csci2271 Spring 2018

PP 1: Pattern Matching

Fun with memory, especially the use of dynamic arrays (malloc) and pointers. In this assignment, you will do some advanced pattern matching. In this assignment you will use a FSM recognizer to keep track of the char string you are searching for. The FSM recognizer simply makes decisions on what to do based on the state and next char value. So, you never really store the historical char values, you move from state to state. In this assignment, you will use pointers and pointer arithmetic (no array notation permitted, so no array[index] allowed). To do this, you will need to read characters into a char array (defined in main), pass parameters by address to functions and return pointers.

String-matching is a very important subject in the wider domain of text processing. String-matching algorithms are basic components used in implementations of practical software existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science (system or software design).

String-matching consists of finding at least one of the occurrences of a string (more generally called a pattern) in a larger text. The text has ***n*** characters in length (unknown before execution, but maximum size is 80), while the pattern has length of ***m*** characters, where ***m*** is less than or equal to ***n***. Both character strings (really just an array of characters) consist of a finite set of characters and will be limited to the standard letters in the English alphabet. The approach is not the most efficient method to solve pattern matching, but for this assignment it will be fine.

Part A:

Build your FSM recognizer on paper to recognize the pattern:  
"**GCAG**".  
Using FSM recognizer, you will "recognize" (that is, determine if the pattern is in the text). The text is read (one char at a time) and stored in a char array. If the pattern is recognized, then you will output a nice statement and indicate where in text the pattern starts.

 For example:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TEXT | | | | | | | | | | | | | | | | | | | | | | | |
| G | C | A | T | C | G | C | A | G | A | G | A | G | T | A | T | A | C | A | G | T | A | C | G |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PATTERN | | | | |
| G | C | A | G |  |

A Match Has Been Found!!!!

And It Starts At location 5 (starting from the left at 0).

Now, what will your program do? First, you should assume you are searching for the pattern: **"GCAG"**.  
You will code several functions as well as a main function. There should be one function that reads in some number of characters from the terminal and stores them into an array. You will use this function to store characters into your text array. You will also have a function that implements the FSM recognizer. This function will have passed to it two char **pointers**, and **returns** **an int which is the index** to the location where the match **starts** or it returns **-1** if it could not find a match (you should use this in your main function to print out the proper phrase "Yippee a match is found at location x" or "Too bad no match").

You will also create a function that is passed the char pointer of the TEXT and an int index where the match was found in the text, and of course the size of the text. This function should print the remaining characters in the TEXT string. For example, in the above the match was found, and the **remaining characters** in the TEXT are

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | G | A | G | T | A | T | A | C | A | G | T | A | C | G |

A possible structure to your program is:

#include <stdio.h>

#include <stdlib.h>

int main(){

char \*text, \*pattern; //pointers for the characters you will read

char \*p,\*q,\*r; //some pointer variables

int n,m,x,y; //some integers

char \* getCharBlock();  
//this would fill in the "string" of chars for the passed in char pointer. YOU SHOULD  
//place a '\0' character at the end of the "string" (as an end of string sentinel)

//this code MUST use pointer arithmetic

int findMatch(//what should be passed?);   
//looks for a match, returns **-1** for failure or **an int which is the index** to the location where the match **starts**.

//the return values can be used to determine IF a match was found, and where.

void printIt(char \*ptr, int index, int size);

//this is simple, just remember how pointer arithmetic works

//prints a "string", starting from the pointer "index"

//and any more functions for clarification of work done by the program. Remember, by designing and implementing //your code well, you simply need to call the functions nicely in main to find **multiple locations** of the **pattern** in the //**text**.

}

This is very similar to the last assignment, however, this time you will be more general and use more C conventions. In particular, you will use malloc, to store the char "strings" on the heap. Be very careful doing this, memory access here is somewhat tricky and you need to take care so start early on this.

Part B:

The above problem was to find a pattern in the text for "contiguous" characters in the text. What if we relaxed the requirement to allow for just order, but not contiguous characters? That means, if we are searching for the pattern : "GCAG" in the above text, we would find a match in the non-contiguous portion of the text **"GCATCG"**(note, if we ignore the "TC" characters, we have found **"GCAG"**). Using the material from Part A, adjust your FSM recognizer (and thus the findmatch() function) to handle this "relaxed" constraint. As before, print out if a match is found and the position in the text of the start of the match.

Part C:

In Part A, the problem was to find ONE instance of the pattern in the text for "contiguous" characters in the text. What if we wanted to find ALL instances of the pattern in the text? For example, if the TEXT was: GCAGCAGCAG  
there would be three matches. Print out how many matches (don't worry were they start, just count them). Using the material from Part A, adjust your FSM recognizer (and thus the function findmatch() ) to handle this new constraint.