There are a total of 3 approaches that I could think of to solve this problem. At first glance, it looked like a case insensitive pattern matching problem and I build my way up from there. I feel the best approach is the last one (DictionaryMatch).

There are a total of 4 executable, including the tests.

**How to Run?**

Build

In the repository, do the following:

1. mkdir build && cd build

2. cmake .. -DBUILD\_TESTING=ON

3. make

Run(inside build foder):

4. Run the executables as:

./PatternMatching

./PatternMatchingStl

./DictionaryMatch

./tests

**Parameters**

Inputs**:**

1. The concepts are read from the file concepts.txt in the project folder. Here, one can add more concepts and delete existing.
2. The input text is read from the console.

Ouputs:

Outputs for each input are displayed in the console

**1. Pattern Matching - with Stl**

Implemented in: PatternMatchers.h

In this approach, we iterate through each of the 'concepts' to see if they exist in the 'input'. This was implemented completely using Stl algorithms. Since, the algorithm should be case insensitive, special consideration was given to changing the 'input' and each 'concept' to lower case. We use Boyer Moore Search for the pattern matching.

Advantages:

1. Fast when the concept list is small.

2. Less code that makes use of stl algorithms, hence very easy to understand.

Disadvantages:

1. Modifies the input in-place (This is done to make the algorithm faster).

2. Slow as the number of concepts increases because each concept is checked for its occurrence.

3. Makes extra allocations for creating the lower case concepts. This was needed to retain the original concepts.

Time Complexity(Worst Case):

Let 'n' be the size of 'input', 'm' be the number of concepts and k be the size of largest concept

Complexity(lower(Input) + lower(m Concepts) + Bayer Moore)= O(n + km + nkm)

Space Complexity:

Creation of copies for each the concepts.

O(m)

**2. Pattern Matching - Without Stl**

Implemented in: PatternMatchers.h

The previous algorithm, makes extra allocations for converting ‘concepts’ to lower case. In this algorithm, I implemented a custom Bayer Moore Search with converts each character to lowercase while comparing. This does not create any copies and hence requires no extra memory allocations.

Advantages:

1. Fast when the concept list is small.

2. Does not modify the input in-place.

3. Does not make extra memory allocations to store a copy of each concept.

Disadvantages:

1. Code hard to understand for a developer unaware of Bayer Moore Search.

2. Slow as the number of concepts increases because each concept is checked for its occurrence.

Time Complexity(worst case):

Let 'n' be the size of 'input', 'm' be the number of concepts and k be the size of largest concept

Bayer Moore for m ‘concepts’ = O(mnk)

Space Complexity:

O(1)

***Note****: Both these algorithms are done using templates and are made to have almost the same interface as STL std::search, returning an iterator to the found location. If the concept is not found, it returns input.end(). To differentiate both the algorithms, we use tag dispatching in C++ with an extra parameter.*

**3. Dictionary Match**

Implemented in: Dictionary.h, Dictionary.cpp

The main problem with the above two apporaches is that as the numbers of concepts grows, they becomes terribly slow. In this approach, I create a dictionary based on the first letter of each concept. Here is the psuedo code.

1. Create a dictionary with key as the first letter, mapping the first letter to all the ‘concepts’ that has the same first letter.

Example: concepts: Indian, West Indian, Indonesian

dictionary = {‘I’: {‘Indian’, ‘Indonesian’}, ‘W’:{‘West Indian’}}

2. For each word in ‘input’:

* find first letter of word
* fetch words beginning with letter from map
* Check if fetched words are present in the input using PatternMatching-WithStl
* If word is present:

add word to an array

3. If array.empty()

add “none” to array

***Note****:*

*1. Each word in the corresponding key of the dictionary is checked for its occurrence in the input. If we do the reverse and check for each ‘Input’ word against the dictionary, then we miss out of ‘concepts’ that consists of two or more words:*

*eg: Input: I like West Indian, Concepts: Indian, West Indian*

*if we check each word in input, we will get only Indian as the matches.*

*2. Once a letter is seen, it is saved into a std::set, to make sure that this letter is not checked again. This is because once a letter is seen, we check the occurrence all the words in the dictionary mapping for that letter in the Input.*

Advantages:

1. Considerably faster than the previous two approaches when the numbers of concepts are large and diverse.

2. Can be made more efficient with a more complex map.

Disadvantages:

1. Modifies the input in-place to convert it to lowercase since it uses PatternMatching-WithStl

2. It is assumed that all concepts start with a capital letter. Thus all the keys of the dictionary are capital letters.

Time Complexity(Worst):

Let 'n' be the size of 'input', 'm' be the number of concepts and k be the size of largest concept

The worst case is when all the concepts begin with the same letter. In this case, we have the same complexity as PatternMatching-WithStl, because the dictionary will just have one key with all concepts as its value.

ie. O(n + km + nkm)

The best case complexity is much better than PatternMatching-WithStl, because lookup in dictionary is O(1), while iterating in std::vector is O(m).

Space Complexity:

Creation of dictionary + PatternMatching-WithStl

O(m) + O(m) = O(m)

Can this approach be made faster?

1. Use a more complex dictionary that maps dictionary to dictionaryies.

2. Use multithreading

In this case, we divide the input text between the different threads and process each part independently.