CS5800 – ALGORITHMS

MODULE 2. COMPLEXITY (AND COMPUTABILITY & CRYPTOGRAPHY)

Lesson 4: Fast Modular Exponentiation

Ravi Sundaram

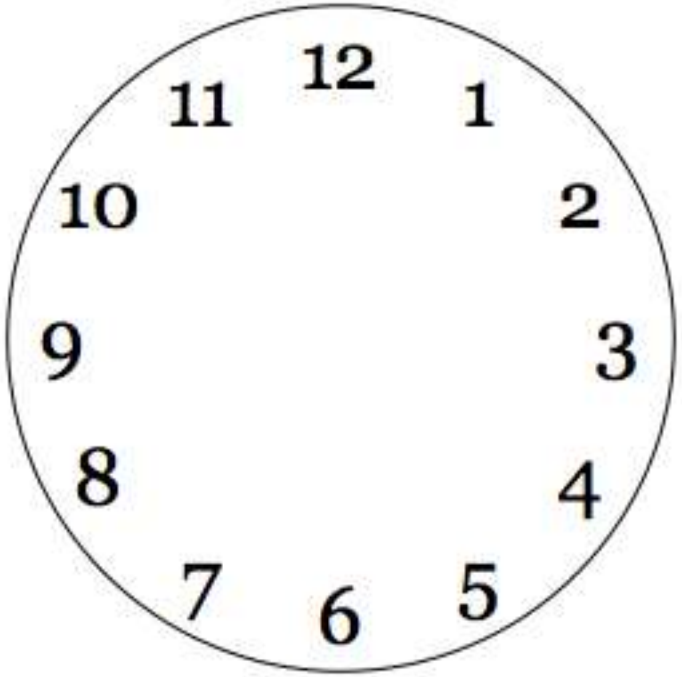
Topics

- Fast modular exponentiation - why
- Modular arithmetic
  - Quotient-remainder theorem
  - Addition and Multiplication mod n
- Fast modular exponentiation
  - Repeated squaring
- Summary

Fast modular exponentiation – why?

- Goal: to see an example of a cryptographic protocol – Diffie-Hellman key exchange – that enables a secure Internet
- Fast modular exponentiation is a critical component of Diffie-Hellman
- Why modular arithmetic?
  - Because numbers are limited in size
  - Exponentiation creates colossal output but taking mods prevents overflow
- Why modular exponentiation?
  - Because the inverse, discrete log, is hard

Modular Arithmetic



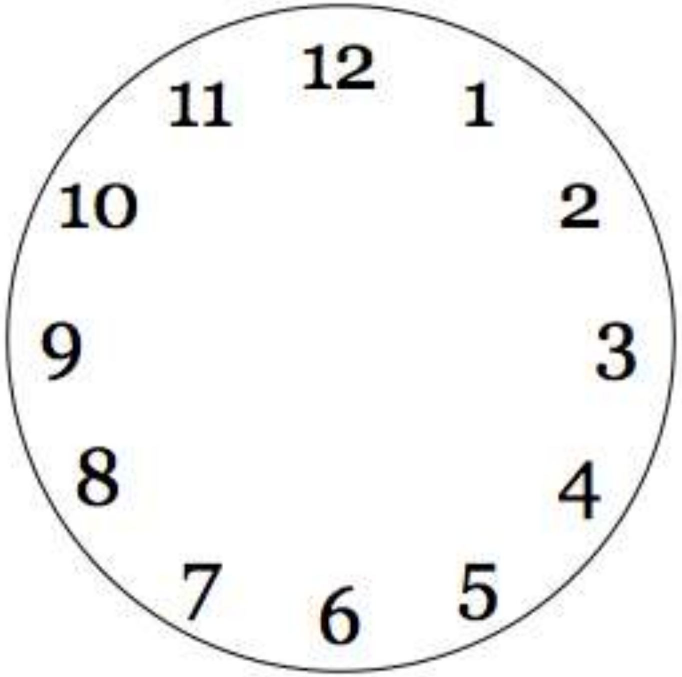
- Also known as "clock arithmetic"
- Quotient-remainder Theorem

  For integers a, $b > 0$, there exist unique q and r such that

  $a = qb + r$ where $0 \le r \le b-1$
- Definition: $a \equiv b \pmod{n}$ if $n \mid (a - b)$

  read $\equiv$ as "congruent to"
- "a mod n" means the remainder when a is divided by n
- $a \equiv b \pmod{n}$ means a and b have the same remainder when divided by n

Modular Arithmetic



- Addition: if a ≡ b (mod n) and c ≡ d (mod n) then
  (a + c) ≡ (b + d) (mod n)
- Multiplication: if a ≡ b (mod n) and c ≡ d (mod n) then
  a*c ≡ b*d (mod n)
- Example: 9876 ≡ 6 (mod 10) and 17642 ≡ 2 (mod 10)
  => 9876 + 17642 (mod 10) ≡ 6 + 2 (mod 10) ≡ 8 (mod 10)
  Also, 9876 * 17642 (mod 10) ≡ 6 * 2 (mod 10) ≡ 2 (mod 10)

Naive modular exponentiation

- Say we wish to compute 3^{2000} (mod 19)
- More generally suppose we wish to compute A^B mod N

- Doing arithmetic mod N limits the numbers to lie between 0 and N-1
- If we naively compute A^B mod N by repeatedly multiplying A with itself (mod N) B times then the complexity of the algorithm is:

  $\Theta(B * \lg^2 N)$
- But this is exponential in the size of B which is lg B

- How can we speed up the algorithm to make it polynomial-time?

Q

Fast Modular Exponentiation – Repeated Squaring

- Compute 3^{2000} (mod 19)

- Main Idea:
  - Get the powers of 3 by repeatedly squaring, but taking mod at each step.
  - Combine the powers of two to get the required exponent

- Example: $2000_{10} = 11111010000_2$ (in base 2)

$$= 1024 + 512 + 256 + 128 + 64 + 16$$

$$= 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^4$$

Modular Exponentiation Technique and Example

(All congruences are mod 19)

- Compute 3^{2000} (mod 19)

- Technique:
  - Repeatedly square 3, but take mod ***at each step***.

$3^2 \equiv 9$

$3^4 = 9^2 \equiv 81 \equiv 5$

$3^8 = 5^2 \equiv 25 \equiv 6$

$3^{16} = 6^2 \equiv 36 \equiv 17$

$3^{32} = 17^2 \equiv 289 \equiv 4$

$3^{64} = 4^2 \equiv 16$

$3^{128} = 16^2 \equiv 256 \equiv 9$

$3^{256} = 9^2 \equiv 81 \equiv 5$

$3^{512} = 5^2 \equiv 25 \equiv 6$

$3^{1024} = 6^2 \equiv 36 \equiv 17$

  - Then multiply the terms you need to get the desired power.

$$3^{2000} \equiv (3^{1024}) * (3^{512}) * (3^{256}) * (3^{128}) * (3^{64}) * (3^{16})$$

$$\equiv 17 * 6 * 5 * 9 * 16 * 17 \equiv 1248480$$

$$\equiv 9$$

Modular Exponentiation is extremely efficient since the partial results are always small

- Key idea: use **repeated squaring** and take mods to reduce size
- Repeated Squaring – to compute A^B mod N
  - Represent B in binary, $B = B_x B_{x-1} \ldots B_1 B_0$, {note x = lg B}
  - Compute A, A^2 $(A^2)^2 = A^{2^\wedge 2}$, $A^{2^\wedge 3}$ …. $A^{2^\wedge x}$, taking mods at each step
  - Multiply the powers corresponding to the bits of B that are 1
- Correctness – straightforward, since:

  A*B mod N = (A mod N) * (B mod N) mod N
- Complexity – takes at most 2x modular multiplications of numbers at most the size of N, i.e. $O(\lg B * (\lg N)^2)$

Alternate way to express Modular Exponentiation through Repeated Squaring

```
Function ModExp(B)
If B = 0
   then return 1
ElseIf B mod 2 = 0
   then return ModExp(B/2)^2 mod N
Else return A * ModExp((B-1)/2)^2 mod N
```

Summary

- Normal arithmetic can cause overflow
- So we use modular arithmetic
- We are looking for a function that is “one way” i.e. fast to compute in the forward direction but slow in the reverse direction
- Modular exponentiation is a candidate since the inverse – discrete log – is slow to compute
- But naïve exponentiation is also slow
- Key idea: **repeated squaring** – speeds up modular exponentiation