Chapter 4: Secrets

Panos Louridas

Athens University of Economics and Business
Real World Algorithms
A Beginners Guide
The MIT Press

Outline

- We call *plaintext* the message that we want to send.
- We call *ciphertext* the message that we want to send, so that only the recipient can be able to read it.
- We call *encryption* the conversion of plaintext to ciphertext.
- Then *decryption* is the recovery of the plaintext from the ciphertext.
- Encryption and decryption are based on the use of a *key*.

Outline

Cryptography in Antiquity

- One of the first encryption methods was the *scytale*. That was a cylinder onto which the message was wrapped.
- Other ancient cryptographic methods were based on substitutions of letters by other letters.
- If we substitute every letter in our message with another letter from our own alphabet, we have a *substitution cipher*.

Caesar Substitution Cipher

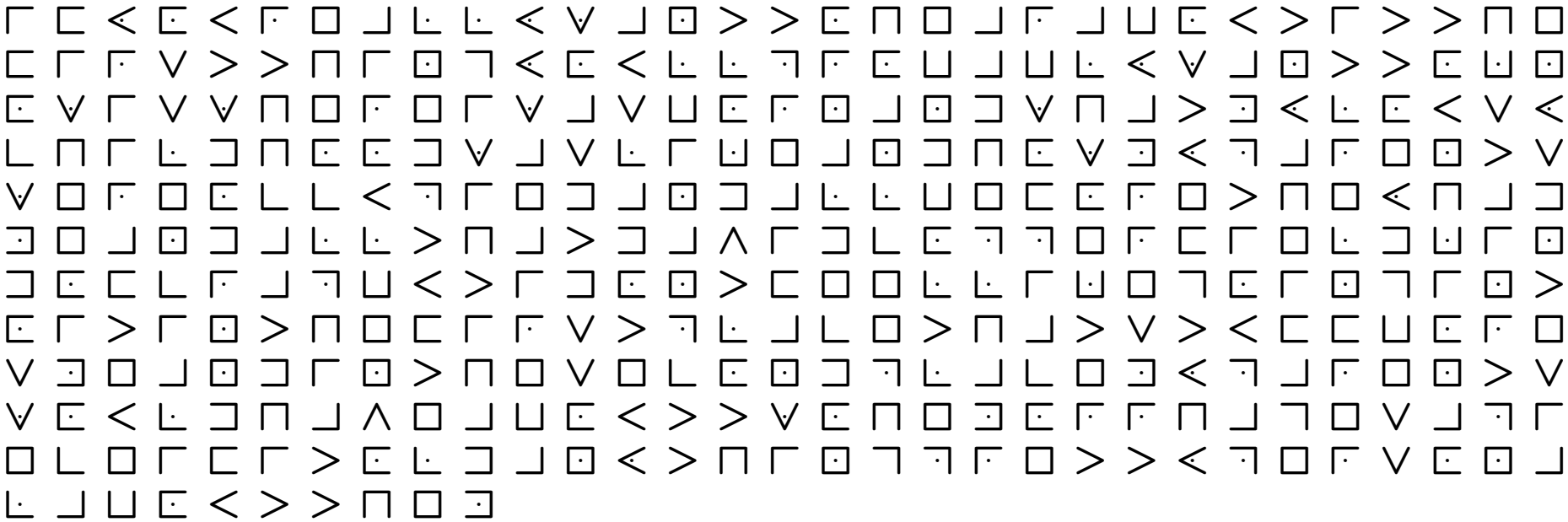
- In the *Caesar cipher*, each letter is substituted by the letter on which we land when we move along the alphabet by a specific number of letters, wrapping around if necessary.
- The number of letters that we use to find the substitute is the key of the cipher.
- If the key is 5, A becomes F, B becomes G, . . . , and Z becomes E.
- Such ciphers are also called *shift ciphers*.

Example of Caesar Cipher

- Suppose we have the phrase “I am seated in an office”.
- With key 5, it is encrypted as: N FR XJFYJI NS FS TKKNHJ.
- To decrypt it we carry out the same process, but we use the letter that is 5 positions before in the alphabet.

Outline

The Message



English Letters Frequencies

- In 1965 Mark Mayzner published English letter frequency tables.
- Using the technology of that time, we was able to process a corpus of 20,000 words.
- On December 17, 2012, he contacted Peter Norvig, Director of Research at Google, asking if Google could use its resources to update those tables.
- Norvig did so, and published the results at `http://norvig.com/mayzner.html`.

English Letters Frequency Table

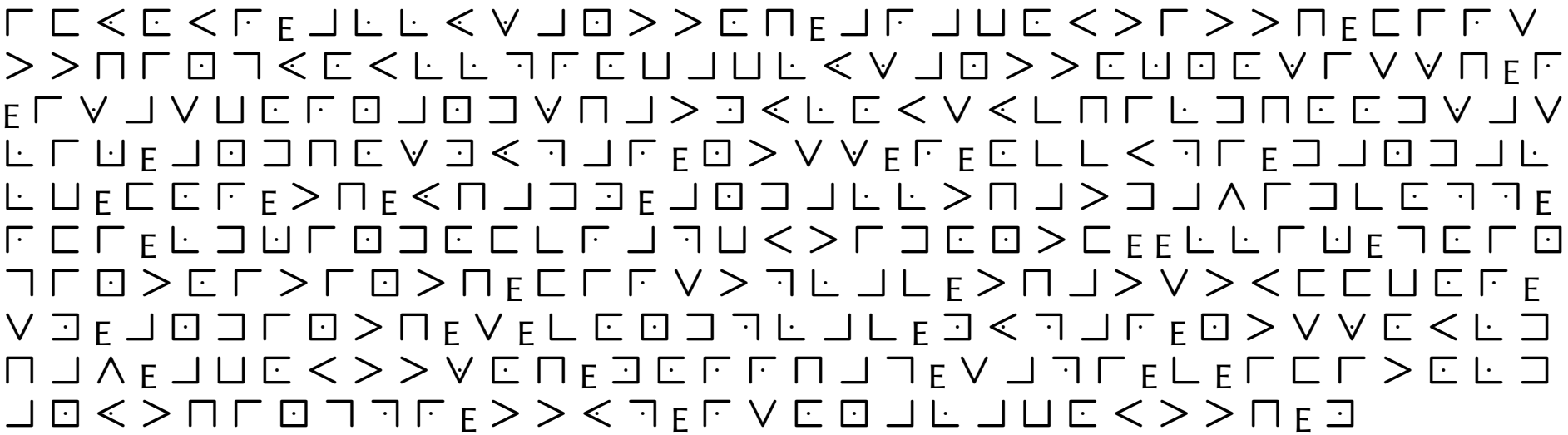
E	445.2	12.49%	M	89.5	2.51%
T	330.5	9.28%	F	85.6	2.40%
A	286.5	8.04%	P	76.1	2.14%
O	272.3	7.64%	G	66.6	1.87%
I	269.7	7.57%	W	59.7	1.68%
N	257.8	7.23%	Y	59.3	1.66%
S	232.1	6.51%	B	52.9	1.48%
R	223.8	6.28%	V	37.5	1.05%
H	180.1	5.05%	K	19.3	0.54%
L	145.0	4.07%	X	8.4	0.23%
D	136.0	3.82%	J	5.7	0.16%
C	119.2	3.34%	Q	4.3	0.12%
U	97.3	2.73%	Z	3.2	0.09%

3,563,505,777,820 letters were counted. The numbers in the table are billions with the corresponding percentages.

Message Decryption (1)

- We count the symbol frequencies in the encrypted message.
- The most common symbol is □ with 35 occurrences.
- The second most common symbol is > with 33 occurrences.
- The third most common symbol is ⌟ with 32 occurrences.

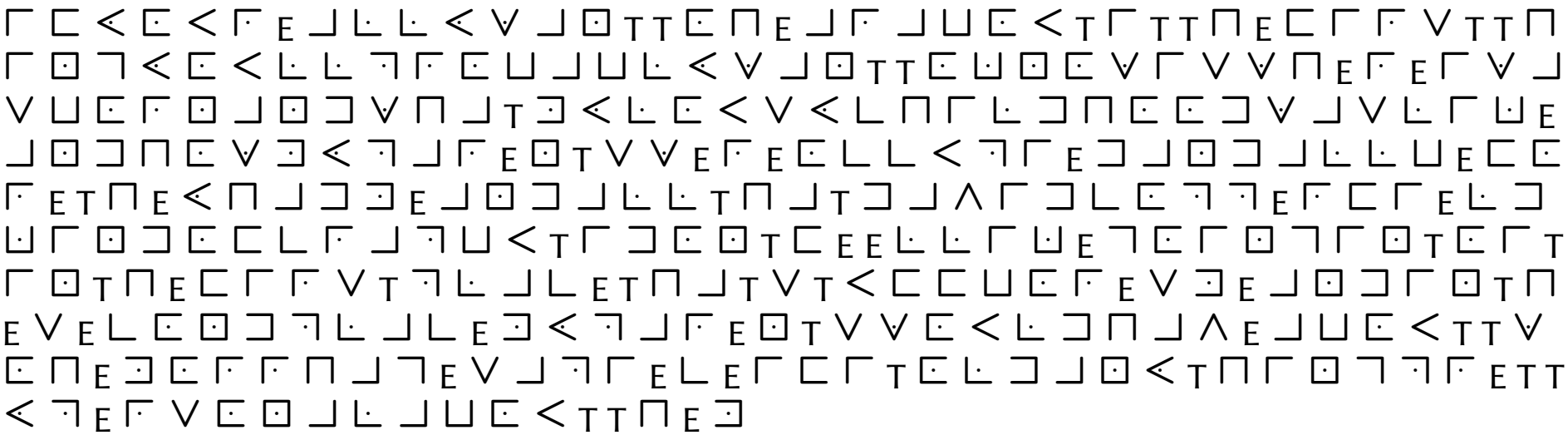
Substituting □ → E



Substituting

□ → E

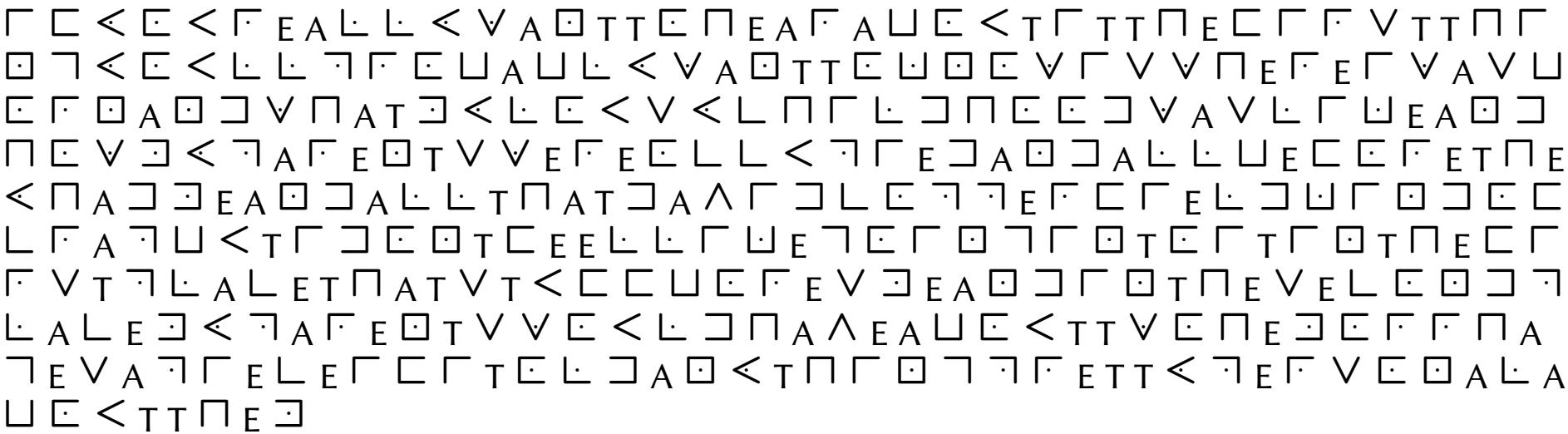
Substituting > → T



Substituting

> → T

Substituting ⌟ → A



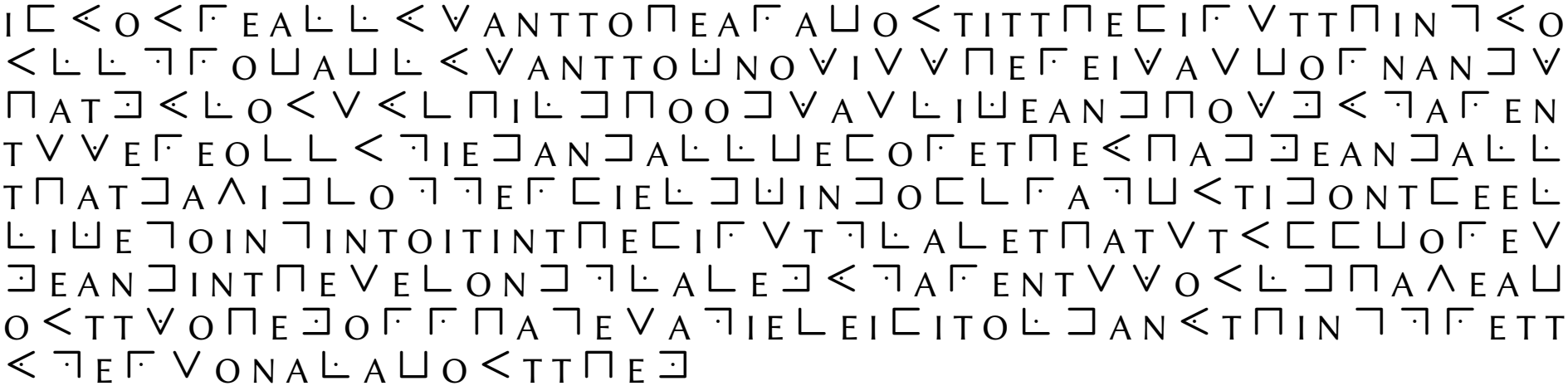
Substituting

⌟ → A

Message Decryption (2)

- Symbol ⊏· has 28 occurrences.
- Symbol ⌜ has 24 occurrences.
- Symbol ⊡ has 22 occurrences.

Substituting ⊏ → O, ⌜ → I, ⊡ → N



Substituting

⊏ → O, ⌜ → I, ⊡ → N

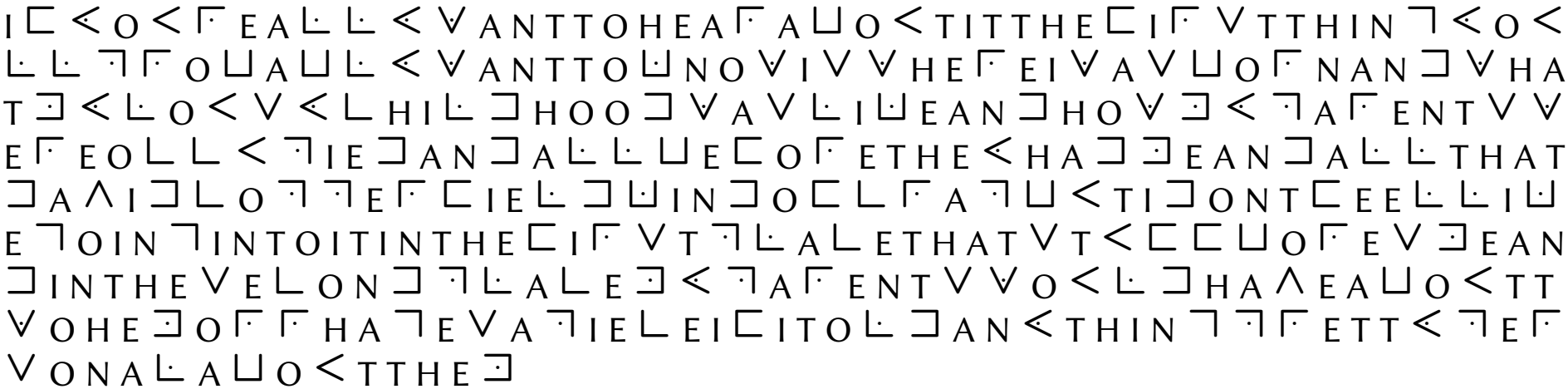
English Bigram Frequencies

TH	100.3	3.56%
HE	86.7	3.07%
IN	68.6	2.43%
ER	57.8	2.05%
AN	56.0	1.99%
RE	52.3	1.85%
ON	49.6	1.76%
AT	41.9	1.49%
EN	41.0	1.45%
ND	38.1	1.35%

Working with Bigrams (1)

- The most common bigram in English is TH.
- In the partially decrypted message, the most common bigram, with 9 occurrences, is T⊓, so we can try substituting ⊓ with H.

Substituting ⊓ → H



Substituting

⊓ → H

Working with Bigrams (2)

- The second most common bigram in the partially decrypted message is ⌜· E with 8 occurrences.
- The RE bigram is also common, so we can try substituting ⌜· with R.

Substituting ⌜ → R

I⊏<O<REA└└<∨ANTTOHEARA⊔O<TITTHE⊏IR∨TTHIN¬<O<
└└¬RO⊔A⊔└<∨ANTTO⊔NO∨I∨∨HEREI∨A∨⊔ORNAN⊐∨HAT
⊐<└O<∨<└HI└⊐HOO⊐∨A∨└I⊔EAN⊐HO∨⊐<¬ARENT∨∨E
REO└└<¬IE⊐AN⊐A└└⊔E⊏ORETHE<HA⊐⊐EAN⊐A└└THAT⊐A
∧I⊐└O¬¬ER⊏IE└⊐⊔IN⊐O⊏└RA¬⊔<TI⊐ONT⊏EE└└I⊔E¬OI
N¬INTOITINTHE⊏IR∨T¬└A└ETHAT∨T<⊏⊏⊔ORE∨⊐EAN⊐INT
HE∨E└ON⊐¬└A└E⊐<¬ARENT∨∨O<└⊐HA∧EA⊔O<TT∨OHE
⊐ORRHA¬E∨A¬IE└EI⊏ITO└⊐AN<THIN¬¬RETT<¬ER∨ONA└A
⊔O<TTHE⊐

Substituting

⌜ → R

Guessing Words (1)

- In the first line we can see the sequence REA𐤋𐤋𐤒.
- Could that be REALLY?

Substituting ⌞· → L, < → Y

I ⊏ YO < REALLY ∀ ANTTOHEARA ⊔ O < TITTHE ⊏ IR V TTHIN ⌝ YO < LL ⌝
RO ⊔ A ⊔ LY ∀ ANTTO ⊔ NO ∀ I V ∀ HEREI ∀ A V ⊔ ORNAN ⊐ ∀ HAT ⊐ YLO <
V Y ⌞ HIL ⊐ HOO ⊐ ∀ A V LI ⊔ EAN ⊐ HO ∀ ⊐ Y ⌝ ARENT V ∀ EREO ⌞ ⌞ < ⌝ I
E ⊐ AN ⊐ ALL ⊔ E ⊏ ORETHEYHA ⊐ ⊐ EAN ⊐ ALLTHAT ⊐ A ∧ I ⊐ ⌞ O ⌝ ⌝ ER
⊏ IEL ⊐ ⊔ IN ⊐ O ⊏ ⌞ RA ⌝ ⊔ < TI ⊐ ONT ⊏ EELLI ⊔ E ⌝ OIN ⌝ INTOITINTHE
⊏ IR V T ⌝ LA ⌞ ETHAT V T < ⊏ ⊏ ⊔ ORE V ⊐ EAN ⊐ INTHE V E ⌞ ON ⊐ ⌝ LA
⌞ E ⊐ Y ⌝ ARENT V ∀ O < L ⊐ HA ∧ EA ⊔ O < TT ∀ OHE ⊐ ORRHA ⌝ E V A ⌝ IE
⌞ EI ⊏ ITOL ⊐ ANYTHIN ⌝ ⌝ RETTY ⌝ ER V ONALA ⊔ O < TTHE ⊐

Substituting

⌞· → L, < → Y

Guessing Words (2)

- In the first two lines we can see two YO$<$, which could be YOU.
- In the last line we can see ANYTHIN$\urcorner$, which could be ANYTHING.
- Also in the last line, $\urcorner$RETTY is probably PRETTY.
- Going on in this way we can decrypt the message in its entirety.

The Decrypted Message

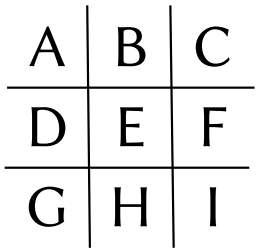
IFYOUREALLYWANTTOHEARABOUTITTHEFIRSTTHINGYOULLPROBA
BLYWANTTOKNOWISWHEREIWASBORNANDWHATMYLOUSYCHILDH
OODWASLIKEANDHOWMYPARENTSWEREOCCUPIEDANDALLBEFORE
THEYHADMEANDALLTHATDAVIDCOPPERFIELDKINDOFCRAPBUTIDO
NTFEELLIKEGOINGINTOITINTHEFIRSTPLACETHATSTUFFBORESMEA
NDINTHESECONDPLACEMYPARENTSWOULDHAVEABOUTTWOHEMO
RRHAGESAPIECEIFITOLDANYTHINGPRETTYPERSONALABOUTTHEM

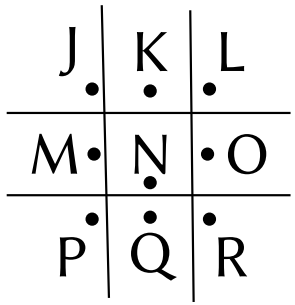
With Punctuation

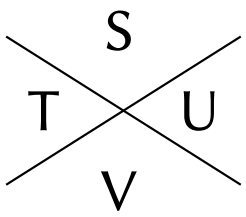
If you really want to hear about it, the first thing you'll probably want to know is where I was born, and what my lousy childhood was like, and how my parents were occupied and all before they had me, and all that David Copperfield kind of crap, but I don't feel like going into it. In the first place, that stuff bores me, and in the second place, my parents would have about two hemorrhages apiece if I told anything pretty personal about them.

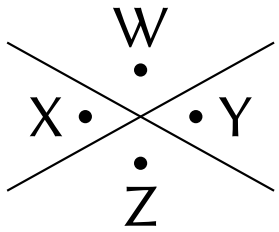
J. D. Salinger, "The Catcher in the Rye".

The Pigpen Cipher









Outline

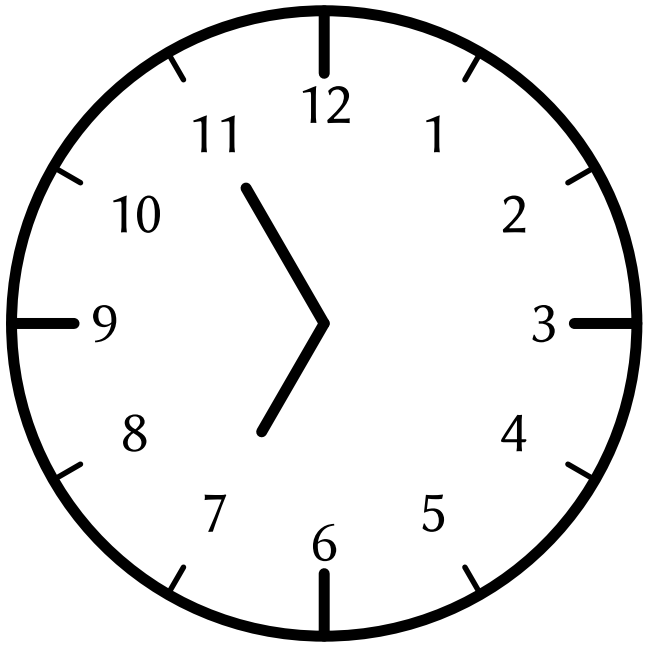
Operation

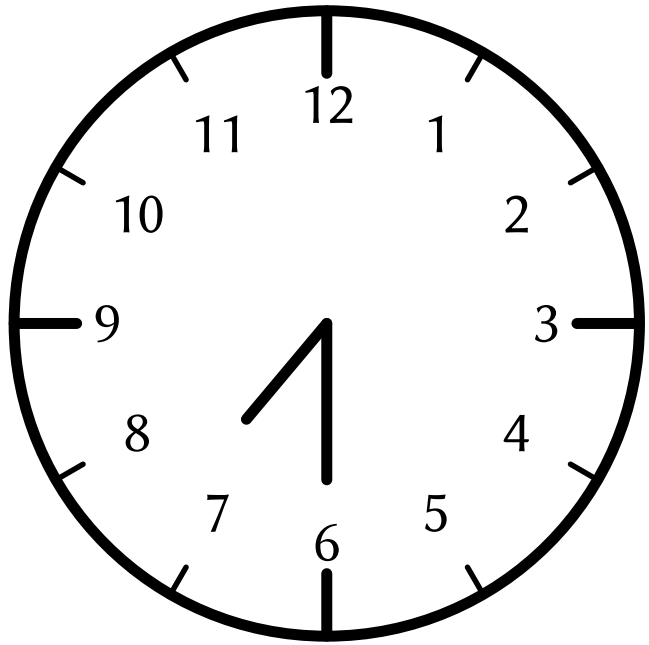
- In the one-time pad, we use a *random* string, equal in length with the message we want to encrypt. The random string is really the one-time pad.
- We take in order one character from the one-time pad and one character from our message.
- We add the two characters and take the remainder of the addition when divided by the number of characters in the alphabet.

Modular Arithmetic

- Modular arithmetic is like adding minutes in an hour.
- The symbol for modular division is mod (modulo), so we have $23 \bmod 5 = 3$ as the remainder of the division of 23 by 5 is 3.

Adding Minutes





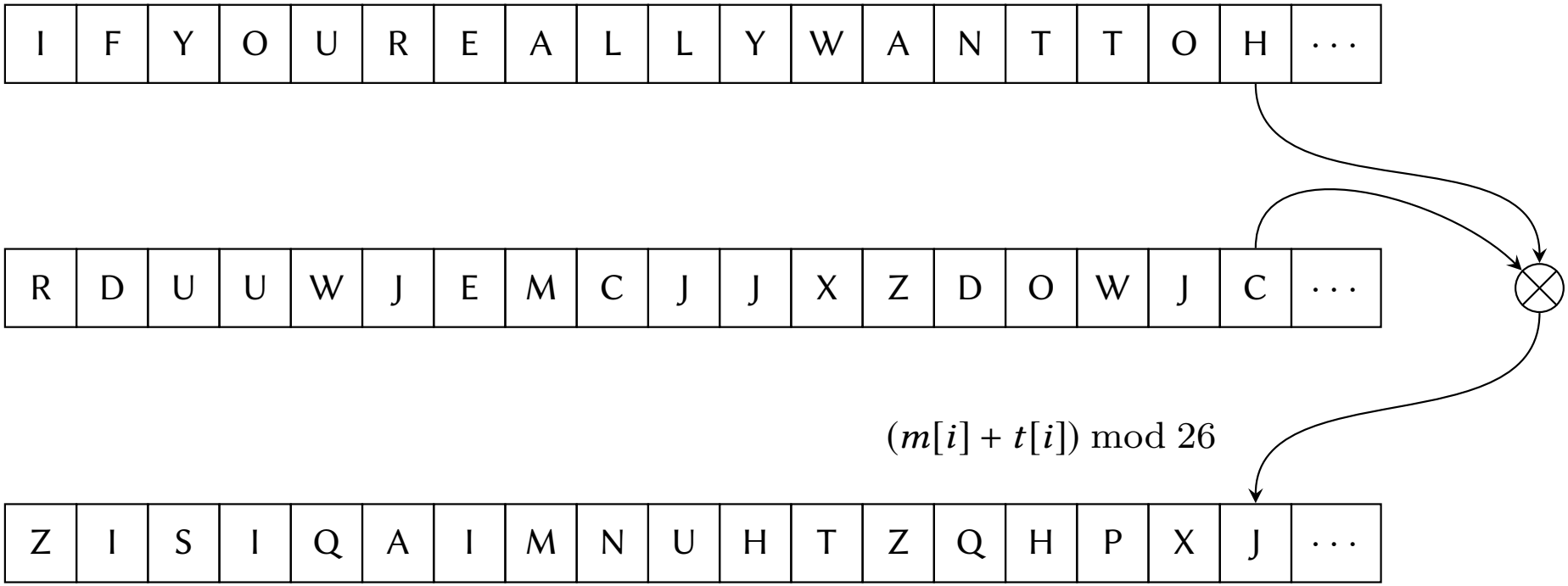
Encryption and Decryption

- We add the ith letter of the message, $m[i]$, modulo 26 (there are 26 letters in the English alphabet) with the ith letter of the one-time pad.
- Letters correspond to numbers, starting from zero, that is $\mathrm{A} = 0$, $\mathrm{B} = 1$, etc.
- So we have $c[i] = (m[i] + t[i]) \bmod 26$.
- Decryption is $m[i] = (c[i] - t[i]) \bmod 26$.

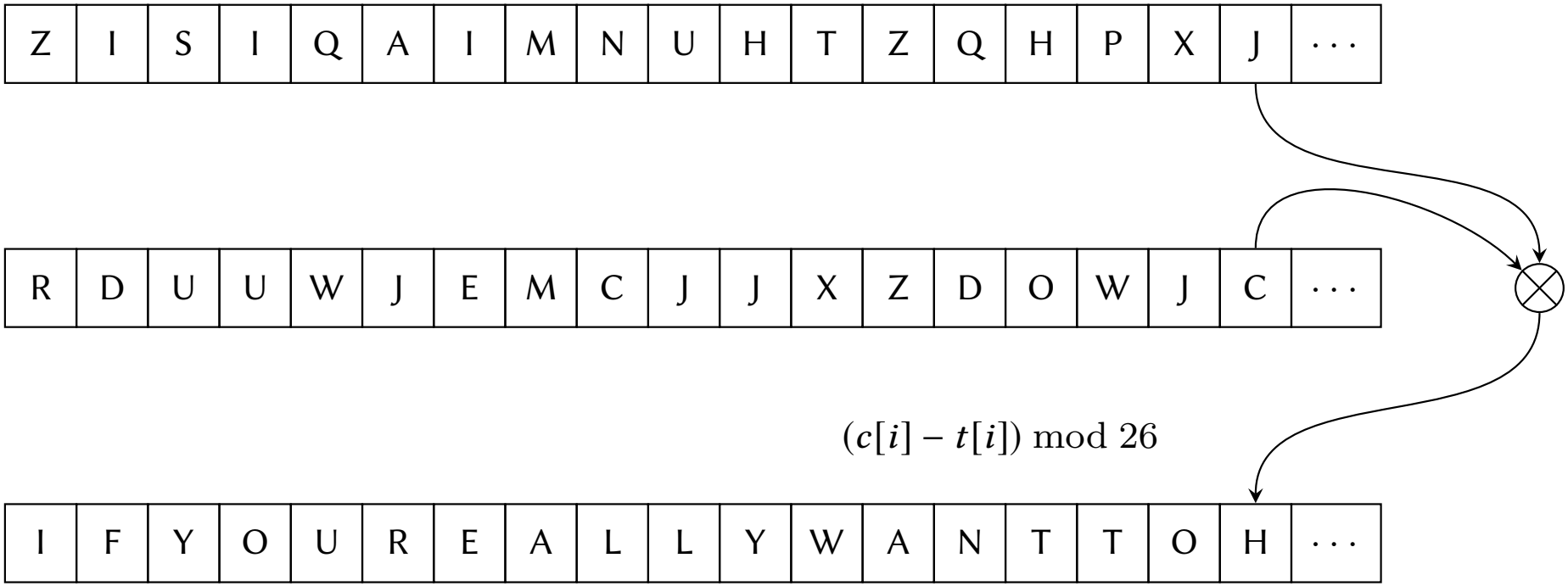
Security of the One-time Pad

- The one-time pad is absolutely secure and cannot be broken.
- The only way to decrypt a message correctly is with possession of the one-time pad.
- Without the one-time pad, each decryption is equally probable!

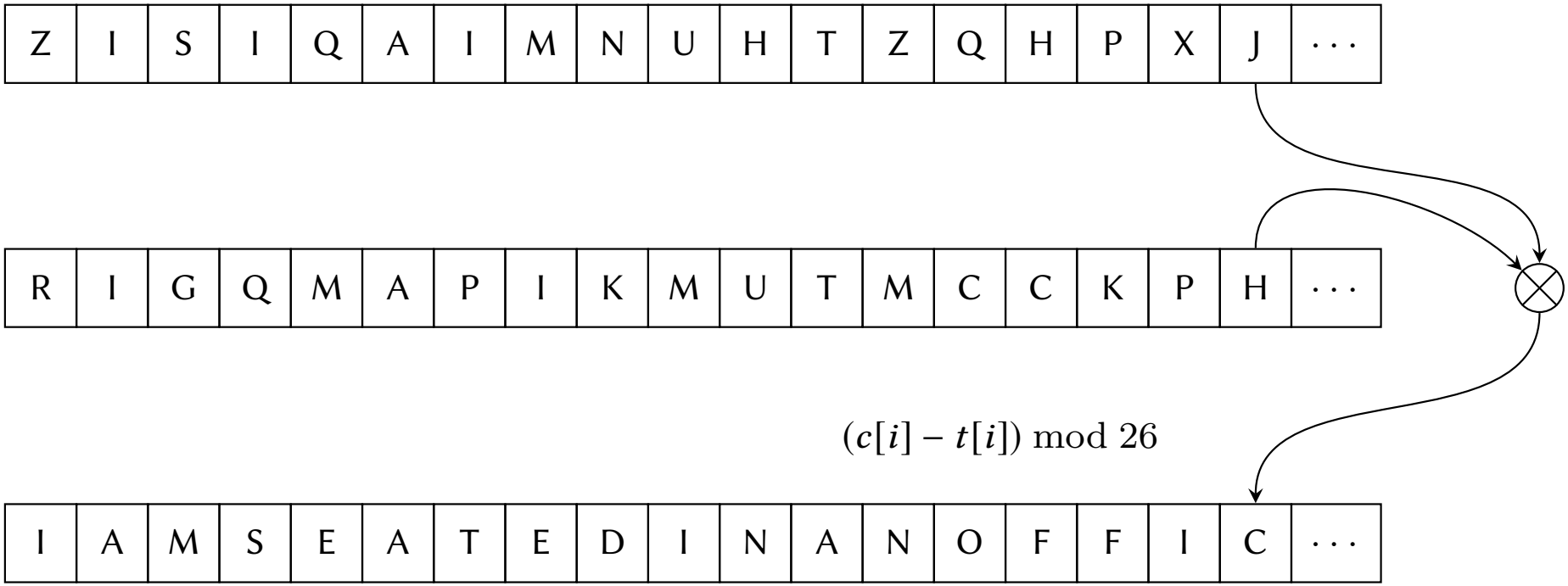
Example


I F Y O U R E A L L Y W A N T T O H ···
R D U U W J E M C J J X Z D O W J C ···
$(m[i] + t[i]) \bmod 26$
Z I S I Q A I M N U H T Z Q H P X J ···

Correct Decryption



Wrong Decryption



Exclusive Or, XOR

		x	
		0	1
y	0	0	1
	1	1	0

- Exclusive or, XOR, is a binary operation whose result is 1 if the two operands are different, 0 if they are the same.
- Its symbol is $\oplus$.
- We have $1 \oplus 1 = 0 \oplus 0 = 0$, $1 \oplus 0 = 0 \oplus 1 = 1$.
- XOR is reversible: if $c = a \oplus b$, then $c \oplus b = a$.

One-time pad Using XOR

- Each character is represented by a binary number, e.g., A is 1100001 using the ASCII code.
- The one-time pad is a random sequence, such as 1101011....
- The ciphertext results by taking the XOR of the plaintext with the one-time pad, bit by bit.
- In our case, $1100001 \oplus 1101011 = 0001010$.
- To decrypt we XOR the ciphertext with the one-time pad: $0001010 \oplus 1101011 = 1100001$.

Problems of the One-time Pad

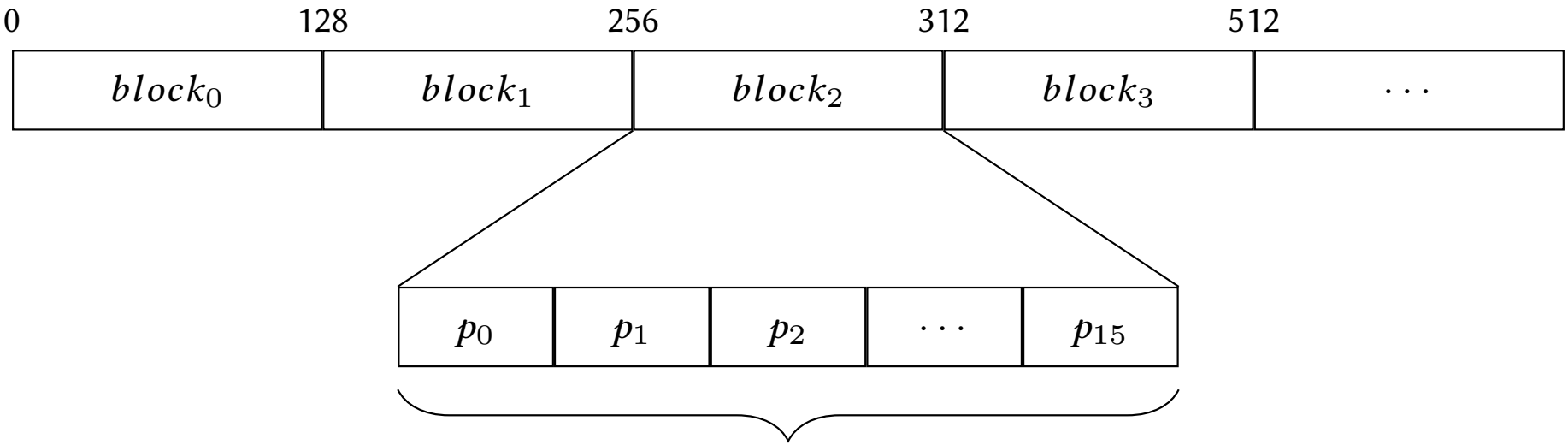
- One-time pads are difficult to use in practice.
- The one-time pad must be *completely random*.
- It must be *unique*.
- It must be at least as long as the plaintext.

Outline

AES Development

- AES is a cryptographic standard developed by the U.S. National Institute of Standards and Technology (NIST) in 2001.
- It was selected after an open process to change an older standard, the Data Encryption Standard (DES).
- The selection process took place from 1997 until 2000.
- NIST had invited the cryptographic community to submit proposals.
- On October 2, 2000 NIST announced that the winner was the proposal submitted by two Belgian cryptographers, Joan Daemen and Vincent Rijmen. The system was called Rijndael.

AES: Creation of the State



p_0	p_4	p_8	p_{12}
p_1	p_5	p_9	p_{13}
p_2	p_6	p_{10}	p_{14}
p_3	p_7	p_{11}	p_{15}

AES: AddRoundKey

p_0	p_4	p_8	p_{12}
p_1	p_5	p_9	p_{13}
p_2	p_6	p_{10}	p_{14}
p_3	p_7	p_{11}	p_{15}

k_0	k_4	k_8	k_{12}
k_1	k_5	k_9	k_{13}
k_2	k_6	k_{10}	k_{14}
k_3	k_7	k_{11}	k_{15}

$\oplus$

x_0	x_4	x_8	x_{12}
x_1	x_5	x_9	x_{13}
x_2	x_6	x_{10}	x_{14}
x_3	x_7	x_{11}	x_{15}

AES: S-Box

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

AES: SubBytes

x_0	x_4	x_8	x_{12}
x_1	x_5	x_9	x_{13}
x_2	x_6	x_{10}	x_{14}
x_3	x_7	x_{11}	x_{15}

$s_{0,0}$	$s_{0,1}$	$\cdots$	$s_{0,F}$
$s_{1,0}$	$s_{1,1}$	$\cdots$	$s_{1,F}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$s_{F,0}$	$s_{F,1}$	$\cdots$	$s_{F,F}$

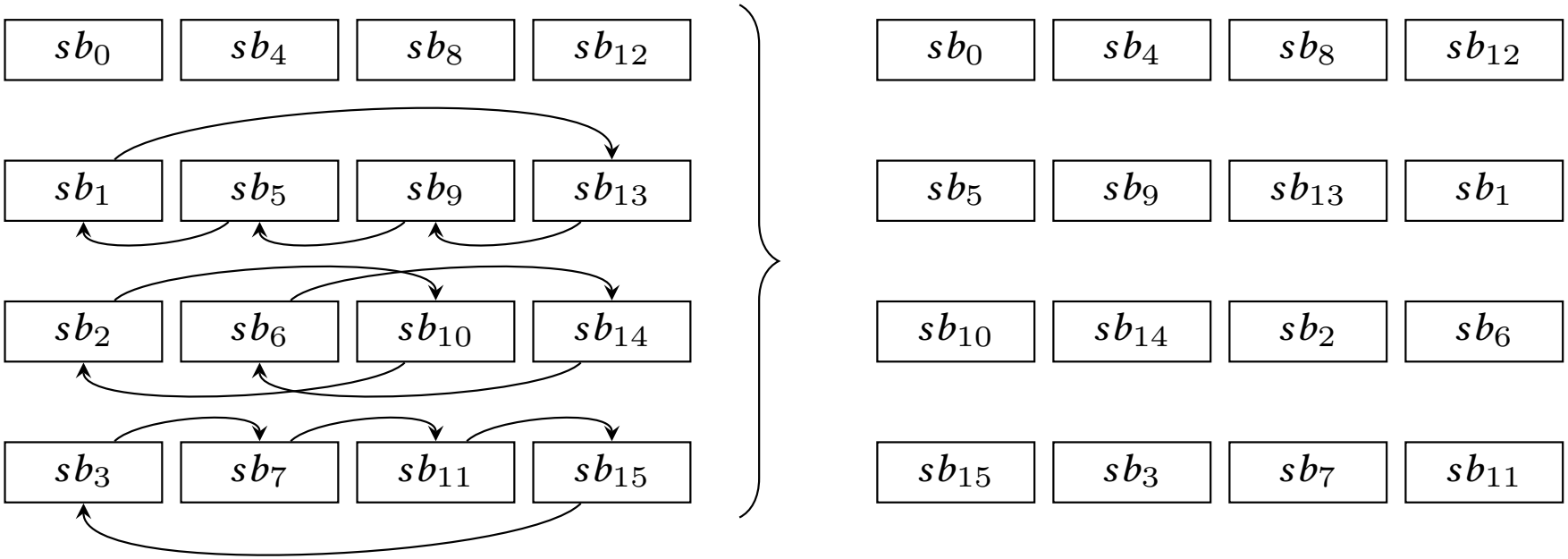
$$x_i = h_1 h_2 \rightarrow sb_i = s_{h_1,h_2}$$

sb_0	sb_4	sb_8	sb_{12}
sb_1	sb_5	sb_9	sb_{13}
sb_2	sb_6	sb_{10}	sb_{14}
sb_3	sb_7	sb_{11}	sb_{15}

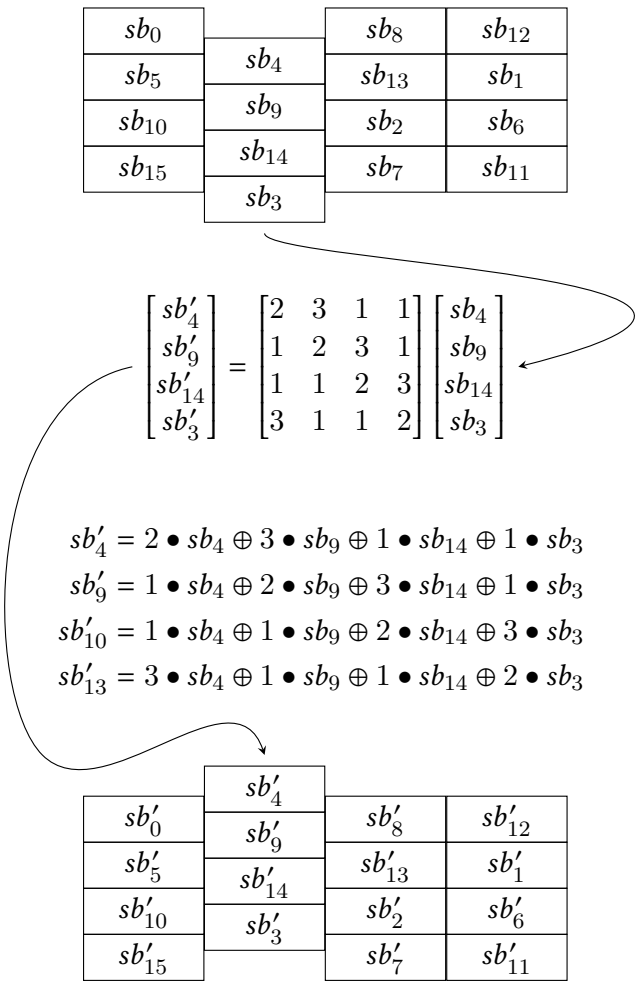
Example: SubBytes

- Suppose we have $x_4 = 168$.
- 168 in decimal is A8 in hexadecimal.
- We go to line A, column 8 of the S-box, where we find the number C2 in hexadecimal, or 194 in decimal.
- So $sb_4 = 194$.

AES: ShiftRows



AES: MixColumns



AES: MixColumns

- The matrix is the same for all columns.
- Addition and multiplication are not the usual arithmetic operations.
- They are addition and multiplication of polynomials modulo an irreducible polynomial of degree 8 in the finite field $GF(2^8)$.

AES: MixColumns

- Addition $\oplus$ is just an XOR of the operand bits.
- For multiplication, we need to see what it means to multiply by 1, 2, and 3 (as they are the only multiplications that take place).
- We have:

$$1 \bullet a = a$$
$$3 \bullet a = 2 \bullet a \oplus a$$

- So we only need to see how to multiply by 2, $2 \bullet a$.

AES: MixColumns

- If a in binary is written as: $a = (a_7, a_6, \ldots, a_0)$
- Then we have:

$$2 \bullet a = \begin{cases} (a_6, \ldots, a_0, 0) & \text{if } a_7 = 0 \\ (a_6, \ldots, a_0, 0) \oplus (0, 0, 0, 1, 1, 0, 1, 1) & \text{if } a_7 = 1 \end{cases}$$

The AES Algorithm

Algorithm: AES cipher algorithm.

AESCipher$(b, k, n) \rightarrow s$

Input: b, a block of 16 bytes
k, the encryption key
n, the number of rounds

Data: s, the state
rk, an array of size $n + 1$ that will contain the round keys

Output: s, the ciphertext corresponding to b

```
1  s ← CreateState(b)
2  rk ← ExpandKey(k)
3  s ← AddRoundKey(s, rk[0])
4  for i ← 1 to n do
5      s ← SubBytes(s)
6      s ← ShiftRows(s)
7      s ← MixColumns(s)
8      s ← AddRoundKey(s, rk[i])
9  s ← SubBytes(s)
10 s ← ShiftRows(s)
11 s ← AddRoundKey(s, rk[n])
12 return s
```

The AES Decipher Algorithm

Algorithm: AES decipher algorithm.

AESDecipher$(b, k, n) \rightarrow s$

Input: b, a block of 16 bytes
k, the encryption key
n, the number of rounds

Output: s, the plaintext corresponding to b

```
1  s ← CreateState(b)
2  rk ← ExpandKey(k)
3  s ← AddRoundKey(s, rk[n])
4  for i ← 1 to n do
5      s ← InvShiftRows(s)
6      s ← InvSubBytes(s)
7      s ← AddRoundKey(s, rk[n − i])
8      s ← InvMixColumns(s)
9  s ← InvShiftRows(s)
10 s ← InvSubBytes(s)
11 s ← AddRoundKey(s, rk[0])
12 return s
```

AES: Inverse S-Box

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Outline

The Problem

- AES, like other modern cryptographic methods, is secure as long as the key is not leaked.
- In particular, all security hinges on the secrecy of the key (Kerckhoffs's principle, 1883).
- How can we exchange keys securely?

Example (1)

1. Alice and Bob agree on two numbers, a prime number p, and another number, not necessarily prime, g such that $2 \leq g \leq p-2$. Suppose that $p = 23$ and $g = 14$. These two numbers are not required to be kept secret.
2. Alice chooses a secret number a, $1 \leq a \leq p-1$. Suppose she chooses $a = 3$. She calculates the number

$$A = g^a \bmod p$$

   or $14^3 \bmod 23 = 2744 \bmod 23 = 7$.
3. Alice sends A, that is, 7, to Bob.

Example (2)

4. Bob chooses a secret number b, $1 \leq b \leq p - 1$. Suppose he chooses $b = 4$. He carries out the same calculations with Alice, that is,

$$B = g^b \bmod p$$

or $14^4 \bmod 23 = 38\,416 \bmod 23 = 6$.

5. Bob sends B, that is, 6, to Alice.

Example (3)

6. Alice calculates the number

$$B^a \bmod p$$

that is, $6^3 \bmod 23 = 216 \bmod 23 = 9$.

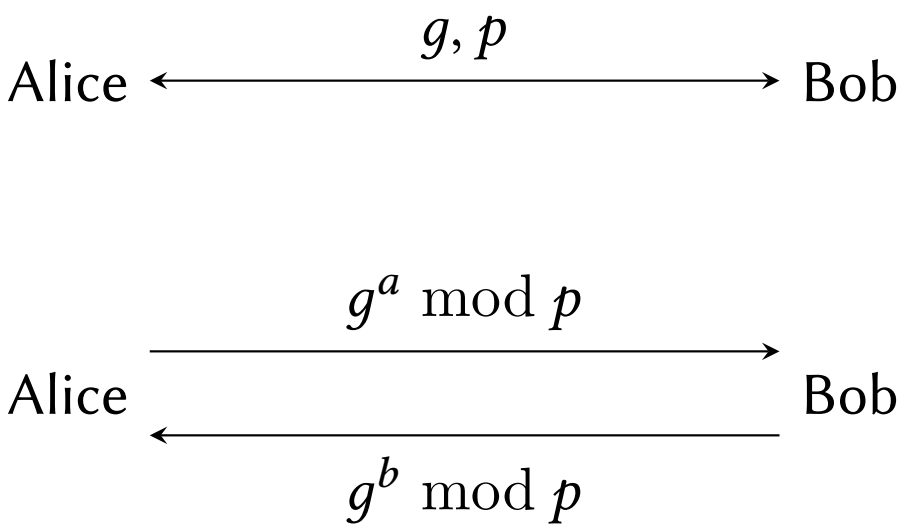
7. Bob calculates the number

$$A^b \bmod p$$

or $7^4 \bmod 23 = 2401 \bmod 23 = 9$.

8. The number 9 is the secret key shared by Alice and Bob.

Diffie-Hellman Communication



Diffie-Hellman Key Exchange

Alice	Bob
Alice and Bob agree on p and g	
Choose a Calculate $A = g^a \bmod p$ Send A to Bob	Choose b Calculate $B = g^b \bmod p$ Send B to Alice
Calculate $s = B^a \bmod p$ $= (g^b)^a \bmod p$ $= g^{ba} \bmod p$	Calculate $s = A^b \bmod p$ $= (g^a)^b \bmod p$ $= g^{ab} \bmod p$

Diffie-Hellman Security

- There is no known efficient way to find the secret from p, g, A, and B.
- That is because to do that we would need to solve the *discrete logarithm problem*, for which we have no efficient solution.
- If p is prime, and we have g and $y = g^x \mod p$, the discrete logarithm problem is finding x, $1 \leq x \leq p - 1$.
- The integer x is called *discrete logarithm of y with base g* and we write $x = \log_g y \mod p$.

One-way Functions

- The calculation $y = g^x \mod p$ is a *one-way function*.
- It is easy to calculate y from g, x, and p.
- However, we do not know an efficient method to calculate x from y, g, and p.
- We can try different values for x until we find the right one—but that is not efficient.

Behavior of g^x and $g^x \bmod p$ (1)

For $g = 2$ and $p = 13$ we have:

x	1	2	3	4	5	6	7	8	9	10	11	12
g^x	2	4	8	16	32	64	128	256	512	1024	2048	4096
$g^x \bmod p$	2	4	8	3	6	12	11	9	5	10	7	1

- While the behavior of 2^x is predictable, the behavior of $2^x \bmod 13$ does not appear to be.
- $2^x \bmod 13$ will take all values from 1 up to 12 before starting repeating itself.

Behavior of g^x and $g^x \bmod p$ (2)

- Indeed:

$$\begin{aligned} 2^{13} \bmod 13 &= (2^{12} \times 2) \bmod 13 \\ &= ((2^{12} \bmod 13) \times (2 \bmod 13)) \bmod 13 \\ &= (1 \times 2) \bmod 13 = 2 \end{aligned}$$

- In general, we can see that the function $2^x \bmod 13$ is periodic with period 12:

$$\begin{aligned} 2^{12+k} \bmod 13 &= ((2^{12} \bmod 13) \times (2^k \bmod 13)) \bmod 13 \\ &= (1 \times 2^k) \bmod 13 \\ &= 2^k \bmod 13 \end{aligned}$$

- 12 is its *fundamental period*, as there is no period smaller than that.

Behavior of g^x and $g^x \bmod p$ (3)

For $g = 3$ and $p = 13$ we have:

x	1	2	3	4	5	6	7	8	9	10	11
g^x	3	9	27	81	243	729	2187	6561	19683	59049	177147
$g^x \bmod p$	3	9	1	3	9	1	3	9	1	3	9

- We see that $3^x \bmod 13$ does not take all values from 1 up to 12 before starting repeating itself. Its fundamental period is 3.
- Therefore if we were to try and find x with $g = 3$ and $p = 13$ we would only need to try 3 different values.
- In general, we want to have g and p such that the fundamental period is very large.

Diffie-Hellman Parameters

- For p we select a large prime number.
- If it takes 4096 bits, it has at least 1233 decimal digits.
- Then we select g according to number theory, so that the fundamental period of $g^x \mod p$ will be very large.

Groups and Generators

- If the successive values of $g^x \mod p$ cover all the numbers from 1 up to and including $p - 1$, then we say that g is a *generator* or a *primitive element*.
- To be more precise, it is called a *group generator*, or a *group primitive element* because the numbers $1, 2, \ldots, p - 1$ when p is a prime form a *multiplicative group* when we multiply them modulo p, an important concept in algebra and number theory.
- We therefore need to pick as g a generator.
- In fact, we can make do without a generator, if the successive values of $g^x \mod p$ are a large enough subset of the numbers $1, 2, \ldots, p - 1$ so that trying to find the solution to the discrete logarithm problem is impractical.

Outline

General

- Raising to a power is an common operation in cryptography.
- The common way to calculate g^x requires x multiplications.
- Is there a better way?

Fast Exponentiation (1)

- Suppose we want to calculate g^x.
- We write x in binary:

$$x = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_0 2^0$$

- That means that:

$$g^x = g^{b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_0 2^0}$$

- Which is equivalent to:

$$g^x = (g^{2^{n-1}})^{b_{n-1}} \times (g^{2^{n-2}})^{b_{n-2}} \times \cdots \times (g^{2^0})^{b_0}$$

Fast Exponentiation (2)

- If we take the last expression from right to left, we start by calculating $(g^{2^0})^{b_0}$.
- Then we calculate:

$$(g^{2^1})^{b_1}$$

$$(g^{2^2})^{b_2}$$

$$(g^{2^3})^{b_3}$$

etc.

Fast Exponentiation (3)

- But:
  $g^{2^0} = g^1 = g$
  g^{2^1} is the square of g
  g^{2^2} is the square of g^{2^1}
  g^{2^3} is the square of g^{2^2}
  and in general $g^{2^k} = (g^{2^{k-1}})^2$ as $(g^{2^{k-1}})^2 = g^{2 \cdot 2^{k-1}}$.
- So we can calculate g^{2^i} for $i = 1, \ldots, n-1$, from the right to the left, squaring the previous factor.
- This is the idea behind the *exponentiation with repeated squaring* algorithm.
- The number of iterations is equal to the bits of the exponent x, so $O(\lg x)$.

Exponentiation by Repeated Squaring

Algorithm: Exponentiation by repeated squaring.

ExpRepeatedSquaring$(g, x) \rightarrow r$

Input: g, the integer base
x, the integer exponent

Output: r, equal to g^x

1 $c \leftarrow g$
2 $d \leftarrow x$
3 $r \leftarrow 1$
4 **while** $d > 0$ **do**
5 **if** $d \bmod 2 = 1$ **then**
6 $r \leftarrow r \times c$
7 $d \leftarrow \lfloor d/2 \rfloor$
8 $c \leftarrow c \times c$
9 **return** r

Exponentiation by Repeated Squaring for 13^{13}

$c = g^{2^i} = 13^{2^i}$	r	d
13	1	1101
169	13	110
28561	13	11
815730721	371293	1
	302875106592253	

- Every line, except for the last, corresponds to the values of c, r, and d, in line 5 of the algorithm.
- From that algorithm we can arrive immediately to an algorithm for *modular exponentiation by repeated squaring*, simply by taking the modulo, when required.
- The number of iterations is again equal to the bits of the exponent, that is, $O(\lg x)$.

Implementation Details (1)

- To find whether an integer has remainder 1 when divided by 2, as in line 5 of the algorithm, we do not need to carry out the division.
- We only need to check whether the last bit is equal to 1.
- To do that, we can use the binary operation AND (its symbol is &), using a number with all bits set to 0, apart from the last bit, set to 1.
- For example, to check 13, we do:

$$1111\&0001 = 0001$$

Implementation Details (2)

- Similarly, to perform an integer division by 2, as in line 7 of the algorithm, we do not need to carry out the division.
- We just need to *shift* the bits of the number one position to the right.
- This operation, *shift right*, has as its symbol $\gg$.
- For example, for 13, we do:

$$1101 \gg 1 = 110$$

Modular Exponentiation with Repeated Squaring

Algorithm: Modular exponentiation by repeated squaring.

`ModExpRepeatedSquaring`$(g, x, p) \rightarrow r$

Input: g, the integer base
x, the integer exponent
p, the divisor

Output: r, equal to $g^x \bmod p$

```
1  c ← g mod p
2  d ← x
3  r ← 1
4  while d > 0 do
5      if d mod 2 = 1 then
6          r ← (r × c) mod p
7      d ← ⌊d/2⌋
8      c ← (c × c) mod p
9  return r
```

Example for $155^{235} \bmod 391$

$c = g^{2^i} = 155^{2^i} \bmod 391$	r	d
155	1	11101011
174	155	1110101
169	382	111010
18	382	11101
324	292	1110
188	292	111
154	42	11
256	232	1
	314	

A Deeper Look in Complexity

- We saw that the algorithms require $O(\lg x)$ iterations.
- But how much time does each iteration require?
- Multiplication and squaring make take some time to complete.
- Can we investigate that?

Single- and Multiple-precision Arithmetic

- Computers use a fixed number of bits to represent an integer, say 32 or 64 bits.
- Operations involving such numbers are carried out very fast and are called *single-precision* operations; arithmetic with these operations is called *single-precision arithmetic*.
- If our numbers cannot fit in the number of bits offered by the computer, then the computer has to use *multiple-precision arithmetic*, also called *arbitrary-precision arithmetic*, or *bignum arithmetic*.

Multiplication in Multiple-precision Arithmetic

- Long multiplication involves n^2 multiplications, plus additions.
- Multiple-precision multiplication of two numbers a and b of n and m bits with an improved algorithm requires nm single-precision multiplications, so it is $O(nm) = O(\lg a \lg b)$.
- Squaring can be performed at $O((n^2 + n)/2)$, for a number of n bits.

Multiple- and Single-precision Complexity

- If we investigate the algorithm for exponentiation by repeated squaring, we find that its overall complexity is $O((x \lg g)^2)$, if we take into account the time required for multiple-precision arithmetic.
- Therefore, our complexity estimate is $O(\lg x)$ if we don't care about multiple-precision arithmetic, or $O((x \lg g)^2)$ if we do care.