Ian Piper
CSCI203

Algorithms and Data Structures

Week 7 – Lecture A

Big Numbers

- Big numbers, numbers larger than 2^{64}, are important in many applications in computing:
  - Perfect hashing – needs a prime number > |U| the size of the universe of keys.
  - Cryptography – operates with numbers of 512, 1024 or even more bits.
  - Arbitrary precision arithmetic, e.g. calculating pi to a million decimal digits.
- Efficient calculation using big numbers is worth examining as it is useful in its own right and provides some useful methods that have wider application.
  - Divide and Conquer.

Big Numbers

- We will start by looking at:
  - How to represent large numbers;
  - How to add them;
  - How to multiply them;
  - How to raise them to large powers.

Representation

Large Number Representation

- This is the easiest question to address:
- We simply break the number into an ordered sequence of manageable chunks.
- We do this already…
- …It is called decimal notation.
- E.g. we represent the twentieth power of 2 as 1048576 where each digit is a chunk in a sequence of powers of ten.

Decimal Numbers

- 1048576
- $1\times10^6 + 0\times10^5 + 4\times10^4 + 8\times10^3 + 5\times10^2 + 7\times10^1 + 6\times10^0$
- We can do the same using any numeration base:
- The same number can be written as:
  - 100000000000000000000 in base 2 (digits are 01);
  - 1222021101011 in base 3 (digits are 012);
  - 232023301 in base 5 (digits are 01234);
  - 11625034 in base 7 (digits are 0123456);
  - c974g in base 17 (digits are 0123456789abcdefg);
  - 18p2g in base 30 (digits are 0123456789abcdefghijklmnopqrst).

Big Numbers—Big Bases

- If we wish to represent large numbers we can break then up into chunks, each of which fits a computer word:
  - 32 bits;
  - 64 bits.
- In this way a 1024-bit number can be represented as a sequence of:
  - 32 32-bit integers;
  - 16 64-bit integers.
- We choose the largest integer for which we can easily and accurately calculate sums and products.

Addition

It All Adds Up

- To add two large integers we note:
- Integers x and y can be written in base b as follows:
  - $x = x_k \times b^k + x_{k-1} \times b^{k-1} + \ldots + x_0 \times b^0$
  - $y = y_k \times b^k + y_{k-1} \times b^{k-1} + \ldots + y_0 \times b^0$
- We can the write $x+y$ as:
  - $x+y = (x_k+y_k) \times b^k + (x_{k-1}+y_{k-1}) \times b^{k-1} + \ldots + (x_0+y_0) \times b^0$
- We simply add the corresponding chunks of the two numbers (plus any possible carry) to get the equivalent chunk of the result.

Additional Efficiency

- If we add two k-chunk integers this involves calculating k additions, each of b-bit integers.
- We can add two n-bit integers in k operations.
- $k = \log_b n$.
- This is pretty good.

Multiplication

Being Productive: I

- Multiplication is a bit harder.
- The product $x \times y$ involves calculating products of all of the chunks of each number taken in pairs, each product being multiplied by an appropriate power of the base:
  - $x \times y = (x_k \times b^k + x_{k-1} \times b^{k-1} + \ldots + x_0 \times b^0) \times (y_k \times b^k + y_{k-1} \times b^{k-1} + \ldots + y_0 \times b^0)$
  - $= x_k \times y_k \times b^{2k} +$
    $(x_k \times y_{k-1} + x_{k-1} \times y_k) \times b^{2k-1} +$
    $\ldots +$
    $x_0 \times y_0 \times b^0$
  - Note: each product of two numbers has up to twice as many bits as the original numbers.

An Example

- Let us calculate the product of 12345×56789 using base-100 chunks.
- 12345 consists of 3 chunks: 1 23 45
- 56789 consists of 3 chunks 5 67 89
- We calculate the product as follows:

01	23	45
05	67	89

01	05	00	05					12345	56789
01	67		00	67					
01	89			00	89				
23	05		01	15					
23	67			15	41				
23	89				20	47			
45	05			02	25				
45	67				30	15			
45	89					40	05		
			01	*02*	*01*				
		00	07	01	06	02	05		701060205

Analysis

- To multiply two 3-chunk integers involved:
  - 9 multiplications;
  - A similar number of additions.
- In general multiplying two n-digit integers together involves:
  - $O(n^2)$ multiplications;
  - $O(n^2)$ additions.
- Can we do better?
  - Multiplication is much slower than addition.

Karatsuba Multiplication

Karatsuba Multiplication

- Instead of breaking each number into chunks let us simply split them into two pieces.
  - $x = b^{n/2} \times x_H + x_L$
  - $y = b^{n/2} \times y_H + y_L$
- Then $x \times y =$
  - $b^n \times (x_H \times y_H) + b^{n/2} \times ((x_H \times y_L) + (x_L \times y_H)) + (x_L \times y_L)$
  - This involves 4 multiplications, 2 additions and 2 shifts (assuming b is a power of 2).
  - If we keep dividing into smaller chunks we still end up with $O(n^2)$.
- Let us look at the multiplications in more detail.

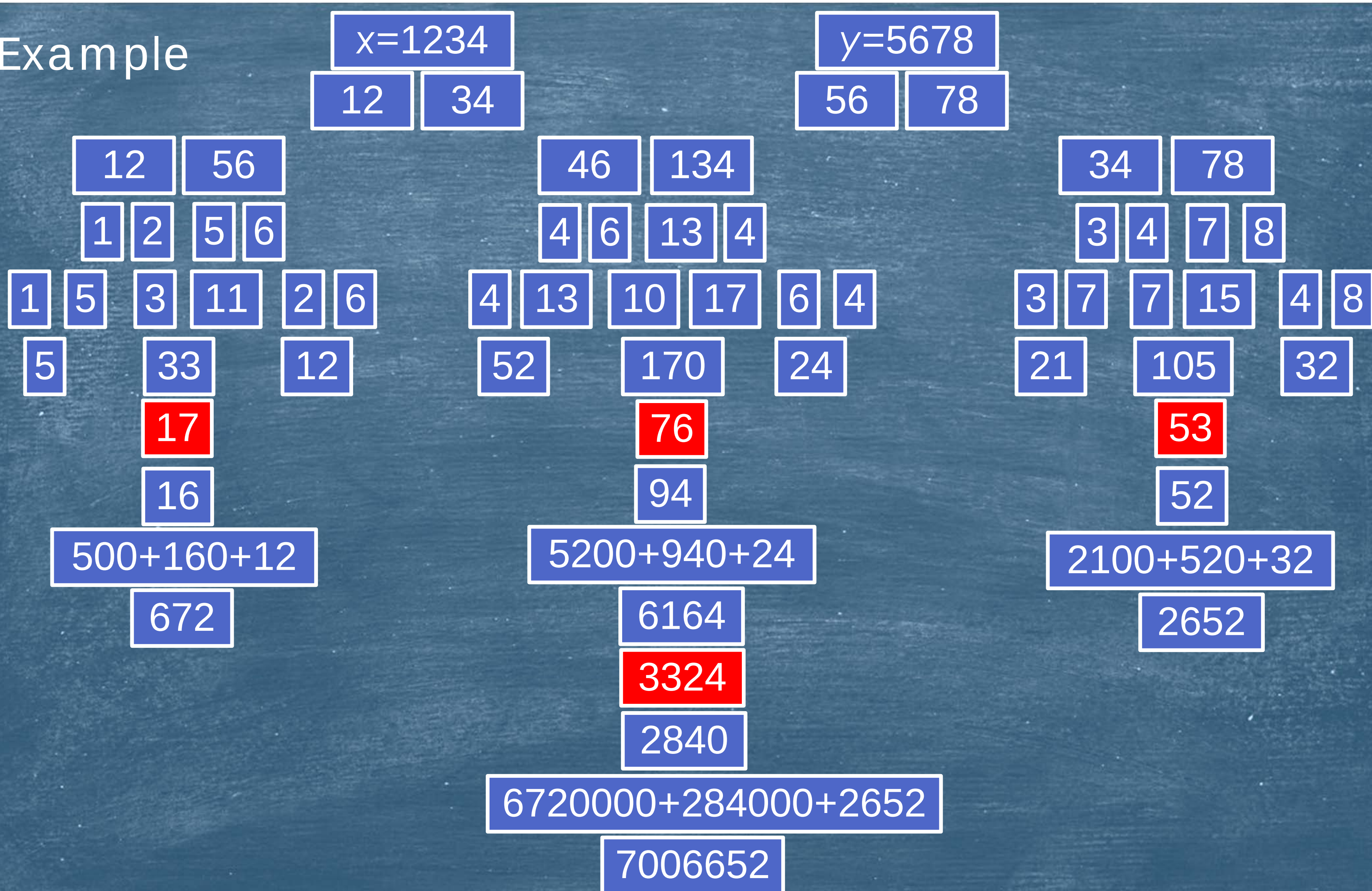
Karatsuba Multiplication

- We calculated four results;
  - $x_H \times y_H$;
  - $x_H \times y_L$;
  - $x_L \times y_H$;
  - $x_L \times y_L$.
- We actually only need three;
  - $x_H \times y_H$;
  - $(x_H + x_L) \times (y_H + y_L)$;
  - $x_L \times y_L$.
- Why is this?

Karatsuba Multiplication

- Consider the three terms we need to calculate the product:
  - $x_H \times y_H$;
  - $x_H \times y_L + x_L \times y_H$;
  - $x_L \times y_L$.
- Our three multiplications allow us to evaluate all of these terms as follows:
  - $x_H \times y_H$;
  - $(x_H + x_L) \times (y_H + y_L) - (x_H \times y_H + x_L \times y_L)$;
  - $x_L \times y_L$.
- Now, if we keep dividing down, our multiplication takes $O(n^{\log 3})$, instead of $O(n^2)$

An Example


x=1234
12
34
y=5678
56
78
12
56
46
134
34
78
1
2
5
6
4
6
13
4
3
4
7
8
1
5
3
11
2
6
4
13
10
17
6
4
3
7
7
15
4
8
5
33
12
52
170
24
21
105
32
17
76
53
16
94
52
500+160+12
5200+940+24
2100+520+32
672
6164
2652
3324
2840
6720000+284000+2652
7006652

Analysis

- Our example required 9 multiplications.
  - $4^{\log 3}=9$
- Brute force would need 16 multiplications
  - $4^2=16$
- If we have longer numbers, the advantage becomes greater:
  - 10 digits: 100 vs. ~38
  - 100 digits: 10000 vs. ~1479

Further Analysis

- Note that, although we perform fewer multiplications we perform many more additions.
- The break-even point will vary, depending on the processor characteristics.
- Generally, when the numbers to be multiplied are more than 320 bits long (10 words) we get an advantage.
- Typically, multiplications of this type are calculated modulo a number which is also of similar size.
- This further reduces the number of operations required.

Powers

Calculating Powers

- To evaluate x^y where x and y are both large integers is a daunting task.
- We recall that:
  - $x^y = x \times x \times x \times \ldots \times x$
  - y-1 multiplications.
- This will take an unacceptable number of operations to complete.
- Can we improve on O(y) multiplications?

Fast Powers

- We can represent x^y recursively as follows:
  - $x^y = (x^{y/2})^2$ if y is even;
  - $x^y = x \times x^{y-1}$ if y is odd.
- Thus, for example, $a^{29} = a \times a^{28}$

$$= a \times (a^{14})^2$$

$$= a \times ((a^7)^2)^2$$

$$= a \times ((a \times a^6)^2)^2$$

$$= a \times ((a \times (a^3)^2)^2)^2$$

$$= a \times ((a \times (a \times (a^2))^2)^2)^2$$

Analysis

- We note that at least half of the operations involved in **`fast_power`** reduce the power by a factor of two.
  - If y is odd at some iteration, it is even next time.
  - If y is even, there is 50% chance that it will be even next time.
  - Even in the worst case, alternating odd and even values, the value of x^y will be computed in O(log y) multiplications.
- This is a big improvement on O(y).
  - For 1000-bit numbers:
    - Conventional power computation would take $O(2^{1000})$ operations;
    - Fast power computation would take $O(1000) = O(2^{10})$ operations;
    - This is a factor of 2^{990} times faster!

Even faster

- We can also create an iterative version:
  - ```
    procedure fast_power_iter(x, y)
        i = y
        result = 1
        a = x
        while i > 0 do
            if i is odd result = big_mult(result, a)
            a = big_mult(a, a)
            i = i % 2
        return result
    ```
- This version removes the cost of the recursive calls.
    ```

Modular Powers

- In practice, we compute all of these results modulo *m*, where *m* is yet another large integer.
- This gives us the modular power procedure:
  - ```
    procedure mod_power(x, y)
        i = y
        result = 1
        a = x
        while i > 0 do
            if i is odd result = mod(big_mult(result, a),m)
            a = mod(big_mult(a, a), m)
            i = i % 2
        return result
    ```
    ```

So What?

- Who would ever want to calculate with such huge numbers?
- Everyone!
  - Even if they don’t know it.
- Many encryption schemes depend on exactly these operations.

Encryption

Meet Alice and Bob (and Carol)

- Alice wishes to send a message *m* to Bob over an insecure transmission channel.
- To prevent eavesdropping, Alice transforms *m* into a cipher text *c* which she then sends to Bob.
- The transformation is achieved by an algorithm that depends on two parameters, the message *m* and a key *k*.
- Bob, who knows the value of *k*, can use it to recover the original message *m from the cipher text* c.
- Unfortunately so can anyone else who knows *k*
  - Like Carol.

Private Key Cryptography

- The fact that the key *k* must be shared between Alice and Bob but kept secret from everyone else is a major problem with this form of ***private key*** cryptography.
- Is there some way for Alice and Bob to communicate securely without first sharing a private key?
- It turns out that the answer is yes.
- The resulting system is known as a ***public key*** system.
- One of the best known public key systems is the RSA system invented by Rivest, Shamir and Adleman.

Fermat's Little Theorem: An Aside

- Consider two numbers p and a where p is prime
- Fermat's little theorem states that a^p mod $p = a$
- For example consider $p = 11$, $a = 10$
  - $10^{11} = 100,000,000,000$
  - $= 9,090,909,090 \times 11 + 10$
  - Therefore 10^{11} mod $11 = 10$
- The proof of Fermat's little theorem involves some elementary number theory.

Proof of Fermat's Little Theorem

- **Theorem:** $a^p = a \bmod p$ if p is prime
- **Proof:** by induction on a
  - For $a = 0$ the result is obvious: $0^p = 0 \bmod p$
  - For $a = 1$ the result is also obvious: $1^p = 1 \bmod p$
  - Assume the result is true for x: $x^p = x \bmod p$

Proof continued

- Assume the result is true for x: $x^p = x \bmod p$
  - Consider $(x + 1)^p$

$$\begin{aligned}(x+1)^p &= (x+1)(x+1)(x+1)\cdots(x+1) \ (p \text{ terms}) \\ &= x^p + px^{p-1} + \cdots {}_pC_i x^i + \cdots + px + 1 \\ &= x^p + 1 + p\times(\text{every other term}) \\ &= x^p + 1 + kp\end{aligned}$$

    - *(this is true only if p is prime)*
  - We can rearrange this to give $(x+1)^p - x^p - 1 = kp$
  - But $x^p = x \bmod p$ so $x^p = x + lp$
  - Combining these equations: $(x+1)^p - x - 1 = kp - lp$ or
- $(x+1)^p = x + 1 \bmod p$

RSA Encryption

RSA Encryption

- Fermat's little theorem was used by Rivest, Shamir, and Adelman to design an encryption algorithm with the following properties:
  - The encryption/decryption algorithm is public knowledge;
  - Anyone can encrypt a message for a given recipient;
  - No one except the recipient, including the sender, can decrypt the message.

RSA in Action

- Let's see how this works on a simple example.
  - Let's assume that the text we want to encrypt is:
    - *WHATEVER*
  - First we have to translate this text into numbers. To do this, we shall use a hash code, which replaces every character by an integer. For the word *WHATEVER* this gives:
    - 23 8 1 20 5 22 5 18
  - To encrypt this text, we will replace every number x by another number y, according to a simple rule.

RSA Encryption

- The key to this encryption rule is given by two numbers n and r. The number n is chosen in a very particular way:
  - n is the product of two primes p and q;
    - say = 29 * 37 = 1073
  - Let's take r = 25 (for reasons we will see in a moment).
- To encrypt m we just compute:
  - $c = x^r \bmod n$.

RSA Encryption: An Example

- So WHATEVER, 23 8 1 20 5 22 5 18, becomes:
  - $23^{25} = 948 \bmod 1073$;
  - $8^{25} = 896 \bmod 1073$;
  - $1^{25} = 1 \bmod 1073$;
  - $20^{25} = 1051 \bmod 1073$;
  - $5^{25} = 796 \bmod 1073$;
  - $22^{25} = 35 \bmod 1073$;
  - $5^{25} = 796 \bmod 1073$;
  - $18^{25} = 764 \bmod 1073$.
- So the encrypted text, c, becomes 948 896 1 1051 796 35 796 764

RSA Decryption

RSA Decryption

- To decrypt c we need a decryption key, which takes the form of a number s. In this particular case, with this choice of n and r, the choice s=121 is the appropriate decryption key.
- The decryption then works via a simple formula, analogous to the encryption: we compute
  - c^s mod n
- and this gives us m back again! (We'll come back to why this happens.)

RSA Decryption: An Example

- The encrypted text is 948 896 1 1051 796 35 796 764:
  - $948^{121} = 23 \bmod 1073$;
  - $896^{121} = 8 \bmod 1073$;
  - $1^{121} = 1 \bmod 1073$;
  - $1051^{121} = 20 \bmod 1073$;
  - $796^{121} = 5 \bmod 1073$;
  - $35^{121} = 22 \bmod 1073$;
  - $796^{121} = 5 \bmod 1073$;
  - $764^{121} = 18 \bmod 1073$.
- And 23 8 1 20 5 22 5 18 is *WHATEVER*.

Why RSA Works

Why RSA Works

- The decryption illustrated on the previous page is possible because r and s have a very special relationship.
- With p = 29, and q = 37, we compute:
  - $z = (p - 1) * (q - 1) = 1008$
- And then we have chosen r and s so that:
  - $r*s = 25 * 121 = 3025 = 1 \bmod z$
- Let's see how this explains why $c^s = m \bmod n$.

Why RSA Works

- We have
  $c^s = (m^r)^s = m^{rs} \bmod n$
- Now rs is 1 + some multiple of m, say L
  $rs = L(p-1)(q-1) + 1$,
- so that:
  $c^s = m^{L(p-1)(q-1)+1} \bmod n$
  $= m^{L(p-1)(q-1)} \times m \bmod n$
  $= m \bmod n$
  - Because $m^{(p-1)(q-1)} = 1 \bmod n$ (proof coming later)

$m^{(p-1)(q-1)} = 1$

- Because p is prime, we know, by Fermat's little theorem, that:
  - $m^p = m \bmod p$;
  - $m^{p-1} = 1 \bmod p$.
- Since all the powers of 1 are 1, it follows that any power of m^{p-1} also equals 1 mod p.
- In particular,
  $m^{(p-1)(q-1)} = 1 \bmod p$.
- or, in other words, $m^{(p-1)(q-1)} - 1$ is a multiple of p.
- Since q is prime, we also have:
  - $m^q = m \bmod q$;
  - $m^{q-1} = 1 \bmod q$.
- so that $m^{(p-1)(q-1)} - 1$ is also a multiple of q.
- Because p and q have no common divisors (they are both prime), $m^{(p-1)(q-1)} - 1$ is therefore divisible by the product $pq = n$, or $m^{(p-1)(q-1)} = 1 \bmod n$.

So What?

- So, why is this any better than a simple substitution cipher?
  - A = 1
  - B = 649
  - C = 855
  - Etc.
- It isn't (at least in the form we have so far presented it).
- So, how is RSA used in the real world?

Remember Alice and Bob (and Carol)?

- Let us consider 3 people Alice, Bob and Carol.
- Alice wants to send a message to Bob without Carol being able to read it.

Bob Does His Bit

- Bob chooses two large (1000-digit) primes p and q and computes their product, n.
- We note that:
  - Bob can easily calculate n from p and q;
  - Nobody else (especially Carol) can easily calculate p and q from n.

Bob Does Some More

- Bob now calculates $z = (p - 1)(q - 1)$
- Next Bob picks an arbitrary integer $1 < r < n$ so that r and m have no common factors.
- Bob can easily calculate s such that $rs = 1 \bmod m$.
  - We will see how in a while.
- Bob now publishes the values of r and n but he keeps the value of s secret.

Alice Does Her Bit

- Alice transforms her message m into a string of bits which is then interpreted as a number a
  - If $0 \leq a \leq n - 1$ we can proceed, otherwise we break m up into a number of chunks so that the equivalent number for each chunk a_i does satisfy the relationship
- Alice now uses **`mod_power`** to calculate $c = a^r \bmod n$ and sends c (or the sequence c_i) to Bob

Back to Bob

- Bob receives c from Alice.
- Using his secret knowledge of s, Bob obtains m by using **`mod_power`** to calculate c^s mod n.
- Now consider the task of an eavesdropper, Carol:
  - She knows n, r and c;
  - She needs to determine m;
  - She needs to determine s, the r^{th} root of c mod n.
- No efficient algorithm is known for this task.
  - The best approach known is the obvious one : factorize n into p and q, compute z as $(p-1)(q-1)$ and compute s from r and z.

So; What is Carol's Problem

- Every step in this attack is feasible except the first, factorizing a 2000-digit number
- Bob's advantage is that he knows the factors – not because he has greater factorizing skills but because he calculated the factors *p* and *q* first and created the 2000-digit product.
- At present there is no mathematical proof of the safety of the RSA system, but, so far, it has proven essentially unbreakable.

The Remaining Details

Unresolved Issues

- This still leaves us with a few questions:
  - How do you find 1000-digit prime numbers, p and q, efficiently?
  - Given $z = (p - 1)(q - 1)$, how do you find a value r such that r and z are mutually prime?
  - Given z and r how do you find a value s such that $rs = 1 \bmod z$?
- It turns out that each of these questions is reasonably easy to answer.

Big Primes

- We have already noted that factorizing a 2000-digit number is impractically hard.
- So is factorizing a 1000-digit number.
- Testing a 1000-digit number for primacy by trial factorization is not practical.

Testing for Primacy

- Fermat's little theorem comes to the rescue once again.
- If $a^p = a \bmod p$ for several values of a, the probability that p is prime is high.
- Note that this does not ensure that p is prime.
- Once again, we can use **`mod_power`** to do the calculations.

Picking Primes

- In practice:
  - Pick a random 1000-digit odd number;
  - Use a few applications of Fermat's little theorem to see if it is a probable prime number;
  - If so, continue;
  - If not, pick another random number and try again.
- The probability of picking a prime number by this method is $1/\log(p)$.
- This means that if p is 1000-digits long the chance that a randomly chosen number is prime is $\approx 1/1000$.
- That is not too bad.

Finding *r*

- Given $z = (p - 1)(q - 1)$, how do you find a value r such that r and z are mutually prime?
  - This at first seems to be another really hard problem.
  - It turns out however that a practical solution is really easy.
  - We just pick a random r and go on to look for s.
  - If we can't find a suitable s value, then r was not mutually prime to z.
  - So we pick another random r and try again.

Finding r (and s)

- Given z and r how do you find a value s such that $rs = 1 \bmod z$?
  - If r and z are mutually prime then $\mathrm{GCD}(r, z) = 1$.
  - Using Euclid's algorithm we can find unique values s and t such that $rs + tz = 1$.
  - The value of s we obtain in this way is the one we are looking for.

Euclid’s Algorithm

Euclid's Algorithm?

- Euclid's theorem states that if the greatest common divisor (GCD) of two integers a and b is equal to some value g then two additional integers x and y can be found so that $ax + by = g$.
- Euclid's algorithm provides us with a mechanism to find;
  - The GCD, g;
  - The values of x and y.

Euclid's Algorithm

- Let us start with two integers a and b and let us assume that $a > b$.
- We can express a as a multiple of b plus some remainder:
  - $a = x_0 b + r_1.$
- We can express b as a multiple of r_1 in the same way:
  - $b = x_1 r_1 + r_2.$
- Repeat this process until we get a remainder of zero:
  - $r_{k-1} = x_k r_k.$
- The value r_k is the GCD.
- We can now back-substitute to get the GCD in terms of a and b.

An Example

- Let a = 131 and b = 71,
- Then:

$$\begin{aligned} 131 &= 1 \times 71 + 60 \\ 71 &= 1 \times 60 + 11 \\ 60 &= 5 \times 11 + 5 \\ 11 &= 2 \times 5 + 1 \\ 5 &= 5 \times 1 + 0 \end{aligned}$$

$$\begin{aligned} 1 &= 11 - 2 \times 5 \\ &= 11 - 2 \times (60 - 5 \times 11) \\ &= 11 \times 11 - 2 \times 60 \\ &= 11 \times (71 - 60) - 2 \times 60 \\ &= 11 \times 71 - 13 \times 60 \\ &= 11 \times 71 - 13 \times (131 - 71) \\ &= 24 \times 71 - 13 \times 131 \end{aligned}$$

- So GCD(a, b) = 1
- So x = –13 and y = 24

Identity Theft

Digital Impersonation

- We have seen how Alice can use Bob's public key to encrypt a message that only Bob can decrypt.
- But so can anyone else.
- What is to stop Carol from sending an encrypted message to Bob and pretending to be Alice?
- Nothing.
- This is a problem.
- In the non-digital world we have a useful mechanism to prevent such impersonation.
- The signature.

Digital Signature

- If Alice encrypts a message using her ***private*** key, Bob can use her corresponding ***public*** key to decrypt and read the message.
  - RSA is symmetric, $(m^r)^s=(m^s)^r=m$.
- Bob is sure that the message came from Alice because the sender knows Alice's private key.
- Carol can also decrypt the message!
- What is the solution to this problem?
- We need a scheme which ensures both identity and security.

Signed and Sealed

- To achieve both proof of identity and security from eavesdroppers we do the following.
- Alice:
  - Alice first encrypts her message, *m*, using her private key, s_a;
  - She then encrypts the result using Bob's public key, r_b;
  - She sends the resulting message to Bob.
- Bob:
  - Bob receives the double-encrypted message from Alice;
  - He first decrypts it using his private key, s_b;
  - He then decrypts the result using Carol's public key, r_a;
  - The result of this is *m*, the original message.