1. Write a C program to Encrypt the message “meet me at the usual place at ten rather than eight oclock” using the Hill cipher with the key. 9 4 5 7 a. Show your calculations and the result. b. Show the calculations for the corresponding decryption of the ciphertext to recover the original plaintext. #!/usr/bin/env python3
2. """
3. Recover Hill-cipher key from known plaintext-ciphertext pairs.
4. This version automatically searches for an invertible plaintext matrix P
5. by trying combinations of n plaintext n-grams.
6. Demo simulates a true key, encrypts some chosen plaintext n-grams to produce
7. observed ciphertext n-grams, and then attempts to recover the key.
8. """
9. from typing import List, Tuple
10. import itertools
11. import math
12. MOD = 26
13. # ---------- modular helpers ----------
14. def egcd(a: int, b: int) -> Tuple[int,int,int]:
15. if b == 0:
16. return a, 1, 0
17. g, x1, y1 = egcd(b, a % b)
18. return g, y1, x1 - (a // b) \* y1
19. def modinv(a: int, m: int = MOD) -> int:
20. g, x, \_ = egcd(a % m, m)
21. if g != 1:
22. raise ValueError(f"No modular inverse for {a} mod {m}")
23. return x % m
24. # ---------- matrix helpers ----------
25. def mat\_mul(A: List[List[int]], B: List[List[int]], mod: int = MOD) -> List[List[int]]:
26. r, k = len(A), len(A[0])
27. k2, c = len(B), len(B[0])
28. assert k == k2, "Incompatible shapes for multiplication"
29. R = [[sum(A[i][t]\*B[t][j] for t in range(k)) % mod for j in range(c)] for i in range(r)]
30. return R
31. def mat\_copy(A: List[List[int]]) -> List[List[int]]:
32. return [row[:] for row in A]
33. # ---------- modular matrix inverse via Gauss-Jordan ----------
34. def mat\_mod\_inv(A: List[List[int]], mod: int = MOD) -> List[List[int]]:
35. n = len(A)
36. # build augmented matrix [A | I]
37. aug = [ [ (A[i][j] % mod) for j in range(n) ] + [ 1 if i==j else 0 for j in range(n) ] for i in range(n) ]
38. row = 0
39. for col in range(n):
40. # find pivot
41. pivot = None
42. for r in range(row, n):
43. if aug[r][col] % mod != 0:
44. pivot = r
45. break
46. if pivot is None:
47. raise ValueError("Matrix is singular (not invertible) modulo {}".format(mod))
48. # swap
49. if pivot != row:
50. aug[row], aug[pivot] = aug[pivot], aug[row]
51. # normalize pivot row
52. pivot\_val = aug[row][col] % mod
53. inv\_pivot = modinv(pivot\_val, mod)
54. aug[row] = [(val \* inv\_pivot) % mod for val in aug[row]]
55. # eliminate other rows
56. for r in range(n):
57. if r != row:
58. factor = aug[r][col] % mod
59. if factor != 0:
60. aug[r] = [ (aug[r][c] - factor \* aug[row][c]) % mod for c in range(2\*n) ]
61. row += 1
62. if row == n:
63. break
64. inv = [ row[n:] for row in aug ]
65. return inv
66. # ---------- text <-> numeric conversions ----------
67. def text\_to\_blocks(text: str, n: int) -> List[List[int]]:
68. s = ''.join([c for c in text.upper() if c.isalpha()])
69. # pad with 'X' if required
70. if len(s) % n != 0:
71. s += 'X' \* (n - (len(s) % n))
72. blocks = []
73. for i in range(0, len(s), n):
74. blocks.append([ord(ch) - ord('A') for ch in s[i:i+n]])
75. return blocks
76. def blocks\_to\_matrix\_cols(blocks: List[List[int]]) -> List[List[int]]:
77. if not blocks:
78. return []
79. n = len(blocks[0])
80. m = len(blocks)
81. M = [[0]\*m for \_ in range(n)]
82. for j, blk in enumerate(blocks):
83. for i in range(n):
84. M[i][j] = blk[i] % MOD
85. return M
86. def matrix\_cols\_to\_blocks(M: List[List[int]]) -> List[List[int]]:
87. n = len(M)
88. m = len(M[0])
89. blocks = []
90. for j in range(m):
91. blocks.append([M[i][j] % MOD for i in range(n)])
92. return blocks
93. def blocks\_to\_ngrams(blocks: List[List[int]]) -> List[str]:
94. return [''.join(chr(x + ord('A')) for x in blk) for blk in blocks]
95. # ---------- invertibility test ----------
96. def det\_2x2\_mod26(P: List[List[int]]) -> int:
97. # only for 2x2 matrix; for general n you would compute determinant modulo 26
98. return (P[0][0]\*P[1][1] - P[0][1]\*P[1][0]) % MOD
99. def is\_invertible\_mod26(P: List[List[int]]) -> bool:
100. # compute determinant (works for n x n by trying mat\_mod\_inv, but this is cheap for 2x2)
101. try:
102. \_ = mat\_mod\_inv(P, MOD)
103. return True
104. except ValueError:
105. return False
106. # ---------- key recovery (tries combinations) ----------
107. def recover\_key\_from\_pairs(plain\_ngrams: List[str], cipher\_ngrams: List[str], n: int) -> List[List[int]]:
108. if len(plain\_ngrams) < n or len(cipher\_ngrams) < n:
109. raise ValueError("Need at least n pairs (here n={})".format(n))
110. # try all combinations of indices of size n
111. indices = range(len(plain\_ngrams))
112. for combo in itertools.combinations(indices, n):
113. # build P and C from these indices
114. P\_blocks = text\_to\_blocks(''.join(plain\_ngrams[i] for i in combo), n)
115. C\_blocks = text\_to\_blocks(''.join(cipher\_ngrams[i] for i in combo), n)
116. P = blocks\_to\_matrix\_cols(P\_blocks) # n x n
117. C = blocks\_to\_matrix\_cols(C\_blocks)
118. # attempt invert
119. try:
120. P\_inv = mat\_mod\_inv(P, MOD)
121. except ValueError:
122. continue # not invertible, try next combo
123. K = mat\_mul(C, P\_inv, MOD)
124. return K # first successful recovery
125. raise ValueError("No invertible plaintext matrix P found among given pairs.")
126. # ---------- demonstration ----------
127. def demo():
128. # True (unknown) key for 2x2 example
129. true\_key = [[9, 4],
130. [5, 7]]
131. n = 2
132. # Attacker collects (or chooses) many plaintext digrams and observes ciphertext digrams.
133. # We'll simulate: create a list of plaintext digrams and compute ciphertext digrams using true\_key.
134. full\_plaintext = "MEETMEATTHEUSUALPLACEATTENRATHERTHANEIGHTOCLOCK"
135. p\_blocks = text\_to\_blocks(full\_plaintext, n) # list of digraph numeric lists
136. # Build matrix P\_all columns
137. P\_all = blocks\_to\_matrix\_cols(p\_blocks)
138. # Encrypt using true key: C\_all = K \* P\_all
139. C\_all = mat\_mul(true\_key, P\_all, MOD)
140. c\_blocks = matrix\_cols\_to\_blocks(C\_all)
141. # Convert blocks to digram strings
142. plain\_ngrams = blocks\_to\_ngrams(p\_blocks)
143. cipher\_ngrams = blocks\_to\_ngrams(c\_blocks)
144. # Show some observed pairs
145. print("Observed plaintext n-grams (first 10):", plain\_ngrams[:10])
146. print("Corresponding ciphertext n-grams (first 10):", cipher\_ngrams[:10])
147. # Now attempt recovery using these observed pairs (attacker doesn't know true\_key)
148. try:
149. recovered\_key = recover\_key\_from\_pairs(plain\_ngrams, cipher\_ngrams, n)
150. print("\nRecovered key (mod 26):")
151. for row in recovered\_key:
152. print(row)
153. print("\nTrue key was:")
154. for row in true\_key:
155. print([x % MOD for x in row])
156. except ValueError as e:
157. print("Key recovery failed:", e)
158. if \_\_name\_\_ == "\_\_main\_\_":
159. demo()
160. 