COSC 320 – 001

Analysis of Algorithms

2022/2023 Winter Term 2

Project Topic Number: 1

Title of project: KeyScript

(Second Milestone)

Group Lead: Brendan Michaud

Group Members: Shreya Vatsa, Shreyasi Chauhan, Brendan Michaud

Abstract: In Milestone 2, we are building off of our previous work to create a more efficient algorithm based on space and time.
Second algorithm: We decided to use a hash table (also known as a python set) for our second algorithm. Using a hash table makes sense for this problem because it allows for efficient lookups of whether a given word is an abbreviation or not. When we store the abbreviations in a hash table, we can use the in operator to check if a given word is in the set of abbreviations in constant time, O(1). This is because the hash table uses a hash function to map each abbreviation to an index in the underlying array, which allows for very fast lookups.
Other data structures, such as lists or tuples, would require iterating over each element to check if a given word is an abbreviation or not, which would have a worst-case time complexity of O(n) where n is the number of elements in the data structure. This could be much slower for large lists of abbreviations. A hash table provides a good balance of efficiency and memory usage for this problem, making it a good choice for storing the set of abbreviations.

Problem Formulation: Given a document D of length n and a HashMap of abbreviations A with k entries (k corresponds to the key value pairs), the problem can be formulated as follows:

Input:

Document D = d[1], d[2], ..., d[n], where each d[i] is a word in the document
HashMap 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. A(a[i], b[i]) is a key value pair where a[i] is the key and b[i] is the value.

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. Hence each d'[i] is the value corresponding to the key (k) of d[i] in A, where k = hash(d[i]): the key value pair as described before.

Mathematical Notation:

Initialize the abbreviation HashMap H with k entries representing key-value pairs where each key is an abbreviation and each value is its corresponding full form.

H = {abbreviation: full_form for abbreviation, full_form in H }

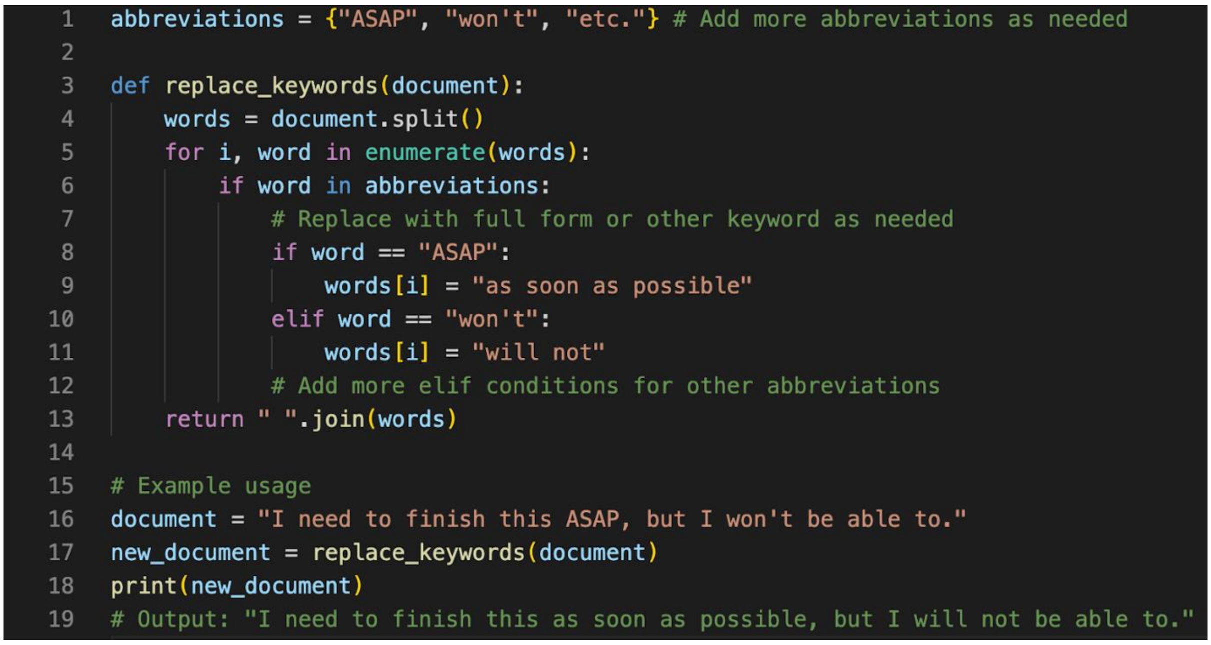
```
For each word w in D, do the following:
for i in range(len(D)):
   w = D[i]
   if w in H:
        If w is an abbrev in H, replace w with its corresponding full form in H.
       D[i] = H[w]
   else:
       leave w unchanged.
       pass
```

Return the output document D', with all abbreviations in D replaced with their corresponding full forms in H.

return D

Algorithm B:

1. Initialize an empty HashMap, H, that stores key-value pairs H(a[i],b[i])
2. For each key-value pair in A from 0 to m, insert it into H(k), where k is the calculated index from the hash function (x).
3. For each word d[i] in input document D, calculate the hash value k using hash function x. If a match is found at H(k), replace d[i] with H(b[k]) and append to the new document D’. If no match is found d’[i] = d[i].
4. Finally return the new Document D’ containing the expanded words also of size n.



```python
abbreviations = {"ASAP", "won't", "etc."} # Add more abbreviations as needed

def replace_keywords(document):
    words = document.split()
    for i, word in enumerate(words):
        if word in abbreviations:
            # Replace with full form or other keyword as needed
            if word == "ASAP":
                words[i] = "as soon as possible"
            elif word == "won't":
                words[i] = "will not"
            # Add more elif conditions for other abbreviations
    return " ".join(words)

# Example usage
document = "I need to finish this ASAP, but I won't be able to."
new_document = replace_keywords(document)
print(new_document)
# Output: "I need to finish this as soon as possible, but I will not be able to."
```

Analysis

In this implementation, the replace_keywords function takes a document as input, tokenizes it into words using the split() method, and loops through each word to check if it is an abbreviation. If the word is an abbreviation, it is replaced with its corresponding full form or other keyword as needed. Finally, the modified list of words is joined back into a string using the join() method and returned as the output.

Proof of Correctness

Loop Invariant: At the start of each iteration of the loop, all words up until the current word have been correctly examined by being replaced with their full forms if they were abbreviations that exist in the HashMap.

Initialization: Before the loop begins, we create a HashMap H with the abbreviations being the keys and the values being their corresponding full forms. Hence, generating a valid HashMap that maps each abbreviation to its full form accurately, proving the loop invariant to be true at the start of the loop.

Maintenance: At each iteration of the loop, we inspect the current word and verify if it is an abbreviation in the HashMap. If yes, we replace it with its full form. If not, we keep the word unchanged. This maintains the loop invariant because we are only changing the value of a word if it is an abbreviation in the HashMap (has a matching key). The rest of the words are unchanged.

Termination: At the loop termination, we have explored every word d[i] in the document and replaced all abbreviations with their corresponding full forms. Therefore, the output file generated by this algorithm will be correct if the HashMap / dictionary is built correctly.

Time Complexity: The time complexity of step 2 is O(m), where m is the number of abbreviations. The time complexity of step 3 is O(n), where n is the number of words in the document. The time complexity of steps 4 and 5 is O(1), since hash table lookups and insertions are constant time operations on average. The time complexity of step 6 is also O(1), since appending to a list is a constant time operation. Therefore, the overall time complexity of the algorithm is O(n+m).

Space Complexity: The space complexity of the algorithm is O(m), since we need to store the abbreviations and their full forms in a hash map. The space complexity of the expanded document is also O(n), since it may be the same size as the original document. Therefore, the overall space complexity of the algorithm is O(m+n).

Unexpected Cases.

1. Duplicate keys: If the list of abbreviations has duplicate keys, only the last key-value pair will be stored in the hash map. This could result in the algorithm replacing the wrong abbreviation in the text.
2. Case sensitivity: The algorithm as it is currently implemented is case-sensitive, meaning that it will only replace abbreviations that match the case of the keys in the hash map. This could result in some abbreviations not being replaced if they are written in a different case than what is stored in the hash map.
3. Word boundaries: The algorithm does not currently take into account word boundaries, meaning that it could potentially replace parts of words that happen to match an abbreviation. For example, if "won't" is in the list of abbreviations and "wonton" appears in the text, the algorithm would replace "wont" with "will not" even though it is part of a larger word.
4. Ambiguity: There could be cases where an abbreviation has multiple possible expanded forms depending on the context. For example, "CIA" could stand for "Central Intelligence Agency" or "Confidentiality, Integrity, and Availability" depending on the context. In such cases, the algorithm may not be able to correctly determine the appropriate expanded form.

Task Separation and responsibilities:

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

Questions: Should some edge cases be features? Or be handled in one main method. Probably separate methods. Edge cases - format of text/ language of text / length of text/ updating abbreviation lists, etc.

We have handled an edge case by using the join function in python to perform concatenation and appending strings (works similarly to String Builder in java). The join function can handle empty strings by adding separators between them; large input lists without causing performance disputes or exhausting memory, as it processes the input list one element at a time and concatenates the strings incrementally though in the case of HashMap this isn’t really a concern. Also join function will raise a Type Error for null values