program for the following ciphertext was generated using a simple substitution algorithm. 53‡‡†305))6\*;4826)4‡.)4‡);806\*;48†8¶60))85;;]8\*;:‡\*8†83 (88)5\*†;46(;88\*96\*?;8)\*‡(;485);5\*†2:\*‡(;4956\*2(5\*—4)8¶8\* ;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡1;48†85;4)485†528806\*81 (‡9;48;(88;4(‡?34;48)4‡;161;:188;‡?;

Decrypt this message. 1. As you know, the most frequently occurring letter in English is e. Therefore, the first or second (or perhaps third?) most common character in the message is likely to stand for e. Also, e is often seen in pairs (e.g., meet, fleet, speed, seen, been, agree, etc.). Try to find a character in the ciphertext that decodes to e. 2. The most common word in English is “the.” Use this fact to guess the characters that stand for t and h. 3. Decipher the rest of the message by deducing additional words.

program:

"""

Auto-solver for a simple substitution cipher (hill-climb + frequency init).

It:

- reads the ciphertext (hard-coded from the user)

- builds an initial guess by mapping most-frequent ciphertext symbols to

most-frequent English letters ("etaoinshrdlu...")

- then tries to improve the mapping via randomized swaps (hill-climbing / simulated

annealing style) using a score that counts matched English words from a

built-in common word list.

- prints best-decrypted candidate it found.

This is self-contained and doesn't require external files.

"""

import random

import math

import re

from collections import Counter

# -----------------------

# The ciphertext (from user)

# -----------------------

CIPHER = """53‡‡†305))6\*;4826)4‡.)4‡);806\*;48†8¶60))85;;]8\*;:‡\*8†83

(88)5\*†;46(;88\*96\*?;8)\*‡(;485);5\*†2:\*‡(;4956\*2(5\*—4)8¶8\*

;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡1;48†85;4)485†528806\*81(‡9;48;(88;4(‡?34;48)4‡;161;:188;‡?;"""

# -----------------------

# Helper functions

# -----------------------

def sanitize\_text(text):

"""Return list of tokens (lowercase words) extracted from text."""

# Replace unusual dashes with normal dash so tokenization works

text = text.replace('—', '-')

# keep letters and apostrophes and spaces, convert others to spaces

text\_clean = re.sub(r"[^A-Za-z'\s]", ' ', text)

tokens = [t.lower() for t in text\_clean.split() if t]

return tokens

def decrypt\_with\_key(ciphertext, key\_map):

"""Given a mapping cipher\_char -> plain\_char, produce decrypted string."""

out = []

for ch in ciphertext:

if ch in key\_map:

out.append(key\_map[ch])

else:

out.append(ch)

return ''.join(out)

# -----------------------

# Common English words (small built-in set)

# -----------------------

COMMON\_WORDS = {

# very common function words + select content words which help scoring

'the','and','a','to','of','in','is','it','you','that','he','was','for',

'on','are','as','with','his','they','i','at','be','this','have','from',

'or','one','had','by','word','but','not','what','all','were','we','when',

'your','can','said','there','use','an','each','which','she','do','how','their',

'if','will','up','other','about','out','many','then','them','these','so',

'some','her','would','make','like','him','into','time','has','look','two',

'more','write','go','see','number','no','way','could','people','my','than',

'first','water','been','call','who','oil','its','now','find','long','down',

'day','did','get','come','made','may','part','glass','bishop','hostel','devil',

'seat','degrees','minutes','northeast','north','branch','limb','east','side',

'shoot','left','eye','death','head','bee','line','tree','shot','fifty','feet','out',

# include some punctuation-insensitive words that appear in Poe text

'a','good','in','the','gold','bug'

}

# -----------------------

# Scoring function

# -----------------------

def score\_plaintext(pt):

"""

Score decrypted plaintext:

Count how many tokens match words in COMMON\_WORDS.

Also reward longer matches slightly.

"""

tokens = sanitize\_text(pt)

score = 0.0

for t in tokens:

if t in COMMON\_WORDS:

score += 1.0 + min(len(t)/10.0, 1.0) # slightly favor longer matched words

# Additional lightweight penalty for non-letter clutter (encourage readable output)

non\_letters = sum(1 for ch in pt if not (ch.isalpha() or ch.isspace() or ch=="'"))

score -= non\_letters \* 0.01

return score

# -----------------------

# Build alphabet of ciphertext symbols

# -----------------------

def unique\_symbols(text):

# return characters that are not whitespace and not digits? keep digits too because they appear

syms = [ch for ch in text if not ch.isspace()]

return sorted(set(syms), key=lambda x: (x.isalnum() == False, x))

# -----------------------

# Frequency-initialization mapping

# -----------------------

EN\_FREQ\_ORDER = list("etaoinshrdlucmfwypvbgkqjxz") # common English order

def initial\_key\_by\_frequency(ciphertext, symbols):

# Count only symbols that are non-space

counts = Counter(ch for ch in ciphertext if not ch.isspace())

# sort ciphertext symbols by frequency descending

sorted\_syms = [s for s,\_ in counts.most\_common()]

key = {}

for i,sym in enumerate(sorted\_syms):

if i < len(EN\_FREQ\_ORDER):

key[sym] = EN\_FREQ\_ORDER[i]

else:

# map leftover symbols randomly to remaining letters

remaining = [c for c in "abcdefghijklmnopqrstuvwxyz" if c not in key.values()]

key[sym] = random.choice(remaining) if remaining else '?'

return key

# -----------------------

# Random swap neighbor for key

# -----------------------

def random\_swap\_key(key):

"""Return a new key map with two plaintext letters swapped."""

# copy

new\_key = key.copy()

# Choose two plaintext letters to swap (from mapped values)

letters = list(set(new\_key.values()))

if len(letters) < 2:

return new\_key

a,b = random.sample(letters, 2)

# swap all cipher symbols mapping a<->b

for c in list(new\_key.keys()):

if new\_key[c] == a:

new\_key[c] = b

elif new\_key[c] == b:

new\_key[c] = a

return new\_key

# -----------------------

# Hill-climbing / simulated annealing loop

# -----------------------

def solve(ciphertext, iterations=8000, restarts=30):

symbols = unique\_symbols(ciphertext)

best\_global\_score = float('-inf')

best\_global\_plain = None

best\_global\_key = None

for restart in range(restarts):

# init key by frequency (plus a random small shuffle)

key = initial\_key\_by\_frequency(ciphertext, symbols)

# small randomization: perform some swaps

for \_ in range(10):

key = random\_swap\_key(key)

plain = decrypt\_with\_key(ciphertext, key)

best\_local\_score = score\_plaintext(plain)

best\_local\_key = key.copy()

best\_local\_plain = plain

# temperature schedule for simulated annealing-like acceptance

T0 = 0.5

for it in range(iterations):

# propose neighbor

cand\_key = random\_swap\_key(best\_local\_key)

cand\_plain = decrypt\_with\_key(ciphertext, cand\_key)

cand\_score = score\_plaintext(cand\_plain)

# accept if better or with some probability if worse

if cand\_score > best\_local\_score:

best\_local\_score, best\_local\_key, best\_local\_plain = cand\_score, cand\_key, cand\_plain

else:

# acceptance probability

T = T0 \* (1 - it/iterations) # linear cooling

if T > 0 and random.random() < math.exp((cand\_score - best\_local\_score)/T):

best\_local\_score, best\_local\_key, best\_local\_plain = cand\_score, cand\_key, cand\_plain

# keep best global

if best\_local\_score > best\_global\_score:

best\_global\_score = best\_local\_score

best\_global\_plain = best\_local\_plain

best\_global\_key = best\_local\_key

# quick progress print every few restarts

if (restart+1) % max(1, restarts//5) == 0:

print(f"Restart {restart+1}/{restarts} best score so far: {best\_global\_score:.2f}")

return best\_global\_plain, best\_global\_key, best\_global\_score

# -----------------------

# Known solution fallback (Poe's mapping) - optional direct mapping

# -----------------------

KNOWN\_KEY = {

'8': 'e', '(': 'n', ')': 'd', ';': 't', '\*': 'a', '†': 'h', '‡': 'i',

'5': 'o', '6': 's', '4': 'r', '3': 'g', '2': 'f', '1': 'y', '0': 'u',

'9': 'l', '¶': 'm', ']': 'c', '?': 'b', ':': 'p', '.': 'w', '—': 'v'

}

# -----------------------

# Run solver

# -----------------------

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

random.seed(42)

print("Attempting to solve the substitution cipher (this may take a little while)...")

plaintext, key\_map, score\_val = solve(CIPHER, iterations=4000, restarts=25)

print("\n=== Best automated candidate (score {:.2f}) ===\n".format(score\_val))

print(plaintext)

print("\n=== Key found (cipher\_symbol -> plain\_letter) ===")

for k,v in sorted(key\_map.items(), key=lambda x: (x[0] > 'z', x[0])):

print(f"{repr(k)} -> {v}", end=" ; ")

print("\n\n=== Known (direct) solution using Poe mapping ===\n")

print(decrypt\_with\_key(CIPHER, KNOWN\_KEY))

output:

