Test 05
Submitted By – Srijana Raut(101134199)

1. Explain Boolean query processing. Let us take an "INTERSECT" operation in a query: Brutus AND Caesar for the following postings lists.
Brutus: −→ 2 → 4 → 8 → 16 → 32 → 64 → 128
Caesar: −→ 1 → 2 → 3 → 5 → 8 → 13 → 22 → 33

Boolean query processing refers to the method of retrieving information from a database or a collection of data using Boolean operators. Boolean operators are logical connectors that allow you to combine or exclude keywords in a search to get more specific or broader results. The three basic Boolean operators are:

1. **AND:** This operator is used to retrieve records that contain both terms. For example, if you search for "cats AND dogs," you would get results that include both "cats" and "dogs."
2. **OR:** This operator is used to retrieve records that contain at least one of the terms. For example, if you search for "cats OR dogs," you would get results that include either "cats" or "dogs" or both.
3. **NOT:** This operator is used to exclude records that contain a specific term. For example, if you search for "cats NOT dogs," you would get results that include "cats" but exclude any results that also mention "dogs."

The Intersect of the given operation is given below.

Brutus: −→ 2 → 4 → 8 → 16 → 32 → 64 → 128
Caesar: −→ 1 → 2 → 3 → 5 → 8 → 13 → 22 → 33

Output is 2->8

Thus, sometimes intersecting posting length is known as merging posting length. The complexity of the above question is O(n+m) where n and m are the respective length.

2. Does the sorting (the postings lists) matter? Explain.
Postings list is a data structure that records occurrences of a term in a document, often including position information. Each entry in the list, known as a posting, corresponds to a specific document. The combined set of postings lists for all terms is referred to as the postings, forming a crucial component in information retrieval systems.

Posting list (Sorting) is popular for query processing optimization. Besides, posting length has the following importance.

- **Effective Retrieval:** Retrieval efficiency can be greatly increased by sorting the postings lists. Sorting postings lists makes it easier to find and combine pertinent data when conducting searches.

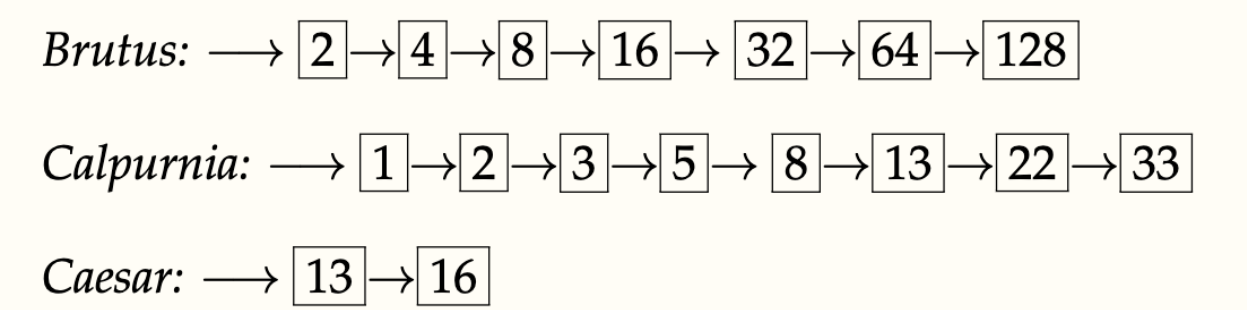
- **Intersection Operations:** During query processing, sorting makes intersection operations more efficient. This is important since it speeds up the intersection of sorted postings lists, which is useful for discovering papers that include multiple query phrases.

- **Compression Techniques:** By applying compression techniques, sorting can lower the amount of storage space needed for posting lists. This is especially crucial for extensive collections of documents.

- **Enhanced Processing Speed:** Sorting makes it possible to process posting lists using more efficient methods, which enhances the overall speed of information retrieval systems' searches and retrievals.

3. Can the size of the postings lists vary from one to another? How can such a query (mentioned above) be optimized?

Yes, the size of postings lists can vary significantly from one term to another. Let us Consider a query that is an AND t terms where for each t terms, retrieve its postings, and then AND them together.



Here the posting length matters according to the following.

$$length_{Caesar} < length_{Brutus} < length_{Calpurnia} \text{ i.e., } 2 < 7 < 8$$

There might be different order for query processing which is also known as query optimization. The query order for Brutus, Calpurnia and Caesar can be.

1. Brutus AND (Calpurnia AND Caesar)
2. (Brutus AND Calpurnia) AND Caesar
3. (Brutus AND Caesar) AND Calpurnia

Among all these three options. The third query is fastest which is consider as optimized one.

4. What are the issues in document parsing? Explain.
The issues in document parsing are categorize into two different points. They are.

i. Obtaining a character sequence in a document – This is simple what format to be used for parsing a document like in the form of pdf, word, excel, html etc. Also, what are the language is it in (English, German, French, Nepali)
ii. Choosing a document unit – It simply refers to the different character set in a use.

Explanation regarding how we can process is explained below.

i. Character sequence decoding — The first step of processing is to convert this byte sequence into a linear sequence of characters. For the case of plain English text in ASCII encoding, which is trivial. However, using encoding schemes, such as Unicode UTF-8, or other problem specific standards or tools would be questionable. The characters may have to be decoded out of some binary representation like Microsoft Word DOC files and/or a compressed format such as zip files. Also, we must determine the document format, and then an appropriate decoder has to be used. Even for plain text documents, additional decoding may need to be done.

ii. Choosing a document unit - After document delineation and character sequence decoding, the next phase is to see what the document unit will be used for indexing.
For example of a file (in the particular folder) as a document. But there are many cases:
- A traditional Unix (mbox-format) email file stores a sequence of email messages (an email folder) in one file.
- Email messages with attachments: we want to regard the email message, and each contained attachment as separate documents.
- Email zipped attachments: you might want to decode the zip file and regard each file it contains as a separate document.
- Latex files and PPT files

In general, in these cases, we need to consider about indexing granularity.

5. Explain tokenization process in detail. Give some examples that are related to English, not other languages.

Given a character sequence and a defined document unit, tokenization is the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation.

Fig. Tokenization process.

In tokenization process, following are the few terminologies, that we need to understand. They are.
- **Token:** A token is a specific instance of a sequence of characters within a document. Tokens are considered as cohesive units during processing and analysis.

- **Type:** A type is a class that encompasses all tokens sharing the same character sequence. In other words, it represents the unique elements or distinct forms in the document.

- **Term:** A term is a type, possibly after normalization processes. Terms are entries in the information retrieval system's dictionary and are derived from tokens. Normalization might involve procedures like stemming or removing stop words.

For example, we have document "to sleep perchance to dream" which need to be indexed. Then it has 5 tokens: 'to', 'sleep', 'perchance', 'to', 'dream'. However, there are only 4 types because 'to' repeats. If you exclude 'to' (considering it as a stop word), you have 3 terms: 'sleep', 'perchance', and 'dream'.

This breakdown of sentence into terms, token helps in creating a more refined and manageable representation of the document for information retrieval and processing.

6. Project: Write a code (preferably in Python) to Lemmatize any text inputs. It needs to be complete by itself (while submitting your code).
Hint: Use Wordnet Lemmatizer with NLTK as Wordnet is the standard and considered as a publicly available lexical database for the English. Your input should be all least a paragraph, not just a single word.

```
# parsing command-line arguments
import argparse
# Natural Language Toolkit
import nltk
#  wordnet lexical database for the English language
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize, sent_tokenize

class Lemmatizer:
    def __init__(self):
        """
        Initialize the Lemmatizer class.
        Downloads WordNet data using NLTK.
        """
        nltk.download('wordnet')

    def lemmatize_text(self, text):
        """
        Lemmatize a given text by tokenizing it into sentences and words,
        lemmatizing each word using the WordNet Lemmatizer, and joining
        the lemmatized words back into sentences and the sentences back
        into a paragraph.

        Parameters:
        - text (str): The input text to be lemmatized.

        Returns:
        - str: The lemmatized text.
```

```
        """
        # Tokenize the text into sentences
        sentences = sent_tokenize(text)

        # Initialize the WordNet Lemmatizer
        lemmatizer = WordNetLemmatizer()

        # Lemmatize each word in each sentence
        lemmatized_sentences = []
        for sentence in sentences:
            # Tokenize the sentence into words
            words = word_tokenize(sentence)

            # Lemmatize each word using the WordNet Lemmatizer
            lemmatized_words = [lemmatizer.lemmatize(word, self.get_wordnet_pos(word))
for word in words]

            # Join the lemmatized words back into a sentence
            lemmatized_sentence = ' '.join(lemmatized_words)

            # Add the lemmatized sentence to the list
            lemmatized_sentences.append(lemmatized_sentence)

        # Join the lemmatized sentences back into a paragraph
        lemmatized_text = ' '.join(lemmatized_sentences)

        return lemmatized_text

    def get_wordnet_pos(self, word):
        """
        Get the WordNet POS (Part of Speech) tag for a given word.

        Parameters:
        - word (str): The input word.

        Returns:
        - str: The WordNet POS tag.
        """
        tag = nltk.pos_tag([word])[0][1][0].upper()
        tag_dict = {"N": wordnet.NOUN, "V": wordnet.VERB, "R": wordnet.ADV, "J":
wordnet.ADJ}
        return tag_dict.get(tag, wordnet.NOUN)

def main():
    """
    Main function to handle command-line interface.
    Parses command-line arguments, creates a Lemmatizer instance,
    lemmatizes the input paragraph, and prints the original and
```

```
    lemmatized texts.
    """
    # Set up the command-line argument parser
    parser = argparse.ArgumentParser(description='Lemmatize a paragraph using NLTK.')
    parser.add_argument('paragraph', type=str, help='Input paragraph for
lemmatization')

    # Parse the command-line arguments
    args = parser.parse_args()

    # Create an instance of the Lemmatizer class
    lemmatizer_instance = Lemmatizer()

    # Lemmatize the input paragraph
    lemmatized_result = lemmatizer_instance.lemmatize_text(args.paragraph)

    # Print the original and lemmatized texts for comparison
    print("Original text:")
    print(args.paragraph)
    print("\nLemmatized text:")
    print(lemmatized_result)

if __name__ == "__main__":
    # Call the main function when the script is executed
    main()
```