

# Assignment Questions

## 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.

## Code:

```
import json
import difflib

class TrieNode:
    def __init__(self):
        self.children = {}
        self.end_of_version = False

class VersionControlSystem:
    def __init__(self):
        self.root = TrieNode()
        self.base_version = ""
        self.deltas = []

    def add_version(self, new_version):
        if not self.base_version:
            self.base_version = new_version
            self._insert_version(new_version, 1)
        else:
            last_version = self.get_version(len(self.deltas) + 1)
            delta = self.compute_delta(last_version, new_version)
            self.deltas.append(delta)
            self._insert_version(new_version, len(self.deltas) + 1)

    def get_version(self, version_number):
        version = []
        node = self.root
        version_count = 0

        def dfs(current_node, current_version):
            nonlocal version_count
            if version_count == version_number:
                return
            if current_node.end_of_version:
                version_count += 1
                if version_count == version_number:
                    version.extend(current_version)
                    return
            for char, child in current_node.children.items():
```

```

        dfs(child, current_version + [char])
        if version_count == version_number:
            return

    dfs(node, [])
    return ''.join(version)

def compute_delta(self, old_version, new_version):
    diff = difflib.ndiff(old_version, new_version)
    delta = ''.join(diff)
    return delta

def _insert_version(self, version, version_number):
    node = self.root
    for char in version:
        if char not in node.children:
            node.children[char] = TrieNode()
        node = node.children[char]
    node.end_of_version = True

def save_to_file(self, filename):
    data = {
        "base_version": self.base_version,
        "deltas": self.deltas
    }
    with open(filename, 'w') as file:
        json.dump(data, file)

def load_from_file(self, filename):
    with open(filename, 'r') as file:
        data = json.load(file)
    self.base_version = data["base_version"]
    self.deltas = data["deltas"]
    self._insert_version(self.base_version, 1)
    for i, delta in enumerate(self.deltas):
        version = self.apply_delta(self.get_version(i + 1), delta)
        self._insert_version(version, i + 2)

def apply_delta(self, version, delta):
    version = list(version)
    index = 0
    for change in delta:
        if change.startswith('-'):
            if index < len(version):
                version.pop(index)
        elif change.startswith('+'):
            if len(change) > 2:
                version.insert(index, change[2])
                index += 1
        elif change.startswith(' '):
            index += 1
    return ''.join(version)

```

```

if __name__ == "__main__":
    vcs = VersionControlSystem()
    vcs.add_version("Hello World")
    vcs.add_version("Hello World!")
    vcs.add_version("Hello, World!")
    vcs.add_version("Hello, World!!!")

    print(vcs.get_version(1))
    print(vcs.get_version(2))
    print(vcs.get_version(3))
    print(vcs.get_version(4))

    # Save to file
    vcs.save_to_file('versions.json')

    # Load from file
    vcs2 = VersionControlSystem()
    vcs2.load_from_file('versions.json')
    print(vcs2.get_version(4))

```

## Explanation:

This code is a simple version control system designed to manage and track different versions of a text. It uses a Trie data structure to store and retrieve versions efficiently, and it keeps track of changes between versions using "deltas."

## Key Components

### 1. Trie Data Structure:

- **What is it?** A Trie is a tree-like structure where each node represents a character. It helps store strings in a way that saves space by sharing common prefixes.
- **Why use it?** In this code, the Trie efficiently stores different versions of the text. Each unique version is a path in the Trie, which makes it easy to find and compare versions.

### 2. VersionControlSystem Class:

- **Purpose:** Manages different versions of a text and tracks changes.
- **How it Works:**
  1. `add_version(new_version)`: Adds a new version. If it's the first one, it's set as the base version. For later versions, it computes the difference (delta) from the previous version and saves it.
  2. `get_version(version_number)`: Retrieves a specific version by number using the Trie.
  3. `compute_delta(old_version, new_version)`: Finds the changes between two versions.
  4. `save_to_file(filename)`: Saves the base version and deltas to a file.
  5. `load_from_file(filename)`: Loads data from a file and reconstructs the stored versions.
  6. `apply_delta(version, delta)`: Applies the recorded changes to get the new version.

## How It Works

1. **Adding Versions:** The first version is stored directly. For subsequent versions, it calculates the differences from the previous version and stores these differences (deltas) instead of the entire new version. This saves space and makes version management more efficient.
2. **Retrieving Versions:** To get a specific version, the code traverses the Trie to find the correct path corresponding to that version.
3. **Saving and Loading:** The system can save all versions and their changes to a file and then reload them when needed. This allows for persistent storage of version history.

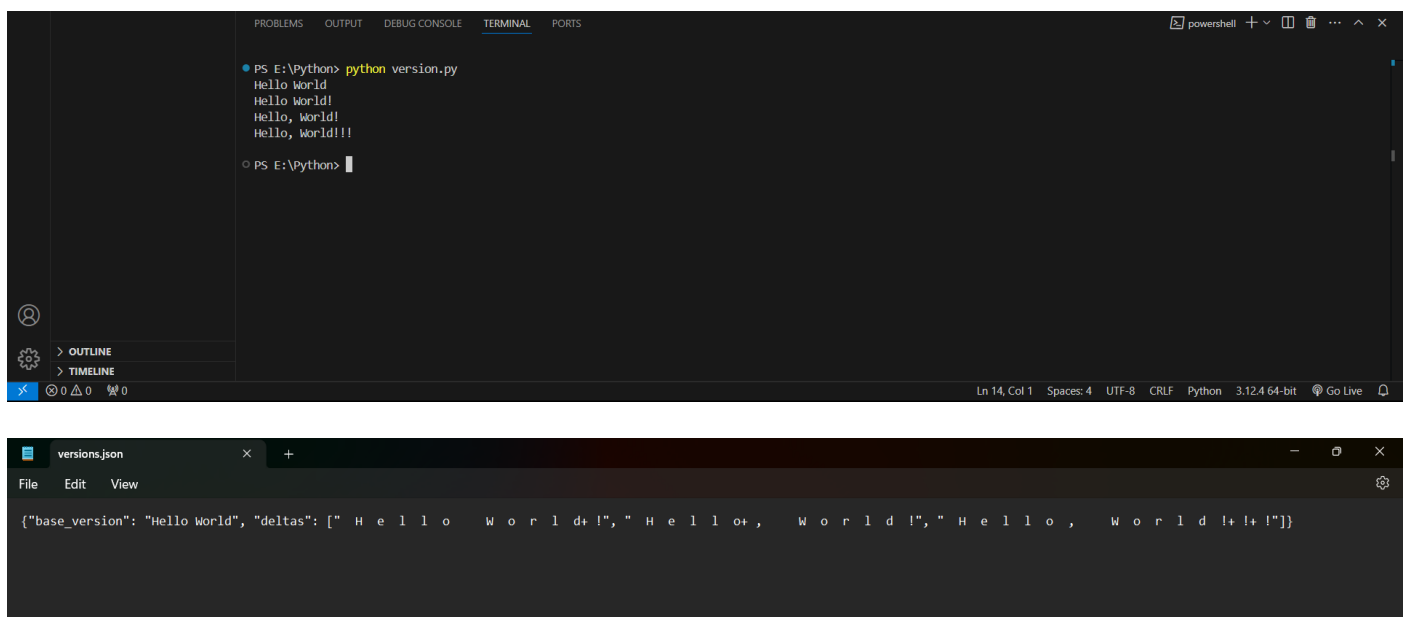
## Complexity

- **Insertion/Searching in Trie:**  $O(n)$ , where  $n$  is the length of the version being inserted or searched for. Since the Trie nodes are stored based on individual characters, each insertion or search operation depends linearly on the length of the version.
- **Delta Computation:**  $O(m + n)$ , where  $m$  and  $n$  are the lengths of the old and new versions, respectively. This complexity is due to the comparison process performed by `difflib.ndiff`.
- **File Operations:**  $O(k)$ , where  $k$  is the size of the data being saved or loaded. JSON operations generally scale with the size of the data.

## Running the Code

1. **Save the Code:** Put the Python code into a file named `version.py`.
2. **Open CMD:** Launch Command Prompt on your computer.
3. **Navigate to the Directory:** Use the `cd` command to go to the folder where `version.py` is saved.
4. **To Run the Code:** Type: `python version.py`
5. **Check Results:** The script will display different versions of the text and save them to a file named `versions.json`. It will then reload the data to demonstrate how it works.

## Output(s):



The image shows two screenshots. The top screenshot is a PowerShell terminal window with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The terminal shows the command `python version.py` being executed, resulting in the output: `Hello World`, `Hello World!`, `Hello, World!`, and `Hello, World!!!`. The bottom screenshot is a text editor window showing the contents of `versions.json`. The JSON data is: `{"base_version": "Hello World", "deltas": [{"H e l l o W o r l d !", "H e l l o , W o r l d !", "H e l l o , W o r l d !+ !+ !"}]}`

## 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.

```
import argparse
import sys
from collections import defaultdict
import math

class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
        self.count = 0

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True
        node.count += 1

    def get_words(self, node=None, prefix=''):
        if node is None:
            node = self.root
        words = []
        if node.is_end_of_word:
            words.append((prefix, node.count))
        for char, next_node in node.children.items():
            words.extend(self.get_words(next_node, prefix + char))
        return words

def read_file_in_chunks(file_path, chunk_size=1024*1024): # 1 MB chunk size
    with open(file_path, 'r') as file:
        while True:
            data = file.read(chunk_size)
            if not data:
                break
            yield data

def process_file(file_path, trie):
    word_buffer = ''
    for chunk in read_file_in_chunks(file_path):
        words = (word_buffer + chunk).split()
```

```

        word_buffer = words.pop() if chunk[-1].isalpha() else ''
        for word in words:
            trie.insert(word.lower())
    if word_buffer:
        trie.insert(word_buffer.lower())

def output_sorted_words(trie):
    words_with_counts = trie.get_words()
    sorted_words = sorted(words_with_counts, key=lambda x: x[0])
    for word, count in sorted_words:
        print(f"{word}: {count}")

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(trie, word, max_distance=1):
    matches = []
    def search(node, current_word):
        if node.is_end_of_word and levenshtein_distance(word, current_word) <=
max_distance:
            matches.append((current_word, node.count))
        for char, next_node in node.children.items():
            search(next_node, current_word + char)
    search(trie.root, '')
    return matches

def main():
    parser = argparse.ArgumentParser(description='Process a large text file to count word
frequencies and perform fuzzy search.')
    parser.add_argument('file_path', type=str, help='Path to the large text file.')
    parser.add_argument('search_word', type=str, help='Word to search for using fuzzy
search.')
    args = parser.parse_args()

    trie = Trie()
    process_file(args.file_path, trie)
    output_sorted_words(trie)

    fuzzy_matches = fuzzy_search(trie, args.search_word)
    for match, count in fuzzy_matches:
        print(f"Fuzzy match: {match} with count {count}")

```

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

## Explanation:

This script is designed to handle large text files efficiently by counting word frequencies and performing fuzzy searches for similar words. It uses a Trie data structure for efficient word management and a Levenshtein distance algorithm for fuzzy searching.

Key Components:

### Levenshtein Distance:

- **What is it?** A measure of how different two strings are by counting the minimum number of single-character edits (insertions, deletions, substitutions) required to change one string into the other.
- **Why use it?** It helps in fuzzy searching by finding words that are similar to the search word within a specified edit distance. This is useful for handling typos or slight variations in words.

## How It Works

### 1. Reading and Processing the File:

- **read\_file\_in\_chunks:** Reads the file in chunks to handle large files without using too much memory. Each chunk is processed separately.
- **process\_file:** Takes the file path and a Trie object. It reads the file in chunks, splits the text into words, and inserts these words into the Trie while keeping track of their frequency.

### 2. Inserting and Counting Words:

- **Trie:**
  - **insert(word):** Adds a word to the Trie and increments its count if it already exists.
  - **get\_words():** Retrieves all words stored in the Trie along with their counts.

### 3. Sorting and Displaying Words:

- **output\_sorted\_words:** Fetches words and their counts from the Trie, sorts them alphabetically, and prints them.

### 4. Fuzzy Search:

- **levenshtein\_distance(s1, s2):** Calculates the Levenshtein distance between two strings to measure their similarity.
- **fuzzy\_search(trie, word, max\_distance):** Searches for words in the Trie that are similar to the given search word within the specified edit distance. It returns words that match the fuzzy search criteria.

## Complexity

### • Trie Operations:

- **Insertion/Search:**  $O(m)$ , where  $m$  is the length of the word being inserted or searched. Operations are efficient because of the Trie's tree structure.

### • Levenshtein Distance:

- **Calculation:**  $O(n * m)$ , where  $n$  and  $m$  are the lengths of the two strings being compared. This is due to the dynamic programming approach used to compute the distance.

- **File Processing:**
  - **Reading in Chunks:** Efficient for handling large files because it processes one chunk at a time.

## Running the Code in CMD

1. **Save the Code:** Save the provided code into a file named `word_count_trie.py`.
2. **Open CMD:** Open Command Prompt on your computer.
3. **Navigate to Directory:** Use the `cd` command to go to the folder where `word_count_trie.py` is saved.
4. **To Run the Script:** `python word_count_trie.py large_text_file.txt search_word`
5. Here, in the `large_text_file.txt` I have used the Bible, as the sample text file, I converted the pdf into txt and used it as a sample text file for this coding question. `search_word` can be the name/word from the bible you want to search.

## Output(s):

```

version.py
versions.json
word_count_trie.py
word_count.py

PS E:\Python> python word_count_trie.py large_text_file.txt Judas

ziza: 2
ziza,: 2
ziza.: 1
zohethy: 1
zohethy,: 1
zomzommims,: 1
zooom.: 1
zorobabel: 14
zorobabel,: 8
zorobabel.: 2
zorobabel?: 2
zuzim: 1
æm: 1
æmt: 1
æmtis: 2
Fuzzy match: judas with count 114

PS E:\Python>

```

## 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).

Code:

```

import json

class TrieNode:
    def __init__(self):
        self.children = {}
        self.product_ids = set()

class Trie:
    def __init__(self):
        self.root = TrieNode()

```



```

def insert(self, key, product_id):
    node = self.root
    for char in key:
        if char not in node.children:
            node.children[char] = TrieNode()
        node = node.children[char]
        node.product_ids.add(product_id)

def search(self, key):
    node = self.root
    for char in key:
        if char not in node.children:
            return set()
        node = node.children[char]
    return self._collect_all_ids(node)

def _collect_all_ids(self, node):
    ids = set(node.product_ids)
    for child in node.children.values():
        ids.update(self._collect_all_ids(child))
    return ids

class ProductDatabase:
    def __init__(self):
        self.trie = Trie()
        self.products = {}

    def add_product(self, product_id, name, category, description, price):
        product_data = {
            "name": name,
            "category": category,
            "description": description,
            "price": price
        }
        self.products[product_id] = product_data
        # Index product attributes in lowercase
        self._index_substrings(name.lower(), product_id)
        self._index_substrings(category.lower(), product_id)
        self._index_substrings(description.lower(), product_id)

    def _index_substrings(self, text, product_id):
        length = len(text)
        for i in range(length):
            for j in range(i + 1, length + 1):
                self.trie.insert(text[i:j], product_id)

    def search(self, query):
        query = query.lower()
        product_ids = self.trie.search(query)
        results = [self.products[pid] for pid in product_ids]
        return results

def main():
    db = ProductDatabase()

```

```

while True:
    print("\n1. Add Product")
    print("2. Search Products")
    print("3. Exit")
    choice = input("Enter your choice 1/2/3: ").strip()

    if choice == '1':
        try:
            product_id = int(input("Enter product ID: ").strip())
            name = input("Enter product name: ").strip()
            category = input("Enter product category: ").strip()
            description = input("Enter product description: ").strip()
            price = float(input("Enter product price: ").strip())

            if price < 0:
                print("Price cannot be negative.")
                continue

            db.add_product(product_id, name, category, description, price)
            print(f"Product {product_id} added successfully.")
        except ValueError as e:
            print(f"Invalid input: {e}. Please enter the correct data type.")

    elif choice == '2':
        query = input("Enter search query: ").strip()
        if not query:
            print("Search query cannot be empty.")
            continue

        results = db.search(query)
        if results:
            print("\nSearch results:")
            for product in results:
                print(json.dumps(product, indent=2))
        else:
            print("No products found.")

    elif choice == '3':
        print("Exiting...")
        break

    else:
        print("Invalid choice. Please select 1, 2, or 3.")

if __name__ == "__main__":
    main()

```

### Explanation:

This code sets up a simple product database where you can add products and search for them using keywords. It uses a Trie data structure to handle efficient search operations, and it allows you to perform substring searches on product attributes.

Key Component:

### How the Code Works:

- **TrieNode Class:**
  - Represents each node in the Trie. Each node has a dictionary (children) for its child nodes and a set (product\_ids) to store IDs of products associated with that node.
- **Trie Class:**
  - **insert(key, product\_id):** Adds a substring (key) to the Trie and associates it with a product ID. It ensures that each node in the Trie keeps track of all product IDs that pass through it.
  - **search(key):** Searches for all product IDs associated with a given substring. It collects all IDs from the node corresponding to the end of the substring and its descendants.
  - **\_collect\_all\_ids(node):** A helper function to gather all product IDs from a given node and its children.
- **ProductDatabase Class:**
  - Manages products and their data, and handles indexing and searching.
  - **add\_product(product\_id, name, category, description, price):** Adds a product to the database and indexes its attributes. It stores product data and indexes all substrings of the product's name, category, and description in the Trie.
  - **\_index\_substrings(text, product\_id):** Creates and inserts all possible substrings of a given text into the Trie, associating each with the product ID.
  - **search(query):** Searches for products based on a query. It uses the Trie to find all matching product IDs and retrieves the product details.
- **Main Function:**
  - Provides a simple command-line interface to add products, search for products, or exit the program. It prompts the user for input and handles various operations based on user choices.

### Complexity

- **Trie Operations:**
  - **Insertion/Search:**  $O(m * n)$ , where  $m$  is the number of substrings and  $n$  is the average length of substrings. Insertion and search operations involve traversing and managing nodes in the Trie.
  - **Indexing Substrings:**  $O(n^2)$ , where  $n$  is the length of the text. This is because it generates all substrings of the text, which involves nested loops.
- **Search Operation:**
  - The search operation in the Trie is efficient, with complexity proportional to the length of the query.

### Running the Code in CMD

1. **Save the Code:** Save the Python code into a file named `product_database.py`.
2. **Open CMD:** Open Command Prompt on your computer.
3. **Navigate to Directory:** Use the `cd` command to go to the folder where `product_database.py` is saved.
4. **Run the Script:** `python product_database.py`
5. **Interact with the Program:**
  - **Add Product:** Select option 1 and enter the required details to add a product.
  - **Search Products:** Select option 2 and enter a search query to find matching products.

- **Exit:** Select option 3 to exit the program.

## Output(s):

```
PS E:\Python> python product_database.py

1. Add Product
2. Search Products
3. Exit
Enter your choice 1/2/3: 1
Enter product ID: 745
Enter product name: Daikin 3000 Coolmaster 1 TON Split AC
Enter product category: Electrical Appliances
Enter product description: Split AC
Enter product price: 55000
Product 745 added successfully.

1. Add Product
2. Search Products
3. Exit
Enter your choice 1/2/3: 1
Enter product ID: 607
Enter product name: Viper Adder Pro V2
Enter product category: Electronics
Enter product description: Gaming Mice
Enter product price: 4500
Product 7 added successfully.
```

```
1. Add Product
2. Search Products
3. Exit
Enter your choice 1/2/3: 2
Enter search query: Daikin

Search results:
{
  "name": "Daikin 3000 Coolmaster 1 TON Split AC",
  "category": "Electrical Appliances",
  "description": "Split AC",
  "price": 55000.0
}
```

```
1. Add Product
2. Search Products
3. Exit
Enter your choice 1/2/3: 2
Enter search query: Mice

Search results:
{
  "name": "Viper Adder Pro V2",
  "category": "Electronics",
  "description": "Gaming Mice",
  "price": 4500.0
}
```