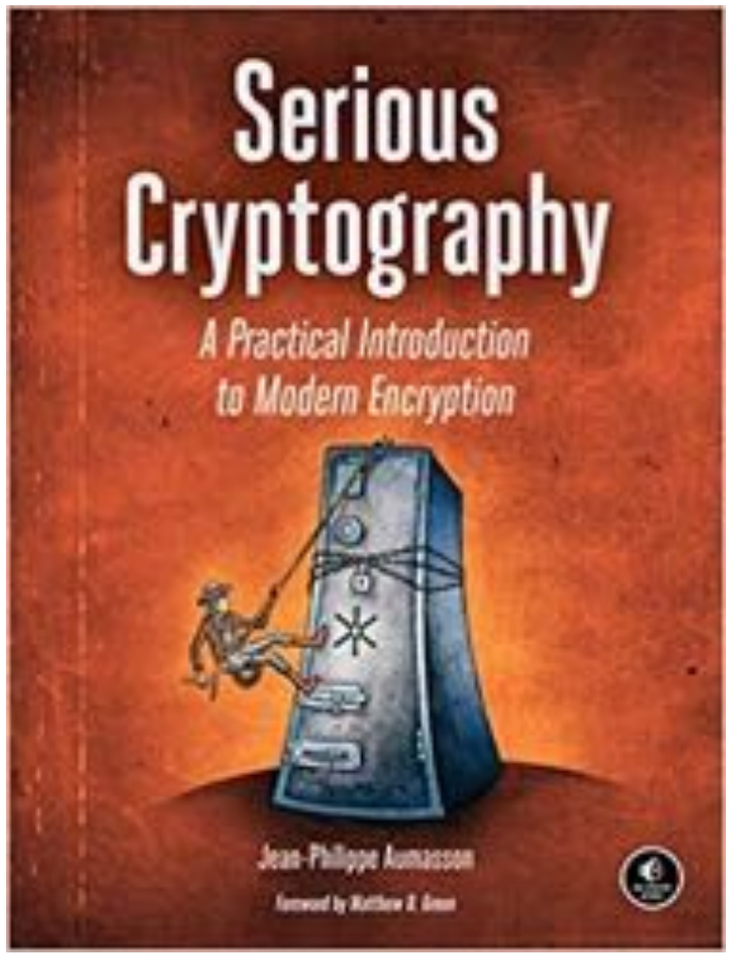
CNIT 141

Cryptography for Computer Networks



10.RSA

Topics

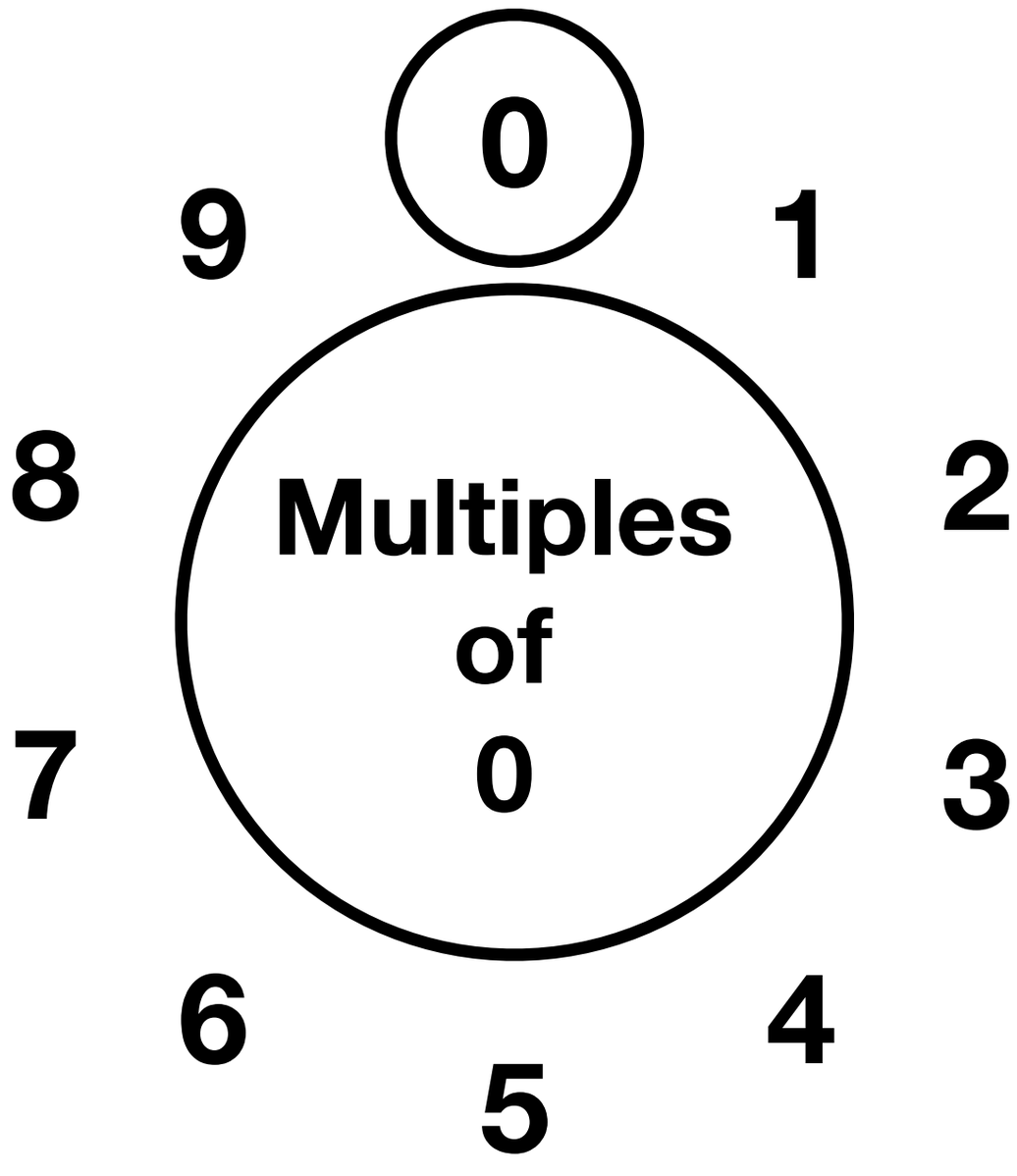
- The Math Behind RSA
- The RSA Trapdoor Permutation
- RSA Key Generation and Security
- Encrypting with RSA
- Signing with RSA
- RSA Implementations
- How Things Can Go Wrong

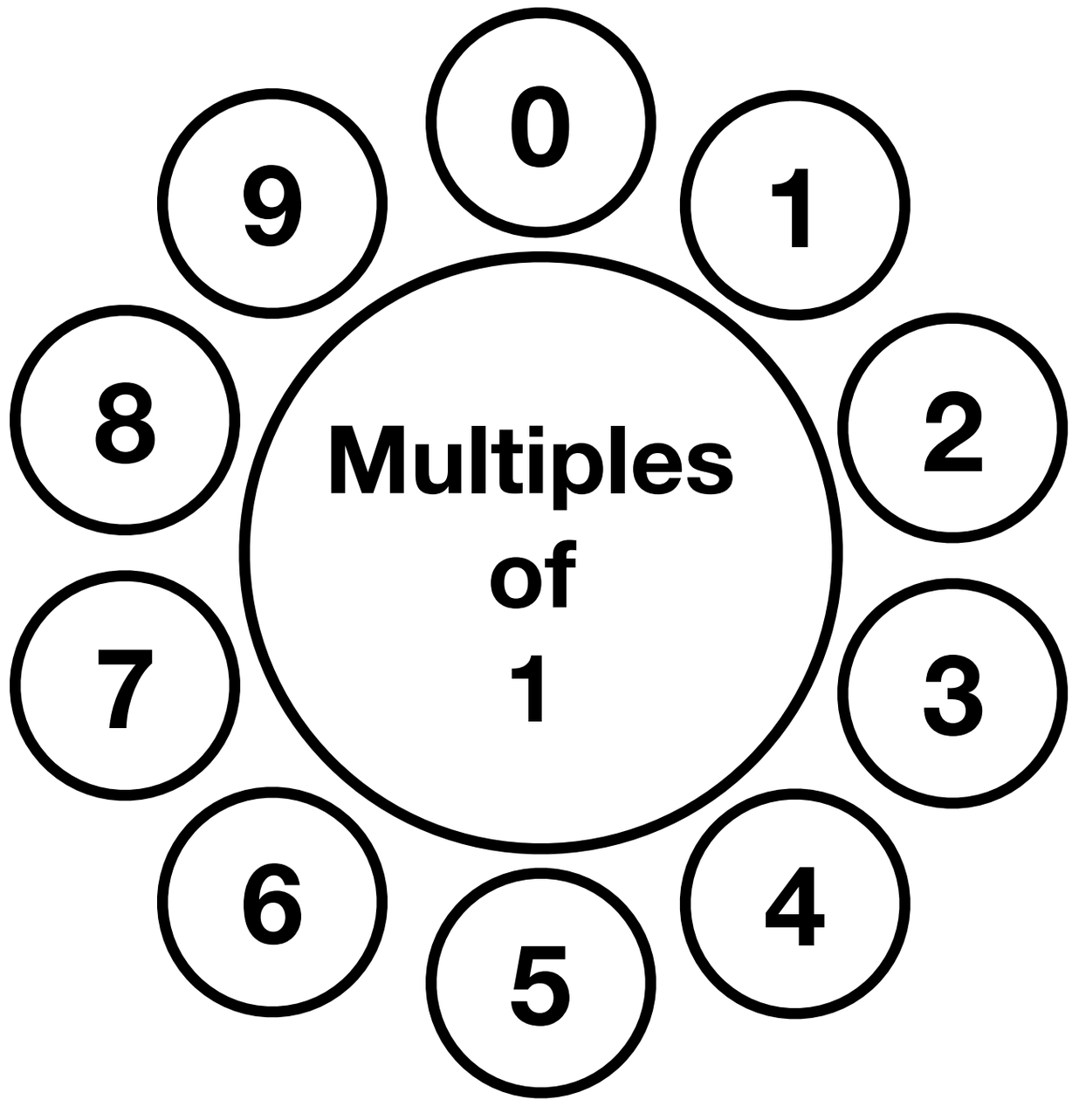
The Math Behind RSA

The Group $\mathbf{Z}_N^*$

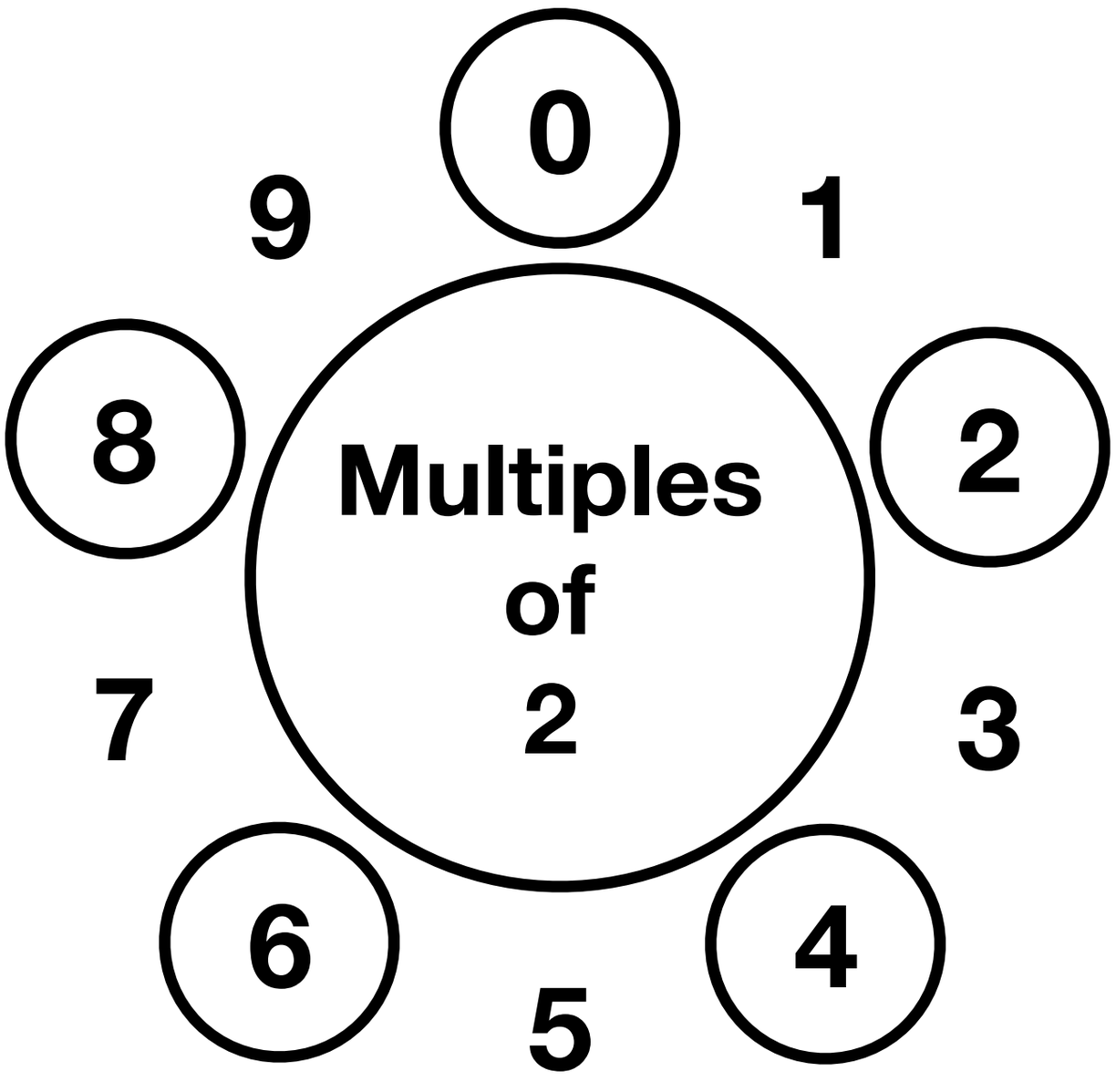
- If N is prime, $\mathbf{Z}_p^*$ contains {1,2, 3, ... p-1}
  - So $\mathbf{Z}_5^*$ contains {1,2,3,4}
    - 0 is excluded because it has no inverse
- But $\mathbf{Z}_4^*$ contains {1,3}
  - 2 has no inverse
    - Because 4 is a multiple of 2

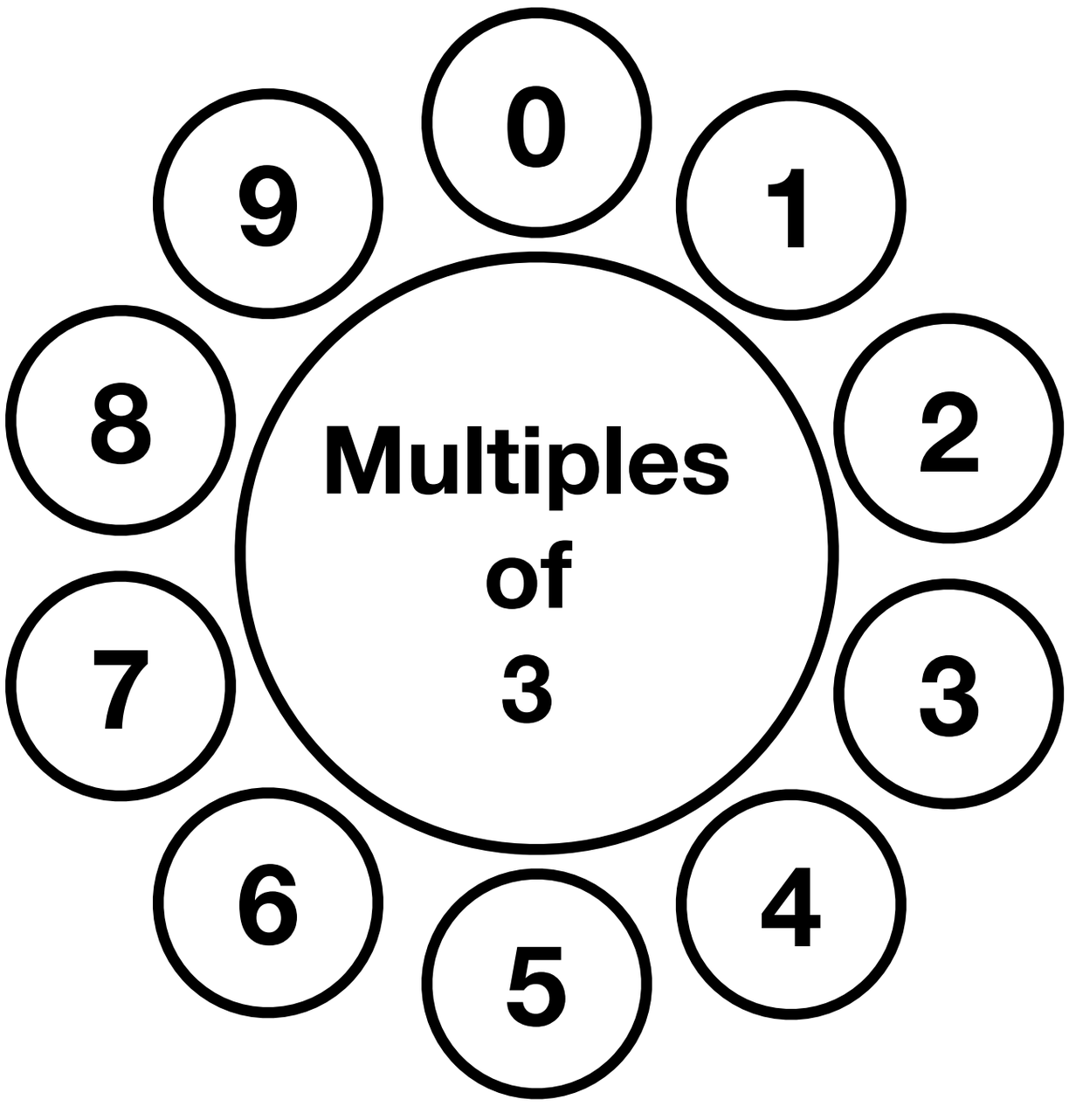
N = 10



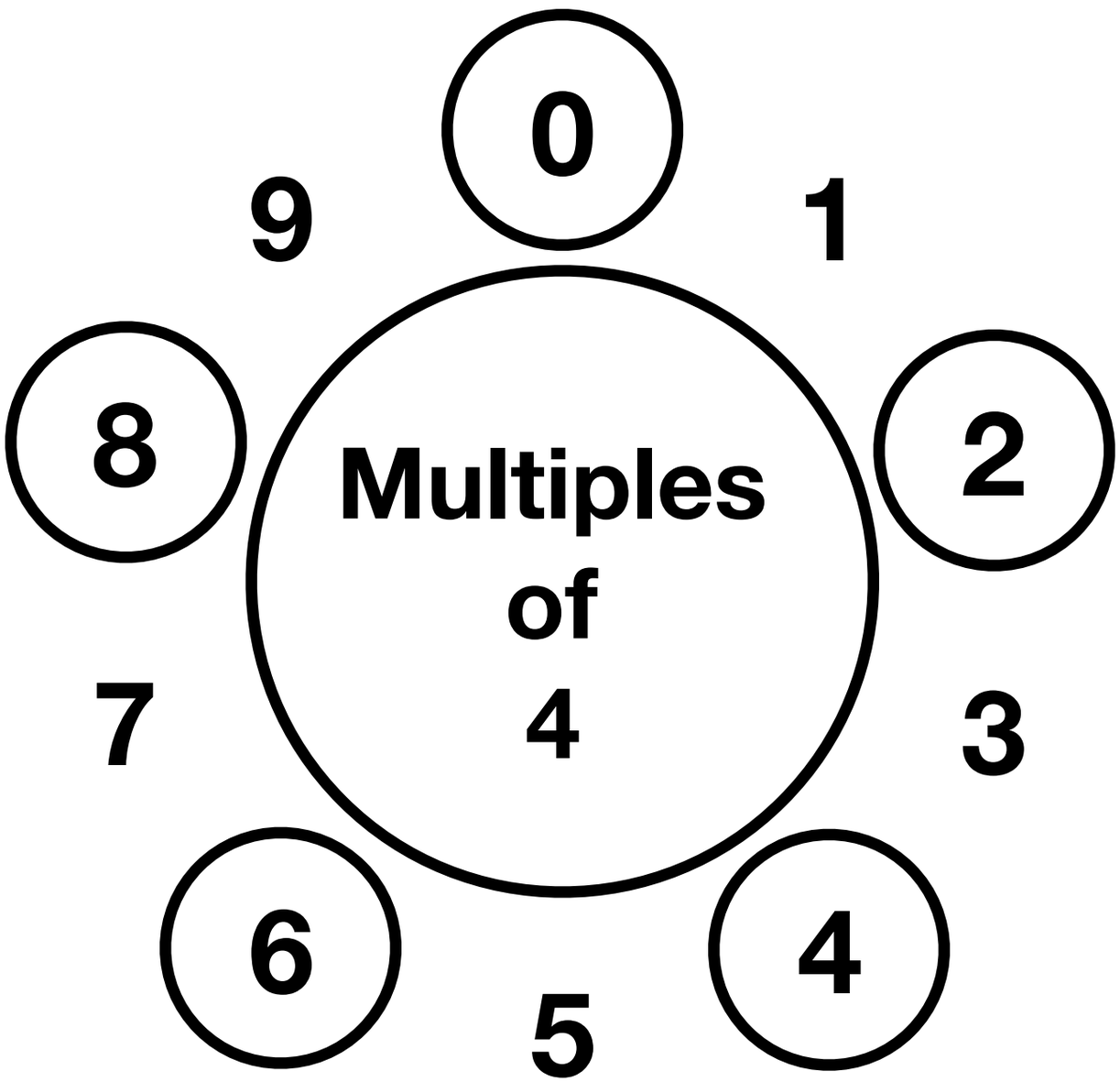


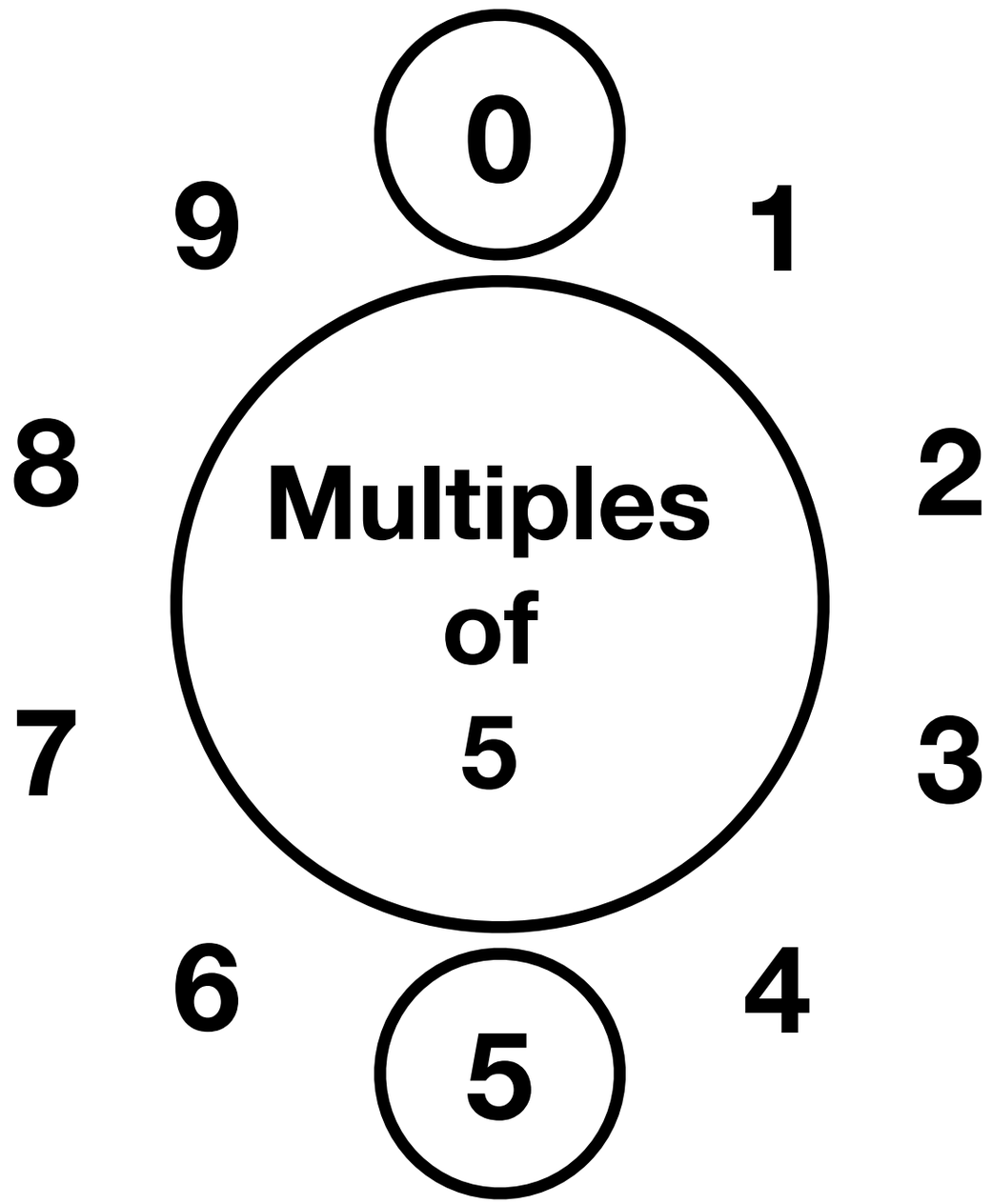
N = 10



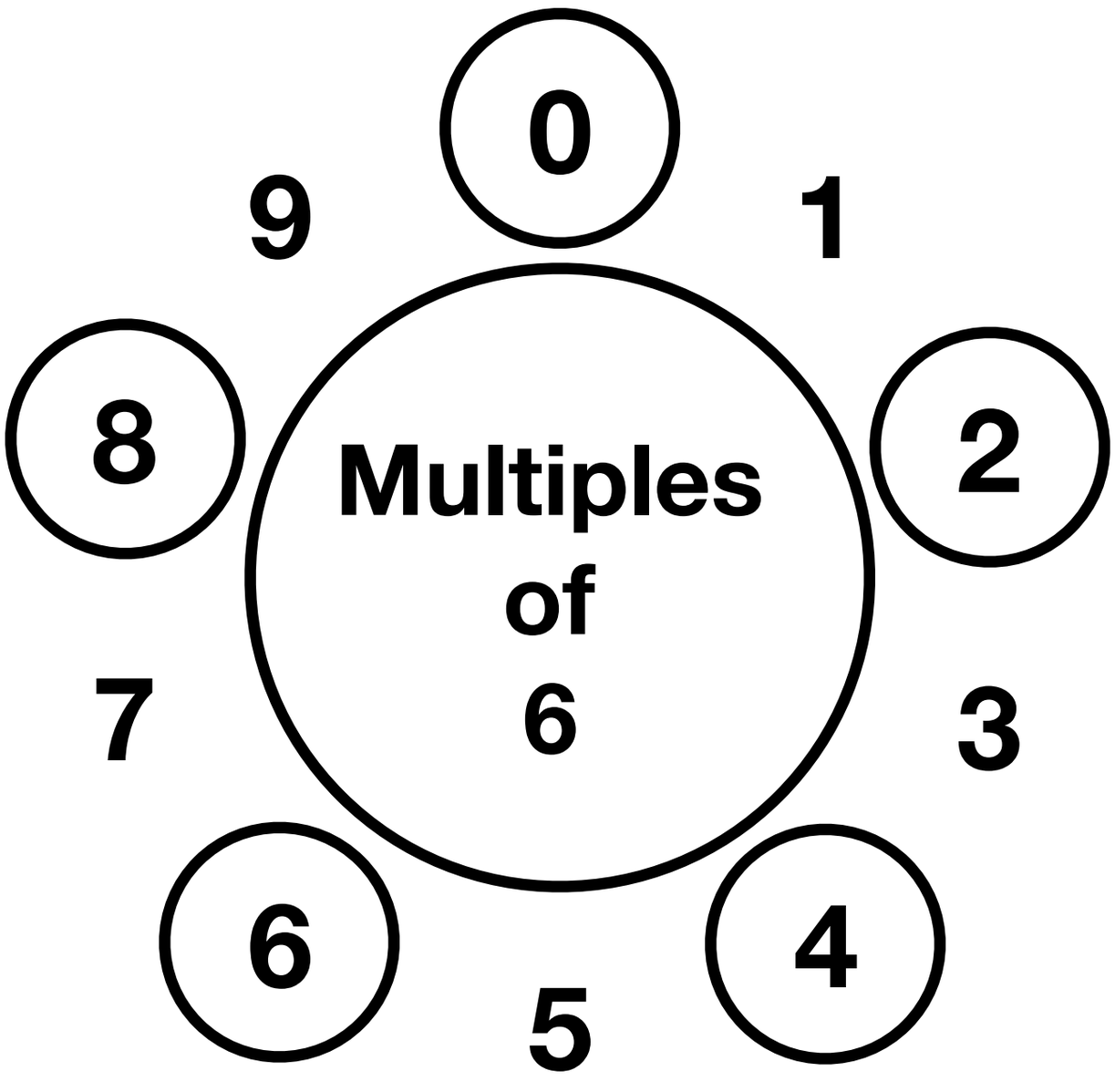


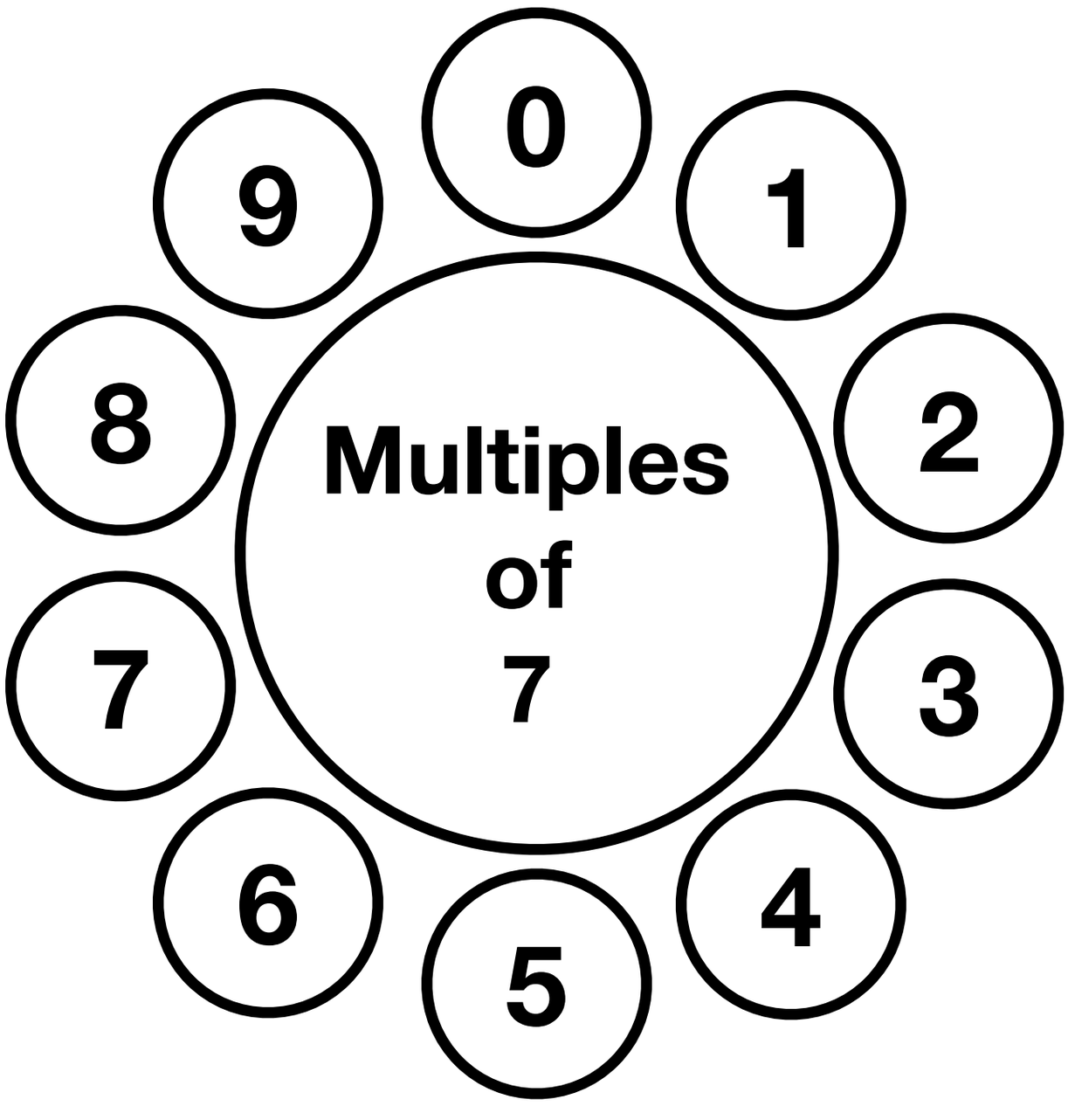
N = 10



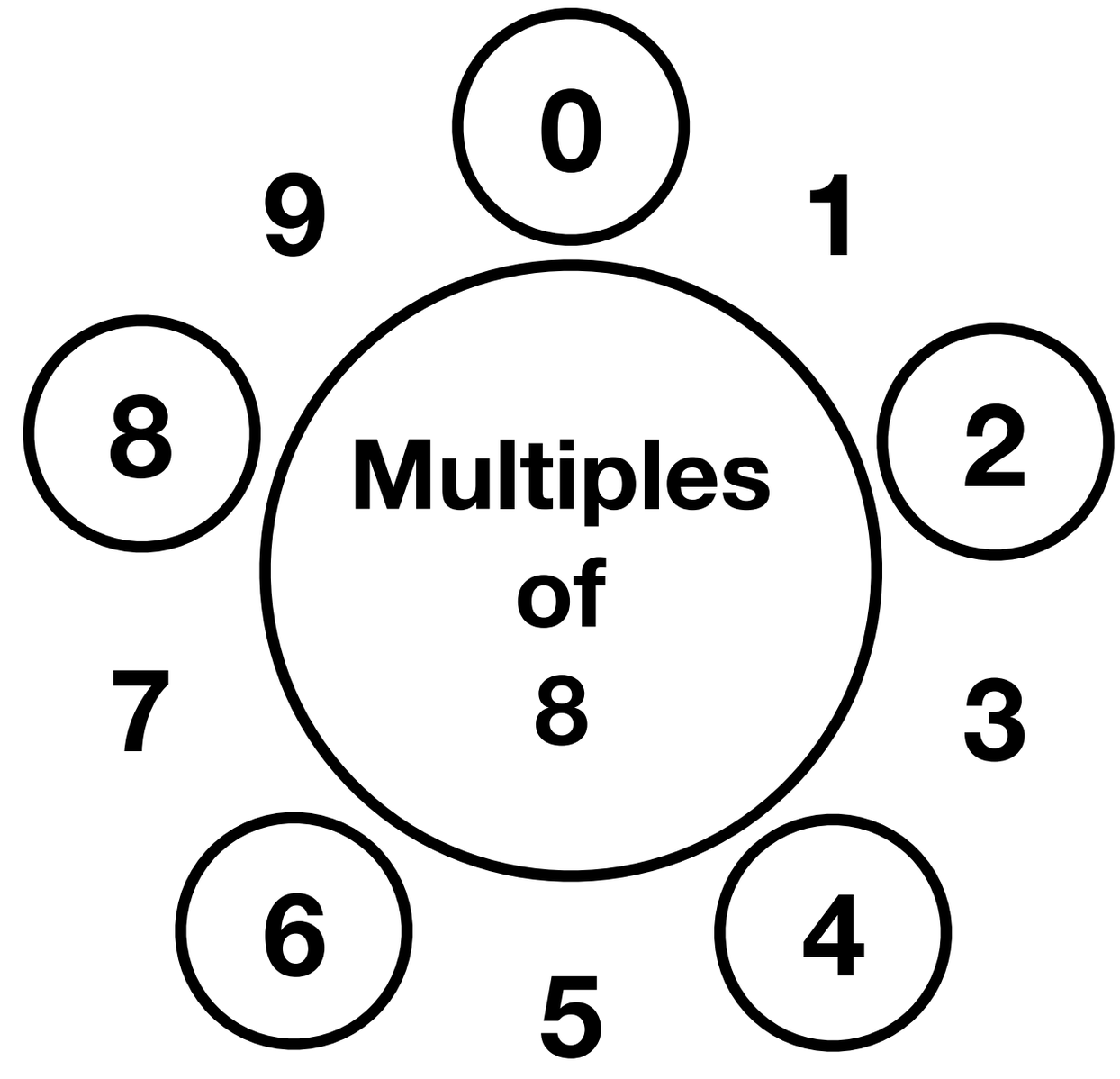


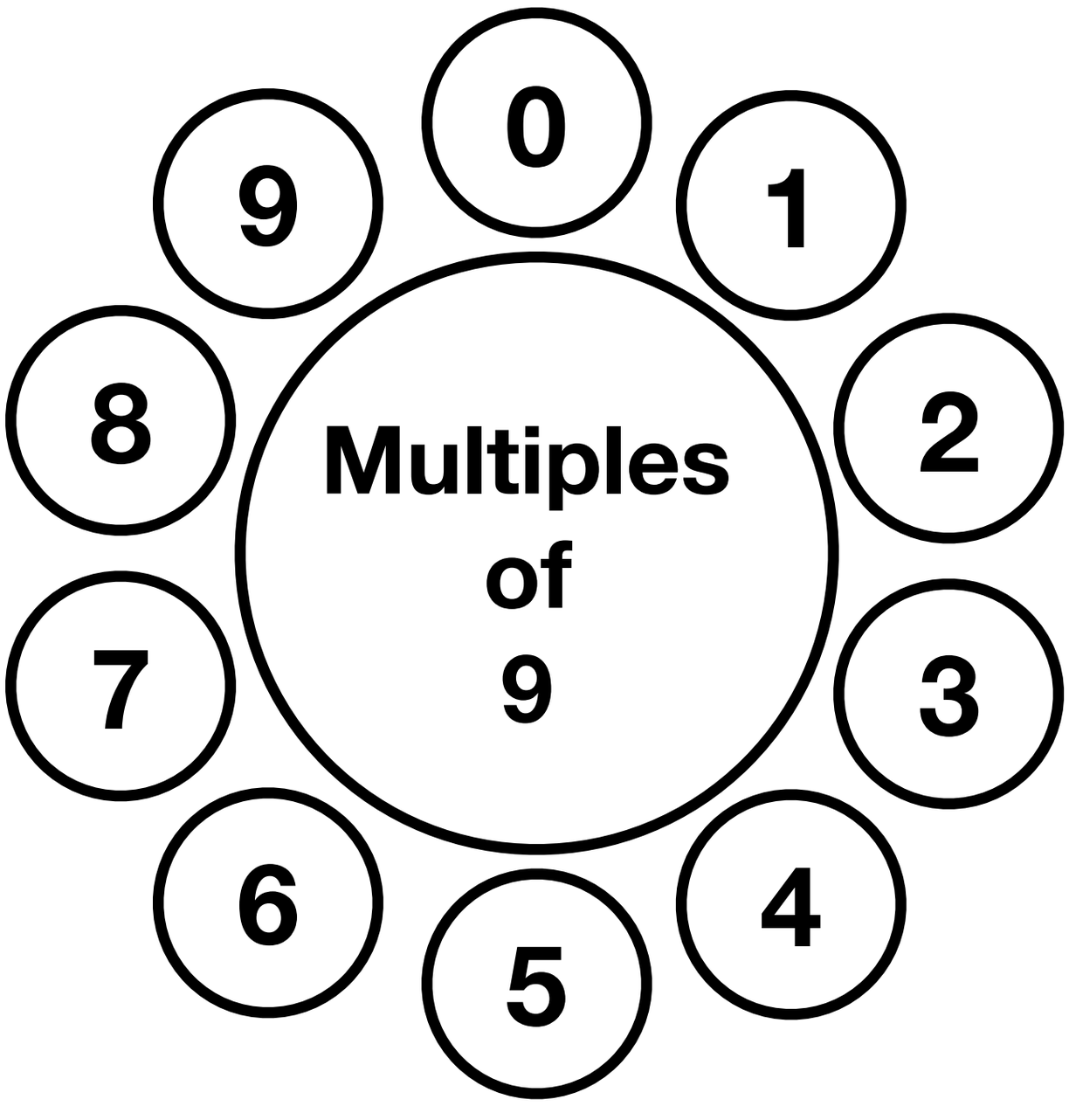
N = 10





N = 10





The Group $\mathbf{Z_{10}^*}$

- $\mathbf{Z_{10}^*}$ contains {1, 3, 7, 9}
- 0, 2, 4, 5, 6, 8 are excluded
- Because they have no inverse
- Because they have a common factor with 10
- They aren't ***co-prime*** with 10

Euler's Totient

- How many elements are in the group Z_n^*?
  - When ***n*** is not prime, but the product of several prime numbers
    - $n = p_1 \times p_2 \times \ldots \times p_m$
- $\phi(n) = (p_1 - 1) \times (p_2 - 1) \times \ldots \times (p_m - 1)$
- For $\mathbf{Z_{10}^*}$
  - $n = 10 = 2 \times 5$
  - $\phi(10) = 1 \times 4$: 4 elements in $\mathbf{Z_{10}^*}$

The RSA Trapdoor Permutation

RSA Parameters

- ***n*** is the ***modulus***
  - Product of two primes ***p*** and ***q***
- ***e*** is the ***public exponent***
  - In practice, usually 65537
- Public key: (***n***, ***e***)
- Private key: ***p*** and other values easily derived from it

Trapdoor Permutation

- ***x*** is the plaintext message
- ***y*** is the ciphertext

  Encryption: $y = x^e \bmod n$

  Decryption: $x = y^d \bmod n$

- ***d*** is the decryption key
  - calculated from ***p*** and ***q***

Calculating *d*

- $ed = 1 \bmod \phi(n)$
- Decryption: $x = y^d \bmod n$

$= (x^e)^d \bmod n$

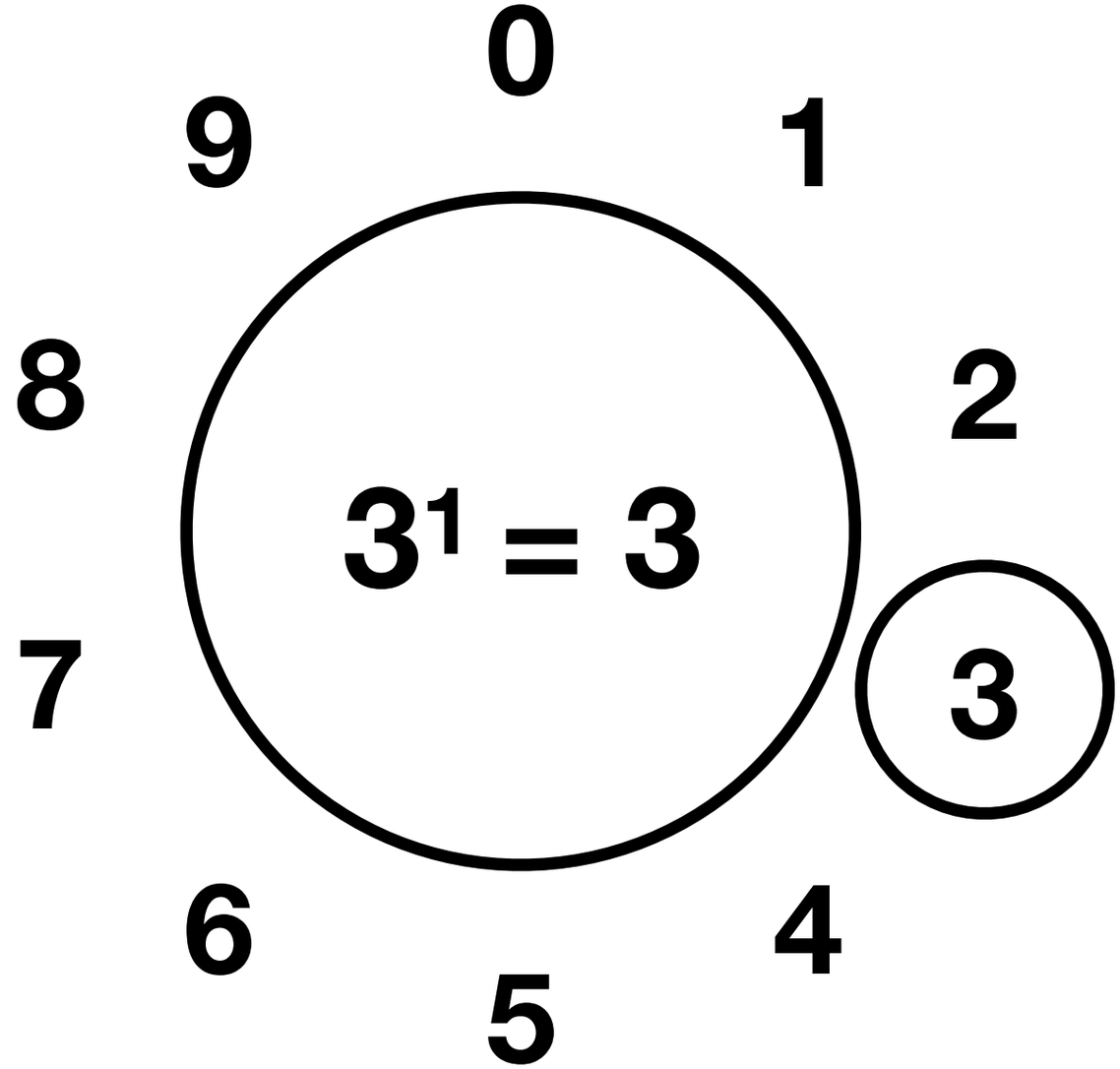
$= x^{ed} \bmod n$

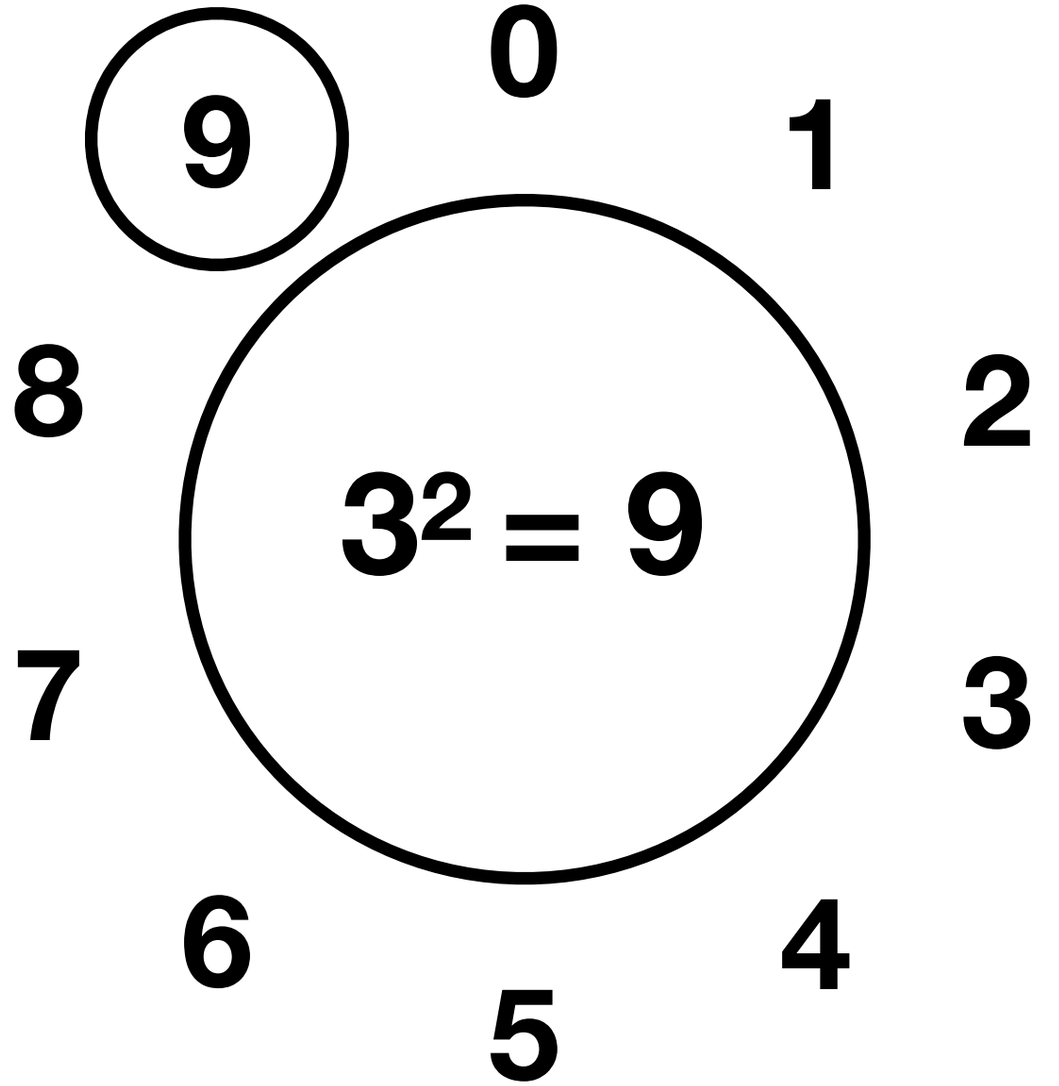
$= x \bmod n$

$= x$

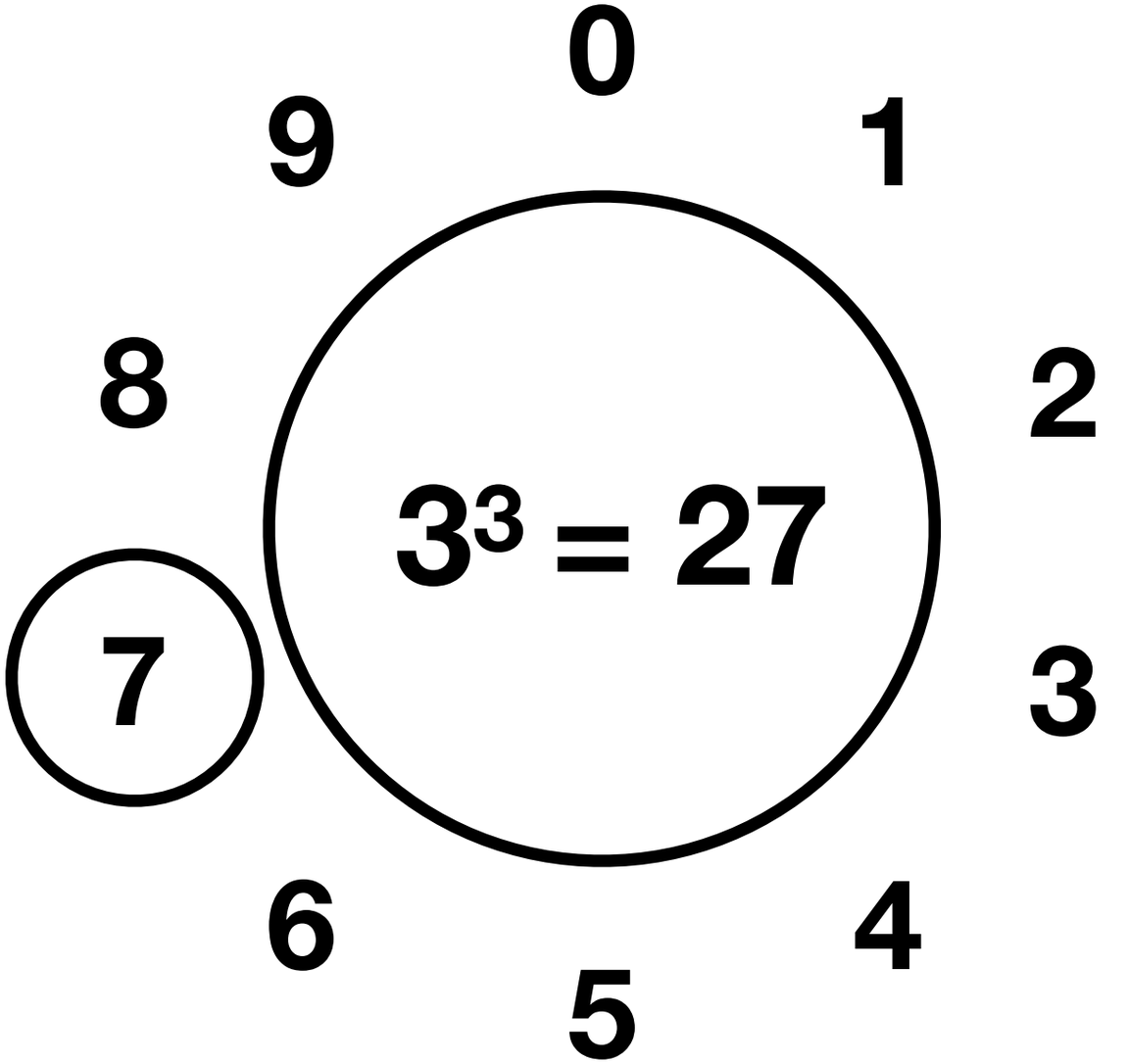
Example: n=10

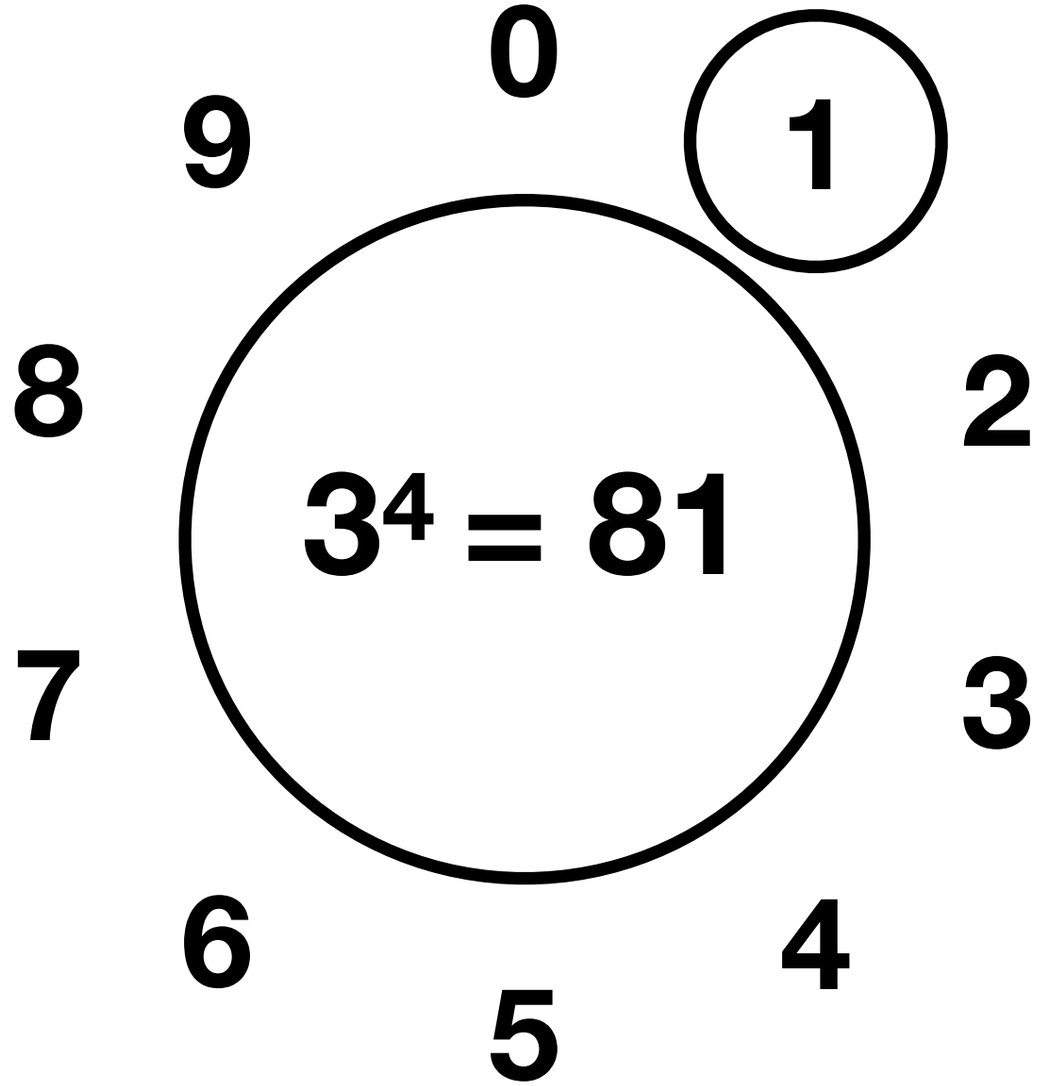
n=10; x=3





n=10; x=3





Powers of 3

$$3^1 \bmod 10 \equiv 3$$
$$3^2 \bmod 10 \equiv 9$$
$$3^3 \bmod 10 \equiv 7$$
$$3^4 \bmod 10 \equiv 1$$
$$3^5 \bmod 10 \equiv 3$$

- $\mathbf{Z_{10}^*}$ contains {1, 3, 7, 9}
  - 4 elements
  - 3 is a ***generator*** of the group
  - Although ***n*** is 10, the powers of 3 repeat with a cycle of 4 ($\boldsymbol{\phi}(\boldsymbol{n})$)
  - Encrypt by raising ***x*** to a power, forming ***y***
  - Decrypt by raising ***y*** to a power, returning ***x***

Finding ***d***
n=10; ***x***=3; ***e***=3

- $\mathbf{Z_{10}^*}$ contains {1, 3, 7, 9}
  - 4 elements
- ***p* = 2** and ***q* = 5**
- **ϕ(n) = (p-1)(q-1) = 1x4 = 4**
- ***ed*** = 1 mod **ϕ(n)**
- For ***e* = 3**, ***d* = 3**

3x1 mod 4 = 3
3x2 mod 4 = 2
3x3 mod 4 = 1

n=10; x=3; e=3; d=7

- x is the plaintext message (3)
- y is the ciphertext

Encryption: $y = x^e \bmod n$

$= 3^3 \bmod 10 = 7$

Decryption: $x = y^d \bmod n$

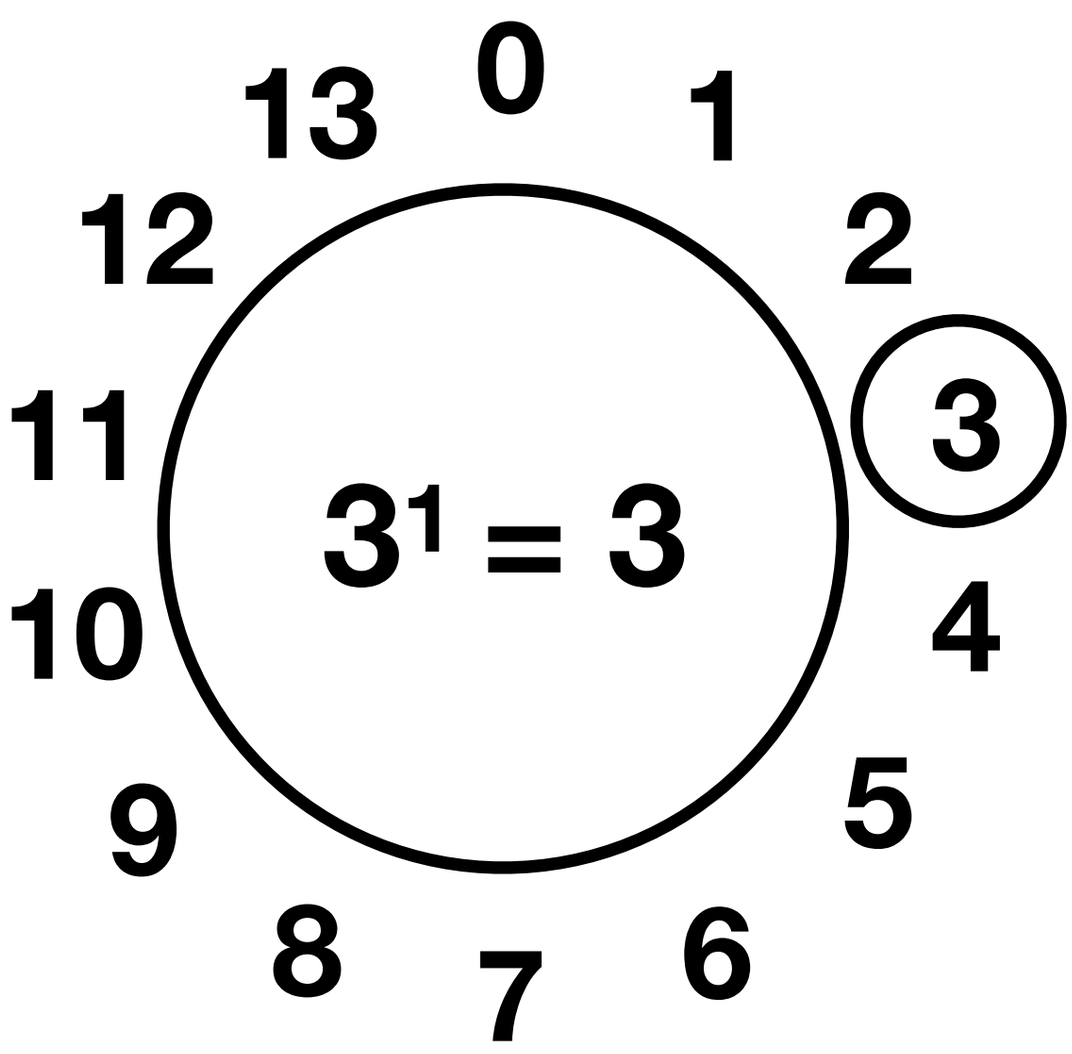
$= 7^3 \bmod 10$

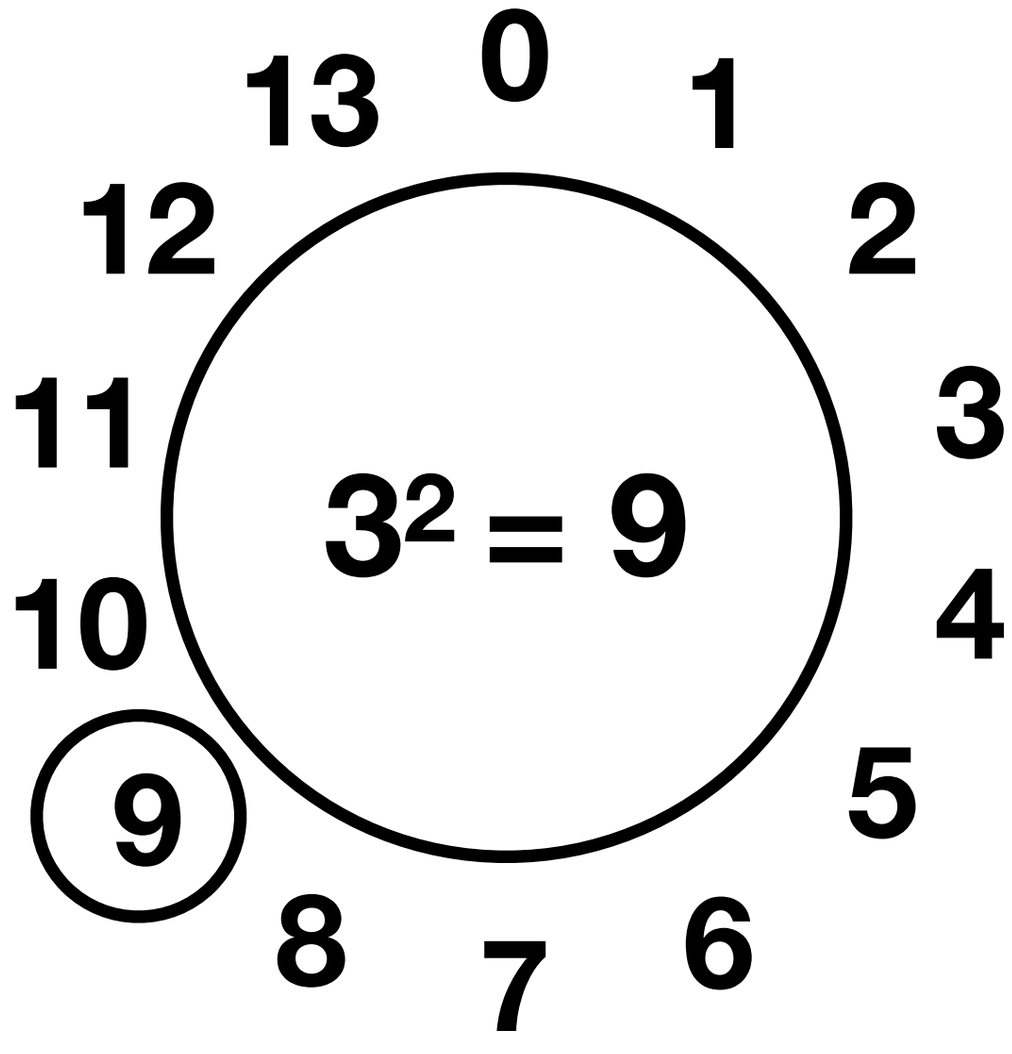
$= 343 \bmod 10$

$= 3$

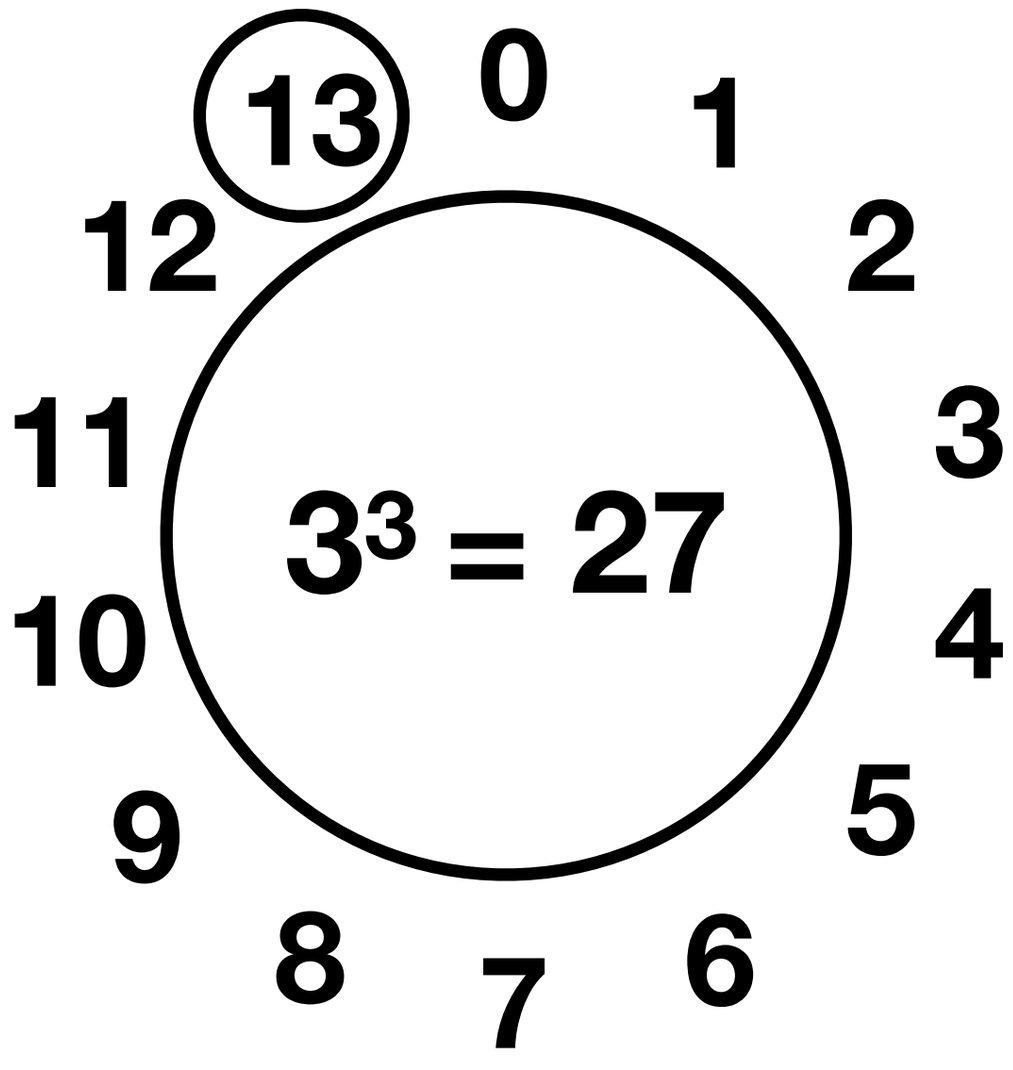
Example: n=14

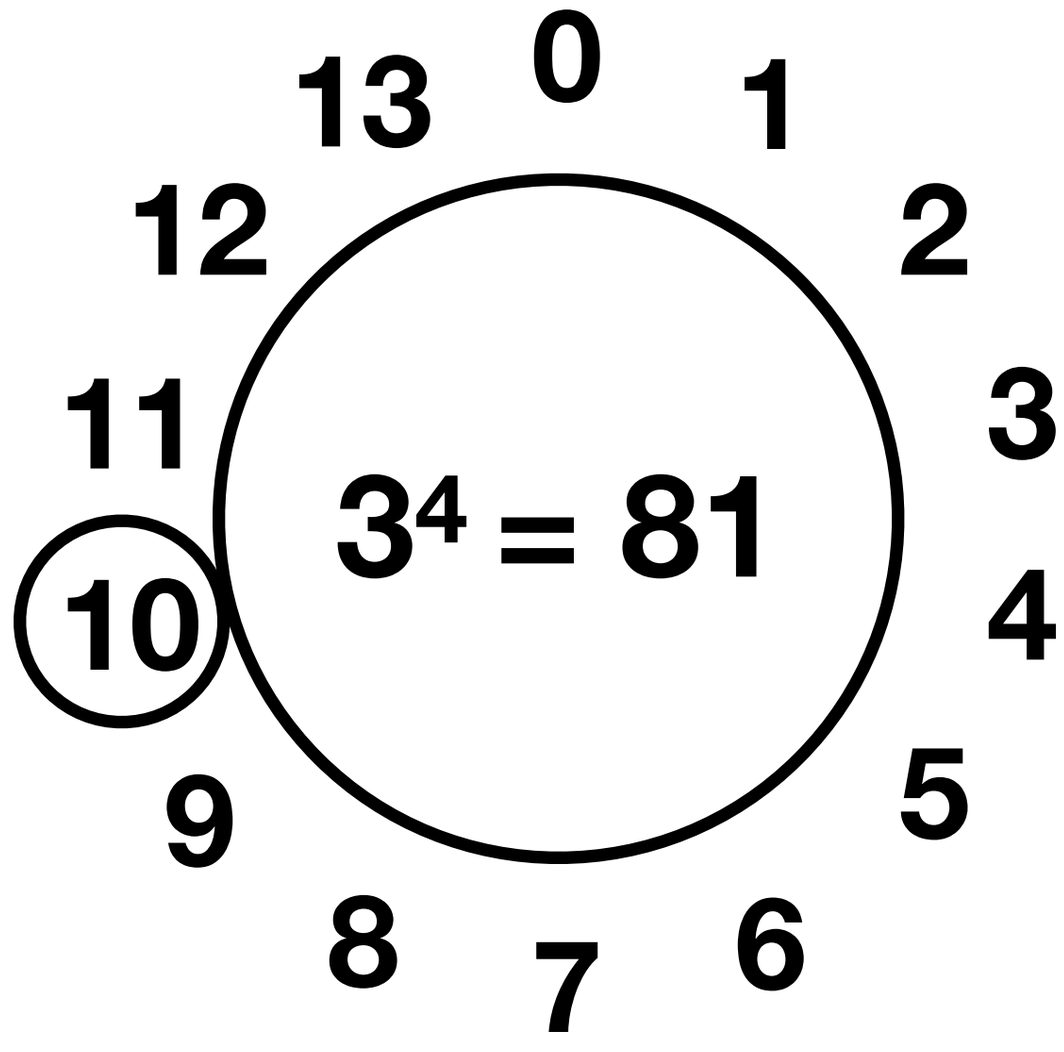
n=14; x=3



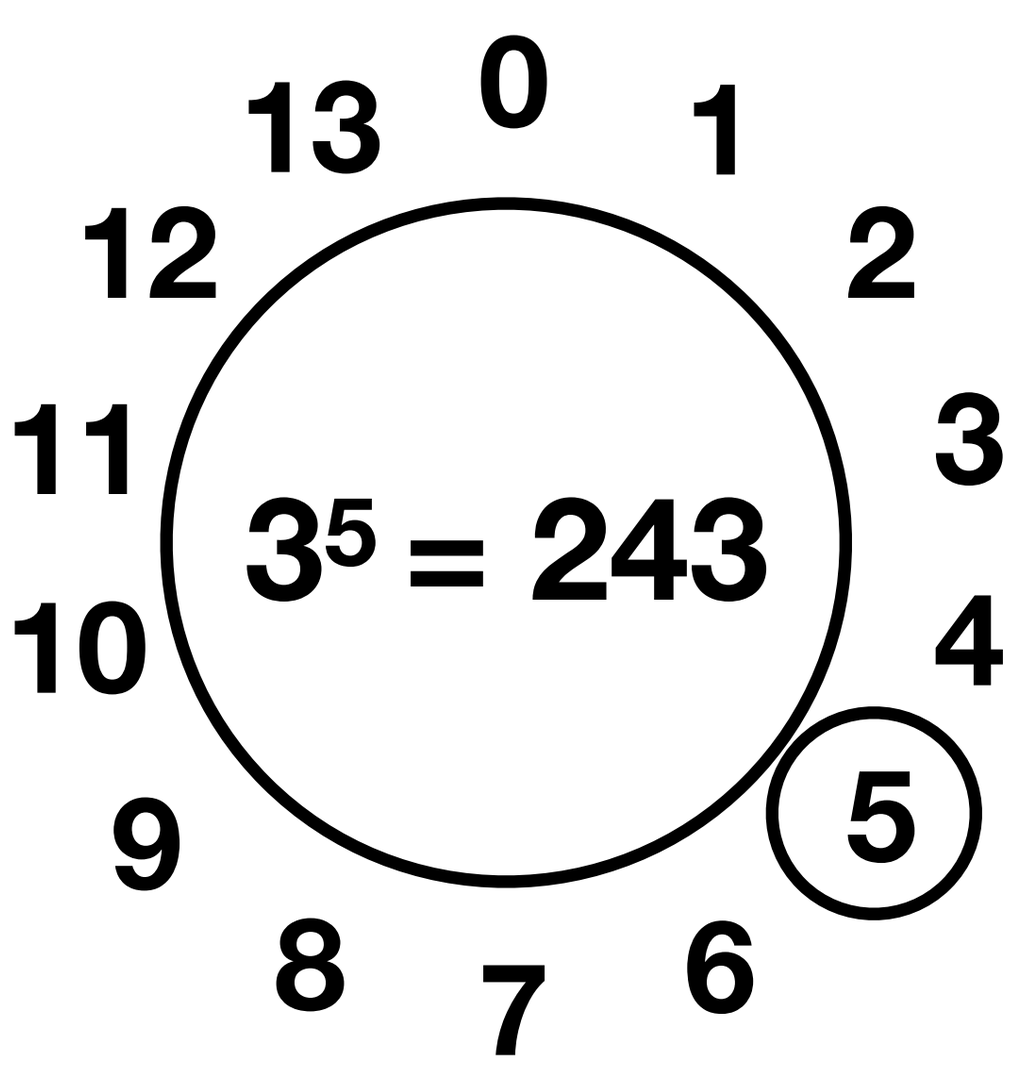


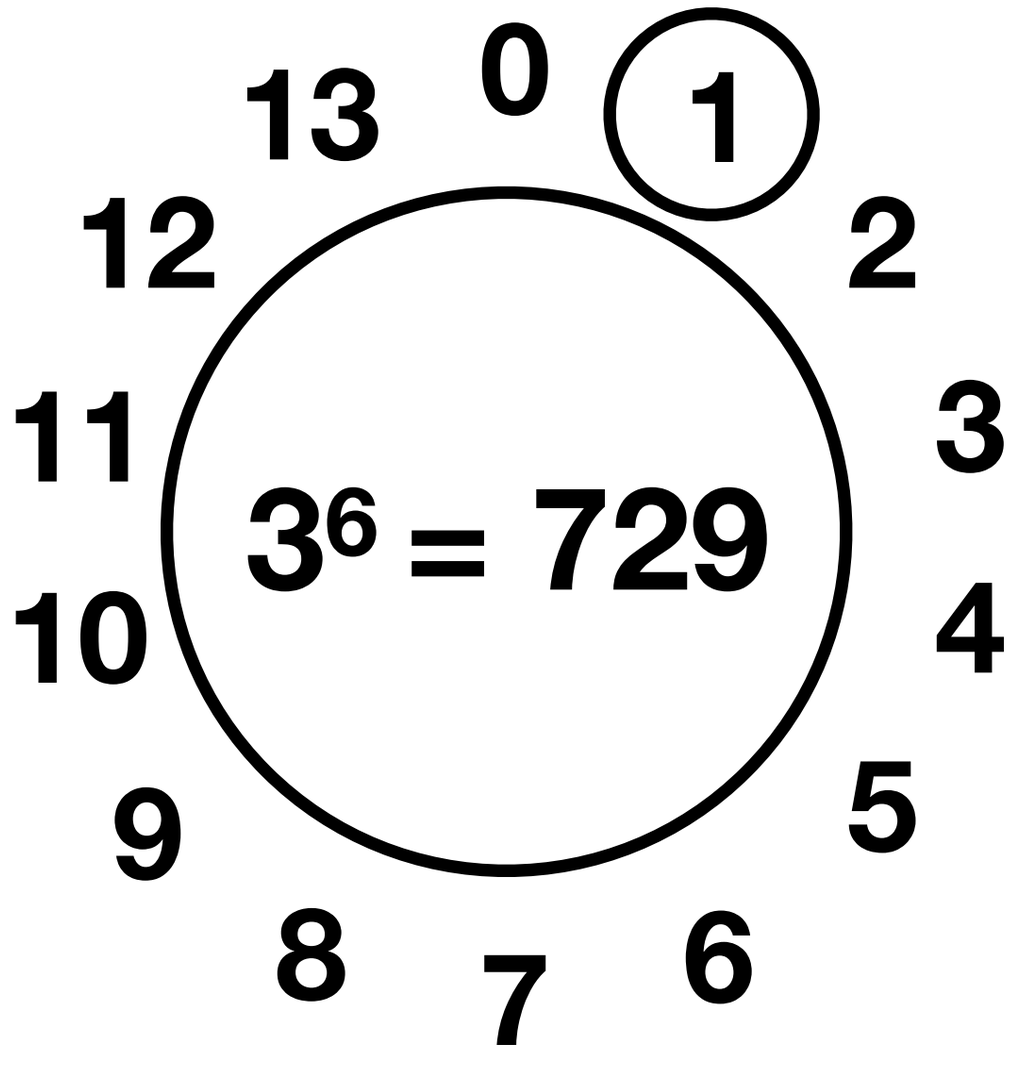
n=14; x=3





n=14; x=3





Powers of 3

- $\mathbf{Z_{14}^*}$ contains
  {1, 3, 5, 9, 11, 13}
  - 6 elements ($\boldsymbol{\phi}(\boldsymbol{n})$)
- 3 is a ***generator*** of the group

$\mathbf{3^1 \bmod 14 = 3}$

$\mathbf{3^2 \bmod 14 = 9}$

$\mathbf{3^3 \bmod 14 = 13}$

$\mathbf{3^4 \bmod 14 = 11}$

$\mathbf{3^5 \bmod 14 = 5}$

$\mathbf{3^6 \bmod 14 = 1}$

Finding ***d***
n=14; ***x***=3; ***e***=5

- $\mathbf{Z_{14}^*}$ contains {1, 3, 5, 9, 11, 13}
  - 6 elements
- ***p* = 2** and ***q* = 7**
- **$\phi(n) = (p-1)(q-1) = 1 \times 6 = 6$**
- $ed = 1 \bmod \phi(n)$
- For ***e* = 5**, ***d* = 5**

5x1 mod 6 = 5
5x2 mod 6 = 4
5x3 mod 6 = 3
5x4 mod 6 = 2
5x5 mod 6 = 1

n=14; x=3; e=5; d=5

- x is the plaintext message (3)
- y is the ciphertext

Encryption: $y = x^e \bmod n$

$= 3^5 \bmod 14 = 5$

Decryption: $x = y^d \bmod n$

$= 5^5 \bmod 14$

$= 3125 \bmod 14$

$= 3$

RSA Key Generation and Security

Key Generation

- Pick random primes ***p*** and ***q***
- Calculate **ϕ(*n*)** from ***p*** and ***q***
- Pick ***e***
- Calculate ***d*** (inverse of ***e***)

RSA Encryption in Python

```
>>> from Crypto.PublicKey import RSA
>>> key = RSA.generate(2048)
>>> publickey = key.publickey()
>>> plain = 'encrypt this message'
>>> ciphertext = publickey.encrypt(plain, 0)[0]
>>> print ciphertext.encode("hex")
536eda071ab9e526442f2b56e71fa5abfc603c88c2eac03d91f22bab6d0ea14bab2e8c8247df477c
5f15ce3ccc551227799d1f4f8943fa8bd278639bd90292c5799d11f9f6601c94d88f10fc314317fb
1d75f55e20d1c5dd4e7448ff39018dab44091b6664610657516bfaf95a3f0e63e9194f1e08343421
f7cf8c35550ed951b240e4c42f94b8bfc73ec3ccd519f7c489c28aaf799c78d6a695707423f72c05
4edfd8f4c2ac0f5c25a996647b8958f160983db8bdf2214fe131b0f3d558aeb7560e67f0621f0224
fd21f18034eebb9c8773e6310f80975539765d7235235a446f037179e94e504b21f9ffac6679570a
95848f238cdd3243723ed4722e549498
```

RSA Decryption in Python

```
>>> decrypted = key.decrypt(ciphertext)
>>> print decrypted
encrypt this message
```

Speed of Calculations

```
LENGTH:  1024
0.150621891022  sec. for one RSA key generation
0.026349067688  sec. for 400 RSA encryptions
0.0133030414581  sec. for 5 RSA decryptions
```

- Encryption is **fastest**
- Decryption is **much slower**
- Key generation is **slowest**

Encrypting with RSA

Used with AES

- RSA typically not used to encrypt plaintext directly
- RSA is used to encrypt an AES private key

Textbook RSA

- Plaintext converted to ASCII bytes
- Placed in x
- RSA used to compute $y = x^e \bmod n$

Malleability

- Encrypt two plaintext messages x_1 and x_2
  - $y_1 = x_1^e \bmod n$
  - $y_2 = x_2^e \bmod n$
- Consider the plaintext $(x_1)(x_2)$
  - Multiplying x_1 and x_2 together
    - $y = (x_1^e \bmod n)(x_2^e \bmod n) = (y_1)(y_2)$
- An attacker can create valid ciphertext without the key

Strong RSA Encryption: OAEP

- Optimal Asymmetric Encryption Padding
- Padded plaintext is as long as ***n***
- Includes extra data and randomness

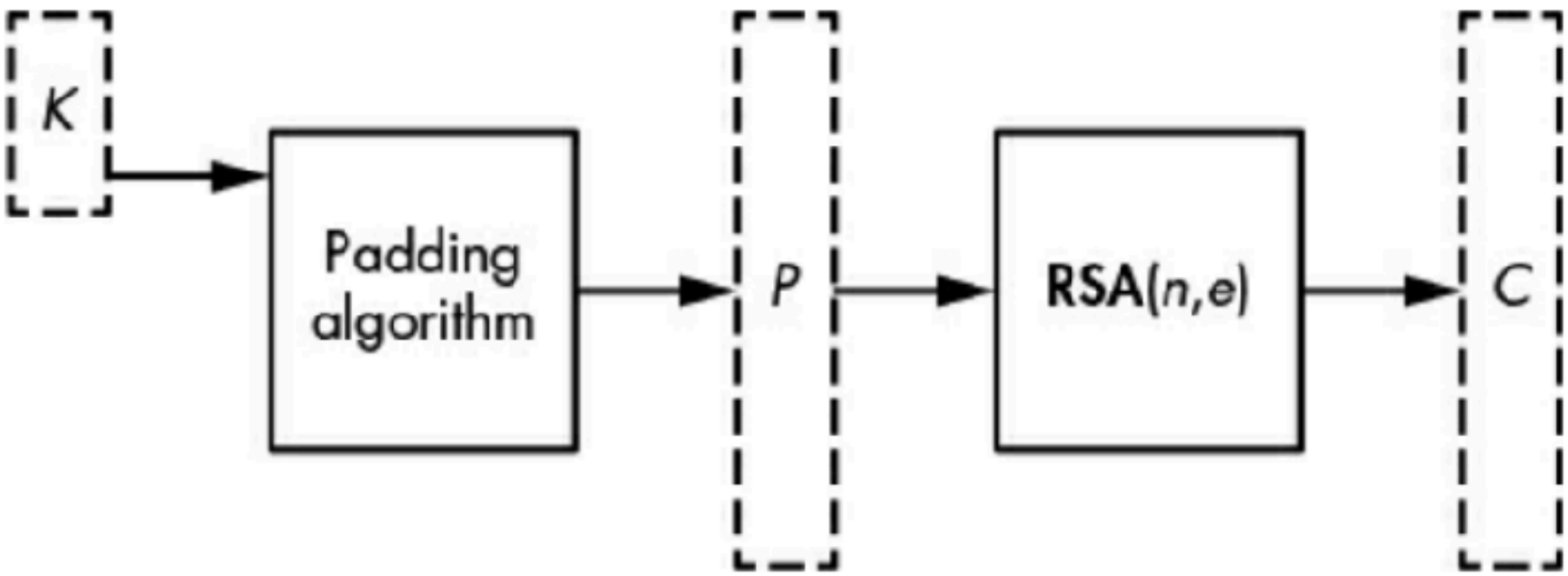
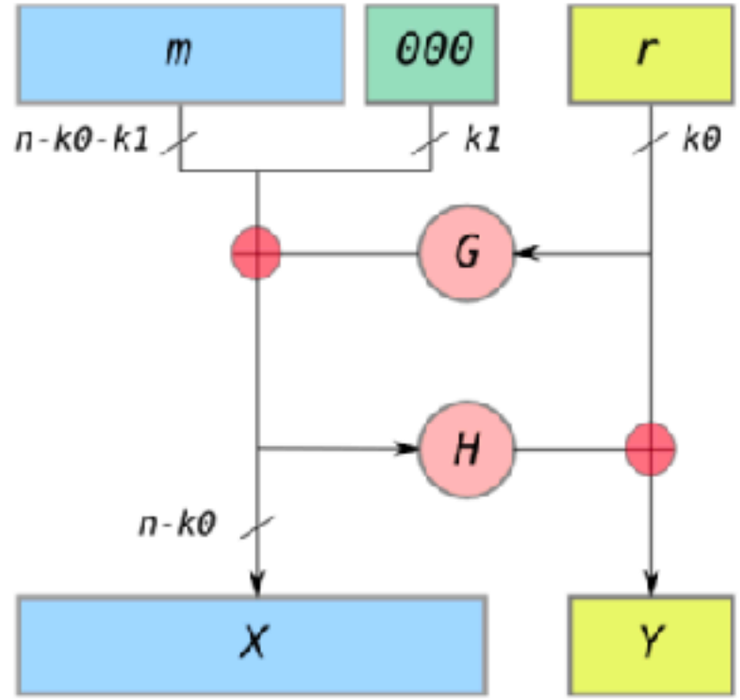


Figure 10-1: Encrypting a symmetric key, K, *with RSA using* (n, e) *as a public key*

Algorithm [edit]

In the diagram,

- n is the number of bits in the RSA modulus.
- k_0 and k_1 are integers fixed by the protocol.
- m is the plaintext message, an $(n - k_0 - k_1)$-bit string
- G and H are random oracles such as cryptographic hash functions.
- $\oplus$ is an xor operation.



OAEP is a Feistel network

To encode,

1. messages are padded with k_1 zeros to be $n - k_0$ bits in length.
2. r is a randomly generated k_0-bit string
3. G expands the k_0 bits of r to $n - k_0$ bits.
4. $X = m00..0 \oplus G(r)$
5. H reduces the $n - k_0$ bits of X to k_0 bits.
6. $Y = r \oplus H(X)$
7. The output is $X \,||\, Y$ where X is shown in the diagram as the leftmost block and Y as the rightmost block.

To decode,

1. recover the random string as $r = Y \oplus H(X)$
2. recover the message as $m00..0 = X \oplus G(r)$

PKCS#1 v1.5

- An old method created by RSA
- Much less secure than OAEP
- Used in many systems

Signing with RSA

Digital Signatures

- Sign a message ***x*** with $y = x^d \bmod n$
- No one can forge the signature because ***d*** is secret
- Everyone can verify the signature using ***e***
- $x = y^e \bmod n$
- Notice that ***x*** is not secret
- Signatures prevent forgeries, they don't provide confidentiality

Signing a Hash

- Signing long messages is slow and uncommon
- Typically the message is hashed
- Then the hash is signed

Textbook RSA Signatures

- Sign a message x with $y = x^d \bmod n$
- Attacker can forge signatures
  - For x =0, 1, or n - 1

Blinding Attack

- You want to get the signature for message ***M***
  - Which the targeted user would never willingly sign
- Find a value ***R*** such that $R^eM \bmod n$
  - Is a message the user will sign
  - That signature ***S*** is $(R^eM)^d \bmod n$
    - = $R^{ed}M^d \bmod n = RM^d \bmod n$
  - The signature we want is $M^d \bmod n = S/R$

The PSS Signature Standard

- Makes signatures more secure, the way OAEP makes encryption more secure
- Combines the message with random and fixed bits

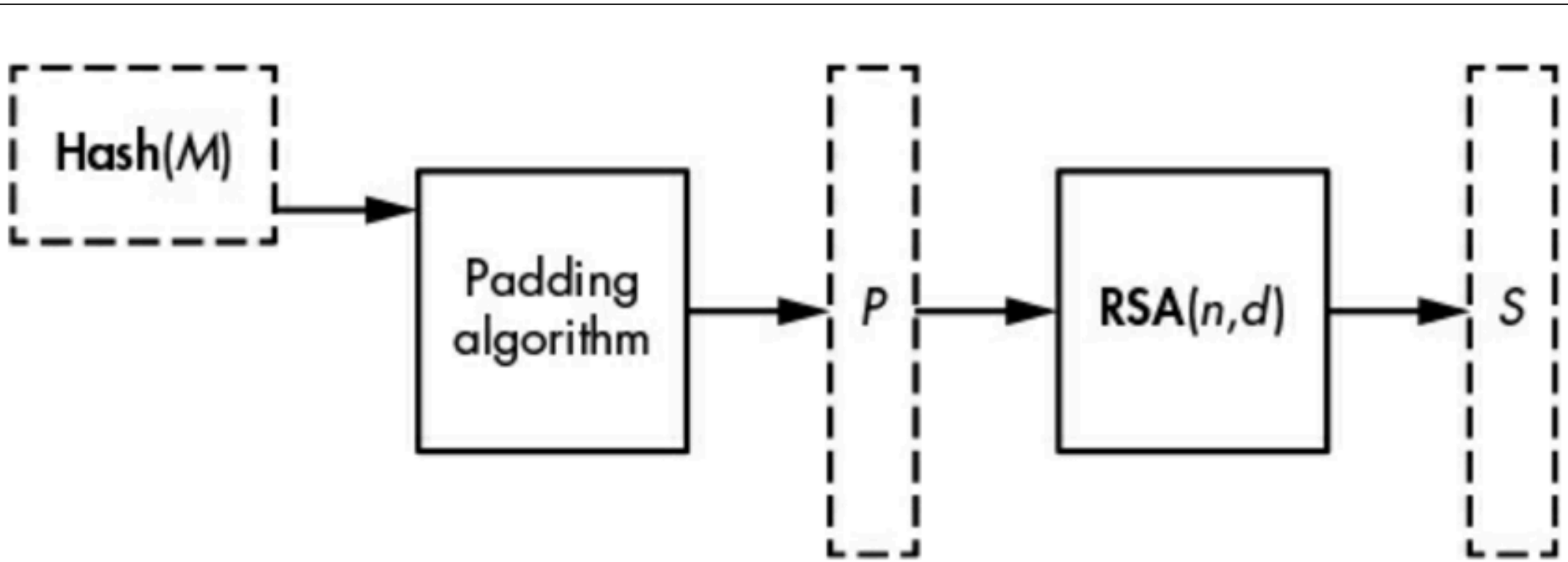


Figure 10-3: Signing a message, M, *with RSA and with the PSS standard, where* (n, d) *is the private key*

Full Domain Hash Signatures (FDH)

- x = Hash(message)
- Signature $y = x^e \bmod n$

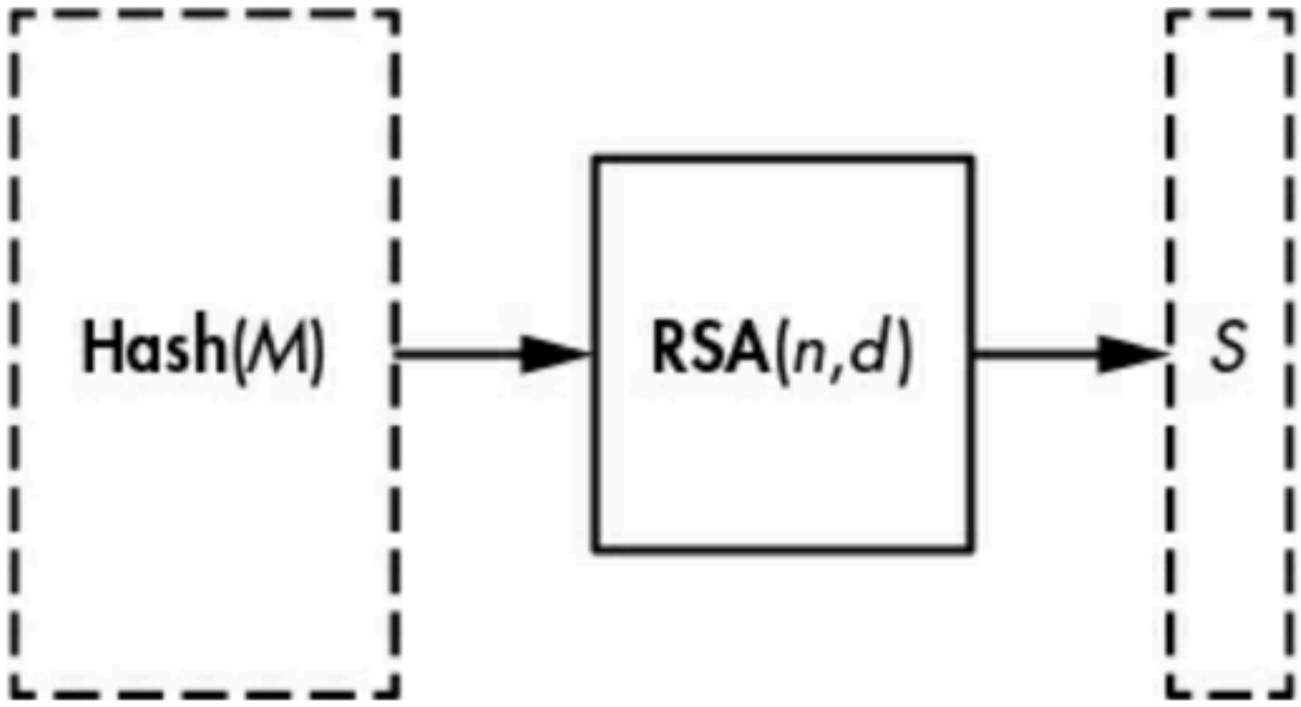


Figure 10-4: Signing a message with RSA using the Full Domain Hash technique

FDH v. PSS

- PSS came later
  - More proof of theoretical security
  - Because of added randomness
- But in practice they are similar in secuity
- But PS is safer against ***fault attacks***

RSA Implementations

Just Use Libraries

- Much easier and safer than writing your own implementation

Square-and-Multiply

$$y = e_{k_{pub}}(x) \equiv x^e \bmod n \quad \text{(encryption)}$$

$$x = d_{k_{pr}}(y) \equiv y^d \bmod n \quad \text{(decryption)}$$

- Consider RSA with a 1024-bit key
- We need to calculate x^e where e is 1024 bits long
- x * x * x * x .... 2^{1024} multiplications
- Competely impossible -- we can't even crack a 72-bit key yet (2^{72} calculations)

Square-and-Multiply

- Use memory to save time
- Do these ten multiplications
  - x2 = x * x
  - x4 = x2 * x2
  - x8 = x4 * x4
  - x16 = x8 * x8
  - ...
  - x1024 = x512 * x512
  - ...
- Combine the results to make any exponent

Square-and-Multiply

- With this trick, a 1024-bit exponent can be calculated with only 1536 multiplications
- But each number being multiplied is 1024 bits long, so it still takes a lot of CPU

Side-Channel Attacks

- The speedup from square-and-multiply means that exponent bits of 1 take more time than exponent bits of 0
- Measuring power consumption or timing can leak out information about the key
- Few libraries are protected from such attacks

Cryptography That Can't Be Hacked

- EverCrypt -- a library immune to timing attacks
  - From Microsoft research
- Link Ch 10b

the purpose of the entire encryption," said Bhargavan. Such "side-channel attacks" were beh the most notorious hacking attacks in recent years, including the Lucky Thirteen attack. The researchers proved that EverCrypt never leaks information in ways that can be exploited by these types of timing attacks.

Small ***e*** for Faster Encryption

- Encryption: $y = x^e \bmod n$
  - Decryption: $x = y^d \bmod n$
- Choosing small ***e*** makes encryption faster, but decryption slower

Chinese Remainder Theorem

- Replaces one operation mod ***n***
  - Encryption: $y = x^e \bmod n$
- With two operations mod ***p*** and ***q***
  - Making RSA four times faster

$$x = x_p \times q \times (1/q \bmod p) + x_q \times p \times (1/p \bmod q) \bmod n$$

How Things Can Go Wrong

Bellecore Attack on RSA-CRT

- ***Fault injection***
  - Alter the power supply
  - or hit chips with a laser pulse
- Can break deterministic schemes like CRT
  - But not ones including randomness like PSS


Kahoot!