CSE 102 Spring 2021
Advanced Homework Assignment 5

Jaden Liu
University of California at Santa Cruz
Santa Cruz, CA 95064 USA

May 25, 2021

1 AdvHW5

6. Regular Expression Problem Given an input string s and a pattern p, design a dynamic programming algorithm that implements regular expression matching with support for . and $*$ where:

   - '.' Matches any single character.
   - '*' Matches zero or more of the preceding element.

   The matching should cover the entire input string (not partial.

   If needed, you may use any or all of the following or similar constraints (state clearly which constraints, if any, you are using):

   - $0 \leq$ s.length ≤ 20
   - $0 \leq$ p.length ≤ 30
   - s contains only lowercase English letters.
   - p contains only lowercase English letters, '.', and '*'.
   - It is guaranteed for each appearance of the character '*', there will be a previous valid character to match.

Solution. We can come up with a 2-dimensional table c[i][j] to represent if first j arugments p[j] matches to first i elements s[i].
1. Proof of optimality
For this question, it satisfy the principle of optimality, which we can prove by contradiction. Let i = len(s), j = len(p), then we have an optimal solution for the question, let $0 < k < i$ and c[k][j-2] or c[k][j-1], depending on if there is a "*", to be the optimal solution for previous matches of substring. Assume k is not the optimal match, then there exist l position in substring for optimal previous math. Then we can replace k with l to obtain a better matches, which generates a contra-

diction that c[i][j] is the optimal one. Thus we prove it has the optimal structure.
2.Recurrence

We have two situation in when we are matching two strings:

a) If the jth character of p is a lowercase letter, then we must match the same lowercase letter in s, that is:

$$c[i][j] = \begin{cases} c[i-1][j-1], s[i] = p[j] \\ false, s[i] \neq p[j] \end{cases}$$

In other words, if the i-th character of s is not the same as the j-th character of p, the matching cannot be performed; Otherwise, we can match the last character of the two strings, and the complete matching result depends on the front part of the two strings.

b)If the j-th character of p is ”*”, then we can match the j-th character of p any number of times.

$$c[i][j] = \begin{cases} c[i-1][j-2], no\ match \\ c[i-2][j-2], one\ match \\ c[i-3][j-2], three\ matches \\ \vdots \\ c[i-1-k][j-2], k\ matches \end{cases}$$

Or, easily written as:

$$c[i][j] = \begin{cases} c[i][j-2], no\ match \\ c[i-1][j], one\ or\ multiple\ matches \end{cases}$$

It’s because when we have a match, just repeat this comparison it with previous character in s until reaching an difference.

c) If j-th character is ”.”, then we must have a match.
Then we have final recurrence:

$$c[i][j] = \begin{cases} if\ \ j\ lower\ case: \begin{cases} c[i-1][j-1], s[i] = p[j] \\ false, s[i] \neq p[j] \end{cases} \\ if\ \ j\ equals\ "*": \begin{cases} c[i][j-2], no\ match \\ c[i-1][j], one\ or\ multiple\ matches\ or\ j-1\ is\ "." \end{cases} \\ if\ \ j\ equals\ ".": \begin{cases} c[i-1][j-1] \end{cases} \end{cases}$$

```

        recurrence(c[][],i,j):
                if i=0 and j=0
                        return true
                if j == "*":
                        if s[i] != p[j]:
                                c[i][j] = recurrence(c[][],i,j-2)
                else:
                        c[i][j] = recurrence(c[][],i-1,j)
        elif j == ".":
                c[i][j] = recurrence(c[][],i-1,j-1)
        elif j == [a-z]:
                if s[i] == p[j]:
                        c[i][j] = recurrence(c[][],i-1,j--1)
                else:
                        return false
return
```

3. DP algorithm

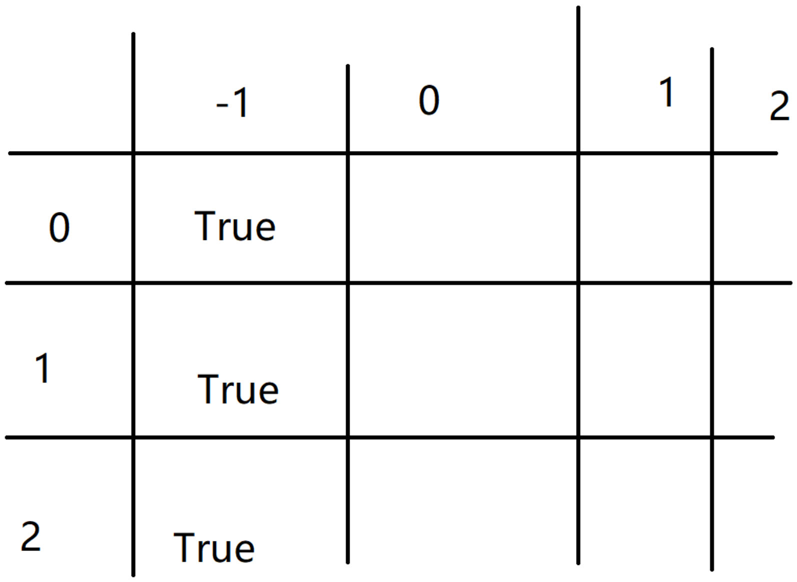
```

fill(len(s),len(p))
        c[][] = string [len(s)][len(p)]
        for i=0 to len(s):
                for j=0 to len(p): # fill the table
                                # from top to down, left to right
                        if p[j] == "*":
                                if match(i,j-1):
                                        c[i][j] = c[i][j-2]
                                else:
                                        c[i][j] = c[i-1][j]
                        elif p[j] == [a-z]:
                                if match(i,j):
                                        c[i][j] = c[i-1][j-1]
                                else:
                                        c[i][j] = false
                        elif p[j] == ".":
                                c[i][j] = c[i-1][j-1]
return c[][]
```

4. Complexity

Since we only have two loops that loop over m = len(s) and n = len(p) times, our time complexity is O(mn). Space complexity is O(mn) as well because we create a m*n matrix to save the result.

5. Example

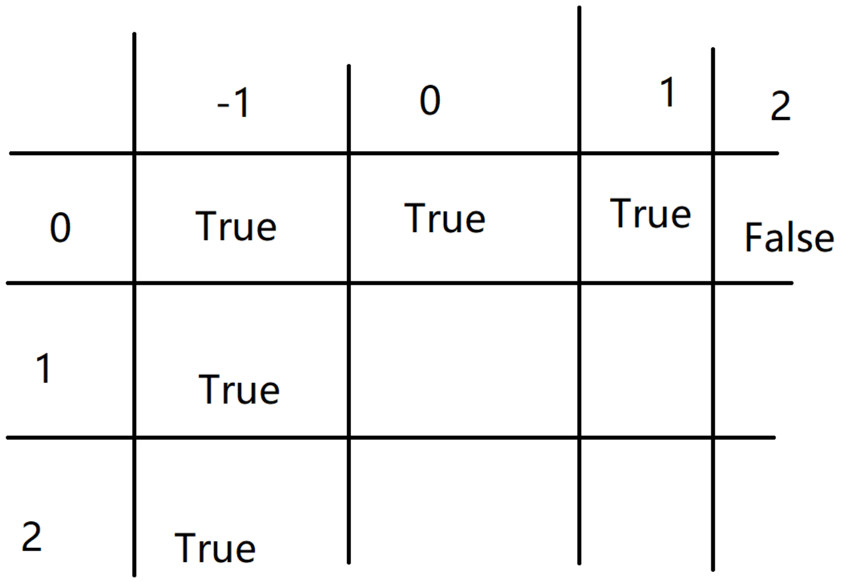
The first step is to initialize the first column to true to represent it will match for empty string.



	-1	0	1	2
0	True			
1	True			
2	True			

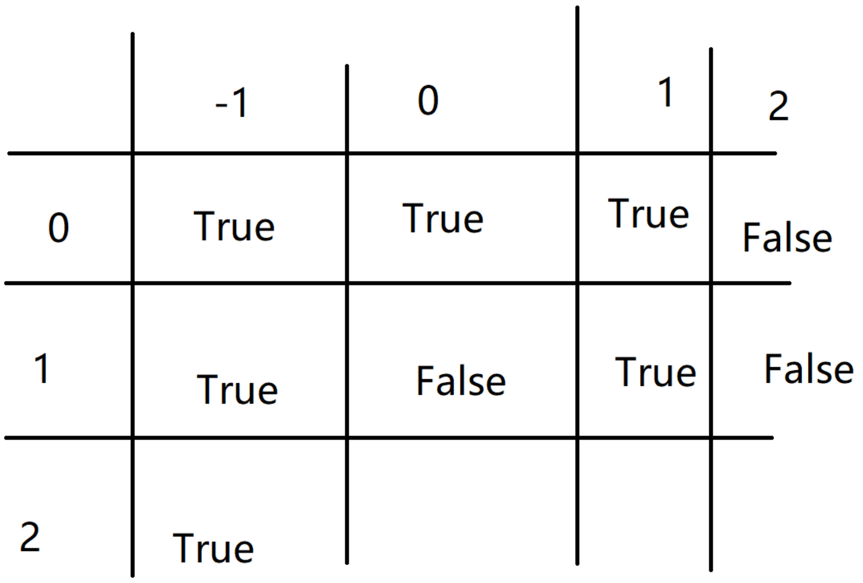
The first row is to check (”a”,”a”), (”a”,”a*”), (”a”,”a*b”), which is true, true, false

s = aab, p=a*b



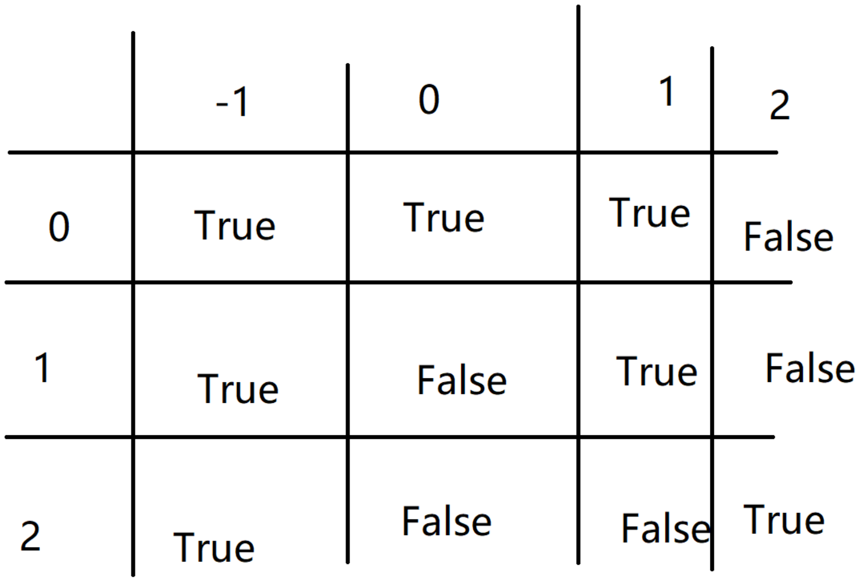
	-1	0	1	2
0	True	True	True	False
1	True			
2	True			

The second row is to check (”aa”,”a”), (”aa”,”a*”), (”aa”,”a*b”), which is false, true, false s = aab, p=a*b



	-1	0	1	2
0	True	True	True	False
1	True	False	True	False
2	True			

The third row is to check (”aab”,”a”), (”aab”,”a*”), (”aab”,”a*b”), which is false, false, true



	-1	0	1	2
0	True	True	True	False
1	True	False	True	False
2	True	False	False	True

□