Prepared by: [Arun Konagurthu]

FIT3155 S2/2020: Advanced Algorithms and Data Structures

Week 8: (Semi-)numerical algorithms

Faculty of Information Technology, Monash University

What is covered in this lecture?

This week's lecture deals with (semi-)numerical algorithms

- Algorithms involving (large) integers
  - Integer multiplication
  - Modular exponentiation
  - Miller-Rabin Primality Testing
    - This also introduces a new problem solving algorithmic category, namely **`Randomized Algorithm`**

Source material

- Cormen et al. Introduction to Algorithms (chapter 31)
- Knuth. The art of computer programming (Vol 2, chapter 4.3.3)
- Mathematicians discover the perfect way to multiply [click]

Introduction

- Number-theoretic algorithms are used widely today.
- Most prominent use is in cryptographic schemes.
- These schemes rely on **`large` `prime numbers`**.
- The **feasibility** of cryptographic schemes rely on the ability to generate large prime numbers.
- The **security** of cryptographic schemes rely on the **in**ability to efficiently factorize large numbers into their prime factors.

Computational complexity as a function of a size of input

- For the algorithms here, we need to change the way we think about the size of input.

Computational complexity as a function of a size of input

- For the algorithms here, we need to change the way we think about the size of input.
- For instance, in sorting algorithms, we thought of the size of input in terms of the number of elements being sorted.

Computational complexity as a function of a size of input

- For the algorithms here, we need to change the way we think about the size of input.
- For instance, in sorting algorithms, we thought of the size of input in terms of the number of elements being sorted.
- Here, we will measure the input in terms of **`number of bits`** required to represent the input.

Computational complexity as a function of a size of input

- For the algorithms here, we need to change the way we think about the size of input.
- For instance, in sorting algorithms, we thought of the size of input in terms of the number of elements being sorted.
- Here, we will measure the input in terms of **`number of bits`** required to represent the input.
- For an integer n, the **`number of bits`** it takes to represent is $d = \lfloor \log n \rfloor + 1$ bits.

Computational complexity as a function of a size of input

- For the algorithms here, we need to change the way we think about the size of input.
- For instance, in sorting algorithms, we thought of the size of input in terms of the number of elements being sorted.
- Here, we will measure the input in terms of **`number of bits`** required to represent the input.
- For an integer n, the **`number of bits`** it takes to represent is $d = \lfloor \log n \rfloor + 1$ bits.
- So, a number-theoretic algorithm runs...:

Computational complexity as a function of a size of input

- For the algorithms here, we need to change the way we think about the size of input.
- For instance, in sorting algorithms, we thought of the size of input in terms of the number of elements being sorted.
- Here, we will measure the input in terms of **number of bits** required to represent the input.
- For an integer n, the **number of bits** it takes to represent is $d = \lfloor \log n \rfloor + 1$ bits.
- So, a number-theoretic algorithm runs...:
  - ...in **linear** time, if it involves $O(d)$ computations.

Computational complexity as a function of a size of input

- For the algorithms here, we need to change the way we think about the size of input.
- For instance, in sorting algorithms, we thought of the size of input in terms of the number of elements being sorted.
- Here, we will measure the input in terms of **`number of bits`** required to represent the input.
- For an integer n, the **`number of bits`** it takes to represent is $d = \lfloor \log n \rfloor + 1$ bits.
- So, a number-theoretic algorithm runs...:
  - ...in **`linear`** time, if it involves $O(d)$ computations.
  - ...in **`quadratic`** time if it involves $O(d^2)$ computations.

Computational complexity as a function of a size of input

- For the algorithms here, we need to change the way we think about the size of input.
- For instance, in sorting algorithms, we thought of the size of input in terms of the number of elements being sorted.
- Here, we will measure the input in terms of **number of bits** required to represent the input.
- For an integer n, the **number of bits** it takes to represent is $d = \lfloor \log n \rfloor + 1$ bits.
- So, a number-theoretic algorithm runs...:
  - ...in **linear** time, if it involves $O(d)$ computations.
  - ...in **quadratic** time if it involves $O(d^2)$ computations.
  - etc.

Computational complexity as a function of a size of input

- For the algorithms here, we need to change the way we think about the size of input.
- For instance, in sorting algorithms, we thought of the size of input in terms of the number of elements being sorted.
- Here, we will measure the input in terms of **`number of bits`** required to represent the input.
- For an integer n, the **`number of bits`** it takes to represent is $d = \lfloor \log n \rfloor + 1$ bits.
- So, a number-theoretic algorithm runs...:
  - ...in **`linear`** time, if it involves $O(d)$ computations.
  - ...in **`quadratic`** time if it involves $O(d^2)$ computations.
  - etc.
  - ...in **`logarithmic`** time if it involves $O(\log d)$ computations.

Integer multiplication

Long multiplication

The usual (manual) approach to multiply **large** numbers:

```
    123
  * 456
-------
    738      // = 123 *   6
+  6150      // = 123 *  50
+ 49200      // = 123 * 400
-------
  56088
```

In general, to multiply two d digit numbers, the above long multiplication method can be converted into an algorithm that requires $O(d^2)$ single-digit multiplications (steps).

Can we do faster integer multiplication?

Brief history

- For millennia (until 1960s) we couldn't do better than in $O(d^2)$ steps.

Can we do faster integer multiplication?

Brief history

- For millennia (until 1960s) we couldn't do better than in $O(d^2)$ steps.
- In 1963 Anatoly Karatsuba gave a faster algorithm that requires $O(d^{\log_2 3})$ steps – this lecture will cover this algorithm.

Can we do faster integer multiplication?

Brief history

- For millennia (until 1960s) we couldn't do better than in $O(d^2)$ steps.
- In 1963 Anatoly Karatsuba gave a faster algorithm that requires $O(d^{\log_2 3})$ steps – this lecture will cover this algorithm.
- In 1971 Arnold Schönhage and Volker Strassen gave an even faster algorithm that takes $O(d \times \log d \times \log \log d)$ steps.

Can we do faster integer multiplication?

Brief history

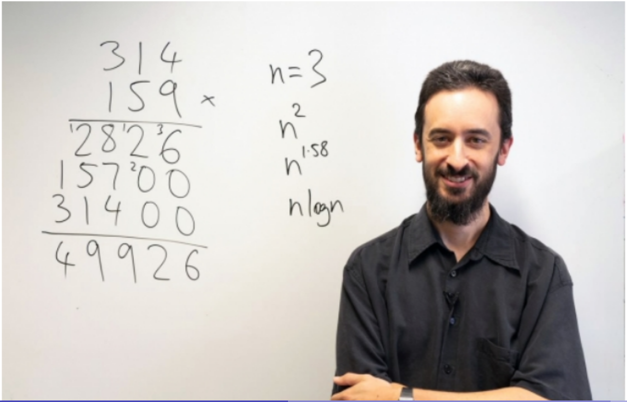
- For millennia (until 1960s) we couldn't do better than in $O(d^2)$ steps.
- In 1963 Anatoly Karatsuba gave a faster algorithm that requires $O(d^{\log_2 3})$ steps – this lecture will cover this algorithm.
- In 1971 Arnold Schönhage and Volker Strassen gave an even faster algorithm that takes $O(d \times \log d \times \log\log d)$ steps.
- **Quite recently**, in March 2019, David Harvey (Australian) and Joris Van Der Hoeven proposed an algorithm (the best possible one) for integer multiplication, requiring $O(d \times \log d)$ steps!

Can we do faster integer multiplication?

Maths whiz solves 48-year-old multiplication problem

04 APR 2019 | LACHLAN GILBERT

A UNSW Sydney mathematician has cracked a maths problem that has stood for almost half a century which will enable computers to multiply huge numbers together much more quickly.



Fast multiplication algorithm on (large) integers

$O(d^{\log_2 3})$-time integer multiplication

Divide and conquer approach

Assume we have two $2d$-bit numbers:

$u = (\underbrace{u_{2d-1}u_{2d-2}\cdots u_d}_{U_1}\underbrace{u_{d-1}u_{d-2}\cdots u_0}_{U_0})_{\text{base-2}}$. That is, $u = 2^d U_1 + U_0$

$v = (\underbrace{v_{2d-1}v_{2d-2}\cdots v_d}_{V_1}\underbrace{v_{d-1}v_{d-2}\cdots v_0}_{V_0})_{\text{base-2}}$. That is, $v = 2^d V_1 + V_0$

$O(d^{\log_2 3})$-time integer multiplication

Divide and conquer approach

Assume we have two $2d$-bit numbers:

$u = (\underbrace{u_{2d-1}u_{2d-2}\cdots u_d}_{U_1}\underbrace{u_{d-1}u_{d-2}\cdots u_0}_{U_0})_{\text{base-2}}$. That is, $u = 2^d U_1 + U_0$

$v = (\underbrace{v_{2d-1}v_{2d-2}\cdots v_d}_{V_1}\underbrace{v_{d-1}v_{d-2}\cdots v_0}_{V_0})_{\text{base-2}}$. That is, $v = 2^d V_1 + V_0$

Therefore, the product $uv = (2^d U_1 + U_0)(2^d V_1 + V_0)$ can be expanded as:

$O(d^{\log_2 3})$-time integer multiplication

Divide and conquer approach

Assume we have two $2d$-bit numbers:

$u = (\underbrace{u_{2d-1}u_{2d-2}\cdots u_d}_{U_1}\underbrace{u_{d-1}u_{d-2}\cdots u_0}_{U_0})_{\text{base-2}}$. That is, $u = 2^d U_1 + U_0$

$v = (\underbrace{v_{2d-1}v_{2d-2}\cdots v_d}_{V_1}\underbrace{v_{d-1}v_{d-2}\cdots v_0}_{V_0})_{\text{base-2}}$. That is, $v = 2^d V_1 + V_0$

Therefore, the product $uv = (2^d U_1 + U_0)(2^d V_1 + V_0)$ can be expanded as:

$$uv = 2^{2d}U_1V_1 + 2^d(U_0V_1 + U_1V_0) + U_0V_0$$

$O(d^{\log_2 3})$-time integer multiplication

Divide and conquer approach

Assume we have two $2d$-bit numbers:

$u = (\underbrace{u_{2d-1}u_{2d-2}\cdots u_d}_{U_1}\underbrace{u_{d-1}u_{d-2}\cdots u_0}_{U_0})_{\text{base-2}}$. That is, $u = 2^d U_1 + U_0$

$v = (\underbrace{v_{2d-1}v_{2d-2}\cdots v_d}_{V_1}\underbrace{v_{d-1}v_{d-2}\cdots v_0}_{V_0})_{\text{base-2}}$. That is, $v = 2^d V_1 + V_0$

Therefore, the product $uv = (2^d U_1 + U_0)(2^d V_1 + V_0)$ can be expanded as:

$$\begin{aligned} uv &= 2^{2d}U_1V_1 + 2^d(U_0V_1 + U_1V_0) + U_0V_0 \\ &= 2^{2d}U_1V_1 + 2^d(U_1V_1 - (U_1 - U_0)(V_1 - V_0) + U_0V_0) + U_0V_0 \end{aligned}$$

$O(d^{\log_2 3})$-time integer multiplication

Divide and conquer approach

Assume we have two $2d$-bit numbers:

$u = (\underbrace{u_{2d-1}u_{2d-2}\cdots u_d}_{U_1}\,\underbrace{u_{d-1}u_{d-2}\cdots u_0}_{U_0})_{\text{base-2}}$. That is, $u = 2^d U_1 + U_0$

$v = (\underbrace{v_{2d-1}v_{2d-2}\cdots v_d}_{V_1}\,\underbrace{v_{d-1}v_{d-2}\cdots v_0}_{V_0})_{\text{base-2}}$. That is, $v = 2^d V_1 + V_0$

Therefore, the product $uv = (2^d U_1 + U_0)(2^d V_1 + V_0)$ can be expanded as:

$$
\begin{aligned}
uv &= 2^{2d}U_1V_1 + 2^d(U_0V_1 + U_1V_0) + U_0V_0 \\
&= 2^{2d}U_1V_1 + 2^d(U_1V_1 - (U_1 - U_0)(V_1 - V_0) + U_0V_0) + U_0V_0 \\
&= (2^{2d} + 2^d)U_1V_1 - 2^d(U_1 - U_0)(V_1 - V_0) + (2^d + 1)U_0V_0
\end{aligned}
$$

$O(d^{\log_2 3})$-time integer multiplication – continued

Divide and conquer approach

$$uv \quad = \quad (2^{2d} + 2^d)U_1V_1 - 2^d(U_1 - U_0)(V_1 - V_0) + (2^d + 1)U_0V_0$$

The form above decomposes the multiplication uv (problem of multiplying two $2d$-bit numbers) into:

- three d-bit multiplications:
  1. U_1V_1
  2. $(U_1 - U_0)(V_1 - V_0)$
  3. U_0V_0
- plus some simple left shifting*
- and some additions/subtractions.

*multiplication by any 2^k requires shifting the bits leftwards by k positions –reason!

$O(d^{\log_2 3})$-time integer multiplication – continued

Divide and conquer approach

$$uv \quad = \quad (2^{2d} + 2^d)U_1V_1 - 2^d(U_1 - U_0)(V_1 - V_0) + (2^d + 1)U_0V_0$$

The form above decomposes the multiplication uv (problem of multiplying two $2d$-bit numbers) into:

- three d-bit multiplications:
  1. U_1V_1
  2. $(U_1 - U_0)(V_1 - V_0)$
  3. U_0V_0
- plus some simple left shifting*
- and some additions/subtractions.

Time complexity

The time complexity of this divide-and-conquer approach can be expressed using the recurrence relationship: $T(2d) = 3T(d) + cd$, where c is some constant. Solving this recurrence yields the $O(d^{\log_2(3)})$ -time algorithm. Try this during self-study. Will be handled as a tute question next week.

*multiplication by any 2^k requires shifting the bits leftwards by k positions –reason!

Divisibility and Divisors

The notion of one integer being **divisible** by another is key to theory of numbers.

Division theorem

For any integer a and any positive integer n, there exists unique integers q (quotient) and r (remainder) such that $a = qn + r$, where $0 \leq r < n$.

Quotient

The value $q = \lfloor \frac{a}{n} \rfloor$ is the **quotient** of the division.

Remainder

The value $a \bmod n = r$ is the **remainder** (sometimes called residue) of division.

Congruence class modulo n

When dividing integers by n, we can divide them into n **Congruence classes**, based on the remainder r $(0 \leq r < n)$ that each integer leaves.

Congruence class modulo n

When dividing integers by n, we can divide them into n **Congruence classes**, based on the remainder r $(0 \leq r < n)$ that each integer leaves.

Congruence class definition

Two integers a and b are in the same congruence class modulo n, **if and only if** $(a - b)$ is an **integer multiple** of n. This relationship is denoted as:

$$a \equiv b \pmod{n}$$

Another way to look at this: Two integers a and b are in the same congruence class modulo n, **if and only if** the remainder when a is divided by n is same as the remainder when b is divided by n.

Congruence class modulo n

When dividing integers by n, we can divide them into n **Congruence classes**, based on the remainder r $(0 \leq r < n)$ that each integer leaves.

Congruence class definition

Two integers a and b are in the same congruence class modulo n, **if and only if** $(a - b)$ is an **integer multiple** of n. This relationship is denoted as:

$$a \equiv b \pmod{n}$$

Another way to look at this: Two integers a and b are in the same congruence class modulo n, **if and only if** the remainder when a is divided by n is same as the remainder when b is divided by n.

Example: A congruence class modulo $n = 7$

Any pair of integers $(a, b) \in \{\cdots, -11, -4, 3, 10, 17, \cdots\}$ are in the same congruence class modulo 7. That is, all numbers in that set leave a remainder of 3 when divided by 7.

Modular exponentiation (a^b mod n) on large integers

Modular Exponentiation

- A frequently occurring operation in number-theoretic computations is raising one number to a power, and then modulo dividing with another number.

$$a^b \text{ mod } n$$

- This is known as **modular exponentiation**.
- For small integers, we can first compute a^b and then modulo divide the result with n to get an final answer.
- However, this method is not attractive when dealing with large numbers. Example:

$$1729^{1023} \text{ mod } 75$$

- A popular algorithm for modular exponentiation is the method of **repeated-squaring**. – discussed below

Running example for modular exponentiation by repeated squaring: $7^{560} \bmod 561$

We want to compute $a^b \bmod n$, where $a = 7, b = 560$ and $n = 561$ (i.e., compute $7^{560} \bmod 561$)

Computation of $7^{560} \bmod 561$

- Binary representation of $b = (560)_{10}$ is $(\overset{9}{1}\ \overset{8}{0}\ \overset{7}{0}\ \overset{6}{0}\ \overset{5}{1}\ \overset{4}{1}\ \overset{3}{0}\ \overset{2}{0}\ \overset{1}{0}\ \overset{0}{0}\ )_2$

Running example for modular exponentiation by repeated squaring: 7^{560} mod 561

We want to compute a^b mod n, where $a = 7, b = 560$ and $n = 561$ (i.e., compute 7^{560} mod 561)

Computation of 7^{560} mod 561

- Binary representation of $b = (560)_{10}$ is $(\overset{9}{1}\ \overset{8}{0}\ \overset{7}{0}\ \overset{6}{0}\ \overset{5}{1}\ \overset{4}{1}\ \overset{3}{0}\ \overset{2}{0}\ \overset{1}{0}\ \overset{0}{0})_2$
- $a^b = 7^{560} = 7^{1\times 2^9 + 0\times 2^8 + 0\times 2^7 + 0\times 2^6 + 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 0\times 2^2 + 0\times 2^1 + 0\times 2^0}$

Running example for modular exponentiation by repeated squaring: $7^{560} \bmod 561$

We want to compute $a^b \bmod n$, where $a = 7, b = 560$ and $n = 561$ (i.e., compute $7^{560} \bmod 561$)

Computation of $7^{560} \bmod 561$

- Binary representation of $b = (560)_{10}$ is $\begin{array}{cccccccccc} \scriptstyle 9 & \scriptstyle 8 & \scriptstyle 7 & \scriptstyle 6 & \scriptstyle 5 & \scriptstyle 4 & \scriptstyle 3 & \scriptstyle 2 & \scriptstyle 1 & \scriptstyle 0 \\ (1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{array})_2$
- $a^b = 7^{560} = 7^{1\times 2^9 + 0\times 2^8 + 0\times 2^7 + 0\times 2^6 + 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 0\times 2^2 + 0\times 2^1 + 0\times 2^0}$
- $\implies 7^{560} = \underbrace{(7^{2^9})^1 (7^{2^8})^0 (7^{2^7})^0 (7^{2^6})^0 (7^{2^5})^1 (7^{2^4})^1 (7^{2^3})^0 (7^{2^2})^0 (7^{2^1})^0 (7^{2^0})^0}_{=7^{2^9} 7^{2^5} 7^{2^4}}$

Running example for modular exponentiation by repeated squaring: $7^{560} \bmod 561$

We want to compute $a^b \bmod n$, where $a = 7, b = 560$ and $n = 561$ (i.e., compute $7^{560} \bmod 561$)

Computation of $7^{560} \bmod 561$

- Binary representation of $b = (560)_{10}$ is $(\overset{9}{1}\ \overset{8}{0}\ \overset{7}{0}\ \overset{6}{0}\ \overset{5}{1}\ \overset{4}{1}\ \overset{3}{0}\ \overset{2}{0}\ \overset{1}{0}\ \overset{0}{0})_2$
- $a^b = 7^{560} = 7^{1\times 2^9 + 0\times 2^8 + 0\times 2^7 + 0\times 2^6 + 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 0\times 2^2 + 0\times 2^1 + 0\times 2^0}$
- $\implies 7^{560} = \underbrace{(7^{2^9})^1 (7^{2^8})^0 (7^{2^7})^0 (7^{2^6})^0 (7^{2^5})^1 (7^{2^4})^1 (7^{2^3})^0 (7^{2^2})^0 (7^{2^1})^0 (7^{2^0})^0}_{=7^{2^9} 7^{2^5} 7^{2^4}}$
- ...continued in the next slide.

...example continued: 7^{560} mod 561

Key property of modular arithmetic that we will use now is:

$x * y \text{ mod } z = (x \text{ mod } z * y \text{ mod } z) \text{ mod } z$

...example continued: 7^{560} mod 561

Key property of modular arithmetic that we will use now is:

$x * y \text{ mod } z = (x \text{ mod } z * y \text{ mod } z) \text{ mod } z$

- We saw in the previous slide that:
- $7^{560} = \underbrace{(7^{2^9})^{1}(7^{2^8})^{0}(7^{2^7})^{0}(7^{2^6})^{0}(7^{2^5})^{1}(7^{2^4})^{1}(7^{2^3})^{0}(7^{2^2})^{0}(7^{2^1})^{0}(7^{2^0})^{0}}_{=7^{2^9}7^{2^5}7^{2^4}}$

...example continued: 7^{560} mod 561

Key property of modular arithmetic that we will use now is:

$x * y \text{ mod } z = (x \text{ mod } z * y \text{ mod } z) \text{ mod } z$

- We saw in the previous slide that:
- $7^{560} = \underbrace{(7^{2^9})^{1}(7^{2^8})^{0}(7^{2^7})^{0}(7^{2^6})^{0}(7^{2^5})^{1}(7^{2^4})^{1}(7^{2^3})^{0}(7^{2^2})^{0}(7^{2^1})^{0}(7^{2^0})^{0}}_{=7^{2^9}7^{2^5}7^{2^4}}$
- To compute 7^{560} mod 561, using of the key property of modular arithmetic above, the **repeated-squaring** method works as follows:

...example continued: 7^{560} mod 561

Key property of modular arithmetic that we will use now is:

$x * y \bmod z = (x \bmod z * y \bmod z) \bmod z$

- We saw in the previous slide that:
- $7^{560} = \underbrace{(7^{2^9})^{1}(7^{2^8})^{0}(7^{2^7})^{0}(7^{2^6})^{0}(7^{2^5})^{1}(7^{2^4})^{1}(7^{2^3})^{0}(7^{2^2})^{0}(7^{2^1})^{0}(7^{2^0})^{0}}_{=7^{2^9}7^{2^5}7^{2^4}}$
- To compute 7^{560} mod 561, using of the key property of modular arithmetic above, the **`repeated-squaring`** method works as follows:

- Start by computing 7^{2^0} mod 561 which is $= 7$.

...example continued: 7^{560} mod 561

Key property of modular arithmetic that we will use now is:

$x * y \bmod z = (x \bmod z * y \bmod z) \bmod z$

- We saw in the previous slide that:
- $7^{560} = \underbrace{(7^{2^9})^{1}(7^{2^8})^{0}(7^{2^7})^{0}(7^{2^6})^{0}(7^{2^5})^{1}(7^{2^4})^{1}(7^{2^3})^{0}(7^{2^2})^{0}(7^{2^1})^{0}(7^{2^0})^{0}}_{=7^{2^9}7^{2^5}7^{2^4}}$
- To compute 7^{560} mod 561, using of the key property of modular arithmetic above, the **`repeated-squaring`** method works as follows:

- Start by computing 7^{2^0} mod 561 which is $= 7$.
- Since 7^{2^1} mod $561 = (7^{2^0} * 7^{2^0})$ mod 561

...example continued: 7^{560} mod 561

Key property of modular arithmetic that we will use now is:

$x * y \bmod z = (x \bmod z * y \bmod z) \bmod z$

- We saw in the previous slide that:
- $7^{560} = \underbrace{(7^{2^9})^{1}(7^{2^8})^{0}(7^{2^7})^{0}(7^{2^6})^{0}(7^{2^5})^{1}(7^{2^4})^{1}(7^{2^3})^{0}(7^{2^2})^{0}(7^{2^1})^{0}(7^{2^0})^{0}}_{=7^{2^9}7^{2^5}7^{2^4}}$
- To compute 7^{560} mod 561, using of the key property of modular arithmetic above, the **`repeated-squaring`** method works as follows:

- Start by computing 7^{2^0} mod 561 which is $= 7$.
- Since 7^{2^1} mod $561 = (7^{2^0} * 7^{2^0})$ mod 561(applying the above key property) $= (\underbrace{7^{2^0} \bmod 561}_{=7} * \underbrace{7^{2^0} \bmod 561}_{=7}) \bmod 561 = 7^2 \bmod 561 = 49$

...example continued: 7^{560} mod 561

Key property of modular arithmetic that we will use now is:

$x * y \text{ mod } z = (x \text{ mod } z * y \text{ mod } z) \text{ mod } z$

- We saw in the previous slide that:
- $7^{560} = \underbrace{(7^{2^9})^{1}(7^{2^8})^{0}(7^{2^7})^{0}(7^{2^6})^{0}(7^{2^5})^{1}(7^{2^4})^{1}(7^{2^3})^{0}(7^{2^2})^{0}(7^{2^1})^{0}(7^{2^0})^{0}}_{=7^{2^9}7^{2^5}7^{2^4}}$
- To compute 7^{560} mod 561, using of the key property of modular arithmetic above, the **`repeated-squaring`** method works as follows:

- Start by computing 7^{2^0} mod 561 which is $= 7$.
- Since 7^{2^1} mod $561 = (7^{2^0} * 7^{2^0})$ mod 561 (applying the above key property) $= (\underbrace{7^{2^0} \text{ mod } 561}_{=7} * \underbrace{7^{2^0} \text{ mod } 561}_{=7}) \text{ mod } 561 = 7^2 \text{ mod } 561 = 49$
- Since 7^{2^2} mod $561 = (7^{2^1} * 7^{2^1})$ mod $561 =$ $(\underbrace{7^{2^1} \text{ mod } 561}_{=49} * \underbrace{7^{2^1} \text{ mod } 561}_{=49}) \text{ mod } 561 = 49^2 \text{ mod } 561 = 157$

...example continued: $7^{560} \bmod 561$

continuing this process, we get values for each $7^{2^i} \bmod 561$:

Values of $7^{2^i} \bmod 561$

- $7^{2^0} \bmod 561 = 7$.
- $7^{2^1} \bmod 561 = 49$.
- $7^{2^2} \bmod 561 = 157$.
- $7^{2^3} \bmod 561 = 526$.
- $7^{2^4} \bmod 561 = 103$.
- $7^{2^5} \bmod 561 = 511$.
- $7^{2^6} \bmod 561 = 256$.
- $7^{2^7} \bmod 561 = 460$.
- $7^{2^8} \bmod 561 = 103$.
- $7^{2^9} \bmod 561 = 511$.

Since $7^{560} = 7^{2^9} 7^{2^5} 7^{2^4}$, we only care about the values highlighted in red. To achieve this:

- Initialize **result**=1;
- While successively finding the values of $7^{2^i} \bmod 561$...
- ...whenever the binary representation of the exponent (here $b = 560 = \overset{9}{1}\overset{8}{0}\overset{7}{0}\overset{6}{0}\overset{5}{1}\overset{4}{1}\overset{3}{0}\overset{2}{0}\overset{1}{0}\overset{0}{0}$) has a **1** at position i...
- ...update **result** = ($\textbf{result} * 7^{2^i} \bmod 561$) mod 561.
- Final **result** is the modular exponentiation of $7^{560} \bmod 561 = 1$, or more generally, $a^b \bmod n$.

Prime

A prime number (or a prime) is a natural number **greater than 1** that has **`NO`** positive divisors/factors other than 1 and itself.

Primes in the first 100 natural numbers:

```
2  3  5  7  11
13 17 19 23 29
31 37 41 43 47
53 59 61 67 71
73 79 83 89 97
```

Primality Test

Question

Given a number (arbitrarily long) as input, **is this number a prime?**

- **Prime numbers** have preoccupied human interest for time immemorial.
- Reliance on primes of modern crypto-systems (eg. RSA) makes **primality testing** an important algorithmic problem.
- From an algorithmic point-of-view, how fast/rapidly can we check if a given number n is prime?

The distribution of prime numbers

- The prime number distribution function $\pi(n)$ specifies the number of primes that are less than or equal to n.
- For example:
  - $\pi(10) = 4$ (since primes ≤ 10 are 2,3,5, and 7).
  - $\pi(100) = 25$ (since there are 25 primes ≤ 100).
  - ...
  - $\pi(1,000,000,000) = 50,847,534$.

Asymptotic distribution of prime numbers

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\log(n)} = 1.$$

That is, $n/\log(n)$ is a good approximation to $\pi(n)$.

Intuitively, this gives a rough **`probability estimate`** of $1/\log(n)$ that any chosen n is a prime.

Naïve Algorithm for Primality testing – trial division

```
function naive_test1( n ) {
  for (k in the range 2 and n-1) {
     if ( n mod k == 0) return "Composite!";
  }
  return "Prime!"
}
```

Naïve Algorithm for Primality testing – trial division

```
function naive_test1( n ) {
  for (k in the range 2 and n-1) {
     if ( n mod k == 0) return "Composite!";
  }
  return "Prime!"
}
```

Complexity

- $O(n)$ **number of divisions**.
- Basic division algorithms between two d-**bit** integers is O(d^2).
  - A (decimal) number n requires $d = \lfloor \log(n) \rfloor + 1$ bits to represent. Therefore each division is $O(d^2)$)
- Therefore, total complexity is $O(nd^2)$-time using the naïve algorithm above.
- However, this can be reduced to $O(\sqrt{n}d^2)$-time – How?

Fermat's little theorem

If p is a **prime number**, then for any integer a, the number $a^p - a$ is an **integer multiple** of p.
In the language/notation of **modular arithmetic**, this is expressed as

$$a^p \equiv a \pmod{p}.$$

If a is **not divisible by** p, Fermat's little theorem stated above is **equivalent** to the statement that $a^{p-1} - 1$ is an integer multiple of p:

$$a^{p-1} \equiv 1 \pmod{p}.$$

Fermat's primality test for any integer n

This gives a necessary (but NOT sufficient) test for primality of any number n. If $a^{n-1} \not\equiv 1 \pmod{n}$, then n is definitely composite. Otherwise, it is probably prime.

An accessible proof of Fermat's little theorem

To be handled during the lecture. NOT EXAMINABLE!

Randomized Fermat test

Before anything, we only care about primality testing when n is **odd**. For even n, one can return 'Composite' without the test below.

```
function FermatRandomizedTest( n ){
  a = choose random number in the range 1 < a < n;
  if (a^{n-1} modulo n   NOT EQUALS   1 )
     return "Composite!"
  return "PROBABLY Prime!"
}
```

- Unfortunately, this test is necessary but not sufficient.
- It tests for primality only probabilistically.
- That is, there are composite numbers (e.g. Carmichael numbers) for which this test returns 'Probably Prime' answer for some chosen values of a. Such values of a are called **Fermat's liars**.
- Refer to the example used for modular exponentiation (see slides #16–18), where $a = 7$ was a Fermat's liar for the composite $n = 561 = 3 \times 11 \times 17$

Miller-Rabin Test

Miller-Rabin primality testing tries to mitigate this problem with two modifications.

1. The algorithm test several **randomly** chosen base values of a instead of just one used previously.
2. In each test, when computing the modular exponentiation a^{p-1}, it makes use of some key observations (next slide), that reduces the chance of falsely calling a 'composite' number 'probably prime'.

Miller-Rabin Test continued

Note, testing primality of n involves testing only whe n is **odd**. Even numbers, on the otherhand, can be trivially returned as 'composite'.

Observation 1

If n is an **odd** number, then $n-1$ can be represented as

$$n - 1 = 2^s.t,$$

where t is some odd number.

Observation 2

Given n is odd, and $n-1 = 2^s.t$, where t is odd, if

$$a^{2^i.t} \equiv 1 \pmod{n} \;\; \textbf{and} \;\; a^{2^{i-1}.t} \not\equiv \pm 1 \pmod{n}, \quad 0 < i \leq s, 1 < a < n-1$$

then, n **has to be composite!**

Intuition for observation 2

- $n - 1 = 2^s.t$
- $\implies a^{n-1}$ mod $n = a^{2^s.t}$ mod n.
- We will compute $a^{2^s.t}$ mod n iteratively starting with...
- ... $a^{2^0.t}$ mod n (denoted as x_0).
- This result is progressively squared (modulo n) yielding:
  - ...$a^{2^1.t}$ mod n (denoted as x_1), which in turn yields
  - ...$a^{2^2.t}$ mod n (denoted as x_2), which in turn yields
  - ...$a^{2^3.t}$ mod n (denoted as x_3)...
  - ...and so on until
- This succesive squaring method yields a sequence of numbers: $\langle x_0, x_1, x_2, \cdots x_s \rangle$

Intuition for observation 2...continued

- The following **cases** apply to this sequence: $\langle x_0, x_1, x_2, \cdots x_s \rangle$
  1. $x_s = r \neq 1$. That is, the sequence does not end with $x_s = 1$ result.
     - ⋆ Return 'Composite'
  2. Some $x_{i-1} = r \neq \pm 1$, and $x_i = 1$. That is, the sequence is $\langle \cdots, r, 1, 1, 1 \cdots 1 \rangle$.
     - ⋆ Return 'Composite'
  3. Some $x_{i-1} = -1$, and $x_i = 1$. That is, the sequence is of the form: $\langle \cdots, -1, 1, 1, 1 \cdots 1 \rangle$.
     - ⋆ Return 'Probably prime'
  4. $x_0 = x_1 = x_2 = \cdots = x_s = 1$. That is, the sequence is **all** ones.
     - ⋆ Return 'Probably prime'

Miller-Rabin's Randomized Primality testing algorithm

```
/* Input: n > 2, is the number being tested for primality
   Input: k, a parameter that determines accuracy of the test */
function MillerRabinRandomizedPrimality( n, k ) {
   if (n is even) return "Composite!";
  /* Factor n-1 as 2^s.t, where t is odd */
  s = 0, t = n-1;
  while (t is even) {
     s = s+1;
     t = t/2
  } // at this stage, n-1 will be 2^s.t, where t is odd
  /* k random tests */
  loop (k times) {
     a = choose random number in [2...n-1);
     if ( a^{n-1}  modulo  n   NOT EQUALS    1 )  return "Composite!";
     for ( i in [1...s] ) {
        if (    a^{2^i.t} modulo n EQUALS 1
                             AND
               a^{2^{i-1}.t} modulo n NOT EQUALS (+1 or -1)
           ) return "composite!";
     }
  }
  return "probably_prime"; // accuracy depends on k
}
```

Accuracy of Miller-Rabin's algorithm

- The more the number of a's that are tested, the more is the accuracy of this test.
- There is a proof (NOT EXAMINABLE) that shows that this algorithm declares a composite number incorrectly prime with a probability of at most $\frac{1}{4^k}$, over k tests.
- That is, if $k = 64$, then the probability of a given odd composite number n to be incorrectly called a prime is $\frac{1}{2^{128}}$.

Next week

Compression-related algorithms (Lempel-Ziv etc.)

-=o0o=-

END

-=o0o=-