2017 SCTF Write-Up

2017년 7월 10일

박현민

1. Mic Check (1 pts.)

1) 문제 요약

The key is SCTF{Welcome_to_SCTF2017_haha}.

2) 정답 : "SCTF{Welcome_to_SCTF2017_haha}"

2. Turing Competition (700 pts.)

1) 문제 요약

튜링 기계를 설계하는 문제이다. 조건이 주어지고, 어떠한 문자열(특히 bit stream)이 들어왔을 때 해당 조건이 만족되는지 여부를 state를 통해 검사하는 TM 객체를 생성해낼 수 있는 json 데이터를 만들어야 한다. 입력 형식은 아래와 같다.


```
{"initial_state":"A","final_states":["Z","Y"],"accepting_
states":["Z"],"transition_function":[
["A","  ","Z","","R"],
["A","0","A","0","R"],
["A","1","B","1","R"],
["B","  ","Z","","R"],
["B","0","C","0","L"],
["B","1","B","1","R"],
["C","  ","D","0","R"],
["C","0","D","0","R"],
["C","1","C","1","L"],
["D","1","E","0","R"],
["E","0","F","1","L"],
["E","1","E","1","R"],
["F","0","A","0","R"],
["F","1","F","1","L"]]}
```


원래는 모든 내용을 한 줄에 써야 하지만, 가독성을 위해 일부 부분에서 임의로 줄을 나눴다. 위 데이터로 생성된 TM 객체는 A state에서 시작하여 B, C, D, E, F, Z의 상태로 전이할 수 있다. 이 자체로 특정한 의미를 갖는 것은 아니며, 단지 형식을 보기 위해 작성하였다.

nc로 접속하면 한 문제가 보이고, 하나를 해결하면 다른 문제가 보이는 식으로 진행된다. 아래에서는 각 문제별로 파트를 나눠 설명한다.

2) '0' * n + '1' * m where (n>=0 and m>=0)

간단한 문제이다. 처음 state는 A로 시작하여, 연속된 0에서는 계속 A state를 유지한다. A에서 문자열의 끝에 도달하면 m=0인 상태로 조건을 만족한 것이므로 TRUE state가 된다. 1이 나오면 B로 전이하며, B 상태 중간에 0이 나오면 조건을 만족하지 않는 것이고 문자열의 끝에 도달하면 조건을 만족한 것이다. 이에 각각 FALSE와 TRUE state로 전이하면 된다. 모든 단계에서 오른쪽으로만 가므로 이전 값을 참고할 필요가 없고, 따라서 다시 기록하는 값은 별로 신경쓰지 않았다. 완성된 데이터는 아래와 같다.


```
{"initial_state":"A","final_states":["TRUE","FALSE"],"accepting_states":["TRUE"],"transition_function":[
["A","0","A","0","R"],
["A","1","B","1","R"],
["A"," ","TRUE","","R"],
["B","0","FALSE","","R"],
["B","1","B","1","R"],
["B"," ","TRUE","","R"]]}
```


3) '1'*x where (7 * x) % 13 == 1

먼저 7x≡1(mod 13)인 x를 찾으면, 간단히 x≡2(mod 13)임을 알 수 있다. 이에 1의 개수를 13으로 나눈 나머지만 확인해주면 된다. 가능한 나머지 13개에 대해 모두 state를 만들고 돌아가는 형태로 제작해 주면 된다. 나머지가 2일 경우에만 문자열의 끝에 도달했을 때 TRUE state가 되고, 나머지는 FALSE state가 된다. 간단한 개념이지만 state의 수가 많아져 길이가 길어졌다.


```
{"initial_state":"0","final_states":["TRUE","FALSE"],"accepting_states":["TRUE"],"transition_function":[
["0","0","FALSE","","R"],
["0","1","1","1","R"],
["0"," ","FALSE","","R"],
["1","0","FALSE","","R"],
["1","1","2","1","R"],
["1"," ","FALSE","","R"],
["2","0","FALSE","","R"],
["2","1","3","1","R"],
["2"," ","TRUE","","R"],
["3","0","FALSE","","R"],
["3","1","4","1","R"],
["3"," ","FALSE","","R"],
["4","0","FALSE","","R"],
["4","1","5","1","R"],
["4"," ","FALSE","","R"],
["5","0","FALSE","","R"],
["5","1","6","1","R"],
["5"," ","FALSE","","R"],
["6","0","FALSE","","R"],
["6","1","7","1","R"],
["6"," ","FALSE","","R"],
["7","0","FALSE","","R"],
["7","1","8","1","R"],
["7"," ","FALSE","","R"],
["8","0","FALSE","","R"],
["8","1","9","1","R"],
["8"," ","FALSE","","R"],
["9","0","FALSE","","R"],
["9","1","10","1","R"],
["9"," ","FALSE","","R"],
["10","0","FALSE","","R"],
["10","1","11","1","R"],
["10"," ","FALSE","","R"],
["11","0","FALSE","","R"],
["11","1","12","1","R"],
["11"," ","FALSE","","R"],
["12","0","FALSE","","R"],
["12","1","0","1","R"],
["12"," ","FALSE","","R"]]}
```

4) '0'*n + '1'*n where (n>=0)

기본적인 아이디어는 먼저 1이 나올 때 까지 오른쪽으로 이동한 후, 1과 0을 한 개씩 짝지어서 없애가며 확인하는 것이다. 즉 왼쪽과 오른쪽이 동시에 문자열의 끝에 도달했다면 조건을 만족했다고 할 수 있는 것이다. 물론 1이 나오다가 다시 0이 나오면 조건을 만족하지 않은 것으로 처리해 주어야 하기도 한다.

```
{"initial_state":"NONE","final_states":["TRUE","FALSE
"],"accepting_states":["TRUE"],"transition_function":[
["NONE","0","A","0","R"],
["NONE","1","FALSE","","R"],
["NONE","  ","TRUE","","R"],
["A","0","A","0","R"],
["A","1","B","*","L"],
["B","0","C","*","R"],
["B","1","FALSE","","R"],
["B","*","B","*","L"],
["C","0","FALSE","","R"],
["C","1","B","*","L"],
["C","  ","D","","L"],
["C","*","C","*","R"],
["D","0","FALSE","","R"],
["D","1","FALSE","","R"],
["D","  ","TRUE","","R"],
["D","*","D","*","L"]]}
```

위 코드에서 NONE state는 n=0인 조건을 확인하기 위한 상태이다. 여기서 0이 1개 이상 있다면 1을 찾기 위한 A state로 넘어간다. A state에서는 가장 먼저 만나는 1을 *로 바꾸고 가장 오른쪽의 0을 *으로 바꾸는 B state로 넘어간다. B는 *을 뛰어넘어 가장 먼저 만나는 0을 *으로 바꾸고 다시 가장 왼쪽의 1을 *로 바꾸는 C state로 넘어간다. 1을 먼저 바꾸기 시작했으므로 0과 1의 개수가 같다면 C state에서 문자열의 오른쪽 끝을 먼저 넘어서게 된다. 그 때 D state로 넘어가며, 이는 문자열의 왼쪽 끝에 도달했는지를 확인한다. D도 문자열의 끝을 확인했다면 조건을 만족한 것이고, 이외의 다른 문자가 있다면 조건을 만족하지 않은 것이다.

5) '0'*p where p is prime

기본적인 아이디어는 소수를 판정하기 위해 p보다 작은 모든 수로 나눠보는 것이다. 나눠보는 것은 앞 3)에서 제시한 동일한 수의 문자를 채워나가는 과정을 응용하여 특정 수의 배수가 특정 상태가 될 수 있도록 처리하여 구현하였다. 아래 표는 2로 나눠보는 개략적인 과정을 나타낸 것이다. #은 곱할 수, ^는 곱할 수가 처리된 후, !는 1번 이어 붙인 후 추가된 문자들, *는 곱할 수 이외의 이어 붙여 나갈 것들, ?는 *가 처리된 후, /는 !를 *로 만들 때 ?로부터 만들어진 문자이다.

```
##00000000
^^!!000000
^^**000000
^^??!!0000
^^//**0000
^^//??!!00
^^////**00
```

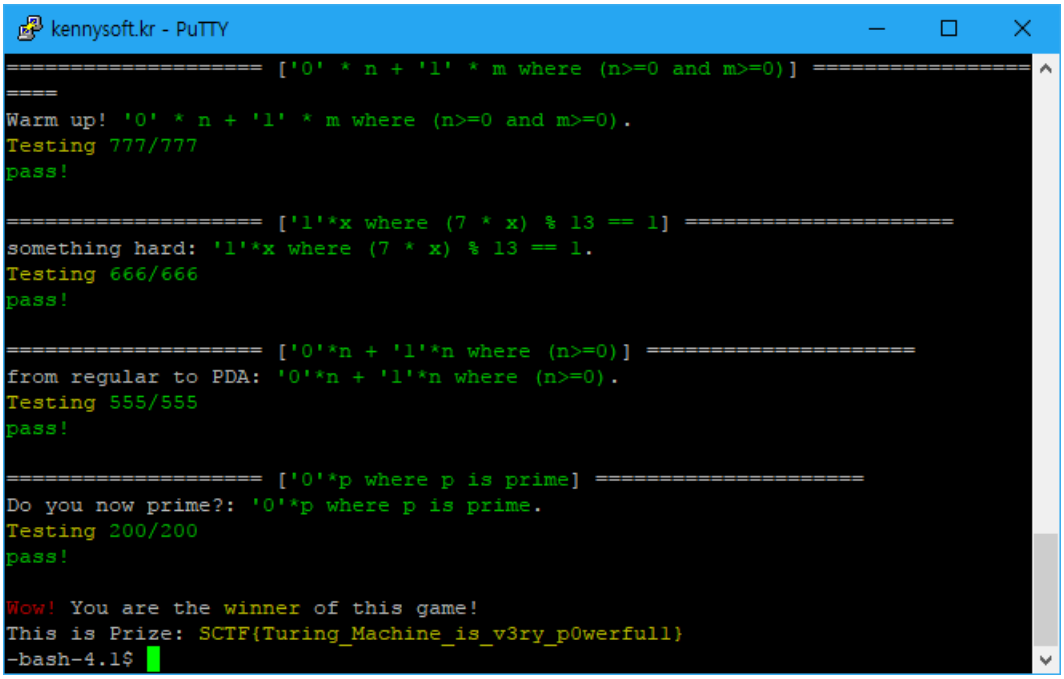
```
{"initial_state":"NONE","final_states":["TRUE","FALSE
"],"accepting_states":["TRUE"],"transition_function":[
["NONE","0","NONE1","#","R"],
["NONE","1","FALSE","","R"],
["NONE","  ","FALSE","","R"],
["NONE1","0","BACK_NEXT_FIRST","0","L"],
["NONE1","1","FALSE","","R"],
["NONE1","  ","FALSE","","R"],
["BACK_NEXT_FIRST","#","BACK_NEXT_FIRST","#","
L"],
["BACK_NEXT_FIRST","^","BACK_NEXT_FIRST","#","
L"],
["BACK_NEXT_FIRST","?","BACK_NEXT_FIRST","0","
L"],
["BACK_NEXT_FIRST","!","BACK_NEXT_FIRST","0","
L"],
["BACK_NEXT_FIRST","/","BACK_NEXT_FIRST","0","
L"],
["BACK_NEXT_FIRST","*","BACK_NEXT_FIRST","0","
L"],
["BACK_NEXT_FIRST","  ","FIRST_PRE","  ","R"],
["FIRST_PRE","#","FIRST_PRE","#","R"],
["FIRST_PRE","0","FIRST","#","R"],
["FIRST_PRE","?","FIRST","#","R"],
["FIRST","#","FIRST","#","R"],
["FIRST","!","FIRST","!","R"],
["FIRST","0","BACK_IN_FIRST","0","L"],
["FIRST","  ","TRUE","","R"],
["BACK_IN_FIRST","#","FIRST2","^","R"],
["BACK_IN_FIRST","^","BACK_IN_FIRST","^","L"],
["BACK_IN_FIRST","!","BACK_IN_FIRST","!","L"],
["BACK_IN_FIRST","  ","NEXT","  ","R"],
["FIRST2","^","FIRST2","^","R"],
["FIRST2","!","FIRST2","!","R"],
["FIRST2","0","BACK_IN_FIRST","!","L"],
["FIRST2","  ","BACK_NEXT_FIRST","  ","L"],
["NEXT","^","NEXT","^","R"],
["NEXT","?","NEXT","/","R"],
["NEXT","!","NEXT","*","R"],
["NEXT","0","BACK_IN_CHECK","0","L"],
["NEXT","  ","FALSE","","R"],
["BACK_IN_CHECK","*","CHECK2","?","R"],
["BACK_IN_CHECK","!","BACK_IN_CHECK","!","L"],
["BACK_IN_CHECK","?","BACK_IN_CHECK","?","L"],
["BACK_IN_CHECK","^","NEXT","^","R"],
["BACK_IN_CHECK","/","NEXT","/","R"],
["CHECK2","!","CHECK2","!","R"],
["CHECK2","?","CHECK2","?","R"],
["CHECK2","0","BACK_IN_CHECK","!","L"],
["CHECK2","  ","BACK_NEXT_FIRST","  ","L"]]}
```


먼저 아무것도 없는 NONE state에서 시작한다. 이후 NONE1 state는 단지 0의 개수가 1개인지 확인하기 위한 용도로, 이들 state에서 문자열의 끝에 도달할 경우 소수가 아니므로 FALSE state가 된다. 물론 1이 있어도 FALSE state로 간다. NONE1 state에서 또 0이 나올 경우에는 0이 2개 이상이라는 뜻이므로 이후 상태로 들어간다. NONE state에서만 0을 #으로 바꾸는 이유는 FIRST_PRE가 처음에도 실행되는데, 이 때 #을 한 개 더하기에 #이 2개부터 시작하게 하기 위함이다.

BACK_NEXT_FIRST state는 초기화하는 단계이다. 맨 왼쪽으로 위치를 옮기면서 #, ^만을 #으로 남기고 나머지 문자들을 모두 0으로 만든다. 이후 FIRST_PRE state에서는 처음으로 나오는 0만을 #으로 바꾸고 난 후 FIRST state로 넘어간다.

FIRST, BACK_IN_FIRST, FIRST2 state는 #~0~을 ^~!~로 바꿔주는 역할이다. 4)의 것과 비슷한 개념으로 동작한다. 이후 이어지는 NEXT state는 ?과 !를 재사용을 위해 각각 /와 *로 바꿔주고 BACK_IN_CHECK, CHECK2 state를 통해 *~0~을 ?~!~로 바꿔준다. FIRST2나 CHECK2 state에서 문자열의 끝이 발견된다면 p가 현재 #의 개수로 나누어떨어지지 않는다는 뜻이므로 다음 개수로 나눠보기 위해 BACK_NEXT_FIRST state로 다시 돌아간다. 만약 NEXT state에서 0이 발견되지 않는다면 이는 p가 현재 #의 개수로 나누어떨어진다는 뜻이므로 바로 FALSE state로 전이하게 된다. 이렇게 반복되다가 FIRST state에서 문자열의 끝이 발견된다면 p가 p보다 작은 모든 값으로 나누어떨어지지 않는다는 것이므로 p가 소수임을 알 수 있고, TRUE state로 전이하게 된다.

6) 실행 화면



7) 정답 : "SCTF{Turing_Machine_is_v3ry_p0werful1}"

3. 소감

이전 SCPC에 참가하며 삼성이 많은 돈을 들여 인재 발굴에 힘쓰고 있다는 것을 느꼈는데, SCTF도 그 연장선에 있음을 느꼈다. 한 가지 놀라웠던 점은 SCPC에서 좋은 활약을 했던 사람들 중 일부가 SCTF에서도 꽤나 좋은 성적을 냈다는 것이다. 이 점에는 Turing Competition의 배점이 높은 것이 어느 정도 비중을 차지한다고 볼 수도 있겠다고 생각한다. 나는 개인적으로 CTF 문제를 많이 풀어보지 않았고 python에 대해서도 익숙지 않아서 다른 문제들은 해결하지 못했기에 달리 할 말은 없다.