Homework #8

Due date: 12/29

Part A: Recognizer

Given a binary string, determine if the number of 0's contained in it is a multiple of 3. The program you are going to write is called a *recognizer*, since it can recognize those binary strings that satisfy the required property.

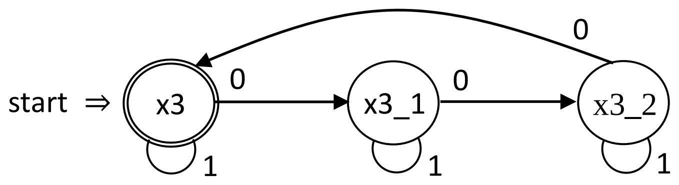
For examples, your recognizer shall accept the following binary strings:

1111111111	# of 0's = 0
0101011110001111	# of 0's = 6

and reject the following binary strings:

1111110111	# of 0's = 1
0101011110101111	# of 0's = 5

Your recognizer shall base on the following DFA:



The three states x3, x3_1, and x3_2 recognize those binary strings in which the number of 0's is a multiple of 3, one more than a multiple of 3, and two more than a multiple of 3, respectively.

Requirements

1. You shall represent each state as a function, say
   ```
   void x3(void);
   void x3_1(void);
   void x3_2(void);
   ```
2. You may assume that a binary string is entered in a line and contains no characters other than `'0'`, `'1'`, and `'\n'`.

Part B: Generator

Given an integer $n \geq 0$, generate all the binary strings of length n in which the number of 0's is a multiple of 3, and count the number of such strings. The program you are going to write is called a *generator*, since it generates all the binary strings that satisfy the required property.

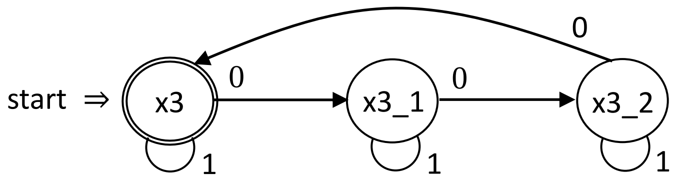
For example, for $n = 4$, your generator shall generate $\binom{4}{3} + \binom{4}{0} = 5$ binary strings:

3 0's: 0001 0010 0100 1000
0 0's: 1111

and for $n = 6$, $\binom{6}{6} + \binom{6}{3} + \binom{6}{0} = 22$ binary strings:

6 0's: 000000
3 0's: 000111 001011 001101 001110 010011 010101 010110 011001 011010 011100
100011 100101 100110 101001 101010 101100 110001 110010 110100 111000
0 0's: 111111

Your generator shall also base on the aforementioned DFA, as replicated below.



In terms of recognition, this DFA says that

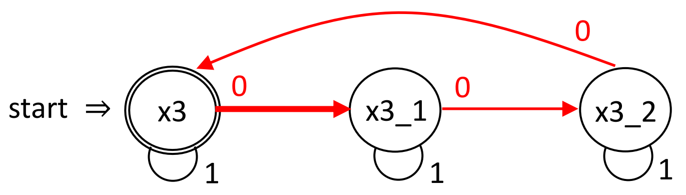
"If you are in state x3 and read in a 0, go to state x3_1 (and from there you shall read in the next digit and transit …); but, if you read in a 1, go to state x3 (and from there you shall read in the next digit and transit …) ."

However, in terms of generation, it says that

"If you are in state x3, you may try to generate a 0 and go to state x3_1 (and from there you may try to generate the next digit and transit …); and, you may also try to generate a 1 and go to state x3 (and from there you may try to generate the next digit and transit …)."

Listening to the advice from the DFA and *trying to* generate a 0 or 1 step by step, we will eventually obtain a binary string of the desired length. At that point, if we are in state x3,

the binary string generated is what we want – thanks to the DFA. Otherwise, we must have done something wrong earlier, and so we have to *go back and retry*.
For example, for $n = 4$,

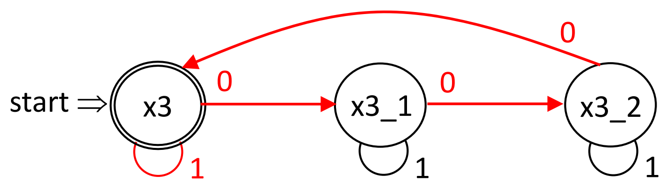


Following the red-colored transitions and going through the heavy arrow twice

$$x3 \xrightarrow{0} x3\_1 \xrightarrow{0} x3\_2 \xrightarrow{0} x3 \xrightarrow{0} x3\_1$$

we have generated 0000, which is undesired, as we aren't in state x3. So, let's undo the *last* choice of generating a 0 in state x3, and generate a 1 in that state instead:

$$x3 \xrightarrow{0} x3\_1 \xrightarrow{0} x3\_2 \xrightarrow{0} x3 \xrightarrow{1} x3$$



Now, we are in state x3 and so 0001 is what we want.

Q: What kind of data structure must be used to store the generated binary string?
A: Obviously, we need a stack, since we must *first* undo the *last* choice.

Q: What shall we do next after generating 0001?
A: In the preceding discussion, we tacitly assume that in each state, we first try 0, and then 1. Thus, our current situation is:

$$\begin{tikzcd} x3 \arrow[r, "0"] \arrow[d, "1"] & x3\_1 \arrow[r, "0"] \arrow[d, "1"] & x3\_2 \arrow[r, "0"] \arrow[d, "1"] & x3 \arrow[r, "0"] \arrow[d, "1"] & x3\_1 \\ {} & {} & {} & x3 & \end{tikzcd}$$

There are three transitions remained to be tried. By the last-in-first-out principle, our next try must be the red-colored transition.

It should now be clear that this x3-binary-string generator behaves similarly to the k-combination generator given in the lecture and the integer-partition generator of HW#7, except that it is more complicated in that three indirectly recursive functions are needed:

`int x3(int n);`

Staring with state x3, this function generates all the binary strings of length n in which the number of 0's is a multiple of 3, and returns the number of such strings as the function value.

`int x3_1(int n);`

This function does similar things to function **`x3`**, except that it starts with state x3_1.

`int x3_2(int n);`

This function does similar things to function **`x3`**, except that it starts with state x3_2.

Q: Considering only the number of binary strings generated, how would you define the functions x3, x3_1, and x3_2?

Comments

1 As usual, these three functions shall cooperate to maintain a global stack declared by, say

```
struct stack {
   int top;
   char stk[10]; // assume that the length of binary string ≤ 10
};
stack s={-1};          // a global stack
```

2 The functions of part A and part B have the same names, namely, **`x3`**, **`x3_1`**, and **`x3_2`**. This is OK in C++, but not in C. So, be sure to name you file with .cpp extension so as to activate the C++ compiler.

User interface

For testing purpose, let's illustrate the user interface with a sample run:

```
Enter your choice: (1) Recognizer (2) Generator: 2   ↵
Enter the length of binary string: 4
0001 0010 0100 1000 1111
5 binary strings in total
```

Observe that if you choose 2, the subsequent spaces (if any) and end-of-line↵ mark are skipped automatically when reading the length of binary string (4, in this example).

```
Enter your choice: (1) Recognizer (2) Generator: 1   ↵
Enter a binary string: 101010
Accepted
```

However, if your choice is 1, the subsequent spaces (if any) and end-of-line ↵ mark have to be skipped manually before reading in the binary string character by character (101010, in this example).

For simplicity, you may assume that all inputs are legal, e.g. the choice you keyed in is either 1 or 2, etc.

Final requirement

You may NOT write any loop. To this end, you need three more recursive functions:

```
void ui(void);              // user interface
void eatline(void);         // skip spaces (if any) and ↵
void show(int i);           // output s.stk[0] to s.stk[i]
```

Sample run

```
Enter your choice: (1) Recognizer (2) Generator: 1
Enter a binary string: 101010111000
Accepted

Enter your choice: (1) Recognizer (2) Generator: 1
Enter a binary string: 00111011100
Rejected

Enter your choice: (1) Recognizer (2) Generator: 1
Enter a binary string: ↵
Accepted

Enter your choice: (1) Recognizer (2) Generator: 2
Enter the length of binary string: 3
000 111
2 binary strings in total
```

```
Enter your choice: (1) Recognizer (2) Generator: 2
Enter the length of binary string: 5
00011 00101 00110 01001 01010 01100 10001 10010 10100 11000 11111
11 binary strings in total

Enter your choice: (1) Recognizer (2) Generator: 2
Enter the length of binary string: 7
0000001 0000010 0000100 0001000 0001111 0010000 0010111 0011011 0011101 0011110
0100000 0100111 0101011 0101101 0101110 0110011 0110101 0110110 0111001 0111010
0111100 1000000 1000111 1001011 1001101 1001110 1010011 1010101 1010110 1011001
1011010 1011100 1100011 1100101 1100110 1101001 1101010 1101100 1110001 1110010
1110100 1111000 1111111
43 binary strings in total
```