DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING AND INFORMATION TECHNOLOGY
YEAR/SEM: III/VI
23CSX503, NATURAL LANGUAGE PROCESSING FUNDAMENTALS
UNIT I
INTRODUCTION

QUESTION BANK

Natural Language Processing tasks in syntax, semantics, and pragmatics - challenges of NLP - NLP Phases - Language Modeling: Grammar-based LM, Statistical LM - Regular Expressions, Finite-State Automata – English Morphology, Transducers for lexicon and rules, Tokenization, Detecting and Correcting Spelling Errors, Minimum Edit Distance.

PART - A (2 MARKS)

Introduction

1. Define Natural Language Processing (NLP). **K1**

Natural Language Processing (NLP) is a sub-field of Computer Science and Artificial Intelligence that is concerned with enabling computers to understand, interpret, and process human language in the form of text or speech.

2. State the applications of NLP. **K1**

NLP is used in applications such as

- Machine translation
- Speech recognition
- Question answering systems
- Dialogue-based expert systems
- Spell checking
- Information extraction

3. Name the two main components of NLP. **K1**

The two main components of NLP are:

1. Natural Language Understanding (NLU)
2. Natural Language Generation (NLG)

Natural Language Understanding (NLU) focuses on mapping the given natural language input into useful internal representations and analyzing different aspects of language in order to understand its meaning.

Natural Language Generation (NLG) is the process of producing meaningful phrases and sentences in the form of natural language from some internal representation.

Natural Language Processing Tasks in Syntax, Semantics, and Pragmatics

4. **What is meant by pragmatics?** **K1**
Pragmatics deals with using and understanding sentences in different situations and explains how the interpretation of a sentence is affected by context and real-world knowledge.

5. **Why is POS tagging important in syntactic analysis?** **K2**
POS tagging is important in syntactic analysis because it assigns grammatical categories (such as noun, verb, adjective) to words, which helps in identifying the syntactic structure and grammatical relationships within a sentence.

6. **How does lack of training data limit NLP performance?** **K2**
Lack of high-quality training data limits NLP performance because models require large, clean, and annotated datasets to learn language patterns; insufficient data reduces accuracy, especially for regional or low-resource languages.

Challenges of NLP

7. **List the challenges faced by NLP systems.** **K1**
The major challenges faced by NLP systems include
- language diversity
- ambiguity in natural language
- lack of high-quality training data
- words with multiple meanings (polysemy)
- spelling and grammatical errors
- difficulty in understanding context
- bias in NLP models
- multilingual processing issues
- high computational requirements

8. **Why does ambiguity affect the accuracy of NLP systems?** **K2**
Ambiguity affects the accuracy of NLP systems because the same word or sentence can have multiple meanings depending on context, making it difficult for the system to identify the correct interpretation without deep contextual understanding.

NLP Phases

9. **Why can a sentence be syntactically correct but semantically incorrect?** **K2**
Sentence can be syntactically correct but semantically incorrect because it may follow grammatical rules while failing to convey meaningful or logical sense, as the meaning of words may not be compatible with real-world knowledge (e.g., *"hot ice-cream"*).

10. **List the five phases of NLP.** **K1**
The five phases of NLP are:
1. Lexical Analysis
2. Syntax Analysis (Parsing)

3. Semantic Analysis
4. Discourse Integration
5. Pragmatic Analysis

Language Modeling

11. Define language modeling. **K1**

Language modeling is the process of predicting the next word in a sequence of words by capturing the statistical structure and patterns of a language.
The main objectives of language modelling are:

- To predict the next word in a sentence
- To generate grammatically correct and meaningful sentences
- To capture syntactic and semantic regularities of a language
- To help machines choose the most probable word sequence among alternatives

12. How does a statistical language model predict the next word? **K2**

A statistical language model predicts the next word by calculating the probability of a word based on the frequencies of previous word sequences (n-grams) observed in a large training corpus.

13. Why do grammar-based language models have limited coverage? **K2**

Grammar-based language models have limited coverage because it is difficult to write comprehensive grammar rules that capture all syntactic and semantic variations of a natural language.

Regular Expression-Finite-State Automaton

14. Why are regular expressions considered greedy? **K2**

- Regular expressions are considered greedy because they attempt to match the longest possible string that satisfies the given pattern.
- When multiple matches are possible for a pattern, the regular expression engine prefers the match that consumes the maximum number of characters.
- For example, the pattern .* will match the entire string rather than stopping at the first possible match.
- This greedy behaviour ensures maximum coverage during pattern matching but may also lead to false positives if not carefully designed.

15. How do character classes simplify pattern matching? **K2**

- Character classes simplify pattern matching by allowing multiple alternative characters to be represented within a single expression**.**
- Square brackets [ ] denote a set of characters where any one character can match.
- For example, [a-z] matches any lowercase letter and [0-9] matches any digit.
- This reduces the need to write separate patterns for each character, making regular expressions more compact, readable, and efficient.

16. How does an FSA recognize a valid string? **K2**

- A Finite-State Automaton recognizes a valid string by processing the input symbol by symbol, starting from the initial state.
- For each input symbol, the FSA follows a corresponding transition to the next state.
- If the automaton reaches an accepting (final) state exactly when the input ends, the string is accepted as valid.
- If the input ends early, no valid transition exists, or the automaton stops in a non-final state, the string is rejected**.**

English Morphology

17. How does derivational morphology differ from inflectional morphology? **K2**

Derivational Morphology	Inflectional Morphology
♦ Derivational morphology is concerned with the formation of new words by adding affixes to a root word.	♦ Inflectional morphology deals with modifying a word to express grammatical information such as tense, number, person, case, or comparison.
♦ It often changes the meaning of the word and may also change its grammatical category.	♦ It does not change the core meaning or word class.
♦ For example, adding *–ness* to *happy* forms *happiness*, changing an adjective into a noun.	♦ For example, adding *–s* to *cat* to form *cats* indicates plurality, and adding *–ed* to *play* forms *played*, indicating past tense.
♦ Derivational affixes are not mandatory for grammatical correctness and are applied before inflectional affixes.	♦ Inflectional forms are required by grammar and do not create new dictionary entries.

18. What is a morpheme? **K1**

A morpheme is the smallest meaningful unit of a language that cannot be further divided without losing its meaning. Morphemes may be words or parts of words that carry meaning.

Example 1:

- Word: *unhappiness*
  - un– → morpheme (prefix, meaning *not*)
  - happy → morpheme (root, meaning *joyful*)
  - –ness → morpheme (suffix, meaning *state or quality*)
  - Here, un, happy, and ness are all morphemes because each carries meaning.

Example 2:

- cats = *cat* (root morpheme) + *–s* (plural morpheme)

19. List the types of affixes. **K1**

The types of affixes are:

- Prefix – added before the root word
- Suffix – added after the root word
- Infix – inserted within the root word
- Circumfix – added around the root word

Transducers for Lexicon and Rules

20. What is the role of a lexicon in NLP? **K1**

- A lexicon in NLP is a structured dictionary that stores information about words (stems) along with their morphological, grammatical, and lexical features.
- It plays a crucial role in lexical analysis and morphological processing by mapping surface word forms to their underlying lexical representations.
- In the module, the lexicon transducer maps words from the lexical level (stems + features such as +N, +PL, +V) to an intermediate level before spelling rules are applied.

Example:
For the word "foxes", the lexicon represents it as:
fox + N + PL,
which is then processed by spelling rules to generate the surface form *foxes*.

21. How does a finite-state transducer differ from a finite-state automaton? **K2**

Finite-State Automaton (FSA)	Finite-State Transducer (FST)
A finite-state automaton is used to recognize or accept strings.	A finite-state transducer is used to transform one string into another.
It processes only input symbols.	It processes input symbols and produces output symbols.
It works with a single input tape.	It works with two tapes: input tape and output tape.
It only decides whether a string is valid or invalid.	It maps between different levels of representation.
It is mainly used for pattern matching and string recognition.	It is mainly used for morphological analysis and spelling rules.
It does not generate any output string.	It generates an output string for each valid input.
Example: It checks whether *baa!* belongs to a language.	Example: It converts **fox + N + PL** into *foxes*.

Tokenization

22. Define tokenization. **K1**

- Tokenization is the process of breaking a stream of text into smaller units called tokens, such as words, punctuation marks, or numbers, which serve as the basic units for further language processing.

- Tokenization is essential because most NLP algorithms operate on tokens rather than raw text.
- For example, the sentence *"NLP is interesting."* is tokenized into *NLP / is / interesting / .*

23. What is text normalization? **K1**

Text normalization refers to the process of converting text into a standardized and uniform format before further processing. Normalization includes operations such as converting text to lowercase, expanding abbreviations, handling spelling variations, and standardizing dates and numbers. This step reduces linguistic variability and improves the accuracy of NLP systems.

24. Why must dates and numbers be treated as single tokens? **K2**

- Dates and numbers must be treated as single tokens because splitting them into smaller components can distort their actual meaning.
- The module explains that expressions such as *"12/08/2024"* or *"₹10,000"* represent single semantic units.
- If tokenized incorrectly, the system may misinterpret them, leading to errors in tasks such as information extraction or question answering.

25. How does subword tokenization reduce the OOV problem? **K2**

- The Out-of-Vocabulary (OOV) problem occurs when a word is not present in the training vocabulary.
- Subword tokenization reduces this problem by splitting unknown or rare words into smaller meaningful subword units.
- This allows the model to process new words by combining known subwords, thereby improving vocabulary coverage and overall system performance.

Detecting and Correcting Spelling Errors

26. Define spelling error detection in NLP. **K1**

- Spelling error detection in NLP is the task of identifying incorrect spellings in a given text.
- It is performed during lexical analysis by checking each word against a dictionary, lexicon, or language model.
- The system flags a word as an error if it does not match any valid word form stored in the lexicon. This step is essential before tasks like parsing, semantic analysis, or machine translation.

Example:
Sentence: *He recieved the letter yesterday.*
Here, **"recieved"** is not present in the dictionary.
Hence, the NLP system detects it as a **spelling error** and suggests **"received"**.

27. List any two causes of spelling errors. **K2**

Spelling errors commonly occur due to the following reasons:
a) Typing or Keyboard Errors

These occur when characters are inserted, deleted, substituted, or transposed while typing.
Examples:

- Insertion: *speell* → spell
- Deletion: *recive* → receive
- Substitution: *tezt* → text
- Transposition: *form* → from

Such errors are very common in fast typing and informal text.
b) Lack of Language Knowledge / Carelessness
Writers may not know the correct spelling or may write carelessly, especially in social media or informal communication.
Examples:

- *definately* instead of *definitely*
- *enviroment* instead of *environment*

These errors occur even though the writer intended the correct word.

28. How do non-word errors differ from real-word errors? **K2**

Spelling errors in NLP are mainly classified into non-word errors and real-word errors.
Non-word Errors

- A non-word error results in a string that is not a valid word in the language.
- These errors are easy to detect because the word does not exist in the dictionary.

Example:
Sentence: *She recieved the gift.*
Here, "recieved" is not a valid English word → Non-word error
Real-word Errors

- A real-word error occurs when a word is spelled correctly but is used incorrectly in the given context.
- These errors are harder to detect because the word exists in the dictionary.

Example:
Sentence: *She went to there house.*
Here, "there" is a valid word, but the correct word should be "their". This requires context analysis to detect.

Minimum Edit Distance

29. List the three basic edit operations used in MED. K1

The three basic edit operations used in Minimum Edit Distance are:

1. Insertion – inserting a character into a string
2. Deletion – deleting a character from a string
3. Substitution – replacing one character with another

These operations are used to convert one string into another with minimum cost.
Example: Convert "cat" → "cut"

- Substitute a → u
  Total edit distance = 1

30. Why is dynamic programming used to compute Minimum Edit Distance? K2

i) Dynamic programming is used to compute Minimum Edit Distance because:
- The problem involves overlapping subproblems
- The optimal solution can be built from optimal solutions of smaller substrings
- It avoids repeated computation, making the algorithm efficient

ii) To compute the MED between two strings, all possible edit paths must be considered.

iii) Dynamic programming ensures that:
- Each subproblem is solved once
- The final distance is obtained in polynomial time

iv) Example:

To convert "intention" → "execution", dynamic programming computes edit distances for smaller prefixes and reuses them to find the final minimum cost.

PART - B (16 Marks)

3.(i) Apply grammar-based language modeling to parse a sentence using CFG rules. 8 K3

Grammar-based language models rely on explicit grammar rules to describe how sentences in a language are structured. These rules specify how smaller components like words and phrases combine to form larger meaningful units such as noun phrases, verb phrases, and complete sentences.

In this approach, Context-Free Grammar (CFG) is commonly used. CFG consists of:

- Non-terminal symbols (like S, NP, VP, PP) which represent grammatical structures
- Terminal symbols (actual words like “the”, “cat”, “chased”)
- Production rules that define how symbols can be expanded

Given CFG Rules:

- S → NP VP
- NP → Det N
- VP → V NP | V NP PP
- PP → P NP
- Det → “the” | “a”
- N → “cat” | “dog” | “ball”
- V → “chased” | “ate”
- P → “on” | “under” | “with”

Step-by-Step Parsing (Derivation)

Start with the start symbol:

S
NP VP
Det N VP
“the” N VP
“the” cat VP
“the” cat V NP
“the” cat chased NP
“the” cat chased Det N
“the” cat chased the dog
Add PP using VP → V NP PP
PP → P NP
“on” Det N

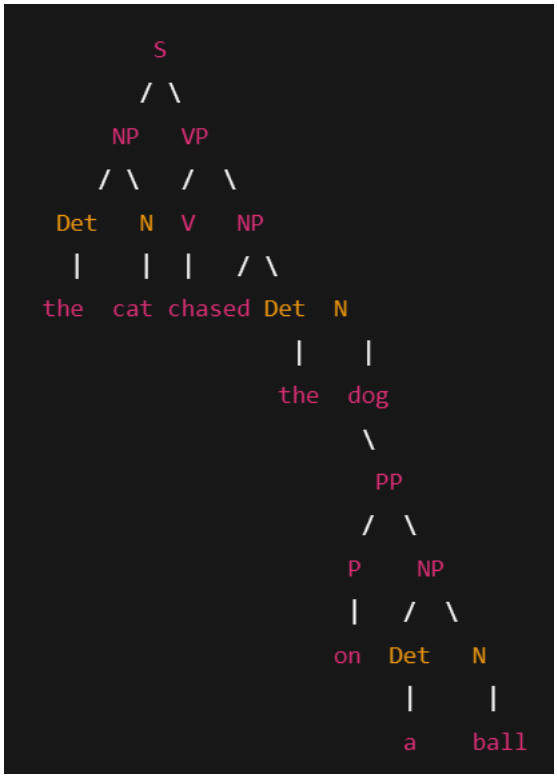
Parsing Process

1. The process begins with the **start symbol S**, representing a sentence.
2. Using the rule **S → NP VP**, the sentence is broken into a noun phrase and a verb phrase.
3. The noun phrase **NP** is expanded using **NP → Det N**, giving a determiner and noun.
4. The determiner and noun are replaced with actual words such as **“the”** and **“cat”**.
5. The verb phrase **VP** is expanded into a verb and another noun phrase using **VP → V NP**.

6. The second noun phrase is again expanded into determiner and noun, forming **"the dog"**.
7. Optionally, a prepositional phrase is added using **VP → V NP PP**, where **PP → P NP**, forming **"on a ball"**.

Final Parsed Sentence: "the cat chased the dog on a ball"

Parse Tree



Explanation

- CFG rules define how words combine into phrases
- The parser starts from S (sentence) and applies production rules
- Each step expands non-terminals until only terminal words remain
- This results in a grammatically valid sentence
- Grammar-based language modeling using CFG applies formal grammatical rules to parse sentences by breaking them into structured components.
- This shows how grammar-based language modeling using CFG systematically parses or generates sentences by following formal grammar rules, ensuring syntactic correctness.
- This approach provides clarity, interpretability, and linguistic correctness, making it useful in early NLP systems and syntax analysis.

3.(ii) Apply a bigram language model to calculate the probability of the sentence: "I like to eat apples" using the provided corpus. **8** **K3**

Given corpus:

1. "I like to eat apples"
2. "Apples are delicious"
3. "I like to eat bananas"

We can use this corpus to build a bigram language model, which estimates the probability of each word given its preceding word. Here's how we can do it:

1. **Tokenization**: First, we tokenize the sentences into individual words, removing punctuation and converting everything to lowercase. This gives us the following tokenized corpus:

   ["i", "like", "to", "eat", "apples"]
   ["apples", "are", "delicious"]
   ["i", "like", "to", "eat", "bananas"]

2. **Counting Bigrams**: Next, we count the occurrences of bigrams (pairs of consecutive words) in the tokenized corpus:
   ("i", "like"): 2
   ("like", "to"): 2
   ("to", "eat"): 2
   ("eat", "apples"): 1
   ("apples", "are"): 1
   ("are", "delicious"): 1
   ("eat", "bananas"): 1

3. **Estimating Probabilities**: We calculate the probability of each word given its preceding word using maximum likelihood estimation (MLE)
   P("like" | "i") = Count("i like") / Count("i") = 2 / 2 = 1.0
   P("to" | "like") = Count("like to") / Count("like") = 2 / 2 = 1.0
   P("eat" | "to") = Count("to eat") / Count("to") = 2 / 2 = 1.0
   P("apples" | "eat") = Count("eat apples") / Count("eat") = 1 / 2 = 0.5
   P("are" | "apples") = Count("apples are") / Count("apples") = 1 / 1 = 1.0
   P("delicious" | "are") = Count("are delicious") / Count("are") = 1 / 1 = 1.0
   P("bananas" | "eat") = Count("eat bananas") / Count("eat") = 1 / 2 = 0.5

Now, we have a bigram language model that can estimate the probability of word sequences. For example, if we want to compute the probability of the sentence "I like to eat bananas," we can multiply the probabilities of the bigrams:

P("i") * P("like" | "i") * P("to" | "like") * P("eat" | "to") * P("bananas" | "eat")
= 1.0 * 1.0 * 1.0 * 1.0 * 0.5
= 0.5

This shows that according to our bigram model, the probability of the sentence "I like to eat bananas" is 0.5.

6.(i) Evaluate how finite-state automata and regular expressions correspond to each other in language representation. **8 K4**

Finite-State Automata (FSA) and Regular Expressions (RE) are two formal methods used to define and represent regular languages. Though they use different

notations and approaches, they are equivalent in expressive power, meaning that any language described by a regular expression can be recognized by an FSA, and any language recognized by an FSA can be represented by a regular expression.

From Regular Expressions to FSA

A regular expression defines patterns using operations such as:

- Concatenation
- Union (|)
- Kleene star (*)

For every regular expression, it is possible to systematically construct a finite-state automaton that accepts exactly the same set of strings. This is typically done using techniques like:

- Thompson's construction
- Subset construction (for converting NFA to DFA)

Thus, a pattern like: /baa+!/ can be represented as an FSA with states and transitions that recognize strings such as:

- baa!
- baaa!
- baaaa!

From FSA to Regular Expressions

Similarly, any finite-state automaton can be converted back into an equivalent regular expression by eliminating states and forming expressions that describe the paths between states. This proves that FSAs and regular expressions define the same class of languages.

Example showing Correspondence between Regular Expression and FSA

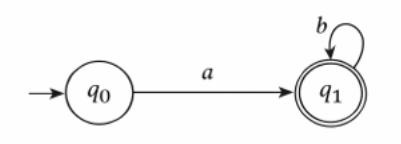
Consider the regular expression: /ab*/

This represents:

- The letter a followed by
- b repeated zero or more times

Accepted strings are: a, ab, abb, abbb,…

Equivalent Finite-State Automaton (FSA)



Explanation:

• Start at q0
• On reading a, move to q1
• At q1, loop on b (accept any number of b's)
• q1 is the final (accepting) state

This FSA accepts exactly the same strings defined by the regular expression **/ab*/**

Regular expressions are compact and easy to write for pattern specification. Finite-state automata are efficient for actual implementation and recognition. Both describe regular languages. Regular expressions provide a declarative way to describe patterns, while finite-state automata provide an operational mechanism to recognize those patterns efficiently.

FSAs are often used in NLP tools such as lexical analyzers and morphological processors. REs are mainly used for specifying token patterns. This equivalence forms the foundation of many NLP systems, especially in lexical analysis and pattern matching.

6.(ii) Analyze the different types of spelling error detection and correction problems in NLP and explain how each requires different handling techniques. **8 K4**

Spelling Error Detection and Correction in NLP Systems

The detection and correction of spelling errors play a vital role in modern NLP applications such as word processors, search engines, OCR systems, and handwriting recognition. NLP systems must identify incorrect spellings and suggest appropriate corrections to ensure text accuracy and readability.

Spelling error problems in NLP can be classified into three major types, each requiring different handling techniques:

1. Non-Word Error Detection: Non-word errors occur when a misspelled word does not exist in the dictionary.

Example: graffe → giraffe

Handling Technique:

- Each word in the text is checked against a dictionary
- If the word is absent, it is marked as an error
- Modern systems use large dictionaries for higher accuracy

Technology Used:

- Finite-State Transducers (FSTs) act as word recognizers
- FSTs can be converted into Finite-State Automata (FSAs) for efficient lookup
- They also support productive morphology (e.g., plural forms, verb inflections)

Advantage:

- Efficient handling of large vocabularies
- Automatically recognizes new inflected forms

2. Isolated-Word Error Correction: Correcting misspelled words without considering sentence context.

Example: graffe → giraffe

Handling Technique:

1. Generate possible candidate corrections
2. Measure similarity between misspelled word and candidates
3. Choose the closest word

Algorithm Used:

Minimum Edit Distance Algorithm: It calculates the number of Insertions, Deletions and Substitutions. They are needed to transform one word into another.

Advantage:

- Simple and effective for common spelling mistakes
- Finds most likely correction based on closeness

3. Context-Dependent Error Detection and Correction (Real-Word Errors):

Errors that result in valid dictionary words but incorrect in context.

Examples:

- there → three
- dessert → desert
- peace → piece

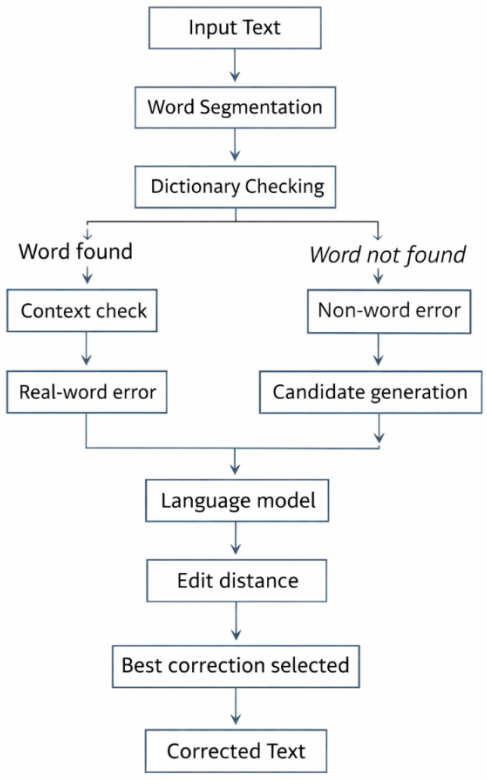
Handling Technique:

- Uses surrounding words (context)
- Applies statistical language models
- Computes probability of word sequences
- Selects the word that best fits the context

Advantage:

- Can detect errors that dictionary lookup cannot

Flow Diagram for Spell Correction in NLP



Spelling error detection and correction in NLP involves multiple layers of processing. Non-word errors are easily detected using dictionary-based methods

supported by finite-state technologies. Isolated-word correction relies on similarity measures such as minimum edit distance. Real-word errors require contextual understanding using statistical language models. Each type demands different techniques because of the nature of the errors involved. Combining these approaches leads to efficient and accurate spell-checking systems used in modern NLP applications.

7.(i) Construct a finite-state automaton (FSA) to recognize the regular expression /baa+!/ and explain how it accepts valid strings. **8 K3**

Construction of Finite-State Automaton for the Regular Expression /baa+!/

The regular expression: /baa+!/ represents strings that: start with b, followed by a, followed by one or more a's (a+), end with !

Valid strings include:

- baa!
- baaa!
- baaaa!

Step 1: Identify pattern structure

b → a → (one or more a's) → !

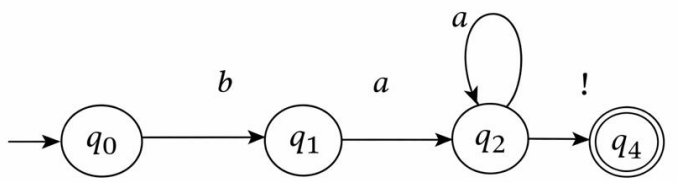
Step 2: Define states

Let:

- q0 – Start state
- q1 – After reading b
- q2 – After reading first a and Loop state for repeated a
- q4 – Final (accepting) state after !

Step 3:

Transition Diagram



Transition Table

Current State	Next State		
Input	**a**	**b**	!
→q0	ϕ	q1	ϕ
q1	q2	ϕ	ϕ
q2	q2	ϕ	q4
*q4	ϕ	ϕ	ϕ

Step 4: Explanation of Acceptance

Let's check the string: baaa!

1. q0 → read b → go to q1 δ(q0, b) = q1
2. q1 → read a → go to q2 δ(q1, a) = q2
3. q2 → read a → stay in q2 (loop) δ(q2, a) = q2
4. q2 → read ! → go to q4 δ(q2, !) = q4

Since q4 is a final state, the string is accepted.

The finite-state automaton constructed above recognizes all strings that match the regular expression /baa+!/. The looping transition ensures one or more occurrences of the character a, while the final transition to the accepting state ensures the string ends with an exclamation mark. This demonstrates how regular expressions can be converted into finite-state automata for efficient pattern recognition in NLP systems such as lexical analyzers.

7.(ii) Apply the Normalization techniques for the given text and comment on the vocabulary before and after the normalization: 8 K3
Raj and Vijay are best friends. They play together with other friends. Raj likes to play football but Vijay prefers to play online games. Raj wants to be a footballer. Vijay wants to become an online gamer.

Normalization Techniques Applied

Given Text:

Raj and Vijay are best friends. They play together with other friends. Raj likes to play football but Vijay prefers to play online games. Raj wants to be a footballer. Vijay wants to become an online gamer.

Normalization Techniques Applied

1. Tokenization

Breaking the text into individual words/tokens.

Tokens:

["Raj", "and", "Vijay", "are", "best", "friends", "They", "play", "together", "with", "other", "friends", "Raj", "likes", "to", "play", "football", "but", "Vijay", "prefers", "to", "play", "online", "games", "Raj", "wants", "to", "be", "a", "footballer", "Vijay", "wants", "to", "become", "an", "online", "gamer"]

2. Lowercasing

Convert all tokens to lowercase for uniformity.

["raj", "and", "vijay", "are", "best", "friends", "they", "play", "together", "with", "other", "friends", "raj", "likes", "to", "play", "football", "but", "vijay", "prefers", "to", "play", "online", "games", "raj", "wants", "to", "be", "a", "footballer", "vijay", "wants", "to", "become", "an", "online", "gamer"]

3. Stopword Removal

Remove common words like *and, are, to, with, the, a, an, but*.

Remaining tokens:

["raj", "vijay", "best", "friends", "play", "together", "other", "friends", "raj", "likes",

"play", "football", "vijay", "prefers", "play", "online", "games", "raj", "wants", "footballer", "vijay", "wants", "become", "online", "gamer"]

4. Stemming / Lemmatization

Reduce words to their root form.

- *friends → friend*
- *likes → like*
- *prefers → prefer*
- *games → game*
- *footballer → football*
- *gamer → game*

Normalized tokens:
["raj", "vijay", "best", "friend", "play", "together", "other", "friend", "raj", "like", "play", "football", "vijay", "prefer", "play", "online", "game", "raj", "want", "football", "vijay", "want", "become", "online", "game"]

Vocabulary Comparison

Stage	Vocabulary Size	Examples
Before Normalization	22 unique words	raj, vijay, friends, play, football, prefers, footballer, gamer
After Normalization	14 unique words	raj, vijay, friend, play, football, prefer, game, want, become, online

- Before normalization: Vocabulary is larger, includes inflected forms (*friends, likes, prefers, games, footballer, gamer*).
- After normalization: Vocabulary is reduced and simplified. Redundant variations are collapsed into root forms (*friend, like, prefer, game*).
- This reduction improves efficiency in NLP tasks (like text classification or retrieval) because the model deals with fewer unique tokens.
- However, some semantic richness (e.g., distinction between *footballer* vs *football*) may be lost depending on the lemmatization strategy.

8. Analyze the working of the Edit Minimum Distance algorithm and explain why dynamic programming is suitable for solving it. **16 K4**

Dynamic Programming in Minimum Edit Distance

- The Minimum Edit Distance (MED) algorithm is a fundamental technique in Natural Language Processing used to measure the similarity between two strings by calculating the minimum number of edit operations required to transform one string into another.

- The Minimum Edit Distance (MED) is computed using dynamic programming, introduced by Bellman (1957).
- Dynamic programming works by:
  - Breaking a large problem into smaller subproblems
  - Solving each subproblem once
  - Storing results in a table (matrix) for efficient reuse
- This method is widely applied in speech and language processing.

Edit Operations

The MED algorithm allows three basic operations:

1. Insertion – adding a character
2. Deletion – removing a character
3. Substitution – replacing one character with another

Each operation usually has a unit cost.

Principle of Optimal Substructure

- If a string like **exention** lies on the optimal path from *intention* to *execution*, then:
  - The path from *intention* to *exention* must also be optimal
- If a shorter path existed, it would reduce the total distance
- This would contradict the optimal solution
- This property forms the basis of dynamic programming

Edit-Distance Matrix Construction

- A **distance matrix** is created where:
  - Rows represent symbols of the source string
  - Columns represent symbols of the target string
- Each cell edit-distance[i, j] stores:
  - The minimum distance between the first *i* characters of target
  - And the first *j* characters of source

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(source[i]) \\ D[i,j-1] + \text{ins-cost}(target[j]) \\ D[i-1,j-1] + \text{sub-cost}(source[i], target[j]) \end{cases}$$

- Each cell value is computed from:
  - Insertion
  - Deletion
  - Substitution (from neighboring cells)

$$D[i,j] = \min \begin{cases} D[i-1,j] + 1 \\ D[i,j-1] + 1 \\ D[i-1,j-1] + \begin{cases} 2; & \text{if } source[i] \neq target[j] \\ 0; & \text{if } source[i] = target[j] \end{cases} \end{cases}$$

- The matrix is filled step by step starting from the initial cell

```
function MIN-EDIT-DISTANCE(source, target) returns min-distance

  n ← LENGTH(source)
  m ← LENGTH(target)
  Create a distance matrix D[n+1,m+1]

  # Initialization: the zeroth row and column is the distance from the empty string
  D[0,0] = 0
  for each row i from 1 to n do
    D[i,0] ← D[i-1,0] + del-cost(source[i])
  for each column j from 1 to m do
    D[0,j] ← D[0,j-1] + ins-cost(target[j])

  # Recurrence relation:
  for each row i from 1 to n do
      for each column j from 1 to m do
        D[i,j] ← MIN( D[i−1,j] + del-cost(source[i]),
                      D[i−1,j−1] + sub-cost(source[i],target[j]),
                      D[i,j−1] + ins-cost(target[j]))
  # Termination
  return D[n,m]
```

Figure 2.17 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g., $\forall x$, ins-cost$(x) = 1$) or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e., sub-cost$(x,x) = 0$).

Example Transformation: *intention* → *execution*

The transformation is performed through a sequence of small edits:

1. Delete **i** → ntention
2. Substitute **n** → **e** → etention
3. Substitute **t** → **x** → exention
4. Insert **u** → exenution
5. Substitute **n** → **c** → execution

Each step represents a smaller subproblem, and combining these steps results in the complete transformation. This demonstrates how dynamic programming efficiently breaks down complex problems.

Src\Tar	#	e	x	e	c	u	t	i	o	n
#	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	6	7	8
n	2	3	4	5	6	7	8	7	8	7
t	3	4	5	6	7	8	7	8	9	8
e	4	3	4	5	6	7	8	9	10	9
n	5	4	5	6	7	8	9	10	11	10
t	6	5	6	7	8	9	8	9	10	11
i	7	6	7	8	9	10	9	8	9	10
o	8	7	8	9	10	11	10	9	8	9
n	9	8	9	10	11	12	11	10	9	8

Figure 2.18 Computation of minimum edit distance between *intention* and *execution* with the algorithm of Fig. 2.17, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions.

Explanation

This table shows the dynamic programming matrix used to compute the Minimum Edit Distance between the words:

- ✓ Source (Src): intention
- ✓ Target (Tar): execution

using Levenshtein distance, where:

- Insertion cost = 1
- Deletion cost = 1
- Substitution cost = 2

1. Structure of the Table

➤ Rows (left side)

Represent characters of the source word:

#, i, n, t, e, n, t, i, o, n

(# represents empty string)

➤ Columns (top)

Represent characters of the target word:

#, e, x, e, c, u, t, i, o, n

2. Meaning of Each Cell

Each cell: edit-distance[i][j] represents the minimum cost to convert:

First i letters of intention into First j letters of execution

Example:

Cell at (i, e):

Shows cost to convert: i → e

That value = 2 (because substitution cost = 2)

3. First Row and First Column

These show converting an empty string:

First row:

0 1 2 3 4 5 6 7 8 9

Means:

- "" → "e" = 1 insertion
- "" → "ex" = 2 insertions
- "" → "execution" = 9 insertions

First column:

0
1
2
3
...
9

Means:

- "i" → "" = 1 deletion
- "in" → "" = 2 deletions
- "intention" → "" = 9 deletions

4. Computation of each Cell

Each value is the minimum of:

- Deletion:
  From top cell + 1
- Insertion:
  From left cell + 1
- Substitution:
  From diagonal cell + cost
  Where:
    - cost = 0 if characters match
    - cost = 2 if they differ

Example:

Look at the cell where:
Row = t , Column = t
Since both are same: Substitution cost = 0
So diagonal value is taken (lower cost)

5. Final Result (Bottom-right cell)

The last cell value: 8
i.e: Minimum edit distance between intention and execution = 8 (using given costs)

Producing String Alignment

- Alignment is viewed as a path through the edit-distance matrix
- Movements represent:
  - Horizontal move → insertion
  - Vertical move → deletion
  - Diagonal move → substitution or match

Backpointer Method for Alignment

- Step 1:
  - Store **backpointers** in each cell indicating where minimum cost came from
- Step 2:
  - Perform **backtrace** from final cell to initial cell
  - Follow pointers to get the minimum-cost alignment

Dark Grey Path (Optimal Alignment)

The dark grey shaded cells represent: One possible minimum-cost edit path
This path is found by:

1. Starting from the final cell (top-right corner)

2. Following arrows backward
3. Moving until reaching the start cell

This process is called backtracing.

n	9	↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙←↓ 12	↓ 11	↓ 10	↓ 9	↙ **8**
o	8	↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↓ 10	↓ 9	↙ **8**	← 9
i	7	↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9	↙ **8**	← 9	← 10
t	6	↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙ **8**	← 9	← 10	←↓ 11
n	5	↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ **8**	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙↓ 10
e	4	↙ 3	← 4	↙← **5**	← **6**	← 7	←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9
t	3	↙←↓ 4	↙←↓ **5**	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙ 7	←↓ 8	↙←↓ 9	↓ 8
n	2	↙←↓ **3**	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↓ 7	↙←↓ 8	↙ 7
i	**1**	↙←↓ 2	↙←↓ 3	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙ 6	← 7	← 8
#	**0**	1	2	3	4	5	6	7	8	9
	#	**e**	**x**	**e**	**c**	**u**	**t**	**i**	**o**	**n**

Figure 3.27 When entering a value in each cell, we mark which of the 3 neighboring cells we came from with up to three arrows. After the table is full we compute an **alignment** (minimum edit path) via a **backtrace**, starting at the **8** in the upper right corner and following the arrows. The sequence of dark grey cell represents one possible minimum cost alignment between the two strings.

The Minimum Edit Distance algorithm efficiently computes string similarity using dynamic programming. By breaking the problem into smaller subproblems and storing results in a matrix, it achieves optimal performance. The principles of optimal substructure and overlapping subproblems make dynamic programming the ideal solution approach.

Applications of Minimum Edit Distance

- Spelling correction
- String alignment
- Speech recognition (Word Error Rate calculation)
- Machine translation (sentence matching in parallel corpora)