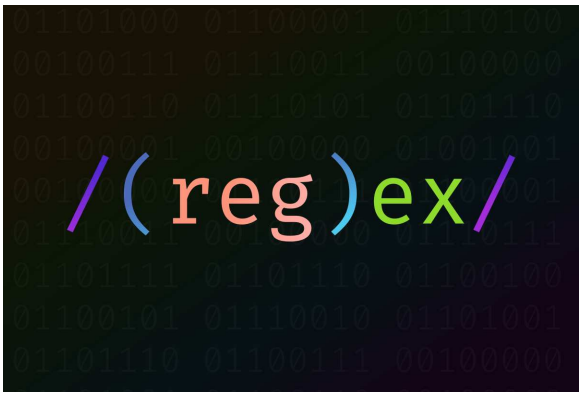
Regular Expressions

In this assignment, you will implement a simple regular expression parser (with a very simplified input format). The goals of this assignment are to:



- Gain an appreciation for how simple our theoretical understanding of regular exprressions can be applied to implement a real regular expression parser.
- To see the connection between our proof that regular expressions are equivalent to DFAs by implementing some code that converts between them.
- To appreciate the lack of memory that is necessary to implement a regular expression parser.

Regular expression matchers are a commonly used tool in computer science, and they directly relate to the concept of a DFA from class. For this, assignment we will give you a simple regular expression and a series of strings that may or may not match that expression. For each string, you will output *YES* if the string matches the expression, and *NO* if it does not.

There are many different ways to implement such features, but I would like you to use the implementation described below so that your work best reflects our discussion from class. This is despite the fact that there are more memory efficient implementations. The rest of this document will describe the input you can expect, the output you should provide, along with some details on how you should implement this.

Input

All input will be provided to standard input (cin, input(), System.in, etc.). The first line of input is the regular expression that needs to be parsed. It has a very simplified input. Each character in the string will be one of the following:

- **d**: The lower-case character d matches to any single numeric digit 0-9.
- **a**: The lower-case character a matches to any lower-case alphabetical letter a-z.
- **2-3**: If you see that character 2, or 3 in the expression, it means the previous character (a or d) must appear exactly 2 times, or exactly 3 times (depending on the number). For example, *d2* would match with *23* but not to *4* because the string MUST have two numeric digits.
- ***:** The star symbol means the previous character can appear 0 or more times. For example, $d\times$ represents zero numeric digits up to any number of numeric digits and would match with, for example, the empty string, 1, 354, 89876, etc.
- **+**: The plus symbol means that the previous character must be present once, but can then be present as many times after it as necessary. Note that $d+ \equiv dd*$.

There will never be a + symbol and symbol adjacent to one another. There will also not be any precedence by parentheses (that makes the assignment a bit harder).

The next line will contain a single number n, which is the number of strings that will be given to match with. The next n lines after that will each give a single string that is a potential match to the regular expression.

Output

For each of the n example strings. Output *YES* on a single line if the string matches the regular expression, and *NO* if it does not.

Getting Started

Even with such a simple format, this is not a trivial implementation. I would like you to proceed using the following approach. As stated earlier, there are several ways to implement this, but I would like everybody to use the approach below.

- **Step 1: Implement a DFA object**: The first thing you should do is create an object (class) that represents a DFA. This object will contain fields for each part of a DFA (list of states, the start state, list of final states, current state, list of transitions). You should create methods that allow you to add states, add transitions between states based on characters (numeric digits or alphabetic characters).
- **Step 2: Parse the input into tokens**: Given a regular expression string (e.g., *a3da*), split the expression into tokens. Each token should be a single character (a or d) or a single character (a or d) with one other character (2, 3, +, or *). For example, you should split *a3da* into the tokens *a3 d a*.
- **Step 3: Instantiate the DFA object that represents the regular expression**: Instantiate a DFA object. Then, for each token from step 2 above, craft a part of the DFA and add the appropriate transitons to previous parts of the DFA. By the end of this step, you should have a complete DFA that represents the entire regular expression.
- **Step 4: Simulate the DFA on each string**: Have your DFA begin at the start state and read in the matchable strings one character at a time. For each character, transition to a new state if possible. If you end in a final state, print *YES*, otherwise print *NO*.

Submission

You should submit your code in Java, C++, or Python. You should configure your makefile properly to run your code. See the course website for details on how to configure submissions for Gradescope

Sample Input

```
a3da+
5
mrf8t
mk5sc
lab2d
bea3ch
lbh1
a*d2a* 4 23 a56 bas98edd as98f7f
```

Sample Output

```
YES
NO
YES
YES
NO
YES
YES
YES
NO
```