DSA 3 – Assignment 1

21002169-Wickrama G.N.S

Limitations and assumptions

This program offers a limited use of regular expression patters as defined below.

- No two special symbols shall be used in a single pattern
- the alphabet of our pattern and text will not include the regular expression symbols { [,] , ^ , $, . } as they will be used for search
- [ ] will define a specific range
    - Ex :
    - text[abc]text will result in positive matches for textatext , textbtext , textctex. However the use of "-" will not be allowed (text[a-c] will not be allowed)
- " ^ " will be used to match a string at the starting point from a line

    Ex:

    Text – "word1 word2 word3 word4 word1"

    Pattern - "^word1"

    Index 0 will be retuned as a match however the second instance if word1 will not be a match

- " $ " similar to "^" however matching will be done at the end of a line
- "." The wildcard can be used to represent any character in our alphabet
    - For the pattern g.t , texts such as got , get , gyt will return as matches
- This algorithm does not match words it matches patterns and " " (space) is part of the alphabet meaning that in the text "helloworld" the pattern "hello" will return the index 0 if one wishes to match the word a space would have to be included.
- Patterns are case sensitive

The string-matching algorithm

The primary advantage of KMP is that, after detecting a mismatch, it doesn't backtrack over the text. It uses the information about how much of the pattern has been examined to skip unnecessary comparisons.

2. Why KMP?

Linear Time Complexity:

The time complexity of KMP is O(n + m) where n is the length of the text and m is the length of the pattern. This ensures a fast search.

No Backtracking:

KMP efficiently uses the previously gathered information, ensuring that characters of the text are never compared more than once.

How to Run

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
4  #include <fstream>
5  #include <algorithm>
6
7  using namespace std;
8
9  string input_Pattern = "pattern1.txt" ;
10 string input_Text = "text1.txt";
11 string output_file = "out1.txt" ;
```

Include the name of the pattern and text as shown in line 9 and 10 and the output file in line 11

If the name of the pattern Is patten1.txt and text is text1.txt and output file is out1.txt;

sting input_Pattern = “pattern1.txt”;
string input_Text = “text1.txt” ;
string output_file = "out1.txt" ;

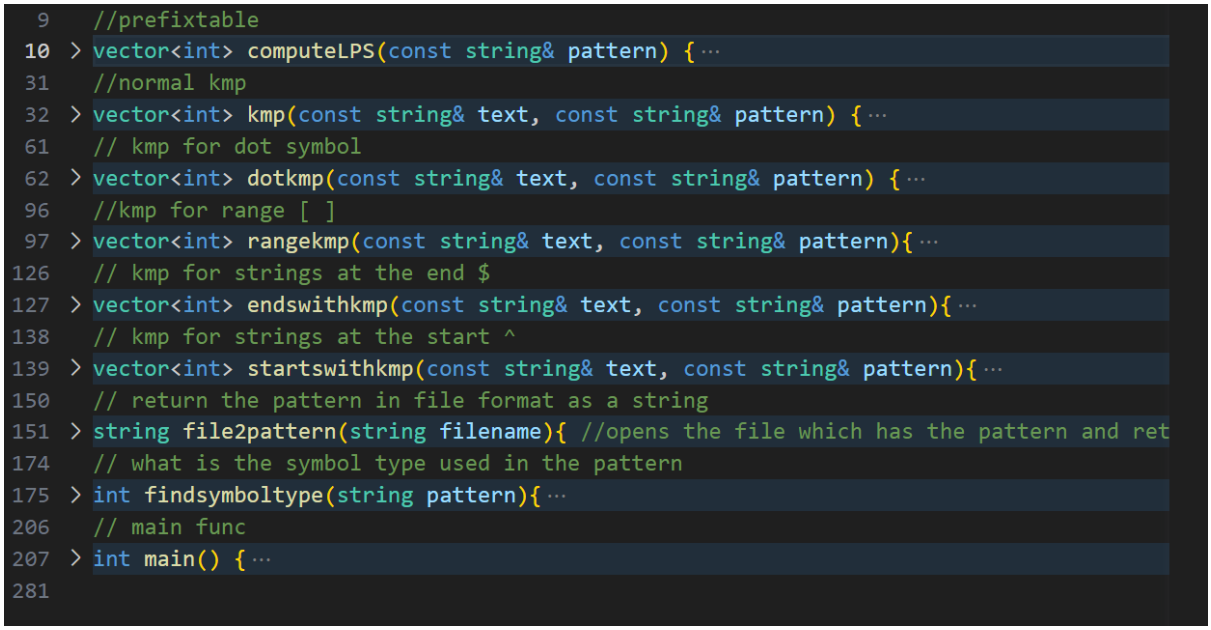
All files must be in the same folder as the KMP_21000603.cpp file

Output will be printed to console as well as saved in the out file

DSA 3 – Assignment 1

21002169-Wickrama G.N.S

Explanation of the code



Since a lengthy discussion of the principles of the selected string-matching algorithm is not needed, we will only briefly discuss the code related to the KMP algorithm (computeLPS and kmp)

computeLPS()

This function calculates the Longest Prefix which is also Suffix (LPS) array for a given pattern string

KMP()

If the character in the text (text[i]) matches with the current character in the pattern (pattern[j]), then both i and j are incremented.

If j becomes equal to m (length of the pattern), then it means a match of the pattern is found in the text. The starting index of this match in the text is i - j, so it's pushed to the occurrences vector. Afterwards, j is reset based on the lps array to continue searching for other occurrences.

If the characters don't match (pattern[j] is not equal to text[i]),

two scenarios arise

If j is not 0, it means we can use the lps array to skip some comparisons. j is updated to lps[j - 1].

If j is 0, then we simply move to the next character in the text by incrementing i.

Finally, after the entire text is traversed, the function returns the occurrences vector.

dotkmp()

Used to match the wildcard " . "

The function uses the KMP algorithm with the only change being made in line 77 where if the dot is encountered in the pattern, then it allows the algorithm to **skip it as a match would have been made** since the dot can match with any character

```
if (pattern[j] == '.'){ //skips the dot
    i++;
    j++;
```

rangekmp()

Used to match a range of characters given as pattern[abcdef]pattern

The idea is to first store all the contents withing [ ] in a string called "listing" .

After this we iterate through the characters of listing replacing the parts enclosed by [ ] with the single char characters found in listing and run kmp for each possible instance

```
97  vector<int> rangekmp(const string& text, const string& pattern){
98
99
100
101     string test = pattern ;
102
103     int start = test.find('['); //find index of [
104     int end = test.find(']'); //find index of ]
105
106     string listing = test.substr(start+1 , end-start-1); //save the substring ie [su
107     //cout << listing ;
108     vector<int> occurrences; //array to store indexs where kmp finds a match
109
110     for (int i = 0; i < listing.size(); ++i) {  //iterates over all the charcters in
111     string character(1,listing[i]);
112
113     string temp = test;
114
115     temp.replace(start, end-start+1 , character); //change example_[ABC]_example to
116
117     vector<int> kmpresult = kmp(text,temp); //run kmp normally on changed string
118
119     occurrences.insert(occurrences.end(),kmpresult.begin(),kmpresult.end()); //appen
120
121     }//this loop will run from A1 to An , where text[A1...An]text <--- pattern
122
```

Start and end in line 103 and 104 will have the starting index of the substring meaning the content which is in the parentheses. This will be used for computation later

Line 106 is used to store the content of the substring in the “listing” string

After this the for loop in like 110 runs through all the characters in the “listing” sting adding them to the pattern and running kmp for each combination

Within this loop all the instances of the matched indexs are added to the array “occurrrences”

This is then finally returned via this function which will be later used in main

endwithkmp()

used for the “$” symbol to match a pattern at the end of the line

```
vector<int> endswithkmp(const string& text, const string& pattern){

    string ptemp = pattern ;
    string ttemp = text ;
    ttemp.push_back('$');   //appends $ to end of pattern
    ptemp.push_back('$') ;  // appends $ to end of text(line of the text passed into

    return kmp(ttemp , ptemp); //now runs kmp normally


}
```

if we were to use the pattern “ABC” and the text “ABC 1 2 3 4 ABC” we should only match the last instance of “ABC”. However, if we simply use our kmp algorithm for this matching it will result in a match for both instances.

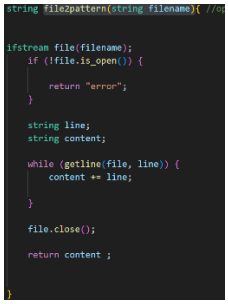
The solution to this problem is to make the last instance of ABC and only the last instance unique. We do by appending the “$” to the end of the line of the text and pattern resulting in the the pattern becoming “ABC$” and text becoming “ABC 1 2 3 4 ABC$”. Now when kmp is used on this only the last instance of ABC will result in a match and since the symbol “$” has been defined as not being part of our language no problems will arise

Startswithkmp()

Used for the “^” symbol to match a pattern at the start of a line

The implementation is similar to endwithkmp() but rather than appending “$” top the end we append “^” to the start of the patter and line. And it works for the same reasons stated above

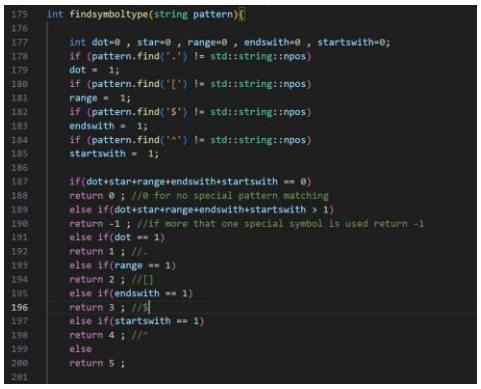
file2pattern()



```
string file2pattern(string filename){ //o

ifstream file(filename);
    if (!file.is_open()) {

        return "error";
    }

    string line;
    string content;

    while (getline(file, line)) {
        content += line;

    }

    file.close();

    return content ;


}
```

simple function to open the pattern file and return the pattern as a string which can be store later

Findsymboltype()



```
int findsymboltype(string pattern){

    int dot=0 , star=0 , range=0 , endswith=0 , startswith=0;
    if (pattern.find('.') != std::string::npos)
    dot =  1;
    if (pattern.find('[') != std::string::npos)
    range =  1;
    if (pattern.find('$') != std::string::npos)
    endswith =  1;
    if (pattern.find('^') != std::string::npos)
    startswith =  1;

    if(dot+star+range+endswith+startswith == 0)
    return 0 ; //0 for no special pattern matching
    else if(dot+star+range+endswith+startswith > 1)
    return -1 ; //if more that one special symbol is used return -1
    else if(dot == 1)
    return 1 ; //.
    else if(range == 1)
    return 2 ; //[]
    else if(endswith == 1)
    return 3 ; //$
    else if(startswith == 1)
    return 4 ; //^
    else
    return 5 ;

```

Function used to identify the symbol in the given pattern

0 if no special character has been used

-1 if no more than one special character is used. As defined in our limitations this will result in an error

Other symbols with their respective return value are shown in the code

Main() function

First the pattern file is opened and stored in the string pattern_fromfile using the file2pattern function

Then the text is opened and read line by line

Each line along with the pattern is passed to the relevant modified kmp algorithm and the indexes where matches occur are store in the indices array which after the line is done reading is printed to the console.