LAB TEST-2

NAME: MANASA ACHINA

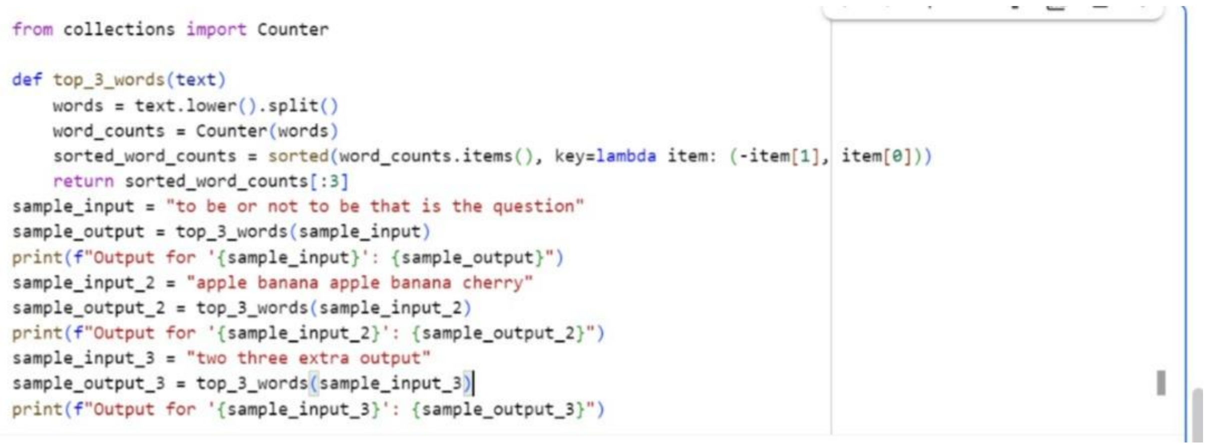
ENROLL NO.:2403A53043

BATCH NO.:24BTCAICYB02

QUESTION: L.1

PROMPT: You are given a string of lowercase words separated by spaces, representing text from urban public transit analytics

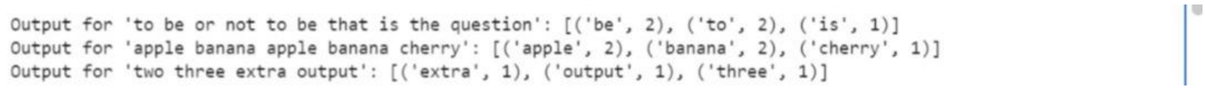
CODE:



```
from collections import Counter

def top_3_words(text)
    words = text.lower().split()
    word_counts = Counter(words)
    sorted_word_counts = sorted(word_counts.items(), key=lambda item: (-item[1], item[0]))
    return sorted_word_counts[:3]
sample_input = "to be or not to be that is the question"
sample_output = top_3_words(sample_input)
print(f"Output for '{sample_input}': {sample_output}")
sample_input_2 = "apple banana apple banana cherry"
sample_output_2 = top_3_words(sample_input_2)
print(f"Output for '{sample_input_2}': {sample_output_2}")
sample_input_3 = "two three extra output"
sample_output_3 = top_3_words(sample_input_3)
print(f"Output for '{sample_input_3}': {sample_output_3}")
```

OUTPUT:



```
Output for 'to be or not to be that is the question': [('be', 2), ('to', 2), ('is', 1)]
Output for 'apple banana apple banana cherry': [('apple', 2), ('banana', 2), ('cherry', 1)]
Output for 'two three extra output': [('extra', 1), ('output', 1), ('three', 1)]
```

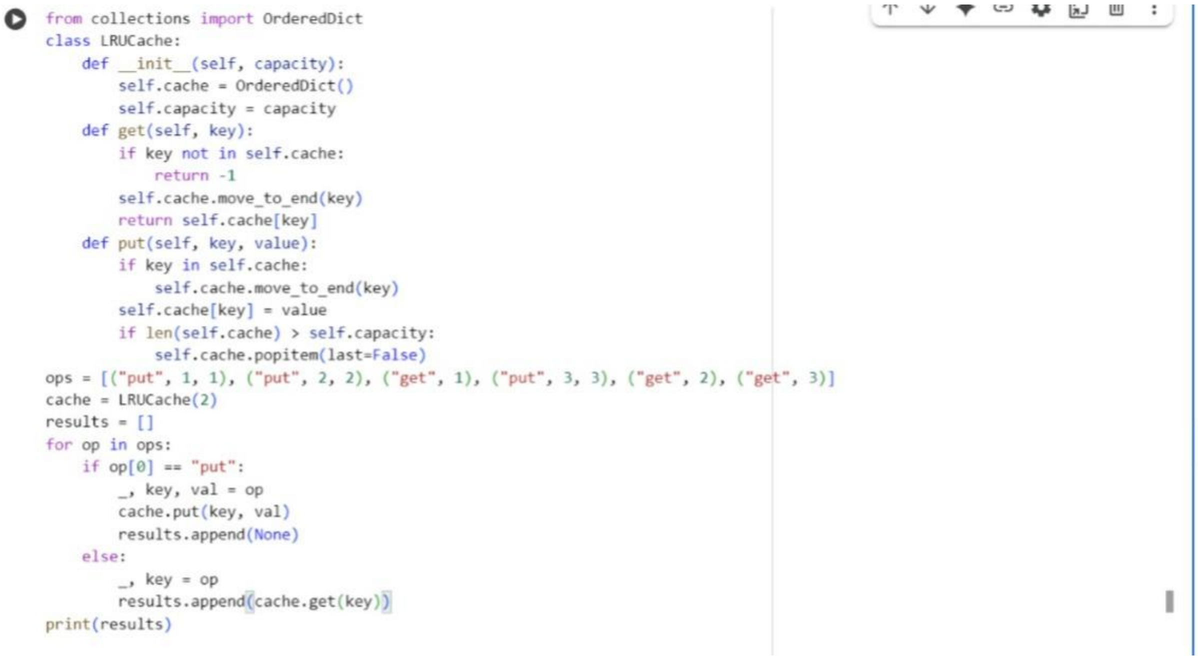
OBSERVATION:

1. For the first input, both "be" and "to" appear twice. Since frequencies tie at 2, they are sorted lexicographically ascending: "be" < "to". So "be" comes before "to".
2. For the second input, "apple" and "banana" both appear twice. Alphabetically, "apple" < "banana", so "apple" is first.
3. For the third input, all words appear once, so tie-break is purely lex order ascending. "extra" < "output" < "three".
4. . The overall sorting principle is frequency descending, then lex ascending for ties.

QUESTION: L.2

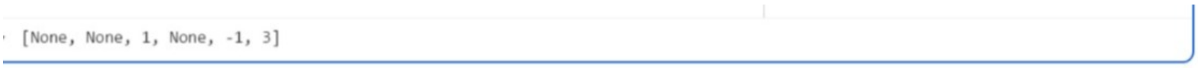
PROMPT: Implement an LRU (Least Recently Used) Cache with capacity 2, designed for urban public transit services like metro or bus systems

CODE:



```
from collections import OrderedDict
class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity
    def get(self, key):
        if key not in self.cache:
            return -1
        self.cache.move_to_end(key)
        return self.cache[key]
    def put(self, key, value):
        if key in self.cache:
            self.cache.move_to_end(key)
        self.cache[key] = value
        if len(self.cache) > self.capacity:
            self.cache.popitem(last=False)
ops = [("put", 1, 1), ("put", 2, 2), ("get", 1), ("put", 3, 3), ("get", 2), ("get", 3)]
cache = LRUCache(2)
results = []
for op in ops:
    if op[0] == "put":
        _, key, val = op
        cache.put(key, val)
        results.append(None)
    else:
        _, key = op
        results.append(cache.get(key))
print(results)
```

OUTPUT:



```
[None, None, 1, None, -1, 3]
```

OBSERVATION:

1. After put(1,1) and put(2,2), the cache stores {1:1, 2:2}.

2. get(1) accesses key 1, making it most recently used, returning 1.

3. put(3,3) adds a new key; since capacity is 2, it evicts the least recently used key, which is 2.

4. get(2) returns -1 because 2 was evicted.

5. get(3) returns 3.