Hash Functions

Some notation: $Z_m = \{0, 1, 2, \ldots, m - 2, m - 1\}$
Define a hash function $h : Z \rightarrow Z_m$ as
$$h(k) = k \mod m$$
That is, $h$ maps all integers into a subset of size $m$ by computing the remainder of $k/m$.

However, the function is clearly not one-to-one. When two elements, $x_1 \neq x_2$ hash to the same value, we call it a collision.

There are many methods to resolve collisions, here are just a few.

- Open Hashing (aka separate chaining) – each hash address is the head of a linked list. When collisions occur, the new key is appended to the end of the list.
- Closed Hashing (aka open addressing) – when collisions occur, we attempt to hash the item into an adjacent hash address. This is known as linear probing.

Hash Functions II

In general, a hash function should have the following properties

- It must be easily computable.
- It should distribute items as evenly as possible among all values addresses. To this end, $m$ is usually chosen to be a prime number. It is also common practice to define a hash function that is dependent on each bit of a key
- It must be an onto function (surjective).

Hashing is so useful that many languages have support for hashing (perl, Lisp, Python).

Hash Functions III

However, the function is clearly not one-to-one. When two elements, $x_1 \neq x_2$ hash to the same value, we call it a collision.

There are many methods to resolve collisions, here are just a few.

- Open Hashing (aka separate chaining) – each hash address is the head of a linked list. When collisions occur, the new key is appended to the end of the list.
- Closed Hashing (aka open addressing) – when collisions occur, we attempt to hash the item into an adjacent hash address. This is known as linear probing.

Pseudorandom Numbers

Many applications, such as randomized algorithms, require that we have access to a random source of information (random numbers).

However, there is not truly random source in existence, only weak random sources: sources that appear random, but for which we do not know the probability distribution of events.

Pseudorandom numbers are numbers that are generated from weak random sources such that their distribution is “random enough”.

Number Theory: Applications

Results from Number Theory have countless applications in mathematics as well as in practical applications including security, memory management, authentication, coding theory, etc. We will only examine (in breadth) a few here.

- Hash Functions
- Pseudorandom Numbers
- Fast Arithmetic Operations
- Cryptography

Number Theory: Applications

Slides by Christopher M. Bourke
Instructor: Berthe Y. Choueiry
Spring 2006
Computer Science & Engineering 235
Introduction to Discrete Mathematics
Sections 2.4–2.6 of Rosen
cse235@cse.unl.edu
Pseudorandom Numbers I

Linear Congruence Method

One method for generating pseudorandom numbers is the linear congruential method.

Choose four integers:
- \( m \), the modulus,
- \( a \), the multiplier,
- \( c \) the increment and
- \( x_0 \) the seed.

Such that the following hold:
- \( 2 \leq a < m \)
- \( 0 \leq c < m \)
- \( 0 \leq x_0 < m \)

Our goal will be to generate a sequence of pseudorandom numbers, \( \{x_n\}_{n=1}^{\infty} \)

with \( 0 \leq x_n \leq m \) by using the congruence

\[ x_{n+1} = (ax_n + c) \mod m \]

For certain choices of \( m, a, c, x_0 \), the sequence \( \{x_n\} \) becomes periodic. That is, after a certain point, the sequence begins to repeat. Low periods lead to poor generators.

Furthermore, some choices are better than others; a generator that creates a sequence \( 0, 5, 0, 5, 0, 5, \ldots \) is obvious bad—its not uniformly distributed.

For these reasons, very large numbers are used in practice.

Linear Congruence Method

Example

Let \( m = 17, a = 5, c = 2, x_0 = 3 \). Then the sequence is as follows.

\[ \begin{align*}
  x_{n+1} &= (5x_n + 2) \mod 17 \\
  x_1 &= (5 \cdot 3 + 2) \mod 17 = 0 \\
  x_2 &= (5 \cdot 0 + 2) \mod 17 = 2 \\
  x_3 &= (5 \cdot 2 + 2) \mod 17 = 12 \\
  x_4 &= (5 \cdot 12 + 2) \mod 17 = 11 \\
  x_5 &= (5 \cdot 11 + 2) \mod 17 = 6 \\
  x_6 &= (5 \cdot 6 + 2) \mod 17 = 15 \\
  x_7 &= (5 \cdot 15 + 2) \mod 17 = 9 \\
  x_8 &= (5 \cdot 9 + 2) \mod 17 = 13 \text{ etc.}
\end{align*} \]

Representation of Integers I

This should be old-hat to you, but we review it to be complete (it is also discussed in great detail in your textbook).

Any integer \( n \) can be uniquely expressed in any base \( b \) by the following expression.

\[ n = a_kb^k + a_{k-1}b^{k-1} + \cdots + a_2b^2 + a_1b + a_0 \]

In the expression, each coefficient \( a_i \) is an integer between 0 and \( b - 1 \) inclusive.

Representation of Integers II

For \( b = 2 \), we have the usual binary representation.

\( b = 8 \), gives us the octal representation.

\( b = 16 \) gives us the hexadecimal representation.

\( b = 10 \) gives us our usual decimal system.

We use the notation

\[ (a_ka_{k-1} \ldots a_2a_1a_0)b \]

For \( b = 10 \), we omit the parentheses and subscript. We also omit leading 0s.

Representation of Integers

Example

\[
\begin{align*}
(B9)_{16} &= 11 \cdot 16^1 + 9 \cdot 16^0 \\
&= 176 + 9 = 185 \\
(271)_{8} &= 2 \cdot 8^2 + 7 \cdot 8^1 + 1 \cdot 8^0 = 128 + 56 + 1 \\
&= 185 \\
(1011\ 1001)_{2} &= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 185 \\
\end{align*}
\]

You can verify the following on your own:

\[
\begin{align*}
134 &= (1000\ 0110)_{2} = (206)_{8} = (86)_{16} \\
44613 &= (1010\ 1110\ 0100\ 0101)_{2} = (127105)_{8} = (AE45)_{16}
\end{align*}
\]
**Base Expansion**

**Algorithm**

There is a simple and obvious algorithm to compute the base $b$ expansion of an integer.

**Base $b$ Expansion**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$q = n$</td>
</tr>
<tr>
<td>2</td>
<td>$k = 0$</td>
</tr>
</tbody>
</table>
| 3    | while $q 
eq 0$ do |
| 4    | $a_k = q \mod b$ |
| 5    | $q = \lfloor \frac{q}{b} \rfloor$ |
| 6    | $k = k + 1$ |
| 7    | end |
| 8    | output $(a_{k-1}a_{k-2} \cdots a_1a_0)$ |

What is its complexity?

---

**Integer Operations I**

You should already know how to add and multiply numbers in binary expansions.

If not, we can go through some examples.

In the textbook, you have 3 algorithms for computing:

1. **Addition of two integers in binary expansion; runs in $O(n)$**.
2. **Product of two integers in binary expansion; runs in $O(n^2)$** (an algorithm that runs in $O(n^{1.585})$ exists).
3. **$\text{div}$ and $\mod$ for**

$$q = a \div d$$

$$r = a \mod d$$

The algorithm runs in $O(q \log a)$ but an algorithm that runs in $O(\log q \log a)$ exists.

---

**Modular Exponentiation I**

One useful arithmetic operation that is greatly simplified is modular exponentiation.

Say we want to compute

$$\alpha^n \mod m$$

where $n$ is a very large integer. We could simply compute

$$\alpha \cdot \alpha \cdots \alpha$$

$n$ times

We make sure to $\mod$ each time we multiply to prevent the product from growing too big. This requires $O(n)$ operations.

We can do better. Intuitively, we can perform a repeated squaring of the base,

$$\alpha, \alpha^2, \alpha^4, \alpha^8, \ldots$$

We still evaluate each term independently however, since we will need it in the next term (though the accumulated value is only multiplied by 1).

---

**Modular Exponentiation II**

requiring $\log n$ operations instead.

Formally, we note that

$$\alpha^n = \alpha^{b_k b_{k-1} b_{k-2} \cdots b_1 b_0}$$

$$= \alpha^{b_k b_{k-1}} \cdot \alpha^{b_{k-2} b_{k-1}} \cdots \cdot \alpha^{b_1 b_0}$$

So we can compute $\alpha^n$ by evaluating each term as

$$\alpha^{b_i 2^i} = \begin{cases} \alpha^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases}$$

We can save computation because we can simply square previous values:

$$\alpha^{2^i} = (\alpha^{2^{i-1}})^2$$

---

**Modular Exponentiation III**

We still evaluate each term independently however, since we will need it in the next term (though the accumulated value is only multiplied by 1).

---

**Modular Exponentiation IV**

**Modular Exponentiation**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\text{Input}$ : Integers $a$, $m$ and $n = (b_k b_{k-1} \cdots b_1 b_0)$ in binary.</td>
</tr>
<tr>
<td>2</td>
<td>$\text{Output}$ : $a^n \mod m$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{term} = a$</td>
</tr>
<tr>
<td>4</td>
<td>if $(b_k = 1)$ then</td>
</tr>
<tr>
<td>5</td>
<td>$\text{product} = a$</td>
</tr>
<tr>
<td>6</td>
<td>end</td>
</tr>
<tr>
<td>7</td>
<td>else</td>
</tr>
<tr>
<td>8</td>
<td>$\text{product} = 1$</td>
</tr>
<tr>
<td>9</td>
<td>end</td>
</tr>
<tr>
<td>10</td>
<td>for $i = 1 \ldots k$ do</td>
</tr>
<tr>
<td>11</td>
<td>$\text{term} = \text{term} \times \text{term} \mod m$</td>
</tr>
<tr>
<td>12</td>
<td>if $(b_i = 1)$ then</td>
</tr>
<tr>
<td>13</td>
<td>$\text{product} = \text{product} \times \text{term} \mod m$</td>
</tr>
<tr>
<td>14</td>
<td>end</td>
</tr>
<tr>
<td>15</td>
<td>end</td>
</tr>
<tr>
<td>16</td>
<td>output $\text{product}$</td>
</tr>
</tbody>
</table>
Euclid’s Algorithm

Example

Compute $12^{26}$ mod 17 using Modular Exponentiation.

```
 1 1 0 1 0 = (26)_2
4 3 2 1 - 1
1 16 13 8 12 term
9 9 8 8 1 product
```

Thus,

$$12^{26} \mod 17 = 9$$

Euclid’s Algorithm I

Intuition

Consider finding the gcd(184, 1768). Dividing the large by the smaller, we get that

$$1768 = 184 \cdot 9 + 112$$

Using algebra, we can reason that any divisor of 184 and 1768 must also be a divisor of the remainder, 112. Thus,

$$\text{gcd}(184, 1768) = \text{gcd}(184, 112)$$

Euclid’s Algorithm II

Intuition

Continuing with our division we eventually get that

$$\text{gcd}(184, 1768) = \text{gcd}(184, 112) = \text{gcd}(184, 24) = \text{gcd}(184, 16) = 8$$

This concept is formally stated in the following Lemma.

Lemma

Let $a = bq + r$, $a, b, q, r \in \mathbb{Z}$, then

$$\text{gcd}(a, b) = \text{gcd}(b, r)$$

Euclid’s Algorithm III

Intuition

The algorithm we present here is actually the Extended Euclidean Algorithm. It keeps track of more information to find integers such that the gcd can be expressed as a linear combination.

Theorem

If $a$ and $b$ are positive integers, then there exist integers $s, t$ such that

$$\text{gcd}(a, b) = sa + tb$$

Algorithm 1: Extended Euclidean Algorithm

```
Input : Two positive integers $a, b$.
Output : $r = \text{gcd}(a, b)$ and $s, t$ such that $sa + tb = \text{gcd}(a, b)$.
1 $r_0 = a, b_0 = b$
2 $s_0 = 0, t_1 = 1$
3 $q = \lfloor \frac{a}{b} \rfloor$
4 $r = q b_0 - q b_0$
5 while $r > 0$ do
6 $t_0 = t_1 - q t_0$
7 $s_0 = s_1 - q s_0$
8 $a_0 = s_0, b_0 = r$
9 $q = \lfloor \frac{a}{b} \rfloor$, $r = a_0 - q b_0$
10 if $r > 0$ then
11 $t_1 = t_0$
12 $s_1 = s_0$
13 $a_0 = a, b_0 = b$
14 $q = \lfloor \frac{a}{b} \rfloor$
15 $r = a_0 - q b_0$
16 end
17 end
18 output $\text{gcd}, s, t$
```
Euclid’s Algorithm

Example

\[
\begin{array}{cccccccc}
\alpha_0 & \beta_0 & t_0 & t & s_0 & s & q & r \\
27 & 58 & 0 & 1 & 1 & 0 & 0 & 27 \\
58 & 27 & 1 & 0 & 0 & 1 & 2 & 4 \\
27 & 4 & 0 & 1 & 1 & -2 & 6 & 3 \\
4 & 3 & 1 & -6 & -2 & 13 & 1 & 1 \\
3 & 1 & -6 & 7 & 13 & -15 & 3 & 0 \\
\end{array}
\]

Therefore,

\[
gcd(27, 58) = 1 = (15)27 + (7)58
\]

Euclid’s Algorithm
Comments
In summary:
- Using the Euclid’s Algorithm, we can compute \( r = \gcd(a, b) \), where \( a, b, r \) are integers.
- Using the Extended Euclid’s Algorithm, we can compute the integers \( r, s, t \) such that \( \gcd(a, b) = r = sa + tb \).

We can use the Extended Euclid’s Algorithm to:
- Compute the inverse of an integer \( a \) modulo \( m \), where \( \gcd(a, m) = 1 \). (The inverse of \( a \) exists and is unique modulo \( m \) when \( \gcd(a, m) = 1 \).)
- Solve an equation of linear congruence \( ax \equiv b (\mod m) \), where \( \gcd(a, m) = 1 \)

Problem: Solve \( ax \equiv b (\mod m) \), where \( \gcd(a, m) = 1 \).
Solution:
- Find \( a^{-1} \) the inverse of \( a \) modulo \( m \).
- Multiply the two terms of \( ax \equiv b (\mod m) \) by \( a^{-1} \).
  \[
  ax \equiv b (\mod m) \Rightarrow \quad a^{-1}ax \equiv a^{-1}b (\mod m) \Rightarrow \quad x \equiv a^{-1}b (\mod m).
  \]

Example: Solve \( 5x \equiv 6 (\mod 9) \).

Chinese Remainder Theorem

We’ve already seen an application of linear congruences (pseudorandom number generators).
However, systems of linear congruences also have many applications (as we will see).

A system of linear congruences is simply a set of equivalences over a single variable.

Example

\[
\begin{align*}
x & \equiv 5 (\mod 2) \\
x & \equiv 1 (\mod 5) \\
x & \equiv 6 (\mod 9)
\end{align*}
\]
For each

II

The solution is the sum

\[ x = \sum_{k=1}^{n} a_k M_k y_k \]

How do we find such a solution?

Example

Give the unique solution to the system

\[ \begin{align*}
x &\equiv 2 \pmod{4} \\
x &\equiv 1 \pmod{5} \\
x &\equiv 6 \pmod{7} \\
x &\equiv 3 \pmod{9}
\end{align*} \]

First, \( m = 4 \cdot 5 \cdot 7 \cdot 9 = 1260 \) and

\[ \begin{align*}
M_1 &= \frac{1260}{4} = 315 \\
M_2 &= \frac{1260}{5} = 252 \\
M_3 &= \frac{1260}{7} = 180 \\
M_4 &= \frac{1260}{9} = 140
\end{align*} \]

Chinese Remainder Theorem

Theorem (Chinese Remainder Theorem)

Let \( m_1, m_2, \ldots, m_n \) be pairwise relatively prime positive integers.

The system

\[ \begin{align*}
x &\equiv a_1 \pmod{m_1} \\
x &\equiv a_2 \pmod{m_2} \\
& \vdots \\
x &\equiv a_n \pmod{m_n}
\end{align*} \]

has a unique solution modulo \( m = m_1 m_2 \cdots m_n \).

How do we find such a solution?

Example

The inverses of each of these is \( y_1 = 3, y_2 = 3, y_3 = 3 \) and \( y_4 = 2 \). Therefore, the unique solution is

\[ x = 2 \cdot 315 \cdot 3 + 1 \cdot 252 \cdot 3 + 6 \cdot 180 \cdot 3 + 3 \cdot 140 \cdot 2 \]

\[ = 6726 \pmod{1260} = 426 \]

Chinese Remainder Theorem II

Example

The inverses of each of these is \( y_1 = 3, y_2 = 3, y_3 = 3 \) and \( y_4 = 2 \). Therefore, the unique solution is

\[ x = a_1 M_1 y_1 + a_2 M_2 y_2 + a_3 M_3 y_3 + a_4 M_4 y_4 \]

\[ = 2 \cdot 315 \cdot 3 + 1 \cdot 252 \cdot 3 + 6 \cdot 180 \cdot 3 + 3 \cdot 140 \cdot 2 \]

\[ = 6726 \pmod{1260} = 426 \]

Chinese Remainder Theorem

Proof/Procedure

This is a good example of a constructive proof; the construction gives us a procedure by which to solve the system. The process is as follows.

1. Compute \( m = m_1 m_2 \cdots m_n \).
2. For each \( k = 1, 2, \ldots, n \) compute

\[ M_k = \frac{m}{m_k} \]

3. For each \( k = 1, 2, \ldots, n \) compute the inverse, \( y_k \), of \( M_k \pmod{m_k} \) (note these are guaranteed to exist by a Theorem in the previous slide set).
4. The solution is the sum

\[ x = \sum_{k=1}^{n} a_k M_k y_k \]

Chinese Remainder Theorem

Wait, what?

To solve the system in the previous example, it was necessary to determine the inverses of \( M_k \) modulo \( m_k \)—how’d we do that?

One way (as in this case) is to try every single element \( a \), \( 2 \leq a \leq m - 1 \) to see if

\[ aM_k \equiv 1 \pmod{m} \]

But there is a more efficient way that we already know how to do—Euclid’s Algorithm!

Computing Inverses

Lemma

Let \( a, b \) be relatively prime. Then the linear combination computed by the Extended Euclidean Algorithm,

\[ \gcd(a, b) = sa + tb \]

gives the inverse of \( a \) modulo \( b \); i.e. \( s = a^{-1} \pmod{b} \).

Note that \( t = b^{-1} \pmod{a} \).

Also note that it may be necessary to take the modulo of the result.
Chinese Remainder Representations

In many applications, it is necessary to perform simple arithmetic operations on very large integers. Such operations become inefficient if we perform them bitwise. Instead, we can use Chinese Remainder Representations to perform arithmetic operations of large integers using smaller integers saving computations. Once operations have been performed, we can uniquely recover the large integer result.

Example

Let \( m_1 = 47, m_2 = 48, m_3 = 49, m_4 = 53 \). Compute 2,459,123 + 789,123 using Chinese Remainder Representations.

By the previous lemma, we can represent any integer up to 5,858,832 by four integers all less than 53.

First,
\[
\begin{align*}
2,459,123 \mod 47 &= 36 \\
2,459,123 \mod 48 &= 35 \\
2,459,123 \mod 49 &= 9 \\
2,459,123 \mod 53 &= 29 \\
\end{align*}
\]

Next,
\[
\begin{align*}
789,123 \mod 47 &= 40 \\
789,123 \mod 48 &= 3 \\
789,123 \mod 49 &= 27 \\
789,123 \mod 53 &= 6 \\
\end{align*}
\]

So we’ve reduced our calculations to computing (coordinate wise) the addition:
\[
(36,35,9,29) + (40,3,27,6) = (76,38,36,35) = (29,38,36,35)
\]

And so we have that
\[
29(124656 \mod 47) + 38(122059 \mod 48) + 36(119568 \mod 49) + 35(110544 \mod 53) = 3,248,246 = 2,459,123 + 789,123
\]
Cryptography is the study of secure communication via encryption.

One of the earliest uses was in ancient Rome and involved what is now known as a Caesar cipher.

This simple encryption system involves a shift of letters in a fixed alphabet. Encryption and decryption is simple modular arithmetic.

**Example**

Let $\Sigma = \{A, B, C, \ldots, Z\}$ so $m = 26$. Choose $k = 7$. Encrypt “HANK” and decrypt “KLHU”.

“HANK” can be encoded (7-0-13-10), so

- $e(7) = (7 + 7) \mod 26 = 14$
- $e(0) = (0 + 7) \mod 26 = 7$
- $e(13) = (13 + 7) \mod 26 = 20$
- $e(10) = (10 + 7) \mod 26 = 17$

so the encrypted word is “OHUR”.

**Example Continued**

“KLHU” is encoded as (10-11-7-20), so

- $e(10) = (10 - 7) \mod 26 = 3$
- $e(11) = (11 - 7) \mod 26 = 4$
- $e(7) = (7 - 7) \mod 26 = 0$
- $e(20) = (20 - 7) \mod 26 = 13$

So the decrypted word is “DEAN”.

**Affine Cipher I**

Clearly, the Caesar cipher is insecure—the key space is only as large as the alphabet.

An alternative (though still not secure) is what is known as an affine cipher. Here the encryption and decryption functions are as follows.

- $e_k(x) = (ax + b) \mod m$
- $d_k(y) = a^{-1}(y - b) \mod m$

Question: How big is the key space?

**Example**

To ensure a bijection, we choose $m = 29$ to be a prime (why?). Let $a = 10, b = 14$. Encrypt the word “PROOF” and decrypt the message “OBGJLK”.

“PROOF” can be encoded as (16-18-15-15-6). The encryption is as follows.

- $e(16) = (10 \cdot 16 + 14) \mod 29 = 0$
- $e(18) = (10 \cdot 18 + 14) \mod 29 = 20$
- $e(15) = (10 \cdot 15 + 14) \mod 29 = 19$
- $e(15) = (10 \cdot 15 + 14) \mod 29 = 19$
- $e(6) = (10 \cdot 6 + 14) \mod 29 = 16$

The encrypted message is “AUPPPG”.

**Caesar Cipher I**

In general, we fix an alphabet, $\Sigma$ and let $m = |\Sigma|$. Second, we fix an secret key, an integer $k$ such that $0 < k < m$. Then the encryption and decryption functions are

- $e_k(x) = (x + k) \mod m$
- $d_k(y) = (y - k) \mod m$

respectively.

Cryptographic functions must be one-to-one (why?). It is left as an exercise to verify that this Caesar cipher satisfies this condition.
Choose 2 (large) primes

Compute

Note that

Keep

One can publish a public key.

Anyone can encrypt messages.

However, decryption is done with a private key.

The system is secure if no one can feasibly derive the private key from the public one.

Essentially, encryption should be computationally easy, while decryption should be computationally hard (without the private key).

Such protocols use what are called “trap-door functions”.

The RSA Cryptosystem

Computing Inverses Revisited

Recall that we can compute inverses using the Extended Euclidean Algorithm.

With RSA we want to find $b = a^{-1}$ modulo $\phi(n)$. Thus, we compute

$$\gcd(a, \phi(n)) = s \cdot a + t \cdot \phi(n)$$

and so $b = s = a^{-1}$ modulo $\phi(n)$.
The RSA Cryptosystem

Example

Let \( p = 13 \), \( q = 17 \), \( a = 47 \).

We have

\[
\begin{align*}
\text{n} & = 13 \cdot 17 = 221, \\
\phi(n) & = 12 \cdot 16 = 192, \\
\text{Using the Euclidean Algorithm, } b & = 47^{-1} = 143 \pmod{\phi(n)} \\
\end{align*}
\]

\[
e(130) = 130^{47} \pmod{221} = 65
\]

\[
d(99) = 99^{143} \pmod{221} = 96
\]

Public-Key Cryptography I

Cracking the System

How can we break an RSA protocol? “Simple”—just factor \( n \).

If we have the two factors \( p \) and \( q \), we can easily compute \( \phi(n) \) and since we already have \( a \), we can also easily compute \( b = a^{-1} \pmod{\phi(n)} \).

Thus, the security of RSA is contingent on the hardness of integer factorization.

Public-Key Cryptography II

Cracking the System

If someone were to come up with a polynomial time algorithm for factorization (or build a feasible quantum computer and use Shor’s Algorithm), breaking RSA may be a trivial matter. Though this is not likely.

In practice, large integers, as big as 1024 bits are used. 2048 bit integers are considered unbreakable by today’s computer; 4096 bit numbers are used by the truly paranoid.

But if you care to try, RSA Labs has a challenge:

http://www.rsasecurity.com/rsalabs/node.asp?id=2091

Public-Key Cryptography

Cracking RSA - Example

Example

Let \( a = 2367 \) and let \( n = 3127 \). Decrypt the message, 1125-2960-0643-0325-1884 (Who is the father of modern computer science?)

Factoring \( n \), we find that \( n = 53 \cdot 59 \) so

\[
\phi(n) = 52 \cdot 58 = 3016
\]

Using the Euclidean algorithm, \( b = a^{-1} = 79 \). Thus, the decryption function is

\[
d(x) = x^{79} \pmod{3127}
\]

Decrypting the message we get that

\[
\begin{align*}
d(1225) & = 1225^{79} \pmod{3127} = 112 \\
d(2960) & = 2960^{79} \pmod{3127} = 114 \\
d(9643) & = 643^{79} \pmod{3127} = 2021 \\
d(325) & = 325^{79} \pmod{3127} = 1809 \\
d(1884) & = 1884^{79} \pmod{3127} = 1407 \\
\end{align*}
\]

Thus, the message is “ALAN TURING”.

Public-Key Cryptography

Cracking RSA - Example

Example

Let \( a = 2367 \) and let \( n = 3127 \). Decrypt the message, 1125-2960-0643-0325-1884 (Who is the father of modern computer science?)

Factoring \( n \), we find that \( n = 53 \cdot 59 \) so

\[
\phi(n) = 52 \cdot 58 = 3016
\]