Problem-1:

Build an approach for storing versions and files. A simple approach is to store each file as a separate version. However that is inefficient, Instead of storing each version as a separate file, look at storing the deltas. Come up with an approach to store the base version and deltas in the file. Persist them all in one file. Come up with an efficient data structure to store the file, and deltas, and persist in that. Write methods to generate any version immediately.

Solution:

import json

class FileVersion:

def \_\_init\_\_(self, content):

self.content = content

self.deltas = []

def add\_delta(self, delta):

self.deltas.append(delta)

def apply\_delta(self, delta):

# Apply the delta to the content

pass

class FileDelta:

def \_\_init\_\_(self, changes):

self.changes = changes

class VersionControl:

def \_\_init\_\_(self, base\_version):

self.base\_version = base\_version

def add\_delta(self, delta):

self.base\_version.add\_delta(delta)

def generate\_version(self, target\_version):

current\_version = self.base\_version

for delta in current\_version.deltas:

current\_version = self.apply\_delta(current\_version, delta)

if current\_version.version == target\_version:

return current\_version

return None

def apply\_delta(self, version, delta):

# Apply the delta to the version and return the updated version

pass

def persist\_to\_file(self, file\_path):

data = {

'base\_version': self.base\_version,

'deltas': [delta for delta in self.base\_version.deltas]

}

with open(file\_path, 'w') as file:

json.dump(data, file)

@staticmethod

def load\_from\_file(file\_path):

with open(file\_path, 'r') as file:

data = json.load(file)

base\_version = data['base\_version']

deltas = data['deltas']

version\_control = VersionControl(base\_version)

for delta\_data in deltas:

delta = FileDelta(delta\_data)

version\_control.add\_delta(delta)

return version\_control

# Example usage

base\_version = FileVersion("Initial content of the file")

version\_control = VersionControl(base\_version)

# Add deltas to version control

delta1 = FileDelta("Changes from version 1 to version 2")

delta2 = FileDelta("Changes from version 2 to version 3")

version\_control.add\_delta(delta1)

version\_control.add\_delta(delta2)

# Generate a specific version

target\_version = 2

specific\_version = version\_control.generate\_version(target\_version)

# Persist version control to a file

file\_path = 'version\_control.json'

version\_control.persist\_to\_file(file\_path)

# Load version control from file

loaded\_version\_control = VersionControl.load\_from\_file(file\_path)

Problem-2:

Build a word count application, where the constraints are that you have 10 MB RAM and 1 GB text file. You should be able to efficiently parse the text file and output the words and counts in a sorted way. Write a program to read a large file, and emit the sorted words along with the count. Try to implement fuzzy search as well (fix the spelling issues) Algorithm should have Log N complexity.

Solution:

import heapq

import re

# Constants

MAX\_RAM = 10 \* 1024 \* 1024 # 10 MB

MAX\_FILE\_SIZE = 1024 \* 1024 \* 1024 # 1 GB

def process\_chunk(chunk):

word\_count = {}

words = re.findall(r'\b\w+\b', chunk.lower()) # Tokenization

for word in words:

word = word.strip()

if word in word\_count:

word\_count[word] += 1

else:

word\_count[word] = 1

return word\_count

def merge\_counts(counts, new\_counts):

for word, count in new\_counts.items():

if word in counts:

counts[word] += count

else:

counts[word] = count

def word\_count(file\_path):

word\_counts = {}

with open(file\_path, 'r') as file:

while True:

chunk = file.read(MAX\_RAM)

if not chunk:

break

new\_counts = process\_chunk(chunk)

merge\_counts(word\_counts, new\_counts)

sorted\_word\_counts = sorted(word\_counts.items(), key=lambda x: (-x[1], x[0])) # Sorting

return sorted\_word\_counts

def levenshtein\_distance(s1, s2):

if len(s1) < len(s2):

return levenshtein\_distance(s2, s1)

if len(s2) == 0:

return len(s1)

previous\_row = range(len(s2) + 1)

for i, c1 in enumerate(s1):

current\_row = [i + 1]

for j, c2 in enumerate(s2):

insertions = previous\_row[j + 1] + 1

deletions = current\_row[j] + 1

substitutions = previous\_row[j] + (c1 != c2)

current\_row.append(min(insertions, deletions, substitutions))

previous\_row = current\_row

return previous\_row[-1]

def fuzzy\_search(word, word\_counts):

similar\_words = []

for w, \_ in word\_counts:

if levenshtein\_distance(word, w) <= 2: # Set a threshold for similarity

similar\_words.append(w)

return similar\_words

# Example usage

file\_path = 'large\_file.txt'

sorted\_word\_counts = word\_count(file\_path)

for word, count in sorted\_word\_counts:

print(word, count)

# Example fuzzy search

search\_word = 'helo'

similar\_words = fuzzy\_search(search\_word, sorted\_word\_counts)

print("Fuzzy Search Results for '{}':".format(search\_word))

for word in similar\_words:

print(word)

Problem-3:

Come up with an approach for product configuration, where multiple products can be stored. Build an in-memory database or in-memory storage. We should be able to have product categories along with product descriptions and details. We should be able to store a wide range of types of products similar to Amazon, we should be able to implement efficient search of the products and flexible configuration of the products. In addition to in-memory storage, build an efficient textual search on any of the parameters (similar to search in Amazon).

Solution:

class ProductConfiguration:

def \_\_init\_\_(self):

self.products = {}

def add\_product(self, category, subcategory, product\_name, description, details):

if category not in self.products:

self.products[category] = {}

if subcategory not in self.products[category]:

self.products[category][subcategory] = {}

self.products[category][subcategory][product\_name] = {

'description': description,

'details': details

}

def search\_product(self, query):

results = []

for category, subcategories in self.products.items():

for subcategory, products in subcategories.items():

for product\_name, data in products.items():

if query.lower() in product\_name.lower() or query.lower() in data['description'].lower():

results.append((category, subcategory, product\_name, data))

return results

def textual\_search(self, query):

results = []

for category, subcategories in self.products.items():

for subcategory, products in subcategories.items():

for product\_name, data in products.items():

if self.similarity(query.lower(), product\_name.lower()) >= 0.7:

results.append((category, subcategory, product\_name, data))

elif self.similarity(query.lower(), data['description'].lower()) >= 0.7:

results.append((category, subcategory, product\_name, data))

return results

@staticmethod

def similarity(s1, s2):

len\_s1 = len(s1)

len\_s2 = len(s2)

if len\_s1 == 0 or len\_s2 == 0:

return 0

matrix = [[0] \* (len\_s2 + 1) for \_ in range(len\_s1 + 1)]

for i in range(len\_s1 + 1):

matrix[i][0] = i

for j in range(len\_s2 + 1):

matrix[0][j] = j

for i in range(1, len\_s1 + 1):

for j in range(1, len\_s2 + 1):

cost = 0 if s1[i - 1] == s2[j - 1] else 1

matrix[i][j] = min(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1, matrix[i - 1][j - 1] + cost)

max\_len = max(len\_s1, len\_s2)

similarity\_ratio = (max\_len - matrix[len\_s1][len\_s2]) / max\_len

return similarity\_ratio

# Example usage

product\_config = ProductConfiguration()

# Add products

product\_config.add\_product('Electronics', 'Smartphones', 'iPhone 12', 'Apple iPhone 12', {'color': 'Black', 'price': '$799'})

product\_config.add\_product('Electronics', 'Smartphones', 'Samsung Galaxy S21', 'Samsung Galaxy S21', {'color': 'White', 'price': '$699'})

product\_config.add\_product('Electronics', 'Laptops', 'MacBook Pro', 'Apple MacBook Pro', {'color': 'Silver', 'price': '$1299'})

# Search products

print("Search results for 'iPhone':")

print(product\_config.search\_product('iPhone'))

# Textual search

print("Textual search results for 'Galaxy':")

print(product\_config.textual\_search('Galaxy'))