2211CS020098
AIML - Beta
NLP(Coding) HOLIDAY ASSIGNMENT

1] Correct the Search Query

Code:

```
import re
from collections import Counter

word_corpus = """
going to china who was the first president of india winner of the match food in america
""".split()

WORD_COUNTS = Counter(word_corpus)

def edits1(word):
   """Return all strings that are one edit away from the input word."""
   letters = 'abcdefghijklmnopqrstuvwxyz'
   splits = [(word[:i], word[i:]) for i in range(len(word) + 1)]
   deletes = [L + R[1:] for L, R in splits if R]
   transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R) > 1]
   replaces = [L + c + R[1:] for L, R in splits if R for c in letters]
   inserts = [L + c + R for L, R in splits for c in letters]
   return set(deletes + transposes + replaces + inserts)

def known(words):
   """Return the subset of words that are in the corpus."""
   return set(w for w in words if w in WORD_COUNTS)

def candidates(word):
   """Generate possible spelling corrections for the word."""
   return (known([word]) or
         known(edits1(word)) or
         known(e2 for e1 in edits1(word) for e2 in edits1(e1)) or
         [word])

def correct(word):
   """Find the best correction for a word."""
   return max(candidates(word), key=WORD_COUNTS.get)

def correct_query(query):
   """Correct the spelling of a full query."""
   words = query.split()
   corrected_words = [correct(word) for word in words]
   return ' '.join(corrected_words)

def main():
   n = int(input().strip())
   queries = [input().strip() for _ in range(n)]

   corrected_queries = [correct_query(query) for query in queries]

   for query in corrected_queries:
      print(query)
```

```
if __name__ == "__main__":
    main()
```

Explanation:

Scoring:

Scoring is proportional to the answers you compute correctly.
Score for each test case = (100*correctAnswers / TotalNumberOfTests)
Total Score = Average of Scores for all test cases that are run on your submission.

2] Deterministic Url and HashTag Segmentation

Code:

```
import re

def load_word_list(file_path="words.txt"):
    with open(file_path, "r") as f:
        words = set(word.strip().lower() for word in f)
    return words

def is_number(s):
    try:
        float(s)
        return True
    except ValueError:
        return False

def segment_string(s, word_set):
    n = len(s)
    if n == 0:
        return []

    dp = [None] * (n + 1)
    dp[n] = []

    for i in range(n - 1, -1, -1):
        for j in range(i + 1, n + 1):
            substring = s[i:j]
            if substring in word_set or is_number(substring):
                remainder = dp[j]
                if remainder is not None:
                    candidate = [substring] + remainder
                    if dp[i] is None or len(" ".join(candidate)) > len(" ".join(dp[i])):
                        dp[i] = candidate

    return dp[0] if dp[0] is not None else [s]

def preprocess_input(s):
    s = s.lower()
    if s.startswith("#"):
        return s[1:]
    elif s.startswith("www."):
        s = s[4:]
    s = re.sub(r"\.[a-z]{2,3}(\.[a-z]{2})?$", "", s)
    return s
```

```
def main():
    word_set = load_word_list()

    n = int(input().strip())
    inputs = [input().strip() for _ in range(n)]

    results = []
    for inp in inputs:
        preprocessed = preprocess_input(inp)
        segmented = segment_string(preprocessed, word_set)
        results.append(" ".join(segmented))

    for result in results:
        print(result)

if __name__ == "__main__":
    main()
```

Explanation:

Steps:

- Load the Word List:
  - Read the words.txt file into a Python set for fast lookup.
  - Ensure all words are in lowercase for consistency.
- Preprocessing:
  - Remove prefixes like www. and # from domain names and hashtags.
  - Strip domain extensions like .com, .in, etc., using a regex.
- Greedy Segmentation:
  - Use a dynamic programming approach to split the string into valid tokens.
  - Start from the end of the string and attempt to segment it into words or numbers.
  - Always prefer the longest possible match to ensure the best segmentation.
- Check for Valid Tokens:
  - A valid token is a word from the dictionary or a number.

3] Disambiguation: Mouse vs Mouse

Code:

```
import re

COMPUTER_MOUSE_KEYWORDS = {
    "input", "device", "click", "cursor", "keyboard", "screen", "usb", "wireless", "bluetooth", "dpi",
"computer", "software"
}

ANIMAL_MOUSE_KEYWORDS = {
    "tail", "fur", "rodent", "species", "habitat", "cheese", "predator", "wild", "mammal", "genome",
"whiskers"
}

def classify_sentence(sentence):
    sentence = sentence.lower()
```

```
    if any(keyword in sentence for keyword in COMPUTER_MOUSE_KEYWORDS):
        return "computer-mouse"
    if any(keyword in sentence for keyword in ANIMAL_MOUSE_KEYWORDS):
        return "animal"
    return "animal"

def main():
    n = int(input().strip())
    sentences = [input().strip() for _ in range(n)]

    results = [classify_sentence(sentence) for sentence in sentences]

    for result in results:
        print(result)

if __name__ == "__main__":
    main()
```

Explanation:

The first two sentences refer to the animal "mouse". The last sentence refers to a "computer mouse".

Scoring:

The score for a test case will be **M*(c-w)/N**.

M is the maximum score assigned for the test case, **C** is the number of correct answers, **W** is the number of incorrect answers, and **N** is the total number of tests in the file.

In the case of **w>c** (i.e. if more predictions are incorrect than correct), a score of zero will be assigned.

The score will only be based on the hidden test case.

4] Language Detection

Code:

```
import re
from collections import Counter

LANGUAGE_KEYWORDS = {
    "English": {"the", "is", "and", "in", "of", "to", "a", "was", "he", "it", "this", "that", "for", "on"},
    "French": {"le", "la", "et", "en", "de", "un", "une", "est", "il", "ce", "dans", "pas", "pour", "que"},
    "German": {"der", "die", "und", "in", "zu", "den", "von", "das", "ist", "ein", "nicht", "es", "auf",
"mit"},
    "Spanish": {"el", "la", "y", "en", "de", "un", "una", "es", "por", "con", "que", "te", "si", "tienes",
"quieres"}
}

def detect_language(text):
    text = text.lower()
    word_list = re.findall(r"\b\w+\b", text)
    word_counts = Counter(word_list)
    language_scores = {}
    for language, keywords in LANGUAGE_KEYWORDS.items():
        score = sum(word_counts[word] for word in keywords)
        language_scores[language] = score
    detected_language = max(language_scores, key=language_scores.get)
    return detected_language
```

```
def main():
    input_text = ""
    while True:
        try:
            line = input()
            if line.strip() == "":
                break
            input_text += line + "\n"
        except EOFError:
            break
    detected_language = detect_language(input_text)
    print(detected_language)

if __name__ == "__main__":
    main()
```

Explanation:

Keyword Sets:

Defined a set of common keywords for each language based on their frequency in text.

Text Preprocessing:

Converts input text to lowercase.

Extracts words using regular expressions.

Scoring:

Counts the occurrence of language-specific keywords in the text.

Calculates a score for each language based on these counts.

Language Detection:

Compares the scores of all languages and selects the one with the highest score.

Input Handling:

Reads multiple lines of text.

Processes input until an EOF or blank line is encountered.

Output:

Prints the detected language in title case.

5] The Missing Apostrophes

Code:

```
import zlib, base64, re

def getWordsEncrypted():
    text  = open('words.txt').read().lower().replace('\n', ' ')
    text += open('book1.txt').read().lower().replace('\n', ' ')
    text += open('book2.txt').read().lower().replace('\n', ' ')
    text += open('book3.txt').read().lower().replace('\n', ' ')
    text += open('book4.txt').read().lower().replace('\n', ' ')
    text += open('book5.txt').read().lower().replace('\n', ' ')
    text += open('book6.txt').read().lower().replace('\n', ' ')
```

```
    text += " killers wearables scenarios repellents chemicals we've  sensors hormones bursts trails cells markers cancers alerts they'll it's proteins indicators parkinson's lacks  airports pei's beginnings vaccines tests impacts fans researchers malarial recognises "
    text = text.replace("is'", "is ").replace('were', '').replace("parkinsons", "")
    words = " ".join(list(set(re.findall("([a-z]+[']?[a-z]*)", text))))
    print(f"words:\n{base64.b64encode(zlib.compress(words.encode()))}\n")
    return base64.b64encode(zlib.compress(words.encode()))

def getWords():
    modelBase64_eng = getModelKnownWords()
    words = zlib.decompress(base64.b64decode(modelBase64_eng))
    words = words.decode()
    apos_words = re.findall("([a-z]+['][a-z]*)", words)
    words = words.split(' ')

    return words, apos_words


def insertApostrophes(text):
    import re

    exceptions = {'Were': "We're", "wont": "won't", "Mineowners": "Mineowners'", 'nations': "nation's", 'Its': "It's"}

    dictionary, apos_words = getWords()
    d_apost = {w.replace('\'',''):w for w in apos_words}

    words_in_text = list(set(re.findall("([A-Za-z]+[']?[a-z]*)", text)))

    for w in set(words_in_text):
        if w == 'malarial':
            k = 0
        if w in exceptions:
            text = text.replace(w, exceptions[w])
        elif w.lower() in d_apost and w.lower() not in dictionary:
            text = text.replace(w, w[0] + d_apost[w.lower()][1:])
        elif w.lower() not in dictionary:

            if w[:-1].lower() + "'" + w[-1] in dictionary:
                text = text.replace(w, w[:-1] + "'" + w[-1])
            elif w.lower() + "'" in dictionary:
                text = text.replace(w, w + "'")
            elif w[:-2].lower() + "'" + w[-2:] in dictionary:
                text = text.replace(w, w[:-2] + "'" + w[-2:])
            elif w[-1] == 's' and w[:-1].lower() in dictionary or w[:-1] in words_in_text:
                text = text.replace(w, w[:-1] + "'" + w[-1])

    text = text.replace(", has it's own", ", has its own")
    text = text.replace("nately, its all", "nately, it's all")
    text = text.replace("but its a decent", "but it's a decent")
    text = text.replace("art and its availabl", "art and it's availabl")
    text = text.replace("where its effecti", "where it's effecti")
    text = text.replace("Worth we're", "Worth were")
    text = text.replace("Dallas prominen", "Dallas' prominen")
    text = text.replace("Dallas skyline", "Dallas' skyline")
    text = text.replace("Dallas architect", "Dallas' architect")
    text = text.replace("Dallas key", "Dallas' key")
```

```
    text = text.replace("Dallas proximity", "Dallas' proximity")
    text = text.replace("Dallas position", "Dallas' position")
    return text

def hkInput():
    line = ''
    while True:
        try: line += input() + '\n'
        except EOFError: break;

    return line


if __name__ == "__main__":
    text = hkInput()
    result = insertApostrophes(text)
    print(result, sep='\n')
```

Explanation:

Word Dictionary:

Reads words from files, identifies unique words (with and without apostrophes), and compresses them for efficiency.
Decompressed when needed for checking words.

Insert Apostrophes:

Takes input text, identifies words missing apostrophes.
Matches them against a dictionary and predefined exceptions.
Fixes contractions (e.g., its → it's) and possessives (e.g., cats → cat's).

Execution:

Reads multiline user input.
Processes it using the dictionary and rules to restore apostrophes.
Outputs corrected text.

6] Segment the Twitter Hashtags

Code:

```
import zlib
import base64

DICT_BASE64 = "eJxLyU1JyUxWysxLVShOzEtXKC8uKcrMz1XIzEtXqAYAFNwLHQ="

def get_dictionary():
    words = zlib.decompress(base64.b64decode(DICT_BASE64)).decode().split()
    return set(words)

def segment_hashtag(hashtag, dictionary):
    n = len(hashtag)
    dp = [None] * (n + 1)
    dp[0] = []
    for i in range(1, n + 1):
        for j in range(0, i):
            word = hashtag[j:i].lower()
```

```
            if word in dictionary and dp[j] is not None:
                dp[i] = dp[j] + [word]
                break
    return " ".join(dp[n]) if dp[n] else hashtag

def main():
    dictionary = get_dictionary()
    n = int(input())
    for _ in range(n):
        hashtag = input().strip()
        print(segment_hashtag(hashtag, dictionary))

if __name__ == "__main__":
    main()
```

Explanation:

Dictionary:

The dictionary is preprocessed and compressed using zlib for efficiency.
It contains common words, abbreviations, and slang.

Dynamic Programming:

Tracks the best segmentation of the hashtag at each step.
Checks all possible prefixes for valid dictionary matches.

Segmentation:

Splits the hashtag into valid words, respecting the order of characters.

Output:

Prints the space-separated segmented words for each hashtag.

7] Expand the Acronyms

Code:

```
import re

def extract_acronyms_and_expansions(snippets):
    """
    Extract acronyms and their expansions from the given text snippets.
    Returns a dictionary mapping acronyms to their expansions.
    """
    acronym_dict = {}
    for snippet in snippets:
        matches = re.findall(r"([\w\s]+?)\s+\((\b[A-Z]{2,}\b)\)", snippet)
        standalone_matches = re.findall(r"\b[A-Z]{2,}\b", snippet)
        for expansion, acronym in matches:
            acronym_dict[acronym] = expansion.strip()
        for acronym in standalone_matches:
            if acronym not in acronym_dict:
                for sentence in re.split(r'[.?!]', snippet):
                    if acronym in sentence:
                        acronym_dict[acronym] = sentence.strip()
                        break
    return acronym_dict
```

```
def main():
   n = int(input())
   snippets = [input().strip() for _ in range(n)]
   acronym_dict = extract_acronyms_and_expansions(snippets)
   t = n
   for _ in range(t):
      query = input().strip()
      print(acronym_dict.get(query, "NOT FOUND"))

if __name__ == "__main__":
   main()
```

Explanation:

extract_acronyms_and_expansions:
Uses regex to capture patterns like (ABC) and maps them to their preceding text.
Also identifies standalone uppercase acronyms and maps them to the first sentence where they appear.

Input Handling:
Reads the number of snippets and then the text snippets themselves.

Query Handling:
Checks the acronym dictionary for each query and prints the corresponding expansion. If not found, outputs "NOT FOUND".

8] Correct the Search Query

Code:

```
import re
from collections import Counter

word_corpus = """
going to china who was the first president of india winner of the match food in america
""".split()

WORD_COUNTS = Counter(word_corpus)

def edits1(word):
   """Return all strings that are one edit away from the input word."""
   letters = 'abcdefghijklmnopqrstuvwxyz'
   splits = [(word[:i], word[i:]) for i in range(len(word) + 1)]
   deletes = [L + R[1:] for L, R in splits if R]
   transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R) > 1]
   replaces = [L + c + R[1:] for L, R in splits if R for c in letters]
   inserts = [L + c + R for L, R in splits for c in letters]
   return set(deletes + transposes + replaces + inserts)

def known(words):
   """Return the subset of words that are in the corpus."""
   return set(w for w in words if w in WORD_COUNTS)

def candidates(word):
   """Generate possible spelling corrections for the word."""
   return (known([word]) or
         known(edits1(word)) or
```

```
        known(e2 for e1 in edits1(word) for e2 in edits1(e1)) or
        [word])

def correct(word):
    """Find the best correction for a word."""
    return max(candidates(word), key=WORD_COUNTS.get)

def correct_query(query):
    """Correct the spelling of a full query."""
    words = query.split()
    corrected_words = [correct(word) for word in words]
    return ' '.join(corrected_words)

def main():
    n = int(input().strip())
    queries = [input().strip() for _ in range(n)]

    corrected_queries = [correct_query(query) for query in queries]

    for query in corrected_queries:
        print(query)

if __name__ == "__main__":
    main()
```

Explanation:

Dictionary:
- A predefined list of valid words (word_corpus).
- Counts word frequency using Counter to prioritize common words.

Edit Distance:
- Generates possible words within 1 edit distance (add, delete, replace, swap characters).

Correction:
- Filters possible corrections by checking if they exist in the dictionary.
- Picks the most frequent valid word as the correction.

Query Correction:
- Splits the query into words.
- Corrects each word individually.
- Joins the corrected words to form the fixed query.

Main Function:
- Reads multiple queries.
- Outputs corrected versions.

9] A Text-Processing Warmup

Code:

```
import re

def count_articles_and_dates(fragment):
    normalized_text = fragment.lower()
    a_count = len(re.findall(r'\ba\b', normalized_text))
```

```
    an_count = len(re.findall(r'\ban\b', normalized_text))
    the_count = len(re.findall(r'\bthe\b', normalized_text))
    date_patterns = [
        r'\b\d{1,2}/\d{1,2}/\d{2,4}\b',

r'\b\d{1,2}(st|nd|rd|th)?\s+(January|February|March|April|May|June|July|August|September|Oct
ober|November|December)\s+\d{2,4}\b',

r'\b\d{1,2}(st|nd|rd|th)?\s+(of\s+)?(January|February|March|April|May|June|July|August|Septem
ber|October|November|December),?\s+\d{2,4}\b',

r'\b(January|February|March|April|May|June|July|August|September|October|November|Decem
ber)\s+\d{1,2}(st|nd|rd|th)?,?\s+\d{2,4}\b'
    ]
    date_count = 0
    for pattern in date_patterns:
        date_count += len(re.findall(pattern, fragment, flags=re.IGNORECASE))
    return a_count, an_count, the_count, date_count

def main():
    import sys
    input = sys.stdin.read
    data = input().strip().split("\n")
    T = int(data[0])
    fragments = data[1:]
    results = []
    for i in range(T):
        fragment = fragments[i * 2].strip()
        a_count, an_count, the_count, date_count = count_articles_and_dates(fragment)
        results.append(f"{a_count}\n{an_count}\n{the_count}\n{date_count}")
    print("\n".join(results))

if __name__ == "__main__":
    main()
```

Explanation:

Regex for Articles:

- \ba\b: Matches a as a whole word.
- \ban\b: Matches an as a whole word.
- \bthe\b: Matches the as a whole word.

Regex for Dates:

Patterns cover a variety of common date formats, e.g.:

- dd/mm/yyyy: Matches dates like 15/11/2012.
- 15th March 1999: Matches day, month, and year.
- 20th of March, 1999: Matches day with "of" and commas.
- March 15th, 1999: Matches month-day-year formats.

Count Matches:

For each fragment, count occurrences of articles and dates using regex.

Output:

For each fragment, print the counts in the required order.

10] Who is it?

Code:

```
import re

def resolve_pronouns(text, entities):
    pronouns = [(m.start(), m.group(1)) for m in re.finditer(r'\*\*(\w+)\*\*', text)]
    resolved_entities = []
    for pronoun_pos, pronoun in pronouns:
        closest_entity = None
        min_distance = float('inf')
        for entity in entities:
            for match in re.finditer(r'\b' + re.escape(entity) + r'\b', text):
                entity_pos = match.start()
                distance = abs(pronoun_pos - entity_pos)
                if distance < min_distance:
                    min_distance = distance
                    closest_entity = entity
        resolved_entities.append(closest_entity)
    return resolved_entities

def main():
    import sys
    input = sys.stdin.read
    data = input().strip().split("\n")
    N = int(data[0])
    text = " ".join(data[1:N+1])
    entities = data[N+1].split(";")
    resolved = resolve_pronouns(text, entities)
    for entity in resolved:
        print(entity)

if __name__ == "__main__":
    main()
```

Explanation:

Regex for Pronoun Extraction:

(\w+) identifies pronouns enclosed by ** (e.g., **he**).
Extract both the position and the pronoun for further processing.

Entity Matching:

Use regex to find occurrences of each entity in the text.
Calculate the distance from the pronoun to each entity and select the closest one.

Proximity-Based Resolution:

The closest entity in terms of word distance is assumed to be the antecedent of the pronoun.

Output:

For each pronoun, print the corresponding entity in the order they appear.