gnomonic Numbers. (Gnomon means carpenter's square in Greek). The first five Gnomonic numbers are pictured below:

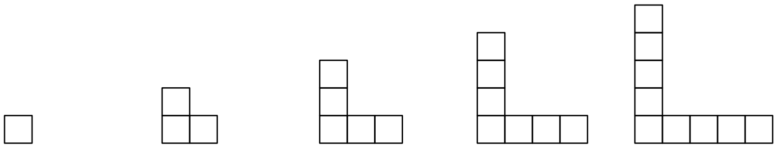


FIGURE 3. The first five Gnomonic numbers – 1, 3, 5, 7 and 9

An interesting relation involving the Gnomonic numbers is illustrated below. We demonstrate that a given square number is in fact composed of a succession of Gnomonic numbers (or, if you prefer, nested carpenter's squares) for the case $1 + 3 + 5 + 7 = 4^2$: We state the result analytically as:

$$\sum_{i=0}^{n} (2i + 1) = (n + 1)^2 . \tag{3.9}$$

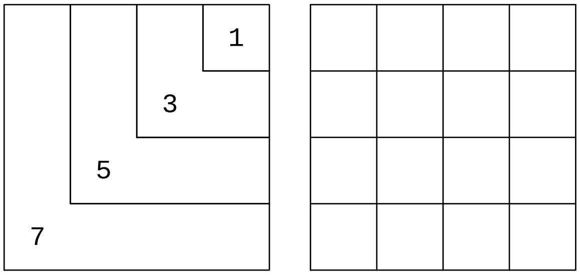


FIGURE 4. The sum of the first five Gnomonic numbers is a square number

3.4. **Sums of powers.**

3.4.1. *Generating a conjecture.* We now demonstrate this approach towards mathematics by attempting to generalize the formula:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{3.10}$$

We might try to generalize to summation of arbitrary linear functions, $ai + b$. But that would be too easy. It would be much more useful and impressive to generalize Equation 3.10 so that we could determine the sum for an arbitrary, say, $k^{\text{th}}$ degree polynomial. That is, we will aim to *discover* formulas for sums of the form:

$$\sum_{i=0}^{n} p_k(i), \text{where } p_k(x) = a_k x^k + a_{k-1}x^{k-1} + ... + a_1 x + a_0 \tag{3.11}$$

How do we begin? The first insight will come from revisiting the arguments concerning the triangular numbers above in an analytic manner. There is an important point worth noting here. Even though we already proved our result, we come back to look at the problem from a different angle. Often this is the route to real understanding. If we view the n-th triangular number as a step function we can graph it in the xy plane, (try this for the 5th triangular number). The area under this graph corresponds to the value of the n-th triangular number as given in Equation 3, i.e. $\sum_{i=0}^{n} i$, where in the example above $n = 5$. It is straightforward to see that another way of obtaining the area under such a graph is to calculate area under the line that passes through the midpoints of the steps with equation:

$$y = x + \frac{1}{2} \tag{3.12}$$

Using a little bit of basic calculus we see that the latter area is of the form:

$$\int x + \frac{1}{2}dx = \frac{x^2}{2} + \frac{x}{2} = \frac{x(x+1)}{2}$$

This gives us a hint. Maybe the sum of an arbitrary polynomial as in Equation 3 is like integration. So we can generate a conjecture:

$$\sum_{i=0}^{n} p_k(i) = q_{k+1}(n), \text{where } q_{k+1}(n) = b_0 + b_1 n + ... + b_{k+1}n^{k+1}$$

i.e. $q_{k+1}(n)$ is a polynomial of degree $k + 1$ in $n$. This very nice, because even if we can't prove that the summation is equal to the integral, we still have generated a conjecture. Though induction was not useful for idea generation, we can later forget all about the presumed connection with calculus and use induction to verify our conjecture.

3.4.2. *An example.* To help illustrate these ideas we now work an example. As a special case of summing over a $k^{\text{th}}$ degree polynomial we consider the sum:

$$\sum_{i=0}^{n} i^7 \tag{3.13}$$

We conjectured above that the solution would be of the form:

$$\sum_{i=0}^{8} b_i n^i = b_0 + b_1 n + ... + b_7 n^7 + b_8 n^8$$

In the case when $n = 0$the sum in 3.13 must be 0, so we have the equation:

$$\sum_{i=0}^{0} i^7 = 0 = \sum_{i=0}^{8} b_i 0^i = b_0 \tag{3.14}$$

When $n = 1$the sum is 1 so we have a second equation:

$$\sum_{i=0}^{1} i^7 = 1^7 = 1 = \sum_{i=0}^{8} b_i 1^i = b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 + b_8 \tag{3.15}$$

We can continue setting $n$ to successive values in the left and right hand sums to obtain 9 equations involving out 9 unknowns $\{b_0, ..., b_8\}$. In this case we have:

$$\begin{aligned}
\sum_{i=0}^{0} i^7 &= 0 = b_0\\
\sum_{i=0}^{1} i^7 &= 1 = b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 + b_8\\
\sum_{i=0}^{2} i^7 &= 129 = b_0 + 2b_1 + 4b_2 + 8b_3 + 16b_4 + 32b_5 + 64b_6 + 128b_7 + 256b_8\\
\sum_{i=0}^{3} i^7 &= 2316 = b_0 + 3b_1 + 9b_2 + 27b_3 + 81b_4 + 243b_5 + 729b_6 + 2187b_7 + 6561b_8\\
\sum_{i=0}^{4} i^7 &= 18700 = b_0 + 4b_1 + 16b_2 + 64b_3 + 256b_4 + 1024b_5 + 4096b_6 + 16384b_7 + 65536b_8\\
\sum_{i=0}^{5} i^7 &= 96825 = b_0 + 5b_1 + 25b_2 + 125b_3 + 625b_4 + 3125b_5 + 15625b_6 + 78125b_7 + 390625b_8\\
\sum_{i=0}^{6} i^7 &= 376761 = b_0 + 6b_1 + 36b_2 + 216b_3 + 1296b_4 + 7776b_5 + 46656b_6 + 279936b_7 + 1679616b_8\\
\sum_{i=0}^{7} i^7 &= 1200304 = b_0 + 7b_1 + 49b_2 + 343b_3 + 2401b_4 + 16807b_5 + 117649b_6 + 823543b_7 + 5764801b_8\\
\sum_{i=0}^{8} i^7 &= 3297456 = b_0 + 8b_1 + 64b_2 + 512b_3 + 4096b_4 + 32768b_5 + 262144b_6 + 2097152b_7 + 16777216b_8
\end{aligned}$$

Then we can solve for the whole system by solving the following matrix equation:

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\
1 & 2 & 4 & 8 & 16 & 32 & 64 & 128 & 256\\
1 & 3 & 9 & 27 & 81 & 243 & 729 & 2187 & 6561\\
1 & 4 & 16 & 64 & 256 & 1024 & 4096 & 16384 & 65536\\
1 & 5 & 25 & 125 & 625 & 3125 & 15625 & 78125 & 390625\\
1 & 6 & 36 & 216 & 1296 & 7776 & 46656 & 279936 & 1679616\\
1 & 7 & 49 & 343 & 2401 & 16807 & 117649 & 823543 & 5764801\\
1 & 8 & 64 & 512 & 4096 & 32768 & 262144 & 2097152 & 16777216
\end{pmatrix} \cdot \begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7\\ b_8 \end{pmatrix} = \begin{pmatrix} 0\\ 1\\ 129\\ 2316\\ 18700\\ 96825\\ 376761\\ 1200304\\ 3297456 \end{pmatrix}$$

We omit the details (see [3] v. I, p. 374) of the straightforward solution of this system of equations which results in:

$$\sum_{i=0}^{n} i^7 = \frac{2n^2 - 7n^4 + 14n^6 + 12n^7 + 3n^8}{24} = \frac{n^2\,(1+n)^2\left(2 - 4n - n^2 + 6n^3 + 3n^4\right)}{24}$$

So in this example our conjecture becomes:

$$\sum_{i=0}^{n} i^7 = \frac{n^2\,(1+n)^2\left(2 - 4n - n^2 + 6n^3 + 3n^4\right)}{24} \tag{3.16}$$

At last we have use for induction to check if our conjecture in fact holds. First we verify the base step:

$$\sum_{i=0}^{0} i^7 = 0 = \frac{0\,(1+0)^2\,(2-0-0+0+0)}{24}$$

Next we must verify the inductive step, that is, if we are allowed to assume that

$$\sum_{i=0}^{n} i^7 = \frac{n^2\,(1+n)^2\left(2 - 4n - n^2 + 6n^3 + 3n^4\right)}{24}$$

holds for some n ≥ 0 we must show that

$$\sum_{i=0}^{n+1} i^7 = \frac{(1+n)^2\,(2+n)^2\left(2 - 4\,(1+n) - (1+n)^2 + 6\,(1+n)^3 + 3\,(1+n)^4\right)}{24} \tag{3.17}$$

This is rather straightforward since

$$\sum_{i=0}^{n+1} i^7 = (n+1)^7 + \sum_{i=0}^{n} i^7 = (n+1)^7 + \frac{n^2\,(1+n)^2\left(2 - 4n - n^2 + 6n^3 + 3n^4\right)}{24} \tag{3.18}$$

by the inductive hypothesis. Continuing, we can fully expand Equation 3.18 to:

$$\frac{24 + 168n + 506n^2 + 840n^3 + 833n^4 + 504n^5 + 182n^6 + 36n^7 + 3n^8}{24} \tag{3.19}$$

At the same time we expand Equation 3.17:

$$\frac{24 + 168n + 506n^2 + 840n^3 + 833n^4 + 504n^5 + 182n^6 + 36n^7 + 3n^8}{24} \tag{3.20}$$

Since the expression in 3.19 is identical to that in 3.20 the inductive step has been verified and so we are now satisfied that formula in 3.16 is in fact correct.

Let us briefly recap. First, we observed a connection between integration and summation. This led us to conjecture that the solution to a summation of a k-th degree polynomial was in fact a polynomial of degree k+1. A lengthy but straightforward calculation to solve the system of linear equations generated by plugging in a sufficient number of values led to a candidate solution in 3.16. Another lengthy but straightforward set of calculations allowed us to verify the correctness of our solution by mathematical induction. If we had chosen to sum over a polynomial other than $i^7$we would have ended up with a different set of straightforward calculations. The important point here is to observe the beauty and power of the single idea – the connection between sums and integration – and how it generates a whole family of conjectures, any of which can be verified by induction. This is also a useful technique in general, and very often is faster than the general solution that we are about to explore.

3.4.3. *Another derivation of $\sum i$.* It is a well kept secret that often, to understand the general case one begins with the most trivial case possible to gain some insight. Here we change gears, setting aside the techniques we have just seen to try a new approach. In this case we begin with the Gnomon formula that can be seen from the proof 3.9, namely

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

This is the most trivial formula relating a square with a non-square. Now, we go "down" by 1, leveraging the identity above :

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

and once again:

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

We can continue in this manner until we hit the "bottom":

$$1^2 = 0^2 + 0 + 1$$

Adding up the left hand sides we get:

$$\sum_{i=1}^{n+1} i^2$$

Adding up the right hand sides we get:

$$\sum_{i=0}^{n} \left(i^2 + 2i + 1\right)$$

Equating

$$\sum_{i=1}^{n+1} i^2 = \sum_{i=0}^{n} \left(i^2 + 2i + 1\right)$$

or

$$\sum_{i=1}^{n+1} i^2 = \sum_{i=0}^{n} i^2 + 2\sum_{i=0}^{n} i + n + 1$$

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

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

The sum of the $i^2$ terms cancel out and we are left with

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

solving for $\sum_{i=0}^{n} i$ gives

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

as before. We have just given you a rather complicated derivation of a formula that Gauss could deduce instantly as an 8 year old. But now we have a clue how to proceed for $\sum_{i=0}^{n} i^2$.

3.4.4. *A derivation of* $\sum i^2$. We enjoyed some success with our second approach to deriving $\sum_{i=0}^{n} i$ Let us apply this same technique to $\sum_{i=0}^{n} i^2$. In the previous case we leveraged the formula:

$$(x+1)^2 = x^2 + (x+1)$$

Since we are now interested in the sum over $i^2$ we try raising the degree by one and see what happens when we start with the identity:

$$(x+1)^3 = x^3 + 3x^2 + 3x + 1$$

From this we proceed as before to generate our sequence of n+1 equations:

$$\begin{aligned}
(n+1)^3 &= n^3 + 3n^2 + 3n + 1\\
n^3 &= (n-1)^3 + 3(n-1)^2 + 3(n-1) + 1\\
(n-1)^3 &= (n-2)^3 + 3(n-2)^2 + 3(n-2) + 1\\
... &= ...\\
1^3 &= 0^3 + 3(0)^2 + 3(0) + 1
\end{aligned}$$

As before, adding up the left hand sides, adding up the right hand sides then equating gives:

$$\sum_{i=1}^{n+1} i^3 = \sum_{i=0}^{n} i^3 + 3\sum_{i=0}^{n} i^2 + 3\sum_{i=0}^{n} i + \sum_{i=0}^{n} 1$$

Solving for $\sum_{i=0}^{n} i^2$we get

$$\begin{aligned}
\sum_{i=0}^{n} i^2 &= \frac{\sum_{i=1}^{n+1} i^3 - \sum_{i=0}^{n} i^3 - 3\sum_{i=0}^{n} i - \sum_{i=0}^{n} 1}{3}\\
\sum_{i=0}^{n} i^2 &= \frac{1}{3}\left(\sum_{i=n+1}^{n+1} i^3 + \sum_{i=1}^{n} i^3 - \sum_{i=1}^{n} i^3 - \sum_{i=0}^{0} i^3 - 3n(n+1)/2 - (n+1)\right)
\end{aligned}$$

Here we cancel and also use our knowledge of $\sum_{i=0}^{n} i$ from the previous section:

$$\begin{aligned}
\sum_{i=0}^{n} i^2 &= \frac{(n+1)^3 - 3n(n+1)/2 - n - 1}{3}\\
\sum_{i=0}^{n} i^2 &= \frac{2(n+1)^3 - 3n(n+1) - 2n - 2}{6}\\
\sum_{i=0}^{n} i^2 &= \frac{1}{6}\left(2n^3 + 6n^2 + 6n + 2 - 3n^2 - 3n - 2n - 2\right)\\
\sum_{i=0}^{n} i^2 &= \frac{2n^3 + 3n^2 + n}{6} = \frac{n(n+1)(2n+1)}{6}
\end{aligned}$$

3.4.5. *A beautiful relation involving $\sum i^3$.* Now that we have established the basic technique we can quickly arrive at the formula for $\sum\limits_{i=0}^{n} i^3$. First we make use of the identity

$$(x+1)^4 = x^4 + 4x^3 + 6x^2 + 4x + 1$$

As usual, adding up the left hand sides, adding up the right hand sides then equating gives:

$$\sum_{i=1}^{n+1} i^4 = \sum_{i=0}^{n} i^4 + 4\sum_{i=0}^{n} i^3 + 6\sum_{i=0}^{n} i^2 + 4\sum_{i=0}^{n} i + \sum_{i=0}^{n} 1$$

Solving for $\sum\limits_{i=0}^{n} i^3$:

$$\sum_{i=0}^{n} i^3 = \frac{\sum\limits_{i=1}^{n+1} i^4 - \sum\limits_{i=0}^{n} i^4 - 6\sum\limits_{i=0}^{n} i^2 - 4\sum\limits_{i=0}^{n} i - \sum\limits_{i=0}^{n} 1}{4}$$

Going through the algebra a little more quickly this time:

$$\sum_{i=0}^{n} i^3 = \frac{1}{4}\left((n+1)^4 - n(n+1)(2n+1) - 2n(n+1) - n - 1\right)$$

$$\sum_{i=0}^{n} i^3 = \frac{1}{4}\left(n^4 + 4n^3 + 6n^2 + 4n + 1 - 2n^3 - 3n^2 - n - 2n^2 - 2n - n - 1\right)$$

$$\sum_{i=0}^{n} i^3 = \frac{n^4 + 2n^3 + n^2}{4} = \frac{n^2\left(n^2 + 2n + 1\right)}{4} = \frac{n^2(n+1)^2}{4} = \left(\frac{n(n+1)}{2}\right)^2$$

The fact that:

$$\sum_{i=0}^{n} i^3 = \left(\sum_{i=0}^{n} i\right)^2$$

is best appreciated if we write it out as:

$$1^3 + 2^3 + 3^3 + ... + n^3 = (1 + 2 + 3 + ... + n)(1 + 2 + 3 + ... + n)$$

That is, the sum of $n$ cubes is equal to the square of the sum of n numbers.What a remarkable and beautiful formula! Knuth also has a nice geometric proof of this relation, on page 19 of volume 1, attributed to R.W. Floyd.

**Exercise 3.3.** Using these same techniques, derive the formulas for $\sum\limits_{i=0}^{n} i^4$ and $\sum\limits_{i=0}^{n} i^5$

3.4.6. *The general case.* To obtain the general formula for $\sum\limits_{i=0}^{n} i^k$ using the above techniques we need to be able to determine the coefficients in of the terms in the expansion of $(n+1)^k$. We recall the binomial theorem from Equation 2.6:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

In particular,

$$(x+1)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$$

Following the strategy of the previous sections we can now give a recurrence for calculating $\sum\limits_{k=0}^{n} x^k$ (note, the variables named $n$ and $k$ below have no relation to those in the statement of the binomial theorem). Beginning as usual with our sequence of $n+1$ equations:

$$\begin{aligned}
(n+1)^{k+1} &= \sum_{i=0}^{k+1} \binom{k+1}{i} n^i \\
n^{k+1} &= \sum_{i=0}^{k+1} \binom{k+1}{i} (n-1)^i \\
(n-1)^{k+1} &= \sum_{i=0}^{k+1} \binom{k+1}{i} (n-2)^i \\
\ldots &= \ldots \\
1^{k+1} &= \sum_{i=0}^{k+1} \binom{k+1}{i} 1^i
\end{aligned}$$

Again adding up the left hand sides, adding up the right hand sides then equating gives:

$$\begin{aligned}
\sum_{i=1}^{n+1} i^{k+1} &= \sum_{j=0}^{n} \sum_{i=0}^{k+1} \binom{k+1}{i} j^i \\
&= \sum_{i=0}^{k+1} \sum_{j=0}^{n} \binom{k+1}{i} j^i \\
&= \sum_{i=0}^{k-1} \sum_{j=0}^{n} \binom{k+1}{i} j^i + \sum_{i=k}^{k} \sum_{j=0}^{n} \binom{k+1}{i} j^i + \sum_{i=k+1}^{k+1} \sum_{j=0}^{n} \binom{k+1}{i} j^i \\
&= \sum_{i=0}^{k-1} \sum_{j=0}^{n} \binom{k+1}{i} j^i + \binom{k+1}{k} \sum_{j=0}^{n} j^k + \binom{k+1}{k+1} \sum_{j=0}^{n} j^{k+1} \\
&= \sum_{i=0}^{k-1} \left( \binom{k+1}{i} \sum_{j=0}^{n} j^i \right) + \binom{k+1}{k} \sum_{j=0}^{n} j^k + \binom{k+1}{k+1} \sum_{j=0}^{n} j^{k+1} \\
&= \sum_{i=0}^{k-1} \left( \binom{k+1}{i} \sum_{j=0}^{n} j^i \right) + (k+1) \sum_{j=0}^{n} j^k + 1 \sum_{j=0}^{n} j^{k+1}
\end{aligned}$$

Solving for $\sum\limits_{j=0}^{n} j^k$:

$$\sum_{j=0}^{n} j^k = \frac{1}{k+1} \left( \sum_{i=1}^{n+1} i^{k+1} - \sum_{i=0}^{k-1} \left( \binom{k+1}{i} \sum_{j=0}^{n} j^i \right) - \sum_{j=0}^{n} j^{k+1} \right)$$

Cancelling the first and last terms on the right the recurrence, we have succeeded in demonstrating the following Theorem.

**Theorem 3.4.**

$$\sum_{j=0}^{n} j^k = \frac{(n+1)^{k+1} - \sum\limits_{i=0}^{k-1}\left(\binom{k+1}{i}\sum\limits_{j=0}^{n} j^i\right)}{k+1} \tag{3.21}$$

*Proof.* □

(Also see Chrystal [3] v.1, p.484-487, and v.2, p.232-233).

**Exercise 3.5.** Write a function `sum(n, k)` that will calculate $\sum\limits_{i=0}^{n} i^k$.

From the formulas for the various $\sum\limits_{j=0}^{n} j^k$ it is straightforward to move to the formula for the sum over n of an arbitrary polynomial in j.

**Exercise 3.6.** Express the solution to

$$\sum_{i=0}^{n} 3i^7 + i^5 + i^2 + 1$$

in terms of the appropriate sums. Then use the formulas for those sums from the sections above to come up with the general solution. Test your answer for small values of n.

## 4. Complexity

There are two things in the world of programming: problems and solutions. A problem could have multiple solutions, also known as algorithms. An example of a problem is sorting. Merge sort on the other hand is an algorithm. When we look at an algorithm we could measure it's performance. How do we measure performance? In the old days it was a macho thing to show how small a piece of code could be made, e.g. we would count instructions. Machines were very small in those days. Then people grew up and decided that it was a useless measure. But it's maybe not as useless as people thought. There is actually a profound theoretical reason for retaining this measure. Andrei Kolmogorov of the Moscow State University and Gregory Chaitin of IBM Research, tried to figure out an information theoretic definition of complexity. Kolmogorov published his results in 1965 and Chaitin, independently, in 1966. They created a new area of computer science known as Kolmogorov complexity. To determine how random a sequence is, its Kolmogorov complexity is defined to be the size of the smallest program generating it. The brilliant intuition here was to claim that a sequence is truly random if it could not be generated by a finite program. It is a wonderful measure, but unfortunately it is not computable. Of course, we can come up with upper bounds for the Kolmogorov complexity. But it is not computable, since if it were, we would be able to solve the equivalence problem.

Kolmogorov complexity, while not computable, are still useful to us. We get this idea that a program with a couple million lines of code must be doing something very useful. But this is not necessarily true. For example, white noise has very high Kolmogorov complexity. Well-engineered things tend to be made from small components with short structure. Beethoven's Ninth or Wagner's *Parsifal* can be encoded with much less information than 5 seconds of white noise precisely because of their order and structure. So code size can be a useful measure after all.

There are two other measures that we commonly use – time and space complexity. In this course we will not consider algorithms which call for more than linear extra space.

There are very few practical algorithms that require, say, a quadratic amount of storage. Two classes of space usage that we will consider: *in-place* algorithms, i.e. algorithms that take no extra space, for some definition of "no". Clearly one extra word doesn't matter. Knuth influenced the Computer Science community to allow in-place algorithms with $O(\lg n)$ extra space, to allow for algorithms that employ a recursive (divide and conquer) approach. Nowadays most people would consider an algorithm that required *polylog* (a polynomial of a logarithm) space as *in situ* (in-place). When Knuth wrote the first edition of volume 3, the hardest non-mathematical problem in the entire book was to design an in-place $n \lg n$ stable sort. Eventually one of his students (Pardo) solved it, and Knuth demoted it from 49 to 33. By the end of this course you will be able to do it yourself.

We are much more interested in time complexity than in space complexity. We can talk about the worst-case complexity or average complexity of an algorithm, or of a problem. The first number you want to know is the worst case measure. It is important to remember that for many practical algorithms worst case is irrelevant. The most common algorithm in use now is probably Dantzig simplex method for linear programming. It has exponential worse case performance, but it never occurs in practice. From time to time people come up with "improvements", but no one uses them. People still use the "slow" Dantzig simplex method algorithm. So we need to be careful – bad worst case does not necessarily mean bad algorithm.

We also are interested in average complexity. What is average complexity? You average over all the input data sets and see how long it takes. Bob Tarjan came up with a (sometimes) better measure called amortized complexity. For example, when you `push_back` a succession of elements to the back of a `std::vector` there may be some reallocation, but not after each call. The cost of reallocation is amortized across multiple calls and `push_back` is said to have constant amortized time complexity. The third measure of average time complexity is Monte Carlo or probabilistic averaging. Probabilistic algorithms give us a different measure of complexity which is the probability of the answer being correct after a given number of operations. Some times these probabilistic methods are very good. In fact we will later encounter a probabilistic algorithm for testing for whether a number is prime.

For any problem we also have a notion of an upper bound and a lower bound complexity. These are very difficult to come up with since we must consider the performance of all of the infinitely many possible algorithms for solving our problem. The lower bound of complexity for a problem is a bound that is lower than the time it takes to solve the problem using any particular algorithm. The upper bound means that we know that a problem can be solved in a certain number of steps. This is easier to come by, since any particular solution gives us an upper bound. Then there is the notion of a tight lower bound. If something is solvable in $O(n^2)$ time then it might also be solvable in $O(n)$ time. In such as case $O(n^2)$ would not be considered a tight lower bound.

For space complexity we looked only at in-place versus non-in-place. For time complexity we have a much finer scale. At one end of the scale we have the undecidable problems. These are not solvable at all. For example, the problem of whether Diophantine equations have integer solutions is known to be undecidable – there can be no algorithm for solving them. This was proved [8]in 1971 by Yuri Matijasevich (based on the work of Julia Robinson).

The next item on the scale is the line of tractability. Borel said there is little difference between infinity and very large finite numbers. That is, the difference between $10^{10^{10^{10^{10}}}}$ and $\infty$ is not practically distinguishable. So even if a problem is decidable, it still might be intractable, for example if it has exponential time complexity. On the other extreme of the scale we have problems that are solvable in sub-linear time, such as binary search (lg n). It is best not to become too preoccupied with achieving sub-linear time complexity since it must be tempered by the fact that it takes linear time to read in or assemble the data. We must also remember that even though an algorithm might be linear, it could still have complexity of, say, $1000n$ which might not be all that great.

Another important class of problems to know about are problems that can be solved in online linear time (those that consider the data one by one once and then are done), or to put it another way, those that depend upon input iterators. Moving up a notch in complexity, there is a very broad class of problems known as P: problems that are solvable in polynomial time. Practically, everything we can solve nowadays is in P.

Another interesting class of problems is known as NP. What does NP mean? NP, or Non-deterministic polynomial time, means that a solution could be checked/verified in polynomial time, even though we don't have a polynomial time algorithm for solving he problem. Unless we had a computer with unbounded parallelism. Then we could take all possible solutions, check them in polynomial time and then recover a polynomial time solution. (There is, therefore, an intriguing possibility that quantum computers will allow us to solve problems in NP in polynomial time.)

In 1972 Steven Cook (USA) and Leonid Levin (USSR) independently introduced the notion of NP completeness. They made an absolutely remarkable discovery. Both of them discovered that there are problems in the NP class that are complete. Complete means that if you could solve one of them in polynomial time then you could solve them all in polynomial time. Cook demonstrated that the 3-satisifiabilty problem is NP-complete. (A boolean expression is *satisfiable* if there is an assignment of boolean values to the variables which causes the expression to be evaluated as true. A boolean expression is said to be in *conjunctive normal form* if it is the conjunction of a number of clauses, where each clause is the disjunction of a number of terms, and each term is either a variable or the negation of a variable. An expression in conjunctive normal form with exactly three terms in each clause is said to be *3-CNF*. As an example of a 3-CNF expression consider:

$$(x_1 \wedge x_7 \wedge \neg x_3) \vee (\neg x_7 \wedge x_2 \wedge x_5)$$

Cook showed that the question of whether a given 3-CNF expression is satisfiable is NP complete.) Then Richard Karp came up with 30 more NP-complete problems. They both won Turing awards for their work. Then there is a book from the late 1970s/early 1980s by Garey and Johnson [9] containing hundreds of them. You might want to browse it so you know what not to try to solve. Or you might want to try to solve one. It could be your path to riches, since there is a $2,000,000 prize if you could solve one of them (in polynomial time). The most famous outstanding problem in computer science is whether P is equal to NP. Apparently, 91% of computer scientists believe that P $\neq$ NP. Our favorite NP-complete problem is the Knapsack problem. Somebody gives you a bunch of numbers. Is there a subset such that the sum of the numbers is equal to a given number $n$? So simple yet so hard. Another class of problems is the class of *NP-hard* problems. An NP-hard problem is one that if solved would imply a solution for NP problems, but the NP-hard problem is allowed to be more difficult than NP. Clearly any NP-complete is NP-hard. But there are other problems, like the computation of Grobner basis, or the travelling salesman, that are NP-hard, but not NP-complete. NP complete problems in practice tend to be yes/no problems. NP-hard problems are of the kind "give me" a solution. It may be helpful to remember that if you can "give me" then you can answer yes/no, so NP-complete $\supset$ NP-hard.

4.1. **Big O notation.** Given two functions $f$, and $g$ we say that $g = O(f)$ when:

$$\exists C, n_0 \text{ such that } \forall n : |g(n)| \leq C f(n), \text{whenever } n \geq n_0$$

**Exercise 4.1.** Show that if $g = O(n)$ then $g = O(n^2)$ also.

Given two functions f, and g we say that $g = \Omega(f)$ when:

$$\exists C, n_0 \text{ such that } \forall n : |g(n)| \geq C f(n), \text{whenever } n \geq n_0$$

An old but useful notation: we say that $f(x)$ $g(x)$, or that $f$ is equivalent to $g$ when

$$\lim_{x \to \infty} f(x)/g(x) = 1$$

For example, $n^2\sim \ n^2 + n$.

**Exercise 4.2.** Prove that $n^2\sim \ n^2 + n$.

We will not generally consider big O, preferring instead to count operations since the C could be anything. That is we will say $3n^2 - 7$.

**Exercise 4.3.** What is the largest number $n$ for which a given problem is solvable? Construct the following table (write a program to do so). Assume that computer X does 1 billion operations per second. Ignoring sub-linear (for reasons mentioned earlier) n, 10n, n lg n, n^2, n^3, 2^n, 3^n,10^n, n!. These are the rows of the table. Columns, 1 second, 1 minute 1 hour, 1 day 1 month 1 year. Values of table – what is the size of the maximum problem that you can solve for this row and column. Our intuition is screwed up in terms of big numbers. The importance to get a sense. Need to get a firm understanding. In other words, write a program to compute the values that will fill in the cells of this table with the largest possible $n$ in each case.

| | sec | min | hour | day | week | mth | year | century |
|---|---|---|---|---|---|---|---|---|
| $n$ | □ | □ | □ | □ | □ | □ | □ | □ |
| $10n$ | □ | □ | □ | □ | □ | □ | □ | □ |
| $n \lg n$ | □ | □ | □ | □ | □ | □ | □ | □ |
| $n^2$ | □ | □ | □ | □ | □ | □ | □ | □ |
| $n^3$ | □ | □ | □ | □ | □ | □ | □ | □ |
| $2^n$ | □ | □ | □ | □ | □ | □ | □ | □ |
| $3^n$ | □ | □ | □ | □ | □ | □ | □ | □ |
| $10^n$ | □ | □ | □ | □ | □ | □ | □ | □ |
| $n!$ | □ | □ | □ | □ | □ | □ | □ | □ |

## 5. Primitive recursion and iteration

5.1. **Recursive factorial.** We will attempt to write an implementation of `factorial()`. We begin with a C version. We will keep refining it and on the way we will learn a number of things. Recall that $n!$ is the product of the first $n$ positive integers. By convention we set $0! = 1$. We will only define factorial for natural numbers, avoiding the Gamma function in this class. We note in passing that the use of the symbol "!" to denote factorial was not introduced until the 19th century. It is the only postfix algebraic operators. But it is much easier to typeset than the previous notation. In any case, we begin with the following inductive definition of the factorial function

$$n! \equiv \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

We might write a C implementation as:

```
if(n==0) return 1;
return n * factorial(n-1);
```

Now that we've written the core of the routine it is appropriate to decide on a signature.

```
int factorial(int n){
  if(n==0) return 1;
  return n * factorial(n-1);
}
```

This may seem like an overly simple example, but it will actually prove to be quite instructive. For now, observe that we carry out the computation by starting at $n$ and going down towards 0. That is, we do the same thing computationally as induction does mathematically.

5.2. **Giuseppe Peano.** Giuseppe Peano was born in Italy in 1858 into a simple peasant family. With the help of his uncle he was able to get an education and eventually became a professor at the University of Turin. In 1886 Peano proved that if $f(x, y)$ is continuous then the first order differential equation dy / dx = $f(x, y)$ has a solution. In 1890 he invented 'space-filling' curves – continuous mappings from [0,1] onto the unit square. This was rather remarkable – it had been thought that such curves could not exist. (The curve is engraved on a monument in Cuneo – a little town in Piedmont where he was born.) In 1889 Peano published his famous axioms, which for the first time defined the natural numbers axiomatically. He decided that it was a travesty that there is no book that describes all of mathematics completely. From around 1892, he began a new project, the Formulario Mathematico:

*"...Of the greatest usefulness would be the publication of collections of all the theorems now known that refer to given branches of the mathematical sciences ... Such a collection, which would be long and difficult in ordinary language, is made noticeably easier by using the notation of mathematical logic ..."*

He began to writing it in French. French proved to be too ambiguous, so he came up with a grand plan: in order to write this book he needed to invent a new unambiguous language with a consistent grammar and an unambiguous vocabulary in order to clearly express mathematics. Others before him, including Leibniz and Pascal had been interested in such a language. Peano's language was called Latino Sine Flexione – Latin without inflections. He came up with beautiful rules for his grammar and by 1908-1909 was able to translate Formulario into his language. Of course this ended his career since almost no one wanted to learn his language. It is sad because his language was so self-evident. Quoting from his axioms:

(1) Zero es numero.
(2) Si a es numero, tunc suo successivo es numero.

Bertrand Russell was one of the people that saw the value of Peano's work. He even claimed Peano as his teacher. Russell popularized the axioms and this is why they carry Peano's name. As we shall see later, Peano's axioms will play a very important role for us.

Back to the factorial function. What we see here is an example of what is known as primitive recursion. A *recursive* function is one that is defined in terms of itself. While recursion is very powerful, as we shall see later, it is impossible to determine whether a general recursive function terminates. *Primitive recursion* is a restricted form of recursion that occurs when a function calculates the value at $n$ by calling itself using the predecessor of *n*. We know that such a function must terminates at 0. We can easily show that if such a function terminates for *n* then it terminates for $n + 1$, then by induction we know that it terminates for all *n*. Thus primitive recursion is intimately connected with induction. This is one reason why Peano axioms are important.

There are two problems here. Firstly, the algorithm is not in-place – it consumes stack space linear in *n*. This isn't so bad because computation of $n!$ for large *n* is essentially intractable anyway. The more significant problem is that this implementation requires a call to predecessor. Successor is the natural function, in fact predecessor might not even exist. Even if predecessor does exist it can be quite slow to calculate compared with successor. Consider singly linked lists for example. So naturally we ask whether we could do the same computation using only successor. It turns out that *any* function of this kind has an iterative representation which doesn't need to use predecessor and does not require extra stack space.

5.3. **Iterative factorial.** Here we start with the function signature since we already discovered it in the previous section.

```
int factorial(int n){
  int result = 1;
  int i = 0; // i is a good name for an inductive variable
  while(i != n) {
    i = i + 1;
    result = result * i;
  }
  return result;
}
```

In the recursive version the inductive variable is hidden in the stack frame, but in the iterative version we make *i* explicit. Now let us try to generalize it. What could be generalized? First of all *n* doesn't need to be an `int, factorial()` should work just as well for `shorts`. Also, the operation * comes to mind. We changed * to + and initialized `result` to 0 instead of 1 we would get another well known function.

**Exercise 5.1.** What function do we get in that case? Implement and test it.

We want to be as abstract as possible without losing efficiency. In order to do this we introduce a C++ idiom that we will use heavily in what follows.

5.4. **C++ function objects.** A *function object* is any object for which the application operator is defined. That is, if *a* is a function object then it is possible to write $a()$, or perhaps $a(\text{foo}, \text{bar})$. One important special case is when *a* happens to be a pointer to a function. But there are other kinds of function objects that are not function pointers, and we will generally use the term function object to refer to them. As an example of a function object that is *not* a function, we will work with a function object that adds 1 to its argument. We begin by defining a class like:

```
struct add1 {
  int operator()(int n) const{ //don't forget the const!
    return n+1;
    }
};
```

Observe that a `struct` is a perfectly good `class`. Recall that the *only* difference between a `class` and a `struct` is that a `struct` treats all members as `public` by default, while a class treats them as `private`. The strangely named member function is known as the application operator, (or “operator paren”, or the “function call operator"). Let’s go slowly here since it is important to really understand how these work. How might you use this class that we just created? There are two ways. First, we can name an object of class `add1` and then use the application operator:

```
add1 myAdd; // declare an object of add1 known as myAdd;
myAdd(3);   // call the application operator. returns 4.
```

Why do we even need the first line? That is why can’t we just call `add1(3)`? Of course, the reason is that we cannot apply/call a `class`. So we need to construct an object. We note that this class has no data members, and as a result all instances of this class are identical. `myAdd` is no different from `yourAdd`. Therefore we prefer to use `add1` in the following form:

```
add1()(3);
```

The thing to remember is that a class name followed by ( ) constructs a temporary, “anonymous” instance of the class. Programmers are often confused by this. This is made worse by the fact that it is not uncommon for programmers to abbreviate the phrase “class of the function object” to “function object”. It is important for you to thoroughly understand this idiom in what follows. We mention in passing that even though this class is stateless, it still uses some storage. C++ inherited the rule in C that no two objects can occupy exactly the same storage. So C++ must guarantee in, say, an array, that the first and second elements don’t have the same address. So even stateless function objects can incur a space overhead. If you start storing these in a class you will pay, especially given data alignment padding. One more thing: it is important to declare the application operator as a `const` member function. This indicates that calling this operator doesn’t change the state *of the function object.* It is easy to forget to do this and such an omission typically leads to unpleasant error messages.

Here is an example of how `add1` might be used. We use templates now, deferring further explanation of how they work until later.

```
//Concepts: A is an Adder
template <class A>
int increment(int n, A a){
  return a(n);
}
```

By convention we capitalize concept names. We defer an explanation of concepts until later. If we pass an `int` *n* and an Adder *a* to `increment()` it will increment *n* by applying *a*. So we can write, for example

```
increment(5, add1());
increment(5, add3()); // if we had already written add3.
```

We would rather not write a new adder each time we want to add a different constant. It is straightforward to automate the creation of a whole parametrized family of adders.

```
template <int k> //convention: we use k for constants
struct add {     //we drop the 1 suffix
  int operator() (int n) const {
    return n + k;
    }
  };
```

Now we can instantiate any kind of adder we like using named function objects:

```
    add<3> myAdd;
    add<5> yourAdd;
    myAdd(5);
    yourAdd(5);
```

Alternatively, we could use anonymously constructed function objects to obtain the same results

```
    increment(5, add<3>());
    increment(5, add<5>());
```

Let us recap some of the reasons that we use function objects. First, passing a function object to an inlined function can be much more efficient than passing a function pointer since if the application operator is inline there will be no overhead for the function call. Compare C's qsort to C++'s sort. Second, even the type of the function object carries some useful information. The argument and result types can be recovered if we follow the nested typedef convention explained in the next section. As another example, our add<k>'s above don't really carry any state, we could have passed them as template parameters instead of runtime parameters to increment. That is we could have rewritten increment so that it could be invoked as increment<add<3> >(5), though this change would come at the cost of a loss of some flexibility.

Alternatively, if we start with a binary function $f(x, y)$and a value $x$, we can construct a unary function object $g(x)$so that $g(x)(y) = f(x, y)$. That is we breakdown a function of multiple arguments into a sequence of functions of one argument. For example, given a binary function $f : X \times Y \longrightarrow Z$, for any x $\epsilon$ X we can construct the unary function $f_x : Y \longrightarrow Z$, by defining $f_x(y) = f(x, y)$.This trick of supplying multiple arguments to a function one by one is known as Currying after Haskell Curry, though it had been observed by Moses Schonfinkel and was implicit in Frege's work. Curry was a student of David Hilbert – we will be discussing Hilbert further in what follows. Let us see if we can come up with a general currying technique in C++. In order to do so we need to create a function with some state, so it is nearly impossible in C, but quite easy in C++ with function objects. We now construct a version of the add function object that corresponds to a curried version of the usual addition operator. If the we let the function $f$above correspond to the usual addition operator, then the curried form $g(x)$ above corresponds to the function object add(x) below::

```
  struct add {
    int k_;
  public:
    add(int k) : k_(k) {}
    int operator() (int n) const {
      return n + k_;
    }
  };
  //...
    increment(5, add<int>(3));
    increment(5, add<int>(5));
```

As a second example we start with the (binary) projection function $p : (x, y) \longrightarrow x$ and the value $x$ to construct a (unary) function object that ignores its argument always returning x:

```
//Concepts: T1 is a RegularType, T2 is a RegularType
#include <functional>
template <typename T1, typename T2 = T1>
class constant_function :
  public unary_function<T1, T2> {
private:
  T2 value;
public:
  constant_function(const T2& x)
    : value(x) {}
  T2 operator()(const T1&) const
  {
    return value;
  }
};
```

We will have more to say about regular types later on. Note that we pass by const& where possible to avoid extra copying since we don't know the expense of copying the parameters. Also we leave the unused parameter unnamed by way of documentation and to suppress compiler warnings. We can use `constant_function` to construct a unary function object `zero_fun(0.0)` that will return 0 when applied to any value:

```
constant_function<double> zero_fun(0.0);
assert(zero_fun(7.0) == 0.0);
```

We could also templatize the above variant of `add` replacing int with AbelianMonoid. Recall that a Monoid is an set together with an associative binary operation and an identity element. An abelian monoid is a monoid in which the binary operation is commutative (e.g. `op(a,b) == op(b,a)`).

```
//Concepts: M is an AbelianMonoid
template <typename M>
struct add : public unary_function<M, M> {
  M k_;
public:
  add(M k) : k_(k) {}
  M operator() (M n) const {
    return n + k_;
  }
};
```

Let's take a moment to discuss naming. It is important to spend some time asking, "What is the central abstraction?". It is often very hard to answer. Sometimes we might revise code for years because the abstractions are not right. In this case, we try to name the broadest collection of types for which the algorithm can work, e.g. abelian monoids. In this case we require the operation to be accessible as +, since this is the conventional mathematical symbol for commutative operations, while * is generally reserved for (possibly) non-commutative operations, such as string concatenation. It is important to make code as readable as possible by following these conventions. If we use + for some binary operation then it is quite possible that some programmer will come along and change an expression $a + b$ to $b + a$. If + was not commutative this would break some code. `std::string` should never have used `operator+()` for the non-commutative concatenation operation.

The C++ standard library provides us with a number of ready to use function objects such as `std::plus` and `std::multiplies`, and others. There are even a functions `bind1st` and `bind2nd` that aid us when we wish to perform basic currying. These can be accessed when we `#include <functional>`.

**Example 5.2.** We simplify the implementation of our most recent version of `add` using `std::plus` and `std::bind1st`.

```
//Concepts: M models AbelianMonoid
template <typename M>
struct add : public binder1st<plus<M> > {
  add(int k) :
    binder1st<plus<M> >(
      plus<M>(), k) {}
};
//...
  increment(5, add<int>(3)); //usage as before
```

Often we can avoid producing this class, instead employing the helper function `bind1st` which directly supports currying of function objects to create an anonymous function object that when applied adds 3 to its argument:

```
  increment(5, bind1st(plus<int>(), 3));
```

5.5. **Primitive recursion.** We're going to now implement a function that encapsulates primitive recursion. We considered how we might want to generalize the `factorial` function above, and we proceed to do so in stages. We start with our original code:

```
int factorial(int n){
  if(n==0) return 1;
  return n * factorial(n-1);
}
```

We begin by generalizing so that we can use initial values different from 1.

```
//Concepts: U models UnaryFunction
template <typename U>
int factorial(int n, U initial){
  if(n==0) return initial(n);
  return n * factorial(n-1, initial);
};
```

The careful reader might wonder why the `initial` is a unary function object instead of a nullary function object. Will we ever make use of the argument to `initial`? We will spend some time on this question in the next class. Unfortunately, we are now saddled with passing `initial` to each recursive call along with $n - 1$. So we take a moment to convert from a function template to a class template for a function object, so that we can change our interface so that we receive our expected `initial` function object earlier, at construction time.

```
template <typename U>
class factorial {
  U initial_;
public:
  factorial(U initial) : initial_(initial) {}
  int operator()(int n) const{
    if(n==0) return initial_(n);
    return n * operator()(n-1);
  }
};
```

It is conventional to supply `typedefs` with `UnaryFunction` and `BinaryFunction` objects that allow users to recover the types of the arguments and the type of the result. That is, inside of our `factorial` function we would like to be able to make use of the argument type expected by `UnaryFunction`. To arrange things so that the different types play nicely together we set the argument type and the result type of `operator()` to be the same as those of the of the `UnaryFunction`. We also expect the result of * to have the same type, as we shall see below.

```
template <typename U>
class factorial {
  U initial_;
public:
  typedef typename U::argument_type argument_type;
  typedef typename U::result_type result_type;

  factorial(U initial) : initial_(initial) {}
    result_type operator()(argument_type n) const{
    if(n==0) return initial_(n);
    return n * operator()(n-1);
  }
};
```

Next we will generalize how to go from an existing value to the next. That is, instead of hard-wiring in multiplication by returning $n * (n - 1)!$, we will modify `factorial` to accept an additional argument. The additional argument must be a binary function object, `composition`, that takes an `argument_type` and a `result_type` and combines them to return a `result_type`. At this point we are moving away from factorial towards an more general function that encapsulates primitive recursion so we change the name accordingly.

```
//Concepts: F is a UnaryFunction, B is a BinaryFunction
template <typename B, typename U>
class primitive_recursion {
    B composition_;
    U initial_;
public:
  typedef typename U::argument_type argument_type;
  typedef typename U::result_type result_type;

  primitive_recursion(U initial, B composition) :
```

```
              initial_(initial),
              composition_(composition) {}
    result_type operator()(argument_type n) const{
    if(n==0) return initial_(n);
    return composition_(n, operator()(n-1));
  }
};
```

Next we generalize the manner in which we detect that we have reached the induction base, instead of always requiring a test against 0. At the same time we remove the assumption that the predecessor of *n* is always found by subtracting 1.

```
template <typename B, typename U>
class primitive_recursion {
    B composition_;
    U initial_;
public:
  typedef typename U::argument_type argument_type;
  typedef typename U::result_type result_type;

  primitive_recursion(B composition, U initial):
              initial_(initial),
              composition_(composition) {}
    result_type operator()(argument_type n) const{
    if(n == induction_base(n)) return initial_(n);
    return composition_(n, operator()(predecessor(n)));
  }
};
```

Changing the test for *n* == 0 to a comparison against `induction_base(n)` is very powerful generalization. We could have stopped short by comparing to, say, something like `induction_base<argument_type>()`. But instead of a nullary version we supply a unary version. That is, instead of a single universal 0 for our type, we find 0 from our given *n*, e.g. zero becomes a function. Mathematically speaking we're doing a quantifier swap. We weaken the assumption of universal 0 from: there exists a zero such that for all *n*... we say: for all *n* there is a 0. This will allow us to apply induction to things very different from numbers, as we shall see when we study iterators.

We can also provide default versions of function templates for operations like `predecessor()` and `induction_base()` that work with types like `int` that already happen to provide `0`, ++, and `-`.

```
//Concepts: P models Peano
//Concepts: I models Incrementable
//Concepts: D models Decrementable

template <typename P>
inline P induction_base(const P&) {return 0;}

template <typename I>
inline I successor(I x) {return ++x;}
```

```
template <typename D>
inline D predecessor(D x) {return --x;}
```

**Example 5.3.** With the help of `primitive_recursion`, and no other loops, we implement factorial using function objects

```
  primitive_recursion<multiplies<int>,
              constant_function<int, int> >
    factorial(multiplies<int>(),
        constant_function<int, int>(1));
  int c = factorial(a); //example usage
```

**Example 5.4.** We implement a function object `modulo_multiplies` that carries out multiplication modulo *n* (see Section 7.3) . Then we use it together with `primitive_recursion` to construct a `modular_factorial` function object.

```
template <class Integer>
class modulo_multiplies :
  public binary_function<Integer, Integer, Integer> {
private:
  Integer n;
public:
    modulo_multiplies(Integer x) : n(x) {}
    Integer operator() (Integer x, Integer y) const {
    return (x * y) % n;
    }
};
//...
  primitive_recursion<modulo_multiplies<int>,
            constant_function<int, int> >
    modular_factorial(modulo_multiplies<int>(b),
              constant_function<int, int>(1));
  int c = modular_factorial(a); //example usage
```

**Example 5.5.** We implement `primitive_iteration`, and also construct `factorial` and the `modular_factorial` as we did for `primitive_recursion` above.

```
template <typename B, typename U>
class primitive_iteration
{
private:
  B composition;
  U initial;
public:
  typedef typename U::argument_type argument_type;
  typedef typename U::result_type result_type;
  primitive_iteration(const B& c, const U& i)
    : composition(c), initial(i) {}
  primitive_iteration() {}
  result_type operator()(const argument_type& n) const {
    argument_type i = induction_base(n);
    result_type result = initial(i);
    while (n != i) {
      i = successor(i);
      result = composition(i, result);
    }
    return result;
  }
};

//...
  primitive_iteration<multiplies<int>,
            constant_function<int, int> >
    factorial(multiplies<int>(),
          constant_function<int, int>(1));
  int c = factorial(a); //example usage

  primitive_iteration<modulo_multiplies<int>,
            constant_function<int, int> >
    modular_factorial((modulo_multiplies<int>(b)),
              (constant_function<int, int>(1)));
  int d = modular_factorial(a); //example usage
```

**Example 5.6.** We implement `predecessor` with the help of only `successor` to illustrate the point that iteration is more fundamental than recursion.

```
//Concepts: P1 and P2 model Peano
template <typename P1, typename P2 = P1>
struct predecessor_compose
  : binary_function<P1, P2, P2>
{
  P2 operator()(const P1& i, const P2& r) const {
    return (i == successor(induction_base(i))) ? r : successor(r);
  }
};

template <typename P>
struct base_of_induction
  : unary_function<P, P>
{
  P operator()(const P& i) const {
    return induction_base(i);
  }
};

template <typename P>
struct predecessor_iterative
  : primitive_iteration<predecessor_compose<P>,
                base_of_induction<P> >
```

```
{
};
```

5.6. **The axiomatic method.** Next we will introduce the Peano axioms and a generalization, but we pause briefly to recap the history of the axiomatic method. (The Greek *axioma* means "that which is considered self-evident". For example, it is self-evident – or, axiomatic– that the axiomatic method was invented by the Greeks; after all they invented everything: science, mathematics, philosophy, democracy, etc.) They figured out that one needs to start with some sort of first principles or else the chain of reasoning would go on forever. For thousands of years the only axiomatic system that was known was that of Euclidean geometry. A lot of what we know about axiomatic systems comes from struggling with the difficulties in the Euclidean axioms. The overall organization and content of Euclid's *Elements* is a great contribution to humanity. But his axioms were in fact one of the weaker parts.

The examination of the foundations of Euclid's elements started relatively late. Proclus, the neo-platonic Greek philosopher, tired to prove the fifth postulate in the fifth century A.D, as this postulate was not as self-evident as the others. By the early 18th century many mathematicians publicly started complaining about it. In 1733, Father Girolamo Saccheri, professor of mathematics at the University of Pavia, wrote *Euclides Vindicatus* (Euclid Vindicated) [10]. In it he attempted to prove the fifth postulate with a *reductio ad absurdum* argument. He went on for 100 pages deriving all sorts of bizarre things from the negation of the fifth postulate. Then he claimed that such a geometry would be complete madness. He did not, however arrive at any contradictions. By this he concluded Euclid was vindicated. Without full awareness of the fact, Saccheri was exploring non-Euclidean geometry .

There were many other attempts to prove the fifth postulate. For example, the postulate can be proved if you assume the "self-evident" fact that the sum of the interior angles of a triangle is 180 degrees. Then Lobachevsky, Bolyai, and Gauss caused a major revolution in mathematics and philosophy. It started with Lobachevsky, who tried assuming the negation of the fifth postulate and then completely developed the resulting geometry, now known as Lobachevsky space. Bolyai did the same thing several years later. He was so disappointed to find that Lobachevsky had done the same before him that he went mad. Gauss later claimed that he had already considered that you could choose different axioms. In so doing you would obtain a different geometry with different consequences. Gauss decided that it was so important to find which geometry applied to the real world that he wanted to measure the sum the interior angles of a huge triangle formed by three mountain peaks. If they totalled 180 then he would conclude that we lived in a Euclidean universe. If less than 180 our world would inhabit Lobachevsky space. He found that his measurements were not sufficiently accurate to demonstrate that we live in a non Euclidean universe.

At this point there was a crisis since the only axiomatic system known was crumbling. So in the 19th century foundations were reexamined. Towards the end of the 19th century the a rigorous axiomatization of euclidean geometry was finally completed. Note that after this point no further work really occurred in Euclidean geometry. This is an important point. Though it may come as a surprise given the way mathematics is currently taught, axioms come last, when a domain is complete. The final step in formalizing Euclidean geometry was done by David Hilbert. Hilbert spent ten years working on the foundations of geometry. He was born 1862 in Koenigsberg, where he late attended university eventually producing a dissertation on invariant theory. Then he did some remarkable work on algebraic number theory, producing the book *Zahlbericht*, building on the work of Kummer, Kronecker and Dedekind. Following the work of Moritz Pasch, Hilbert developed complete axiomatic system for Euclidean geometry. He came up with the first ideas on how we might deal with axioms. That is, he came up with a procedure that would allow a system of axioms to be analyzed. To determine whether a given axiom of a system was in fact necessary, remove the axiom. Then try to come up with a model which satisfies the reduced system of axioms. If the model turns out to be unacceptable, then the axiom was necessary. He labored at dropping various axioms from Euclid and discovering the consequences. The 21 axioms that he settled on were published in 1899 in his *Grundlagen der Geometrie* which finally put geometry on a formal foundation .

We can partition a mathematical concept into a theory and a model. A *theory*, consists of the body of axioms, proofs, inference rules, etc. A *model* contains objects (e.g. lines, points, planes). Then there is a mapping between theories and models. In computer science we view it as a mapping between specifications to implementations. The same notions exist in mathematics. Notice that a theory can have multiple models, and a model can have multiple theories. The important point is to remember that theories and models are distinct. A commonly forgotten truth, is that models comes before theories, or to paraphrase Kronecker, God created natural numbers, Peano created the axioms. (A much quoted sentence [11] from the mathematician Leopold Kronecker (1823-1891) goes as follows "Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk" – "God created the natural numbers, all else is the work of humankind"). Natural numbers are clearly something objective, they are not a figment of the human imagination. Numbers come first. Then we make axioms for them. The same thing is true in programming. The specification exists to serve the needs of the code. Of course there is an interplay between specification/interface and the algorithm.

For better of for worse, Hilbert had the most profound influence on 20th century mathematics. In 1900 during the Paris conference on world mathematics he stated 23 problems, the Hilbert problems, that defined the important things for a mathematician to be working on. Solving a single one of them could make a mathematician's career.

Prior to Hilbert, and perhaps during his time, there were many who did not believe in set theory, such as his main rival Henri Poincare. Hilbert legitimized set theory declaring that, "No one will drive us from the paradise that Cantor has created." By the turn of the century it became clear that, right or wrong, set theory was flawed with paradoxes. Part of the next major activity in Hilbert's life was to come up with a solid foundation of mathematics that would avoid these paradoxes. It was he who introduced the idea of primitive recursion when working on foundations.

5.7. **Peano types and a generalization.** Now we return to Peano's axioms. Peano claimed that there were 6 postulates for natural numbers (0-5). In the final version of his postulates he claimed that natural numbers started with 0. Nowadays most teachers will tell you that natural numbers start with 1. But we stick with the conventions of C, C++, and Peano's final version. First we look at the axioms in Peano's original language

*Ergo nos sume tres idea,* 0, *+ut idea primitivo, per que nos defini omni symbolo de Arithmetica. Nos determina valore de symbolo non definito* $N_0$, 0, *+per systema de propositio primitivo sequente.*

*.0* $N_0$*es classe, vel numero es nomen commune.*

*.1 Zero es numero.*

*.2 Si aes numero, tunc suo successivo es numero.*

*.3 $N_0$es classe minimo, que satisfac ad conditione* .0.1.2*id es, si ses classe, que contine* 0*et si a pertine ad classe s, seque pro omni valore che a, que et a+pertine ad s; tunc omni numero es s. Ce propositione es dicto principio de inductione, et nos indica illo per abbreviatione Induct.*

*.4 Duo numero, que habe successivo æquale, es æquale inter se.*

*.5* 0*non seque nullo numero.*

Here they are translated into English:

*.0 $N_0$is a class, which is commonly known as numbers.*

*.1 Zero is a number.*

*.2 If ais a number, then its successor is a number.*

*.3 $N_0$is the smallest class, that satisfies the conditions* 0*,* 1*and* 2*such that, if sis a class that contains* 0*and if for each value ain class s, a's successor a+is contained in s; then all numbers are s. This proposition is called the principle of induction.*

*.4 Two numbers, which have equal successors, are equal to each other.*

*.5 0 does not succeed any number.*

Finally here is a version in modern mathematical style:

*0. There is a set* $\mathbb{N}$*called the natural numbers*

*1.* $0 \in \mathbb{N}$*(*$\in$*means element of)*

*2. For each* $x \in \mathbb{N}$*, there exists a successor* $x' \epsilon \mathbb{N}$

*3. (Induction) Let* $\mathcal{S} \subset \mathbb{N}$*such that (*$\subset$*means contained in)*

*i)* $0 \in \mathcal{S}$

*ii)* $a \in \mathcal{S} \Rightarrow a' \in \mathcal{S}$*(*$\Longrightarrow$*means implies)*

*Then* $\mathcal{S} = \mathbb{N}$.

*4.* $x' = y' \Longrightarrow x = y$

*5.* $\forall x \in \mathbb{N} : \ x' \neq 0$*(*$\forall$*means for all)*

We apply Hilbert's methodology, removing the axioms one by one. In each case we offer an undesirable model of the reduced theory, thus demonstrating the necessity of all of the axioms.

1: {} - satisfies axioms 2 - 5

2: {0, 1, 2} - satisfies axioms 1 and 3-5

3: {0, 1, 2, 3 ... $\omega$} - satisfies axioms 1,2, 4, 5

4: {0, 1, 2, 3, 2} - satisfies axioms 1-3 and 5

5: {0, 1, 2, 3, 0} - satisfies axioms 1 through 4

Mathematics tends to push full specification, but we claim that the Peano axioms are in fact over-specified. Instead we should aim for the weakest possible specification for which the desired operations can be supported. In this way an algorithm can support the broadest possible range of types. This also makes algorithms much more robust, since the less we assume about the supplied types the more flexible we can be when types change during the life of the code. For example, if we write some code that works for *any* kind of number, then tomorrow if the we switch from `ints` to `floats` or event `complex` numbers code has a better chance of working. Nor, for example, should a piece of code depend on total ordering if not needed, so we prefer != to < for comparisons.

**Example 5.7.** We will now explore what a more loosely specified version of the Peano axioms might look like.

The reasons for this generalization will be made clear when we study `iterators` in a later chapter. We weakened the notion of zero in `primitive_iteration` from a simple constant to a function dependent on $n$. Instead of starting with a single 0 from which any object can be reached by repeatedly applying successors, we require that for every object there is some (not necessarily unique) base from which it can be reached. Our newly modified axioms now look like:

*0. A class* $\mathbb{T}$*is called an extended Peano Type (which we will refer to as simply Peano Type) if the following axioms hold*

*1. There is a function zero*: $\mathbb{T} \rightarrow \mathbb{T}$

*2. For each* $x \in \mathbb{T}$*, there exists a successor* $x' \epsilon \mathbb{T}$

*3. (Structural Induction) Let* $\mathcal{S} \subseteq \mathbb{T}$*such that*

*i)* $\forall x \in \mathbb{T} : \text{induction\_base}(x) \in \mathcal{S}$

*ii)* $a \in \mathcal{S} \Rightarrow a' \in \mathcal{S}$

*Then* $\mathcal{S} = \mathbb{T}$.

*4.* $x' = y' \Rightarrow x = y$

*5.* $\forall x \in \mathbb{T} : x' \neq \text{induction\_base}(x)$ Some of these ideas are addressed in Burstall's structural induction work [12] and also by Manna and Waldinger [13], but none of them appears to suggest anything like induction_base to weaken the notion of a universal zero as above.

One of the operations that must exist for the types supplied to our implementation of `primitive_iteration` is

```
template <typename P> //Concepts: P models Peano
PeanoType induction_base(const P&);
```

induction_base() corresponds to the zero function in the extended Peano Type axioms. Of course the `successor` operation is also required. For those use to the object oriented style of programming, it may be worth taking a moment to contrast the way that we use types in an interface. Notice that we do not require that clients inherit from any sort of PeanoType base class. We argue that such a requirement would be a needless overspecification. Nonetheless, we have been discussing some requirements on the actual types that will be supplied to our algorithms as PeanoTypes, e.g. the existence of an appropriate `induction_base` and `successor` operation. To indicate this we take care in naming are template type parameters, but unfortunately the compiler does not enforce these conventions in the current version of C++. Later on we will delve deeper into all of this, and the notion of Concepts.

Primitive recursion was formalized in the early 1910s by David Hilbert. Much earlier, the first person to successfully use primitive recursion was Hermann Grassmann. Grassmann was born in Prussia in 1809. His father was a high school math teacher. When he was about twelve the father decided that the boy had no talent for intellectual work and said he should become a gardener. During his last year in high school the boy became good at mathematics, finishing among the top students in his class and was able to attend university. After completing university, studying independently without any advisor, he produced a remarkable dissertation inventing linear algebra, tensor calculus, and Grassmann algebras, nowadays used to describe quantum mechanics. But though he sent his dissertation around to many mathematicians, it was not understood by anyone at the time, so he could not become a professor. Instead he became a high school teacher. In spite of all this, for twenty years he produced a remarkable stream of results in mathematics, mechanics and theoretical physics. None of his results were recognized. Now looking back we see that everything he tried was correct. In the early 1860s he wrote a high school text in which arithmetic was presented formally, where for the first time ever he came up with an inductive definition of addition and multiplication. As before his work was marginalized. He stopped doing math around 1865, though he continued teaching it, instead turning toward Sanskrit studies. He became so good at it that he was counted amongst the greatest Sanskrit scholars in the world in the 1870s, which was no mean feat. His Rig Veda dictionary is still widely used. In 1876 received a honorary doctorate from the University of Tübingen. Today his work is appreciated by the mathematical community.

**Example 5.8.** Grassmann's inductive definition of addition looked like this:

$$a + 0 = a$$

$$a + b' = (a + b)'$$

As an example tying together the notions of a Peano type, function objects, and inductive definition we will implement addition using only `successor` and `primitive_iteration`. But remember that we include this as a teaching example, not as an effective means of computation. This example is somewhat complicated but if you spend the time working through this and the next example, it will pull together your understanding of a number of the themes that we have been discussing.

```
//Concepts: P1 and P2 model Peano
//Concepts: I models Incrementable
template <typename Pair>
struct projection_first :
  public unary_function<Pair, typename Pair::first_type>
{
  typename Pair::first_type operator()(const Pair& x) const {
    return x.first;
  }
};

template <typename Pair>
struct projection_second :
  public unary_function<Pair, typename Pair::second_type>
{
  typename Pair::first_type operator()(const Pair& x) const {
    return x.second;
  }
};

template <typename P1, typename P2 = P1>
struct inductive_pair_first : public pair<P1, P2>
{
  inductive_pair_first(const P1& x, const P2& y) :
    pair<P1, P2>(x, y) {}
};

template <typename P1, typename P2>
inline inductive_pair_first<P1, P2>
induction_base(const inductive_pair_first<P1, P2>& x) {
  return inductive_pair_first<P1, P2>(
              induction_base(x.first), x.second);
}
```

```
template <typename P1, typename P2>
inline inductive_pair_first<P1, P2>
successor(const inductive_pair_first<P1, P2>& x) {
  return inductive_pair_first<P1, P2>(
                successor(x.first), x.second);
}

template <typename I>
struct range_pair : public pair<I, I>
{
public:
  range_pair(const I& x, const I& y) :
    pair<I, I>(x, y) {}
};

template <typename I>
inline range_pair<I>
induction_base(const range_pair<I>& x) {
  return range_pair<I>(x.first, x.first);
}

template <typename I>
inline range_pair<I>
successor(const range_pair<I>& x) {
  return range_pair<I>(x.first, successor(x.second));
}

template <typename T, typename I>
struct grassmann_counting : public binary_function<T,
                            I,
                            I>
{
  I operator()(const T&, const I& result) const
  {
    return successor(result);
  }
};

template <typename P1, typename P2 = P1>
struct grassmann_addition : public
  primitive_iteration<grassmann_counting<
              inductive_pair_first<P1, P2>, P2>,
              projection_second<
                inductive_pair_first<P1, P2> > >
{

};
```

**Example 5.9.** We continue with Grassmann's inductive definition of multiplication:

$$a \times 0 = 0$$

$$a \times b' = a \times b + a$$

Here, as we did for addition, we implement multiplication using only `grassmann_addition` from the last exercise with `successor` and `primitive_iteration`.

```
//Concepts: P1 and P2 model Peano
template <typename P1, typename P2 = P1>
struct grassmann_plus
{
  typedef inductive_pair_first<P1, P2> first_argument_type;
  typedef P2 second_argument_type;
  typedef P2 result_type;

  result_type operator()(const first_argument_type& x, const result_type&
y) const {
    return grassmann_addition<result_type>()(first_argument_type(x.second,
y));
  }
};

template <typename P1, typename P2 = P1>
struct grassmann_multiplication : public
  primitive_iteration<grassmann_plus<P1, P2>,
    projection_first<inductive_pair_first<P1, P2> > >
{
};

//... sample usage
int a=2, b=3, c=7, d=5;
grassmann_multiplication<range_pair<int>, range_pair<int> > times;
times(inductive_pair_first<range_pair<int>, range_pair<int> >(
  range_pair<int>(a, b), range_pair<int>(c,d)));
```

## 6. Exponentiation

6.1. **Introduction.** In this section we will try to communicate the various thought processes that are involved in successfully coding a generic version of the Russian peasant power algorithm. We will employ the method of successive refinement, starting with some slow incorrect code that contains the core ideas, and then gradually improving it until we are satisfied. The aim is to help drive a discussion of the various principles involved in designing high quality algorithms.

We will start with a non-generic recursive version, then move to the an iterative version, and finally to a generic version. We begin by remarking upon some general philosophical principles, and we will introduce more of these principles as we progress.

*Remark* 6.1. There are two kinds of programmers, those that write bad code and then keep refining it, and those that write bad code and leave it alone. Don't worry if you don't get it perfect initially. Be the first kind of programmer. Get as many people to criticize your code as possible.

*Remark* 6.2. Understanding only comes after you know something in 10 different ways.

6.2. **The Russian peasant algorithm.** We left off at the end of the last section with some examples regarding addition and multiplication, so it is only natural that we consider exponentiation next. We will produce an algorithm that can efficiently compute $a^n = a \cdot a \cdot a \cdot ... \cdot a$for as wide a range of different types of $a$ and $n$ as possible.

The notion of a general power function was a relatively late invention. The Greeks were interested in squares an cubes, but their interest was grounded in geometrical concerns and they did not dwell on four dimensional objects. It wasn't until the late 16th century that people began to consider the general power function. In the 17th century Descartes introduced the $a^n$ notation. Fermat was also quite fluent with the notion of powers, as evidenced by his notorious last theorem and his little theorem $a^{p-1} \equiv 1(\bmod p)$ which, incidentally, will be of great interest to us later. Much later George Boole suggested the notation a^n to simplify typesetting. Giuseppe Peano, who bequeathed to us many of the symbols that we use in mathematics today offered the notation $a \upharpoonleft b$. People began to consider powers of real numbers, powers with non-positive exponents, the binomial theorem, and so on. Consider for a moment what the definition of $a^0$. Why should $a^0 = 1$? So that the multiplication rule works out correctly. That is, for

$$a^i \cdot a^j = a^{i+j}$$

To hold in the case when $i = 0$ we must define $a^0 = 1$. We will not spend much time exploring non-negative exponents for the time being since they turn out not to be all that useful algorithmically. The notion of power developed in further as people began to move beyond powers of concrete numbers to consider powers of variables and polynomials. The advent of matrices created another domain in which powers were applied.

To get a sense of the nature of possible types for which such an operation might apply we start from first principles. We observe that we must have associativity, i.e.

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Actually, we wouldn't need associativity for all elements under $\cdot$ . It might suffice to require only that

$$(a \cdot a) \cdot a = a \cdot (a \cdot a)$$

but there are not many known useful examples of such a partially associative mathematical structure, so we exercise some judgement and require that the binary operation be associative.

**Definition 6.3.** Semigroup

A set with an associative binary operation is known as a *semigroup*.

We also would like an identity element (e.g., 1) so that we may define $a^0$.

**Definition 6.4.** Monoid

A semigroup equipped with an identity element is known as a *monoid*.

We don't require commutativity, $a \cdot b = b \cdot a$ but it is interesting to note that when both operands are powers of $a$ commutativity follows directly from associativity.

$$a^3 \cdot a^4 = (a \cdot a \cdot a) \cdot (a \cdot a \cdot a \cdot a) = a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a = (a \cdot a \cdot a \cdot a) \cdot (a \cdot a \cdot a) = a^4 \cdot a^3$$

We can now begin to write our first generally useful function. Here is an obvious implementation.

```
  while(n!=0) {
    if(!is_even(n)) result *=a;
    a *= a;
    n /=2;
  }
```

This implementation is wrong. While it will generate the correct answer, it carries out too many multiplication operations. To improve things we will begin by observing that, for exponents that are powers of 2 we have

$$a^{2^n} = \left(a^{2^{n-1}}\right)^2 \tag{6.1}$$

For example we calculate $a^8$by as $((a^2)^2)^2$, requiring only 3 multiplications. This doesn't quite work for odd numbers – they will cost us extra multiplications. The algorithm that we will use is known as the Russian peasant algorithm. In the 19th century western travellers noticed that peasants in Russia used such an algorithm as a way of carrying out multiplication, thus the name, although examples of its use can be found as far back as the Egyptian Rhind Papyrus. The algorithm, from the perspective of exponentiation, leverages Equation 6 to drastically reduce the number of multiplications. That is, we will base our function on the fact that

$$a^n = \begin{cases} \left(a^{n/2}\right)^2 & \text{if } n \text{ is even} \\ a(a^{n/2})^2 & \text{if } n \text{ is odd} \\ 1 & \text{if } n \text{ is zero} \end{cases} \tag{6.2}$$

6.3. **Recursive power().** We start with a recursive implementation.

*Remark* 6.5. Start with the core of the algorithm. Before you discover the signature or the end cases you need to look at the core – what does it do?

In this case the core is to divide an conquer to actually compute the power $a^n$ in terms of smaller powers:

```
  return power(a, n/2) * power(a, n/2);
```

Of course, this line of code is not correct. Well, it almost would work if $n$happened to be a power of 2. But even then it would be way too slow – why are we calling power(a, n/2) twice when a single call would do? So our first refinement is to make it more efficient:

```
  SomeType tmp = power(a, n/2);
  return tmp * tmp;
```

It would be handy at this point to have a name for the type of tmp, so we spend a moment on the signature of power(). At first glance it seems reasonable to write:

```
  int power(int a, int n);
```

However it is probably better to change it. The signature above gives a bit of a misleading impression that the variable $a$ and $n$ are required to be of the same type. This is not the case. One common idiom to avoid this impression is to choose another type for $a$ in this case let's make it a double. That is, we prefer the following signature:

```
  int power(double a, int n);
```

*Remark* 6.6. Always ask yourself: "Do a given set of types really need to be the same or could the algorithm work just as well if they were different?"

This will also be of help when we genericize the things. In any case we can now write:

```
double tmp = power(a, n/2);
return tmp * tmp;
```

What if $n$ does not happen to be a power of 2? To accommodate different values of $n$ we employ the identity from Equation 6:

$$a^n = \left(a^{n/2}\right)^2 a^{n\%2} \tag{6.3}$$

in our return statement:

```
return n%2 == 0 ? tmp * tmp : a * tmp * tmp;
```

or simplifying slightly:

```
double tmp = power(a, n/2);
double tmp *= tmp;
return n % 2 == 0 ? tmp : a * tmp;
```

Note that Equation 6 does not help to reduce $n$ in the case that $n$ is already 1. So we must add a test and terminate condition:

```
if(n==1) return a;
double tmp = power(a, n/2);
double tmp *= tmp;
return n % 2 == 0 ? tmp : a * tmp;ó‘
```

One last refinement can be made to this recursive version in the case when the type of a is in fact a MonoidElement. In such cases we can relax `power`'s precondition to allow $n$ to be 0. Our code then becomes:

```
double power(double a, int n)
{
  if(n==0) return 1.0;
  if(n==1) return a;
  double tmp = power(a, n/2);
  double tmp *= tmp;
  return n % 2 == 0 ? tmp : a * tmp;
}ó‘
```

It will be useful to make note here that the expression `return 1;` above might better be written as `return 1.0;` This will come up again when we try to make the algorithm generic. Another issue that will arise in the generic case is the tension between our desire to support the case where $n$ is zero on the one hand, and the fact that such support would exclude variables $n$ of a type that does not provide a 0 element.

Let's consider the cost in terms of the number of multiplications performed by this algorithm. We see that we need $\lg n$ multiplications + 1 additional multiply for each a 1 bit in the binary representation of $a$, not counting the last (leftmost) 1 bit. And this does turn out to be a win in the case of number types such as `double` that have fixed storage size. But what if we were using some sort of arbitrary precision `BigNum` type or even polynomials? The cost of multiplying, say, two polynomials will be linear in the product of their lengths (e.g. bilinear, or quadratic). But the length of the result of multiplying two polynomials will be linear in the sum of their lengths (e.g. additive). So for example, let's consider our initial naive version of `power` applied to the simple polynomial $x$. The cumulative cost of the multiplication operation (repeatedly multiplying by $x$) to calculate

$x^n$ will be on the order of $1 + 2 + 3 + ... + n - 1 \approx n^2/2$. In the case of the Russian peasant algorithm, we consider only the cost of the last step, that is, the cost of calculating

$$x^{n/2} \cdot x^{n/2}$$

Even by itself this last operation will cost on the order of approximately $n^2/4$. So there may be a slight win over the naive algorithm. If the size of the result is smaller than linear in the size of the arguments then the scales tip in favor of the Russian peasant algorithm, for example when computing `power` modulo $p$.

6.4. **Iterative power().** Now we implement `power()` iteratively, since even though the recursion would not cost us much space, the cost of the functional call is high relative to the expense of invoking *. We will once again start from incorrect code and gradually make it correct.

*Remark* 6.7. It is better to spend more time on your code up front constantly refining it. Programmers tend to stop designing code to early, claiming that they can fix things up later in the debugger. We claim that it is much more efficient to do the work up front.

It is tempting to write the iterative `power` as follows (Thanks to Mike Schuster for improving the code by suggesting that we make explicit the loop invariants).

```
double result = 1.0;
while (n != 0){
  //loop invariant: a^n * result
  //remains constant as the correct answer
    if(n%2 == 1) result *=a; // Note 1
    a *= a;          // Note 2
    n /= 2;
}
return result;
```

There are some problems here. At *Note 1* we are performing an unnecessary multiplication the first time through the loop when we already know that `result` is equal to 1.0. There is also one extra operation during the last iteration at *Note 2*: we square $a$, but then we don't use it. We would like to eliminate both of these.

*Remark* 6.8. Pay attention to the code and ask if there are extra unnecessary operations.

How do we get rid of these extra operations? In some sense the two operations are connected. We would like to place the code at Note 2 *before* the code at Note 1. Then we would never waste the squaring operation. Let us do so even though our code will become temporarily incorrect:

```
double result = 1.0;
while (n != 0){
    a *= a;
    if(n%2 == 1) result *=a;
    n /= 2;
}
return result;
```

Now we don't have an extra square at the end. But the result is wrong. Actually, it is the right result for the wrong exponent – this calculates $a^{2n}$. We could try to patch the exponent by inserting $n/ = 2$ before the loop.

```
double result = a;
n /= 2;
while (n != 0){
    a *= a;
    if(n%2 == 1) result *=a;
    n /= 2;
}
return result;
```

This works for odd $n$, but not when $n$ is even. Please verify this for $n = 3, 5$ and $n = 2, 4$. Observe the following remarkable fact: if you take a positive even number and keep dividing it by 2, it will eventually become an odd number. That is, is you keep shifting a positive number to the right, eventually you will be left will a single 1 bit. This observation suggests the solution. We avoid computing result until we obtain an odd number, then we use the technique above.

```
while (n%2 == 0){
    n /= 2;
    a *=a;
};
assert(n%2 != 0);

double result = a;
n /=2;
while (n != 0){
//loop invariant: a^(2n) * result remains constant = the correct
answer
    a *= a;
    if(n%2 == 1) result *=a;
    n /= 2;
}
return result;
```

We are close to a solution that satisfies us. Let us now take a moment to make our code a little bit more generic. Conceptually, we have been thinking in terms of odd and even numbers, and of halving a number. For `ints` it is fine to use % and /= for these purposes. But other types that we wish to accommodate may not overload these operators. Perhaps the type that we use for the exponent doesn't offer division by arbitrary number, offering instead only some sort of halving operation. Or perhaps we might be using some BigNum package that offers a function for testing whether a BigNum is odd in a much more efficient manner than examining its remainder mod 2. Not to mention that readability can also be improved. So we raise the level of abstraction a little bit by introducing names and corresponding hook points for these ideas.

*Remark* 6.9. Strive to make your code accommodate as many types as possible by making it generic. That is, raise the level of abstraction and consider the minimum requirements that must be imposed on the client types for the algorithm to perform successfully and efficiently.

**Example 6.10.** We transform the above version of the algorithm into a generic algorithm.

That is, we will write a templated version of `power` that depends on three template type concepts: `Monoid, BinaryInteger, BinaryOperation.` We take care in selecting the names for the concepts (collections of operations on types that must be available) to which the template parameters must adhere. This stems in part from the lamentable state of existing programming languages. Sadly, there is no direct enforcement of the constraints on the supplied types offered in current C++ (or in other useful programming languages). But the algorithm exists nonetheless, and we do our best to express the platonically ideal algorithm in the shadows of C++. In any case, the aforementioned type parameters must conform to the following type requirements

1) *n* must be a type that models a `BinaryInteger` concept. That is, it must be possible to use the following operations on *n*: `is_even(n), bool is_odd(n), is_zero(n)` must be usable where a bool is need. `halve_non_negative(n)` must be usable wherever *n* can be used.

2) *a* must model a `MonoidElement`, together using a `BinaryOperation` called, say, `bop`. So $a* = a$ above could become `a = bop(a,a);` Why `halve_non_negative()`? This allows for types that only supply right-shifting. The same considerations apply to the *= that we use for multiplication – you could have multiple interesting operations on the same type. For example we could compute the power of one integer to another in the usual way or instead we might wish to calculate power mod 7, as we will do later. Or we might want to calculate the some power of a matrix in the usual we, or in a very different way in the case that we are working with adjacency matrices to compute shortest path. In such a case instead of using * and + we will use + and `min()`.

We us the name `BinaryInteger` to signify that the type is allowed to be weaker than an `Integer`. That is, `BinaryInteger` need not provide full Integer division – only a `halve_non_negative()` operation is required. Finally, we add code to handle the case where *n* is zero.

```
//Concepts: M models Monoid under B which models BinaryOperation
//Concepts: I models BinaryInteger
template <class M>
inline M identity_element(const multiplies<M>&)
{ return M(1); }

template <class M>
inline M identity_element(const plus<M>&)
{ return M(0); }

template <class M, class I, class B>
M power(M x, I n, B operation) {

  if (is_zero(n)) return identity_element(operation);

  assert(is_positive(n));

  while (is_even(n)) {
    halve_non_negative(n);
    x = operation(x, x);
  }

  M result = x;
  halve_non_negative(n);
```

```
    while (is_positive(n)) {
      x = operation(x, x);
      if (is_odd(n)) result = operation(result , x);
      halve_non_negative(n);
    }
    return result;
  }
```

**Example 6.11.** Implement versions of the operations necessary to allow plain old `ints` to model the `BinaryInteger` concept.

Actually, we implement general versions that will by default work with types that supply the operations using the same names for the operators as do the integral types.

```
//Concepts: I models BinaryInteger
template <class I>
inline bool is_zero(const I& x) {
  return x == I(0);
}

template <class I>
inline bool is_positive(const I& x) {
  return x > I(0);
}


template <class I>
inline void halve_non_negative(I& x) {
  assert(is_positive(x));
  x >>= 1;
}

template <class I>
inline void double_non_negative(I& x) {
  assert(is_positive(x));
  x <<= 1;
}

template <class I>
  inline bool is_odd(const I& x) {
  return x & I(1);
}


template <class I>
  inline bool is_even(const I& x) {
  return !(x & I(1));
}
```

```
template <class I>
inline void normalize(I& x) {
  if (x < I(0))
  x = -x;
}
```

**Example 6.12.** With the help of the implementations from the previous two examples, write a test program that calculates powers of 2x2 matrices.

```
//Concepts: S models SemiRing
template <typename S>
struct matrix_2
{
  typedef S value_type;

  S a00;
  S a01;
  S a10;
  S a11;

  matrix_2(S x00, S x01, S x10, S x11)
    : a00(x00), a01(x01), a10(x10), a11(x11) {}
  matrix_2(S x)
    : a00(x), a01(S(0)), a10(S(0)), a11(x) {}
};

template <typename S>
S matrix_element00(matrix_2<S>& m)
{
  return m.a00;
}

template <typename S>
S matrix_element01(matrix_2<S>& m)
{
  return m.a01;
}

template <typename S>
S matrix_element10(matrix_2<S>& m)
{
  return m.a10;
}

template <typename S>
S matrix_element11(matrix_2<S>& m)
{
  return m.a11;
}

template <typename S>
bool operator==(const matrix_2<S>& x,
         const matrix_2<S>& y)
{
  return  x.a00 == y.a00 &&
      x.a01 == y.a01 &&
      x.a10 == y.a10 &&
      x.a11 == y.a11;
}

template <typename S>
matrix_2<S> operator+(const matrix_2<S>& x,
        const matrix_2<S>& y)
{
  return  matrix_2<S>(x.a00 + y.a00,
            x.a01 + y.a01,
            x.a10 + y.a10,
            x.a11 + y.a11);
}

template <typename S>
matrix_2<S> operator*(const matrix_2<S>& x,
            const matrix_2<S>& y)
{
  return  matrix_2<S>(x.a00*y.a00 + x.a01*y.a10,
            x.a00*y.a10 + x.a01*y.a11,
            x.a10*y.a00 + x.a11*y.a10,
            x.a10*y.a01 + x.a11*y.a11);
}
//...
matrix_2<int> m(1, 1, 1, 0);
matrix_2<int> result = power(m, 7, multiplies<matrix_2<int> >());
//...
```

6.5. **Addition chains.** We have come a long way from the initial naive implementation to the Russian peasant power implementation. However there are still times when we can do better. The first such case is when n = 15. According to our analysis of the Russian peasant algorithm, the number of multiplications needed when $n = 15$ is

$$\lg n + \#\,\text{of 1 bits in}\, n - 1 = \lg 15 + 4 - 1 = 6$$

But $a^{15}$can also be calculated as $(a^5)^3$. Now calculating a fifth power requires $\lg 5 + (\#\text{of 1 bits in 5}) - 1 = 3$ multiplications. Calculating a third power requires $\lg 3 + (\#\text{of 1 bits in 3}) - 1 = 2$ more multiplications. So the alternate method only requires $3 + 2 = 5$ multiplications, which beats the Russian peasant algorithm by 1. The problem of figuring out the most efficient way to calculate $a^n$ by multiplication relates to an interesting domain of mathematics known as the theory of addition chains. It boils down to finding the shortest *addition chain* for $n$, i.e. the shortest sequence of integers

$$1 = a_0, a_{1,}a_2, ..., a_r = n$$

for a given $n$ with the property that

$$a_i = a_j + a_k, \text{for some} k \leq j < i$$

for all $i = i, 2, ..., r$. For example, $1, 2, 4, 5$and $1, 2, 3, 5$are addition chains for 5 but $1, 2, 5$ is not. Restating our example in the case of $n = 15$ in this language, the shortest addition chain is not $1, 2, 3, 4, 7, 8, 15$as suggested by the Russian peasant algorithm, since we can do better with $1, 2, 3, 6, 12, 15$. There is an interesting discussion of addition chains in Knuth, v2. p465. The problem of finding the shortest addition chain as a function of $n$ remains unsolved, though some very great mathematicians have tried. We believe the bound is asymptotically $\lg n + \lg\lg n$but we have no proof of this. And as Knuth says, every self-evident fact about addition chains is false.

We finish up this section with a pretty implementation of our algorithm for use in the (rare) case that the exponent is known at *compile time* and an inline efficient `power` computation is needed. That is, we might know that we will need to calculate $a^{15}$efficiently in our program in such a way that the power function expands inline into something like:

```
x = a * a * a;  //1,2,3
y = x * x;      //6
y = y * y;      //12
y = y * x;      //15
```

We offer the code to calculate $a^k$for a constant $k$ that is known at compile time below. We even use explicit template specialization to give the optimal addition chains for the case when $k = 15$.

**Exercise 6.13.** Although the code below will work, it will not use the shortest addition chains in all cases after $k = 15$. Figure out the shortest addition chains up to $k = 40$and provide the additional explicit specializations below.

**Exercise 6.14.** (Extra credit) Do the same up to $k = 100$.

```
#include <functional>

template <class T>
inline T identity_element(const multiplies<T>&)
{ return T(1); }

template <class T>
inline T identity_element(const plus<T>&)
{ return T(0); }


template <int k>
struct conditional_operation;

//Concepts: M models monoid under B which models BinaryOperation
template <>
struct conditional_operation<0>
{
    template <typename M, typename B>
    M operator()(const M& a, const M& b, B)
    {
    return a;
    }
};

template <>
struct conditional_operation<1>
{
    template <typename M, typename B>
    M operator()(const M& a, const M& b, B operation)
    {
    return operation(a, b);
    }
};

template <int k>
struct power_k;

template <>
struct power_k<0>
{
    template <typename M, typename B>
    M operator()(const M& a, B operation)
    {
    return identity_element(operation);
    }
};

template <>
struct power_k<1>
{
    template <typename M, typename B>
    M operator()(const M& a, B operation)
    {
    return a;
    }
};

template <>
struct power_k<2>
{
    template <typename M, typename B>
    M operator()(const M& a, B operation)
    {
    return operation(a, a);
    }
```

```
};

template <int k>
struct power_k
{
    template <typename M, typename B>
   M operator()(const M& a, B operation)
    {
    return conditional_operation<k%2>()(
            power_k<2>()(
                 power_k<k/2>()(a, operation),
                 operation),
            a,
            operation);
    }
};

template <>
struct power_k<15>
{
    template <typename M, typename B>
   M operator()(const M& a, B operation)
    {
    return power_k<3>()(power_k<5>()(a, operation), operation);
    }
};
//A simple test driver
#include <iostream>

int main()
{
    int a=2;
    cout << a << "^8␣␣=␣"
          << power_k<8>() (2, multiplies<int>())
          << endl;
    cout << a << "^15␣=␣"
          << power_k<15>()(2, multiplies<int>())
          << endl;
    cout << a << "^30␣=␣"
          << power_k<30>()(2, multiplies<int>())
          << endl;

    cout << a << "*8␣␣=␣"
          << power_k<8>() (2, plus<int>())
          << endl;
    cout << a << "*15␣=␣"
          << power_k<15>()(2, plus<int>())
          << endl;
    cout << a << "*30␣=␣"
         << power_k<30>()(2, plus<int>())
         << endl;

    return 0;
}
```

6.6. **Fibonacci sequences.** We will now consider the problem of “Quot paria coniculorum in uno anno ex uno pario germinentur”or “How Many Pairs of Rabbits Are Created by One Pair in One Year.”We will discover in this course one of the most important and useful sequences – nowadays such sequences are known as Fibonacci sequences. It was introduced as a minor problem by a great Italian mathematician Leonardo Pisano, known now as Fibonacci. Near the end of the 12th century he was one of the first who went to study mathematics at Arab universities, as that civilization was the most advanced intellectually at the time . When he came back he wrote the first “computer science” book, *Liber Abaci*, the book of the calculation, where he introduced many important algorithms such as long subtraction, long multiplication, long division, etc. He wrote several other important books, and made great contributions to number theory. He was probably the greatest number theorists between Diophantus and Fermat. Sadly enough, he is mainly known today only for the sequence from the rabbit problem for which he was barely concerned.

The problem of the rabbits was stated as follows [14] .

" A certain man had one pair of rabbits together in a certain enclosed place, and one wishes to know how many are created from the pair in one year when it is the nature of them in a single month to bear another pair, and in the second month those born to bear also. Because the abovewritten pair in the first month bore, you will double it; there will be two pairs in one month. One of these, namely the first, bears in the second month, and thus there are in the second month 3 pairs; of these in one month two are pregnant, and in the third month 2 pairs of rabbits are born, and thus there are 5 pairs in the month; ... and thus one after another until we added the tenth to the eleventh, namely the 144 to the 233, and we had the abovewritten sum of rabbits, namely 377, and thus you can in order find it for an unending number of months."

He continues to solve the problem in the manner above, but here we depict the solution more compactly with a table of the number of rabbit of various kinds after a given number of months.

| Month : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Adults : | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 |
| Babies : | 1 | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 |
| Total : | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 |

The sequence that appears in the “Adults” row and also in shifted forms in other rows is known as the Fibonacci sequence. This name might be a bit unfair since Fibonacci did not seem to really recognize the importance of this sequence.

The person who recognized the importance, over 400 years later, was one of the greatest figures in the history of science, Johannes Kepler (1572-1630). He was born to a poor family in Germany, and he did quite well as a student. When he was about 25 he decided to develop a new model of the universe. One of the central ideas of his life was that the world was governed by good mathematics. His first try was to relate the motion of the planets to spheres inscribed in the regular polyhedra, a rather beautiful but totally bizarre idea. Despite the beauty of the mathematics, the theory did not fit the numbers. But it attracted the attention of the greatest astronomer of the time, the imperial mathematician (of the Holy Roman Empire) the Tycho Brahe, Danish by birth.

Brahe gathered a great deal of data concerning the movement of the planets. Tycho Brahe rejected both the Ptolomeian model (earth at the center of the moving planets) and the heliocentric Copernican model in favor of his own model where the planets were supposed to orbit around the sun while the sun was supposed to orbit around the earth. Surprisingly, this mathematical model conformed to his observations better than the previous models. Brahe asked why a parallax effect wasn't observed if the earth was actually orbiting around the sun. He didn't observe any effect and so he reasoned that either the universe would have to be unbelievably large or that Copernicus was wrong. He chose the latter conclusion. He hired Kepler as his assistant. By that time the old numbers were not particularly good, and Kepler was not satisfied with Brahe's model, so he kept on working on the problem. Around 1609 he came up with a draft of the first two of his laws, while his third law was published in 1619:

(1) The orbits of the planets are ellipses, with the Sun at one focus of the ellipse
(2) The line joining the planet to the Sun sweeps out equal areas in equal times as the planet travels around the ellipse
(3) The squares of the periods of the planets are proportional to the cubes of their semimajor axes

The first law was a fundamental breakthrough. Kepler was the first person to conclude that since all attempts to prove that the planets traveled in circles didn't work that he needed to try something else. He had read a book by Apollonius of Perga on conic sections and remarkably he decided to apply it to the problem, trying to describe the motion in terms of ellipses. As long as people believed in circular motion they assumed that the speed of the planets was uniform, but elliptical motion wouldn't allow this. So he came up with his second law. But this wasn't all that he invented. He also founded optics, figuring out the modern model of how the eye works. He also developed the first theory of telescopes. He observed that wine merchants would measure a volume of a wine by poking a stick diagonally in the barrel, and developed the mathematical theory of volumes of solids of revolution. This was perhaps the firs great example of integration after Archimedes.

Incidentally, the first decade of the 1600 was a wonderful time. Hamlet was written 1601, Don Quixote in 1608. At the court of Mantua, Claudio Monteverdi composed and staged the first opera, Orpheo, while thinking that he is reinventing Greek drama. the Authorised Version of the Bible (known to Americans as the King James Bible) appeared in 1611. Galileo was building his telescopes, Simon Stevin invented hydrodynamics.

In 1611 Kepler decided to give a present to his friend, the chancellor of the Holy Roman empire. While walking to a New Year's reception he was wondering, "What do you give somebody very important as a present?" He wanted to give nothingness (*nihil*) as a present. What could be more ephemeral than a snowflake. So he wrote his treatise on the Six Cornered Snowflake [15]. We quote one passage from it here.

*Of the two regular solids, the dodecahedron and the icosahedron, the former is made up precisely of pentagons, the latter of triangles but triangles that meet five at a point. Both of these solids, and indeed the structure of the pentagon itself, cannot be formed without the divine proportion [golden ratio] as modern geometers call it. It is so arranged that the two lesser terms of a progressive series together constitute the third, and the two last, when added, make the immediately subsequent term and so on to infinity, as the same proportion continues unbroken.*

E.g. $\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \tau$ To understand Kepler's references to pentagons we need to understand that the regular pentagon is built around the golden ratio. In a pentagon with sides of length 1 unit, all of the diagonals are the golden ratio. The dodecahedron and icosahedron are also related to the golden ratio since they are both are based on constructions related to the regular pentagon. A bit later he continues

*...the further we advance from the number one, the more perfect the example becomes. Let the smallest numbers be 1 and 1, which you must imagine as unequal. Add them, and the sum will be 2; add to this the greater of the 1's, result 3; add 2 to this, and get 5; add 3, get 8; 5 to 8, 13; 8 to 13, 21. As 5 is to 8, so 8 is to 13, approximately, and as 8 to 13, so 13 is to 21, approximately.*

*It is in the likeness of this self-developing series that faculty of propagation is, in my opinion, formed; and so in a flower the authentic flag of this faculty is flown, the pentagon.* The idea that live objects somehow obey Fibonacci sequences is all the more remarkable because it is true, the looping sequence of flowers, or the arrangement of the spirals of sunflower seeds for example.

Kepler mentions the divine proportion, now known as the golden ratio, which goes back at least as far as Pythagoras. In early Pythagorean geometry it was known as the extreme and mean ratio, which occurs when a line segment is so divided that the whole is to the greater segment as the greater is to the less. The ancient Greeks figured out that this was the ratio that symbolizes beauty – of all the rectangles that you could have, the rectangle with a side of length 1, and the other of length $\tau$ is the most pleasing to the eye. This proportion occurs throughout the Parthenon, in the works of Brunelleschi, Palladio or in the human being as evidenced in the well known depiction of the human figure by Leonardo da Vinci.

How did Kepler connect $\tau$ to Fibonacci sequence? Recall that for the standard Fibonacci sequence we have

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, ...$$

$$\frac{F_2}{F_1} = \frac{F_1 + F_0}{F_1} = 1 + \frac{1}{1}$$

$$\frac{F_3}{F_2} = \frac{F_2 + F_1}{F_2} = 1 + \frac{1}{F_2/F_1} = 1 + \frac{1}{1 + \frac{1}{1}}$$

$$\frac{F_4}{F_3} = \frac{F_3 + F_2}{F_3} = 1 + \frac{1}{F_3/F_2} = 1 + \frac{1}{1 + \frac{1}{1+\frac{1}{1}}}$$

and so on. Most people were not taught about these so-called *continued fractions* in school, but fortunately a wonderful exposition appears as two chapters in the second volume of Chrystal. If we think about what these ratios approach as $n$ grows larger and larger we notice that in each fraction appears as part of the succeeding fraction i.e. we have $x = 1 + \frac{1}{x}$, or, equivalently.

$$x^2 - x - 1 = 0$$

This quadratic polynomial equation has two solutions (if you don't remember how to solve this read the relevant part of Chrystal).

$$\frac{1 + \sqrt{5}}{2} = \tau = \lim_{n\to\infty} \frac{F_{n+1}}{F_n}$$

and

$$\frac{1 - \sqrt{5}}{2} = \sigma$$

$\lim\limits_{n\to\infty} \frac{F_{n+1}}{F_n} = \tau$ $\tau$ is the limit, but $\sigma$ is also important in our understanding of the sequence.

Now we will develop a simple mathematical theory *Fibonacci Sequences* or *Fibonacci-type sequences.* These are sequences where

$$a_i = a_{i-1} + a_{i-2}$$

and there are some initial values for $a_0$and $a_1$. The particular sequence that we get will depend on how we define the first two terms. So we have a whole family of sequences and we observe that we can perform operations upon them. For example given two Fibonacci sequences $\{a_i\}$ and$\{b_i\}$ then we claim that $\{c_i = a_i + b_i\}$ is also a Fibonacci sequence since

$$c_i = a_i + b_i = (a_{i-1} + a_{i-2}) + (b_{i-1} + b_{i-2}) = (a_{i-1} + b_{i-1}) + (a_{i-2} + b_{i-2}) = c_{i-1} + c_{i-2}$$

We can also make the same claim about the difference of two Fibonacci sequences or of the product of a constant $k$ times a Fibonacci sequence $\{ka_i\}$ (***Exercise***: proof). A mathematical structure that is equipped with addition and subtraction and that is closed under multiplication by "scalars" is known as a vector space in the case when scalars form a field, or more generally, a module over a ring. We have to always try to recognize familiar patterns. A vector space is a true design pattern. We now try to find a basis for this vector space of sequences. Recall that a basis is a minimal set of elements of the vector space such that every other element is a linear combination thereof. Observe that the first two values (together with the unchanging rule) determine the entire sequence, so the dimension (i.e. the number of elements in the basis) of the vector space is two. Note that one of our goals is to come up with a closed form formula for the $n$th Fibonacci number. E.g. wouldn't it be nice if something like $F_n = k^n$. We will not quite get this but we will get pretty close. Notice that

$$\tau^2 = 1 + \tau$$
$$\tau^3 = \tau + \tau^2$$
$$\tau^4 = \tau^2 + \tau^3$$

and so on. The powers of $\tau$, $\tau^i$ forms a Fibonacci type sequence. We can say the same for $\{\sigma^i\}$.These sequences are nice because we don't need to carry out $n$ additions to arrive at the $n$th element; instead we perform only the order lg n multiplications (using the Russian peasant algorithm for computing powers). The above two sequences are also a good basis for the vector space of Fibonacci sequences.

After Kepler Fibonacci sequences were mostly forgotten, except for isolated work by de Moivre, Euler and Lagrange, but none of them saw much significance there. Kepler, however, called the divine proportion one of the "one of the two Jewels of Geometry," the other being Pythagorean theorem. In the beginning of the 19th century, three names to remember are Binet, Lamé, and Lucas. Binet should be immortalized for inventing matrix multiplication. Toward the 1840s he decided to analyze the complexity of Euclid's algorithm. The first work on analysis of algorithms came from Binet and Lamé in analyzing Euclid's gcd algorithm.

*Remark* 6.15. The Fibonacci sequence is key in the analysis of the Euclid's algorithm because such numbers are exactly those for which the algorithm exhibits worst case performance. This is because, as we shall see when we study Euclid's algorithm later (see Corollary 7.25), for the algorithm to go slowly we require the smallest possible quotients at each stage. The smallest possible quotient is 1. But for all $n > 1$ the Fibonacci sequence always has $\lfloor F_{n+1}/F_n \rfloor = 1$, and the remainder is $F_{n-1}$so it if we start with a Fibonacci number the worst case performance will continue. That is, the “hardest adversary” for this gcd algorithm is generated by the Fibonacci sequence.

Binet figured this out in 1841, as did Lamé (independently) in 1844. Then in the 1880s Lucas, the greatest figure in recreational mathematics, who published 4 thick volumes thereon, invented towers of Hanoi puzzle, made contributions to number theory, analysis of algorithms, and came up with the largest known prime computed without computers: $2^{127} - 1$. In fact he wanted to build a computer. He was not as important to mathematics as Gauss or Cauchy but he did some great things, and his work was later very influential in computer science. He also developed a theory of Fibonacci numbers, which he called Lamé numbers. Now of course they are known as Fibonacci numbers and there is even a journal called the Fibonacci Quarterly.

How can we come up with a closed form formula for the nth Fibonacci number using our knowledge of the basis $\sigma = \{\sigma^i\}$ and $\tau = \{\tau^i\}$. If we can express “the” Fibonacci sequence $F = \{F_i\}$ as a linear combination of our two basis sequences, which as we mentioned above offer an easy way to calculate the $n$th element using $O(\lg n)$ multiplications then we will be done. That is, we would like to find two numbers $a$ and $b$ so that $a \cdot \tau + b \cdot \sigma = F$, or:

$$a \cdot \tau^0 + b \cdot \sigma^0 = F_0 = 0$$
$$a \cdot \tau^1 + b \cdot \sigma^1 = F_1 = 1$$
$$... = ...$$
$$a \cdot \tau^i + b \cdot \sigma^i = F_i$$

and so on. In fact, we only need to show that equality holds in the first two cases since if two Fibonacci sequences are equal at the first two elements then the entire sequences will be equal. So all we need to do is to solve for $a$ and $b$ below:

$$a + b = 0$$
$$a\tau + b\sigma = \frac{1+\sqrt{5}}{2}a + \frac{1-\sqrt{5}}{2}b = 1$$

Solving for $b$ in terms of $a$ in the first equation and then substituting for $b$ in the second equation gives:

$$a\tau + b\sigma = a\tau - a\sigma = \frac{1+\sqrt{5}}{2}a - \frac{1-\sqrt{5}}{2}a = 1$$

So $a = \frac{1}{\sqrt{5}}$and $b = \frac{-1}{\sqrt{5}}$.

$$a = \frac{1}{\sqrt{5}}, b = \frac{-1}{\sqrt{5}}$$

So at last we have our closed form formula to calculate the $n$th Fibonacci number in a logarithmic number of steps.

**Theorem 6.16.** *Binet Formula for the nth Fibonacci Number*

$$F_n = \frac{\tau^n - \sigma^n}{\sqrt{5}} = \frac{\tau^n - \sigma^n}{\tau - \sigma} \tag{6.4}$$

There are a number of formulas like the one in Theorem 6.16. For example, we can use the fact that $\sigma \approx 0.6$, to argue that $\sigma^n$ shrinks quite quickly, so that we can approximate $F_n$ as $\frac{\tau^n}{\sqrt{5}}$. We give the precise version below (see Knuth for a proof)

$$F_n = \left\lfloor \frac{\tau^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

(Recall that $\lfloor\rfloor$ denotes the floor function.) We would also like to be able to calculate $F_n$ using only integers (no floating point). This is not only possible, but elegant. It also illustrates the profound connection of Fibonacci sequences with linear algebra.

*Remark* 6.17. If we want to solve something we find out how this thing fits into linear algebra. We sometimes use various definitions of addition and multiplication when we do. Mapping into linear algebra is good. Linear algebra is a true design pattern.

Let us write a couple of our pairs $F_{n+1,}F_n$vertically instead of horizontally.

*Remark* 6.18. We try to look at a problem from a different angle.

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}\begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix}$$

**Exercise 6.19.** Before reading further, for a small exercise try drawing each of these vectors (based at the origin) on a piece of graph paper (experimental science).

When we see the two pairs written in the manner above we ask whether there might be a linear transformation (matrix) that maps the first pair to the second. Remarkably, there is such a transformation since

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 \cdot F_{n+1} + 1 \cdot F_n \\ 1 \cdot F_{n+1} + 0 \cdot F_n \end{pmatrix} = \begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix}$$

Alternatively we could contrive some symmetric matrices to represent our Fibonacci numbers

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$$

Notice that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$$

so we have the following fascinating identity

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$$

Applying the determinant to both sides of the above equation yields

$$\left|\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1}\right| = \begin{vmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{vmatrix}$$

in other words

$$(-1)^{n+1} = F_{n+2}F_n - F_{n+1}^2$$

another interesting relationship amongst Fibonacci numbers. Generalizing slightly we see that for *any* Fibonacci sequence, you can start with the vector of the two initial values and generate the entire sequence in the manner above.

For those who remember a little more about linear algebra, we remark that the $x^2 - x - 1$ equation pops up again if we consider the characteristic polynomial of the generating matrix

$$\begin{vmatrix} 1 - x & 1 \\ 1 & 0 - x \end{vmatrix} = x^2 - x - 1$$

Setting the polynomial to 0, we get eigenvalues equal to $\tau$ and $\sigma$. So it is all wonderfully connected. The sequence of vectors that we asked you to graph earlier converge to the associated eigenvector.

Let us consider implementations that calculate the *n*th Fibonacci number. We might be tempted to write a implementation that looks like this

```
int fib(int) {
  if(n==0) return 0;
  if(n==1) return 1;
  return fib(n-1) + fib(n-2);
}
```

Note that we use general recursion, as opposed to primitive recursion, but is very inefficient. It has exponential complexity. Even more amusing, fib(int n) defines its own complexity! The complexity of computing fib(n) is equal to the complexity of computing fib(n-1) plus that of computing fib(n-2). So the recurrence that we must solve to compute the complexity mirrors exactly the one that defines the very numbers that we are trying to calculate.

For some reason, many programmers in the functional programming community enjoy making this function as efficient as they can. One technique that they put forward is known as memoization, a technique that caches the result of these each calculation so that the complexity will be reduced to linear time. In this case such a technique is useless since we could always produce an iterative version that runs in linear time. But armed with the Russian peasant algorithm we can do better.

**Exercise 6.20.** Implement a logarithmic algorithm for computing Fibonacci numbers. Implement it using the Russian peasant algorithm to compute

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}$$

and then extract the result from the top left corner.

This is still not quite optimal. Remember that even though this is just a toy exercise – we will rarely want to efficiently calculate Fibonacci numbers – it illustrates a practical technique. A 2x2 matrix multiplication requires 8 multiplications in the general case. But in our case we can take advantage of the symmetry and the fact that we know how to calculate $F_{n+2}$ from $F_{n+1}$and $F_n$ to implement these matrices storing only 2 pieces of state. Let's christen such matrices Fibonacci matrices. With two pieces of state we will need far fewer multiplications and additions.

**Example 6.21.** implement Fibonacci matrices. They should have exactly the same interface as the 2x2 matrices from the previous exercise, but they should require fewer multiplications to carry out matrix multiplication and there should only be two data members. Implement `fib()` using the Russian peasant algorithm on these fibonacci matrices.

```
//Concepts: S models SemiRing
template <typename S>
struct fibonacci_matrix
{
  typedef S value_type;
  S a01;
  S a11;
  fibonacci_matrix(S x01, S x11)
    : a01(x01), a11(x11) {}
  fibonacci_matrix(S x)
```

```
    : a01(S(0)), a11(x) {}
};
template <typename S>
S matrix_element00(fibonacci_matrix<S>& m)
{
  return m.a01 + m.a11;
}
template <typename S>
S matrix_element01(fibonacci_matrix<S>& m)
{
  return m.a01;
}
template <typename S>
S matrix_element10(fibonacci_matrix<S>& m)
{
  return m.a01;
}
template <typename S>
S matrix_element11(fibonacci_matrix<S>& m)
{
  return m.a11;
}

template <typename S>
fibonacci_matrix<S> operator+(const fibonacci_matrix<S>& x,
                              const fibonacci_matrix<S>& y)
{
  return  fibonacci_matrix<S>( x.a01 + y.a01, x.a11 + y.a11 );
}

template <typename S>
fibonacci_matrix<S> operator*(const fibonacci_matrix<S>& x,
                              const fibonacci_matrix<S>& y)
{
  return  fibonacci_matrix<S>( (x.a01+x.a11)*y.a01 + x.a01*y.a11,
                                 x.a01*y.a01 + x.a11*y.a11 );
}
template <typename S>
S fibonacci_matrix_2(int n) {
  if (n == 0) return 0;
  if (n < 3) return 1;
  matrix_2<S> m(S(1), S(1), S(1), S(0));
  matrix_2<S> s = power(m, n-1);
//  to see how it works, uncomment the next two lines
//  std::cout << s.a00 << " " << s.a01 << std::endl;
//  std::cout << s.a10 << " " << s.a11 << std::endl;
  return matrix_element00(s);
}
```

```
template <typename S>
S fibonacci(int n) {
  if (n == 0) return 0;
  if (n < 3) return 1;
  fibonacci_matrix<S> m(S(1), S(0));
  fibonacci_matrix<S> s = power(m, n-1);
  return matrix_element00(s);
}
```

## 7. Elementary Number Theory

7.1. **Greatest Comon Divisor.** We would like to teach you everything that there is to know about number theory as it is such a wonderful part of mathematics. Of course this will not be possible, since the field is too large. Gauss, often known as the "prince of mathematics," called mathematics the "queen of the sciences," and considered number theory the "queen of mathematics". Sadly, nowadays you could attend high school and college without learning a thing about number theory. We will begin with a study of the greatest common divisor (gcd) algorithm. Our approach to teaching about the gcd algorithm will focus on what is known as an extended Euclid algorithm. Dirichlet called it Euler's algorithm. Weil called it Bachet's algorithm. The extended Euclid algorithm is based on the following remarkable fact that given integers $x, y$, there are integers $a, b$such that $ax + by = \gcd(x, y)$. The extended Euclidean algorithm will produce the coefficients $a, b$.

The (non-extended) Euclidean algorithm is found in Book 7, Propositions 1 and 2 of his *Elements*, though it is commonly believed to be known as long as 200 years earlier. Euclid, in spite of his interest in number theory, was primarily a geometer. The second important figure in the history of number theory is Diophantus who was indeed a number theorist. Then followed a long period of darkness except for perhaps the work Aryabhata who lived in India around 500 A.D. Aryabhata was the first person to invent and describe the extended Euclid algorithm, then known as kuttaka ("the pulverizer"). The renaissance of number theory in Europe started with the discovery in 1470 by Regiomontanus of the only extant manuscript of Diophantus's Arithmetic. He planned to publish it, and establish a major scientific publishing house. He didn't succeed. The Diophantus manuscript was eventually published in Greek in the 1550s, but it did not have much impact until 1621 when Claude Bachet published a Latin translation by Xylander with extensive commentaries. That was a key event in the development of modern number theory. One of the great contributions of Bachet was the rediscovery that you could indeed find the $a, b$mentioned above (Euclid did not discover this). Bachet refers to it without describing it in his notes on Diophantus. He planned to write a text on number theory, but he didn't. However in 1626 he published the first book of recreational mathematics, entitled "Problems Pleasant and Delectable", in which he described the extended Euclid algorithm and this is probably where Fermat picked it up. Fermat is the father of modern number theory. Fermat bought Bachet's edition of Diophantus by Bachet in which he wrote his famous marginal notes. The next major person figure in our story was Euler. Unlike Fermat, who was a magistrate, Euler was a professional mathematician. Euler read Fermat's marginal notes because Fermat's son published the text of Diophantus with his father's notes. Euler eventually independently rediscovered the extended Euclid algorithm, so it is often referred to (e.g., by Dirichlet) as Euler's algorithm.

In previous sections we implemented addition and multiplication of integers. The fact that we can't always divide numbers was eventually discovered. Fractions don't help us divide seven cows between three people. So division is not always possible. This fact, and the theory of divisibility, are at the center of number theory. Of course we can always divide with remainder (except when the divisor is 0). We will consider quotient and remainder . If we wish to divide an integer $a$ by an integer $b$ we can write

$$a = bq + r \tag{7.1}$$

with

$$0 \leq r < b. \tag{7.2}$$

Here $q$ is known as the *quotient* and $r$ is called the *remainder*. This is roughly how division of `ints` works in C++: there are operators for quotient "/", remainder "%", etc. where we must have:

```
a==b  *(a/b)+a%b
```

Every mathematician knows that whether $a$ is positive or negative, $r$ will be positive. For example, if we divide -1 by 5, mathematicians expect the remainder to be 4. Unfortunately, most programming languages do not guarantee that the remainder is between 0 and $b$. Most languages/implementations are allowed to return either positive or negative remainders. Computer architects often choose to round things down for quotient in terms of absolute value, so they get negative remainders. This sad fact causes no end of trouble. For this reason, following Knuth we introduce the symbol "mod" to represent the mathematical notion of remainder. That is, unlike %, we *require* that $0 \leq a \bmod b < b$. But we can limp along with this state of affairs as long as the following essential property remains assured

$$|a\%b| < |b| \tag{7.3}$$

or in C++

$$\text{abs}(a\%b) < \text{abs}(b).$$

We now consider the fundamental rules that we know about divisibility. In what follows we will use the symbol "|" in expressions such as $a|b$ to indicate that $a$ divides $b$. For the cases when $a$ does not divide $b$ we may write $a \nmid b$. We can easily see the following

$$\forall a \neq 0, a|a$$

$$\gcd(a, a) = a$$

$$\forall a \neq 0 \gcd(a, 0) = 0$$

The fact that Euclid uses in his gcd algorithm is

$$n|a \,\text{and}\, n|b \Rightarrow n|a + b \,\text{and}\, n|a - b$$

This can be seen by noting that

$$n|a \Rightarrow a = ni \,\text{for some}\, i, \text{and}\, n|b \Rightarrow b = nj \,\text{for some}\, j$$

so that

$$a + b = ni + nj = n\,(i + j)$$

and thus

$$n|a + b$$

A similar argument can be used to show that the $n|a - b$.

Now for positive integers (Euclid did not have negative numbers or zero) we can define the greatest common divisor as, so then

$$\gcd(a,b) = \begin{cases} \text{if } a = b & a \\ a < b & \gcd(a, b-a) \\ a > b & \gcd(a-b, b) \end{cases}$$

Consider now the set of integers divisible by $n$, which we will write as $n\mathbb{Z}$. Obviously the set of such numbers is closed under addition, subtraction and multiplication by other members of $n\mathbb{Z}$. Furthermore, $n\mathbb{Z}$ is closed under multiplication by *any* integer. A structure that enjoys these properties is called an *ideal.*

**Definition 7.1.** Ideal
An ideal, $I$, is a subset of a ring, $R$, that is

- closed under addition of other elements of $I$
- closed under multiplication by elements of $R$

**Definition 7.2.** Module structure of integers
A set of integers that is closed under subtraction is called a *module structure of integers*

**Theorem 7.3.** *Any module structure of integers is an ideal of* $\mathbb{Z}$

*Proof.* Let $a, b \in I$ where $I$ is a module structure of integers. Then, since by definition $I$ is closed under subtraction we have

$$a - a = 0 \in I$$

and thus

$$0 - a = -a \in I$$

consequently

$$a - (-a) = 2a \in I$$

and thus

$$2a - (-a) = 3a \in I$$

Continuing in this manner we see that all multiples of $a$ are contained in $I$, so $I$ is closed under multiplication by elements of $\mathbb{Z}$. Then

$$b \in I \implies -b \in I \implies a - (-b) = a + b \in I$$

so $I$ is closed under addition too, and thus $I$ is an ideal of $\mathbb{Z}$. □

**Theorem 7.4.** *Any ideal I in* $\mathbb{Z}$, *is closed under the remainder operation*

*Proof.* Recall that

$$a\%b = a - b * \lfloor a/b \rfloor$$

where $\lfloor\rfloor$ denotes the floor function. But $a, b \in I \implies (-1)b \in I$ therefore any multiple of $-b$ by an element of $R$ is in $I$. So $-b * \lfloor a/b \rfloor$ is in $I$ and thus so is $a - b * \lfloor a/b \rfloor$. (Here we use $\lfloor a/b \rfloor$ which in C++ will simply be written as $a/b$). □

**Definition 7.5.** Principal ideal
The set $(a) = \{ar : r \in R\}$ of all multiples of a ring element $a$ is called the *principal ideal generated by a.*

**Theorem 7.6.** *Every ideal in* $\mathbb{Z}$ *is a principal ideal generated by an element of smallest absolute value*

*Proof.* Let $I \neq 0$ be an ideal in $\mathbb{Z}$. Since every subset of the integers has an element of smallest absolute value, we can select one, say $b \in I$. Let $a \in I$. Then, as in Equation 7.1 and Equation 7.2 we can write

$$a = bq + r$$

with

$$0 \leq r < b$$

where, by Theorem 7.4, we have $r \in I$. But $b$ was selected to be have the smallest positive absolute value, so we see that $r$ must be 0 and thus any element $a$ must be a multiple of $b$. □

**Corollary 7.7.** *Every element of a pricnipal ideal is divisible by the principal element.*

*Proof.* Trivial since by defintion every element is a multiple of the principal element. □

**Theorem 7.8.** *For any a, b ∈ $\mathbb{Z}$, the set S, of linear combinations of a and b, is an ideal.*

*Proof.* Suppose $x_0, x_1, y_0, y_1 \in \mathbb{Z}$. We see that $S$ is closed under addition since

$$(ax_0 + by_0) + (ax_1 + by_1) = a(x_0 + x_1) + b(y_0 + y_1).$$

Also, $S$ is closed under multiplication by elements of $\mathbb{Z}$ since for $x, y, k \in \mathbb{Z}$ we have

$$k\,(ax + by) = a\,(kx) + b\,(ky)\,.$$

Thus $S$ is an ideal. □

**Corollary 7.9.** *For any a, b ∈ $\mathbb{Z}$, every element in the set S of linear combinations of a and b is divisible by the smallest element, c*

*Proof.* Follows from Theorem 7.8 and Theorem 7.6. □

**Corollary 7.10.** *Any common divisor of a and b divides all elements of the set S of linear combinations of a and b. In particular this includes c, the smallest element of S.*

*Proof.* If $d|a$ and $d|b$ then $d$ will divide any linear combination of $a$ and $b$ and so $d|c$. □

**Corollary 7.11.** *For $a, b \in \mathbb{Z}$ the smallest positive element in the set S of linear combinations of a and b is the gcd(a,b).*

*Proof.* Follows directly from the previous two corollaries. □

The last sequence of Theorems shows us that we gcd(a,b) always exists for $a, b \in \mathbb{Z}$ and that it can be written as a linear combination of $a$ and $b$. We restate this for the case where the gcd is 1 as it will be quite useful to us in what follows.

**Corollary 7.12.** *For $a, b \in \mathbb{Z}$*

(1) $\gcd(a, b)$ *always exists, it divides every linear combination of a and b and it is the minimum positive such linear combination.*
(2) $\forall x, y \in \mathbb{Z} : \gcd(a, b)|ax + by$
(3) $\exists x, y \in \mathbb{Z} : \gcd(a, b) = ax + by$

*Proof.* Directly follows from Corollary 7.11. □

**Corollary 7.13.** $\gcd(a, b) = 1$ *if and only if there exists some x and y such that* $xa + by = 1$

*Proof.* Follows from the previous Corollary. □

Now we move towards an implementation of a remainder based gcd algorithm. If we start with $a_1$ and $a_2$ then if $a_2 \neq 0$ then we can write

$$a_1 = a_2 \left\lfloor \frac{a_1}{a_2} \right\rfloor + a_3 \tag{7.4}$$

for some number $a_3$with $0 \leq a_3 < a_2$, and . Then either $a_3$ is 0 or else we can write

$$a_2 = a_3 \left\lfloor \frac{a_2}{a_3} \right\rfloor + a_4 \tag{7.5}$$

and so on until we arrive at

$$a_{n-2} = a_{n-1} \left\lfloor \frac{a_{n-2}}{a_{n-1}} \right\rfloor + a_n \tag{7.6}$$

$$a_{n-1} = a_n \left\lfloor \frac{a_{n-1}}{a_n} \right\rfloor + 0 \tag{7.7}$$

***(Exercise:*** Why do we know that this process will terminate?) From Equation 7.7 we see that $a_n | a_{n-1}$. We can then use this fact together with Equation 7.6 to see that $a_n$also must divide $a_{n-2}$. Continuing all the way back up to the top (Equation 7.4) in this manner back up we see that $a_n | a_1$(and also $a_n | a_2$). So $a_n$ is a common divisor of $a_1$and $a_2$. We can also proceed from top to bottom in the above set of equations to see that any divisor of both $a_1$and $a_2$ will divide $a_3$and hence will divide $a_4$and so on. That is any divisor of $a_1$and $a_2$ will also divide $a_n$. Thus $a_n$ is not only a common divisor, it is the greatest common divisor of $a_1$and $a_2$. This leads us to Euclid's beautiful definition.

**Definition 7.14.** Greatest common divisor

A greatest common divisor of two numbers is a divisor that is divisible by every other divisor. See Euclid book VII, proposition 2 , porism

Notice that this definition doesn't even mention ">".

To see how to implement an algorithm for computing gcd$(a, b)$ based on the above, we first rewrite Equation 7.4 through Equation 7.7 as

$$a_3 = a_1 - a_2 \left\lfloor \frac{a_2}{a_3} \right\rfloor \tag{7.8}$$

$$a_4 = a_2 - a_3 \left\lfloor \frac{a_2}{a_3} \right\rfloor \tag{7.9}$$

and so on until we arrive at $a_{n+1} = 0$

$$0 = a_{n-1} - a_n \left\lfloor \frac{a_{n-1}}{a_n} \right\rfloor \tag{7.10}$$

or more succinctly, using the % operation

$$a_3 = a_1 \% a_2 \tag{7.11}$$

$$a_4 = a_2 \% a_3 \tag{7.12}$$

and so on until we arrive at $a_{n+1} = 0$

$$0 = a_{n-1} \% a_n \tag{7.13}$$

where, as before, $a_n = \gcd(a_1, a_2)$.

We can discern a pattern in Equation 7.11 to Equation 7.13, namely that for $i > 2$, we have$a_i = a_{i-2}\%a_{i-1}$.This suggests that we move towards an implementation of $\gcd(a, b)$ by rewriting the sequence:

$$a_1, a_2, a_{3,}..., a_{n,}a_{n+1}$$

as

$$a, b, a \leftarrow a\%b, b \leftarrow b\%a, a \leftarrow a\%b, b \leftarrow b\%a, ..., 0 \tag{7.14}$$

By the arguments above, the penultimate term in sequence 7.14 will be the $\gcd(a, b)$.This leads us to directly to our implementation. We introduce a concept *Euclidean Domain* as any ring for which Euclid's algorithm always terminates. Mathematical texts give a more complicated definition which is totally equivalent to ours.

```
template <typename R> // R is a Euclidean domain
R gcd(R a, R b){
   while(true){
      if(b == 0) return a;
      a %= b;
      if(a == 0) return b;
      b %= a;
   }
}
```

The above presentation is a bit different from our usual implementation, but it has the nice feature that by unrolling the loop once we can avoid the need to swap *a* and *b*

Now we get to an interesting part. For a given *a* and *b*, how can we determine an *x* and *y* that solve the equation $ax + by = \gcd(a, b)$. It turns out that this is very important computationally. The remarkable fact that it is always possible to write the gcd(a,b) as a linear combination of *a* and *b* became evident to Euler through his work with continued fractions. We will slightly extend the reasoning above to produce an implementation that finds such coefficients.

It will suffice to compute just one of the coefficients of the linear combination, since if we know that $ax + by = d$, and if we can determine *x*, then we can calculate $y = (d - ax)/b$. So we only really need the first coefficient. Furthermore many times we are given numbers *a* and *n* with $\gcd(a, b) = 1$ and we want to find *x* so that $ax = 1(\bmod n)$. Such an *x* is called a (multiplicative) inverse of *a*, which we will often write as $a^{-1}$. For such problems we only care to find *x*, not *y*. So we will try to implement `extended_inverse` that will return *x* and the gcd, but not *y*. In case $\gcd(a, n) \neq 1$ then there is no inverse for $a(\bmod n)$. But we can still view *x* as an "extended inverse" since we will have $ax = \gcd(a, n)(\bmod n)$.

We will begin from the (less succinct) equations 7.8-7.10 that led us to our gcd implementation and see if we can find out how to determine *x*. From these equations we observe the pattern that for $i > 2$, we have

$$a_{i+1} = a_{i-1} - \left\lfloor \frac{a_{i-1}}{a_i} \right\rfloor a_i \tag{7.15}$$

As before, this suggests rewriting our sequence

$$a_1, a_2, a_{3,}..., a_{n,}a_{n+1}$$

as

$$a \leftarrow m, b \leftarrow n, a \leftarrow a - \left\lfloor \frac{a}{b} \right\rfloor b, b \leftarrow b - \left\lfloor \frac{b}{a} \right\rfloor a, a \leftarrow a - \left\lfloor \frac{a}{b} \right\rfloor b, b \leftarrow b - \left\lfloor \frac{b}{a} \right\rfloor a, ..., 0 \tag{7.16}$$

where once again the penultimate term will be the gcd$(m, n)$. This time we carry our calculations a little bit farther however, by keeping track of how to write each term as a linear combination of $m$ and $n$. This is easy to do for the first two terms:

$$1m + 0n, 0m + 1n$$

From this point forward, the i-th term is obtained from the previous two by the rule

$$a_i = a_{i-2} - q_{i-2}a_{i-1} \tag{7.17}$$

where $q_{i-2} = \lfloor \frac{a_{i-2}}{a_{i-1}} \rfloor$, so we that we can now write out the next terms in our gcd sequence as linear combinations of $m$and $n$ as follows

$$1m + 0n, 0m + 1n, 1m - q_1n, -q_2m + (1 + q_1q_2)\,n, ..., 0 \tag{7.18}$$

In each case we use the rule in Equation 7.17 to figure out the coefficients of $m$ and $n$ from the coefficients of the previous two linear combinations. That is, let $x_i$ denote the coefficient of $m$ in the $i$-th term of Equation 7.18 , and let $y_i$ be the corresponding coefficient for $n$. Then we write $a_i$ as a linear combination of $m$ and $n$

$$a_i = x_im + y_in$$

We can determine how the $x_i$'s and $y_i$'s relate to their predecessors, by examining the previous two elements of the gcd sequence

$$a_{i-2} = x_{i-2}m + y_{i-2}n \tag{7.19}$$

and

$$a_{i-1} = x_{i-1}m + y_{i-1}n$$

Then we use Equation 7.17 to see that

$$\begin{aligned} a_i &= (x_{i-2}m + y_{i-2}n) - q_{i-2}\,(x_{i-1}m + y_{i-1}n) \\ &= (x_{i-2} - q_{i-2}x_{i-1})\,m + (y_{i-2} - q_{i-2}y_{i-1})\,n \end{aligned}$$

so that

$$x_i = x_{i-2} - q_{i-2}x_{i-1} \tag{7.20}$$

and

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

In particular, we know from Equation 7.18 that $x_1 = 1$and $x_2$= 0, and from Equation 7.20 we see how for $i > 2$, $x_i$ is obtained from the previous two such coefficients. This suggests that we can write the sequence $\{x_i\}$of coefficients as

$$x_1 \leftarrow 1, x_2 \leftarrow 0, x_3 \leftarrow x_1 - q_1x_2, x_4 \leftarrow x_2 - q_2x_3, ..., x_{n+1} \tag{7.21}$$

where by the reasoning above, the penultimate term, $x_n$is equal to the coefficient of $m$ in the linear combination for

$$\gcd(m, n) = a_n = x_nm + y_nn$$

As in 7.16 we take advantage of the fact that each term only depends on the previous two to eliminate necessary variables rewriting sequence 7.21 as

$$u \leftarrow 1, v \leftarrow 0, u \leftarrow u - q_1v, v \leftarrow v - q_2u, ..., x_{n+1} \tag{7.22}$$

But we are to keep track of $u$ and $v$ to produce the coefficient $x$ at the same time as we keep track of $a$ and $b$ to compute the gcd$(m, n)$so we replace the values $q_i$ in the above sequence with those from 7.16 to rewrite this as

$$u \leftarrow 1, v \leftarrow 0, u \leftarrow u - \left\lfloor \frac{a}{b} \right\rfloor v, v \leftarrow v - \left\lfloor \frac{b}{a} \right\rfloor u, u \leftarrow u - \left\lfloor \frac{a}{b} \right\rfloor v, v \leftarrow v - \left\lfloor \frac{b}{a} \right\rfloor u, ..., x_{n+1} \tag{7.23}$$

This leads directly to our implementation of extended inverse.

```
//Concepts: R models EuclideanDomain
template <typename R>
pair<R, R> extended_inverse(R a, R b){
  // initially a == m, b == n
  R u(1);
  R v(0);
  while(true){
                       // exists(w), u*m + w*n == a
                       // exists(z), v*m + z*n == b
  if(b == 0) return pair<R, R>(u, a);
    R q = a/b;
    a -= q*b;
    u -= q*v;
                       // exists(w), u*m + w*n == a
                       // exists(z), v*m + z*n == b
  if(a == 0) return pair<R, R>(v, b);
    q = b/a;
    b -= q*a;
    v -= q*u;
  }
}
```

For the invariants in the code above, *m* and *n* are the values originally passed in for *a* and *b* respectively. We optimize the above implementation by moving the expression “$u- = q*v$;” below the following if statement, and similarly for the expression "$v- = q*u$;".

```
//Concepts: R models EuclideanDomain
template <typename R>
pair<R, R> extended_inverse(R a, R b)
{
  // initially a == m, b == n
  if (b == 0) return pair<R, R>(R(1), a);

  R u(1);
  R v(0);

  while (true) {
                       // exists(w), u*m + w*n == a
                       // exists(z), v*m + z*n == b
    R q = a/b;
    a -= q*b;
    if (a == 0) return pair<R, R>(v, b);
    u -= q*v;
                       // exists(w), u*m + w*n == a
                       // exists(z), v*m + z*n == b
    q = b/a;
    b -= q*a;
    if (b == 0) return pair<R, R>(u, a);
```

```
        v -= q*u;
    }
}
```

**Exercise 7.15.** Figure out what is the probability of 2 integers less than 100,000 that their gcd = 1

**Exercise 7.16.** Try to figure out what the probability that gcd will equal 2, 3, ... . Write code to do this, don't try to solve it mathematically.

This will demonstrate that there are many numbers whose gcd is 1.

**Definition 7.17.** Relatively Prime (Coprime)
Two integers are *relatively prime* if their gcd is 1. We sometimes express this relationship using the term *coprime*, for example we might say $a$ and $b$ are coprime, or equivalently, $a$ is coprime to $b$. We now demonstrate a few of theorems about coprimes that we will use later.

**Theorem 7.18.** *Closure of Coprimes Under Multiplication*
*If*

$$\gcd(a, n) = 1 \tag{7.24}$$

*and*

$$\gcd(b, n) = 1 \tag{7.25}$$

*then*

$$\gcd(ab, n) = 1.$$

*Proof.* We know from Corollary 7.13 and Equation 7.24 that there exist some numbers $x$ and $y$such that

$$xa + yn = 1. \tag{7.26}$$

Similarly, from Corollary 7.13 and Equation 7.25 we know that there are some number $z$ and $w$ such that

$$zb + wn = 1. \tag{7.27}$$

Multiplying Equation 7.26 and Equation 7.27 gives

$$(xz)\, ab + (xwa + ywn + yzb)\, n = 1. \tag{7.28}$$

□

One more application of Corollary 7.13 with Equation 7.28 tells us $\gcd(ab, n) = 1$. If we relax the assumption that $\gcd(b, n) = 1$ in Theorem 7.18 but follow the almost the same proof we will have a proof of the following generalization

**Theorem 7.19.** *If* $\gcd(a, n) = 1$ *then* $\gcd(ab, n) = \gcd(b, n)$

*Proof.* Exercise. Hint: Use Corollary 7.11 □

**Exercise 7.20.** (From Chrystal) If $a, b, A, B \in \mathbb{Z}$ where $a/b$ is an fraction in lowest terms and $a/b = A/B$ show that $\exists n \in \mathbb{Z} : A = na, B = nb$.

**Corollary 7.21.** *If* $\gcd(a, n) = 1$ *and* $n \nmid b$ *then* $n \nmid ab$

*Proof.* We prove the contrapositive: If $n | ab$ then $\gcd(ab, n) = n$which in turn equals $\gcd(b, n)$ by Theorem 7.19. But $\gcd(b, n) = n \implies n | b$. □

Notice that it is much easier to discover if two numbers are coprime than it is to determine if a number is prime. We can use the above results to find all integer solutions to linear equations of the form $ax + by = c$ as demonstrated by the next two Theorems.

**Theorem 7.22.** *For $a, b, c \in \mathbb{Z}$ an equation of the form $ax + by = c$ has an integer solution $x, y \iff \gcd(a, b)|c$.*

*Proof.* Direct consequence of Corollary 7.12. □

**Theorem 7.23.** *For $a, b, c \in \mathbb{Z}$ the general integer solution $x, y$ of the equation*

$$ax + by = c \tag{7.29}$$

*is of the form*

$$\begin{cases} x = x_0 + t\frac{b}{\gcd(a,b)} \\ y = y_0 - t\frac{a}{\gcd(a,b)} \end{cases} \tag{7.30}$$

*where $x = x_0, y = y_0$ is a particular solution and t runs through all values of $\mathbb{Z}$.*

*Proof.* Let $x_0, y_0 \in \mathbb{Z}$ be a particular solution to Equation 7.29 and assume that $x', y' \in \mathbb{Z}$ is another solution. Then $ax_0 + by_0 = c$ and $ax' + by' = c$ so we can subtract the corresponding parts of the first equation from the second to obtain the equation $a(x' - x_0) + b(y' - y_0) = 0$. That is we see that for any solution $x', y' \in \mathbb{Z}$ we will have $m_0 = x' - x_0, n_0 = y' - y_0$ as a solution to the *homogeneous* linear equation

$$am + bn = 0. \tag{7.31}$$

Furthermore, we claim that any integer solution $m, n$ of Equation 7.31 can be added to $x_0, y_0$ to obtain a solution to Equation 7.29. For by adding Equation 7.31 to $ax_0 + by_0 = c$ we get

$$a(x_0 + m) + b(y_0 + n) = c.$$

So the solutions to Equation 7.29 will be all $x, y$ of the form

$$\begin{cases} x = x_0 + m \\ y = y_0 + n \end{cases} \tag{7.32}$$

where $m, n$ run through all integer solutions to Equation 7.31. But

$$am + bn = 0 \iff \frac{a}{\gcd(a,b)}m + \frac{b}{\gcd(a,b)}n = 0 \iff \frac{a}{\gcd(a,b)}m = -\frac{b}{\gcd(a,b)}n.$$

Since $a/\gcd(a, b)$ and $b/\gcd(a, b)$ are relatively prime integers we can see by Theorem 7.19 that the above holds if and only if $b/\gcd(a, b)|m$ and $a/\gcd(a, b)|n$ with

$$t\frac{b}{\gcd(a,b)} = m$$

and

$$t\frac{a}{\gcd(a,b)} = -n.$$

Putting this together with Equation 7.32 completes the proof. □

Before turning our attention away from the gcd we explore the worst case performance of Euclid's algorithm.

**Lemma 7.24.** *If a and b are positive with $a > b$ where $a < F_{n+2}$* or *$b < F_{n+1}$ then Euclid's algorithm for* $\gcd(a, b)$ *will perform less than n division operations .*

*Proof.* We prove the contrapositive. That is, we will show that given if Euclid's algorithm performs $n$ divisions then $a \geq F_{n+2}$ and $b \geq F_{n+1}$. We proceed by induction. The base step is easy since for $n = 1$ we have $a \geq 2 = F_{n+2}$ since $a > b \geq 1$ and clearly $b \geq 1 = F_{n+1}$. For the inductive step we must demonstrate that the Lemma holds for $n$ when we are given that it holds for $n - 1$. In the general case the $\gcd(a, b)$ works by first calculating $a \bmod b$, at a cost of 1 division, then operates in the same manner as $\gcd(b, a \bmod b)$, at a cost of $n - 1$ additional division operations. By the induction hypothesis we can conclude that since $\gcd(a, a \bmod b)$ costs $n - 1$ divisions we must have $b \geq F_{n+1}, a \bmod b \geq F_n$. Then, since $a > b$ we have $b + a \bmod b \leq a$. Substituting we get $F_{n+1} + F_n \leq a$, so $a \geq F_{n+2}$. □

**Corollary 7.25.** *(Lamé) If $a$ and $b$ are positive with $a > b$ where $b < F_{n+1}$ then Euclid's algorithm for* $\gcd(a, b)$ *will perform less than $n$ division operations .*

*Proof.* Immediate from the previous Lemma. □

We know that Corollary 7.25 gives the tightest upper bound on the size of $b$ since we know that $\gcd(F_{n+2}, F_{n+1})$ will require $n$ divisions (see Remark 6.15). So the number of divisions needed given a second argument of $b$ will be the largest $n$ with $F_{n+1} < b$. In order to figure $n$ given $b$ we note that by one form of the Binet Formula (Theorem 6.16 ) we have

$$b > F_{n+1} = \left\lfloor \frac{\tau^{n+1}}{\sqrt{5}} + \frac{1}{2} \right\rfloor \geq \frac{\tau^{n+1}}{\sqrt{5}}.$$

Taking $\log_{10}$ of both sides gives:

$$\log_{10} b \geq \log_{10} \frac{\tau^{n+1}}{\sqrt{5}} = (n + 1) \log_{10} \tau - \log_{10} \sqrt{5} > (n + 1) \log_{10} \tau$$

(since $\log_{10} \sqrt{5}$ is negative). But $\log_{10} \tau \approx 0.208988 > 1/5$ so we arrive at

$$n < \frac{\log_{10} b}{\log_{10} \tau} - 1 < (5 \log_{10} b) - 1$$

In other words, the number of remainder operations is less than 5 times the number of decimal digits in the smaller argument.

7.2. **Primes.** The Greeks were apparently the first people to look at primes as special numbers. First Euclid proved the remarkable fact that there are infinitely many prime numbers. As a reminder of as proof, suppose that we have primes $p_1, ..., p_n$. Multiplying all of them together and adding 1

$$\prod_{i=0}^{n} p_i + 1$$

This number is not divisible by any $p_i$ then either it is prime or there must be another prime that divides it. In either case there is a prime that is not in the original list. Deep down this proof uses the fact that any number is decomposable into primes. Indeed this is an important fact that we will prove later on.

For now we consider how many primes there are (without proof). Could there be primes very far apart, and if so, how big could the gaps become. The answer is that the gaps can become arbitrarily large. Too see this we start with a sequence of numbers related to $n!$

$$n! + 2, ...n! + n$$

None of these are prime, since 2 divides first element, 3 divides the next, and so on.

There are also remarkable primes that are close to each other. Only two primes are within one of each other, namely 2 and 3. Primes that are 2 apart are known as twin primes. The question of how many twin primes there are remains open. In about 1920 Vigo Brun proved a remarkable result – the sum of the series of reciprocals of twin primes converges to a very small number known as the Brun constant $\approx$ 1.9. So even though we don't know how many there are we know the sum of their reciprocals! It is amusing to note that the Intel chip floating point errors were discovered by someone attempting to compute Brun's constant.

Thanks to Fermat, we know that there are infinitely many primes of the form $4k + 1$and also of the form $4k + 3$. What about arbitrary arithmetic progressions $ak + b$ with $a$, $b$ coprime? There are also infinitely many of this form – the difficult proof of this by Dirichlet required analysis to prove it and gave birth to analytic number theory. Euler proved that the sum of the reciprocals of *all* primes diverges. So primes are not that rare, since for example the sum of reciprocals of squares converges. There is a famous question: how many primes less than a given number $n$ are there? This answer to this question is known as the prime number theorem. Around the 1790s Gauss realized that $\pi$(n), the numbers of primes less than $n$was approximately

$$\pi(n) \approx \frac{n}{\ln n} \tag{7.33}$$

But Gauss did not publish this. Then in the early 1800s Adrien-Marie Legendre published the estimate

$$\pi(n) \approx \frac{n}{\ln n + B} \tag{7.34}$$

Later Gauss came up with better bound

$$\pi(n) \approx \mathrm{Li}(n) = \int\limits_{t=2}^{n} \frac{1}{\ln t} dt$$

There was a great race to prove it. Around 1850 Pafnuty Lvovich Chebeyshev proved that

$$\frac{C_1}{\ln n} < \pi(n) < \frac{C_2}{\ln n}$$

where $C_1 = 7/8$, $C_2 = 9/8$ (in other words he was one quarter shy of the prize). He also proved that if it converged to a limit than the limit is 1. Then there were fifty more years of trying to figure out how to prove it. Finally in 1898 using advanced techniques of analytic number theory, two people independently proved it. Jacques Hadamard, a great French mathematician (buy his two-volume text of geometry if you can find it). It was also proved independently by Charles Jean Gustave Nicolas Baron de la Vallée Poussin. For many years, many mathematicians believed that

$$\pi(n) < \mathrm{Li}(n)$$

But then Littlewood showed that these two curves cross infinitely often. The first crossing occurs at some astronomically large number. For the "elementary" proof (Erdös-Selberg) of the Prime Number Theorem see [16] p. 340.

**Theorem 7.26.** *Prime Number Theorem*

*The number of primes less than n, $\pi(n)$ is approximated by*

$$\pi(n) \approx \frac{n}{\ln n}$$

Given the prime number theorem we can get a better idea of the size of the gaps. For example if we are looking at a hundred decimal digit number $n$, then we can calculate the average size of a gap between primes using the prime number theorem since if there are

$$\frac{n}{\ln n}$$

primes then the gaps between primes will on average be of size $\ln n$. In our case

$$\ln n = (\log_{10} n)(\ln 10) \approx 100 * .27$$

In other words, if we are interested in looking for large primes we may pick a 100 decimal digit candidate number at random. Even if it is not prime the above argument suggests that we will succeed in finding a nearby prime before testing a few hundred neighbors (less if we eliminate obvious composite). We will make use of this fact in our RSA implementation. [Note: Meissel technique]

Chebeyshev also proved in 1850:

**Theorem 7.27.** *Bertrand's Postulate*
*Between any n and 2n there is a prime.*

Of course we suspect there are more but this is our tightest estimate. If you could do better you would be famous.

**Exercise 7.28.** Generate all of the primes from 1 to $n$. Use the Sieve of Eratosthenes. Eratosthenes was roughly a contemporary of Archimedes. Born in Libya, he became a mathematician, and was one of the greatest astronomers of all time. His accomplishments included measuring the diameter of the earth. His estimate was amazingly accurate for his time. He also estimated the distance to the sun and moon. For some reason he was disliked by his contemporary scientists and was given the nickname beta (not alpha, second in all things). He wrote an interesting text on arithmetic but it is no longer extant. We have third-hand knowledge that in this text he introduced the algorithm now named the Sieve of Eratosthenes. Here is how it works. We start with the sequence of numbers

$$2, 3, 4, ...n.$$

Starting with 2 we cross out all multiples of two. Then starting with 3 we cross out multiples of 3. We don't want to keep track of which numbers have already been crossed out. It is sufficient to stop at $\sqrt{n}$ (this was figured out by Leonardo Pisano). Every number remaining in the table is prime. For extra credit do not store the even numbers, so as to minimize the size of the table.

We will now prove a remarkable fact, that was first explicitly proved by Gauss, known as the fundamental theorem of arithmetic. it is very important and we will use it all of the time. Gauss is very fortunate, there is the fundamental theorem of arithmetic and the fundamental theorem of algebra and he proved them both. There is no fundamental theorem of geometry, but there is one for calculus, but Gauss didn't prove it. Gauss was the person who codified number theory. Number theory existed prior to Gauss but there was no standard text or language. One of Gauss's greatest accomplishments was the publication in Latin in 1801 of his *Disquisitiones Arithmaticae* (investigations in arithmetic) [17]. In it there is a proof that every number is uniquely decomposable into primes

**Theorem 7.29.** *Fundamental Theorem of Arithmetic*
*A composite number can be decomposed into primes in exactly one way*

*Proof.* We call the decomposition into primes

$$p_1^{k_n}...p_n^{k_n}$$

the *canonical decomposition* into primes if

$$i < j \Rightarrow p_i < p_j$$

For the proof we will use the *extended decomposition*, where we write all of the $p_i's$ out. Let us assume that there are numbers with two distinct prime decompositions. Then there is a smallest such number, since every subset of the natural numbers contains a smallest element. We will construct an even smaller one to obtain a contradiction and so conclude that there can be no numbers with two distinct prime decompositions. Let $N$ be a number with two distinct prime decompositions

$$N = p_1...p_u$$

$$N = q_1...q_v$$

We know that no $p_i$ equals any $q_j$ for otherwise we could easily construct a new smaller number with two distinct decompositions, contrary to the assumption that $N$ is smallest such number. Without loss of generality, we can then assume that $p_1 < q_1$.Consider the number

$$M = (q_1 - p_1)\, q_2...q_v \tag{7.35}$$

$M$ is smaller then $N$. Multiplying it out

$$M = q_1...q_v - p_1q_2...q_v$$

and using the fact that the product of the $q_i's$ is equal to $N$, which in turn equals the product of the $p_i's$:

$$M = p_1...p_u - p_1q_2...q_v$$

or

$$M = p_1(p_2...p_u - q_2...q_u) \tag{7.36}$$

In Equation 7.35 a decomposition is given for $M$ with no factors divisible by $p_1$ (since no $p_i$ has a factor in common with any $q_j$, and since $p_1$ is not a factor of $q_1 - p_1$). *Exercise: why?.* In Equation 7.36 a decomposition involving $p_1$is given for $M$. Thus $M$ can be decomposed as a product of primes in two different ways, one way involving $p_1$and one way without $p_1$giving us the desired contradiction. □

Gauss was extremely proud of this theorem. He even recommended calculating the gcd from the prime decompositions of both numbers [17]. It is somewhat funny to try to obtain a result easily computable via Euclid's algorithm with the help of an exponential algorithm for computing prime factorizations. "Given many numbers $A, B, C$, etc the greatest common divisor is found as follow. Let all the numbers be resolved into their prime factors, and from these extract the ones which are common to $A, B, C$, etc..." At least he also remarks. "Moreover, we know from elementary considerations how to solve these problems when the resolution of the numbers $A, B, C$ etc. into factors is not given." Despite the fact that Gauss doesn't mention Euclid (one might detect a hint of jealousy there), his successor at the University of Göttingen, Dirichlet, acknowledges in his number theory book that "It is now clear that the whole structure of number theory rests on a single foundation, namely the algorithm for finding the greatest common divisor of two numbers."

We conclude this section with a brief discussion of *Fermat Primes.* Fermat conjectured that all numbers of the form $2^{2^n} + 1$ are prime. Such numbers are known as *Fermat primes.*

We do not know why he suddenly became interested in these. It turns out the first five such numbers are in fact prime:

$$2^{2^0} + 1 = 1, 2^{2^1} + 1 = 5, 2^{2^2} + 1 = 17, 2^{2^3} + 1 = 257, 2^{2^4} + 1 = 65537 \tag{7.37}$$

Sadly these are the only primes of such form known to humanity. It took nearly a century before Euler discovered that the next number,

$$2^{2^5} + 1 = 4294967297 = 641 \times 6700417$$

was not prime. In any case, at that time this question was merely a curiosity. But there was an ancient problem from Greek geometry that remained unsolved: for which numbers $n$ can you construct a regular $n$-sided polygon using (unmarked) ruler and compass. Triangles and square can be constructed easily enough. Pentagons can be constructed with some difficulty. Hexagons can also be constructed but then there is a gap. Gauss eventually showed that you could construct a regular 17-gon. He was very proud of this particular construction as it had eluded people for 2000 years. In general to construct a regular $n$-gon, $n$ must be of the form $2^k pq$where $p$ and $q$ are distinct Fermat primes (or 1). What an amazing connection with geometry! From that point on everybody wanted to find out if there are infinitely many Fermat primes, or if there are only those in Equation 7.37 or even if there is another one. We still do not know.

7.3. **Modular Arithmetic.** Many questions in arithmetic can be reduced to questions about remainders that are more easily answered. For each integer $n$ there is an arithmetic "mod $n$" that is similar to ordinary arithmetic but it finite, involving only the remainders $0, 1, ..., n-1$ upon division by $n$. We study such arithmetic from a few different perspectives in this section.

We begin by studying the multiplication table used in regular arithmetic

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 |
| 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 |
| 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 |
| 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 |
| 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 |
| 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 |
| 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 |
| 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 |

A lot of number theory can be derived from a careful study of tables like this. For example, after multiplying we will divide each entry by the number 10, here called the *modulus*, and consider only the remainders (this is easy to do for a modulus of 10, we simply delete all but the right-most digit). We obtain a new table, the multiplication table *modulo 10*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 4 | 6 | 8 | 0 | 2 | 4 | 6 | 8 |
| 3 | 6 | 9 | 2 | 5 | 8 | 1 | 4 | 7 |
| 4 | 8 | 2 | 6 | 0 | 4 | 8 | 2 | 6 |
| 5 | 0 | 5 | 0 | 5 | 0 | 5 | 0 | 5 |
| 6 | 2 | 8 | 4 | 0 | 6 | 2 | 8 | 4 |
| 7 | 4 | 1 | 8 | 5 | 2 | 9 | 6 | 3 |
| 8 | 6 | 4 | 2 | 0 | 8 | 6 | 4 | 2 |
| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

Some of the rows possess interesting properties. For example, the rows that begin with 1, 3, 7, or 9 contain every number from 1 to 9 exactly once but never include zeros. Another way to state this is that the aforementioned rows are *permutations* (1 to 1 rearrangements) of each other. We could have predicted that the rows beginning with numbers coprime with $n$ (e.g. $a = 1, 3, 7 \,\text{or}\, 9$) would not contain zeros by applying Corollary 7.21 with $n = 10$ and letting $a$ be coprime with $n$ and letting $b$ run through the numbers greater than 0 and less than 10. This starts us on a very interesting line of thinking followed by Fermat, and later by Euler, when they observed these patterns.

**Definition 7.30.** Congruence

We say that *a is congruent to b relative to the modulus n*, or simply *a is congruent to b mod n* when $n|a - b$. We utilize such a relationship often enough that we use the more compact notation for it:

$$a \equiv b \,(\operatorname{mod} n)$$

when $n|a - b$. We also say that $b$ is a *residue* of $a$ Otherwise we may write

$$a \not\equiv \; b \,(\operatorname{mod} n)\,.$$

Sometimes, when the modulus is apparent from the context we may instead write

$$a \equiv b \,\text{or}\, a \not\equiv \; b.$$

Alternatively, we could have defined it so that two number are congruent if and only if they have the same mathematical remainders upon division by $n$ (see the discussion of the problems with the C++ % operator after Equation 7.2). The second definition is equivalent to Definition 7.30 as demonstrated in the lemma below, and in what follows we will use whichever defnition is most convenient.

*Remark* 7.31. Note that in the statement of Lemma 7.32 we use the symbol "mod" in two different but related ways. When it appears at the end of a congruence in parenthese it tells us the modulus of the congruence. When it appears as an infix binary operator it denotes the mathematical remainder operator (similar to %, but remainders are guaranteed to be non-negative). While at first this may seem confusing, the following lemma shows that the two notions are closely related.

**Lemma 7.32.** *Numbers Are Congruent Exactly When They Have The Same Remainders*
$a \equiv b (\operatorname{mod} n) \iff a \operatorname{mod} n = b \operatorname{mod} n$

*Proof.* We know that there must $\exists q_0, q_1 \in \mathbb{Z}$ such that $a = q_0 n + a \operatorname{mod} n \; b = q_1 n + b \operatorname{mod} n$. Then

$$\begin{aligned} a \equiv b &\iff \exists k : a - b = kn \\ &\iff \exists k : q_0 n + a \operatorname{mod} n - q_1 n - b \operatorname{mod} n = kn \\ &\iff \exists k : a \operatorname{mod} n - b \operatorname{mod} n = (k - q_0 + q_1)\, n \end{aligned}$$

But $|a \operatorname{mod} n|$ and $|b \operatorname{mod} n|$ are smaller than $|n|$ so we must have $k = q_1 - q_0$ and the right hand side is of the last equation must be 0:

$$\iff a \operatorname{mod} n = b \operatorname{mod} n$$

□

As examples of congruences, from the multiplication table above we have $8 * 3 \equiv 6 * 9 (\operatorname{mod} 10)$. Also, $11 \equiv 1 (\operatorname{mod} 10)$ and $11 \equiv -29 (\operatorname{mod} 10)$, but $11 \not\equiv \; 1 (\operatorname{mod} 12)$. It is also useful to observe that congruences can be added and multiplied.

**Theorem 7.33.** *Addition of Congruences*
*For $a, b, c, d, n \in \mathbb{Z}$, if $a \equiv b(\operatorname{mod} n)$ and $c \equiv d(\operatorname{mod} n)$ then $a + c \equiv b + d(\operatorname{mod} n)$.*

*Proof.* $a \equiv b$ means that $n|(a - b)$. $c \equiv d$ means that $n|(c - d)$. But then divides their sum, $a - b + c - d = a + c - (b + d)$ therefor $a + c \equiv b + d(\operatorname{mod} n)$. □

**Theorem 7.34.** *Multiplication of Congruences*
*For $a, b, c, d, n \in \mathbb{Z}$, if $a \equiv b(\operatorname{mod} n)$ and $c \equiv d(\operatorname{mod} n)$ then $ac \equiv bd(\operatorname{mod} n)$.*

*Proof.* Exercise. □

These last two results suggest that congruent numbers have several properties in common. In many respects the set of numbers congruent with respect to a modulus $n$ behaves as a single number. We will see that we can often learn about a number by considering its remainder upon division by $n$ instead.

**Definition 7.35.** Invertible
We say that $a \in \mathbb{Z}$ is a *invertible with respect to the modulus n*, or more compactly, *a is invertible* $\operatorname{mod} n$ if $\exists b \in \mathbb{Z}$ such that

$$ab \equiv 1 \,(\operatorname{mod} n) \tag{7.38}$$

In such we also say that $a$ is the inverse of $b$. Often we will write such a $b$ as $a^{-1}$. Sometimes we will use the term *unit* for instead of using invertible.

**Theorem 7.36.** *Coprime $\iff$ Invertible*
*If $a, n \in \mathbb{Z}$ then a is invertible mod n* $\iff \gcd(a, n) = 1$

*Proof.* If $\gcd(a, n) = 1$ then by Corollary 7.13 we can write

$$1 = ax + ny$$

but

$$1 = ax + ny \equiv ax \,(\operatorname{mod} n)$$

so we have

$$1 \equiv ax \,(\operatorname{mod} n)\,. \tag{7.39}$$

To prove the converse suppose that there is an $x$ for which Equation 7.39 holds. Then by the definition of congruence we know that $ax - 1$ must be a multiple of $n$. That is, there must be a $y$ such that

$$1 - ax = ny$$

But then $ax + ny = 1$, so by Corollary 7.13 $\gcd(a, n) = 1$. □

We can also restate Theorem 7.18 as “the invertible elements mod $n$ are closed under multiplication".

**Corollary 7.37.** *Cancellation Law*
*If $a, b, c, n \in \mathbb{Z}$and* $\gcd(a, n) = 1$ *then*

$$ba \equiv ca \,(\operatorname{mod} n) \tag{7.40}$$

*implies that*

$$b = c \,(\operatorname{mod} n)$$

*In other words if a is invertible mod n then we can “cancel” a from both sides of a congruence with respect to the modulus n.*

*Proof.* By Theorem 7.36 we can find an inverse $x$ for $a$ such that

$$ax \equiv 1 \pmod{n} \tag{7.41}$$

Multiplying both sides of the congruence 7.40 by $x$ (Theorem 7.34) we get

$$bax \equiv cax \pmod{n}$$

which can be simplified using Equation 7.41 to

$$b \equiv c \pmod{n}.$$

□

Some numbers are self-reciprocal, such as 1 and 9 modulo 10. They are also called square roots of 1. This is not an accident. Clearly 1 is always self-reciprocal. But so is $n - 1$ since

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

So

$$(n-1)(n-1) = (n-2)n + 1 \equiv 1 \pmod{n}$$

since both sides have the same remainders upon division by $n$.

In a ring the set of invertible elements is called the group of multiplicative units: e.g. 1 and -1 in $\mathbb{Z}$, in the ring of real polynomials the units are the polynomials of degree zero, i.e. the real numbers. Notice however, that the set of multiplicative inverses is not closed under addition.

**Definition 7.38.** Zero-Divisor

A number $a \in \mathbb{Z}$ is called a *zero-divisor with respect to the modulus n*, or more compactly a *zero-divisor mod n* if $\exists b \in \mathbb{Z}$ with $b \not\equiv 0 \pmod{n}$ such that

$$ab \equiv 0 \pmod{n}.$$

**Theorem 7.39.** *Zero-divisor $\iff$ Not Coprime*

*$\forall\ a, n \in \mathbb{Z}$ a is a zero divisor mod n if and only if a and n are not coprime. That is*

$$\exists b \neq 0 : ab \equiv 0 \pmod{n} \tag{7.42}$$

*if and only if*

$$\gcd(a, n) \neq 1 \tag{7.43}$$

*Proof.* Suppose that for some $b$ we have $ab \equiv 0 \pmod{n}$. Then by the definition of congruence $n|ab$. If $\gcd(a, n) = 1$we can find $x, y$ such that

$$ax + ny = 1$$

so

$$ax \equiv 1 \pmod{n}.$$

Multiplying both sides of this congruence by $b$ (Theorem 7.34) we get

$$axb \equiv 0 \pmod{n}$$

or

$$b = 0 \pmod{n}.$$

We have just shown that if $\gcd(a, n) = 1$then $a$ is not a zero-divisor mod $n$. To show the converse, suppose $\gcd(a, n) \neq 1$which implies that

$$0 < b = \frac{n}{\gcd(a, n)} < n$$

and so

$$ab = \frac{an}{\gcd(a, n)} = \left(\frac{a}{\gcd(a, n)}\right) n$$

But by definition $\gcd(a, n)|a$ , so $a/\gcd(a, n)$ is a whole number, thus $n|ab$, and we see that $a \equiv 0(\bmod\, n)$. □

Notice however, that the set of zero divisors is not closed under addition. As a direct consequence of Theorem 7.36 and Theorem 7.39 we see that

**Corollary 7.40.** *Every integer is either invertible (a unit) or a zero-divisor mod n*

As we saw above, 4 is a zero-divisor mod 10. For some moduli there will be no zero-divisors, for example, when $p$ is prime.

For encryption we will be interested in exponentiation modulo $n$. Fermat was the first person who started looking at powers and remainders together. Such notions are central to his "last" theorem, his "little" theorem (which should be part of the high school mathematics curriculum).

**Theorem 7.41.** *Fermat's Little Theorem (Strong Form)*
*For $a \in \mathbb{Z}$, p prime, a is coprime to p we have*

$$a^{p-1} \equiv 1\,(\bmod\, p)\,. \tag{7.44}$$

*Proof.* (First proof of the Little Fermat Theorem) Consider the set $S = \{a, 2a, ..., (p-1)a\}$. No two elements of $S$ are congruent, for if $ai \equiv aj$ then, since $a$ and $n$ are coprime, we can apply the Cancellation Law to see that $i \equiv j$. Since there are $p - 1$ non-congruent elements in $S$ the product of all the elements of $S$ will be congruent to the product of all of the numbers between 1 and $(p - 1)$ inclusive, modulo $p$ ( Lemma 7.32, Theorem 7.34). That is

$$\prod_{i=1}^{p-1} ai \equiv \prod_{i=1}^{p-1} i\,(\bmod\, p)\,.$$

which we can rewrite as

$$a^{p-1}(p-1)! \equiv (p-1)!\,(\bmod\, p)\,.$$

But all of the numbers from 1 to $(p - 1)$ are relatively prime to $p$, so we can repeatedly apply the cancellation law simplify the last equation further:

$$a^{p-1} \equiv 1\,(\bmod\, p)\,.$$

□

Incidentally, the Last Fermat Theorem was the last marginal note that remained unproved, not the last chronologically. Fermat's son published Bachet's translation of Diophantus. It wasn't a particularly good version except for one thing–it included his father's notes. For the next century or so number theorists would work from this edition of Diophantus with Fermat's notes. Fermat proved his Last Theorem, $a^n + b^n \neq c^n$ for the case $n = 4$. Interestingly, Fermat published the proof for $n = 4$ towards the end of his life. Why would he offer a special case proof if he indeed could prove the general case as he claimed in his famous marginal note. Then 80 years or so passed with no progress until Euler was able to prove Fermat's last theorem for $n = 3$ (harder than the $n = 4$ case).

The first proof of the Little Fermat Theorem that we just gave is an important proof to know: it shows some fundamental group-theoretic properties of the remainders. It also allows us to see multiplication by a remainder as a permutation of the set of the remainders.

It was not, however, the original proof. The proof given by Fermat in his letter to Frenicle for the particular case of $a = 2$, and which was discovered independently by Euler (for any a co-prime with p) about 100 years later, is much simpler. It beautifully illustrates the fundamental importance of the Binomial Theorem. The proof depends on a remarkable fact that for any prime number $p$, and for any $i$ between 2 and $p - 1$

$$\binom{p}{i} \equiv 0 \pmod{p}.$$

Indeed, we know that

$$\binom{n}{i} \equiv \frac{(p-i)\cdot(p-i+1)\cdot\ldots\cdot p}{1\cdot 2\cdot\ldots\cdot i}$$

The numerator is divisible by $p$, the denominator is not. Therefore the result, a whole number, must be divisible by $p$. Armed with this fact we can give the alternate proof.

*Proof.* (Second proof of the Little Fermat Theorem)

$$(a+1)^p \equiv a^p + 1 \pmod{p}$$

because all the elements of the binomial expansion of $(a+1)^p$ except the first ($a^p$) and last (1) are divisible by $p$. We can subtract $a$ from both sides of this congruence and move 1 to the left side. That gives us

$$(a+1)^p - (a+1) \equiv a^p - a \pmod{p}$$

which looks remarkably like an induction step from $a$ to $a+1$. Since we know that $0^p - 0 \equiv 0 (\bmod\, p)$, we obtain that for any $a$: $a^p - a \equiv 0 (\bmod\, p)$. If we decompose this to

$$a\left(a^{p-1} - 1\right) \equiv 0 \pmod{p}$$

we see that one of the factors has to be divisible by $p$. So if $a$ is not divisible by $p$ then $a^{p-1} - 1$ must be divisible by $p$. That is,

$$a \not\equiv 0 \pmod{p} \Longrightarrow a^{p-1} \equiv 1 \pmod{p}.$$

□

That was essentially Euler's proof of the Little Fermat Theorem. Remember that we don't have many proofs from Fermat.

**Corollary 7.42.** *Fermat's Little Theorem (Weak Form)*
*For $a \in \mathbb{Z}$, $p$ prime, $a$ is coprime to $p$ we have*

$$a^p \equiv a \pmod{p}.$$

**Historical Note 7.43.** In those days there were no scientific magazines. The French priest Marin Mersenne (1588–1648), also known for Mersenne Primes, was at the center of the world scientific community. If you wanted to be a scientist you had to know him and write him letters. Galileo and Fermat wrote to him, he was friends with Pascal's father. He was a clearing house and catalyst for scientific research. All the scientists were connected through him. He had extensive correspondence with 78 scientists. He became famous in number theory by conjecturing for $p$ prime when numbers of the form $2^{p-1}$are prime. Fermat was writing to Mersenne . After Mersenne introduced them he started a correspondence with Pascal. Probability theory was founded because of this wonderful exchange of letters between Fermat and Pascal. Science was done in a very personal way.

Notice that the first proof of the little Fermat Theorem depended cancelling $(p-1)!$ without knowing what its reciprocal was. As a matter of fact, we do know that it is equal to $p-1$ and is therefor self-reciprocal. For the primes $p = 2, 3, 5, 7$ we have $(p-1)! = 1, 2, 4, 6 (\bmod p)$respectively. In each case we have $(p-1)! = p-1 (\bmod p)$. This theorem is known as Wilson's theorem, after Wilson who could not prove it–Lagrange did in 1771. Later it was independently proved by Euler. We will prove it now as it will give us a better understanding of the structure of multiplication modulo a prime number. To prove it we need to prove a simple lemma

**Lemma 7.44.** *For p prime*

$$x^2 \equiv 1 \pmod p$$

*has only two solutions.*

*Proof.* Subtracting 1 from each side we have

$$x^2 - 1 \equiv 0 \pmod p$$

Factoring gives

$$(x-1)(x+1) \equiv 0 \pmod p$$

We know that there are no zero divisors mod a prime number so one of these factors must be 0. Therefor $x \equiv 1$ or $x \equiv p-1$. □

Another view is that 1 and $p-1$ are the only numbers that are self-reciprocal (mod $p$). Armed with this lemma the following theorem becomes self-evident

**Theorem 7.45.** *Wilson's Theorem*
*For p prime*

$$(p-1)! \equiv p-1 \pmod p$$

*Proof.* In the lemma and the following comments we have seen that the only self-reciprocal elements of $\{1, 2, ..., p-1\}$ are 1 and $p-1$. Thus when calculating the factorial all of the terms will cancel in pairs (since the reciprocal pairs will have product 1) except for 1 and $p-1$. □

Armed with this lemma, Wilson's theorem is self evident since every factor of $(p-1)!$ and all but the first and last are not self-reciprocal. So each remainder cancels in pairs except for the first and last giving the desired result. It is interesting to consider how easy these results are for us today. These problems were difficult in the $18^{\text{th}}$ century before these structures were well understood. Over centuries things that are hard appear to decline in difficulty.

7.4. **Euler.** For the rest of this section we will focus on results obtained by Leonhard Euler. We will explain Euler's theorem, a generalization of Fermat's Little theorem to composite numbers. His generalization involves a function known as the Euler $\phi$ function (though Euler used $\pi$ – it was Gauss who named it $\phi$), or the Euler *totient* function. Euler's theorem is central to the RSA encryption algorithm. For very large numbers $n$ it is believed to be intractable to calculate $\phi(n)$. But if we happen to know the prime decomposition of $n$ then Euler's theorem gives us a very quick way to calculate $\phi$. In the RSA algorithm, the essential idea is that if $n = p_1 p_2$ where $p_1$ and $p_2$ are large primes, we can safely reveal $n$ as a public key, but $\phi(n)$ can effectively only be calculated by those that know the private key, consisting of $p_1$ and $p_2$.

Euler was the greatest 18th century scientist bar none. There are a few times in history when it is possible to say such a thing. Born in Basel, Switzerland in 1707 and he died in

St. Petersburg, Russia in 1783. His life spanned almost exactly the "real" 18th century. In European history, centuries are not necessarily calendar centuries. They coincide with major historical events. We observe that long centuries follow short centuries. Eric Hobsbawn observes this phenomena in the context of the $19^{\text{th}}$ and $20^{\text{th}}$ For example, the $20^{\text{th}}$ century began in 1914 with the first world war, (some might argue that the century began in 1917). It ended in 1991 with the collapse of the soviet union. The short $20^{\text{th}}$ century was preceded by the long $19^{\text{th}}$ century which began in 1789 with the French revolution. There is a clear demarcation between these "centuries" in that there are dominant mindsets which seem to last for roughly 100 years.

The $18^{\text{th}}$ century began in 1715 with the death of Louis XIV, (Louis le Grand, or the sun king). This ended the great $17^{\text{th}}$ century, with scientists like Pascal, Kepler, Galileo, Newton and Leibniz. The 18th century ended in 1789, as the French mob stormed the Bastille with cries of *Liberté*, *Egalité*, *Fraternité*. Before this there was no idea of "egalité"–it was viewed as a totally unacceptable idea. Even the most progressive minds of the time were supporters of absolute monarchy. For example Voltaire was quite happy to be a friend of Friedrick the Great, the king of Prusssia. We call the $18^{\text{th}}$ century the Age of Enlightenment, or specifically the age of the Encyclopedists. There was a great publishing activity with an aim to summarize all of human knowledge in order to liberate the mind from superstition, i.e. the christian religion. The Encyclopedie, led by people d'Alembert, Diderot and others had the clear mission: unbinding human reason from this "evil" thing called religion. Contrast this with the preceding century, where from Pascal to Leibniz scientists were very religious. In fact scientists were more religious than bishops. Newton was radical Protestant, studying the apocalypse, trying to prove that the pope was the antichrist. Leibniz was a great theological mind. Then came this period of total negation.

Though not a throwback to the previous century, Euler was a committed Christian throughout his life – he did not share the spirit of the age. He was extremely kind and, unlike many scientists treated others with respect, never concerned with his priority. There is a (totally false, but instructive) story about Euler, fabricated by de Morgan in the 1840s, that is nevertheless characteristic of the times. The story concerns Euler's encounter with the Encyclopediste Diderot. Denis Diderot came to St. Petersburg 1773. Political theory at the time is one of enlightened monarchy. Enlightened doesn't imply democrat, it was still an absolute monarchy. It was enlightened in the sense that the monarch was a friend to the philosophe. Voltaire was indeed a personal friend of Friedrick the Great of Prussia, wrote letters to Catherine the Great of Russia. The monarchs wanted all of these Diderots and Voltaires to come and schmooze with them, and so they did. The enlightened minds supported oppressive absolute regimes. The intellectual elite and the power elite were united. In any case, Diderot came to St. Petersburg and according to the story begins attacking the church. Catherine was a little bit upset with that and decided that Diderot should be put down as he was a bit too obnoxious and she didn't want to shake things up. Catherine's attitude, shared by Friedrick the Great and Voltaire was that "If God did not exist we would need to invent him." So she called Euler to help. Euler said, well, since

$$\frac{a+b^n}{n} = x$$

it follows that God exists. The story ends when Diderot, who didn't know how to respond to this (non-sensical) mathematics, departed in shame.

Somewhere around 1727 Euler went to St. Petersburg which was suddenly becoming the intellectual capital of Europe, at least in terms of science. This was in no small part due to Euler's arrival. Peter the Great was probably the first enlightened monarch. He traveled around Europe, he met Leibniz who suggested that he found an Imperial Academy of Science. Peter planned to do it but died. His wife Catherine I started the academy in 1726 and invites Goldbach, Daniel Bernoulli and other great scientists. Somehow Euler who is just a 19 year old kid gets a job. Goes there does everything beautifully, publishing in astronomy, music, cartography, acoustics, physics and mathematics. Eventually he became a European celebrity so Friedrick the Great of Prussia has to have him. In the Age of Enlightenment monarchs actually wanted to get the top scientists. Frequently an ambassador would get assignment to recruit a scientist. For example, in th 1770s a primary mission of the French ambassador to Berlin was to convince Lagrange to move from the Berlin Academy of Science to the French Academy of Science. (Lagrange was the second best, but they could not get Euler). In 1740, Friedrick the Great starts the Prussian and recruits Euler. Russia went back to St. Petersburg in 1765 when Catherine the Great recruited him back. It was good to be a scientist in those days.

Euler was a person of prodigious learning and memory. His favorite poem was Virgil's Aeneid. He knew it by heart, so that in his later years he could recite the first line of any page of the edition which he used as a child. Arguably he was one of the three greatest mathematician's of all time. Learn more about Euler!

Euler proved Fermat's Little Theorem, Theorem 7.41. He wanted very much to see if it generalized it to composite numbers. Then, after experimenting, he came up with the idea that restricting the theorem to the set of invertible elements would give the right result.

Instead of considering all of the numbers $\{0, 1, ..., n-1\}$, following Euler we will consider only those that are invertible with respect to the modulus $n$. For example, with respect to the modulus 10, we eliminate the zero divisors, $\{0, 2, 4, 5, 6, 8\}$ to get the set of invertible elements: $\{1, 3, 7, 9\}$. Of course for prime numbers we only need to eliminate 0. Then, still following the proof of Fermat's Little Theorem we multiply each element in the set by the invertible element $a$ and take remainders upon division by $n$

$$\{a, 3a \bmod n, 7a \bmod n, 9a \bmod n\} \tag{7.45}$$

But the product of two invertible elements is invertible by Theorem 7.18, and invertible elements are closed under taking remainders upon division by $n$ (Exercise: why?). So the set in Equation 7.45 is a permutation of the original set of invertible elements. So the product of the set of invertible elements is equal to the product of the elements in Equation 7.45, which is in turn congruent to the product of the elements of $\{a, 3a, 7a, 9a\}$, and we can carry out the proof below in an identical manner to that of Theorem 7.41

Euler's Theorem is one of the great basic results in Number Theory. We define the $\phi$ function so that we may state Euler's Theorem. (Note that he was the first mathematician to deal with functions in the modern sense, prior to that people dealt with curves and graphs.)

**Definition 7.46.** Euler $\phi$ function

For $n \in \mathbb{Z}$ we define $\phi(n)$ as the size of the set of elements between 1 and $n$ that are coprime to $n$.

**Theorem 7.47.** *Euler's Theorem*

*For $a, n \in \mathbb{Z}$ with a coprime to n we have*

$$a^{\phi(n)} \equiv 1 \pmod{n}.$$

*Proof.* Consider the set $S = \{am_0, am_1, ..., am_{\phi(n)}\}$, where the $m_i$'s run through all of the remainders relatively prime to $n$. No two elements of $S$ are congruent, for if $ai \equiv aj$ then, since $a$ and $n$ are coprime, we can apply the Cancellation Law to see that $i \equiv j$. Since there are $\phi(n)$ non-congruent elements in $S$ the product of all the elements of $S$ will be congruent to the product of all of the $m_i$'s, modulo $n$ ( Lemma 7.32, Theorem 7.34). That is

$$\prod_{i=1}^{\phi(n)} m_i \equiv a^{\phi(n)} \prod_{i=1}^{\phi(n)} m_i \pmod{n}.$$

But all of the $m_i$'s relatively prime to $n$, so we can repeatedly apply the cancellation law simplify the last equation further:

$$a^{\phi(n)} \equiv 1 \pmod{n}.$$

□

Now we have a definition of the Euler $\phi$ function. The brute force way of calculating $\phi(n)$ is to run from 0 and $n$ checking for numbers $x$ with $\gcd(x, n) = 1$. But this quite expensive to compute. Euler found a more efficient way in the case when the prime decomposition of $n$ is known. We will work through the derivation step by step. Before we do the ultimate formula, let's look at some particular special cases.

First we consider the case when $n$ is prime. Then all of the elements between 1 and $n$ that are coprime to $n$ except for $n$, so for $n$ prime $\phi(n) = n - 1$.

Next we consider the case when $n$ is the power of a prime, i.e. $n = p^k$. The numbers that are not coprime to $n$ in this case are the multiples of $p$. There are $p^k/p$, or $p^{k-1}$ of these. So the size of the set of numbers which *are* coprime is $\phi(n) = p^k - p^{k-1}$. We could rewrite this as $\phi(n) = p^k(1 - 1/p) = n(1 - 1/p)$. We could also rewrite the first case in this way. Collecting our cases so far we have:

$$\phi(n) = \begin{cases} n = p & p - 1 = n(1 - 1/p) \\ n = p^k & p^k - p^{k-1} = n(1 - 1/p) \end{cases}$$

Next we consider the case where $n = p_1p_2$ and $p_1$, $p_2$ are primes. As in the previous cases when counting coprimes we would like to subtract the number of multiples of $p_1$(there are $n/p_1 = p_2$ of them) and the number multiples of $p_2$ (there are $p_1$ of them). But if we do this we will have subtracted the multiples of $p_1p_2$ (there is only one) twice, so we must add 1 back in to get the number of non-coprimes $p_1 + p_2 - 1$. So $\phi(n) = p_1p_2 - p_2 - p_1 + 1 = (p_1 - 1)(p_2 - 1)$. We try to rewrite this in the same form as the previous two results, $(p_1 - 1)(p_2 - 1) = p_1(1 - 1/p_1)p_2(1 - 1/p_2) = n(1 - 1/p_1)(1 - 1/p_2)$. These investigations suggest the general formula, where the product is over all primes $p_i$ in the prime decomposition of $n$.

$$n \prod (1 - 1/p_i)$$

Remarkably, this formula is independent of the power to which $p^i$ is raised. We will now prove it. Note that at this point we could do so by induction, but instead we do it the long way, as Euler did. When a simpler method is available we generally avoid combinatorial demonstrations. But here we have an opportunity to show how to rewrite the result of a number of counting considerations in a much simpler form. It is valuable to be able to recognize and take advantage of such opportunities.

**Theorem 7.48.** *Closed Form Formula for $\phi(n)$*

*If the prime decomposition of $n = p_1^{m_1} p_2^{m_2} ... p_q^{m_q}$ then*

$$\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)...\left(1 - \frac{1}{p_q}\right) \tag{7.46}$$

*Proof.* We know that there are $n$ numbers between 1 and $n$. For each number $i$ between 1 and $q$ let $A_i$ be the set of numbers less than $n$ which is divisible by $p_i$. Then $\phi(n)$will be equal to $n - | \cup A_i|$. We can calulate this using the Inclusion-Exclusion Theorem 2.9. Namely,

$$|\bigcup_{i=1}^{n} A_i| = \sum_{\emptyset \neq K \subseteq \{1,2,\ldots,n\}} (-1)^{|K|+1} \mid \cap_{i \in K} A_i|.$$

□

*Proof.* The prime factor $p_i$ has $n/p_i$ distinct multiples between 1 and $n$, so $|A_i| = n/p_i$. So if we add up the multiples over all the $p_i$'s we get

$$\sum_i |A_i| = \sum_i \frac{n}{p_i}.$$

For each $i \neq j$, $|A_i \cap A_j|$ equals the number multiples of $p_i p_j$. Since there are $n/p_i p_j$ of them we get

$$\sum_{i<j} |A_i \cap A_j| = \sum_{i<j} \frac{n}{p_i p_j}$$

Continuing this line of reasoning, we see that

$$|\bigcup_{i=1}^{n} A_i| = \sum_i \frac{n}{p_i} - \sum_{i<j} \frac{n}{p_i p_j} + \sum_{i<j<h} \frac{n}{p_i p_j p_h}.$$

Continuing to compensate for our overcompensation in this manner we arrive at the final count

$$|\bigcup_{i=1}^{n} A_i| = \sum_i \frac{n}{p_i} - \sum_{i<j} \frac{n}{p_i p_j} + \sum_{i<j<k} \frac{n}{p_i p_j p_h} - \ldots + \frac{(-1)^q n}{p_1 p_2 \cdots p_q}$$

There is no summation at the end since there is only possible term when $1 \leq i < j < \ldots < z \leq q$. Factoring out $n$ we have

$$|\bigcup_{i=1}^{n} A_i| = n\left(\sum_i \frac{1}{p_i} - \sum_{i<j} \frac{1}{p_i p_j} + \sum_{i<j<k} \frac{1}{p_i p_j p_h} - \ldots + \frac{(-1)^q}{p_1 p_2 \cdots p_q}\right)$$

so

$$\phi(n) = n - |\bigcup_{i=1}^{n} A_i| = n\left(1 - \left(\sum_i \frac{1}{p_i} - \sum_{i<j} \frac{1}{p_i p_j} + \sum_{i<j<k} \frac{1}{p_i p_j p_h} - \ldots + \frac{(-1)^q}{p_1 p_2 \cdots p_q}\right)\right).$$

But this is exactly what we get when we multiply out the right side of 7.46.

$$n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\ldots\left(1 - \frac{1}{p_q}\right).$$

□

**Exercise 7.49.** Verify the last statement in the proof above.

We have a couple of easy corollaries.

**Corollary 7.50.** *$\phi$ is multiplicative*
*If $m, n \in \mathbb{Z}$ with* $\gcd(m, n) = 1$ *then* $\phi(mn) = \phi(m)\phi(n)$.

*Proof.* Since $m$ and $n$ are coprime their prime decompositions have no primes in common so we can divide the prime factors of $mn$ into those that divide $m$, say $\{p_1, ..., p_a\}$ and those that divide $n$, say $\{q_1, ..., q_b\}$

$$\phi(mn) = mn\left(1 - \frac{1}{p_1}\right)...\left(1 - \frac{1}{p_a}\right)\left(1 - \frac{1}{q_1}\right)...\left(1 - \frac{1}{q_b}\right)$$

which is easily seen to be equal to $\phi(m)\phi(n)$. □

**Corollary 7.51.** *Useless but interesting*
$\phi(n^2) = n\phi(n)$

*Proof.* Exercise. □

7.5. **Primality Testing.** Gauss remarks ([17]p.396-397): "The problem of distinguishing prime numbers from composite numbers and of resolving the latter into prime factors is known to be one of the most important and useful in arithmetic. It has engaged the industry and wisdom of ancient and modern geometers to such an extent that it would be superfluous to discuss the problem at length. Nevertheless, we must confess that all methods that have been proposed thus far are either restricted to special cases or are so laborious and prolix that even for numbers that do not exceed the limits of tables constructed by estimable men, ...they try the patience of even the practiced calculator. And these methods do not apply at all to larger numbers. ...the dignity of the science itself seems to require that every possible means be explored for the solution of a problem so elegant and so celebrated. ...The techniques that were previously known would require intolerable labor even for the most indefatigable calculator."

Gauss has told us why he thinks primality testing is important and difficult. We have seen some conditions that must hold for a number to be prime, for example Fermat's Little Theorem and Wilson's Theorem. It is natural to ask whether the converse of these Theorems might be true, for if so we might be able to use them for primality testing. We first prove the converse of Wilson's Theorem.

**Theorem 7.52.** *If* $(n-1)! \equiv n - 1 (\bmod n)$ *then n is prime.*

*Proof.* Assume that $n$ is composite. Let $p$ be the smallest prime divisor of $n$. Then either $p < \sqrt{n}$or $p = \sqrt{n}$. In the former, case$n/p \neq p$, both $p$ and $n/p$ will appear in our factorial expansion. But since $n|(p \cdot n/p)$ we would have $(n-1)! \equiv 0$, contrary to hypothesis. In the case where $p = \sqrt{n}$ then $p^2 = n$ and either $p > 2$or $p = 2$. If $p > 2$ then $2p < n$, so both $p$and $2p$ appear in $(n-1)!$ so once again we would have $(n-1)! \equiv 0$. In the last sub-case $p = 2$ and $n = 4$, but then $(4-1)! \not\equiv (4-1)(\bmod 4)$. Interestingly, $n = 4$ is the only case where $(n-1)! \not\equiv 0$, but it still not congruent to $n - 1$. We have exhausted all the cases and we see that the congruence will never hold if $n$ is composite. □

Now taking Wilson's Theorem together with its converse above we see that a number $n$ is prime if and only if $(n-1)! \equiv n - 1(\bmod n)$. Unfortunately this is not a practical test for primality since it is too expensive to calculate factorial (exponential in the number of digits).

2000 years before Gauss the ancient Chinese practiced experimental mathematics. For example, they considered numbers of the form

$$\begin{array}{rcccc}
2^2 - 2 & = & 2 & = & 2 \\
2^3 - 2 & = & 6 & = & 2 \cdot 3 \\
2^4 - 2 & = & 14 & = & 2 \cdot 7 \\
2^5 - 2 & = & 30 & = & 2 \cdot 3 \cdot 5 \\
2^6 - 2 & = & 62 & = & 2 \cdot 31 \\
2^7 - 2 & = & 126 & = & 2 \cdot 3 \cdot 3 \cdot 7 \\
2^8 - 2 & = & 254 & = & 2 \cdot 127 \\
2^9 - 2 & = & 510 & = & 2 \cdot 3 \cdot 5 \cdot 17 \\
2^{10} - 2 & = & 1022 & = & 2 \cdot 7 \cdot 73 \\
2^{11} - 2 & = & 2046 & = & 2 \cdot 3 \cdot 11 \cdot 31
\end{array}$$

They noticed that, as in the table above, $2^n - 2$ contained $n$ in its prime decomposition exactly when $n$ is prime. In the Chinese text "The nine chapters of mathematical art" by Ch'in Chiu-Shao the hypothesis was made that a number $n$ is prime if and only if $n|(2^n - 2)$. Sadly enough they did not study it any further (no attempt was made to use numbers other that 2, or to prove the conjecture), and the Europeans were unaware of this work. We can prove it directly in one direction by the weak form Fermat's Little Theorem (Corollary 7.42). Fermat should have realized that the converse to his Little Theorem could be used to catch many composite numbers, but he didn't. We know this because he struggled for a long time to determine whether the sixth Fermat number, $2^{2^5} + 1$ was prime. If he attempted to use the divisibility test he would have known that it was composite. Somehow it never occurred to him that you could invert a theorem and use it to test for primality. These ideas are the theme of this section.

The first person to conjecture that this could be used as a primality test was one of the greatest men of the age, Gottfried Wilhelm von Leibniz (1646-1716). He could prove Fermat's Little Theorem, so he knew that when $n$ is prime the divisibility criterion $n|(2^n - 2)$ must hold, and that the criterion was unsatisfied by all of the composite numbers that he tried. So in 1680 he conjectured that $n|(2^n - 2)$ if and only if $n$ is prime. Amazingly, the mathematical community did not know if he was right or wrong until 1819 when Pierre Frédérique Sarrus (1798-1861) came up with the first counterexample. That is, Sarrus found that the composite number $341 = 11 \cdot 31$ fooled Leibniz's test since $2^{341} - 2$ is divisible by 341. Note that this test is very fast since, as we have seen with the Russian peasant algorithm, raising things to a very large power can be done quickly.

People began to wonder if they could extend this test, for example they tried using numbers besides 2. Then they started looking at how many numbers could fool the test. Statistically it turns out that most composite numbers do not fool the test. For example $\pi(10^{10})$, the number of primes less than $10^{10}$ is approximately $10^{10}/\ln(10^{10})$, or about 450 million. But there are only about 15,000 composites that fool the test, also known as *pseudo-primes.* Other terms that you will hear arebase two pseudo-prime, and base two Fermat pseudo primes. So, the density of pseudo-primes is very small, in fact if we pick a number at random and it passes this test the probability of hitting a base two pseudo-prime is only $1/50000$. For a while people struggled to find out how many base two pseudo-primes there were, then in 1903 E. Malo proved that if you have a base 2 pseudo-prime which is equal to a product of 2 primes, e.g. 341, then the number $2^{n-1}$ will be a base 2 pseudo-prime, so there are an infinite number of pseudo-primes. Then people asked how many base three pseudo-primes there were. It turns out that such pseudo-primes exist, i.e. composite numbers $n$ for which $n|(3^n - 3)$.

Then the question came to be whether there exists a number which is really a bad pseudo-prime that *always* fools the Leibniz/Fermat Test. That is, is there a composite number $n$ such that $\forall a : n|(a^n - a)$? It took people quite a while to figure out if there were any such numbers. At the end of the 19th century A. Korselt showed that such a bad pseudo-prime exists, if and only if (1) it is square-free (not divisible by the square of any prime) and (2) $\forall p : p \, \text{prime}, p|n \implies (p-1)|(n-1)$. It seemed very unlikely that a number could meet the second condition. For about 15 years nobody knew if such numbers could be constructed. It turns out that there are such numbers, though we know that can't be small for we have already seen that such a number must be at least 341. They were discovered around the turn of the 20th century by an American mathematician Robert Carmichael. This was a time when the North American mathematicians were beginning to make significant contributions. Carmichael discovered the first so-called *Carmichael number* $561 = 3 \cdot 11 \cdot 17$, a composite number with the property that $\forall a : 561|(a^{561} - a)$. This started the long process of figuring out how many Carmichael numbers there are. The next two Carmichael numbers are $1105 = 5 \cdot 13 \cdot 17$, and $1729 = 7 \cdot 13 \cdot 19$. It is worth remembering these three numbers, especially the last, which has a wonderful story behind it, the so-called taxicab number.

The story goes back to early 20th century Cambridge, England. At that time there was a great convergence of number theorists at the university. Two great English number theorists Hardy and Littlewood managed to get Ramanujan, one of the great number theorists of the time, to come there too. Ramanujan was one of the most brilliant minds of all time. He was born in 1887 in Tamil Nadu, a state in the southeastern tip of India. At that time it was part of the British empire. The British did not teach colonial populations to become scientists but only minor admistrators. They taught them just enough so that they could enter the lower ranks of colonial administration. Ramanujan was taught in that way, but by chance he found an old and obsolete math book in high school and he read it. Since he hardly knew any mathematics, he started inventing it from scratch. He learned that he could solve quadratic equations. He decided to try to figure out how to solve cubic equations, and quickly succeeded. The same happened for quartic equations. Never having heard of Galois, he spent a year trying to figure out how to solve quintics and failed. He figure out all sorts of amazing things and in 1913 he sent a letter to Hardy. In the introduction to Hardy's book [18] C.P. Snow writes, "The script appeared to consist of theorems, most of them wild or fantastic looking, one or two already well-known, laid out as though they were original. There were no proofs of any kind...Wild theorems. Theorems such as he had never seen before, not imagined....A fraud or genius?...Before midnight they knew, and knew for certain. The writer of these manuscripts was a man of genius. That was as much as they could judge, that night. It was only later that Hardy decided that Ramanujan was, in terms of *natural* mathematical genius, in the class of Gauss or Euler..." In the twentieth century one just does not receive letters from high school graduates with great mathematics. Hardy worked hard to get Ramanujan to come. Ramanujan was a Brahmin so he could not travel over water, in addition he was a vegetarian, so it was not easy for him to come to Cambridge, but in 1914 he did. Ramanujan together with Hardy and Littlewood in about 5 years produced a remarkable string of very great results. He died in 1920. Even now many mathematicians study his notebooks for insights. He didn't prove things, but he would come up with these unbelievably beautiful conjectures, mostly correct. In Hardy's book about Ramanujan [19] he writes, "I remember once going to see him when he was lying ill at Putney. I had ridden in taxi cab number 1729 and remarked that the number seemed to me rather a dull one, and that I hoped it was not an unfavorable omen. 'No,' he replied, 'it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways.'" ($1729 = 1^3 + 12^3 = 9^3 + 10^3$). After Littlewood heard the taxicab story, he said "Every positive integer is one of Ramanujan's personal friends." But natural numbers were even better friends of Euler: after all when Hardy asked Ramanujan if he knew a number that is the sum of two 4th powers in two different ways, Ramanujan did not know of one. Euler did! $133^4 + 134^4 = 158^4 + 59^4 = 635318657$.

Back to the question of how many Carmichael numbers exist. Carmichael postulated that there were infinitely many in 1910. Paul Erdös attacked the problem time and time again. In 1992, Alford, Granville and Pomerance showed that there are more than $n^{2/7}$ Carmichael numbers up to $n$, far fewer then the number $\pi(n) \approx n/\ln n$ of primes up to $n$. Erdös proved in the 1949's that the reciprocals of the Carmichael numbers converge. So there are numbers that can fool our test all of the time. But we can make a trivial improvement that will catch even the Carmichael numbers.It was known to Euler, that $n$ is prime $\Longleftrightarrow \forall a : 0 < a < n, a^{n-1} \equiv 1 (\bmod n)$. So this slight simplification, checking whether $a^{n-1} \equiv 1$ instead of whether $a^n \equiv a$, prevents Carmichael numbers from fooling us for every $a$. For the strong version of Fermat's Little Theorem tells us that the congruence must hold for primes. On the other hand, if the congruence holds then $a^{n-2}a \equiv 1$ which tells us that every non-zero number is invertible (mod $n$). But the factors of a composite number $n$ are zero-divisors (mod $n$) and so cannot be invertible(Theorem 7.36). We call the simplified test the strong Fermat Test, while we call the other test the Weak Fermat Test. Of course it is not a computationally effective test, but it tells us that at least we have some criteria which will let us catch all composites, even the Carmichael numbers. We can now restate the definition of a Carmichael number as a number $n$ such that $\forall a : 0 < a < n, \gcd(a, n) = 1 \Longrightarrow a^{n-1} \equiv 1 (\bmod n)$.

Now we come to this amazing discovery, in the 1970s first by G. Miller, then by M. O. Rabin, who discovered a way to modify the test in such a way that one could say with certainty that the probability of being fooled was less than 0.25. In reality the chances are less, since there are so few Carmichael numbers, but the key fact is that we can guarantee that their test will succeed at least three quarters of the time when testing a single random base $a$, where $0 < a < n$. With this probabilistic algorithm , we can repeat the test for, say, 100 different random bases and we will be guaranteed that probability that we have failed to detect a composite number will be less than $1/4^{100}$. That is, with a sufficient number of tests we can be as certain as we like of our results. In fact, we can arrange that for a relatively small number of tests, the likelihood of a false result can be made small enough so that it is more likely that cosmic radiation would affect a bit in memory than it would be for their test to claim that a composite number is prime. Or consider another way to view probabilistic algorithms, put forward by Knuth. Miller showed that if the generalized Riemann Hypothesis about Dirichlet L-functions holds, (and most mathematician's believe it does), and if we test all the $a$'s up to $2\ln^2 n$ then the test will return the correct results–definitely, not probabilistically. Knuth ([4] v.2, p.396) asks: even if the generalized Riemann Hypothesis is true, which would you rather believe with certainty, a 100 page mathematical proof or the probabilistic primes test?

We will explain the Miller-Rabin test but will not give the proof that the probability of failure is less than 0.25. Interested readers can find it in Knuth, but it is somewhat technical. In practice people use the Miller-Rabin algorithm developed over 1972-1974 to test for primes. In the meantime various other methods of testing for primes were developed, for example the ECPP (elliptic curve primality proving) allows you to deterministically decide whether a number is prime, but there is a small probability that the algorithm could run for a very long time without terminating. The randomness goes not towards the truthfulness of the algorithm, but towards the running time. Then three Indian researchers Manindra Agarwal and his two students Neeraj Kayal, and Nitin Saxena, known collectively as AKS, quite recently demonstrated that there is a polynomial-time deterministic algorithm. It runs quite a bit slower than Miller-Rabin, so the AKS algorithm hasn't had much practical effect. Miller-Rabin uses one more simple mathematical fact, which we encountered in the when proving Wilson's theorem in Lemma 7.44: for a prime $p$, $x^2 \equiv 1 (\bmod p)$ has only two solutions, 1 and $p - 1$.

Now suppose that we want to test whether the number $n$ is prime. Let's assume that $n > 2$ so that it is odd. Then $n - 1$ is must be even, so we can represent it as $2^k q$ where $q$ is odd and $k > 0$. Given $n$ we start the Miller-Rabin test by computing $k$ and $q$. This is fast since it is a simple matter of shifting and testing. Then we select a random *witness* $a$. We call the number a witness because if the test decides that the number $n$ is not prime then $a$ is a witness to that fact. But if the test decides that $n$ is prime it is still possible that $n$ is a false witness. So we can never get false negatives, but we may get false positives. Given a witness $a$, consider the sequence of the remainders of $a^q, a^{2q}, a^{4q} ..., a^{2^k q}$ (mod $n$). We claim:

**Lemma 7.53.** *If $n$ is prime where $n = 2^k q + 1, k > 0$ then the sequence $a^q, a^{2q}, a^{4q} ..., a^{2^k q}$ will either begin with a residue of 1 (mod n) or will contain an element that is a residue of $n - 1 (\bmod n)$*

*Proof.* By Fermat's Little Theorem we know that since $n$ is prime, $a^{n-1} \equiv 1$ and so $a^{2^k q} \equiv 1$. Therefor we know that the sequence ends with a residue of 1. If the sequence does not begin with a residue of 1 then, since it ends with one and since we move from element to element by squaring, there must be an element $x \not\equiv 1$ where $x^2 \equiv 1$. But Lemma 7.44 tells us that such an $x$ must be a residue of 1 or $n - 1$, and since we are assuming that it is not congruent to 1 it must be congruent to $n - 1$. □

**Example 7.54.** Consider what happens when $n = 9 = 2^3 \cdot 1 + 1$. Then $k = 3, q = 1$. We randomly select $a = 2$ for our witness. The our sequence is $a^q = 2^1 \equiv 2 (\bmod 9)$, $a^{2q} = 2^2 \equiv 4 (\bmod 9)$, $a^{4q} = 16 \equiv 7 (\bmod 9)$, $a^{8q} \equiv 7^2 \equiv 4 (\bmod 9)$,... and we see that $n - 1 = 8$ will not occur in the sequence so 2 is a witness that $n$ is no prime.

**Example 7.55.** For $n = 17 = 2^4 + 1, k = 4, q = 1$. Then we randomly select $a = 3$ and our sequence becomes 3, 9, 13, 16 (mod 17). We have hit $n - 1 = 16$, so 3 is not a witness against the primality of 17.

On to implementation the test. We begin with Fermat Test. We create a function object for a given $n$ and run it on as many integer witnesses as we like:

```
//Concepts: I models integer
template <typename I>
class fermat_test :
  public std::unary_function<I, bool>
{
private:
  I n;
public:
  fermat_test(I potential_prime) : n(potential_prime){}
  bool operator()(const I& x) const {
    return power(x,
           n - I(1),
           modulo_multiplies<I>(n)) == I(1);
  }
};
```

**Exercise 7.56.** Using the above implementation of the Fermat Test, see how often it is fooled by the first randomly selected witness from 3 to 1000. The first Carmichael number has the best chance of fooling the test. Notice that $\phi(561) = \phi(3)\phi(11)\phi(17) = 2 \cdot 10 \cdot 16 = 320$ and so a single witness has a $320/559 \approx 0.57$ chance of being fooled (we don't check 0 or 1).

Now we implement the Miller-Rabin test. Note we put $q$ and $k$ in the constructor since we construct the function object once then apply it to multiple witnesses. We settle on 5 times the number of decimal digits as sufficient number of tests as a sufficient degree of probabilistic certainty. In an industrial strength test we would try something easier first, for example we would test for divisibility by each of the first 100 primes (except 2) from a table, since, for example, there is a 1 in 3 chance that $n$ is divisible by 3, etc. Only after this test failed to witness compositeness would we try Miller-Rabin.

```
//Concepts: I models integer
template <typename I>
class miller_rabin_test :
  public std::unary_function<I, bool>
{
private:
  modulo_multiplies<I> times;
  I one;
  I minus_one;
  I q;
  int k;
public:
  miller_rabin_test(I potential_prime) :
    times(potential_prime),
    one(I(1)),
    minus_one(potential_prime - I(1)),
    q(potential_prime - I(1)),
    k(0) {
      while (is_even(q)) {
        ++k;
        halve_non_negative(q);
      }
    }
  bool operator()(I x) {
    x = power(x, q, times);
    if (x == one || x == minus_one) return true;
    int i = k;
    while (--i != 0) {
      x = times(x, x);
        if (x == minus_one) return true;
        if (x == one) return false;
        }
        return false;
    }
};
```

**Exercise 7.57.** Using the above implementation of the Miller-Rabin test, see how often it is fooled by the first randomly selected witness from 3 to 1000.

To summarize, we started with Weak Fermat Test. Then we strengthened it to the strong Fermat Test, but we still had 60% chance of a false witness in the case of 561 for example. The probability of being fooled $\phi(n)/n$ is maximized when $n$ is prime and nearly as high for products of two very large primes. We want to assure that, even for very bad numbers, we can with high probability catch the false witnesses. The trick of checking for $n - 1$ guarantees that the likelihood of finding a false witness is less than 25% (very much smaller in reality). So by repeating the test for a sufficient number of random witnesses we can reduce our uncertainty to a negligible amount. While we may not directly need to know this test for our day to day work, it is one of the crown jewels of 20th century computer science/mathematics. On that basis we all need to know Miller and Rabin's remarkable result.

7.6. **Cryptology.** Cryptography develops ciphers, cryptanalysis breaks ciphers and there is a constant tension between the two. The history of cryptology holds a great deal of intellectual interest. Many great figures dabbled in cryptography, including Julius Caesar, Thomas Jefferson and quite a few others. It is quite common for people to believe that they can invent a "better" cipher, or that they can break an existing cipher. David Kahn's book [20] is an interesting source for the history of cryptology. Vladimir Arnold claims that mathematics is driven by mechanics, hydrodynamics and cryptography.

Encryption begins with some *plaintext,* or un-encrypted text. The plain text is fed to an *encryption algorithm* along with a *key* to produce the encrypted *ciphertext.* Decryption works in the other direction. The first ciphers that we know go back to the Romans. Apparently Caesar was very interested in cryptography, inventing what we now know as *shift substitution ciphers* or *Caesar ciphers.* Suetonius, author of "Of Twelve Caesars", narrates that Caesar used the following cipher in both his military and domestic documents. He would replace every character in his plaintext with the character three places away (mod 26). That is, A would become D, B would become E, ..., Z would become C. Some readers may be familiar with a modern day instance known as, rot-13, which adds 13 to each character (mod 26) and is often used to obscure possibly offensive newsgroup messages. Rot-13 is notorious in the case of Dmitry Sklayrov, a russian cryptanalyst who gave a paper at a conference demonstrating the lack cryptographic security in eBook software. He discovered that a plug-in from one content vendor used a "profound" encryption algorithm: rot-13. By exposing this fact he incurred the wrath of the lawyers and the FBI. He even earned a few months of jail time for his efforts–he was arrested after the conference.

It is not uncommon for people who use bad encryption to try to impose severe consequences on those that decipher their messages. For example at the end of the 16th century, there was a mathematician named François Viète. He was one of the inventors of algebra (in Europe)–he learned to manipulate symbols as if they were numbers. He was also cryptanalyst to Henry III and Henry IV. He was sufficiently skilled to break the Spanish "unbreakable" code. As a result Spain, which was at war with France, suffered major military defeats. The Spanish court was very upset, and as there was no FBI to call on, they called upon the Vatican to pronounce a diabolical compact between Viète and the devil, because only diabolical powers would allow him to break the code. The Spanish court became the laughing stock of Europe.

Substitution codes are fairly easy to break. If we know that our opponent uses a Caesar cipher we need try at most 25 possible keys before the message will be decrypted. So people began to look for stronger substitution codes. Instead of $x \to x + n (\mathrm{mod}\, 26)$ as in the Caesar codes, we can pick $a$ with $\gcd(a, 26) = 1$ and then set $x \to ax + b (\mathrm{mod}\, 26)$. Such an *affine cipher* gives a far greater number of potential keys, since

$$\phi(26) = \phi(2 \cdot 13) = \phi(2)\phi(13) = (2 - 1)(13 - 1) = 12,$$

so there are 12 choices for $a$ and 26 choices for $b$ giving $26 \times 12$ possible keys to choose from when deciphering. This is better than 25 but still not good enough. The second major historical advance is in an Indian classic known as the Kama Sutra. The Kama Sutra contains a great deal of material on sexual techniques. The book mentions cryptography in as the 44$^{\text{th}}$ and 45$^{\text{th}}$ of the arts that should be practiced. After all a message reading "meet me at midnight" might best be sent as ciphertext. In his commentary on the Kama Sutra, Yasodhara describes a scheme, which might be implemented as for English plaintext: randomly partition the alphabet into two disjoint sets of 13 letters. Pair each letter in the first group with a corresponding letter in the second group. By definition there are $\binom{26}{13} \approx$10,000,000 such possibilities ways to do so. It is clear that we cannot such a cipher by trying all of the possible keys.

Instead of using brute force to try all of the keys, it is possible to analyze the relative frequency with which the various letters (and certain combinations of letters) appear in the ciphertext and match these against the known statistical distribution of characters in, say, the English language. Cryptanalysts determined that even though there are $n!$ possible substitution ciphers for an alphabet of size $n$, they are easily breakable given access to a couple of dozen messages in a known language. To combat this sort of analysis cryptographers thought of a way to make all characters in a ciphertext appear with *equal* probability. Even the great Gauss believed that it was possible to patch substitution ciphers to make them unbreakable by using what we now call *homophonic* substitution ciphers. The algorithm requires us to start with an alphabet of extended size. Instead of using single byte characters, we might use two-byte characters, that is we use roughly a 64-thousand character alphabet. Then we assign the appropriate number of characters to represent the each letter, based on the frequency with which that letter appears in the target language. In the English language for example, more characters would have be encodings of $e$ than of any other letter. Encryption involves randomly selecting one of the representations for each plaintext character. In the resulting ciphertext, by design no characters should appear much more frequently than any others. This idea defeats simple statistical analysis. However if the extended alphabet is small enough, given a large ciphertext cryptanalysts can still succeed by also looking for occurrences of common *digraphs* (pairs of letters). Interested readers may want to look at Christoph Günter's paper on variable length homophonic substitution ciphers [21]. His idea is to compress the extended alphabet using a Huffman-like encoding scheme to make it more difficult to tell where the character breaks are since characters end up as different sizes. This raises the point that encrypting a message using two methods makes deciphering much more difficult.

**Exercise 7.58.** Read the Edgar Allan Poe short story "The Gold Bug" [22] which beautifully describes the idea of statistical cryptanalysis.

Of course it is possible to create an absolutely unbreakable encryption scheme using a *one-time pad*. The idea is to generate a huge sequence of random bits (*not* pseudo random) and make a copy for each of the two correspondents. Encryption is simply a matter of XOR-ing together the message with the random sequence. Decryption occurs in the same manner. We could prove that any possible sequence of bits in the ciphertext is equally likely since the pad could be anything at all. This method was used for the hotline between Whitehouse and the Kremlin during the cold war. Once a message is transmitted, the part of the one-time pad that was used must be destroyed. For some unknown reason, Soviet agencies began reusing one time pads for transmission of secret messages in the 1940s. This fact was discovered by their US and British agencies and a project was launched with code name Venona. After intercepting thousands of messages it still took two years for top cryptanalysts to decipher some of the messages. This ultimately led to the revelation of the identities of agents including Klaus Fuchs and Julius Rosenberg. While one-time pads are absolutely secure if used properly, they suffer from the fact that one must distribute the pads and keep them secure.

Before the second world war cryptography became virtually unbreakable given that an attacker only had access to ciphertext. Other attacks are possible if some plaintext is available. A third kind of attack is possible if you can feed arbitrary plaintext to an encryption machine and see the results. The US transmitted an open message from Midway then the Japanese encrypted the message and sent it out. When the Americans intercepted they were able to progress in breaking the code.

So far all of the encryption methods that we have discussed have been *symmetric* in the sense that the encryption key is the same as the decryption key. For 2000 years it was assumed that the encryption mechanism and the key needed to be kept secret. Remarkably, from the middle 1960s to the end of the 1970s there were multiple independent breakthroughs which showed otherwise. Apparently the breakthroughs happened (at least) twice, although some of the breakthroughs were kept secret by the various intelligence agencies until many years later. We will first discuss the freely published work. In 1976 Whitfield Diffie and Martin Hellman at Stanford University published the paper "New directions in cryptography" [23] in which they described the possibility of *public key cryptography*; an encryption scheme in which users would have a pair of keys, a public key $e$ for encrypting, and a private key $d$ for decrypting. When Alice wants to send a message to Bob she uses Bob's encryption key, $e$. Bob will use his private decryption key $d$ to decipher the message. Such a method is also called *asymmetric.* Diffie and Hellman didn't have an algorithm, but they explained how public key cryptography could be achieved *if* a function could be found that would meet the following two requirements. The first requirement is that the function needs to be a *one-way function*, that is a function that is easy to compute with an inverse function that is hard to compute. The second requirement is that the inverse function has to be easy to compute in the case that you had access to a certain additional piece of information. A function meeting these two requirements is known as a *trapdoor one-way function.* Given such a function $f$, Alice sends a message to Bob by applying $f$ to the message, while Bob decrypts the message by applying $f^{-1}$ to the message using the trapdoor information that only he knows. Public key cryptography enjoys the remarkable benefit that encryption key distribution is no longer necessary. Diffie and Hellman went so far as to suggest that it was likely that one-way trapdoor functions could be found in mathematics, but they didn't produce one. Interestingly, their paper was published in 1976, but in 1975 one of their students, Ralph Merkle, submitted a paper about the same idea [24], but it wasn't published until 1978, so Diffie and Hellman got the credit. In the foreword to [25] Hellman writes, "Thank you also for helping Ralph Merkle receive the credit he deserves. Diffie, Rivest, Shamir, Adelman and I had the good luck to get expedited review of our papers, so that they appeared before Merkle's seminal contribution. Your noting his early submission date and referring to what has come to be called 'Diffie-Hellman key exchange' as it should, 'Diffie-Hellman-Merkle key exchange', is greatly appreciated."

In 1977 Ron Rivest, Adi Shamir and Len Adleman discovered a one-way trapdoor function [26], which opened the door for public key cryptography Twenty years after the Diffie-Hellman and the RSA papers were published the British intelligence agency GCHQ (Government Communications Headquarters) disclosed that one of their cryptographers named James Ellis had discovered the same thing [27] as Diffie and Hellman back in 1970. And they further claimed that in 1973 Clifford Cocks, another cryptographer working for their agency, came up with the one-way trapdoor function [28] later patented by RSA. So Rivest Shamir and Adleman got rich, while Cocks and Ellis remain mostly unknown. To make the story even more interesting, in the 1990's the Admiral Bobby Inman, the director of the United States' secretive National Security Agency disclosed that this was discovered by some (unnamed) Americans in the 1960s. We may yet hear from the Russians that they had in fact discovered public key cryptography in the 1940s. Ellis recounts [29] that he found an obscure unnamed technical report from Bell Labs, "Final report on project C43". The report had an idea for encrypted analog phone conversations without having to pass along a secret key. The idea was that if Bob wanted to receive a secure message from Alice he could ask Alice to speak for, say, 10 seconds while he played back a tape of random noise into the phone. When Alice was done Bob could subtract the noise that he generated from a recording of the conversation to recover what Alice actually send. This gave Elllis the idea that cryptography without the need for key distribution might actually be possible.

The RSA algorithm is intuitively based on the fact that modular exponentiation scrambles bits beyond recognition. As an example, consider that the linear congruential pseudorandom number generator which passes many tests for randomness. To carry out the RSA algorithm we must begin with two large primes $p_1$ and $p_2$, for simplicity's sake let's say 100 decimal digits each, and we multiply $p_1 p_2 = n$. We can easily compute $\phi(n) = \phi(p_1)\phi(p_2) = (p_1 - 1)(p_2 - 1)$. Next, since most number are coprime to $\phi(n)$, we can easily find a large number $e$ such that $\gcd(e, \phi(n)) = 1$. By using the extended_inverse() algorithm we can find a number $d$ inverse to $e$ where $ed = k\phi(n) + 1$. We no longer need $\phi(n)$, $p_1$, and $p_2$. We publish $e$ and $n$ as our *public key,* keeping $d$ as our private decryption key. For example, when Alice wants to send Bob a message $M$ she first breaks $M$ up into a sequence of blocks of, say, 512 bits each: $X_0, X_1, ..., X_n$. She encrypts $X_i$ by replacing it with $Y_i = X_i^e \bmod n$. To decipher $Y_i$ Bob can simply raise $Y_i$ to the $d^{\text{th}}$ power to get

$$Y_i^d = (X_i^e)^d = X_i^{ed} = X_i^{k\phi(n)+1} = \left(X_i^{\phi(n)}\right)^k X_i$$

which by Euler's theorem is

$$\equiv 1^k X_i \,(\bmod\, n) = X_i,$$

as long as Alice was careful to arrange that the $X_i$'s are coprime to $n$. So we know that $X_i \equiv Y_i^d$. But by construction $X_i$ is positive and less than $n$ so in fact we can make the stronger claim that $Y_i^d \bmod n = X_i$, and thus we have demonstrated how Bob can decipher the block. Note that $n$ can be any number at all, we don't need to use two large primes. But such a choice enables our trapdoor since armed with the knowledge of how to factor of $n$ calculation of $\phi(n)$ becomes tractable. Other applications, such as ATM machines, use 3 primes, which offers certain advantages.

**Exercise 7.59.** Implement a toy RSA with 32-bit blocks and smallish primes. Use the Russian peasant algorithm and modular multiplication with long longs.

The technical community believes that RSA is very hard to break. It is easy to show that the complexity of figuring out the private key $d$ from the public key is equivalent to that of calculating $\phi(n)$, which in turn is equivalent in difficulty to factoring $n$. Nobody can prove that factoring is hard, but it is widely believed to be true. The fact that the algorithm is publicly available, and that prizes for breaking it remain unclaimed despite many attempts, gives the community confidence in the method. In [30], R. A. DeMillo, R. J. Lipton and A. J. Perlis explain that truth in science is a social process. We cannot work in a vacuum, we must publish our results to be assured of their correctness.

Remarkably, not only do these techniques address the problem of encryption, but the public key infrastructure also supports authentication with digital signatures. In our description we will call Alice's keys $d_a$ and $e_a$ to distinguish them from Bob's, which well call $d_b$ and $e_b$. In the digital signature scenario Alice first uses here own *private* key to *sign* her message. Then if she also wants the message to be secret she encrypts the result with Bob's public key $e_b$. That is, she maps $X_i \rightarrow X_i^{d_a e_b} \bmod n$. Upon receipt, Bob raises the block to the $d_b^{\text{th}}$ power mod $n$ to obtain $X_i^{d_a}$. Then he exponentiates mod $n$ to the power $e_a$ from Alice's *public* key to recover the plaintext block, thereby guaranteeing that the message was in fact from Alice. This demonstrates that the public key infrastructure is invertible: if we encode with our private key others can verify that the message was really from us by exponentiation involving our public key.

Unfortunately, public key methods are several orders of magnitude slower than symmetric encryption, so for large messages it is conventional to use public key cryptography only to exchange keys to facilitate secret transmission via DES, Triple DES, or AES. For signatures, we could sign a checksum instead of a large message. Note that there are various pitfalls to avoid. For example each signed message needs a unique piece of information, such as a timestamp to prevent opponents from copying the message for future verbatim use at an inopportune time. We also need to be aware of possible *man-in-the-middle* attacks, where Mallory positions herself between Alice and Bob. Suppose that Alice believes that she is using Bob's public key when in fact she is using Mallory's. Let's also suppose that Bob believes that he is using Alice's public key when in fact he is using one of Mallory's. Then Bob and Alice will believe that they are communicating securely with each other when in fact they are both talking to Mallory, who can say whatever she likes. In fact a usable pubic key infrastructure also requires a *trusted third party* that can vouch for the validity of people's public keys.

Additional readings for this section include Knuth [4] v.2, p.403-406 and also section 4.5.4, and two papers from M. Williamson: [31] and [32].

## 8. Background

This course was originally developed and taught by Alex Stepanov to the Silicon Graphics compiler group from 1996 to 1999. In 2004 it was taught to Adobe engineers in the USA and India. Some material in the course goes back to courses taught by Alex from 1984 through 1988 at Polytechnic University in Brooklyn, NY, and at General Electric Research in 1986.

## References

[1] B. Stroustrup, *The C++ Standard : Incorporating Technical Corrigendum No. 1*, Wiley, Indianapolis, 2003. http://webstore.ansi.org/ansidocstore/product.asp?sku=INCITS%2FISO%2FIEC+14882%2D2003

[2] E. W. Dijkstra, On a methodology of design, available at http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD317.PDF

[3] G. Chrystal, *Algebra, an Elementary Textbook for the Higher Classes of Secondary Schools and for Colleges: Seventh Edition*, American Mathematical Society, Providence, 1999.

[4] D. E. Knuth, *The Art of Computer Programming (3 Volume Set)*, Addison-Wesley Longman Publishing Co., Inc., Reading, 1998.

[5] A. Edwards, *Pascal's Arithmetical Triangle: The Story of a Mathematical Idea* , John's Hopkins University Press, Baltimore, 2002.

[6] T. L. Heath , *History of Greek Mathematics, vol. I*, Dover Publications, New York, 1981.

[7] V. I. Arnold, On teaching mathematics, available at http://pauli.uni-muenster.de/~munsteg/arnold.html

[8] Y. Matiyasevich , My Collaboration with Julia Robinson, available at http://logic.pdmi.ras.ru/~yumat/Julia/

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of Np-Completeness* , W.H. Freeman & Company, New York, 1979.

[10] G. B. Halsted, *Girolamo Saccheri's Euclides Vindicatus*, Chelsea, New York, 1986.

[11] H. Weber, Leopold Kronecker, *Jahresberichte D.M.* **2** (1893), 5–31.

[12] R. M. Burstall , Proving properties of programs by structural induction, *The Computer Journal* **12** (1969), 41–48. http://www3.oup.co.uk/computer_journal/hdb/Volume_12/Issue_01/120041.sgm.abs.html

[13] Z. Manna and R. Waldinger, *The Deductive Foundations of Computer Programming*, Addison-Wesley Publishing Co., Reading, 1993.

[14] L. Sigler, *Fibonacci's Liber Abaci*, Springer, New York, 2002.

[15] J. Kepler, *The Six-Cornered Snowflake*, Oxford University Press, Oxford, 1966.

[16] G. H. Hardy and E. M. Wright, *An introduction to the theory of numbers*, The Clarendon Press Oxford University Press, 1979.

[17] C. F. Gauss, *Disquisitiones arithmeticae*, Springer-Verlag, 1986.

[18] G. H. Hardy, *A mathematician's apology*, Cambridge University Press, 1992.

[19] G. H. Hardy, *Ramanujan: twelve lectures on subjects suggested by his life and work.*, Chelsea Publishing Company, New York, 1959.

[20] D. Kahn, *The Codebreakers : The Comprehensive History of Secret Communication from Ancient Times to the Internet*, Scribner, New York, 1996.

[21] C. G. Günther, A universal algorithm for homophonic coding, *Advances in cryptology-EUROCRYPT '88 (Davos, 1988)*, Springer, New York, 1988, 405–414.

[22] E. A. Poe, The Gold Bug, *Poems and poetics / Edgar Allan Poe*, Library of America, New York, 2003.

[23] W. Diffie and M. E. Hellman, New directions in cryptography, *IEEE Trans. Information Theory* **IT-22** (1976), 644–654.

[24] R. C. Merkle, Secure communications over insecure channels, *Secure communications and asymmetric cryptosystems*, Westview, Boulder, 1982, 181–196.

[25] S. Y. Yan, *Number theory for computing*, Springer-Verlag, New York, 2002.

[26] R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Comm. ACM* **21** (1978), 120–126.

[27] J. H. Ellis, The possibility of non-secret digital encryption, available at http://www.cesg.gov.uk/site/publications/media/possnse.pdf

[28] C. C. Cocks, A note on 'non-secret encryption', available at http://www.cesg.gov.uk/site/publications/media/notense.pdf

[29] J. H. Ellis, The history of Non-Secret Encryption, available at http://www.cesg.gov.uk/site/publications/media/ellis.pdf

[30] R. DeMillo, R. Lipton and A. Perlis, Social processes and proofs of theorems and programs, *New directions in the philosophy of mathematics*, Birkhäuser Boston, Boston, 1986, 267–285.

[31] M. J. Williamson, Non-Secret encryption using a finite field, available at http://www.cesg.gov.uk/site/publications/media/secenc.pdf

[32] M. J. Williamson, Thoughts on cheaper Non-Secret encryption, available at http://www.cesg.gov.uk/site/publications/media/cheapnse.pdf

A. STEPANOV, ADOBE SYSTEMS, INC.
*E-mail address*: `stepanov@adobe.com`

M. MARCUS, ADOBE SYSTEMS, INC.
*E-mail address*: `mmarcus@adobe.com`