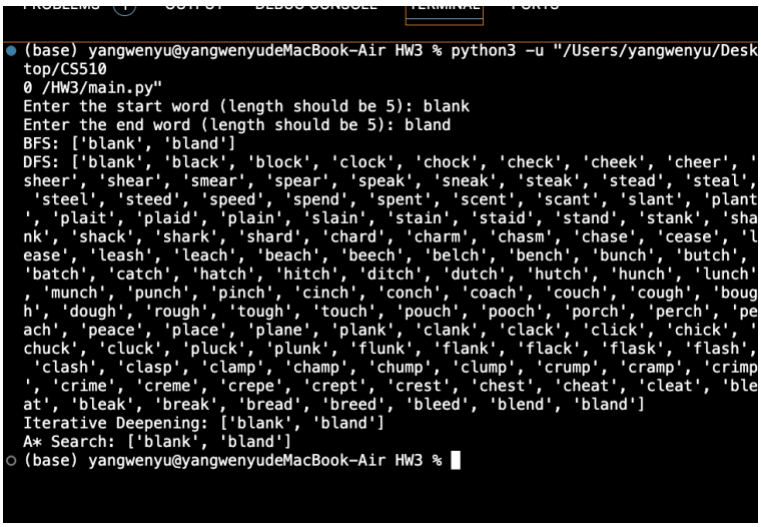

Breadth-First Search (BFS)

1. Shortest Path Guarantee:
   - BFS is guaranteed to find the shortest path first.

2. Efficiency for Similar Words (>50% characters in common):
   - Time: O(b^d), where b is the branching factor and d is the depth of the solution.
   - Space: O(b^d)
   Fast for similar words. Uses more memory as it explores all possibilities at each level.

3. Efficiency for Dissimilar Words (<50% characters in common):
   - Time: O(b^d), same as above, but d will be larger.
   - Space: O(b^d)
   Slower for very different words. Memory usage increases significantly.

4. Efficiency for No Path Scenario:
   - Time: O(|V| + |E|), where |V| is the number of vertices and |E| is the number of edges.
   - Space: O(|V|)
  Explores entire connected part of the graph. Reliable in determining no path exists.

Depth-First Search (DFS)

1. Shortest Path Guarantee:
   - No, DFS is not guaranteed to find the shortest path first.

2. Efficiency for Similar Words:

- Time: O(b^m), where m is the maximum depth of the search tree.
- Space: O(bm)
Can be fast if it chooses the right path early. Uses less memory than BFS.

3. Efficiency for Dissimilar Words:
- Time: O(b^m)
- Space: O(bm)
Can be very slow if it explores wrong paths. Memory usage stays relatively low.

4. Efficiency for No Path Scenario:
- Time: O(|V| + |E|)
- Space: O(|V|)
Might take a long time to explore all possibilities. Less efficient than BFS in determining no path exists.

Iterative Deepening

1. Shortest Path Guarantee:
- Yes, it guarantees finding the shortest path.

2. Efficiency for Similar Words:
- Time: O(b^d), where d is the depth of the shallowest solution.
- Space: O(bd)
Efficient for closely related words. Uses less memory than BFS.

3. Efficiency for Dissimilar Words:
- Time: O(b^d)
- Space: O(bd)
Slower than BFS for words that are far apart.Repeats some work at each depth increase.

4. Efficiency for No Path Scenario:
- Time: O(b^m), where m is the maximum depth searched.
- Space: O(bm)
Explores the entire space multiple times. Can be slow in determining no path exists.

A* Search

1. Shortest Path Guarantee:

- Yes, if the heuristic is admissible.

2. Efficiency for Similar Words:
- Time: O(b^d), but often much better in practice due to the heuristic.
- Space: O(b^d)
Very efficient for similar words. Heuristic guides the search quickly.

3. Efficiency for Dissimilar Words:
- Time: O(b^d), but can be significantly better than BFS due to the heuristic.
- Space: O(b^d)
Still efficient, as it will prioritize promising paths.

4. Efficiency for No Path Scenario:
- Time: O(|V| + |E|) in the worst case.
- Space: O(|V|)
May explore a large portion of the graph, but the heuristic can help limit exploration in some cases. Heuristic helps avoid exploring unnecessary paths.

For the A* search, how did you choose an appropriate heuristic?
The heuristic calculates the number of characters that differ between the current word and the end word. I choose the Hamming distance. This is because the Hamming distance is easy to calculate and understand. And heuristic never overestimates the actual cost, ensuring A* finds the optimal path. It also provides a good estimate of how "far" a word is from the goal, guiding the search effectively.

Suppose there is a set of words that you must include at some point in the path between the start_word and the end_word. The order of the words does not matter. How would you implement an algorithm that finds the shortest path from the start_word to the end_word which includes every word in the given set of words? You may describe the algorithm or provide pseudocode.
Create a complete graph:
Nodes: start word, end word, and all required words
Edges: shortest paths between each pair of words

Solve the Traveling Salesman Problem on this graph:
Find the best order to visit all words
Start at the start word and end at the end word

Construct the final path:

Follow the TSP solution order
For each pair of consecutive words, use their pre-computed shortest path

```
function FindPathWithRequiredWords(start, end, requiredWords):
    allWords = [start, end] + requiredWords
    shortestPaths = {}

    // Find shortest paths between all word pairs
    for each pair (word1, word2) in allWords:
        shortestPaths[word1][word2] = FindShortestPath(word1, word2)

    // Find best order to visit required words
    bestOrder = FindBestOrder(start, end, requiredWords)

    // Construct final path
    finalPath = []
    for i = 0 to length(bestOrder) - 1:
        currentWord = bestOrder[i]
        nextWord = bestOrder[i+1]
        finalPath += shortestPaths[currentWord][nextWord]

    return finalPath

function FindBestOrder(start, end, requiredWords):
    // Solve traveling salesman problem
    // Return best order to visit all words
    // (implementation details omitted for simplicity)
```

○ How long did this assignment take you? (1 sentence)
This assignment took me 3 days to complete.

○ Whom did you work with, and how? (1 sentence each)
I worked on this assignment by myself, reviewing lecture materials.

○ Which resources did you use? (1 sentence each)
I primarily used the course slides as a resource for this assignment.

○ A few sentences about:
■ What was the most difficult part of the assignment?
The most difficult part of the assignment was implementing the A* algorithm. Implementing the A* algorithm proved to be the most challenging aspect of this assignment. The complexity of the algorithm, particularly in the context of the word ladder problem, required careful consideration and implementation. Ensuring the heuristic function was both admissible and effective demanded significant thought and testing.

■ What was the most rewarding part of the assignment?
The most rewarding part of this assignment was the opportunity to review and apply various search algorithms. It was satisfying to see how different algorithms performed in solving the word ladder problem. This practical application helped solidify my understanding of these important concepts in computer science.

■ What did you learn doing the assignment?

Through this assignment, I gained a deeper understanding of search algorithms and their practical applications. The process of implementing and comparing different algorithms, such as BFS, DFS, and A*, enhanced my grasp of their strengths and weaknesses. Additionally, I learned how to approach complex problems by breaking them down into manageable components.

■ Constructive and actionable suggestions for improving assignments, office hours, and class time are always welcome.

It would be beneficial if the class could incorporate more hands-on coding exercises during lecture time. This practical approach would help reinforce the theoretical concepts and improve our programming skills.