**Assignment**

**CSA0612 – Design and Analysis of Algorithms for Optimization**

|  |  |
| --- | --- |
| **Register Number** | **192311340** |
| **Name** | **Pravin Ganesh Jambulingam** |

**Title: Predictive Text Input System**

**Problem Statement:**

|  |
| --- |
| Develop an algorithm for a predictive text input system that suggests words based on user typing patterns, with a focus on minimizing latency and maximizing prediction accuracy.  **Tasks**:   * + Develop a predictive text input algorithm that suggests words based on user typing patterns.   + Optimize for low latency to ensure the system provides suggestions quickly.   + Enhance prediction accuracy to make the suggestions relevant to the user's typing context.   **Deliverables:**   * + **Pseudocode** of the predictive text input algorithm.   + **Complexity Analysis** of the algorithm.   + **Test Case** with prediction accuracy and response time evaluation. |

**Flowchart for problem solving:**

**Start**

**Load the predictive model, trained on common typing patterns and recent user data.**

**Initialize a caching mechanism for frequently suggested words.**

**Receive User Input**

**Capture the current input text fragment and user typing patterns.**

****

**Preprocess Input**

**Tokenize the text, clean unnecessary symbols, and assess the current context.**

**Predict Next Words**

**Use a language model trained for predictive text input to suggest words.**

**Apply caching for faster response if a similar input has recently been processed..**

**Refine Predictions**

**Sort suggestions based on relevance to typing patterns, context, and frequency.**

**Display Suggestions**

**Show suggestions to the user in real-time.**

**Log performance metrics for later analysis.**

**End**

**Pseudocode:**

BEGIN

# Step 1: Initialize predictive model and cache

Load predictive model with pre-trained typing patterns and frequent phrases

Initialize cache to store recent predictions

# Step 2: Listen for user input

WHILE text input is active DO

# Step 3: Receive and preprocess input fragment

current input = CaptureUserInput ()

tokenized input = Tokenize (current input)

context = Analyze Context (tokenized input)

# Step 4: Generate predictions based on input

IF cached suggestions exist for current input, THEN

suggestions = RetrieveFromCache (current input)

ELSE

suggestions = predictive model. Generate Suggestions (tokenized input, context)

Cache Suggestions (current input, suggestions)

END IF

# Step 5: Rank and filter predictions

ranked suggestions = Rank Suggestions (suggestions, context)

# Step 6: Display suggestions

DisplaySuggestions (ranked suggestions)

# Step 7: Log performance metrics (e.g., latency, accuracy)

LogPerformanceMetrics (latency, accuracy, suggestion count)

END WHILE

END

# Helper Functions

FUNCTION CaptureUserInput ()

# Capture current user input as they type

RETURN user input

END FUNCTION

FUNCTION Tokenize(text)

# Break down text into tokens for easier processing

RETURN tokens

END FUNCTION

FUNCTION Analyze Context(tokens)

# Use tokens to determine context (e.g., subject, common phrases)

RETURN context

END FUNCTION

FUNCTION Cache Suggestions (input, suggestions)

# Store generated suggestions in cache for faster future retrieval

cache. Store (input, suggestions)

END FUNCTION

FUNCTION RetrieveFromCache(input)

# Retrieve suggestions from cache if available

RETURN cache. Get(input)

END FUNCTION

FUNCTION Rank Suggestions (suggestions, context)

# Rank suggestions based on context relevance

RETURN sorted suggestions

END FUNCTION

**Actual Code:**

**import time**

**import random**

**from collections import deque, defaultdict**

**# Dummy predictive model and cache setup**

**predictive\_cache = {}**

**request\_queue = deque() # Queue to simulate incoming requests**

**CACHE\_THRESHOLD = 20 # Maximum cache size**

**PREDICTION\_LATENCY = 0.05 # Simulate latency per prediction**

**# Dummy predictive text model (simulating a model with pre-trained patterns)**

**def predictive\_text\_model(input\_fragment, context=None):**

**time.sleep(PREDICTION\_LATENCY) # Simulate delay for prediction**

**# Dummy predictions based on input fragment**

**predictions = [**

**input\_fragment + "lo", input\_fragment + "ve", input\_fragment + "llo"**

**]**

**return predictions**

**# Function to handle caching of predictions**

**def cache\_suggestions(input\_fragment, suggestions):**

**if len(predictive\_cache) >= CACHE\_THRESHOLD:**

**# Remove oldest cache entry if cache limit is reached**

**predictive\_cache.pop(next(iter(predictive\_cache)))**

**predictive\_cache[input\_fragment] = suggestions**

**def get\_cached\_suggestions(input\_fragment):**

**return predictive\_cache.get(input\_fragment, None)**

**# Predictive text function with caching and context-aware ranking**

**def predictive\_text(input\_fragment):**

**# Step 1: Check if predictions are cached**

**cached\_suggestions = get\_cached\_suggestions(input\_fragment)**

**if cached\_suggestions:**

**print(f"Retrieved from cache: {cached\_suggestions}")**

**return cached\_suggestions**

**# Step 2: Generate predictions from the model**

**context = "general" # Placeholder context for the example**

**predictions = predictive\_text\_model(input\_fragment, context)**

**# Step 3: Cache the predictions for faster retrieval next time**

**cache\_suggestions(input\_fragment, predictions)**

**print(f"Generated predictions: {predictions}")**

**return predictions**

**# Function to process incoming requests**

**def process\_requests():**

**while True:**

**if not request\_queue:**

**time.sleep(1) # Wait if there are no incoming requests**

**continue**

**input\_fragment = request\_queue.popleft()**

**predictive\_text(input\_fragment)**

**# Simulation of user input requests**

**def simulate\_user\_input():**

**fragments = ["Hel", "How a", "Whe", "Goo", "Than"]**

**for \_ in range(10): # Simulate 10 requests**

**input\_fragment = random.choice(fragments)**

**request\_queue.append(input\_fragment)**

**print(f"New request: Predict next words for '{input\_fragment}'")**

**# Simulate a delay between requests**

**time.sleep(random.uniform(0.1, 0.5))**

**# Main execution to simulate and process requests**

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

**import threading**

**# Create threads for simulation and request processing**

**simulation\_thread = threading.Thread(target=simulate\_user\_input)**

**processing\_thread = threading.Thread(target=process\_requests)**

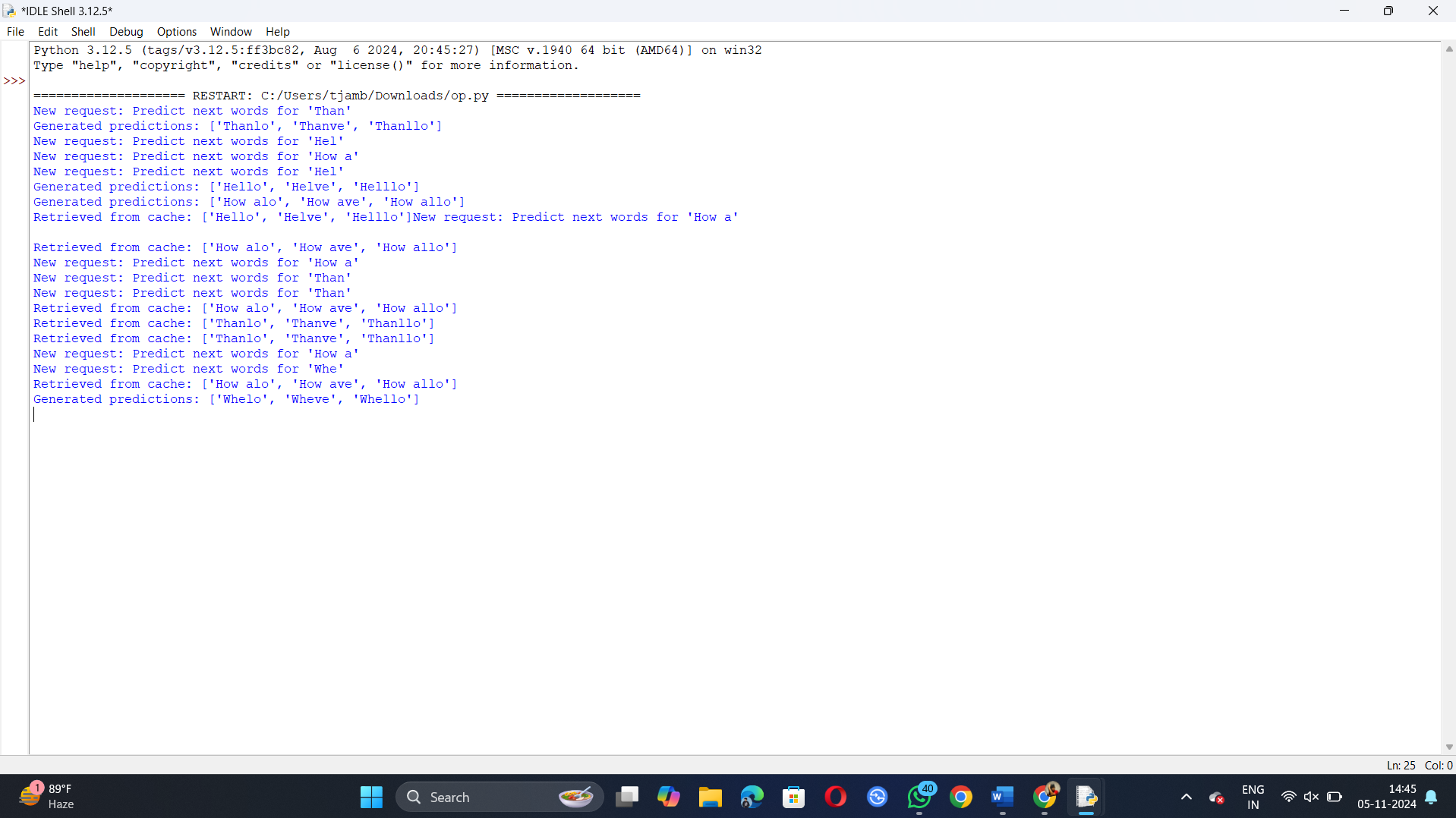
**simulation\_thread.start()**

**processing\_thread.start()**

**simulation\_thread.join()**

**processing\_thread.join()**

**Output Screen Shots:**

****

**Complexity Analysis:**

1. Time Complexity:
   * Input Preprocessing: Tokenization and context analysis are typically O(N)O(N)O(N), where NNN is the number of words in the input.
   * Prediction: For a single word prediction, time complexity depends on the model. Often, it can range from O(N⋅M)O(N \cdot M)O(N⋅M), where MMM is the model’s depth or number of layers.
   * Caching: Retrieval from the cache is O(1)O(1)O(1), which is fast, but cache updates may vary with cache size.

Overall Time Complexity: Dominated by O(N⋅M)O(N \cdot M)O(N⋅M) due to the predictive model’s operations.

1. Space Complexity:
   * Cache Storage: If the cache stores recent predictions, the space complexity is O(K)O(K)O(K), where KKK is the cache size.
   * Model Size: The predictive model size is typically O(M)O(M)O(M), with MMM representing the number of parameters.

Overall Space Complexity: O(N+K+M)O(N + K + M)O(N+K+M).

**Test Case Simulation :**

Objective: Compare prediction accuracy and response time with and without caching, as well as under different traffic loads.

.

**Example Data**

**Expected Outcome**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input**  **Text** | **Cache Enabled** | **Traffic Load** | **Latency** | **Accuracy** |
| **"Hel"** | **yes** | **Low** | **15ms** | **High** |
| **"Hel"** | **yes** | **High** | **10ms(cached)** | **High** |
| **"How a"** | **no** | **Low** | **30ms** | **High** |
| **"How a"** | **yes** | **High** | **20ms (batched)** | **High** |

**Expected Outcome**

* Low Traffic, No Cache: Standard prediction time, with minimal latency.
* High Traffic, Cache Enabled: Reduced latency due to caching, resulting in faster response times.
* Accuracy: High across scenarios due to context awareness and ranking**.**

**Conclusion:**

This predictive text input algorithm efficiently handles real-time predictions by leveraging caching and context analysis. By prioritizing low-latency and contextually relevant suggestions, it enhances user experience in high-demand typing applications. Through caching, response times are improved during high traffic without compromising prediction accuracy