Practice Arena

Practice problems aimed to improve your coding skills.

How Mr C reads your code

LABEXAM-PRAC-02_ENDSEM

How Mr C reads your code [70 marks]

Problem Statement
When you write code in the C language, the compiler has to make sense of what you have written, for example, what are the variables in your program, are they integer or character variables, what are the functions in your program, where are the brackets in your program, etc. To do so Mr C does a lot of pattern matching using what are called regular expressions. Regular expressions are basically patterns and correct C programs always obey those patterns.

The first line of the input will be a string that will only contain the following characters: upper and lower case English characters i.e a-z and A-Z, digits 0-9, the plus symbol + and the multiplication symbol *. Let us call it the pattern string. The second line of the input will be another string which will only contain upper and lower case English characters i.e a-z and A-Z. Let us call it the message string. You have to find out whether the message string follows the pattern given in the pattern string or not. The pattern and message strings will contain no more than 99 characters.

The pattern string will always give you an alphabet character (either upper or lower case) followed by either a single digit, or else the symbols + or *. Let us call this pair a token. The pattern string will basically be a sequence of such tokens. For example a4c+a* has 3 tokens [a4][c+][a*]. Read the string from the input and store and create an array of variables of type Token (see structure definition below). The above pattern is interpreted as follows

1. The token [a4] means that the character 'a' (without quotes) should appear exactly 4 times
2. The token [c+] means that the character 'c' (without quotes) should appear one or more times
3. The token [a*] means that the character 'a' (without quotes) should appear zero or more times

Thus, the second character of the token tells us how many times the first character must repeat. This is why we call the second character a repetition character. Thus, the above pattern is interpreted as a command "a should appear exactly 4 times followed by one or more appearances of c followed by zero or more appearances of a". Suppose the message string is "aaaac" (without quotes). Then it is clear that this message follows the command as it has four 'a' followed by one 'c' (the third token asks for zero or more 'a' after the 'c's are over and this string has 0 'a's after the 'c's are over which is fine). However, the string "aacccaa" does not follow the pattern since it starts off with 2 'a' whereas the pattern demands that the string start with four 'a'.

Some other strings that satisfy the pattern are "aaaacccaaaa" and "aaaacccc". Some other strings that violate the pattern are "aaaa" (there is no c but there should be at one or more c after the 4 a), "aaaaa" (missing c), "Aaaaca" (case error) and "abaaca" (the pattern does not allow b to appear anywhere in the string).

Now, we might also give you patterns of the following kind "a*a4a+b+" where two consecutive tokens in the pattern have the same character. You should not try to match the message string with such patterns because it will be very hard to do so (since it is hard to figure out which ‘a’ in the message string corresponds to which token in the pattern string). For such cases, you must first "reduce" the pattern string to an equivalent pattern string - we will call this new pattern string the reduced pattern string). The reduced pattern string has two properties

1. The reduced pattern string cannot have '0' or '+' as a repetition character for any token.
2. In a reduced pattern string, if two consecutive tokens have the same first character, then the first token must have repetition character as a digit and the second token must have the repetition character as *. For example a6a* is a reduced pattern that demands 6 or more a but

a5a+, a4a2a*, a+a* are not reduced strings. However a+ is also not a reduced string since it contains the repetition character +

3. If a message obey the pattern string it must obey the reduced pattern string and vice versa.

To find the reduced pattern string, let us see what command does the original pattern string give. The pattern "a*a4a+b+" commands that "there should be zero or more 'a' followed by exactly 4 'a' followed by one or more 'a' followed by one or more 'b'". However, it is easy to see that this command is the same as "there should be 5 or more 'a' followed by one or more 'b'" which corresponds to the pattern "a5a*b+".

Some other examples of pattern reductions are given below

a+ => a1a* (both demand 1 or more a)
a+a5 => a6a* (both demand 6 or more a)
a+a+ => a2a* (both demand 2 or more a)
a+a* => a1a* (both demand one or more a)

a5a4 => a9 (both demand 9 a)
a5a+ => a6a* (both demand 6 or more a)
a5a* => a5a* (already in reduced form)

a*a5 => a5a* (ordering of consecutive tokens with the same first character as per rule above)
a*a+ => a1a* (both demand one or more a)
a*a* => a* (both demand zero or more a)

p4p2q1 => p6q1 (both patterns demand 6 'p' followed by a q)
h4h+ => h5h* (both patterns demand 5 or more 'h')
a4a2a* = > a6a* (both demand 6 or more a)
A1A*A5A+s5 => A7A*s5 (1 A then zero or more A then 5 A then one or more A then 5 s is the same as demanding 7 or more A then 5 s)

In your output you have to first print the reduced pattern string in the first line (if the given pattern is already in reduced form, simply print the original pattern string once more). Then in the second line you have to print "YES" (without quotes) if the message string given to you obeys the pattern string or not (due to the properties of the reduced pattern strings described above, the message string will either obey, both the original pattern string and the reduced pattern string or it will obey neither.

Compulsory structure usage in your code
In your code, you should use variables of type Token where the structure is defined below:

```
struct Token{
   char c;
   char repetition;
};
```

Create an array of Token variables to represent the given and reduced pattern strings.

```
int n;
struct Token arr[n];
```

Be warned that not using such a structure to write your code will cause you to lose a small number of manual grading marks.

Problem-specific Words of Caution:

1. **Do not forget to submit your code**. You can submit multiple times. Your last submission will get graded.
2. Although the pattern string we give you may have tokens where the repetition character is '0', your reduced pattern must never contain a token with a repetition character '0'.
3. We assure you that we will never give you a case where the reduced pattern string looks like a11b5c* since here, the repetition count is 11 which cannot be stored in a single character. The reduced patterns in our test cases will always require only single digits as the repetition characters.
4. We will not insist that you write separate functions, other than main, to solve this problem. However, solutions that do write functions to, for example, reduce the pattern to its lowest form, will get more marks.
5. Reduce the pattern to its absolute lowest form. Partially reducing the pattern will not receive any marks form the autograder.

General Grading Policy

1. **TOTAL MARKS OF THE EXAM** 20 + 40 + 40 + 70 = 170
2. **TOTAL DURATION OF THE EXAM** 3 hours 30 minutes
3. See below for question-specific details of how partial marking would be done by the autograder in this question
4. Your submissions will be inspected by the autograder as well as a human grader
5. Human graders will (among other things) allot marks for the following

    1. Neatly structured code that uses at least one function other than the main function to process the input. The questions will usually suggest how to use functions to process the input. Submissions that ignore these suggestions and use only the main function to solve the entire problem, will lose a small fraction of marks.

    2. Proper and meaningful variable names

    3. Nice looking and consistent indentation

    4. At least a couple of comments explaining to the human grader what are you doing, especially when the steps are not obvious

    5. Comments, good indentation and meaningful variable names are very important for the human grader to understand what are you doing and why. If they cannot understand your code, do not expect them to give you (partial) marks either.

6. Solutions that indulge in hard-coding **will get a straight zero** even if they are passing some test cases. Hard-coding is a form of cheating strategy where someone write code of the form "if(input == A) printf(B)" without doing any calculations on A to obtain B. The values of A and B are either read from the evaluation/submission window or else guessed.
7. Be careful about extra/missing lines and extra/missing spaces if you do not want to lose autograder marks
8. Proportion of marks allotted to autograder (in particular, weightage to visible and hidden test cases) and human grader will be revealed when marks and grading rubrics are released

9. You are allowed to use the libraries stdio.h, math.h, string.h, stdlib.h **but not any other library**. Use of unpermitted libraries will carry a penalty. You may use any programming tools that we have discussed in lectures/tutorials or in lab questions such as arrays (1D, 2D, 3D, arrays of arrays etc), strings, loops, structures, functions, recursion, pointers, linked lists, stacks, queues, graphs, enumerations, flags, conditionals, global, static and shadowed variables.

EXAMPLE:
INPUT
```
a4c+a*
aaaac
```

OUTPUT:
```
a4c1c*a*
YES
```

Grading Scheme:
Total marks: **[70 Points]**

There will be partial grading in this question. There are two lines in your output. Printing each line correctly, in the correct order, carries 50% weightage. There are 4 visible and 4 hidden test cases.

Please remember, however, that when you press Submit/Evaluate, you will get a green bar only if all parts of your answer are correct. Thus, if your answer is only partly correct, Prutor will say that you have not passed that test case completely, but when we do autograding afterwards, you will get partial marks.

Start Solving! (/editor/practice/6252)