COSC 320 – 001
Analysis of Algorithms
2022/2023 Winter Term 2

Project Topic Number: 1
Title of project: KeyScript

Group Lead: Brendan Michaud
Group Members: Shreya Vatsa, Shreyasi Chauhan, Brendan Michaud

Problem Formulation: Given a document D of length n and a list of abbreviations A of length m, the problem can be formulated as an algorithmic problem as follows:

Input:

Document D = d[1], d[2], ..., d[n], where each d[i] is a word in the document
List of abbreviations A = (a[1], b[1]), (a[2], b[2]), ..., (a[m], b[m]), where a[i] is the abbreviation and b[i] is the corresponding full form.

Output:

Document D' = d'[1], d'[2], ..., d'[n], where each d'[i] is the expanded form of the corresponding word d[i] in the original document D.

Naive Algorithm:

Initialize an empty list D' to store the expanded form of the document
For each word d[i] in the document D, do the following:
a. For each abbreviation a[j] in the list of abbreviations A, do the following:
i. If d[i] is equal to a[j], replace d[i] with b[j] and break the inner loop
b. Append d[i] to the list D'
Return D'

Mathematical Notation:
Let f(D, A) be the function that takes the document D and the list of abbreviations A as input, and returns the expanded form of the document D'. The algorithm can be represented mathematically as:

```
f(D, A) = D'
for i = 1 to n:
  for j = 1 to m:
    if d[i] = a[j]:
      d[i] = b[j]
      break
  D'.append(d[i])
return D'
```

Initial Pseudo-Code:

```
def algorithmA(inputFile):
    read input file to dataframe
    keywords = []
    keywords = df.columns
    buffer = [keywords.size]

    foreach(index in keywords):
```

```
        buffer[index] = searchForMatch(keywords[index])

    outputFile = replaceText(buffer)
    return outputFile

def searchForMatch(keyword):
    load in dataset
    if dataset contains keyword:
        store column.value at row[keyword]
    else
        value = keyword //abbreviation is not replaced

    return value

def replaceText(wordsToReplace):
    create new document
    foreach value in wordsToReplace:
        write value to document

    return document
```

Reviewed/Final pseudoCode – using the pandas dataframe *(naive approach)*:

```
import pandas as pd

abbreviations = {'abbrev': ['etc', 'i.e', 'e.g'], 'full_form': ['and so on', 'that is', 'for example']}
abbreviation_dict = pd.DataFrame(abbreviations)

input_file = 'input.txt'
df = pd.read_csv(input_file, header=None)

expanded_text = []
for text in df[0]:
   words = text.split()
   expanded_line = []
   for word in words:
      if word in abbreviation_dict['abbrev'].values:
         full_form = abbreviation_dict.loc[abbreviation_dict['abbrev'] == word, 'full_form'].values[0]
         expanded_line.append(full_form)
      else:
```

```
            expanded_line.append(word) #word is not replaced.
        expanded_text.append(' '.join(expanded_line))

df[1] = expanded_text

output_file = 'output.txt'
df.to_csv(output_file, header=False, index=False)
```

(*Note: semicolon delimiter is unnecessary in python syntax*).

Algorithm Analysis: Properties: The algorithm is a polynomial-time and linear-space algorithm.

Time complexity: This algorithm is O(n * m), where n is the length of the document and m is the length of the list of abbreviations. This is because for each word in the document, the algorithm needs to check all the abbreviations in the list. Since we are using a nested loop worst case becomes O(n^2) - with n being the number of words in the input file as the inner loop iterates through each word in the text and the outer loop iterates through each line of text. This is a time complexity generated for the naive approach using the panda package in python.

Space Complexity: The space complexity of this algorithm is O(n), where n is the length of the document, as the algorithm needs to store the expanded form of the document.

It's worth noting that the time and space complexities discussed here are for the core logic of the algorithm, and do not include the I/O operations of reading the input file and writing the output file. These operations will have their own time and space complexities, but they are dependent on the size of the file and the speed of the I/O operations and are not fully explored in the scope of this analysis.

Proof of Correctness for the naive algorithm: Show that the algorithm correctly replaces all instances of the abbreviations with the corresponding full forms for each document D in the input.

Initialization: The algorithm starts by creating a new set of documents D' to store the expanded documents.

Loop Invariant: At the start of each iteration of the outer loop over the documents d in D, the algorithm has correctly expanded all the previous documents and stored them in D'.

Maintenance: During each iteration of the inner loop over the abbreviations (b, p) in A, the algorithm replaces all instances of the brief term b in the current document d using the replace method ensuring that all instances of the abbreviations in the document are correctly expanded.

Termination: After all the documents have been processed and all instances of the abbreviations have been replaced, the expanded documents are stored in D'. The loop terminates and the algorithm returns D'.

Therefore, correctly replaces all instances of the abbreviations with the appropriate full forms for each document and is correct.

Unexpected Cases/Difficulties: This is a naive approach, so reducing time complexity is priority. Also, because this algorithm reads each word line by line and not through a constant lookup method. Which is what is causing the polynomial time complexity and also, we can try to reduce space complexity, but reducing time complexity is more important. The time complexity will be apparent in behavior given the size of the input file and can be more problematic for larger input size. We should also handle errors in case of an unrecognizable/ restricting input format.

Task Separation and responsibilities:

Problem Formulation - Brendan, Shreya, Shreyasi
Pseudocode: Brendan, Shreya
Algorithm analysis: Brendan, Shreya
Unexpected Cases and Difficulties: Brendan, Shreyasi, Shreya

Questions: Should some edge cases be features? Or be handled in one main method. Probably separate methods.

Edge cases - storing the known database/ format of text/ language of text / length of text/ updating abbreviation lists, etc.

Search algorithms used are to be considered.
Would sorting required - depending on the search algorithm.
We use the join function in python to perform concatenation and appending strings (works similarly to String Builder in java).