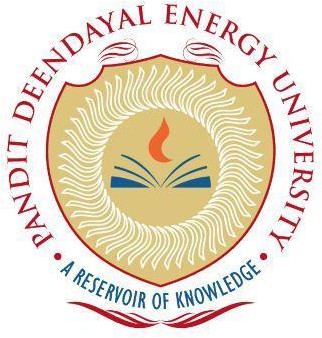
**PANDIT DEENDAYAL ENERGY UNIVERSITY SCHOOL OF TECHNOLOGY**

****

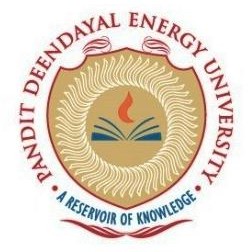
**Course: Information Security Lab Course Code: 20CP304P**

**LAB MANUAL**

|  |  |
| --- | --- |
| **Submitted To:** | **Submitted By:** |
| DR. Rutvij H Jhaveri | NAME: Prince Patel |
|  | ROLL NO: 23BCP002 |

**Pandit Deendayal Energy University**

Raisan, Gandhinagar – 380 007, Gujarat, India



**Computer Science Engineering Department**

**Certificate**

**This is to certify that**

**Mr./Ms. Roll no.**

**Exam No. of 5th Semester Degree course in Computer Science and Engineering has satisfactorily completed his/her term work in Information Security Lab (20CP304P) subject during the semester from to at School of Technology, PDEU.**

**Date of Submission:**

**Signature:**

**Faculty In-charge Head of Department**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **List of experiments** | **Date** | **Sign** |
| **1** | Study and Implement a program for Caesar Cipher |  |  |
| **2** | Study and Implement a program for 5x5 Playfair  Cipher to encrypt and decrypt the message. |  |  |
| **3** | Study and Implement a program for Rail Fence  Cipher |  |  |
| **4** | Study and implement a program for columnar  Transposition Cipher |  |  |
| **5** | Study and implement a program for Vigenère Cipher |  |  |
| **6** | Study and Implement a program for n-gram Hill  Cipher |  |  |
| **7** | Study and Use of RSA algorithm (encryption and  decryption) |  |  |
| **8** | Study and implement a program of the Digital  Signature with RSA algorithm (Reverse RSA) |  |  |
| **9** | Study and Use of Diffie-Hellman Key Exchange |  |  |
| **10** | Design cipher with detailed explanation. Sample as follows.   1. Write a program to encrypt the plaintext with the given key. E.g. plaintext GRONSFELD with the key 1234. Add 1 to G to get H (the letter 1 rank after G is H in the alphabet), then add 2 to C or E (the letter 2 ranks after C is E), and so on. Use smallest letter from plaintext as filler. 2. Encrypt the input words PLAINTEXT= RAG BABY to obtain CIPHERTEXT = SCJ DDFD |  |  |
| **11** | Mini project ( Need for Post Quantum Cryptography) |  |  |

*For each experiment, students need to perform their variation of the standard algorithm mentioned as the AIM of the experiment and perform the* ***Security Analysis,*** *where the student must perform a security analysis of the algorithm mentioned above. Analysis may include a) Cryptanalysis, b)Performance analysis*

**EXPERIMENT-1**

**AIM:** Study and Implement a program for Caesar Cipher

**Introduction :** In cryptography, a Caesar cipher, also known as Caesar's cipher, the shift cipher, Caesar's code, or Caesar shift, is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a left shift of 3, D would be replaced by A, E would become B, and so on. The method is named after Julius Caesar, who used it in his private correspondence.

**Example :** The transformation can be represented by aligning two alphabets; the cipher alphabet is the plain alphabet rotated left or right by some number of positions. For instance, here is a Caesar cipher using a left rotation of three places, equivalent to a right shift of 23 (the shift parameter is used as the [key](https://en.wikipedia.org/wiki/Key_(cryptography))):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Plain** | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| **Ciphe r** | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |

When encrypting, a person looks up each letter of the message in the "plain" line and writes down the corresponding letter in the "cipher" line.

Plaintext: THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD

Deciphering is done in reverse, with a right shift of 3.

The encryption can also be represented using [modular arithmetic](https://en.wikipedia.org/wiki/Modular_arithmetic) by first transforming the letters into numbers, according to the scheme, A → 0, B → 1, ..., Z → 25.

**Code:**

1. class **Normal**:

2.     def encrypt(self, text: str, shift: int) -> str:

3.         result = []

4.         for c in text:

5.             if c.isupper():

6.                 result.append(chr(((ord(c) - ord('A') + shift) % 26) + ord('A')))

7.             elif c.islower():

8.                 result.append(chr(((ord(c) - ord('a') + shift) % 26) + ord('a')))

9.             else:

10.                 result.append(c)

11.         return "".join(result)

12.

13.     def decrypt(self, text: str, shift: int) -> str:

14.         return self.encrypt(text, 26 - (shift % 26))

15.

16. class **Revised**:

17.     def encrypt(self, text: str, shift: int) -> str:

18.         result = []

19.         for c in text:

20.             if c.isupper():

21.

22.                 mirrored = chr(ord('z') - (ord(c) - ord('A')))

23.                 shifted = chr(((ord(mirrored) - ord('a') + shift) % 26) + ord('a'))

24.                 result.append(shifted)

25.             elif c.islower():

26.

27.                 mirrored = chr(ord('Z') - (ord(c) - ord('a')))

28.                 shifted = chr(((ord(mirrored) - ord('A') + shift) % 26) + ord('A'))

29.                 result.append(shifted)

30.             else:

31.                 result.append(c)

32.         return "".join(result)

33.

34.     def decrypt(self, text: str, shift: int) -> str:

35.         result = []

36.         for c in text:

37.             if c.islower():

38.

39.                 unshifted = chr(((ord(c) - ord('a') - shift + 26) % 26) + ord('a'))

40.                 original = chr(ord('A') + (ord('z') - ord(unshifted)))

41.                 result.append(original)

42.             elif c.isupper():

43.

44.                 unshifted = chr(((ord(c) - ord('A') - shift + 26) % 26) + ord('A'))

45.                 original = chr(ord('a') + (ord('Z') - ord(unshifted)))

46.                 result.append(original)

47.             else:

48.                 result.append(c)

49.         return "".join(result)

50.

51. def main():

52.     text = input("Enter the text to Encrypt: ")

53.

54.     print("Choose Caesar Cipher type:")

55.     print("1. Normal Caesar Cipher")

56.     print("2. Twisted Revised Caesar Cipher")

57.

58.     try:

59.         choice = int(input("Enter choice: "))

60.     except **ValueError**:

61.         print("Invalid input.")

62.         return

63.

64.     if choice == 1:

65.         shift = int(input("Enter shift value (0-25): "))

66.         normal = **Normal**()

67.         encrypted = normal.encrypt(text, shift)

68.         decrypted = normal.decrypt(encrypted, shift)

69.         print("\n--- Normal Caesar Cipher ---")

70.         print("Encrypted Text:", encrypted)

71.         print("Decrypted Text:", decrypted)

72.

73.     elif choice == 2:

74.         shift = int(input("Enter shift value (0-25): "))

75.         revised = **Revised**()

76.         encrypted = revised.encrypt(text, shift)

77.         decrypted = revised.decrypt(encrypted, shift)

78.         print("\n--- Twisted Revised Caesar Cipher ---")

79.         print("Encrypted Text:", encrypted)

80.         print("Decrypted Text:", decrypted)

81.

82.     else:

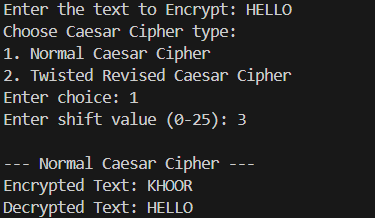
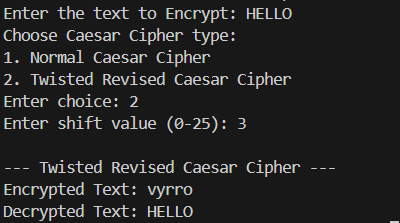
83.         print("Invalid option.")

84.

85. if \_\_name\_\_ == "\_\_main\_\_":

86.     main()

**Output:**

** **

**Security Analysis:**

**Cryptanalysis**

* **Normal Caesar:**
  + Simple letter shift.
  + Very weak: Only 26 possible keys, easily brute-forced or broken by frequency analysis.
* **Revised Caesar:**
  + Mirrors the character before shift and changes the case (upper to lower, lower to upper).
  + Slightly more obfuscation, but still simple substitution—remains insecure and easy to break.

**Design**

* Modular class structure for each cipher variation.
* Clear input/output flow using CLI.
* Handles both upper/lower case; preserves non-alphabetic characters.
* Easy to extend for other variants.

**Performance**

* Linear time (), very efficient for any practical text length.
* Memory usage is minimal (result list sized to input text).

**Sign of Faculty Date of performance**

**EXPERIMENT-2**

**AIM:** Study and Implement a program for 5x5 Playfair Cipher to encrypt and decrypt the message.

**Introduction :** The Playfair cipher or Playfair square or Wheatstone–Playfair cipher is a manual [symmetric](https://en.wikipedia.org/wiki/Symmetric_key_algorithm) [encryption](https://en.wikipedia.org/wiki/Encryption) technique and was the first literal [digram substitution](https://en.wikipedia.org/wiki/Polygraphic_substitution) cipher. The scheme was invented in 1854 by [Charles Wheatstone](https://en.wikipedia.org/wiki/Charles_Wheatstone), but bears the name of [Lord](https://en.wikipedia.org/wiki/Lord_Playfair) [Playfair](https://en.wikipedia.org/wiki/Lord_Playfair) for promoting its use.

The technique encrypts pairs of letters ([*bigrams*](https://en.wikipedia.org/wiki/Bigram)or *digrams*), instead of single letters as in the simple [substitution cipher](https://en.wikipedia.org/wiki/Substitution_cipher) and rather more complex [Vigenère cipher](https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher) systems then in use. The Playfair cipher is thus significantly harder to break since the [frequency analysis](https://en.wikipedia.org/wiki/Frequency_analysis) used for simple substitution ciphers does not work with it. The frequency analysis of bigrams is possible, but considerably more difficult.

**Example :** For the encryption process let us consider the following example:



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **The** | **Playfair** | **Cipher** | **Encryption** |  | **Algorithm:** |
| The | Algorithm | consists | of | 2 | steps: |

1. **Generate the key Square(5×5):**
   * The key square is a 5×5 grid of alphabets that acts as the key for encrypting the plaintext. Each of the 25 alphabets must be unique and one letter of the alphabet (usually J) is omitted from the table (as the table can hold only 25 alphabets). If the plaintext contains J, then it is replaced by I.
   * The initial alphabets in the key square are the unique alphabets of the key in the order in which they appear followed by the remaining letters of the alphabet in order.
2. **Algorithm to encrypt the plain text:** The plaintext is split into pairs of two letters (digraphs). If there is an odd number of letters, a Z is added to the last letter. **For**

**example:**

**PlainText**: "instruments"

**After Split:** 'in' 'st' 'ru' 'me' 'nt' 'sz

**Plain Text:** "instrumentsz"

**Encrypted Text:** gatlmzclrqtx

**Code:**

1. class **PlayfairCipher**:

2.     def \_\_init\_\_(self):

3.         self.key\_table = [['' for \_ in range(5)] for \_ in range(5)]

4.

5.     def to\_lowercase(self, text: str) -> str:

6.         return text.lower()

7.

8.     def remove\_spaces(self, text: str) -> str:

9.         return text.replace(" ", "")

10.

11.     def replace\_j\_with\_i(self, text: str) -> str:

12.         return text.replace("j", "i")

13.

14.     def prepare\_text(self, text: str) -> str:

15.         text = list(text)

16.         i = 0

17.         while i < len(text) - 1:

18.             if text[i] == text[i + 1]:

19.                 text.insert(i + 1, 'x')

20.             i += 2

21.         if len(text) % 2 != 0:

22.             text.append('z')

23.         return "".join(text)

24.

25.     def generate\_key\_table(self, key: str):

26.         used = [0] \* 26

27.         i = j = 0

28.

29.         for ch in key:

30.             if ch == 'j':

31.                 ch = 'i'

32.             if used[ord(ch) - ord('a')] == 0:

33.                 used[ord(ch) - ord('a')] = 1

34.                 self.key\_table[i][j] = ch

35.                 j += 1

36.                 if j == 5:

37.                     j = 0

38.                     i += 1

39.

40.         for ch in range(ord('a'), ord('z') + 1):

41.             if chr(ch) == 'j':

42.                 continue

43.             if used[ch - ord('a')] == 0:

44.                 used[ch - ord('a')] = 1

45.                 self.key\_table[i][j] = chr(ch)

46.                 j += 1

47.                 if j == 5:

48.                     j = 0

49.                     i += 1

50.

51.     def find\_position(self, ch: str):

52.         if ch == 'j':

53.             ch = 'i'

54.         for i in range(5):

55.             for j in range(5):

56.                 if self.key\_table[i][j] == ch:

57.                     return (i, j)

58.         return (-1, -1)

59.

60.     # -------- Normal Playfair --------

61.     def encrypt\_normal(self, text: str) -> str:

62.         text = list(text)

63.         for i in range(0, len(text), 2):

64.             r1, c1 = self.find\_position(text[i])

65.             r2, c2 = self.find\_position(text[i + 1])

66.             if r1 == r2:  # same row

67.                 text[i] = self.key\_table[r1][(c1 + 1) % 5]

68.                 text[i + 1] = self.key\_table[r2][(c2 + 1) % 5]

69.             elif c1 == c2:  # same column

70.                 text[i] = self.key\_table[(r1 + 1) % 5][c1]

71.                 text[i + 1] = self.key\_table[(r2 + 1) % 5][c2]

72.             else:  # rectangle

73.                 text[i] = self.key\_table[r1][c2]

74.                 text[i + 1] = self.key\_table[r2][c1]

75.         return "".join(text)

76.

77.     def decrypt\_normal(self, text: str) -> str:

78.         text = list(text)

79.         for i in range(0, len(text), 2):

80.             r1, c1 = self.find\_position(text[i])

81.             r2, c2 = self.find\_position(text[i + 1])

82.             if r1 == r2:

83.                 text[i] = self.key\_table[r1][(c1 + 4) % 5]

84.                 text[i + 1] = self.key\_table[r2][(c2 + 4) % 5]

85.             elif c1 == c2:

86.                 text[i] = self.key\_table[(r1 + 4) % 5][c1]

87.                 text[i + 1] = self.key\_table[(r2 + 4) % 5][c2]

88.             else:

89.                 text[i] = self.key\_table[r1][c2]

90.                 text[i + 1] = self.key\_table[r2][c1]

91.         return "".join(text)

92.

93.     # -------- Revised Playfair --------

94.     def encrypt\_letter\_revised(self, letter: str) -> str:

95.         if letter == 'j':

96.             letter = 'i'

97.         row, col = self.find\_position(letter)

98.         first = self.key\_table[row][(col + 1) % 5]

99.         second = self.key\_table[row][(col + 2) % 5]

100.         return first + second

101.

102.     def encrypt\_revised(self, text: str) -> str:

103.         result = []

104.         for ch in text:

105.             result.append(self.encrypt\_letter\_revised(ch))

106.         return "".join(result)

107.

108.     def decrypt\_block\_revised(self, block: str) -> str:

109.         if len(block) != 2:

110.             return 'x'

111.         first, second = block[0], block[1]

112.         r1, c1 = self.find\_position(first)

113.         r2, c2 = self.find\_position(second)

114.         if r1 == r2:

115.             for col in range(5):

116.                 if (self.key\_table[r1][(col + 1) % 5] == first and

117.                         self.key\_table[r1][(col + 2) % 5] == second):

118.                     return self.key\_table[r1][col]

119.         return self.key\_table[r1][c1]  # fallback

120.

121.     def decrypt\_revised(self, text: str) -> str:

122.         result = []

123.         for i in range(0, len(text), 2):

124.             block = text[i:i + 2]

125.             result.append(self.decrypt\_block\_revised(block))

126.         return "".join(result)

127.

128.     def print\_key\_table(self):

129.         print("Key Table:")

130.         for row in self.key\_table:

131.             print(" ".join(row))

132.

133. def main():

134.     cipher = **PlayfairCipher**()

135.     key = input("Enter keyword: ")

136.     text = input("Enter plaintext: ")

137.

138.     key = cipher.to\_lowercase(key)

139.     text = cipher.to\_lowercase(text)

140.

141.     key = cipher.remove\_spaces(key)

142.     text = cipher.remove\_spaces(text)

143.

144.     key = cipher.replace\_j\_with\_i(key)

145.     text = cipher.replace\_j\_with\_i(text)

146.

147.     text\_prepared = cipher.prepare\_text(text)

148.

149.     cipher.generate\_key\_table(key)

150.     cipher.print\_key\_table()

151.

152.     print("\nChoose Cipher Type:")

153.     print("1. Normal Playfair Cipher")

154.     print("2. Revised Playfair Cipher")

155.     choice = input("Enter choice: ")

156.

157.     if choice == "1":

158.         encrypted = cipher.encrypt\_normal(text\_prepared)

159.         decrypted = cipher.decrypt\_normal(encrypted)

160.         print("\n--- Normal Playfair Cipher ---")

161.         print("Original text:", text\_prepared)

162.         print("Encrypted text:", encrypted)

163.         print("Decrypted text:", decrypted)

164.     elif choice == "2":

165.         encrypted = cipher.encrypt\_revised(text)

166.         decrypted = cipher.decrypt\_revised(encrypted)

167.         print("\n--- Revised Playfair Cipher ---")

168.         print("Original text:", text)

169.         print("Encrypted text:", encrypted)

170.         print("Decrypted text:", decrypted)

171.     else:

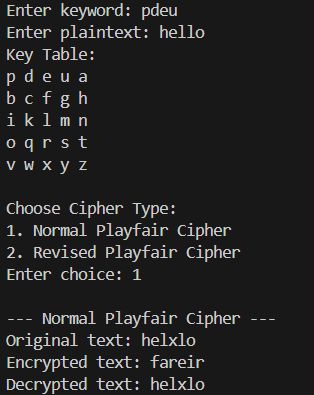
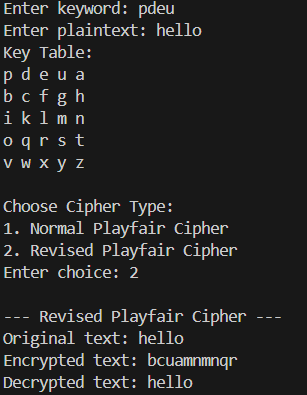
172.         print("Invalid option.")

173.

174. if \_\_name\_\_ == "\_\_main\_\_":

175.     main()

**Output:**

** **

**Security Analysis:**

**Cryptanalysis**

* **Normal Playfair:**
  + Uses bigrams (pair-wise substitution) for encryption.
  + More secure than monoalphabetic ciphers but easily broken today via digraph analysis and known-plaintext attacks.
* **Revised Playfair:**
  + Each letter maps to two table letters in its row (“row skip” method).
  + Slightly increased complexity, but still fully predictable and not resistant to cryptanalysis with modern computational tools.

**Design**

* All logic wrapped in a class.
* Clear separation of preprocessing (case, space, j/i, padding), table generation, and main encryption/decryption flows.
* Modular functions for both cipher types.
* Prints key table for inspection and debugging.

**Performance**

* **Key table**: O(1) (always 25 slots).
* **Encrypt/decrypt**: O(n) (linear to message length).
* Low memory and suited for educational/historic demo use.

**Sign of Faculty Date of performance**

**EXPERIMENT-3**

**AIM:** Study and Implement a program for Rail Fence Cipher with columnar transposition

**Introduction :** The rail fence cipher (also called a zigzag cipher) is a [classical](https://en.wikipedia.org/wiki/Classical_cipher) type of [transposition cipher.](https://en.wikipedia.org/wiki/Transposition_cipher) It derives its name from the manner in which encryption is performed, in analogy to a fence built with horizontal rails.

**Example :** In the rail fence cipher, the plaintext is written downwards diagonally on successive "rails" of an imaginary fence, then moving up when the bottom rail is reached, down again when the top rail is reached, and so on until the whole plaintext is written out. The ciphertext is then read off in rows.

For example, to encrypt the message 'WE ARE DISCOVERED. RUN AT ONCE.' with 3 "rails", write the text as:

W . . . E . . . C . . . R . . . U . . . O . . .

. E . R . D . S . O . E . E . R . N . T . N . E

. . A . . . I . . . V . . . D . . . A . . . C .

(Note that spaces and punctuation are omitted.) Then read off the text horizontally to get the ciphertext:

WECRUO ERDSOEERNTNE AIVDAC

**Code:**

1. import sys

2.

3. # === Normal Rail Fence Methods ===

4.

5. def rail\_fence\_encrypt(text, rails):

6.     if rails <= 1:

7.         return text

8.

9.     fence = ["" for \_ in range(rails)]

10.     rail = 0

11.     direction = 1  # 1 = down, -1 = up

12.

13.     for c in text:

14.         fence[rail] += c

15.         rail += direction

16.

17.         if rail == 0 or rail == rails - 1:

18.             direction \*= -1

19.

20.     return "".join(fence)

21.

22. def rail\_fence\_decrypt(cipher\_text, rails):

23.     if rails <= 1:

24.         return cipher\_text

25.

26.     # Create pattern

27.     pattern = [['\n'] \* len(cipher\_text) for \_ in range(rails)]

28.     rail, direction = 0, 1

29.

30.     for i in range(len(cipher\_text)):

31.         pattern[rail][i] = '\*'

32.         rail += direction

33.

34.         if rail == 0 or rail == rails - 1:

35.             direction \*= -1

36.

37.     # Fill pattern with ciphertext

38.     index = 0

39.     for r in range(rails):

40.         for c in range(len(cipher\_text)):

41.             if pattern[r][c] == '\*' and index < len(cipher\_text):

42.                 pattern[r][c] = cipher\_text[index]

43.                 index += 1

44.

45.     # Read plaintext

46.     result = []

47.     rail, direction = 0, 1

48.     for i in range(len(cipher\_text)):

49.         result.append(pattern[rail][i])

50.         rail += direction

51.         if rail == 0 or rail == rails - 1:

52.             direction \*= -1

53.

54.     return "".join(result)

55.

56. # === Revised Rail Fence Methods (start from bottom rail) ===

57.

58. def rail\_fence\_encrypt\_revised(text, rails):

59.     if rails <= 1:

60.         return text

61.

62.     fence = ["" for \_ in range(rails)]

63.     rail, direction = rails - 1, -1  # start from bottom, going up

64.

65.     for c in text:

66.         fence[rail] += c

67.         rail += direction

68.

69.         if rail == 0 or rail == rails - 1:

70.             direction \*= -1

71.

72.     return "".join(fence)

73.

74. def rail\_fence\_decrypt\_revised(cipher\_text, rails):

75.     if rails <= 1:

76.         return cipher\_text

77.

78.     pattern = [['\n'] \* len(cipher\_text) for \_ in range(rails)]

79.     rail, direction = rails - 1, -1

80.

81.     for i in range(len(cipher\_text)):

82.         pattern[rail][i] = '\*'

83.         rail += direction

84.         if rail == 0 or rail == rails - 1:

85.             direction \*= -1

86.

87.     index = 0

88.     for r in range(rails):

89.         for c in range(len(cipher\_text)):

90.             if pattern[r][c] == '\*' and index < len(cipher\_text):

91.                 pattern[r][c] = cipher\_text[index]

92.                 index += 1

93.

94.     result = []

95.     rail, direction = rails - 1, -1

96.     for i in range(len(cipher\_text)):

97.         result.append(pattern[rail][i])

98.         rail += direction

99.         if rail == 0 or rail == rails - 1:

100.             direction \*= -1

101.

102.     return "".join(result)

103.

104. # === Caesar Cipher Methods ===

105.

106. def caesar\_encrypt(text, shift):

107.     result = []

108.     for c in text:

109.         if c.isalpha():

110.             base = ord('A') if c.isupper() else ord('a')

111.             result.append(chr((ord(c) - base + shift) % 26 + base))

112.         else:

113.             result.append(c)

114.     return "".join(result)

115.

116. def caesar\_decrypt(text, shift):

117.     return caesar\_encrypt(text, -shift)

118.

119. # === Combined Encryption/Decryption for Revised Mode ===

120.

121. def encrypt\_revised(text, rails):

122.     rail\_encrypted = rail\_fence\_encrypt\_revised(text, rails)

123.     return caesar\_decrypt(rail\_encrypted, rails)

124.

125. def decrypt\_revised(cipher\_text, rails):

126.     caesar\_dec = caesar\_encrypt(cipher\_text, rails)

127.     return rail\_fence\_decrypt\_revised(caesar\_dec, rails)

128.

129. # === Visualization Helper ===

130.

131. def visualize\_rail\_fence(text, rails, revised=False):

132.     if rails <= 1:

133.         print("Text:", text)

134.         return

135.

136.     pattern = [[' '] \* len(text) for \_ in range(rails)]

137.     rail = rails - 1 if revised else 0

138.     direction = -1 if revised else 1

139.

140.     for i, c in enumerate(text):

141.         pattern[rail][i] = c

142.         rail += direction

143.         if rail == 0 or rail == rails - 1:

144.             direction \*= -1

145.

146.     mode = "Revised" if revised else "Normal"

147.     print(f"Rail Fence Pattern ({mode}):")

148.     for r in range(rails):

149.         print(f"Rail {r}: ", " ".join(pattern[r]))

150.

151.

152.

153. def main():

154.     print("=== Rail Fence Cipher ===")

155.     print("Choose mode:")

156.     print("1) Normal Rail Fence (starts from top rail)")

157.     print("2) Revised Rail Fence (starts from bottom rail + Caesar shift)")

158.     mode = int(input("Enter 1 or 2: "))

159.

160.     message = input("Enter the message to encrypt: ")

161.     num\_rails = int(input("Enter the number of rails: "))

162.

163.     if num\_rails <= 0:

164.         print("Number of rails must be positive!")

165.         sys.exit()

166.

167.     print("\nOriginal message:", message)

168.     print("Number of rails:", num\_rails)

169.

170.     if mode == 1:

171.         print("\n--- Normal Rail Fence Mode ---")

172.         visualize\_rail\_fence(message, num\_rails, revised=False)

173.

174.         encrypted = rail\_fence\_encrypt(message, num\_rails)

175.         print("\nEncrypted:", encrypted)

176.

177.         decrypted = rail\_fence\_decrypt(encrypted, num\_rails)

178.         print("Decrypted:", decrypted)

179.

180.         if message == decrypted:

181.             print("✓ Normal encryption/decryption successful!")

182.         else:

183.             print("✗ Error in normal encryption/decryption!")

184.

185.     else:

186.         print("\n--- Revised Rail Fence Mode ---")

187.         visualize\_rail\_fence(message, num\_rails, revised=True)

188.

189.         rail\_encrypted = rail\_fence\_encrypt\_revised(message, num\_rails)

190.         print("\nAfter Rail Fence encryption:", rail\_encrypted)

191.

192.         fully\_encrypted = caesar\_decrypt(rail\_encrypted, num\_rails)

193.         print("Fully encrypted (with Caesar left shift):", fully\_encrypted)

194.

195.         caesar\_dec = caesar\_encrypt(fully\_encrypted, num\_rails)

196.         print("After Caesar right shift:", caesar\_dec)

197.

198.         fully\_decrypted = rail\_fence\_decrypt\_revised(caesar\_dec, num\_rails)

199.         print("Fully decrypted:", fully\_decrypted)

200.

201.         if message == fully\_decrypted:

202.             print("Revised encryption/decryption successful!")

203.         else:

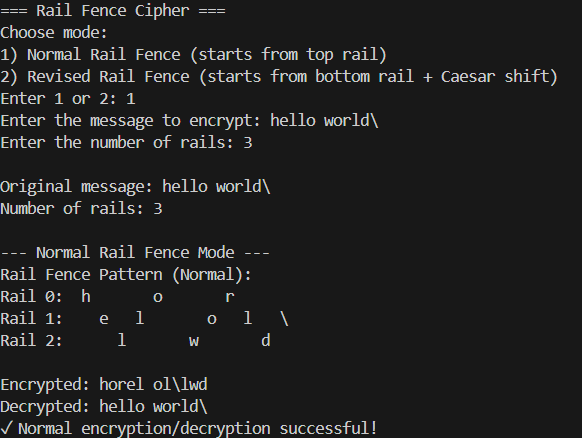
204.             print("Error in revised encryption/decryption!")

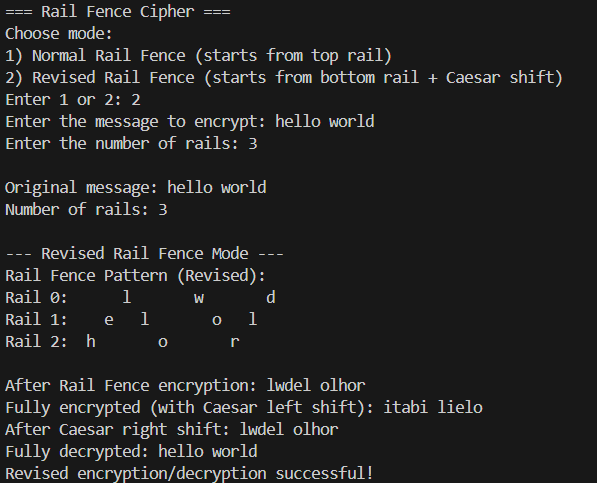
205.

206. if \_\_name\_\_ == "\_\_main\_\_":

207.     main()

**Output:**

****

****

**Security Analysis:**

**Cryptanalysis**

* **Normal Rail Fence:**
  + Rearranges text across several rails, reading row by row.
  + Easy to brute-force (try all rail counts), vulnerable to known-plaintext attacks. Does not hide letter frequencies.
* **Revised Rail Fence:**
  + Starts from bottom rail instead of top, then applies a Caesar left-shift (simple letter substitution).
  + Slightly more obfuscated, but both Rail Fence and Caesar are **simple transposition/substitution** ciphers, breakable with frequency analysis or brute-forcing rail/shift values.
  + Combined method improves resistance only against the most basic attacks.

**Design**

* Modular functions for each encryption/decryption method (normal/revised).
* Helper for visualizing rail patterns—very useful for learning/demo.
* Code is clear, implements error-handling for input.
* Main function offers easy mode selection between classic and revised.

**Performance**

* All methods are linear () where  message length.
* Each encryption/decryption is a loop with constant-time operations.
* Visualization adds small overhead but is efficient for short texts.

**Sign of Faculty Date of performance**

**EXPERIMENT-4**

**AIM:** Study and implement a program for columnar Transposition Cipher

**Introduction :** The columnar transposition cipher is a fairly simple, easy to implement cipher. It is a transposition cipher that follows a simple rule for mixing up the characters in the plaintext to form the ciphertext. Although weak on its own, it can be combined with other ciphers, such as a substitution cipher, the combination of which can be more difficult to break than either cipher on it's own.

**Example :** The key for the columnar transposition cipher is a keyword e.g. GERMAN. The row length that is used is the same as the length of the keyword. To encrypt a piece of text, e.g.

defend the east wall of the castle

we write it out in a special way in a number of rows (the keyword here is GERMAN):

G E R M A N

d e f e n d t h e e a s t w a l l o f t h e c a s t l e x x

In the above example, the plaintext has been padded so that it neatly fits in a rectangle. This is known as a regular columnar transposition. An irregular columnar transposition leaves these characters blank, though this makes decryption slightly more difficult. The columns are now reordered such that the letters in the key word are ordered alphabetically.

A E G M N R

n e d e d f a h t e s e l w t l o a c t f e a h x t s e x l

The ciphertext is read off along the columns:

nalcxehwttdttfseeleedsoaxfeahl

**Code:**

1. import math

2.

3. # === Standard Columnar Encryption ===

4. def encrypt\_standard(text, key):

5.     col = len(key)

6.     row = math.ceil(len(text) / col)

7.

8.     # Fill matrix row-wise with padding 'X'

9.     matrix = [['X'] \* col for \_ in range(row)]

10.     k = 0

11.     for i in range(row):

12.         for j in range(col):

13.             if k < len(text):

14.                 matrix[i][j] = text[k]

15.                 k += 1

16.

17.     # Get column order

18.     order = get\_key\_order(key)

19.

20.     # Read columns according to order

21.     cipher = []

22.     for idx in order:

23.         for i in range(row):

24.             cipher.append(matrix[i][idx])

25.     return "".join(cipher)

26.

27. # === Standard Columnar Decryption ===

28. def decrypt\_standard(cipher, key):

29.     col = len(key)

30.     row = math.ceil(len(cipher) / col)

31.

32.     # Empty matrix

33.     matrix = [[''] \* col for \_ in range(row)]

34.

35.     # Column order

36.     order = get\_key\_order(key)

37.

38.     k = 0

39.     for idx in order:

40.         for i in range(row):

41.             if k < len(cipher):

42.                 matrix[i][idx] = cipher[k]

43.                 k += 1

44.

45.     # Read row-wise

46.     plain = []

47.     for i in range(row):

48.         for j in range(col):

49.             plain.append(matrix[i][j])

50.

51.     return "".join(plain).rstrip("X")  # remove padding

52.

53. # === Revised Columnar Encryption ===

54. def encrypt\_revised(text, key):

55.     col = len(key)

56.     row = math.ceil(len(text) / col)

57.

58.     # Fill matrix row-wise

59.     matrix = [['X'] \* col for \_ in range(row)]

60.     k = 0

61.     for i in range(row):

62.         for j in range(col):

63.             if k < len(text):

64.                 matrix[i][j] = text[k]

65.                 k += 1

66.

67.     # Shift each row by (i+1)

68.     for i in range(row):

69.         matrix[i] = shift\_row(matrix[i], i + 1)

70.

71.     order = get\_key\_order(key)

72.

73.     # Read columns

74.     cipher = []

75.     for idx in order:

76.         for i in range(row):

77.             cipher.append(matrix[i][idx])

78.     return "".join(cipher)

79.

80. # === Revised Columnar Decryption ===

81. def decrypt\_revised(cipher, key):

82.     col = len(key)

83.     row = math.ceil(len(cipher) / col)

84.

85.     matrix = [[''] \* col for \_ in range(row)]

86.     order = get\_key\_order(key)

87.

88.     k = 0

89.     for idx in order:

90.         for i in range(row):

91.             if k < len(cipher):

92.                 matrix[i][idx] = cipher[k]

93.                 k += 1

94.

95.     # Undo row shifts

96.     for i in range(row):

97.         matrix[i] = shift\_row(matrix[i], col - (i + 1))

98.

99.     # Read row-wise

100.     plain = []

101.     for i in range(row):

102.         for j in range(col):

103.             plain.append(matrix[i][j])

104.

105.     return "".join(plain).rstrip("X")

106.

107. # === Helpers ===

108. def get\_key\_order(key):

109.     # Returns order of column indices sorted by key chars

110.     return [i for i, \_ in sorted(enumerate(key), key=lambda x: x[1])]

111.

112. def shift\_row(row, shift):

113.     n = len(row)

114.     return row[-shift % n:] + row[:-shift % n]

115.

116. # === Main Demo ===

117. if \_\_name\_\_ == "\_\_main\_\_":

118.     text = "HELLOWORLD"

119.     key = "SECRET"

120.

121.     print("=== Standard Columnar Cipher ===")

122.     enc\_std = encrypt\_standard(text, key)

123.     dec\_std = decrypt\_standard(enc\_std, key)

124.     print("Plaintext: ", text)

125.     print("Encrypted:", enc\_std)

126.     print("Decrypted:", dec\_std)

127.

128.     print("\n=== Revised Columnar Cipher ===")

129.     enc\_rev = encrypt\_revised(text, key)

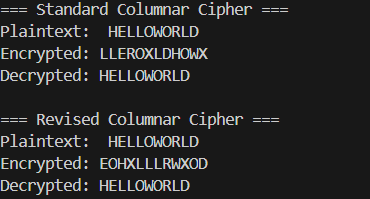
130.     dec\_rev = decrypt\_revised(enc\_rev, key)

131.     print("Plaintext: ", text)

132.     print("Encrypted:", enc\_rev)

133.     print("Decrypted:", dec\_rev)

**Output:**

****

**Security Analysis:**

**Cryptanalysis**

* **Standard columnar cipher:**
  + Rearranges plaintext by columns based on keyword order.
  + Security depends on obscurity of key and message length; vulnerable to frequency analysis, pattern recognition, and known-plaintext attacks. Not secure for modern use.
* **Revised variant:**
  + Adds row-wise shifts to each row before column reading, then undoes shifts in decryption.
  + Slightly harder for simple attacks, but still predictable and not resistant to contemporary cryptanalysis — composite method, not fundamentally stronger.

**Design**

* Clean function separation for standard/revised encryption and decryption.
* Uses matrix for text arrangement and key-based column ordering.
* Helpers improve modularity (row shift, key sorting).
* Padding managed with 'X'; stripped after decryption.
* Main block demonstrates usage, prints results.

**Performance**

* All steps are , efficient for normal message sizes.
* Uses minimal extra memory (matrix and lists).

**Sign of Faculty Date of performance**

**EXPERIMENT-5**

**AIM:** Study and implement a program for Vigenère Cipher

**Introduction :** The Vigenère cipher (French pronunciation: [viʒnɛːʁ]) is a method of [encrypting](https://en.wikipedia.org/wiki/Encryption) [alphabetic](https://en.wikipedia.org/wiki/Alphabetic) text where each letter of the [plaintext](https://en.wikipedia.org/wiki/Plaintext) is encoded with a different [Caesar cipher,](https://en.wikipedia.org/wiki/Caesar_cipher) whose increment is determined by the corresponding letter of another text, the [key.](https://en.wikipedia.org/wiki/Key_(cryptography)) The Vigenère cipher is therefore a special case of a [polyalphabetic substitution](https://en.wikipedia.org/wiki/Polyalphabetic_cipher).

**Example :** If the plaintext is attacking tonight and the key is OCULORHINOLARINGOLOGY, then

* the first letter a of the plaintext is shifted by 14 positions in the alphabet (because the first letter O of the key is the 14th letter of the alphabet, counting from zero), yielding o;
* the second letter t is shifted by 2 (because the second letter C of the key means 2) yielding v;
* the third letter t is shifted by 20 (U) yielding n, with wrap-around;

and so on; yielding the message ovnlqbpvt hznzouz. If the recipient of the message knows the key, they can recover the plaintext by reversing this process.

**Code:**

1. # === Vigenère Cipher (Python) ===

2.

3. REVISED\_CHARSET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 .,!?"

4. REVISED\_MOD = len(REVISED\_CHARSET)

5.

6. # === Utility Functions ===

7. def clean\_text\_normal(text):

8.     return "".join([c.upper() for c in text if c.isalpha()])

9.

10. def clean\_text\_revised(text):

11.     return "".join([c.upper() if 'a' <= c <= 'z' else c if c in REVISED\_CHARSET else ' ' for c in text])

12.

13. def char\_to\_num\_normal(c):

14.     return ord(c) - ord('A')

15.

16. def num\_to\_char\_normal(n):

17.     return chr(n + ord('A'))

18.

19. def modM(x, m):

20.     x %= m

21.     return x if x >= 0 else x + m

22.

23. def char\_to\_num\_revised(c):

24.     return REVISED\_CHARSET.index(c)

25.

26. def num\_to\_char\_revised(n):

27.     return REVISED\_CHARSET[n]

28.

29. # === Normal Vigenère Cipher ===

30. def encrypt\_vigenere\_normal(plaintext, key):

31.     text = clean\_text\_normal(plaintext)

32.     key = clean\_text\_normal(key)

33.     ciphertext = []

34.     key\_len = len(key)

35.

36.     for i, ch in enumerate(text):

37.         p = char\_to\_num\_normal(ch)

38.         k = char\_to\_num\_normal(key[i % key\_len])

39.         c = (p + k) % 26

40.         ciphertext.append(num\_to\_char\_normal(c))

41.     return "".join(ciphertext)

42.

43. def decrypt\_vigenere\_normal(ciphertext, key):

44.     key = clean\_text\_normal(key)

45.     plaintext = []

46.     key\_len = len(key)

47.

48.     for i, ch in enumerate(ciphertext):

49.         c = char\_to\_num\_normal(ch)

50.         k = char\_to\_num\_normal(key[i % key\_len])

51.         p = (c - k) % 26

52.         plaintext.append(num\_to\_char\_normal(p))

53.     return "".join(plaintext)

54.

55. # === Revised Vigenère Cipher ===

56. def encrypt\_vigenere\_revised(plaintext, key):

57.     text = clean\_text\_revised(plaintext)

58.     key = clean\_text\_revised(key)

59.     ciphertext = []

60.     key\_len = len(key)

61.

62.     for i, ch in enumerate(text):

63.         p = char\_to\_num\_revised(ch)

64.         k = char\_to\_num\_revised(key[i % key\_len])

65.         c = modM(p + k, REVISED\_MOD)

66.         ciphertext.append(num\_to\_char\_revised(c))

67.     return "".join(ciphertext)

68.

69. def decrypt\_vigenere\_revised(ciphertext, key):

70.     key = clean\_text\_revised(key)

71.     plaintext = []

72.     key\_len = len(key)

73.

74.     for i, ch in enumerate(ciphertext):

75.         c = char\_to\_num\_revised(ch)

76.         k = char\_to\_num\_revised(key[i % key\_len])

77.         p = modM(c - k, REVISED\_MOD)

78.         plaintext.append(num\_to\_char\_revised(p))

79.     return "".join(plaintext)

80.

81. # === Demo ===

82. if \_\_name\_\_ == "\_\_main\_\_":

83.     print("=== Vigenere Cipher ===")

84.     print("Choose mode:\n1) Normal (A-Z only)\n2) Revised (A-Z,0-9, space,.,!?)")

85.     mode = int(input("Enter 1 or 2: "))

86.

87.     plaintext = input("Enter plaintext: ")

88.     key = input("Enter key: ")

89.

90.     if mode == 1:

91.         ciphertext = encrypt\_vigenere\_normal(plaintext, key)

92.         print("Ciphertext:", ciphertext)

93.         decrypted = decrypt\_vigenere\_normal(ciphertext, key)

94.         print("Decrypted:", decrypted)

95.     else:

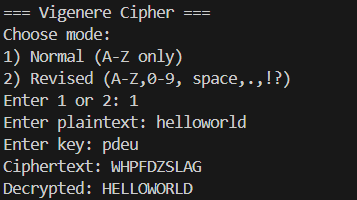
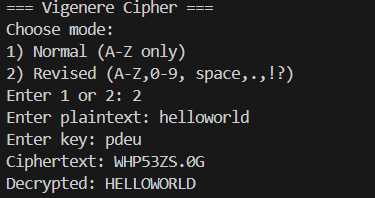
96.         ciphertext = encrypt\_vigenere\_revised(plaintext, key)

97.         print("Ciphertext:", ciphertext)

98.         decrypted = decrypt\_vigenere\_revised(ciphertext, key)

99.         print("Decrypted:", decrypted)

**Output:**

** **

**Security Analysis:**

**Cryptanalysis**

* **Normal Vigenère:**
  + Uses a repeating-key Caesar shift.
  + More secure than Caesar, but vulnerable if key is short or reused—can be broken via Kasiski examination or frequency analysis.
* **Revised Vigenère:**
  + Same logic, but works with expanded charset, so it can encrypt numbers, spaces, and punctuation.
  + Slightly increases complexity and keyspace, but fundamentally shares the same vulnerabilities as classic Vigenère.

**Design**

* Well-organized: function separation for cleaning, conversion, encrypt, decrypt.
* Utility functions clearly abstract key steps; revised variant uses modular arithmetic for wider charset.
* CLI interface for mode selection and I/O.

**Performance**

* Linear time (), efficient for any message length.
* Memory usage is minimal (lists the size of input/output).

**Sign of Faculty Date of performance**

**EXPERIMENT-6**

**AIM:** Study and Implement a program for n-gram Hill Cipher

**Introduction :** Hill cipher is a polygraphic substitution cipher based on linear algebra.Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, …, Z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible n × n matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption. The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible n × n matrices (modulo 26).

**Example :** Input : Plaintext: ACT

Key: GYBNQKURP

Output : Ciphertext: POH

**Code:**

1. import math

2.

3. # === Utility functions for normal (A..Z) mode ===

4. def char\_to\_num(c):

5.     return ord(c) - ord('A')

6.

7. def num\_to\_char(n):

8.     return chr(n + ord('A'))

9.

10. def mod26(x):

11.     x %= 26

12.     return x if x >= 0 else x + 26

13.

14. # === Revised mode settings (A-Z, 0-9, space, punctuation) ===

15. REVISED\_CHARSET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 .,!?"

16. REVISED\_MOD = len(REVISED\_CHARSET)

17.

18. def modM(x, m):

19.     x %= m

20.     return x if x >= 0 else x + m

21.

22. def r\_char\_to\_num(c):

23.     return REVISED\_CHARSET.index(c)

24.

25. def r\_num\_to\_char(n):

26.     return REVISED\_CHARSET[n]

27.

28. def inv\_mod(a, m):

29.     a = modM(a, m)

30.     for x in range(1, m):

31.         if modM(a \* x, m) == 1:

32.             return x

33.     return -1

34.

35. # === Matrix operations ===

36. def det2x2(k):

37.     a, b = k[0][0], k[0][1]

38.     c, d = k[1][0], k[1][1]

39.     return mod26(a \* d - b \* c)

40.

41. def inv\_mod26(a):

42.     a = mod26(a)

43.     for x in range(1, 26):

44.         if mod26(a \* x) == 1:

45.             return x

46.     return -1

47.

48. def inverse2x2(k):

49.     det = det2x2(k)

50.     detInv = inv\_mod26(det)

51.     if detInv == -1:

52.         raise **ValueError**(f"Key matrix not invertible mod 26 (det={det})")

53.     a, b, c, d = k[0][0], k[0][1], k[1][0], k[1][1]

54.     adj = [[d, -b], [-c, a]]

55.     return [[mod26(adj[0][0] \* detInv),

56.              mod26(adj[0][1] \* detInv)],

57.             [mod26(adj[1][0] \* detInv),

58.              mod26(adj[1][1] \* detInv)]]

59.

60. def inverse2x2\_mod(k, m):

61.     a, b = k[0][0], k[0][1]

62.     c, d = k[1][0], k[1][1]

63.     det = modM(a \* d - b \* c, m)

64.     detInv = inv\_mod(det, m)

65.     if detInv == -1:

66.         raise **ValueError**(f"Key not invertible mod {m} (det={det})")

67.     adj = [[d, -b], [-c, a]]

68.     return [[modM(adj[0][0] \* detInv, m),

69.              modM(adj[0][1] \* detInv, m)],

70.             [modM(adj[1][0] \* detInv, m),

71.              modM(adj[1][1] \* detInv, m)]]

72.

73. def mul2x2\_vec(k, v):

74.     return [mod26(k[0][0] \* v[0] + k[0][1] \* v[1]),

75.             mod26(k[1][0] \* v[0] + k[1][1] \* v[1])]

76.

77. def mul2x2\_vec\_mod(k, v, m):

78.     return [modM(k[0][0] \* v[0] + k[0][1] \* v[1], m),

79.             modM(k[1][0] \* v[0] + k[1][1] \* v[1], m)]

80.

81. # === Preprocessing ===

82. def clean\_and\_pad(text):

83.     sb = []

84.     for ch in text:

85.         if 'a' <= ch <= 'z':

86.             ch = chr(ord(ch) - 32)

87.         if 'A' <= ch <= 'Z':

88.             sb.append(ch)

89.     if len(sb) % 2 == 1:

90.         sb.append('X')

91.     return "".join(sb)

92.

93. def revised\_clean\_and\_pad(text):

94.     sb = []

95.     for ch in text:

96.         if 'a' <= ch <= 'z':

97.             ch = chr(ord(ch) - 32)

98.         if ch in REVISED\_CHARSET:

99.             sb.append(ch)

100.         else:

101.             sb.append(' ')

102.     if len(sb) % 2 == 1:

103.         sb.append(' ')

104.     return "".join(sb)

105.

106. # === Normal Hill Cipher ===

107. def encrypt\_normal(plaintext, key):

108.     p = clean\_and\_pad(plaintext)

109.     c = []

110.     for i in range(0, len(p), 2):

111.         block = [char\_to\_num(p[i]), char\_to\_num(p[i + 1])]

112.         enc = mul2x2\_vec(key, block)

113.         c.append(num\_to\_char(enc[0]))

114.         c.append(num\_to\_char(enc[1]))

115.     return "".join(c)

116.

117. def decrypt\_normal(ciphertext, key):

118.     inv = inverse2x2(key)

119.     p = []

120.     for i in range(0, len(ciphertext), 2):

121.         block = [char\_to\_num(ciphertext[i]), char\_to\_num(ciphertext[i + 1])]

122.         dec = mul2x2\_vec(inv, block)

123.         p.append(num\_to\_char(dec[0]))

124.         p.append(num\_to\_char(dec[1]))

125.     return "".join(p)

126.

127. # === Revised Hill Cipher ===

128. def encrypt\_revised(plaintext, key):

129.     p = revised\_clean\_and\_pad(plaintext)

130.     c = []

131.     for i in range(0, len(p), 2):

132.         block = [r\_char\_to\_num(p[i]), r\_char\_to\_num(p[i + 1])]

133.         blockIndex = i // 2

134.         # dynamic multiplier

135.         s = 1 + (blockIndex % (REVISED\_MOD - 1))

136.         dyn = [[modM(key[0][0] \* s, REVISED\_MOD), modM(key[0][1] \* s, REVISED\_MOD)],

137.                [modM(key[1][0] \* s, REVISED\_MOD), modM(key[1][1] \* s, REVISED\_MOD)]]

138.         enc = mul2x2\_vec\_mod(dyn, block, REVISED\_MOD)

139.         c.append(r\_num\_to\_char(enc[0]))

140.         c.append(r\_num\_to\_char(enc[1]))

141.     return "".join(c)

142.

143. def decrypt\_revised(ciphertext, key):

144.     p = []

145.     for i in range(0, len(ciphertext), 2):

146.         y = [r\_char\_to\_num(ciphertext[i]), r\_char\_to\_num(ciphertext[i + 1])]

147.         blockIndex = i // 2

148.         s = 1 + (blockIndex % (REVISED\_MOD - 1))

149.         dyn = [[modM(key[0][0] \* s, REVISED\_MOD), modM(key[0][1] \* s, REVISED\_MOD)],

150.                [modM(key[1][0] \* s, REVISED\_MOD), modM(key[1][1] \* s, REVISED\_MOD)]]

151.         invDyn = inverse2x2\_mod(dyn, REVISED\_MOD)

152.         dec\_block = mul2x2\_vec\_mod(invDyn, y, REVISED\_MOD)

153.         p.append(r\_num\_to\_char(dec\_block[0]))

154.         p.append(r\_num\_to\_char(dec\_block[1]))

155.     return "".join(p)

156.

157. # === Demo ===

158. if \_\_name\_\_ == "\_\_main\_\_":

159.     print("=== Simple Hill Cipher (2x2) in Python ===")

160.     print("Choose mode:\n1) Normal (A..Z)\n2) Revised (A..Z,0-9, space,.,!?)")

161.     mode = int(input("Enter 1 or 2: "))

162.

163.     if mode == 1:

164.         print("\n-- Normal Mode --")

165.         print("1) Use default key [[3,3],[2,5]]")

166.         print("2) Enter custom key letters")

167.         choice = int(input("Enter 1 or 2: "))

168.

169.         if choice == 1:

170.             key = [[3, 3], [2, 5]]

171.             print("Using default key [[D,D],[C,F]]")

172.         else:

173.             key = [[char\_to\_num(input("a (row0,col0): ").upper()),

174.                     char\_to\_num(input("b (row0,col1): ").upper())],

175.                    [char\_to\_num(input("c (row1,col0): ").upper()),

176.                     char\_to\_num(input("d (row1,col1): ").upper())]]

177.

178.         det = det2x2(key)

179.         detInv = inv\_mod26(det)

180.         print(f"det(key) mod 26 = {det}")

181.         if detInv == -1:

182.             print("This key is NOT invertible mod 26.")

183.             exit()

184.         else:

185.             print(f"det^-1 mod 26 = {detInv}")

186.

187.         plaintext = input("Enter plaintext: ")

188.         ciphertext = encrypt\_normal(plaintext, key)

189.         print("Ciphertext:", ciphertext)

190.         decrypted = decrypt\_normal(ciphertext, key)

191.         print("Decrypted: ", decrypted)

192.

193.     else:

194.         print("\n-- Revised Mode --")

195.         print("Character set:", REVISED\_CHARSET)

196.         print("1) Use default key [[1,2],[3,5]]")

197.         print("2) Enter custom key (characters from set)")

198.         choice = int(input("Enter 1 or 2: "))

199.

200.         if choice == 1:

201.             key = [[1, 2], [3, 5]]

202.             print("Using default key [[1,2],[3,5]]")

203.         else:

204.             key = [[r\_char\_to\_num(input("a (row0,col0): ").upper()),

205.                     r\_char\_to\_num(input("b (row0,col1): ").upper())],

206.                    [r\_char\_to\_num(input("c (row1,col0): ").upper()),

207.                     r\_char\_to\_num(input("d (row1,col1): ").upper())]]

208.

209.         a, b, c, d = key[0][0], key[0][1], key[1][0], key[1][1]

210.         det = modM(a \* d - b \* c, REVISED\_MOD)

211.         detInv = inv\_mod(det, REVISED\_MOD)

212.         print(f"det(key) mod {REVISED\_MOD} = {det}")

213.         if detInv == -1:

214.             print("This key is NOT invertible.")

215.             exit()

216.         else:

217.             print(f"det^-1 mod {REVISED\_MOD} = {detInv}")

218.

219.         plaintext = input("Enter plaintext: ")

220.         ciphertext = encrypt\_revised(plaintext, key)

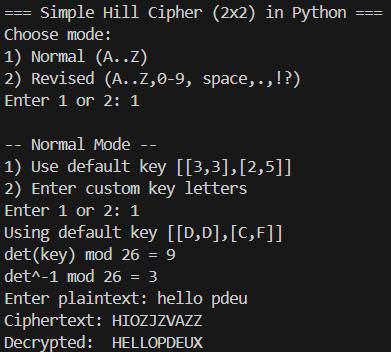
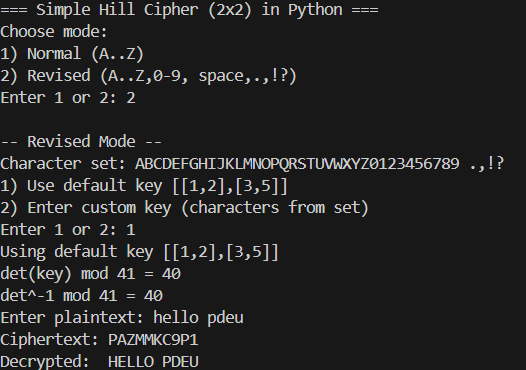
221.         print("Ciphertext:", ciphertext)

222.         decrypted = decrypt\_revised(ciphertext, key)

223.         print("Decrypted: ", decrypted)

224.

**Output:**

**Security Analysis:**

**Cryptanalysis**

* **Normal Hill Cipher:**
  + Polyalphabetic; encrypts pairs of letters using linear algebra mod 26.
  + More secure than Caesar/Vigenère if key is random and invertible, but still vulnerable to known-plaintext or ciphertext-only attacks if block structure is detected. Small block size (2x2) makes attacks easier.
* **Revised Hill Cipher:**
  + Supports broad charset for more realistic data.
  + Adds dynamic matrix multiplier per block (blockIndex feature), increasing complexity.
  + Slightly harder to break by hand, not fundamentally secure against modern cryptanalysis or brute-force with short texts.

**Design**

* Clean separation by mode/charset.
* Utility functions for matrix math and character conversions.
* Checks key invertibility before use (critical for decryption).
* Revised mode features per-block dynamic transformation (makes it harder to brute-force).
* CLI interface and flexible key input for experimentation.

**Performance**

* All methods operate in linear time () on message length.
* Matrix operations and modular arithmetic are fast for typical texts.
* Designed for efficiency and educational value.

**Sign of Faculty Date of performance**

**EXPERIMENT-7**

**AIM:** Study and Use of RSA algorithm (encryption and decryption)

**Introduction :** RSA (Rivest–Shamir–Adleman) is a [public-key cryptosystem](https://en.wikipedia.org/wiki/Public-key_cryptography), one of the oldest widely used for secure data transmission. The [initialism](https://en.wikipedia.org/wiki/Initialism) "RSA" comes from the surnames of [Ron Rivest](https://en.wikipedia.org/wiki/Ron_Rivest), [Adi Shamir](https://en.wikipedia.org/wiki/Adi_Shamir) and [Leonard Adleman](https://en.wikipedia.org/wiki/Leonard_Adleman), who publicly described the algorithm in 1977. An equivalent system was developed secretly in 1973 at [Government Communications](https://en.wikipedia.org/wiki/Government_Communications_Headquarters)

|  |  |  |  |
| --- | --- | --- | --- |
| [Headquarters](https://en.wikipedia.org/wiki/Government_Communications_Headquarters) (GCHQ), | the British [signals](https://en.wikipedia.org/wiki/Signals_intelligence) | [intelligence](https://en.wikipedia.org/wiki/Signals_intelligence) agency, by the | English |
| mathematician [Clifford](https://en.wikipedia.org/wiki/Clifford_Cocks) | [Cocks](https://en.wikipedia.org/wiki/Clifford_Cocks). That | system was [declassified](https://en.wikipedia.org/wiki/Classified_information) in | 1997. |

In a public-key [cryptosystem](https://en.wikipedia.org/wiki/Cryptosystem), the [encryption key](https://en.wikipedia.org/wiki/Encryption_key) is public and distinct from the [decryption](https://en.wikipedia.org/wiki/Decryption_key) [key,](https://en.wikipedia.org/wiki/Decryption_key) which is kept secret (private). An RSA user creates and publishes a public key based on two large [prime numbers](https://en.wikipedia.org/wiki/Prime_number), along with an auxiliary value. The prime numbers are kept secret. Messages can be encrypted by anyone, via the public key, but can only be decrypted by someone who knows the private key. The security of RSA relies on the practical difficulty of [factoring](https://en.wikipedia.org/wiki/Factorization) the product of two large [prime numbers,](https://en.wikipedia.org/wiki/Prime_number) the "[factoring problem](https://en.wikipedia.org/wiki/Factoring_problem)". Breaking RSA encryption is known as the [RSA](https://en.wikipedia.org/wiki/RSA_problem) [problem.](https://en.wikipedia.org/wiki/RSA_problem) Whether it is as difficult as the factoring problem is an open question. There are no published methods to defeat the system if a large enough key is used.

**Example :** First, let’s assume you calculated your keys as follows:

1. p=17 and q =7. Notice 17 and 7 are both prime numbers
2. n= 17 x 7 = 119
3. f(n) = (17-1)(7-1)=96
4. e=11, notice that gcd(96,11)=1 and 1<11<96
5. d=35

The keys are:

* + private key: {35,119}
  + public key: {11,119}

Now, you published somehow your public key and I want to send you a message only you can read. The message I want to send you is M=21. Notice that you can always find a numeric representation for any message. At the end of the day, all data in a computer is represented as numbers.

*C* = *Me* mod *n=2111 mod 119 = 98*

When you receive the encrypted message C=45, you use your private key to decrypt it.

*M* = *Cd* mod *n=9835 mod 119 = 21*

**Code:**

1. def modInverse(e, phi):

2.     for d in range(2, phi):

3.         if (e \* d) % phi == 1:

4.             return d

5.     return -1

6.

7. def generateKeys():

8.     p = 3

9.     q = 11

10.

11.     n = p \* q

12.     phi = (p - 1) \* (q - 1)

13.

14.     e = 0

15.     for e in range(2, phi):

16.         if gcd(e, phi) == 1:

17.             break

18.

19.     d = modInverse(e, phi)

20.

21.     return e, d, n

22. def gcd(a, b):

23.     while b != 0:

24.         a, b = b, a % b

25.     return a

26. def encrypt(M,e,n):

27.

28.     x = M \*\* e

29.     C=x%n

30.     return C

31.

32. def decrypt(C,d,n):

33.     y =C \*\* d

34.     M =y%n

35.     return M

36.

37. if \_\_name\_\_ == "\_\_main\_\_":

38.

39.     e, d, n = generateKeys()

40.

41.     print(f"Public Key (e, n): ({e}, {n})")

42.     print(f"Private Key (d, n): ({d}, {n})")

43.

44.     M = 16

45.

46.     print(f"Original Message: {M}")

47.

48.     C = encrypt(M, e, n)

49.     print(f"Encrypted Message: {C}")

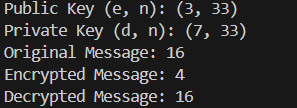
50.

51.     decrypted = decrypt(C, d, n)

52.     print(f"Decrypted Message: {decrypted}")

53.

**Output:**



**Security Analysis:**

**Cryptanalysis**

* Uses tiny values for  and , making it entirely insecure—factorization is instant.
* Encrypts only numbers <  (33), not practical plaintext or text blocks.
* Real-world RSA requires large primes (hundreds of digits), padding schemes, and secure randomness.
* **Vulnerable** to all modern attacks; just for concept demonstration.

**Design**

* Functions are modular (key generation, encryption, decryption, modular inverse, gcd).
* Lacks input validation, message conversion (plaintext to int), padding, or error handling.
* Euler’s phi is correctly calculated; key pair returned for example use.

**Performance**

* Very fast for these small numbers, but not implementable for real RSA keys (lacks optimized math).
* Exponentiation is naive (no modular exponentiation, which is required for big numbers).

**Sign of Faculty Date of performance**

**EXPERIMENT-8**

**AIM:** Study and implement a program of the Digital Signature with RSA algorithm (Reverse RSA)

**Introduction :** Assume that there is a sender (A) and a receiver (B). A wants to send a message (M) to B along with the digital signature (DS) calculated over the message.

Step-1 : Sender A uses SHA-1 Message Digest Algorithm to calculate the message digest (MD1) over the original message M.

Step-2 : A now encrypts the message digest with its private key. The output of this process is called Digital Signature (DS) of A.

Step-3 : Now sender A sends the digital signature (DS) along with the original message (M) to B.

Step-4 : When B receives the Original Message(M) and the Digital Signature(DS) from A, it first uses the same message-digest algorithm as was used by A and calculates its own Message Digest (MD2) for M.

Step-5 : Now B uses A’s public key to decrypt the digital signature because it was encrypted by A’s private key. The result of this process is the original Message Digest (MD1) which was calculated by A.

Step-6 : If MD1==MD2, the following facts are established as follows.

* B accepts the original message M as the correct, unaltered message from A.
* It also ensures that the message came from A and not someone posing as A.

**Code:**

**Output:**

**Security Analysis:** The student must perform a security analysis of the algorithm mentioned above. Analysis includes a) Cryptanalysis, b)Design analysis, c)Performance analysis

**Sign of Faculty Date of performance**

**EXPERIMENT-9**

**AIM:** Study and Use of Diffie-Hellman Key Exchange

**Introduction :** The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

* For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime P and G (a primitive root of P) and two private values a and b.
* P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

Step-by-Step explanation is as follows:

|  |  |
| --- | --- |
| **Alice** | **Bob** |
| Public Keys available = P, G | Public Keys available = P, G |
| Private Key Selected = a | Private Key Selected = b |
| Key generated =  𝑥=𝐺𝑎𝑚𝑜𝑑P | Key generated =  𝑦=𝐺𝑏𝑚𝑜𝑑𝑃 |
| Exchange of generated keys takes place | |
| Key received = y | key received = x |
| Generated Secret Key =  𝑘𝑎=𝑦𝑎𝑚𝑜𝑑𝑃 | Generated Secret Key =  𝑘𝑏=𝑥𝑏𝑚𝑜𝑑𝑃 |
| Algebraically, it can be shown that  𝑘𝑎=𝑘𝑏 | |
| Users now have a symmetric secret key to encrypt | |

**Example :**

Step 1: Alice and Bob get public numbers P = 23, G = 9 Step 2: Alice selected a private key a = 4 and

Bob selected a private key b = 3

Step 3: Alice and Bob compute public values Alice: x =(9^4 mod 23) = (6561 mod 23) = 6

Bob: y = (9^3 mod 23) = (729 mod 23) = 16 Step 4: Alice and Bob exchange public numbers Step 5: Alice receives public key y =16 and

Bob receives public key x = 6

Step 6: Alice and Bob compute symmetric keys Alice: ka = y^a mod p = 65536 mod 23 = 9 Bob: kb = x^b mod p = 216 mod 23 = 9

Step 7: 9 is the shared secret.

**Code:**

1. import random

2.

3. def R1(g, x, p):

4.     return (g \*\* x) % p

5.

6. def R2(g, y, p):

7.     return (g \*\* y) % p

8.

9. def **Kalice**(R2, x, p):

10.     return (R2 \*\* x) % p

11.

12. def **Kbob**(R1, y, p):

13.     return (R1 \*\* y) % p

14.

15. def comp(K1, K2):

16.     return (K1 == K2)

17.

18. def unqxy(P):

19.     x = random.randrange(P)

20.     y = random.randrange(P)

21.     return x, y

22.

23. if \_\_name\_\_ == '\_\_main\_\_':

24.

25.     P = 19

26.     G = 3

27.

28.     X, Y = unqxy(P)

29.

30.     r1 = R1(G, X, P)

31.     print(f"Alice's private key (X): {X}")

32.     print(f"Alice's public key (r1): {r1}")

33.

34.     r2 = R2(G, Y, P)

35.     print(f"Bob's private key (Y): {Y}")

36.     print(f"Bob's public key (r2): {r2}")

37.

38.     K1 = **Kalice**(r2, X, P)

39.     print(f"Alice's shared secret (K1): {K1}")

40.

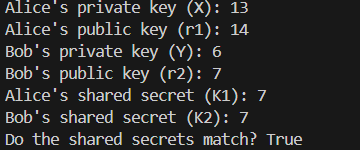
41.     K2 = **Kbob**(r1, Y, P)

42.     print(f"Bob's shared secret (K2): {K2}")

43.     print(f"Do the shared secrets match? {comp(K1, K2)}")

44.

**Output:**

****

**Security Analysis:**

**Cryptanalysis**

* Uses very small prime (), extremely insecure—trivial to brute-force.
* No vulnerability to passive eavesdropping (if using large, real primes), but this toy version is only educational.
* Real Diffie-Hellman requires primes 1024+ bits for security; this is instantly crackable.

**Design**

* Functions are modular and clear (generation, computation, comparison).
* Private keys generated randomly in valid range.
* Each step is explicit, ideal for classroom demonstrations.

**Performance**

* Efficient and fast for these small values.
* Uses naive exponentiation; for realistic primes, modular exponentiation methods needed.

**Sign of Faculty Date of performanc**

**EXPERIMENT-10**

1. **AIM:** Write a program to encrypt the plaintext with the given key.

E.g. plaintext GRONSFELD with the key 1234. Add 1 to G to get H (the letter 1 rank after G is H in the alphabet), then add 2 to C or E (the letter 2 ranks after C is E), and so on. Use smallest letter from plaintext as filler. In the given example last three letters are filler.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Plain letter | G | R | O | N | S | F | E | L | D | D | D | D |
| Key (repeated)1 | | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| Cipher Letter H | | T | R | R | T | H | H | P | E | F | G | H |

The encrypted message is HTRRTHHPEFGH.

Input : Plaintext (in capital letters only), key (in numbers only) Output : ciphertext (in capital letters only)

|  |  |
| --- | --- |
| Test Case 1 | Test Case 2 |

**Code:**

1. # plaintext = "GRONSFELD"

2. # key = 1234

3. # ciphertext = "HTRRTHHPEFGH"

4.

5. def encrypt(p,k):

6.     listP =list(p)

7.     last=listP[-1]

8.     listK =list(k)

9.     **Cipher**=""

10.     lp = len(listP)

11.     lk =len(listK)

12.     if(lp % lk != 0):

13.         rep=lk - (lp % lk)

14.         for i in range(0,rep):

15.             listP.append(last)

16.     print("after padding:")

17.     print(listP)

18.     for i in range(0,lp):

19.        **Cipher** = **Cipher** + (chr)(ord(listP[i])+ int(listK[i % len(k)]))

20.     return **Cipher**

21.

22. if \_\_name\_\_ == "\_\_main\_\_":

23.     # plaintext = input("Enter your plaintext:")

24.     # key = input("Enter your key:")

25.     plaintext = "GRONSFELD"

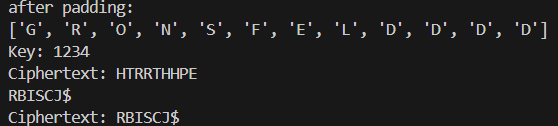
26.     key = "1234"

27.     ciphertext = encrypt(plaintext,key)

28.     print(f"Key: {key}")

29.     print(f"Ciphertext: {ciphertext}")

**Output:**

****

**Security Analysis:**

**Cryptanalysis**

* **Security:**
  + Very weak; similar to a multi-digit Caesar shift.
  + Directly reversible if key is known; susceptible to frequency analysis and brute-force attacks.
  + Padding by duplicating the last character can reveal plaintext structure and weaken security.
* **Use case:** Suitable only for demos and exercises, not for practical protection.

**Design**

* Modular and readable function for encryption.
* Padding ensures input matches key multiples, which can aid analysis.
* No input validation—non-digit keys or non-ASCII characters can cause errors.

**Performance**

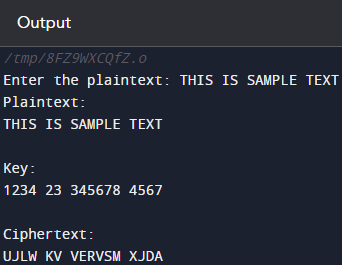
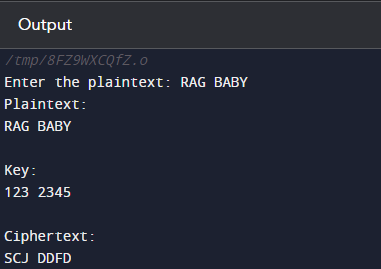
* Fast (); memory use is minimal and scales linearly.

1. **AIM:** Encrypt the input words PLAINTEXT= RAG BABY to obtain CIPHERTEXT = SCJ DDFD

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Plain letter | R | A | G |  | B | A | B | Y |
| word i | 1 | 1 | 1 |  | 2 | 2 | 2 | 2 |
| letter j | 0 | 1 | 2 |  | 0 | 1 | 2 | 3 |
| i+j | 1 | 2 | 3 |  | 2 | 3 | 4 | 5 |
| Cipher letter | S | C | J |  | D | D | F | D |

Input : Plaintext (in capital letters only, upto nine words only) Output : ciphertext (in capital letters only)

Note : Students can use string.h if required.



Test Case 2

Test case 1

Output : ciphertext (in capital letters only)

**Code:**

1. def encryption(p):

2.     listw = p.split()

3.     wordlen = len(p.split())

4.     cipher =""

5.     for i in range(wordlen):

6.         for j in range(len(listw[i])):

7.             encrypted\_char = chr((i + j) + ord(p[j]))

8.             cipher += encrypted\_char

9.

10.

11.     return cipher

12.

13. if \_\_name\_\_ == "\_\_main\_\_":

14.     PLAIN = "RAG BABY"

15.     # CIPHER = "SCJ DDFD"

16.     CIPHER = encryption(PLAIN)

17.     print(f"Ciphertext: {CIPHER}")

**Output:**

****

**Security Analysis:**

**Cryptanalysis**

* **Security:**
  + Not a standard or secure cipher; no key, relies only on word/character position for shifting.
  + Highly predictable and easy to reverse or brute-force, especially for short words.
  + Not properly reversible—some characters may map to non-printable ASCII, plain-text recovery is uncertain.

**Design**

* Experimental function to show dynamic shifts based on text structure.
* Clear looping structure and readable code.
* Lacks input checks (may error on spaces, special chars).
* No decryption method—output loses original message structure.

**Performance**

* Linear time (), fast for any input size, minimal memory.

**Sign of Faculty Date of performance**

**EXPERIMENT-11**

1. **AIM: Mini Project** To understand the need for Post-Quantum Cryptography.

To understand the need for post-quantum cryptography, analyse how quantum computing threatens traditional cryptographic schemes.

* + Explain the threat posed by quantum computers to RSA, ECC, and other traditional cryptosystems.
  + Understand the basics of post-quantum cryptographic schemes.
  + Evaluate security properties and draw comparisons with classical cryptography.

# Quantum Threats:

* + **Shor's Algorithm:** Breaks RSA, ECC efficiently.
  + **Grover's Algorithm:** Weakens symmetric key strength (AES-128 → AES- 64-bit equivalent).

# NIST PQC Standardization Effort:

* + Leading candidates: CRYSTALS-Kyber (encryption), CRYSTALS-Dilithium (signature), Falcon, etc.

**Sign of Faculty Date of performance**