ABIN MID SEM RUBRICS

Question 1 Rubric

1.1 What is memoization in the context of Dynamic Programming? Explain with an example. (2 Marks)

Key Points for Memoization:

- Definition of memoization.
- Explanation of how it is used in dynamic programming.
- Example showcasing the principle of storing previously computed results.

Rubric:

2 Marks:

•Clear and correct definition of memoization.

•Correctly explains how it avoids redundant calculations by storing previous computations.

•Provides a valid example (e.g., Fibonacci series, matrix chain multiplication) to support the explanation.

1 Mark:

•Partial explanation, e.g., defining memoization but failing to provide a valid or detailed example.

0 Marks:

•Incorrect or irrelevant answer, no mention of the core concept.

Plausible Answers:

1. Memoization is a technique where we store previously calculated values to avoid redundant calculations in dynamic programming. For example, in the Fibonacci series, once we calculate Fibonacci(5), we store its value so that when it's needed again, we don't recompute it.
2. Memoization is a way to optimize recursive algorithms by storing already calculated results. In dynamic programming, this helps in improving efficiency by

avoiding recomputation. For example, when solving matrix chain multiplication, we save result subproblems in a table.

3. Memoization involves storing the results of expensive function calls and reusing them when the same inputs occur again. It is a top-down approach in dynamic programming, for instance, in calculating factorials or Fibonacci numbers recursively.

1.2 State two applications for each of the global and local alignments. (2 Marks)

Key Points for Global and Local Alignment:

•Correctly identifying global and local alignment as sequence comparison techniques.
•Providing practical examples for both global and local alignments.

Rubric:

2 Marks:
•Two valid applications for global alignment.
•Two valid applications for local alignment.
1 Mark:
•One valid application for global alignment and/or one valid application for local alignment.
0 Marks:
•No valid applications mentioned.

Plausible Answers:

1. Global alignment is used for comparing sequences when they are of similar length and origin, like DNA sequence comparison between different species. Local alignment is used for finding similar regions within long sequences, like identifying conserved protein domains.
2. Global alignment helps in comparing gene sequences between two organisms for evolutionary studies, while local alignment is useful for detecting short conserved sequences like motifs in proteins.

3. Global alignment is used for full genome alignments in comparative genomics, while local alignment helps identify similar sub-sequences, such as detecting a virus sequence within a host genome.

1.3 What are algorithms and their complexities associated with finding Hamming distance and Edit distance between two strings? Clearly explain. (4 Marks)

Key Points for Hamming and Edit Distance:

•Correct identification of both algorithms.
•Correct explanation of how each algorithm works.
•Complexity for both algorithms.

Rubric: 4 Marks:

- Correct explanation of Hamming distance and its O(n) complexity.
- Correct explanation of Edit distance (Levenshtein distance) and its O(m*n) complexity.
- Provides clear, concise explanations of how these algorithms function.

3 Marks:

- Partial explanation, such as explaining one of the algorithms well and the other only partially.

2 Marks:

- Incomplete explanation for both algorithms or missing complexities.

1 Mark:

- Mention of algorithms but lacks explanation or complexity.

0 Marks:

- Incorrect or irrelevant answer.

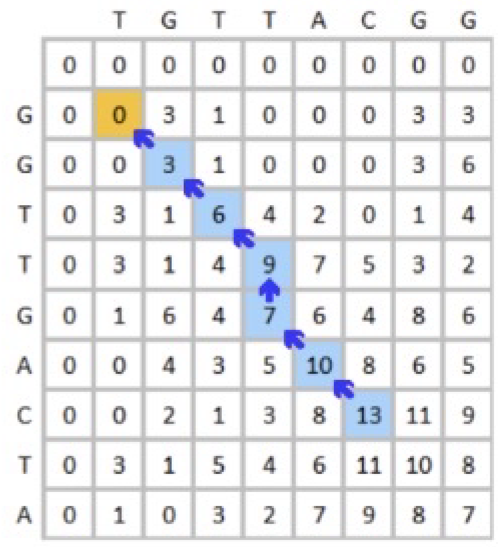
Plausible Answers:

1. Hamming distance is the number of positions at which two strings of equal length differ. Its complexity is O(n), where n is the string length. Edit distance, also known as Levenshtein distance, calculates the minimum number of insertions, deletions, or substitutions needed to change one string into another, with a time complexity of O(m*n).

2. Hamming distance counts mismatches in two equal-length strings, with O(n) complexity. Edit distance uses dynamic programming to compute the minimum operations needed to transform one string into another, with O(m*n) complexity.

3. Hamming distance is used when the two strings are of the same length and compares character by character (O(n)). Edit distance is more general and calculates the transformation needed to convert one string into another with operations such as insertion, deletion, or substitution (O(m*n)).

1.4 For the below case of local alignment, provide the alignment of the two strings with gaps based on the filled matrix in the image. (2 Marks)



		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1	0	0	0	3	6
T	0	3	1	6	4	2	0	1	4
T	0	3	1	4	9	7	5	3	2
G	0	1	6	4	7	6	4	8	6
A	0	0	4	3	5	10	8	6	5
C	0	0	2	1	3	8	13	11	9
T	0	3	1	5	4	6	11	10	8
A	0	1	0	3	2	7	9	8	7

Key Points for Alignment:

- Correct identification of the optimal local alignment based on the provided matrix.
- Correct use of gap penalties where applicable.
- Correct sequence of alignment corresponding to the highest scoring path.

Rubric:

2 Marks:

- Correct identification of the optimal local alignment from the matrix.
- Correct alignment with gap insertions in the appropriate places.

0 Marks:

- Incorrect alignment or no alignment provided.

Plausible Answers:

1. Based on the matrix, the optimal local alignment is:

```
GGTTGACTA
*|||*||**
TGTT_ACGG
```

The alignment follows the highest scoring path with appropriate gap placement. Pipes are alignment, gaps are mismatches/indels.

2.Write the pseudocode for alignment backtracking using the Smith-Waterman (SW) algorithm

Key steps in the algorithm:

1.Matrix Initialization: A matrix is created with dimensions (n+1) by (m+1), where n and m are the lengths of the two sequences being compared. The first row and column are initialized to zero.

2.Matrix Filling: The matrix is filled using a scoring function based on match/mismatch scores and gap penalties.

3.Backtracking: Once the matrix is filled, the highest score indicates the end of the best local alignment. From that position, backtracking is performed to trace back the optimal alignment.

Rubric:

10 Marks:

- Complete and correct pseudocode for the Smith-Waterman algorithm.
- Includes all necessary steps: matrix initialization, filling based on scoring, and backtracking to extract the alignment.
- Well-organized and readable code structure.
- Accurate description of gap penalties and match/mismatch scoring.

6-8 Marks:

- Partially correct pseudocode (e.g., missing some initialization details or backtracking steps).
- The overall flow of the algorithm is mostly accurate.
- Minor mistakes in variable definitions or scoring steps.

3-4 Marks:

- Incomplete pseudocode, missing crucial parts such as matrix initialization or scoring logic.
- A general idea of the SW algorithm is conveyed, but no valid pseudocode structure is provided.

0 Marks:

- Incorrect or irrelevant answer, with no valid pseudocode or incorrect algorithm.

Plausible Answers for SW Algorithm Pseudocode & Complexity:

1.Pseudocode detailed:

```
# Fill the matrix
3. for i from 1 to n:
     for j from 1 to m:
        if seq1[i-1] == seq2[j-1]:
           match = M[i-1][j-1] + match_score
        else:
           match = M[i-1][j-1] - mismatch_penalty

        delete = M[i-1][j] - gap_penalty
        insert = M[i][j-1] - gap_penalty

            # The Smith-Waterman condition ensures that we choose max(0, ...) for local
alignment
        M[i][j] = max(0, match, delete, insert)

        # Track the position of the highest scoring cell
        if M[i][j] > max_score:
           max_score = M[i][j]
```

```
                max_position = (i, j)

# Backtracking to get the best local alignment
4. i, j = max_position
5. aligned_seq1 = ""
6. aligned_seq2 = ""

7. while M[i][j] != 0:   # Continue backtracking until a cell with score 0 is reached
     if M[i][j] == M[i-1][j-1] + match_score or M[i][j] == M[i-1][j-1] - mismatch_penalty:
        aligned_seq1 = seq1[i-1] + aligned_seq1
        aligned_seq2 = seq2[j-1] + aligned_seq2
        i = i - 1
        j = j - 1
     elif M[i][j] == M[i-1][j] - gap_penalty:
        aligned_seq1 = seq1[i-1] + aligned_seq1
        aligned_seq2 = "-" + aligned_seq2   # Gap in seq2
        i = i - 1
     else:
        aligned_seq1 = "-" + aligned_seq1   # Gap in seq1
        aligned_seq2 = seq2[j-1] + aligned_seq2
        j = j - 1

8. return aligned_seq1, aligned_seq2, max_score
```

Time Complexity Analysis:

•Matrix Filling: The time complexity for filling the matrix is O(n * m), where n and m are the lengths of the two sequences. This is because every cell in the matrix needs to be computed based on its neighbors.

•Backtracking: The time complexity for the backtracking step is at most O(n + m), as the algorithm traces a single path through the matrix, which in the worst case may involve visiting every row and column.

Thus, the overall time complexity is dominated by the matrix-filling step, giving an O(n * m) complexity for the algorithm.

OR

2.1 Write the pseudocode of the Pattern Branching or greedy motif search algorithm and provide a detailed explanation of its time complexity. (5 Marks)

Key Points for Pattern Branching Algorithm:

- Correct representation of pattern branching (e.g., k-mers and their extensions).
- Handling of mismatches or "branching" out from patterns.
- Detailed explanation of how the algorithm explores potential alignments.
- Complexity analysis that clearly explains time complexity based on input size.

Rubric:

5 Marks:

- Complete and correct pseudocode for the Pattern Branching algorithm.
- Clear handling of branching and mismatch cases during pattern exploration.
- Correct complexity analysis, explaining that time complexity depends on the number of mismatches and the pattern length (e.g., O(n * d) where n is the number of patterns and d is the maximum number of allowed mismatches).
- Detailed and correct explanation of each step in the algorithm.

3-4 Marks:

- Mostly correct pseudocode, but missing minor details in handling mismatches or pattern exploration.
- Correct complexity analysis, but may lack clarity or depth in explanation.

1-2 Marks:

- Incomplete pseudocode, missing key steps of pattern extension or branching.
- Limited or incorrect time complexity explanation.

0 Marks:

- Incorrect or irrelevant pseudocode, or no valid explanation of complexity.

Plausible answers:

1. Pseudocode:

```
1. Input: A pattern P and a set of strings S.
2. Initialize a list of patterns to explore, starting with P.
3. For each string s in S:
   - For each pattern p in the list of patterns:
     - Generate all possible extensions of p by allowing up to d mismatches.
     - Compare each extension to the current string s.
4. If a match is found, record it.
5. Continue exploring until all patterns have been extended or matches found.
```

2.Explanation of Complexity:

The time complexity of the Pattern Branching Algorithm for a DNA sequence consisting of n nucleotides and t sequences, with a pattern of length l and up to d allowed mismatches, is approximately O(l^d * t * n) . For a pattern of length l with up to d mismatches, the number of possible branches (mutations) is:

$$\text{Number of branches} = \sum_{i=0}^{d} \binom{l}{i} \cdot 3^i$$

Searching for a pattern of length l in a sequence of length n can be done in O(n). Since there are t sequences, the cost of searching for each branch in all sequences is O(t * n) . The total complexity of the algorithm is the number of branches generated, multiplied by the cost of searching each branch in all sequences.

- Therefore, the time complexity is:

$$O\left(\left(\sum_{i=0}^{d} \binom{l}{i} \cdot 3^i\right) \cdot t \cdot n\right)$$

For large values of d, this simplifies to approximately:

O(l^d * t * n) or

$$O\left(\binom{l}{d} \cdot 3^d \cdot t \cdot n\right)$$

Key Points for Greedy Motif Search Algorithm:

•Random motif initialization: The algorithm starts by randomly selecting one k-mer from each of the sequences.

•Iteration over a fixed number of trials: The algorithm performs multiple iterations, each time starting with different randomly selected k-mers.

•Profile matrix construction: A profile matrix is built using the selected motifs, which represents nucleotide frequencies.

•Motif scoring: For each iteration, motifs are evaluated by how well they fit the profile, and the score is calculated.

•Greedy motif selection: In each iteration, the algorithm updates the motif set to maximize the consensus score.

•Complexity analysis: Time complexity depends on the number of sequences (t), their length (n), the motif length (k), and the number of iterations (I).

Rubric:

5 Marks:

•Complete and correct pseudocode that incorporates all steps: random initialization of motifs, profile matrix construction, motif scoring, and iteration through multiple trials.

•Accurate handling of the motif selection process, including updates based on the profile and comparison to previous motif sets.

•Correct time complexity analysis, explaining that the time complexity is dependent on the number of iterations (I), the number of sequences (t), sequence length (n), and motif length (k). The overall complexity should be expressed as O(I * t *t n * k) , where I is the number of iterations, t is the number of sequences, n is the sequence length, and k is the motif length.

3-4 Marks:

•Mostly correct pseudocode, but missing minor details, such as how motifs are randomly selected during iterations or how the profile is constructed.

- Reasonable time complexity analysis, but may lack depth or may not clearly explain all factors contributing to the complexity.
- Partial explanation of the algorithm, with most of the key steps, but may omit important details like profile matrix updates or motif selection.

1-2 Marks:

- Incomplete pseudocode, missing key steps like profile creation or motif updates.
- Limited or incorrect complexity analysis, possibly ignoring significant factors affecting complexity.
- Brief or unclear explanation, not fully explaining how the greedy search operates or how motifs are selected.

0 Marks:

- Incorrect or irrelevant pseudocode, failing to represent the Greedy Motif Search algorithm in a valid form.
- No valid complexity analysis, or the explanation is entirely wrong.
- Incorrect or no explanation of key parts of the algorithm, like motif selection or profile construction.

Plausible Answer:
Input: A set of t DNA sequences, each of length n, and motif length k
Output: A set of k-mers (motifs), one from each sequence, that maximizes the score

```
1. Function GreedyMotifSearch(DNA, k, t, I):
2.    Initialize BestMotifs randomly by selecting one k-mer from each sequence
3.    For iteration in range(1 to I):   # Perform I iterations
4.        Randomly select one k-mer from each sequence as initial Motifs
5.            Construct a profile matrix from the current Motifs
6.            From this profile matrix, find the P-most a probable l-mer in each sequence and
change the starting position to the starting position of a.
7.        If the score of the current Motifs is better than the score of BestMotifs:
8.            Update BestMotifs to the current Motifs
9.    Return BestMotifs

Function Score(Motifs):
    # Calculate how well the motifs agree across all sequences
    # Sum the score of each column in the motifs matrix (most frequent base at each position)

Function Profile(Motifs):
    # Build a profile matrix from the motifs to track nucleotide frequencies
    # For each nucleotide at each position, compute the frequency of occurrence

Function MostProbableKmer(sequence, k, profile):
    # Find the k-mer in the sequence that has the highest probability according to the given profile
    # For each k-mer in the sequence, calculate its probability using the profile
```

Explanation:

1.Initialization: The algorithm starts by selecting a random k-mer from each sequence. This initial set of k-mers serves as the starting point for the greedy motif search.

2.Iterations: The algorithm performs a fixed number of iterations (I). In each iteration, a new set of initial k-mers is randomly chosen from the sequences, and these motifs are evaluated.

3.Profile Matrix Construction: For each iteration, a profile matrix is constructed based on the current set of motifs. The profile represents the frequency of each nucleotide (A, C, G, T) at each position of the motifs.

4.Scoring and Updating: The motifs are scored based on how well they align with each other (how well they fit the profile). If the score improves compared to the previous best motif set, the current set is saved as the best one.

5.Greedy Selection: In each iteration, the algorithm selects the most probable k-mers from the sequences based on the profile matrix. This step is repeated for each sequence until a complete motif set is built.

Time Complexity:

The time complexity of Greedy Motif Search depends on several factors:

1. Number of Iterations (I): The algorithm performs a fixed number of iterations to explore different sets of motifs.
2. Motif Selection: For each iteration, the algorithm selects a k-mer from each of the t sequences, where each sequence has length n . Finding the most probable k-mer for each sequence takes O(n \cdot k) .
3. Profile Construction: Constructing a profile matrix from t motifs takes O(t \cdot k) , where t is the number of sequences and k is the motif length.

Thus, the overall time complexity is:

$$O(I \cdot t \cdot n \cdot k)$$

Where:

- I is the number of iterations (a fixed parameter).
- t is the number of sequences.
- n is the length of each sequence.
- k is the motif length.

This complexity reflects the fact that in each iteration, the algorithm evaluates a new set of motifs, builds a profile, and scores the motifs.

2.2 Burrows-Wheeler Transformation is lossless and can accelerate search. Explain with an example sequence of nucleotides. (5 Marks)

Key Points for Burrows-Wheeler Transformation (BWT):

- Explanation of how BWT works.
- Explanation of why BWT is a lossless transformation.
- Example using a nucleotide sequence, showing transformation and searching steps.
- Highlighting how BWT speeds up searching (e.g., FM-index, better compression, etc.).

Rubric:

5 Marks:

- Complete and correct explanation of BWT, including why it is lossless.
- Example using a valid nucleotide sequence, showing transformation and steps.
- Clear and accurate explanation of how BWT accelerates search (e.g., FM-index or backward search).
- Thorough explanation of why it is lossless, i.e., the original sequence can be fully reconstructed.

3-4 Marks:

- Mostly correct explanation of BWT, but some minor details are missing (e.g., incomplete transformation example or vague explanation of acceleration).
- Example sequence is mostly accurate but may lack some steps.

1-2 Marks:

- Incomplete explanation of BWT.
- Example provided, but missing key steps or explanations (e.g., lack of clarity on why it is lossless or how it accelerates searching).

0 Marks:

- Incorrect or irrelevant explanation with no valid example or no discussion on the lossless aspect or acceleration of search.

Plausible Answers:

1.Explanation:

Burrows-Wheeler Transformation (BWT) is a reversible transformation used in data compression and text searching. It works by rearranging the string's characters into a format that is more amenable to compression. The transformation is lossless because the original sequence can be recovered using the transformation's output and an additional index.

2.Example:

Consider the nucleotide sequence ATCG. The BWT involves rotating the sequence and sorting the rotations:

- Rotations: ATCG, TCGA, CGAT, GATC

- Sorted rotations: ATCG, CGAT, GATC, TCGA
- BWT result: GCTA

To accelerate search, the BWT allows backward searching, using an index like FM-index, which enables fast substring searches without fully decompressing the sequence. For example, searching for the pattern CG is accelerated as BWT reorganizes the data in a way that groups similar patterns together.

3.Explanation of Losslessness:

The transformation is lossless because we can use techniques like last-to-first mapping to reconstruct the original sequence.

3. **This question requires the use of the Viterbi algorithm to find the most likely sequence of states (Exon or Intron) given a nucleotide sequence and a Hidden Markov Model (HMM) with transition, emission, and initial probabilities.**

Rubric

1. Correct Setup of the Viterbi Matrix (4 marks):

•Matrix Initialization (2 marks):

•Correctly initializing the Viterbi matrix for the first nucleotide (‘A’), using the given initial probabilities and emission probabilities for both Exon and Intron states.

•Properly setting up the first column of the matrix (time step 1).

Mark Distribution:

- 2 Marks: Both states (Exon and Intron) are initialized correctly with proper expressions using initial and emission probabilities.
- 1 Mark: One of the states is initialized correctly but the other has an error.
- 0 Marks: Incorrect or no initialization.

•Matrix Construction (2 marks):

•Correctly filling out the subsequent columns (time steps 2 and 3) of the Viterbi matrix using the given transition and emission probabilities.

•Using the appropriate recursive formula to propagate the values across the matrix, combining transition and emission probabilities.

Mark Distribution:

- 2 Marks: Correct recursive expressions used for both Exon and Intron states across all columns.
- 1 Mark: Partial correctness in recursive expressions, but a clear effort is made to set up the problem.
- 0 Marks: No valid recursive expressions or no attempt to propagate values across the matrix.

2. Correct Calculation of Expressions (3 marks):

•Recursive Expressions:

•At each time step, students should write out the expression combining transition probabilities and previous state probabilities.

•Correct use of max function to determine the best path for each state at each time step.

Mark Distribution:

•3 Marks: All expressions written out correctly, combining previous probabilities, transition, and emission probabilities at each time step.

•2 Marks: Some expressions are correct, but there are minor mistakes (e.g., incorrect transition probabilities or missing max function).

•1 Mark: Only partial expressions are correct, with missing details or incorrect recursive steps.

•0 Marks: No valid expressions or incorrect setup.

3. Backtracking (2 marks):

•Traceback of Path:

•After completing the Viterbi matrix, students need to trace back the path from the final state to the initial state to obtain the most likely sequence of states (Exon or Intron).

•Identify the most probable state in the last column and trace back the path using the matrix.

Mark Distribution:

•2 Marks: Correct backtracking to identify the optimal path.

•1 Mark: Partial backtracking but with errors in identifying the most probable states.

•0 Marks: No attempt at backtracking or a completely incorrect path.

4. Final Predicted State Sequence (1 mark):

- •Prediction of the Most Likely Sequence:
- •Correctly outputting the most likely sequence of states (e.g., Exon -> Intron -> Exon).

Mark Distribution:

- •1 Mark: Correct final sequence of states.
- •0 Marks: Incorrect sequence or no sequence given.

Answer using the Viterbi Algorithm:

The following steps show how to approach the problem and set up the Viterbi matrix step by step.

Step 1: Initialization of the Viterbi Matrix

Let the observed sequence be "ATG" with three positions.

Initialize the matrix for the first observation ('A'):

- •For Exon (E) at time step 1:

$$V_E(1)=P(\text{start in E})\times P(\text{'A' emitted by E})=0.5\times 0.4=0.2$$

- •For Intron (I) at time step 1:

$$V_I(1)=P(\text{start in I})\times P(\text{'A' emitted by I})=0.5\times 0.1=0.05$$

Step 2: Recursive Calculation for Time Step 2 (Observation 'T')

- •For Exon (E) at time step 2:

$$V_E(2)=\max\big(V_E(1)\times P(E\to E),V_I(1)\times P(I\to E)\big)\times P(\text{'T' emitted by E})$$

$$V_E(2) = \max\left(0.2 \times 0.7, 0.05 \times 0.4\right) \times 0.3 = \max(0.14, 0.02) \times 0.3 = 0.14 \times 0.3 = 0.042$$

•For Intron (I) at time step 2:

$$V_I(2) = \max\left(V_E(1) \times P(E \to I), V_I(1) \times P(I \to I)\right) \times P(\text{'T' emitted by I})$$

$$V_I(2) = \max\left(0.2 \times 0.3, 0.05 \times 0.6\right) \times 0.2 = \max(0.06, 0.03) \times 0.2 = 0.06 \times 0.2 = 0.012$$

Step 3: Recursive Calculation for Time Step 3 (Observation 'G')

•For Exon (E) at time step 3:

$$V_E(3) = \max\left(V_E(2) \times P(E \to E), V_I(2) \times P(I \to E)\right) \times P(\text{'G' emitted by E})$$

$$V_E(3) = \max\left(0.042 \times 0.7, 0.012 \times 0.4\right) \times 0.2 = \max(0.0294, 0.0048) \times 0.2 = 0.0294 \times 0.2 = 0.00588$$

•For Intron (I) at time step 3:

$$V_I(3) = \max\left(V_E(2) \times P(E \to I), V_I(2) \times P(I \to I)\right) \times P(\text{'G' emitted by I})$$

$$V_I(3) = \max\left(0.042 \times 0.3, 0.012 \times 0.6\right) \times 0.4 = \max(0.0126, 0.0072) \times 0.4 = 0.0126 \times 0.4 = 0.00504$$

Step 4: Backtracking

Now that we have filled out the Viterbi matrix, we need to backtrack to find the most likely sequence of states.

•At time step 3, the higher value is in Exon (E) (0.00588 vs. 0.00504). Therefore, the most likely state at time step 3 is Exon (E).

•At time step 2, we came to Exon (E) from Exon (E) (since 0.042 came from Exon state at step 1). Therefore, the state at time step 2 is Exon (E).

•At time step 1, we started in Exon (E) (as this gives the higher probability in the first column).

Final State Sequence:

The most likely sequence of states is Exon -> Exon -> Exon.

Conclusion:

- Final Viterbi Matrix:

	Time 1 ('A')	Time 2 ('T')	Time 3 ('G')
Exon	0.2	0.042	0.00588
Intron	0.05	0.012	0.00504

Therefore, Predicted Sequence of States: Exon -> Exon -> Exon.